Implementing a Reusable Design Pattern Library in C#

By

Alastair van Leeuwen

M.Sc. Computer Science

Department of Computer Science

University of Pretoria

2013

ABSTRACT

Implementing a Reusable Design Pattern Library in C#

By

Alastair van Leeuwen, 2012

Design patterns in software systems are described as a universal reusable solution to a commonly recurring problem in software design. Design patterns were, however, not intended to be reusable in terms of code. A symptom of their non-reusability is the problems experienced with the way the implementation of design patterns negatively affects their traceability, maintainability and contribution to productivity. This thesis shows how design patterns can be elevated to a higher level of reusability. This work presents design patterns as reusable components that developers can use to implement solutions that utilise patterns, without having to implement a major part of a pattern's structure and behaviour anew each time. A component is a reusable software section, with possible library classes, that is usually in source form. Previous research has shown that a high proportion of patterns (65%) can be "*componentized*" in Eiffel, which leads to the idea that a language supporting the same set of features would also have the same success in pattern componentization. This thesis has looked at the componentization of twelve design patterns in C#. The C# language has more advanced language features than Eiffel, including functional and dynamic language features and, as such, should lend itself better to pattern componentization than Eiffel does. The language features that are reviewed in this thesis are inheritance, design by contract™, attributes, method references (or delegates), anonymous functions, lambda expressions, mixins (or extension methods), duck typing, dynamic types and meta-programming. Each pattern's reusable components are discussed in detail, including the success of the reusable component transformation. All the design patterns reviewed in this thesis could be transformed into fully or partially reusable components. Implementing design patterns using reusable library components is thus a step in the right direction in making design pattern implementations more traceable, reusable, maintainable and more productive. Other object-oriented languages implementing the same or similar language features as those reviewed in this thesis should have the same level of success in transforming design patterns into reusable components.

Keywords: design patterns, C# 4.0, language features, reusable, duck typing

Supervisor: Prof. J. Bishop

Department: Department of Computer Science, University of Pretoria

Degree: Magister Scientiae

ACKNOWLEDGMENTS

I am wholeheartedly grateful to my supervisor, Professor Judith Bishop, whose guidance, patience, encouragement and assistance enabled me to cultivate an improved understanding of the subject.

I also offer my special thanks to Russell Politzky, who painstakingly reviewed the technical details of this report.

Lastly, I offer my thanks and appreciation to all of those who helped me in any respect during the completion of this thesis, especially to all my colleagues at Dariel Solutions who had to endure endless discussions, ideas and thoughts on the subject matter, most especially Gareth Baars, Sugendran Pillay and Kabelo Kgabale.

DEDICATION

For my loving wife, who endured endless lonely nights and who offered me unconditional love, support, patience and inspiration throughout the course of this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

1 INTRODUCTION

1.1 Problems with Design Patterns

A design pattern (Gamma, Helm, Johnson, & Vlissides, 1994) is a formal mechanism for documenting solutions to recurring software design problems. Christopher Alexander first introduced the concept of design patterns in civil architecture (Alexander & Ishikawa, 1977). This was later adapted to software design. The academic and commercial interest in design patterns has shown a dramatic growth in the last decade. Design patterns have been catalogued by a number of research projects including *Patterns languages of program design* (Coplien & Schmidt, 1995), *Design patterns for object-oriented software development* (Pree, 1995) and *Design patterns, elements of reusable object-oriented software* (Gamma, Helm, Johnson, & Vlissides, 1994).

"A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design" (Gamma, Helm, Johnson, & Vlissides, 1994, p. 3).

Design patterns can be classified according to the underlying problem that they solve. These classifications include creational, structural, behavioural (Gamma, Helm, Johnson, & Vlissides, 1994), concurrency (Schmidt, 1995) and architectural patterns (Avgeriou & Zdun, 2005).

Design patterns offer a number of benefits, as shown below (Chambers, Harrison, & Vlissides, 2000) (Schmidt, 1995). Design patterns

- promote design reuse.
- have names which form a common vocabulary and improve communication within and across software development teams.
- improve documentation.
- help one restructure a software system whether or not patterns were used up-front.
- explicitly capture knowledge that experienced developers already understand implicitly.

- may lead developers to think they know more about the solution to a problem than they actually do.

- help with the training of new developers.

- help to transcend "programming language-centric" viewpoints.

Design patterns are mostly seen as a solution to recurring problems encountered in software design. Not much emphasis has yet been placed on the physical implementation of design patterns in traditional object-oriented languages. The physical implementations of design patterns do suffer from problems.

The main difficulties with design patterns are the lack of traceability in the implementation, language expressiveness and the implementation overhead, as shown below (Bosch, 1998b) (Bosch, 1998a):

- **Traceability**

  The traceability of design patterns is lost because the programming language does not directly support the underlying pattern. The physical implementation of the design pattern in the programming language is scattered across a number of classes and is thus hard to trace.

- **Reusability**

  Design patterns are implemented and recycled in the design of a software system. A developer is constrained to implement a design pattern over and over in a physical programming language. A design pattern does not give a developer the same benefits as a reusable component.

- **Implementation Overhead or Writability**

  Design patterns force a developer to implement several methods with trivial behaviour. This leads to a huge programming burden on the developer, made even worse by the fact that the design pattern implementations cannot be reused. These methods are tedious to develop and maintain without the help of powerful programming or *integrated development environment* (IDE) tools (Bishop, 2008).

- **Maintainability**

  It has also been argued that using multiple patterns in the same implementation can lead to a large cluster of mutually dependent classes (Soukup, 1995). Using a traditional object-oriented programming language can cause maintainability problems when working with mutually dependent classes (Soukup, 1995).

Pinto, Amor, Fuentes and Troya state "The DPs fail providing a solution because it is necessary to apply and implement the same design over and over, for each component" (Pinto, Amor, Fuentes, & Troya, 2001, p. 5). This is the same as the reusability problem defined by Bosch, as discussed above (Bosch, 1998b) (Bosch, 1998a).

Another criticism of design patterns is the fact that some patterns can be consolidated (Agerbo & Cornils, 1998). The physical implementation of a design pattern can be confused with another pattern because they are, in fact, closely related. An increase in the number of new design patterns will actually threaten their original benefits. Agerbo and Cornils argue that the rapid evolution of design patterns has hampered the benefits gained from using patterns. They note that an increase in design patterns impairs communication within and across software development teams. Vlissides also had the same belief, quoting Kahlil Gibran in his paper "We shall never understand one another until we reduce the language to seven words." (Chambers, Harrison, & Vlissides, 2000, p. 283).

I have also noticed from experience on a number of projects that I have been involved with that design patterns are not implemented properly in object-oriented programming languages by developers. The incorrect implementations usually do not follow the structure of the pattern as defined in the *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) catalogue. The incorrect implementation can also have the wrong name, in which case it actually implements another design pattern.

1.2 The Goal of this Thesis

The goal of this thesis is to report on the success of the design and development of a reusable design pattern library, called *Adaptive Pattern Library* (APL), in C#. A reusable design pattern library solves most of the problems mentioned above. Using a design pattern from a library makes it clear to a developer which pattern is being implemented, thus solving the **Traceability** problem. It also solves the **Implementation Overhead** problem because a developer is not tasked with implementing the core of the pattern. A developer only needs to use the implementations in the pattern library. It also directly solves the **Reusability** problem, because a reusable component for a specific pattern exists and can thus be reused.

This thesis explores the implementation of reusable design pattern components in the C# programming language. The focus of my research was to transform design patterns into reusable artefacts so that developers would not have to implement the same design pattern core logic and structure over and over. The concept of reusability uses Meyer's definition as defined in *Object-Oriented Software Construction* which states: "Reusability is the ability of software elements to serve for the construction of many different applications" (Meyer, 2000, p. 7). In the context of design patterns, a specific language feature or features can be used to implement a language library or a component

which might solve the pattern implementation reusability problem. In this thesis I have therefore explored the creation of a design pattern class library with reusable components in C#. It concentrates on the design patterns defined in the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, & Vlissides, 1994), which is referred to as *Design Patterns* in the rest of this thesis. Four design patterns each were chosen from the structural, behavioural and creational design pattern categories. Concurrency patterns (Schmidt, 1995) and architectural patterns (Avgeriou & Zdun, 2005) would benefit from the same techniques used in this thesis, but they were not explored.

Meyer's "*component*" definition is used extensively in this thesis. His definition of "*component*" must satisfy the following criteria (Meyer, 2000): A component can be used by other program elements which are known as "*clients*". The supplier of a component does not need to know who its clients are. Clients can use a component on the sole basis of its official information. A C# class and interface thus adheres to Meyer's definition of a "*component*". This thesis does not use the component definition specified by Szyperski (Szyperski, 2002).

Meyer and Arnout define a componentizable design pattern as "A design pattern is componentizable if it is possible to produce a reusable component, which provides all the functions of the pattern" (Meyer & Arnout, 2006, p. 24). Meyer also stresses (Meyer, 2000, p. 72) that "A successful pattern cannot just be a book description: it must be a software component or a set of components". In this thesis, I argue and show that a design pattern is reusable if it is implemented as a component that adheres to the pattern's intent and functionality and where the component is also usable and practical.

Note the difference between design reuse and software implementation reuse. The *Design Patterns* book does mention that design patterns are there to create "*a reusable object-oriented design*" (Gamma, Helm, Johnson, & Vlissides, 1994, p. 3). However, this reuse is in the context of design and not implementation. Bosch and Soukup discuss the problems regarding the actual physical implementation of design patterns, including their current lack of reusability (Bosch, 1998b) (Soukup, 1995).

Arnout remarks that from a software engineering perspective, design patterns could be seen to represent a step backwards as regards implementation reuse, because patterns must be implemented and re-implemented manually (Arnout, 2004).

Jézéquel, Train and Mingins note that "Patterns are not, by definition formalized descriptions. They can't appear as a deliverable" (Jézéquel, Train, & Mingins, 1999, p. 22). Arnout challenges this perception, asking why one has to step back to pre-reuse times when implementing design patterns (Arnout, 2004). My research also challenges this statement with examples of reusable design pattern components in C#.

1.3 Previous Solutions

Others have already challenged the statement by Jezequel that design patterns are not reusable on an implementation level. In the book *Modern C++ Design: Generic Programming and Design Patterns Applied* Alexandrescu explores the reusability of design patterns using the generic features of C++ (Alexandrescu, 2001). The result is a reusable library called *Loki* with reusable implementation solutions for certain design patterns. *Loki* is a popular class library in C++ showing that it is possible to reuse certain design patterns in a language with similar features to those of C++.

Schmidt has successfully implemented concurrent and networking reusable design pattern implementations in the *ACE* (Adaptive Communication Environment) C++ library (Schmidt, Stal, Rohnert, & Buschmann, 2000). He has also relied heavily on C++ generics (templates) with which to implement the reusable design patterns. For example, as part of his extensive library he has created a generic class that implements the singleton design pattern. The reusable singleton C++ class uses generics in order to turn ordinary C++ classes into singletons optimised with the double-checked locking optimisation pattern (Schmidt & Harrison, 1996). The following code snippet shows an example of the usage of the *ACE* singleton class in C++. The **`SingletonImpl`** class is transformed into a singleton by the **`SingletonTest`** type definition, using the **`ACE_Singleton`** class:

```
C++
-----------------------------------------------------------------------------------------------------------
typedef ACE_Singleton<SingletonImpl, ACE_Null_Mutex> SingletonTest;

int main(int argc, char* argv[]) {
   SingletonTest singleton = SingletonTest::instance(); // Aquire a reference to the singleton instance
   // …
}
```

Bosch takes the standpoint that it is the task of the programming language to represent the implementation of a design pattern as closely as possible. He does concede that it would be impossible for a language to represent all design patterns (Bosch, 1998c). He argues further that most design patterns have well defined semantics that could be used as the basis for defining language constructs that explicitly support the representation of a certain design pattern in the programming language. He complains that some engineers and researchers believe that design patterns should only be used in software design. Bosch disagrees with these engineers and researchers and wants to see more explicit language support or language features for design patterns. Bosch strongly disagrees that this would increase the complexity of the language, because the language will represent the paradigm concepts used by the developer (Bosch, 1998b). He further argues that it is, in fact, the lack of language support for design patterns that increases the complexity. This is because a developer is forced to implement the patterns in terms of lower level language constructs, thereby reducing traceability and understandability. He also argues that a developer is free to use the available language constructs, but is not forced to use them. He states that as a developer gains experience, his usage of language constructs

increases. He finally states that it is "beneficial for a programming language to provide constructs for representing design patterns" (Bosch, 1998b, p. 9).

Some modern languages, such as Ruby, already have some design patterns implemented in their standard class libraries (Matsumoto, 2001). Here is an example of how to implement a singleton (Gamma, Helm, Johnson, & Vlissides, 1994) in Ruby (Williams, 2006):

```
Ruby
-----------------------------------------------------------------------------------------------------------
require 'singleton'

class Example
   include Singleton
end
```

The above code snippet shows that Ruby provides a module for making classes singletons, which is defined in the standard library inside '*singleton.rb*'. The following example shows how a singleton in Ruby could have been implemented without the singleton standard library support (Williams, 2006):

```
Ruby
-----------------------------------------------------------------------------------------------------------
class Example
  def initialize
    # do something?
  end

  def self.instance
    return @@instance if defined? @@instance
    @@instance = new
  end
  private_class_method  :new
end
```

MultiJava is an extension to the Java programming language that adds symmetric multiple-dispatch (Clifton, Millstein, Leavens, & Chambers, 2006). The multiple-dispatch language feature eliminates the need for the accept element when implementing the visitor pattern (Gamma, Helm, Johnson, & Vlissides, 1994). Multimethods or multiple-dispatch is a special feature in certain object-oriented programming languages where a function or method can be specialised on the type of more than one of its arguments. Multiple-dispatch is a type of language feature that is part of *The Common Lisp Object System* (CLOS) (DeMichiel & Gabriel, 1987).

The following example from Arnout shows a possible implementation in *MultiJava* (Clifton, Millstein, Leavens, & Chambers, 2006) of the visitor pattern (Gamma, Helm, Johnson, & Vlissides, 1994) without the use of the accept element (Arnout, 2004):

```
Java
-----------------------------------------------------------------------------------------------------------
public class MaintenanceVisitor {
    public void visit (Borrowable borrowable) {
        throws new Error("An abstract class cannot be instantiated.");
```

```
    }

    public void visit (Borrowable@Book borrowable) { // Special treatment for books }
    public void visit (Borrowable@VideoRecorder borrowable) { // Special treatment for video recorder }
}
```

Arnout states in her Ph.D. thesis *From Patterns to Components* that "Design patterns are good but components are better" (Arnout, 2004, p. 5). She argues that reusing software improves the overall quality of software, including its correctness, maintainability and performance.

She correctly notes that design patterns are naturally reusable in software design, but not in software implementation. Her thesis explores the componentization of design patterns. She focused mainly on Eiffel (ECMA, 2006), but did also briefly explore the componentization of design patterns in Java and C# (Arnout, 2004). Arnout did note, however, that not all design patterns could be componentized. This thesis builds on Arnout's research. C# has more advanced language features than Eiffel (Meyer, 1991) and this thesis shows that this improves the possibility for componentization of design patterns.

In the publication *A Debate on Language and Tool Support for Design Patterns* Chambers, Harrison and Vlissides (Chambers, Harrison, & Vlissides, 2000) question whether languages should be extended with features corresponding to particular patterns. They further note that design patterns "have proved so useful that some have called for their promotion to programming language features" (Chambers, Harrison, & Vlissides, 2000, p. 277). Chambers argues that some design patterns do have native support in mainstream object-oriented languages. Vlissides argues that advances in computer language features have come from abstracting what programmers do most in their existing code. He notes that there are design patterns that naturally lend themselves towards language constructs, using the singleton as an example:

```
public singleton class WindowManager { … }
```

The programming language implementing the singleton design pattern as a language feature will ensure that only one instance of the object is created. The language will also handle advanced singleton issues such as multi-threading and locking problems. For example, the language can use the double-checked locking pattern internally with the singleton pattern in order to solve advanced locking problems (Schmidt & Harrison, 1996). Vlissides does warn, however, that not all design patterns should be implemented as language features. He argues that some design patterns included as a feature in a programming language could make that language too complicated. He gives multiple-dispatch (Stroustrup, 1994) as an example. Multiple-dispatch is a language feature that can be used to implement the visitor pattern. He argues that current mainstream languages such as C# and Java do not implement multiple-dispatch as a language feature because of the extra complexity. This is in contrast to Bosch, who believes that design pattern language features do not necessarily make a language more complex (Bosch, 1998a). Today, some of the following programming languages

support multiple-dispatch, either directly or indirectly, as a built in language feature: Common Lisp (via the Common Lisp Object System) (DeMichiel & Gabriel, 1987), Haskell via Multi-parameter type classes, Dylan, Cecil, R, Groovy, Perl 6, Seed7, Clojure, C# 4.0 (Burchall, 2009) and Fortress.

The following code snippet by Burchall shows how the dynamic keyword in C# 4.0 can be used in order to implement multiple-dispatch functionality (Burchall, 2009):

```
C#
-----------------------------------------------------------------------------------------------------------
class Program {
    class Thing { }
    class Asteroid : Thing { }
    class Spaceship : Thing { }

    static void CollideWithImpl(Asteroid x, Asteroid y) {
        Console.WriteLine("Asteroid hits an Asteroid");
    }

    static void CollideWithImpl(Asteroid x, Spaceship y) {
        Console.WriteLine("Asteroid hits a Spaceship");
    }

    static void CollideWithImpl(Spaceship x, Asteroid y) {
        Console.WriteLine("Spaceship hits an Asteroid");
    }

    static void CollideWithImpl(Spaceship x, Spaceship y) {
        Console.WriteLine("Spaceship hits a Spaceship");
    }

    static void CollideWith(Thing x, Thing y) {
        dynamic a = x;
        dynamic b = y;
        CollideWithImpl(a, b);
    }

    static void Main(string[] args) {
        var asteroid = new Asteroid();
        var spaceship = new Spaceship();
        CollideWith(asteroid, spaceship);
        CollideWith(spaceship, spaceship);
    }
}
```

In C# a virtual method is polymorphic (Cardelli & Wegner, 1985) only on a singular level. Multimethods or multiple-dispatch takes polymorphism a step further, where a method is polymorphic on multiple levels, which can be advantageous in some situations. In the above code the **`dynamic`** keyword permits a method to be selected that is dependent on the type of arguments at runtime, not just the connected object. In the above example the **`CollideWith`** method takes in two arguments of type **`Thing`**. The **`CollideWith`** method passes the request to the correct **`CollideWithImpl`** during runtime, depending on the type of argument. A **`CollideWith(asteroid, spaceship)`** request is thus passed on to the **`CollideWithImpl(Asteroid x, Spaceship y)`** implementation that will execute the correct algorithm. The example thus shows an implementation of genuine multiple-dispatch in C#. In the above trivial example the **`CollideWith(Thing x, Thing y)`** method can be removed and the **`CollideWithImpl`**

method can still be called correctly. Functionally, this solves the same problem; however the method resolution occurs during compile time, using method overloading. The usage of the **`dynamic`** keyword in the **`CollideWith(Thing x, Thing y)`** method allows for runtime method resolution and thus makes multiple-dispatch possible.

The Seed7 programming language implements multimethods directly as a language feature. It is a higher level language than Ada, C++ or Java (Mertes, 2011). In Seed7 methods are not associated with just one type. True to the functionality of multiple-dispatch, the decision which method is executed at runtime is dependent on the types of the arguments. In the example below, from the Seed7 manual, the type **`Number`** is used to amalgamate numerical types. The type **`Number`** is also defined as an interface that defines the contract behaviour for the '**`+`**' operation (Mertes, 2011):

```
Seed7
-----------------------------------------------------------------------------------------------------------
const type: Number is sub object interface;
const func Number: (in Number param) + (in Number param) is DYNAMIC;
```

The interface type part **`Number`** can denote an **`Integer`** or a **`Float`**:

```
Seed7
-----------------------------------------------------------------------------------------------------------
const type: Integer is new struct
    var integer: val is 0;
  end struct;

type_implements_interface(Integer, Number);

const type: Float is new struct
    var float: val is 0.0;
  end struct;

type_implements_interface(Float, Number);
```

The following shows the implementations of the converting **`'+'`** operators:

```
Seed7
-----------------------------------------------------------------------------------------------------------
const func Float: (in Integer: a) + (in Float: b) is func
  result
    var Float: result is Float.value;
  begin
    result.val := flt(a.val) + b.val;
  end func;

const func Float: (in Float: a) + (in Integer: b) is func
  result
    var Float: result is Float.value;
  begin
    result.val := a.val + flt(b.val);
  end func;
```

The following shows the implementations of the normal **'+'** operators that do not do any conversions:

```
Seed7
------------------------------------------------------------------------------------------------------------
const func Integer: (in Integer: a) + (in Integer: b) is func
  result
    var Integer: result is Integer.value;
  begin
    result.val := a.val + b.val;
  end func;

const func Float: (in Float: a) + (in Float: b) is func
  result
    var Float: result is Float.value;
  begin
    result.val := a.val + b.val;
  end func;
```

More operators can be added to the **Number** type such as **'-'** or **'*'**. More implementations can also be added such as **Complex**, **Decimal** or **Double**. The **Number** type defined above can thus be used as a common type for mathematical calculations.

Vlissides further argues that the problem is to decide which design patterns should be included as a language feature and which should be excluded. Vlissides calls this the "kitchen sink problem" and says: "While several of the more fundamental design patterns may be transliterated easily into programming language constructs, many others cannot – and at least should not" (Chambers, Harrison, & Vlissides, 2000, p. 284). Arnout shares the same views as Vlissides, as discussed in her Ph.D. thesis (Arnout, 2004). She argues that some design patterns just cannot be transformed into language features. She suggests the idea of design pattern componentization through software libraries, arguing that libraries do not add complexity to the language as a language feature would. Arnout does concede that certain design patterns, which could not be componentized, can be made reusable by extending the Eiffel language (Arnout, 2004). In depth research has also been done regarding the implementation of design patterns using aspect oriented programming (AOP). Aspect-oriented programming is a programming concept the goal of which is to boost modularity by implementing the separation of cross-cutting concerns (Kiczales, et al., 1997). Hannemann notes that 52% of the design patterns defined in *Design Patterns* are reusable, when using aspect oriented programming (Hannemann & Kiczales, 2002). Arnout argues that using aspects does have its weaknesses (Arnout, 2004). She notes that it may become difficult to master a whole system where the design patterns are implemented using aspects.

The technique of automatic code generation from models depicting design patterns can be seen as another solution to pattern reuse. Budinsky describe a tool for generating source code from models of design patterns (Budinski, Finnie, Yu, & Vlissides, 1996).

Hedin has proposed that design patterns be formalised in the implementation language, using attribute grammars (Hedin, 1997). This, however, forces the developer to learn the formalised grammar.

Agerbo and Cornils have argued that design patterns can be partitioned into the following categories (Agerbo & Cornils, 1998):

- Fundamental Design Patterns (FDPs)
- Language Dependant Design Patterns (LDDPs)
- Related Design Patterns (RDPs)
- Library Design Patterns (LDPs)

They define Fundamental Design Patterns (FDPs) as the core patterns, which should capture good object-oriented design on a high enough level so that they can be used in various kinds of applications. They state that design patterns covered by language constructs are not Fundamental Design Patterns. It is their belief that a Fundamental Design Pattern must be independent of any implementation language. They have analysed the patterns and found that only 11 of the 23 design patterns in *Design Patterns* can be classified as Fundamental Design Patterns (Agerbo & Cornils, 1997). The first column in Table 1, by Bishop and Horspool (Bishop & Horspool, 2008), shows these Fundamental Design Patterns:

Table 1: Fundamental patterns identified as FDPs by Agerbo and Cornils and as cadets by Gil and Lorenz.

Design Pattern	**FDPs**	**Cadets**
Bridge	✓	✓
Builder	✓	✓
Composite	✓	✓
Decorator	✓	✓
Mediator	✓	✓
Proxy	✓	✓
State	✓	✓
Adapter		✓
Chain of Responsibility		✓
Interpreter		✓
Observer		✓
Strategy		✓

Visitor		✓
Abstract Factory	✓	
Flyweight	✓	
Iterator	✓	
Memento	✓	

It is noted, correctly, (Bishop & Horspool, 2008) that the list above is dated, because both the iterator and memento design patterns can be covered by new language features that have been added to both Java and C# (iterators and serializable respectively). There is also a strong case that the events and delegates language features in C# are implementations of the observer design pattern (Purdy & Richter, 2002) (Gasiūnas, Satabin, Mezini, Núñez, & Noyé, 2010).

Agerbo and Cornils also define Language Dependant Design Patterns (LDDPs). These are design patterns that are covered by a language construct in some programming languages, but not all. For example, multiple-dispatch can be seen as a language feature that implements the visitor design pattern (Gamma, Helm, Johnson, & Vlissides, 1994). The first column in Table 2, by Bishop and Horspool (Bishop & Horspool, 2008), shows the patterns that are supported by language features and defined as LDDPs by Agerbo and Cornils.

Table 2: Patterns supported by language features: the LDDPs of Agerbo and Cornils and the clichés/idioms of Gil and Lorenz.

Design Pattern	**LDDPs**	**Clichés and Idioms**
Chain of Responsibility	Delegates	
Command	Procedure classes	Classes
Facade	Nested classes	Encapsulation
Factory Method	Virtual classes	
Memento		Persistence
Prototype	Pattern variables	Deep copy
Singleton	Singular objects	Module
Template Method	Complete block structure	Overriding
Visitor	Multiple dispatch	Multi-methods

Agerbo and Cornils also define Related Design Patterns (RDP) as an application of another design pattern. As an example, they show that the observer design pattern can be implemented using the mediator. Another example is the interpreter pattern that uses the visitor (Agerbo & Cornils, 1998).

Agerbo and Cornils note that the more new design patterns are applied to a certain software implementation, the more difficult it is to recognise the structure of the participating design patterns (Agerbo & Cornils, 1997). This is known as the tracing problem (Bosch, 1998b). They further argue that the solution to this problem could be the use of Library Design Patterns or LDPs. An LDP is a design pattern which is implemented in a reusable library. When using LDPs in the application code, it is possible to trace the design pattern from which the implementation ideas came. They believe that a way of promoting the habit of using design patterns is to have the design patterns available as LDPs in a library where they are easily accessible. They state that another benefit of having a design pattern as an LDP is that one doesn't have to duplicate the implementation anew each time a design pattern is applied in a new context. Agerbo and Cornils warn that when using a pattern as an LDP the design pattern implementation is fixed (Agerbo & Cornils, 1997). It is thus not possible to adapt an LDP for other desired pattern scenarios.

Agerbo and Cornils have formulated the following three guidelines with regard to design patterns (Agerbo & Cornils, 1997, p. 3):

- Design patterns covered by language constructs are not Fundamental Design Patterns (FDPs).
- Applications and variations of design patterns are not Fundamental Design Patterns (FDPs).
- A design pattern may not be an inherent object oriented way of thinking.

In the article *Design Patterns vs. Language Design* Gil and Lorenz have done similar research on design patterns to that of Agerbo and Cornils, looking at how far they are from becoming actual language features by classifying patterns in groups (Gil & Lorenz, 1998). They classified patterns as clichés, idioms or cadets. These classifications correspond to the guidelines from the design pattern analyses of Agerbo and Cornils. They note that cadets are current contenders for language support, whereas clichés and idioms imitate features found in languages. It is their standpoint that design patterns should eventually evolve into language features. This set of patterns is shown in the second column of Table 2. The second column in Table 2 shows the patterns that they have identified as still requiring language support.

1.4 Design Pattern Reusability

The purpose of this thesis is to determine whether design patterns, as explained in *Design Patterns,* can be made reusable in C#, given the features and mechanisms in the C# programming language. The following language features and mechanisms are studied, all of which are available in C#:

- Inheritance (Mitchell, Mitchell, & Krzysztof, 2003)
- Interfaces (Pattison & Box, 2000)
- Generics (Jagger, Perry, & Sestoft, 2007)
- Design by Contract™ (Mitchell & McKim, 2001)
- Attributes (Nagel, Evjen, Glynn, & Watson, 2010)
- Method References (Microsoft, 2010e)
- Anonymous Functions (Ierusalimschy, 2003)
- Lambda Expressions (Michaelis, 2010)
- Mixins (Extension Methods) (Esterbrook, 2001) (Jesse & Xie, 2008)
- Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005)
- Duck Typing (Koenig & Moo, 2005)
- Meta-Programming (Perrotta, 2010)
- Dynamic Typing (Tratt, 2009)

The componentization process in each chapter shows how the above mentioned C# language features helped with the implementation of the pattern components.

The original design pattern catalogue discussed in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) shows implementations that do not take modern language features into consideration. In the paper *On the Efficiency of Design Patterns Implementation in C# 3.0* Bishop and Horspool argue that new language features such as delegates, generics, nested classes, reflection and built-in iteration must be taken into consideration when implementing design patterns in C# 3.0. It is shown that the advances

in language features make design pattern implementations more efficient and also easier to produce and reproduce (Bishop & Horspool, 2008). It is also argued that the usage of modern language features improves the **Traceability**, **Reusability**, **Writability** and **Maintainability** problems of pattern implementation (Bishop, 2008). This thesis shows that modern language features are also important when implementing reusable design pattern components.

In each chapter in this thesis that examines a design pattern, the components discussed are declared reusable if the pattern conforms to the following criteria:

- **Completeness**: Does the reusable component cover all cases of the core pattern implementation described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994)?
- **Usefulness**: Is the reusable component beneficial compared to an implementation from scratch of the design pattern?
- **Faithfulness**: Is the reusable component faithful to the original pattern description?
- **Type-safety**: Is the reusable component type-safe?
- **Extended applicability**: Does the reusable component cover more cases than the original design pattern?
- **Performance**: Is the performance of the reusable component acceptable?

The above mentioned criteria were also used by Arnout in her exploration of reusable design patterns in Eiffel (Arnout, 2004). Each chapter in this thesis describing reusable design pattern componentization ends with a discussion about the quality of the reusable component compared to the above criteria.

1.5 Reusable Design Pattern Exploration

Each chapter in this thesis that explores the componentization of a specific design pattern does so by dividing the exploration into the following sections:

1. Introduction
In this section a short discussion and the formal definition of the design pattern is given. It also shows the pattern participants and the formal UML structure of the design pattern.

2. Reusable Library Implementation

This section discusses the reusable library implementations or components that were developed in C# for this thesis, for the specific design pattern that is the subject of each chapter. It discusses their technical implementation in detail, how they satisfy the intent of the pattern and also possible caveats. It is possible that a design pattern does not have a reusable library implementation in C#. It is also possible that a design pattern is only partially reusable when implemented in C#. Partial reusability means that some parts of the pattern must still be coded by hand. Lastly, it is also possible that not all of a pattern's functionality or intent could be realised with a reusable component in C#.

3. Theoretical Examples

In this section formal implementation examples of the specific design pattern are given in C#, using the reusable library components described in the previous section.

4. Outcome

This section discusses the success of the reusable library using the criteria discussed in the previous section.

The source code shown in this thesis sometimes omits code that is seen as redundant. The **"… S N I P …"** or **"…"** snippets are used to show that there is more code than that which is shown:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionChainOfResponsibility : ICommand { // The handler is also a command
    private readonly Action _successor; // Successor defined as an Action delegate
    private readonly Action _handler;   // Handler defined as an Action delegate
    // … S N I P …
}
```

The **ActionChainOfResponsibility** class thus has more elements present, but they are not shown. The **"… M O R E …"** snippet is used to express that there is more of the same coding methodology that is not shown:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionComposite<T> : IComponent<Action<T>> { … } // One argument
public sealed class ActionComposite<T1, T2> : IComponent<Action<T1, T2>> { … } // Two
// … M O R E …
```

The code above thus shows that there are more **ActionComposite** delegates present. The **"… C O N T R A C T …"** snippet indicates that there is contract code that is not shown.

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public TProduct Create() {
    // … C O N T R A C T S …
    return new TProduct();
}
```

The code on the previous page thus shows that there are contracts present in the **`Create`** method that are not shown.

Participants are the directory of the objects and classes used in a design pattern and their direct roles in the design. All of the design pattern participants in this thesis are underscored in order to clearly identify them as such. For example an *AbstractFactory* participant is thus written as AbstractFactory.

1.6 C# 4.0 and .NET

"C# (pronounced "See Sharp") is a simple, modern, object-oriented, and type-safe programming language. C# has its roots in the C family of languages and will be immediately familiar to C, C++ and Java programmers." (Microsoft, 2007, p. 1)

C# is also a memory managed programming language with hybrid functional and dynamic language features that have evolved from C++, Delphi and Java (Jagger, Perry, & Sestoft, 2007).

This thesis uses C# 4.0 and the .NET framework version 4.0 for its design pattern research. Hejlsberg is the principal architect of C#. He was also involved with the design of Turbo Pascal (Savitch, 1993) and Borland Delphi (Cantu, 2008). C# forms part of the Microsoft .NET universal framework.

Since the first release of C# it has supported features such as inheritance, garbage collection, type-safety, value types, reflection and events. New features in C# 2.0 include static classes, generics, partial classes, covariance and contravariance for delegates, null-coalesce operator, ability to set the accessibility of property accessors independently, nullable types, anonymous delegates and new iterators with the **`yield`** statement (Microsoft, 2005) (Jon, 2010).

New features in C# 3.0 include (Hejlsberg & Torgersen, 2007) extension methods, LINQ, lambda expressions, collection initialisers, object initialisers, local variable type inference, anonymous types, partial methods and automatic properties.

New features in C# 4.0 include (Torgersen, 2008) dynamic language features, contravariant and covariant generic types parameters, optional ref keyword when using COM, optional parameters and named arguments and indexed properties.

Microsoft's active commitment towards programming language improvement makes C# an attractive choice for language research.

1.7 Features used to Implement Reusable Components

1.7.1 Design by contract™.

"Reliability is even more important in object-oriented programming than elsewhere." (Meyer, 1992, p. 40)

Design by contract™ or DbC™ is used to enforce the behavioural and functional rules in most of the pattern components in this thesis. DbC™ is a programming methodology where a contract is placed on a method or a class (Meyer, 1992). Arnout has shown that DbC™ can be used successfully in Eiffel to help componentize design patterns (Arnout, 2004).

Both a pre and a post-condition are placed on a method in order to validate the contract validity of the method. The pre-condition on a method that is defined on a subclass, where the method overrides the original method in the super-class, will weaken the contract because the original interface and contract of the method must be upheld. Adding stronger conditions leads to the possibility of breaking base class method calls and, in turn, breaking the interface. The post-condition on a method that is defined on a subclass and that overrides the original method will strengthen the contract. The reason why a contract is strengthened is because it doesn't affect the interface. An invariant can also be placed on a class to define a contract. An invariant is a predicate that will continuously maintain its truth value during an exact sequence of operations. A subclass would weaken an invariant contract, because the interface and contract of the base class must be upheld.

A large number of the reusable components in this thesis were originally developed in C# 3.0. A basic yet correct DbC™ feature was implemented in C# 3.0 as part of this thesis in order to apply the desired contracts on the patterns. A custom built DbC™ implementation was thus developed for this thesis using the *PostSharp* Aspect Oriented Programming framework (Fraiteur, 2008). Aspect Oriented Programming or AOP is a programming methodology that employs techniques to improve the *separation of concern principle* (Chris, 1989). Separation of concern is a programming principle whereby distinct features of a computer program are separated into non-overlapping pieces of functionality. Dijkstra was the first to mention the principle of *separation of concerns* in his 1974 paper *On the role of scientific thought* (Dijkstra, 1974). A major part of AOP is the *separation of concerns* with regard to *Cross-cutting concerns*. *Cross-cutting concerns* are *aspects* of a program that affect or cross-cut other concerns. An *aspect* is a part of a program that cross-cuts its main concerns and thus violates its *separation of concerns* (Kiczales, et al., 1997). One of the benefits of AOP is that it improves the logical decoupling of components. This can also be a drawback, in that it could create a high number of scattered classes that would be difficult to track and maintain.

Fortunately the latest version of C#, at the time of writing this thesis, does implement design by contract™ through library components. With the release of C# 4.0 the reusable components used in this thesis were refactored to use the C# 4.0 design by contract™ library. DbC™ is implemented in C# 4.0 by a Microsoft DevLabs project called *Code Contracts* (Microsoft, 2011a). DevLabs (Microsoft, 2011c) implement developer focused technology projects and offer them to a large developer community well before they are officially released. *Code Contracts* implements all of the DbC™ requirements, including runtime contract checking, static contract checking and also documentation generation that includes the defined contracts. *Code Contracts* is an offshoot of the Spec# project. Spec# is a research project that tried to evaluate the implementation of contracts in a programming language with features such as aliasing, delegates, call backs, inheritance and multi-threading. Spec# is based on C# 2.0 (Barnett, Leino, & Schulte, 2005) and uses a source rewriter in order to weave the contracts into the code. *Code Contracts* is the outcome of knowledge gained from Spec# in order to evaluate which parts of the research were successful and which were not.

The following code, from the *Code Contracts User Manual,* shows how contracts can be added to C# source code using the *Code Contracts* library (Microsoft, 2011a):

```
C#
------------------------------------------------------------------------------------------------------------
class Rational {
    int numerator;
    int denominator;

    public Rational(int numerator, int denominator) {
        Contract.Requires(denominator != 0); // Add a Requires contract
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public int Denominator {
        get {
            Contract.Ensures(Contract.Result<int>() != 0); // Add a Ensures contract
            return this.denominator;
        }
    }

    [ContractInvariantMethod] // Add an Invariant contract
    void ObjectInvariant() {
        Contract.Invariant(this.denominator != 0);
    }
}
```

The code above shows the implementation of the most important features of contracts, namely pre-conditions, post-conditions and object-invariants. The code shows a subset of a **`Rational`** class with which to model rational numbers. In order to implement an accurate rational number instance, the **`denominator`** must be non-zero. This contract is conveyed as a pre-condition in the constructor with the **`Contract.Requires`** statement. An object-invariant **`ObjectInvariant`** attributed with the **`ContractInvariantMethod`** attribute ensures that the **`denominator`** is always non-zero. Finally, the **`Contract.Ensures`** on the **`Denominator`** getter property ensures that the return value will never be zero.

1.7.2 Mixins or extension methods.

Mixins (Bracha & Cook, 1990) or extension methods (Microsoft, 2010g) allow one to enhance existing types by adding additional methods without the need for a new derived type or altering the original type. Extension methods are a specific variety of static methods; however, they are available as instance methods on a certain enhanced type. There is thus no apparent difference for user code between calling an extension method and a calling a normal method.

Using mixins is thus a technique that adds additional behaviour or functionality to an existing class. More traditional techniques to achieve this are just to modify the existing class and add the desired behaviour. This, however, is not always possible, as the class could be part of a third-party assembly. Without mixins the programmer must inherit from the class in order to implement the desired instance method on the derived class or implement the behaviour in added helper classes. Aggregation can also be used instead of inheritance in order to achieve the same desired effect.

Mixins were first introduced at a company called *Symbolics* into the object-oriented Flavors (Moon, 1986) programming language. Flavors, which is an extension of Lisp, was developed by Howard Cannon at the MIT Artificial Intelligence Laboratory for the Lisp machine and its programming language *Lisp Machine Lisp*. The name "*Mixin"* was motivated by an ice cream shop in Massachusetts called *Steve's Ice Cream Parlour* (Esterbrook, 2001). The ice cream shop offered a special service called a "*Mixin*" that adds extra items such as nuts, fudge or cookies to a basic flavour such as vanilla or chocolate. This term was trademarked by the shop (Mariani, 1999).

Mixins, used in the correct scenarios, can help avoid well-known nuisances linked with multiple inheritance (Balagurusamy, 2008) and boost code reuse.

The APL library uses extension methods in a number of places. For example, it is used by the prototype reusable component in order to add a **`DeepCopy`** method to all objects:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
namespace Apl.Pattern.Gof.Creational.Prototype {
    public static class PrototypeExtention {
        static public T DeepCopy<T>(this T obj) {
            return PrototypeHelper<T>.DeepCopy(obj);
        }
    }
}
```

The **`DeepCopy`** method can now be used by any object that is used in an environment where the **`Apl.Pattern.Gof.Creational.Prototype`** namespace is included. The following code shows how the **`DeepCopy`** extension method is used by the memento pattern in order to make a snapshot of the internal state of the Originator:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public IMemento<TOriginator> CreateMemento() {
    var memento = GetMemento();
    memento.SnapshotState = _originator.DeepCopy(); // Make a copy of the originator
    return memento;
}
```

1.7.3 Attributes.

In C# there is a technique for defining declarative tags, called attributes (Microsoft, 2010d), that can be placed on certain entities in the source code to specify additional meta, or declarative, information. An attribute is a special object in C# that holds meta-information that is linked to an element. A linked element with attributed meta-information is known as the target of that attribute (Liberty, 2001). This meta-information that the attribute contains can be acquired and used during run-time using reflection (Hejlsberg, Torgersen, Wiltamuth, & Golde, 2010). A programmer can define their own custom attributes. An attribute can be attached to entities such as classes, interfaces, namespaces or methods. An attribute can also be global, where it applies to a whole module or assembly. A class in C# is an attribute if it directly or indirectly inherits from the **`System.Attribute`** (Microsoft, 2010c) class.

There are two different types of attribute, namely intrinsic and custom (Liberty, 2001). Intrinsic attributes are provided as a component of the Common Language Runtime (CLR) and are integrated into .NET and C#. The second type is custom attributes (Microsoft, 2010d). Custom attributes are attributes that are created manually for custom purposes. A programmer creates custom attributes in order to add more declarative information to entities in the code.

For example, the **`Obsolete`** intrinsic attribute (Microsoft, 2010m) is associated with a method on a target class to indicate that the method is deprecated. This will cause the C# compiler to issue a warning that the method is obsolete:

```
C#
-----------------------------------------------------------------------------------------------------------
public interface IOrder {
    [System.Obsolete("Use EnterOrder instead.")]
    public void CreateOrder() {}
    public void EnterOrder() {}
}
```

Most systems usually make use of intrinsic attributes only. Custom attributes, however, are a useful mechanism when used in conjunction with reflection (Smith, 1982). Custom attributes are used in certain parts of the APL library.

The following example shows how an APL attribute **`StateAttribute`** is used to define an **`IMyState`** State interface that is used in the implementation of a state design pattern:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Flyweight)]
public interface IMyState {
    void Foo(IFlyweightContext<IMyState> context);
    void Bar(IFlyweightContext<IMyState> context);
}
```

1.7.4 **Generics.**

Generics (Dehnert & Stepanov, 2005) are among the most powerful new features introduced in C# 2.0. With generic programming, algorithms or logic are coded with generic types that are statically defined. The types are defined with generic arguments that are passed through statically during compile time. This allows the coding of templatised functions, or types, that differ only in the type used and thereby duplication is reduced. C# generics are similar to C++ templates (Vandevoorde & Josuttis, 2003) in concept but in implementation they are significantly different. Generics can be used for static polymorphism. Static polymorphism involves the binding of methods to logic during compile time (Meyer, 1986).

Generic programming was made popular by the Ada programming language (Ichbiah, et al., 1979) when it was introduced in that language in 1983 (Musser & Stepanov, 1989). Today, generic programming is found in programming languages such as Ada, Eiffel, Java, C#, Scala, Haskell, C++ (in the form of templates), D and Object Pascal.

Stepanov, who is the chief architect and implementer of the C++ Standard Template Library (Stepanov & Lee, 1995), wrote: "Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream" as quoted in (Stroustrup, 2007, p. 18).

Generics are used extensively in the APL library. The following code shows how generics are used with regard to the command pattern:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ICommand<in TArgument> { void Execute(TArgument arg); }

class ConcreteCommand : ICommand<string> { public void Execute(string text) { Console.WriteLine(text); } }
```

The generic argument **`TArgument`** specifies the type of the argument to the **`Execute`** method on the **`ICommand`** interface. The **`TArgument`** type can thus be supplied during compile time, as seen with the **`ConcreteCommand`** example above.

Generics are also used extensively by the APL library to implement the *curiously recurring template pattern* (CRTP) (G´eraud & Duret-Lutz, 2000). CRTP was first defined in C++ as an *idiom* by Coplien where a

class **Foo** derives from a class template instantiation using **Foo** itself as a template argument (Coplien, 1995). The singleton APL component uses CRTP, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class TheSingleton : Singleton<TheSingleton> {
    // … S N I P …
    private TheSingleton () { … }
    public void Foo() { … }
    public void Bar() { … }
    // … S N I P …
}
```

Note how the instantiated class **TheSingleton** is also passed to the derived class **Singleton** as a generic argument, thus implementing the *curiously recurring template pattern* (CRTP).

1.7.5 Reflection, meta-programming and duck typing.

Reflection is the mechanism by which a computer program can query and possibly alter its own structure and behaviour during runtime (Malenfant, Jacques, & Demers, 1996). The thought of runtime reflection was introduced in 1982 by Brian Cantwell Smith's Ph.D. thesis, which discussed adding structural and behavioural information to 3-Lisp (Smith, 1982).

Reflection is an integral part of C#. The following code shows how reflection can be used in C# to instantiate an instance of a new class **X** and call a method **Y** on the instance:

```
C#
-----------------------------------------------------------------------------------------------------------
// No reflection
var x = new X();
x.Y();

// Reflection
var x = Activator.CreateInstance(null, "X");
var method = x.GetType().GetMethod("Y");
method.Invoke(x, null);
```

Reflection is used extensively by the APL library. For example, the **AutoComposite** component uses reflection to create a new instance of the generic argument **TComposite** in its **Create** factory:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public static AutoComposite<TComponent> Create<TComposite>()
where TComposite : IAutoComponent<TComponent> {
    var autoComposite = Activator.CreateInstance<TComposite>();
    var composite = new AutoComposite<TComponent>(autoComposite);
    // … S N I P …
    return composite;
}
```

Reflection is one of the most fundamental concepts of meta-programming (Klint, 1993). Meta-programming is the creation of computer instructions that manipulate other computer instructions, or

themselves. This manipulation can be done during compile time or run time. Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005) is thus an important language feature employed in order to implement meta-programming. In some programming languages, the language itself is a first-class data type as in Forth, Rebol and Lisp (Lee & Zachary, 1995). This helps make reflection more natural in the language.

Meta-programming is used by the APL library to implement what is known as *duck typing* (Koenig & Moo, 2005). *Duck typing* is a type of dynamic typing where an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface. *Duck typing* refers to the duck test that was coined by James Whitcomb Riley, which may be phrased as follows: "When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck" (Flanagan, 2011, p. 213).

Simple *duck typing* is possible in C# 4.0 with its new dynamic language features (Nierstrasz, et al., 2005). The following example shows how the **`Run`** method can successfully invoke both **`Foo`** and **`Bar`** on the **`dyn`** argument:

```
C#
-----------------------------------------------------------------------------------------------------------
public class X {
    public void Foo() { }
    public void Bar() { }
}

public class Y {
    public void Foo() { }
    public void Bar() { }
}

class Program {
    private static void Run(dynamic dyn) {
        dyn.Foo();
        dyn.Bar();
    }

    private static void Main() {
        var x = new X();
        var y = new Y();
        Run(duck);
        Run(person);
    }
}
```

A more advanced *duck typing* implementation can be found in the *DuckTaper* third party open source library. This library tries to bridge the gap between the dynamic and static worlds in C#, by allowing a dynamic object to be used with a static interface.

The article *Introducing 'The C# Ducktaper' – Bridging the dynamic world with the static world* (de Smet, 2008) explains this phenomenon with the following example:

```
C#
-----------------------------------------------------------------------------------------------------------
interface IDuck {
    void Walk();
    void Walk(int steps);
    object Quack(string name);
    event EventHandler Walking;
}

object possibleDuck = GetDuckFrom(DuckSource.Pond); // Acquire a possibleDuck reference,
                                                    // where possibleDuck probably does not realize
                                                    // the IDuck interface
IDuck duck = possibleDuck.AsIf<IDuck>(); // Try to convert the reference to an IDuck interface
duck.Walking += (o, e) => { Console.WriteLine("Duck is walking"); }; // Use a method on the interface
duck.Walk(); // Use a method on the interface
Console.WriteLine(duck.Quack("Bart")); // Use a method on the interface
duck.Walk // Use a method on the interface
```

An object **possibleDuck** is acquired where its contract is unknown. The **AsIf** extension method, which is part of the *DuckTaper* library, will try to convert the **possibleDuck** instance into an **IDuck** instance. The **AsIf** extension method creates a new class during runtime that implement all of the methods of the **IDuck** interface. Thereafter, every request on the **duck** instance delegates an invocation to the **possibleDuck** instance where the method signatures are exactly the same. If no method signature is available, an exception is thrown. The newly created instance, which implements the **IDuck** interface, can be seen as a proxy (Gamma, Helm, Johnson, & Vlissides, 1994) that thunks the method request to the appropriate target. Thunking can be seen as a wrapper function that directs an invocation to an appropriate target (Driesen & Hölzle, 1996) (Stroustrup, 1987).

The *DuckTaper* library can be extended by implementing one's own method lookup table and forwarding an invocation request to the appropriate method in the table. This can be achieved by using the **IDynamicInvoker** *DuckTaper* interface, as seen in the following example:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class AutoAdapter<TTarget, TAdaptee> : IDynamicInvoker
    where TTarget : class {
    private TAdaptee _adaptee;

    // … S N I P …

    public AutoAdapter(TAdaptee adaptee) { … }

    // … S N I P …

    public void RegisterAction(string methodName, AdapterAction<TAdaptee> operation) {
        // … C O N T R A C T S …
        Validate(methodName);
        var method = typeof(TTarget).GetMethod(methodName);
        _operationDictionary.Add(new DynamicMethod(method), operation);
    }

    public void RegisterAction(MethodInfo method, Action operation) { … }

    // … S N I P …

    public object Invoke(string methodName, object[] args) {
        // … S N I P …
```

```
        var operation = GetAdapteeOperation(methodName, args);
        return operation != null ? operation.DynamicInvoke(_adaptee, args) : null;
        // … S N I P …
    }

    public TTarget Target {
        get { return DoubleCheckedLock<TTarget>.Create(_target, this, () => this.AsIf<TTarget>(true)); }
    }
}
```

The **`IDynamicInvoker`** interface enforces the implementation of the **`Invoke(string methodName, object[] args)`** contract. Every invocation made on the reference returned by the **`Target`** property is delegated to the **`Invoke`** method. If an **`AdapterAction`** was registered with the instance of the **`AutoAdapter`** with the same method signature as the one received by the **`Invoke`** method, then the **`AdapterAction`** is invoked.

`AdapterAction` APL delegates can be registered with the adapter using the **`RegisterAction`** methods. The simplest **`RegisterAction`** method identifies a method just by its name:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
Adapter.RegisterAction("Foo", x => Console.WriteLine("Hello World" + x.FooBar());
```

This **`RegisterAction`** method cannot be used if method overloading is desired. This is because multiple methods with the same name, but with different arguments, would then exist. The **`RegisterAction`** method that is supplied with the reflective type **`MethodInfo`** can be used for method overloading:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
Adapter.RegisterAction(myMethod, x => Console.WriteLine("Hello World") + x.FooBar());
```

The techniques shown above are used extensively by the APL library in order to create reusable design pattern components.

The *duck typing* most used in the APL library is that implemented by means of the *DuckTaper* library. The *duck typing* used is thus not a direct language feature of C#. The *DuckTaper* library uses reflection and meta-programming language features in order to achieve *duck typing*.

1.7.6 Anonymous methods (anonymous functions).

An anonymous method (Microsoft, 2010b) or an anonymous function is an un-named method defined inside source code and is thus not linked to an identifier. In C# an anonymous method allows a code block to be passed as arguments instead of a standalone coded method.

Anonymous methods or functions were first added in the Lisp programming language in 1958 (Stoyan, 1984).

Anonymous methods can be created in C# by using the delegate keyword:

```
C#
-----------------------------------------------------------------------------------------------------------
delegate void Action(string text);
Action action = delegate(string text) { Console.WriteLine(text); };
```

Anonymous methods can also be created in C# by using lambda expressions:

```
C#
-----------------------------------------------------------------------------------------------------------
Action action = (x) => Console.WriteLine(x);
```

In C# anonymous methods can only be passed as a code block to a delegate parameter.

1.7.7 **Method references or delegates.**

In C# a delegate (Microsoft, 2010e) is a special type that references a method signature. A delegate can be assigned to method implementations with the same method signature. When a delegate is assigned to a method it can be used in exactly the same way as any other normal method. A delegate can also be used like any other reference type in C#, where it can be passed in as a method argument or be used as a class attribute.

Any method that has the same signature as a specific delegate can be linked to that delegate. With delegates new code can be plugged into defined classes and method calls can be changed during runtime.

Delegates are used extensively in the APL library. For example, in the **`ActionCommand`** APL component the **`Action`** family of delegates is used to describe the Receiver logic:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class ActionCommand : ICommand { // No arguments
    protected Action ExecuteReceiver;

    public ActionCommand() { }
    public ActionCommand(Action executeReceiver) { ExecuteReceiver = executeReceiver; }

    public void Execute() {
        if(ExecuteReceiver == null) return;
        ExecuteReceiver();
    }
}

C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var concreteCommand = new ActionCommand(() => Console.WriteLine("The command was invoked!"));
invoker.Process(concreteCommand);
```

In the example above the code block is passed to the **`ActionCommand`** constructor with a lambda expression (J'arvi, Freeman, & Crowl, 2007).

1.7.8 Action and Func family of library delegates.

The C# **Func** group of delegates (Microsoft, 2010h) is used to embody a method that can be used as an argument without explicitly creating a custom delegate. The referenced method must match the method definition that is defined by this specific delegate.

The **Func** group of delegates is implemented in the C# standard library as shown below:

```
C#
-----------------------------------------------------------------------------------------------------------------
public delegate TResult Func<T,TResult>(T arg)
public delegate TResult Func<T1,T2,TResult>(T1 arg1, T2 arg2)
public delegate TResult Func<T1,T2,T3,TResult>(T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T1,T2,T3,T4,TResult>(T1 arg1, T2 arg2, T3 arg3, T4 arg4)
/* … M O R E … */
```

A **Func** delegate can also be used with anonymous methods (Eric, 2007). A lambda expression (Microsoft, 2010i) can be assigned to a **Func** delegate, as the below example shows:

```
C#
-----------------------------------------------------------------------------------------------------------------
Func<string> function1 = () => return "Hello World";
Console.WriteLine(function1);
/* Output
Hello World
*/

Func<string, string> function2 = (x) => return x;
Console.WriteLine(function2("Hello World"));
/* Output
Hello World
*/

Func<string, string, string> function3 = (x) => return x + " " + y;
Console.WriteLine(function3("Hello", "World"));
/* Output
Hello World
*/
```

The **Action** group of delegates (Microsoft, 2010a) in the C# standard library is almost the same as the **Func** delegates (Microsoft, 2010h), except that they do not have a return value. They are therefore actions and not functions. The following example shows the usage of **Action** delegates together with lambda expressions (Microsoft, 2010i):

```
C#
-----------------------------------------------------------------------------------------------------------------
Action action1 = () => Console.Write("Hello World");
Action1()
/* Output
Hello World
*/

Action<string> action2 = (x) => Console.Write(x);
Action1("Hello World");
/* Output
Hello World
*/
```

```
Action<string, string> action3 = (x) => Console.Write(x + y);
Action1("Hello", "World");
/* Output
Hello World
*/
```

Both the **Func** and **Action** delegates are used extensively by the APL library. For example, the **ActionCommand** APL component takes an **Action** as a Receiver that it will invoke in its executing method:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class ActionCommand : ICommand {
    protected Action ExecuteReceiver;

    public ActionCommand() { }
    public ActionCommand(Action executeReceiver) { ExecuteReceiver = executeReceiver; }

    public void Execute() {
        if(ExecuteReceiver == null) return;
        ExecuteReceiver();
    }
}
```

The **Action** and **Func** C# groups of delegates are not a language feature. They are functionality that is available because of standard library components. The delegates are mentioned here because they are used extensively within the APL library components. A language that offers method references or delegates should be able to offer the same functionality as the **Action** and **Func** C# groups of delegates.

1.7.9 Dynamic typing.

C# 4.0 provides a **dynamic** keyword (Microsoft, 2011b) that adds dynamic typing language features in what used to be a statically typed language. With static typing, type checking is performed by the programming language during compile time. With dynamic typing, type checking is performed during runtime. A dynamic language thus does not do type checking during compile time (Scott, 2009). The type of an expression or variable is not necessarily known at compile time. Storage limitations are verified only during run time and are overlooked at compile time. Semantic analysis thus transpires only at run time.

The C# 4.0 programming language can be seen as both dynamic and static because it has features that support both (Hejlsberg, Torgersen, Wiltamuth, & Golde, 2010). C# first started off as a statically typed language. It has been transformed into a hybrid dynamically typed language in which one uses the newly added dynamic features. In C# 4.0 an object defined as type **dynamic** sidesteps static type checking entirely.

The Dynamic Language Runtime (DLR) (Hugunin, 2007) is one of the latest APIs in the .NET Framework. It offers the mechanism that implements the dynamic type features in C# and is used

extensively by new dynamic programming languages in .NET such as IronPython (Python Software Foundation, 2011) and IronRuby (Ruby-Doc.Org, 2011).

The new dynamic features in the C# language are not used extensively by the APL library. This is because design patterns are usually strongly typed by nature. The chain of responsibly (Gamma, Helm, Johnson, & Vlissides, 1994) APL component **`DynamicChainOfResponsibility`** uses the **`dynamic`** keyword in C#:

```
C# (APL Example)
----------------------------------------------------------------------------------------------------
var factory = new DynamicChainOfResponsibilityFactory();
dynamic handler = factory.Create( … );
handler.Foo( … );
```

The **`Foo`** method on the chain of responsibility Handler instance **`handler`**, which is of type **`dynamic`**, is evaluated during runtime. If the **`Foo`** method is not found on the Handler then it is invoked on the next Handler in the chain until one is found or the end of the chain is reached.

1.7.10 Lambda expressions.

A lambda expression is an anonymous function containing statements and expressions (Samko, et al., 2006). In C# a lambda expression can be used to create expression tree types and delegates (Torgersen, 2007). Lambda expressions offer an abridged and functional syntax for writing anonymous methods. In C#, the arguments of a lambda expression can be explicitly or implicitly typed. The arguments of a lambda expression may thus be explicit or inferred.

Church invented lambda expressions with his creation of lambda calculus in 1936, where all methods are anonymous (Church, 1936). Landin's classical paper of 1965 shows that lambda calculus can be successfully implemented and used in a procedural programming language such as ALGOL 60 (Landin, 1965).

Lambda expressions in C# use the operator **`=>`** (Microsoft, 2010i). The lambda operator is read as *"goes to"* (Microsoft, 2010i). Input parameters are specified by the left side of the operator. The right side of the operator defines the statement block or expression. Lambda expressions can also be assigned to delegates (Kennedy, 2006):

```
C#
----------------------------------------------------------------------------------------------------
delegate int Sum(int i);
static void Main() {
    Sum theDelegate = x => x + x;
    int i = theDelegate(15);
    Console.WriteLine("Sum : " + i); // Sum : 30
}
```

The code on the previous page can be refined by replacing the **Sum** delegate with the **Func<T, TResult>** generic delegate that is supplied by the .NET framework:

```
C#
-----------------------------------------------------------------------------------------------------------
static void Main() {
    Func<int, int> sum = x => x + x;
    int i = sum(15);
    Console.WriteLine("Sum : " + i); // Sum : 30
}
```

Lambda expressions are available in a large number of programming languages, especially functional languages, such as Haskell (Thompson, 1999), Lisp (Seibel, 2004), Erlang (Armstrong, 2007), Scala (Wampler & Payne, 2009) and F# (Smith, 2009).

The inclusion of lambda functions in C# shows its shift to a more declarative style of programming (Lloyd, 1994) as in functional languages. Lambda expressions are used extensively in the APL library. For example the **AutoAbstractFactory** APL component takes in the **Factory<TResult>** family of creational delegates. The code below shows how a lambda expression **() => new ProductA1()** is used to inject a creational anonymous function with the **RegisterOperation** for the **"CreateProductA"** method available on the **IAbstractFactory** interface:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var factory = new AutoAbstractFactory<IAbstractFactory>();
// Register a creational lambda expression representing the CreateProductA method on the AbstractFactory
factory.RegisterOperation<IAbstractProductA>("CreateProductA", () => new ProductA1());
```

1.8 Contributions of this Thesis

This section provides an overview of the scientific contributions of this thesis. The thesis has

- shown that modern language features are beneficial in the creation of reusable design pattern components.
- shown that duck typing (Koenig & Moo, 2005) is a powerful language feature with which to implement reusable design patterns.
- built on the argument that reusable design patterns are a useful solution for the traceability, reusability, implementation overhead and maintainability problems associated with design patterns.
- shown that it is possible to implement reusable design pattern components in C#.

Chapter 2

2 PROTOTYPE

2.1 Introduction

Prototypes (Gamma, Helm, Johnson, & Vlissides, 1994) (Meyer, 2000) enable clients to select at run-time what objects they want to create. The prototype pattern provides a simple solution for facilitating dynamic object creation (Nierstrasz, et al., 2005) and run-time management of a registry of objects.

The intent of the prototype design pattern is to create a new instance by making a copy of an existing prototype object during run time.

2.1.1 Structure.

The following figure shows the formal structure of the prototype design pattern:

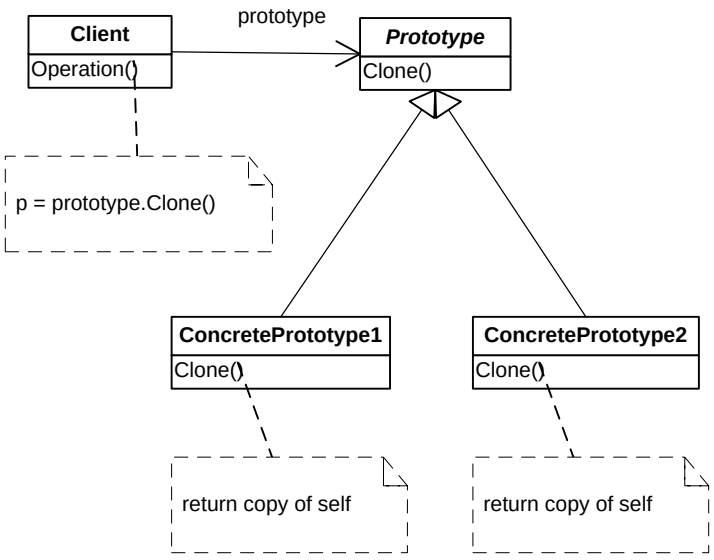

Figure 1. Prototype formal structure.

2.1.2 Participants.

The classes and/or objects participating in the prototype design pattern are:

- Prototype

  The Prototype is the class that declares the cloning interface.

- ConcretePrototype

  The ConcretePrototype is the class that implements the cloning interface.

- Client

  The Client is the user of the Prototype asking it to clone itself.

2.2 Library Components

2.2.1 The Prototype component.

The prototype component is implemented by a **`DeepCopy`** generic extension method (Microsoft, 2010g). The **`DeepCopy`** extension method thus makes a clone from an original object. The extension method is generic, and so the method is available on all classes of type **`T`**:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
namespace Apl.Pattern.Gof.Creational.Prototype {
    public static class PrototypeExtention {
        static public T DeepCopy<T>(this T obj) {
            Contract.Requires<ArgumentNullException>(obj != null, "Input argument obj cannot be null");
            Contract.Ensures(Contract.Result<T>() != null);
            return PrototypeHelper<T>.DeepCopy(obj);
        }
    }

    // … S N I P …
}
```

The actual clone or copy processing of the original object is delegated to the **`DeepCopy`** method on the generic **`PrototypeHelper`** APL component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public static T DeepCopy(T obj) {
        Contract.Requires<ArgumentNullException>(obj != null, "Input argument obj cannot be null");
        Contract.Ensures(Contract.Result<T>() != null);
        var memoryStream = new MemoryStream(); // Create a new memory stream
        var binaryFormatter = new BinaryFormatter(); // Create a new binary formatter
        binaryFormatter.Serialize(memoryStream, obj); // Serialize the object to the memory stream
        memoryStream.Seek(0, SeekOrigin.Begin); // Go back to the beginning of the stream
        var copy = (T)binaryFormatter.Deserialize(memoryStream); // Deserialize the memory to an object
        memoryStream.Close(); // Close the stream
        return copy; // Return the deserialized object
}
```

The **`DeepCopy`** extension method first serializes the state of the prototype to a memory stream. Next, it deserializes the memory stream back into a new copy with the original state of the Prototype. The **`DeepCopy`** extension method uses the same technique shown by Bishop in her book *C# 3.0 Design Patterns* (Bishop, 2007).

The **DeepCopy** extension method is thus available on any object that uses the **Apl.Pattern.Gof.Creational.Prototype** namespace:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
// Namespace that makes the DeepCopy extension method available
uses Apl.Pattern.Gof.Creational.Prototype {
    // … S N I P …
    var newFoo = foo.DeepCopy();
    // … S N I P …
}
```

A slight drawback of the prototype component is that the class being copied must be attributed with the C# **Serializable** (Bishop, 2007) (Albahari & Albahari, 2007) (Microsoft, 2010k) attribute.

The C# **NonSerialized** (Albahari & Albahari, 2007) attribute can be used to control which fields of a prototype class must not be copied:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[Serializable]
class ThePrototype {
    private string _state1;
    [NonSerialized] private string _state2; // Don't serialize this field…
    private string _state3;
    // … S N I P …
}
```

In the above code, the **_state2** field will not be serialized when the **DeepCopy** method is invoked on an instance of the **ThePrototype** class, thus giving more control on how the object must be cloned.

2.3 Theoretical Examples

The following example shows how the prototype component is used in a formal pattern setting. At the heart of the prototype design pattern is the clone operation. When using the APL library the clone operation is automatically added to all classes by using the extension method C# language feature (Microsoft, 2010g). The hand coded implementation of the clone method is thus no longer necessary as per the traditional pattern. The only constraint that exists is that the ConcretePrototype must be attributed with the **Serializable** attribute.

In the following theoretical example, **Prototype** defines a Prototype with a base state:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[Serializable]
abstract class Prototype {
    private readonly string _state;
    protected Prototype(string state) { _state = state; }
    public string State { get { return _state; } }
}
```

Note that no hand coded clone method is defined on the **`Prototype`** class. Next, **`ConcretePrototype1`** and **`ConcretePrototype2`** classes are implemented, that define the ConcretePrototypes:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[Serializable]
class ConcretePrototype1 : Prototype { public ConcretePrototype1(string state) : base(state) {} }
[Serializable]
class ConcretePrototype2 : Prototype { public ConcretePrototype2(string state) : base(state) {} }
```

Both of the ConcretePrototypes must have the **`Serializable`** attribute. This is because the internal engine of the prototype component serializes and deserializes the entire prototype in order to make the copy. The ConcretePrototypes can now use the **`DeepCopy`** extension method from the APL library in order to make a clone of themselves:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var concretePrototype1 = new ConcretePrototype1("Foo");
var copy1 = concretePrototype1.DeepCopy(); // Make a clone of the concretePrototype1 object
Console.WriteLine("Cloned : {0}", concretePrototype1.State);
var concretePrototype2 = new ConcretePrototype2("Bar");
var copy2 = concretePrototype2.DeepCopy(); // Make a clone of the concretePrototype2 object
Console.WriteLine("Cloned : {0}", concretePrototype2.State);

/*
Output:
Cloned : Foo
Cloned : Bar
*/
```

The output of this example shows that the state of the newly copied objects is the same as the original state.

2.4 Outcome

The componentization of the prototype design pattern is a partial success because it meets most of the requirements listed in section 1.4:

- **Completeness**: The prototype design pattern library component covers all cases described in the original design pattern.

- **Usefulness**: The prototype design pattern library component is partially useful because it does not solve all of the prototype scenarios desired by a developer. A developer can fine-tune which parts of a Prototype instance must or must not be cloned, using the NonSerialized C# (Albahari & Albahari, 2007) attribute. When cloning, however, deep and shallow copying should be taken into consideration for composition, aggregation and association relationships. Aggregation characterises a part-of or part-whole relationship. Composition is a stronger type of association relationship. Composition generally has a strong life cycle dependency. If a class

holding a composition relationship is destroyed, usually every composition instance that it holds is destroyed as well. An association represents a weaker relationship, for example where a container instance needs to send messages to the associated dependent instance. An association can thus be a reference to a service instance. Ideally, when cloning an instance, a deep copy should be performed on composition relationships and a shallow copy should be performed on aggregation and association relationships. Unfortunately, in C# there is no meta-information available that define the three different relationships. The reusable component thus can only implement a deep copy.

- **Faithfulness**: The implementation of the prototype pattern follows a path slightly different from the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). In *Design Patterns* an interface is defined with a clone method. All classes which implement the prototype design pattern implement the prototype interface. The implementation of the prototype pattern in the APL library injects a clone method into classes using C# extension methods (Esterbrook, 2001) (Jesse & Xie, 2008). The end structure is the same however, where a clone method is available on a certain class. A slight drawback is that a class must be attributed with the `Serializable` C# (Albahari & Albahari, 2007) attribute in order to participate within the reusable prototype pattern.

- **Type-safety**: The prototype library component is fully type-safe.

- **Extended applicability**: The prototype library component does not cover more cases than the original prototype pattern.

- **Performance**: Serialization will always be slower than manually creating a clone algorithm for a certain class. This is because serialization must use reflection and must thus evaluate the meta-information of a certain object during runtime. Serialization is, however, used extensively in C# libraries such as WCF and Object Relational Mappers (ORM), and is thus a valid solution.

The prototype library component is partially componentized because the developer using it does not have to implement any prototype boiler plate code. The prototype library component however can only be used as a deep copy.

The following language features are fundamental to the implementation or usage of the reusable prototype design pattern component: Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Mixins (Extension Methods) (Esterbrook, 2001) (Jesse & Xie, 2008) and Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005).

Chapter 3

3 SINGLETON

3.1 Introduction

The singleton design pattern ensures that that there is only one instance of each class and offers a universal point of access to it (Gamma, Helm, Johnson, & Vlissides, 1994).

The intent can thus be described as:

- The ability to enforce that a class has only a single instance.
- The ability to avoid redundant instance creation, especially for stateless objects.
- The ability to manage the responsibility of maintaining universal access to the single instance of a class.

3.1.1 Structure.

The following figure shows the formal structure of the singleton design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

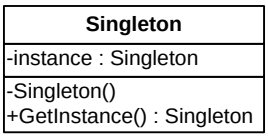

Figure 2. Singleton structure.

3.1.2 Participants.

The classes and/or objects participating in the singleton design pattern are:

- **Singleton**

  A Singleton defines a static `GetInstance` operation on a class that lets clients access its unique instance. It also governs the creation and controls the subsequent management of its own unique instance.

3.2 Library Components

3.2.1 The Singleton component.

The singleton component in the APL library is implemented using the *curiously recurring template pattern* (Coplien, 1995). This means that the generic singleton component must be inherited from the Singleton being implemented, as seen below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
class TheSingleton : Singleton<TheSingleton> { … }
```

The following code shows how the reusable **`Singleton`** class is implemented in the APL library:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class Singleton<T> : BaseSingleton<T>
    where T : class {

    // Boolean indicating a Thread Local Static Singleton
    private static readonly bool TLS;

    // Aquire a reference to a Singleton for class T
    public static T Instance {
        get {
            Contract.Ensures(Contract.Result<T>() != null);

            return TLS ? SingletonTLSCreator.Instance : SingletonCreator.Instance;
        }
    }

    static Singleton() {
        var singletonAttribute = GetAttribute();

        if(singletonAttribute != null && singletonAttribute.ThreadStatic)
            TLS = true;
    }

    // … S N I P …

    // Protected Singleton constructor
    protected Singleton() { }

    // A normal Singleton creator
    private class SingletonCreator {
        internal static readonly T Instance = CreateHelper<T>.CreateFromPrivateConstructor();
    }

    // A Thread Local Storage Singleton creator
    private class SingletonTLSCreator {
        [ThreadStatic]
        private static T _instance;

        internal static T Instance {
            // Create a Singleton instance from its private constructor if an instance
            // does not exist already
            get {
                return _instance ?? (_instance = CreateHelper<T>.CreateFromPrivateConstructor());
            }
        }
    }
}
```

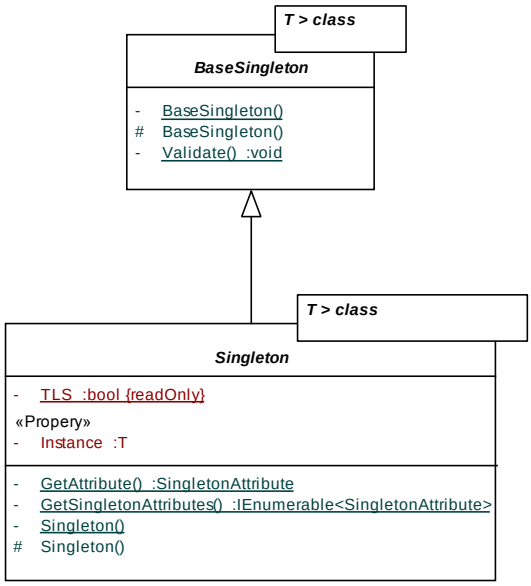

Figure 3. UML class diagram of the Singleton APL component.

Figure 3 shows a UML class diagram of the **`Singleton`** APL component.

The **`Singleton`** component implements two different singleton variants. One is per process and the other is per thread as shown in the article *A Per-Thread Singleton Class* (Chaudhry, 2002) and also in the paper *Thread-Specific Storage - An Object Behavioral Pattern for Efficiently Accessing per-Thread State* (Harrison & Schmidt, 1997). The above-mentioned singleton variant is disclosed using the **`ThreadStatic`** property on the APL **`SingletonAttribute`** attribute. The implementation of the **`SingletonAttribute`** is shown below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false, Inherited = false)]
public class SingletonAttribute : System.Attribute, IPatternClassAttribute {
    public SingletonAttribute() { ThreadStatic = false; }
    public SingletonAttribute(bool threadStatic) { ThreadStatic = threadStatic; }

    public bool ThreadStatic { get; set; }
    public bool Validate(Type classType) { … }
}
```

The **`SingletonAttribute`** APL attribute's **`bool ThreadStatic`** property thus defines whether the Singleton is single per process or per thread. If it is a Singleton per process, then the single instance is created using the **`SingletonCreator`** inner class in the **`Singleton`** component. The Singleton instance is only created on the very first instantiation of the internal **`SingletonCreator`**. The **`CreateFromPrivateConstructor`** method on the **`CreateHelper`** helper class creates an instance of the **`T`**

class using reflection (Sobel & Friedman, 1996), because the constructor of the Singleton must be private.

In the **`Singleton`** component, the **`SingletonTLSCreator`** inner class is used to instantiate a Singleton per thread (Chaudhry, 2002). A mechanism known as thread local storage is used (Stein & Shah, 1992) in which a variable is assigned per thread. The **`ThreadStatic`** (Microsoft, 2010n) attribute on the **`_instance`** field on the **`SingletonTLSCreator`** inner class tells the runtime that a unique instance of the field must exist per thread. The **`Instance`** property on the **`SingletonTLSCreator`** class creates an instance of the thread static Singleton only if one does not already exist for the specific thread on which the logic is executed.

The **`Singleton`** component also inherits from an abstract **`BaseSingleton<T>`** class that defines the most common functionality for all APL Singleton components, such as validations:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class BaseSingleton<T>
    where T : class {
    static BaseSingleton() { Validate(); }
    protected BaseSingleton() { }
    private static void Validate() { … }
}
```

3.3 Theoretical Examples

The following example shows the usage of the **`Singleton`** APL component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
class TheSingleton : Singleton<TheSingleton> { private TheSingleton() { } }

class Program {
    static void Main() {
        var s1 = TheSingleton.Instance;
        var s2 = TheSingleton.Instance;

        if(s1 == s2) {
            Console.WriteLine("Objects are the same instance");
        }

        Console.ReadKey();
    }
}

/* Output
Objects are the same instance
*/
```

The **`TheSingleton`** hand coded class inherits from the **`Singleton`** component, passing itself as the generic argument. The constructor must be made private because the validation in the **`Singleton`** component throws an exception during runtime if the constructor is not private.

In this example, the client calls the **`Instance`** property of **`TheSingleton`** twice, storing it in two separate variables. If no **`SingletonAttribute`** APL attribute is placed on the Singleton then the pattern variant defaults to a singleton per process. The output shows that the variables reference the same instance, thus the **`Instance`** property has returned the same single object.

The next example shows the usage of the **`Singleton`** component configured to return an instance per thread:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[Singleton(ThreadStatic = true)]
public class TheSingleton : Singleton<TheSingleton> {
    private TheSingleton() {
        Console.WriteLine("A new singleton was creater on thread id: " +
                          Thread.CurrentThread.ManagedThreadId);
    }

    public void DoSomething() { Console.WriteLine("Doing something on thread id: " +
                                                  Thread.CurrentThread.ManagedThreadId); }
}

public class ThreadStaticExample {
    static void Main() {
        var thread1 = new Thread(() => {
            var s1 = TheSingleton.Instance;
            s1.DoSomething();
            var s2 = TheSingleton.Instance;
            s2.DoSomething();
            if(s1 == s2) { Console.WriteLine("Objects are the same instance for thread 1"); }
        });

        var thread2 = new Thread(() => {
            var s1 = TheSingleton.Instance;
            s1.DoSomething();
            var s2 = TheSingleton.Instance;
            s2.DoSomething();
            if(s1 == s2) { Console.WriteLine("Objects are the same instance for thread 2"); }
        });

        thread1.Start();
        thread2.Start();
        Thread.Sleep(10000);

        Console.WriteLine();
        Console.Write("Press any key to exit.");
        Console.Read();
    }
}

/* Output
A new singleton was creater on thread id: 11
Doing something on thread id: 11
Doing something on thread id: 11
Objects are the same instance for thread 1
A new singleton was creater on thread id: 12
Doing something on thread id: 12
Doing something on thread id: 12
Objects are the same instance for thread 2
*/
```

The **`TheSingleton`** class is attributed with the **`Singleton`** attribute where **`ThreadStatic`** is set to true. The client creates two separate threads during runtime. Each thread calls the **`Instance`** property twice. The output shows that the constructor of the Singleton instance was called twice, once for each thread. The second call to **`Instance`** in each thread has thus not created a new instance of the **`TheSingleton`** class, but returned the instance already allocated to the specific thread.

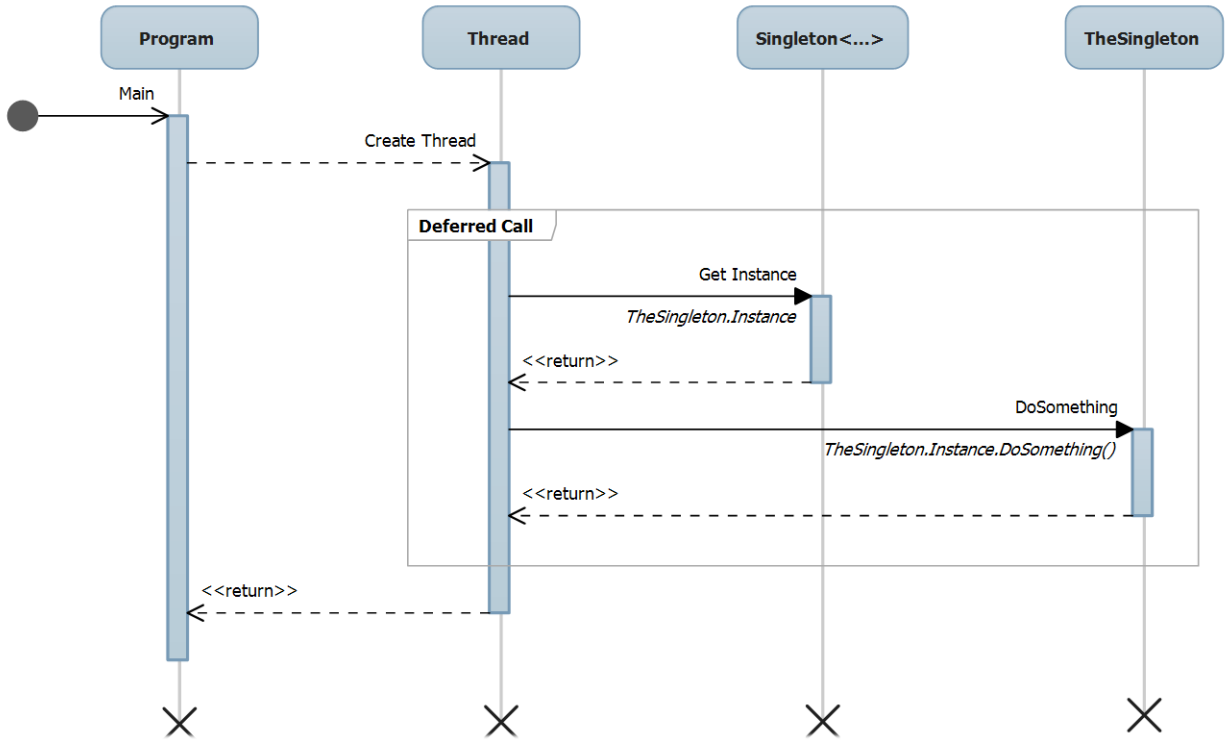

Figure 4. UML sequence diagram for the thread static Singleton APL component example.

Figure 4 shows a sequence diagram for the thread static **`Singleton`** APL component example.

3.4 Outcome

The componentization of the singleton design pattern is a success because it meets all the requirements listed in section 1.4:

- **Completeness**: The singleton design pattern library components cover all cases described in the original core design pattern.

- **Usefulness**: The singleton design pattern library component is useful because it solves all of the singleton's defined intent. The singleton design pattern library component is also easy to understand and simple to use.

- **Faithfulness**: The implementation of the singleton design pattern library component mostly follows the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). The Singleton implementation has been slightly changed where the static **`GetInstance`** operation that lets clients access its unique instance is replaced by a static read-only **`Instance`** property.

- **Type-safety**: All of the library components are fully type-safe.

- **Extended applicability**: The singleton library component covers more cases than the original singleton pattern. The reusable singleton component allows a Singleton to be created per thread.

- **Performance**: Using the singleton component does not have a performance impact.

A developer still needs to make the default constructor of a class private when implementing a Singleton using the singleton library component. The singleton library component, however, is still classified as fully componentized because the boiler plate code that must be implemented by a developer is not significant.

The following language features are fundamental to the implementation or usage of the reusable singleton component: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Attributes (Nagel, Evjen, Glynn, & Watson, 2010) and Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005).

Chapter 4

4 ABSTRACT FACTORY

4.1 Introduction

The abstract factory design pattern offers an interface for creating families of related objects that assist in decoupling applications from the concrete implementation of an entire framework or library (Gamma, Helm, Johnson, & Vlissides, 1994) (McConnell, 1993).

The intent can thus be described as:

- The ability to decouple the concrete family of objects from their users.
- The ability to be able to choose at runtime a concrete factory that implements creational contracts whose sole responsibility is to instantiate a specific family of related classes.

4.1.1 Structure.

The following figure shows the formal structure of the abstract factory design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

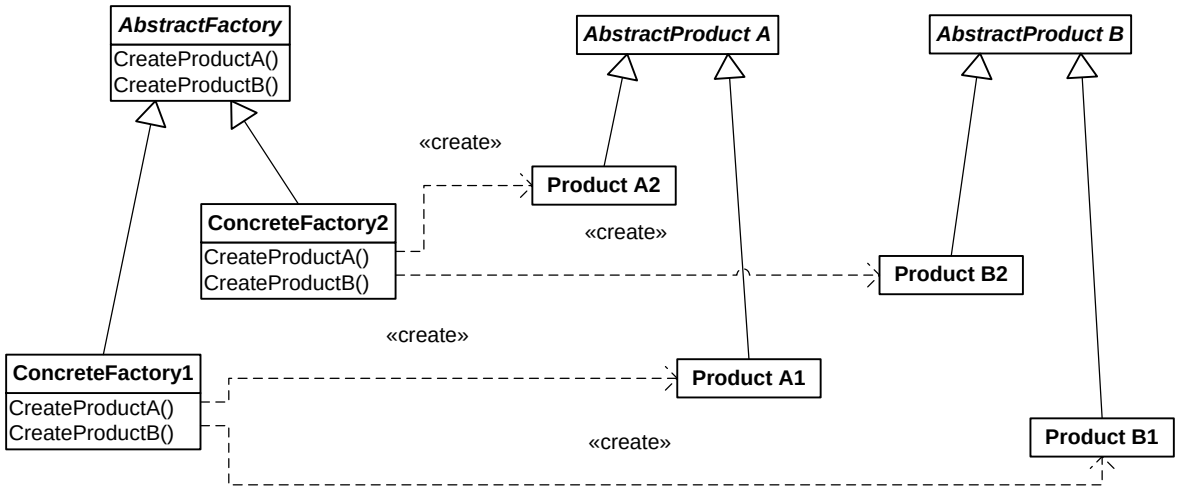

Figure 5. Abstract factory structure.

4.1.2 Participants.

The classes and/or objects participating in the abstract factory design pattern are:

- **AbstractFactory**

  An AbstractFactory defines an interface for creational operations that instantiates an AbstractProduct.

- **ConcreteFactory**

  A ConcreteFactory implements the creational operations with which to instantiate Product objects.

- **AbstractProduct**

  An AbstractProduct defines an interface for a specific type of Product object.

- **Product**

  A Product defines a concrete product object that implements the AbstractProduct interface. It is instantiated by the corresponding ConcreteFactory.

- **Client**

  A Client uses the interfaces defined by the AbstractFactory and AbstractProduct participants.

4.2 Library Components

4.2.1 The AutoAbstractFactory component.

The **`AutoAbstractFactory`** APL component uses dynamic *duck typing* (Koenig & Moo, 2005) in order to hook up creational methods or creational anonymous functions (Ierusalimschy, 2003) with methods defined in an AbstractFactory contract. The **`AutoAbstractFactory`** has one generic argument **`TAbstractFactory`** that defines the AbstractFactory contract. The implementer of the **`AutoAbstractFactory`** must use the **`RegisterOperation`** methods to register the creational methods or creational anonymous functions. Each **`RegisterOperation`** method validates whether the registered creational method signature exists on the **`TAbstractFactory`** AbstractFactory interface and adds it to the internal dictionary if it does:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class AutoAbstractFactory<TAbstractFactory> : IDynamicInvoker
    where TAbstractFactory : class {
    private readonly Dictionary<DynamicMethod, Delegate> _operationDictionary =
        new Dictionary<DynamicMethod, Delegate>();
    private volatile TAbstractFactory _targetCache;
```

```
public AutoAbstractFactory() { _targetCache = null;  } // Constructor

[ContractInvariantMethod]
private void ContractInvariant() {
    Contract.Invariant(operationDictionary!= null, "The dictionary cannot be null");
    // … M O R E  C O N T R A C T S …
}

// Register methods for the Factory set of delegates:

// Register a creational method with no arguments
public void RegisterOperation<TResult>(string methodName, Factory<TResult> operation) {
    Contract.Requires<ArgumentException>(!string.IsNullOrEmpty(methodName) != null,
                                         "Argument methodName cannot be null or empty");
    Contract.Requires<ArgumentException>(operation != null,
                                         "Argument operation cannot be null");
    // … C O N T R A C T S …
    Validate();
    _operationDictionary.Add(new DynamicMethod(operation.Method), operation);
}

// Register a creational method with no arguments
public void RegisterOperation<TResult>(MethodInfo method, Factory<TResult> operation) {
    // … C O N T R A C T S …
    Validate();
    _operationDictionary.Add(new DynamicMethod(method), operation);
}

// Register a creational method with one argument
public void RegisterOperation<TResult, TArg1>(string methodName,
                                               Factory<TResult, TArg1> operation) { … }

// Register a creational method with one argument
public void RegisterOperation<TResult, TArg1>(MethodInfo method,
                                               Factory<TResult, TArg1> operation) { … }

// Register a creational method with two arguments
public void RegisterOperation<TResult, TArg1, TArg2>(string methodName,
                                                     Factory<TResult, TArg1, TArg2> operation) { … }

// Register a creational method with two arguments
public void RegisterOperation<TResult, TArg1, TArg2>(MethodInfo method,
                                                     Factory<TResult, TArg1, TArg2> operation) { … }

// … M O R E …

// Register methods for the IFactory set of interfaces:

public void RegisterOperation<TResult>(string methodName, IFactory<TResult> factory) { … }
public void RegisterOperation<TResult>(MethodInfo method, IFactory<TResult> factory) { … }
public void RegisterOperation<TResult, TArg1>(string methodName,
                                                   IFactory<TResult, TArg1> factory) { … }
public void RegisterOperation<TResult, TArg1>(MethodInfo method,
                                                   IFactory<TResult, TArg1> factory) { … }

// … M O R E …

public object Invoke(string methodName, object[] args) {
    Contract.Requires<ArgumentException>(!string.IsNullOrEmpty(methodName),
                                         "Argument path cannot be null");
    var operation = GetOperation(methodName, args);
    if(componentOperation != null) { return operation.DynamicInvoke(args); }

    throw new Exception("Creational method not found");
}

public TAbstractFactory Target {
```

```
        get {
            Contract.Ensures(Contract.Result<TAbstractFactory>() != null);
            return DoubleCheckedLock<TAbstractFactory>.Create(
                _targetCache, this, () => this.AsIf<TAbstractFactory>(true));
        }
    }

    // … S N I P …
}
```

The **RegisterOperation** methods accept a **string**, which defines the creational method name, or a **MethodInfo** (Microsoft, 2010j) as the type for its first argument. The second argument defines the actual creational method. The registered method must be linked to a corresponding method on the **TAbstractFactory** AbstractFactory interface. An exception is thrown if the method signature is not found on the AbstractFactory. When a string is used to identify the method (as opposed to using a **MethodInfo**), then method overloading (Meyer, 2001) is not allowed, because no argument information is supplied and the **RegisterOperation** thus does not know with what method on the AbstractFactory it must hook up with. The **MethodInfo** (Microsoft, 2010j) type, which is an internal .NET type, does allow method overloading on the AbstractFactory.

Only **Factory** APL delegates or **IFactory** APL interfaces can be registered with the **AutoAbstractFactory** component. Both the **Factory** delegates and the **IFactory** interfaces define methods that return a newly created instance. A number of **Factory** delegates exist in the APL library, each with a different set of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate TResult Factory<out TResult>();
public delegate TResult Factory<out TResult, in T>(T arg);
public delegate TResult Factory<out TResult, in T1, in T2>(T1 arg1, T2 arg2);
// … M O R E …
```

A number of **IFactory** interfaces also exist in the APL library also with a different set of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IFactory<out TResult> { TResult Create(); }
public interface IFactory<out TResult, in T> { TResult Create(T arg); }
public interface IFactory<out TResult, in T1, in T2> { TResult Create(T1 arg1, T2 arg2); }
// … M O R E …
```

The **Invoke** method on the **AutoAbstractFactory** queries the internal dictionary in order to see whether an operation was registered for the received method signature. The method signature is part of the **Invoke** method's argument list. If one exists, the operation is invoked and the newly created instance is returned. If a method is not found, then an exception is thrown. The validation if an implementation is registered against a method signature on the **TAbstractFactory** AbstractFactory interface is thus done only when the method is being invoked by a client.

The **`Target`** property on the **`AutoAbstractFactory`** returns an instance of a dynamically created class that has implementations for all the methods on the **`TAbstractFactory`** AbstractFactory interface. Every invocation on the instance is channelled to the **`Invoke`** method, which then calls the appropriate creational method.

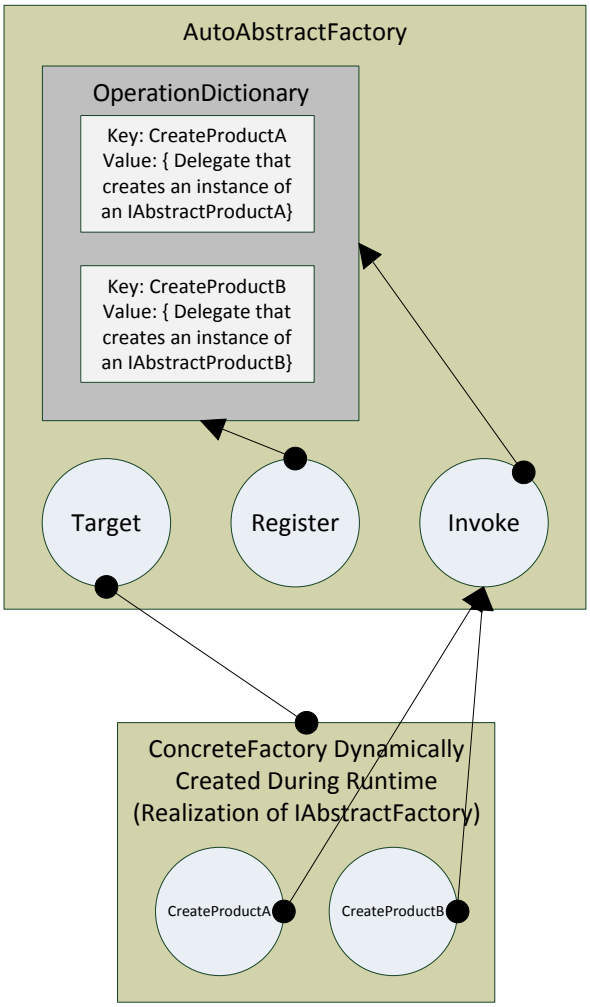

Figure 6. AutoAbstractFactory APL component overview.

Figure 6 shows a graphical overview of the **`AutoAbstractFactory`** component. It indicates the three main contracts of an **`AutoAbstractFactory`**. These are: first, the registration contracts used to register creational methods; secondly, the **`Target`** contract used to retrieve a dynamically created instance of a ConcreteFactory during runtime and; thirdly, the **`Invoke`** contract that is used by the *duck typing* (Koenig & Moo, 2005) runtime in order to invoke one of the delegates stored inside the dictionary. The dynamically created ConcreteFactory, which realizes an AbstractFactory **`IAbstractFactory`**, forwards all local invocations to the **`Invoke`** method on the **`AutoAbstractFactory`** instance, from where the call is forwarded to the correct method in the dictionary. For example, a call to the **`CreateProductA`**

method on the ConcreteFactory is forwarded to the **`Invoke`** method on the **`AutoAbstractFactory`**. From there, a delegate that represents the **`CreateProductA`** method, and thus a creator of an instance that realizes the **`IAbstractProductA`** interface, is retrieved from the dictionary and executed. The Product result is then passed to the ConcreteFactory, from where it is passed back to the caller.

The **`SimpleAutoAbstractFactory`** component does almost exactly the same as the **`AutoAbstractFactory`** component. However, an AbstractProduct type is registered on the **`SimpleAutoAbstractFactory`** component and not the creational method. The AbstractProduct type is registered together with its corresponding Product type, as can be seen in the **`Register<TAbstractProduct, TProduct>()`** method. The **`Register<TAbstractProduct, TProduct>()`** method adds a creational method for the AbstractProduct type in the component's internal dictionary. In order to eliminate ambiguities, only one creational method that returns a certain AbstractProduct type is allowed on the AbstractFactory interface when using the **`SimpleAutoAbstractFactory`**.

The code below shows the implementation of the **`SimpleAutoAbstractFactory`** component in the APL library:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class SimpleAutoAbstractFactory<TAbstractFactory> : IDynamicInvoker
    where TAbstractFactory : class {
    private readonly Dictionary<Type, Factory<object>> _factoryDictionary =
        new Dictionary<Type, Factory<object>>();
    private volatile TAbstractFactory _abstractFactoryCache;

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_factoryDictionary != null, "The FactoryDictionary cannot be null");
    }

    public SimpleAutoAbstractFactory() { _abstractFactoryCache = null; }

    public void Register<TAbstractProduct, TProduct>()
        where TConcreteFactory : class, TFactoryInterface, new() {
        // … C O N T R A C T S …

        _factoryDictionary.Add(typeof(TAbstractProduct), () => new TProduct());
    }

    public TAbstractFactory Target {
        get {
            Contract.Ensures(Contract.Result<TAbstractFactory>() != null);
            return DoubleCheckedLock<TAbstractFactory>.Create(
                   _abstractFactoryCache, this, () => this.AsIf<TAbstractFactory>(true));
        }
    }

    public object Invoke(string methodName, object[] args) {
        Contract.Requires<ArgumentException>(!string.IsNullOrEmpty(methodName),
                                             "Argument methodName cannot be null");

        // Go through all of the factory interfaces and find the method
        // with the argument contract.
        var factory = GetFactory(methodName, args);
        if(factory != null) return factory;
```

```
        throw new Exception("The factory was not found.");
    }

    // … S N I P …
}
```

The **`Invoke`** method of the **`SimpleAutoAbstractFactory`** component queries its internal dictionary for the ConcreteFactory which holds a method that creates, and thus returns, the specific Product. The query is performed using the AbstractProduct type as a key. If this method is found, then it invokes the creational method and returns the newly created Product. An exception is thrown if no method is found. An exception is also thrown if any ambiguity is found.

For example an AbstractFactory creating two AbstractProducts can be used as follows:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IAbstractFactory { // AbstractFactory interface
    IProductA CreateProductA(); // Creational method that creates an IProducA AbstractProduct
    IProductB CreateProductB(); // Creational method that creates an IProducB AbstractProduct
}

var factory = new SimpleAutoAbstractFactory<IAbstractFactory>(); // Create a ConcreteFactory
factory.Register<IProductA, ProductA>(); // Register a ProductA against an IProductA AbstractProduct
factory.Register<IProductB, ProductB>(); // Register a ProductB against an IProductB AbstractProduct
```

The Products **`ProductA`** and **`ProductB`** are registered against the AbstractProducts they realize. Both of the AbstractProducts in the above example are return types on creational methods defined on the **`IAbstractFactory`** interface. The **`factory`** instance can now be used with the **`Target`** property that returns an instance of a dynamically created ConcreteFactory, which implements the **`IAbstractFactory`** interface. The code snippet below returns an instance of the **`ProductA`** class that was registered with the **`factory`** instance:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var productA = factory.Target.CreateProductA();
```

Abstract factories can also be implemented using the prototype (Gamma, Helm, Johnson, & Vlissides, 1994) (Zimmer, 1995) design pattern. For this reason a **`PrototypeAbstractFactory`** component also exists in the APL library. This component behaves almost exactly as the **`SimpleAutoAbstractFactory`**, except that Prototype instances, not Products types, are registered against AbstractProducts.

The registration of creational operations against a certain method available on the Target interface can be improved by using C# dynamics or lambda expressions, as shown in Appendix I. The same mechanism can be used for all the components in this thesis that have to register a method that will be used in a *duck typing* (Koenig & Moo, 2005) environment.

4.3 Theoretical Examples

The following theoretical example shows the usage of the **`AutoAbstractFactory`** component defined in the previous section. It defines two AbstractProducts **`IAbstractProductA`** and **`IAbstractProductB`**:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IAbstractProductA { void Bar(); }                      // AbstractProduct
public interface IAbstractProductB { void Foo(IAbstractProductA a); }   // AbstractProduct
```

Figure 7 shows a sequence diagram for the **`AutoAbstractFactory`** example. The full example is shown after the sequence diagram. It illustrates the registration of an **`IAbstractProductA`** AbstractProduct and the subsequent creation of a Product using the **`Target`** property.

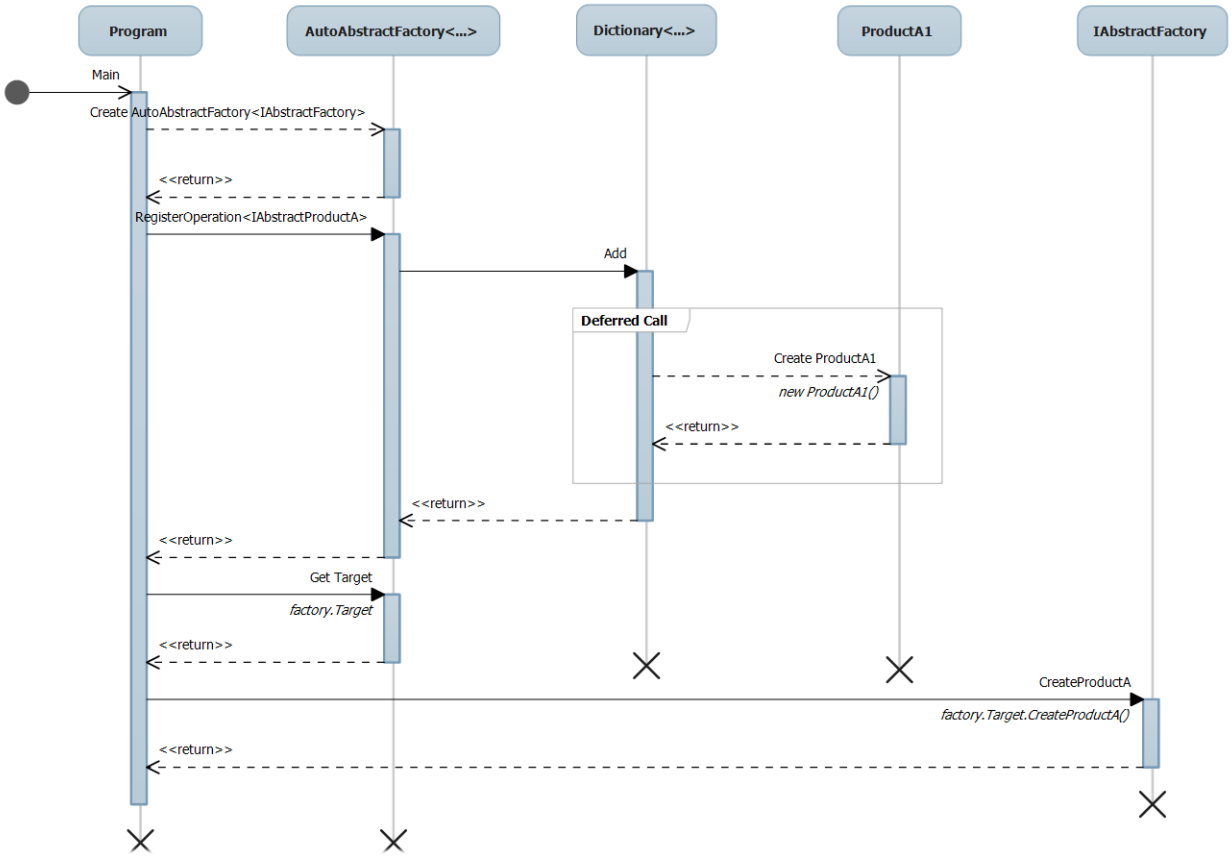

Figure 7. UML sequence diagram for the AutoAbstractFactory component example.

The example creates implementations for the AbstractProducts. Each AbstractProduct is given two implementations:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[Serializable]
public class ProductA1 : IAbstractProductA { // Product
    public void Bar() { Console.WriteLine("ProductA1: Called Bar"); }
}

[Serializable]
public class ProductB1 : IAbstractProductB { // Product
    public void Bar() { Console.WriteLine("ProductB1: Called Bar"); }
    public void Foo(IAbstractProductA a) {
        Console.WriteLine("ProductB1: Called Foo - uses " + a.GetType().Name);
    }
}

[Serializable]
public class ProductA2 : IAbstractProductA { // Product
    public void Bar() { Console.WriteLine("ProductA2: Called Bar"); }
}

[Serializable]
public class ProductB2 : IAbstractProductB { // Product
    public void Foo(IAbstractProductA a) {
        Console.WriteLine("ProductA2: Called Foo - uses " + a.GetType().Name);
    }
}
```

An AbstractFactory interface is then defined with two methods that return each AbstractProduct. No ConcreteFactories are defined, as they are automatically implemented by the abstract factory components:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IAbstractFactory { //AbstractFactory
    IAbstractProductA CreateProductA(); // Creational Method
    IAbstractProductB CreateProductB(); // Creational Method
}
```

The following code shows the usage of the **`AutoAbstractFactory`** component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var factory = new AutoAbstractFactory<IAbstractFactory>();
// Register a creational lambda expression representing the CreateProductA method on the AbstractFactory
factory.RegisterOperation<IAbstractProductA>("CreateProductA", () => new ProductA1());

// Register a creational lambda expression representing the CreateProductB method on the AbstractFactory
factory.RegisterOperation<IAbstractProductB>("CreateProductB", () => new ProductB1());

var productA = factory.Target.CreateProductA(); // Create a productA using the CreateProductA method
productA.Bar(); // Use the productA instance
var productB = factory.Target.CreateProductB(); // Create a ProductB using the CreateProductB method
productB.Foo(productA); // Use the productB instance

/* Output
ProductA1: Called Bar
ProductB1: Called Foo - uses ProductA1
*/
```

In the example above, a **factory** instance is created with the **AutoAbstractFactory** component with an **IAbstractFactory** AbstractFactory interface. Both of the methods on the AbstractFactory are then registered with the **factory** instance, using the **RegisterOperation** method. The creational method type is defined by its name **"CreateProductA"** and the creational logic is injected with a lambda expression (J'arvi, Freeman, & Crowl, 2007):

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
factory.RegisterOperation<IAbstractProductA>("CreateProductA", () => new ProductA1());
```

The **factory** instance is then used in a client environment. The **Target** property is used to acquire a dynamically created instance during runtime that realizes the **IAbstractFactory** AbstractFactory interface. The **AutoAbstractFactory** component thus creates a new class during runtime and returns an instance of it to the calling client. All of the invocations done through the **Target** property are forwarded to the **AutoAbstractFactory**, where the appropriate creational logic is invoked. In the example, two Products **productA** and **productB** are created using the **Target** property on the **factory** instance. The Products are also used in the example.

The output shows that both of the **factory** calls were successful.

4.4 Outcome

The componentization of the abstract factory design pattern is a success because it meets all the requirements listed in section 1.4:

- **Completeness**: The abstract factory design pattern library components cover all cases described for the original core abstract factory design pattern in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994).

- **Usefulness**: The abstract factory design pattern library components are useful because they solve all of the abstract factory's defined intent. With the **AutoAbstractFactory** component, a developer need only inject the creational logic with an instance of the abstract factory component. A different abstract factory component implementation, **PrototypeAbstractFactory**, exists where a Prototype is used for the creation of the Products, giving the user a different implementation choice. The **SimpleAutoAbstractFactory** component is useful when the creation of the Product can be done using the default constructor. The abstract factory design pattern library components are simple to understand and easy to use.

- **Faithfulness**: The implementation of the abstract factory pattern differs from the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). In *Design Patterns* the ConcreteFactory participant is manually coded. With the **AutoAbstractFactory**

reusable component, the ConcreteFactory is dynamically created using meta-programming (Perrotta, 2010). The `SimpleAutoAbstractFactory` component implementation is also slightly different where the default constructor is automatically used for Product creation. The functionality and original intent, however, of the abstract factory pattern, are satisfied for all the reusable abstract factory library components. All the abstract factory components in the APL library offer an instance that realizes the AbstractFactory interface.

- **Type-safety**: The `RegisterOperation` methods on the `AutoAbstractFactory` component use non type-safe string literals for the specification of the method names. Lambda expressions (expressions trees) (Albahari & Albahari, 2007, p. 317) however, can be used to solve the type-safe registration problem, as shown in Appendix I. Other than that, all the library components are fully type-safe.

- **Extended applicability**: The abstract factory library components do not cover more cases than the original abstract factory pattern.

- **Performance**: The abstract factory components do have a performance impact because of the usage of *duck typing* (Koenig & Moo, 2005). Appendix II shows the performance impact of *duck typing*. The performance impact is, however, acceptable in normal situations.

The abstract factory is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable abstract factory library components.

The following language features are fundamental to the implementation or usage of the reusable abstract factory design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010), Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005), Duck Typing (Koenig & Moo, 2005) and Meta-programming (Perrotta, 2010).

Chapter 5

5 FACTORY METHOD

5.1 Introduction

The factory method design pattern is one of the humblest creational patterns. The design pattern is also known as the virtual constructor (Gamma, Helm, Johnson, & Vlissides, 1994). The pattern defines an interface for creating a specific object. However, it allows subclasses to resolve which concrete class to create. The factory method design pattern thus allows a class to delegate object creation to its subclasses (Gamma, Helm, Johnson, & Vlissides, 1994).

The intent of the factory method pattern can be defined as:

- The ability to support polymorphic object creation.
- The ability to define a contract for instantiating objects and to let the instances of subclasses decide which concrete objects to create.

5.1.1 Structure.

The following figure shows the formal structure of the factory method design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

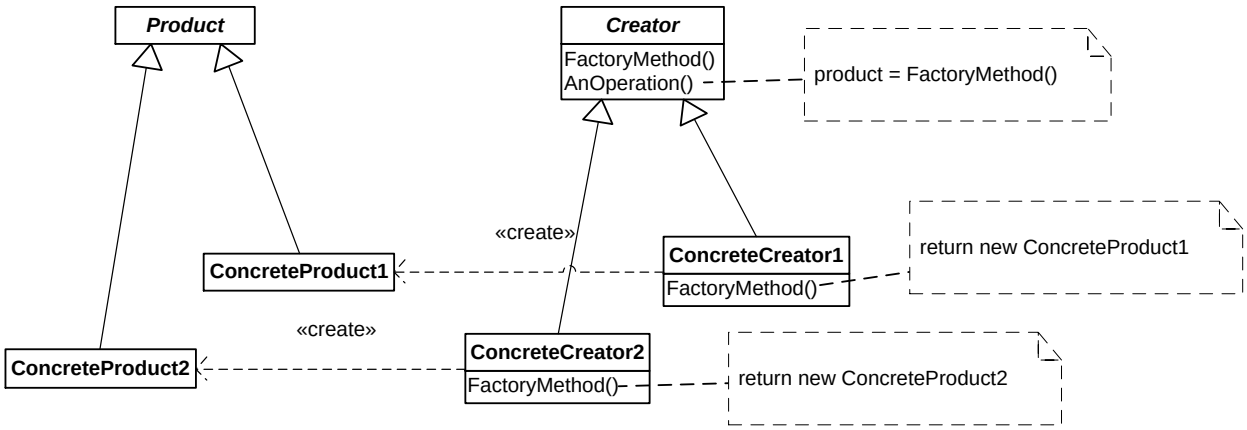

Figure 8. Factory method structure.

5.1.2 Participants.

The classes and/or objects participating in the factory method design pattern are:

- **Product**

  The Product defines the interface of the objects that the factory method creates.

- **ConcreteProduct**

  The ConcreteProduct implements the Product interface.

- **Creator**

  The Creator defines the virtual creational operation that returns an object of type Product. The Creator may also realize a standard implementation of the factory method that returns a standard ConcreteProduct object.

- **ConcreteCreator**

  The ConcreteCreator overrides the virtual factory operation in order to return an instance of a ConcreteProduct.

5.2 Library Components

5.2.1 The ActionCreator component.

The **`ActionCreator`** APL component utilises generics (Jagger, Perry, & Sestoft, 2007) in order to implement a reusable factory method pattern. The user must supply the **`ActionCreator`** component with the Product and ConcreteProduct generic arguments as seen below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
var concreteCreator1 = new ActionCreator<IProduct, ConcreteProduct>(x => x.Operation());
```

The **`ActionCreator`** component defines a specific implementation for the factory method pattern. The **`ActionCreator`** has a public constructor that takes in an **`Action`** C# delegate (Microsoft, 2010a). The **`Action`** delegate itself takes in the Product as an argument. The delegate, which is supplied by the client, thus defines what action must be performed with the Product. The **`ActionCreator`** has two public methods, **`Create`** and **`Execute`**. The **`Create`** method returns a new instance of the ConcreteProduct. The ConcreteProduct type, which is supplied as a generic argument, must have a

default constructor. This is because the **ActionCreator** component creates an instance of the ConcreteProduct using the **new** C# keyword on the ConcreteProduct generic type, as seen below:

```
public TProduct Create() { return new TConcreteProduct(); }
```

The **Execute** method defines a universal method that invokes a registered action using the newly created Product. This reusable pattern component thus generalises and componentizes one of the most common usages of the factory method pattern.

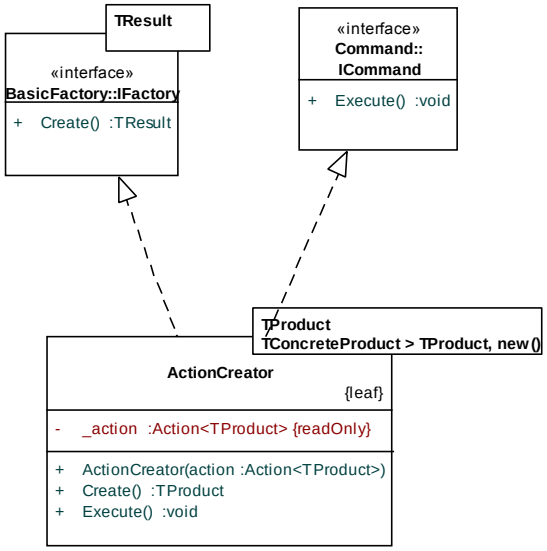

Figure 9. UML class diagram of the ActionCreator APL component.

Figure 9 shows a UML class diagram of the **ActionCreator**. The **ActionCreator** component implements the **IFactory<TProduct>** and **ICommand** APL interfaces that make the component more flexible and adaptable in other pattern scenarios. Multiple **ActionCreator** components exist in the APL library, where each one accommodates the different number of arguments possible for the **Execute** method. The **ActionCreator** can only be used for a specific factory method solution where a specific method performs a certain action on a newly created Product:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionCreator<TProduct, TConcreteProduct> : IFactory<TProduct>, ICommand
    where TConcreteProduct : TProduct, new() {
    private readonly Action<TProduct> _action;

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_action != null, "The action cannot be null");
    }

    public ActionCreator(Action<TProduct> action) {
```

```
        _action = action;
    }

    // A well known Create
    public TProduct Create() {
        return new TConcreteProduct();
    }

    // Execute which uses a Factory Method
    public void Execute() {
        _action(Create());
    }
}
```

The **`ActionFactoryCreator`** component is a special variant of the **`ActionCreator`** where the creation of the Product is entrusted to an implementation of the **`Factory<TProduct>`** delegate or **`IFactory<TProduct>`** interface. The **`Factory<TProduct>`** delegate and **`IFactory<TProduct>`** interface are part of the APL library.

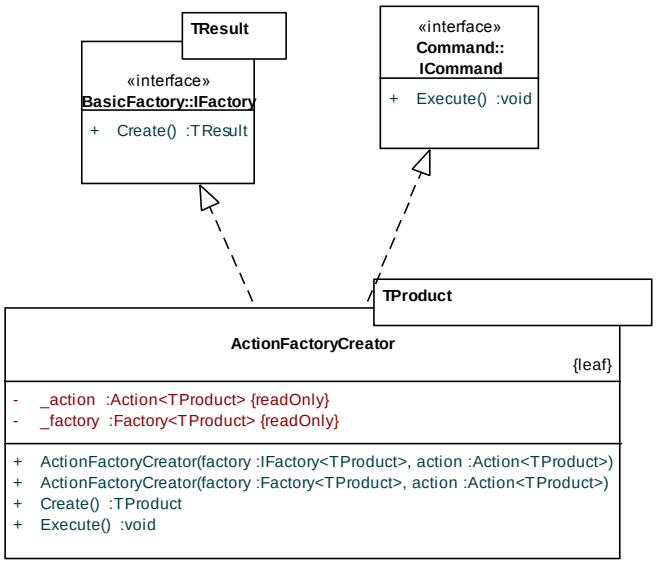

Figure 10. UML class diagram of the ActionFactoryCreator APL component.

Figure 10 shows a UML class diagram of the **`ActionFactoryCreator`**. It shows the following: first, the registration of the **`Factory`** and **`Action`** delegates in the component's constructor; secondly, the **`Create`** method that routes its invocation logic to the registered **`_factory`** delegate and thirdly, the **`Execute`** method that routes its invocation logic to the registered **`_action`** delegate. The **`IFactory<TProduct>`** interface is converted into a **`Factory<TProduct>`** delegate in the constructor, where the interface is used.

The code snippet on the next page shows the implementation of the **`ActionFactoryCreator`** component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionFactoryCreator<TProduct> : IFactory<TProduct>, ICommand {
    private readonly Factory<TProduct> _factory;
    private readonly Action<TProduct> _action;

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_factory != null, "The factory cannot be null");
        Contract.Invariant(_action != null, "The action cannot be null");
    }

    // Register the action and the factory through the constructor
    public ActionFactoryCreator(IFactory<TProduct> factory, Action<TProduct> action) {
        _factory = factory.Create;
        _action = action;
    }

    // Register the action and the factory through the constructor
    public ActionFactoryCreator(Factory<TProduct> factory, Action<TProduct> action) {
        _factory = factory;
        _action = action;
    }

    public TProduct Create() { // Route the Create invocation to _ factory
        Contract.Ensures(Contract.Result<TProduct>() != null);
        return _factory();
    }

    public void Execute() { _action(Create()); } // Route the Execute invocation to the _action delegate
}
```

In the above code, the **`Create`** method delegates the creation of the Product to the registered **`_factory`** delegate. This is slightly more adaptable than the original **`ActionCreator`** component, in which the generic Product is forced to have a default constructor.

There are also multiple **`ActionFactoryCreator`** components in the APL library, each one catering for the different number of possible arguments.

C# **`Action`** delegates (Microsoft, 2010a) define methods that take in a specific set of arguments and that do not return anything. The APL library also has **`FuncCreator`** components that use **`Func`** (Microsoft, 2010h) delegates rather than **`Action`** delegates:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class FuncCreator<TProduct, TConcreteProduct, TResult> : IFactory<TProduct>
    where TConcreteProduct : TProduct, new() {
    private readonly Func<TProduct, TResult> _func

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_func != null, "The function cannot be null");
    }

    public FuncCreator(Func<TProduct, TResult> func) {
        _func = func;
    }

    public TProduct Create() {
        Contract.Ensures(Contract.Result<TProduct>() != null);
```

```
        return new TConcreteProduct();
    }

    public TResult Execute() {
        return _func(Create());
    }
}
```

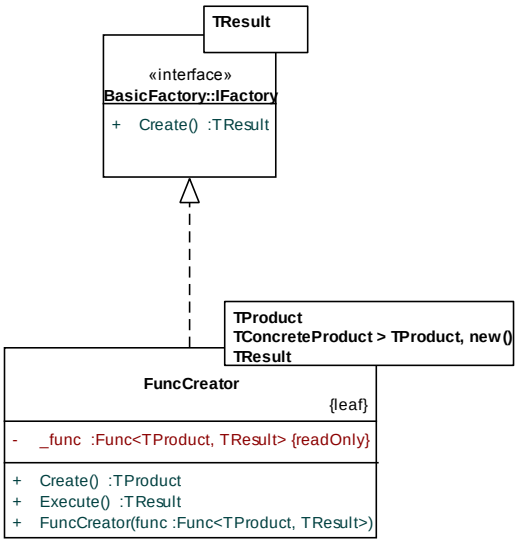

Figure 11. UML class diagram of the FuncCreator APL component.

Figure 11 shows a UML class diagram of the **`FuncCreator`** component. An **`Execute`** method is defined on the **`FuncCreator`** that returns a certain value. The **`Execute`** method is thus a function, because it has a return value. Note that the **`FuncCreator`** no longer implements the **`ICommand`** interface. This is because the **`Execute`** method on the **`ICommand`** interface does not return any value and is thus not a function.

A **`FuncFactoryCreator`** also exists in the APL library. The **`FuncFactoryCreator`** component is a special variant of the **`FuncCreator`** where the creation of the Product is entrusted to an implementation of the **`Factory<TProduct>`** delegate or an **`IFactory<TProduct>`** interface. The execute method of a **`FuncFactoryCreator`** is a function, and thus returns a value.

The code below shows the implementation of the **`FuncFactoryCreator`** APL component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class FuncFactoryCreator<TProduct, TResult> : IFactory<TProduct>
    private readonly Factory<TProduct> _factory;
    private readonly Func<TProduct, TResult> _func;

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_factory != null, "The factory cannot be null");
```

```
        Contract.Invariant(_func != null, "The action cannot be null");
    }

    // Register the action and the factory through the constructor
    public ActionFactoryCreator(IFactory<TProduct> factory, Func<TProduct> func) {
        _factory = factory.Create;
        _func = func;
    }

    // Register the action and the factory through the constructor
    public ActionFactoryCreator(Factory<TProduct> factory, Func<TProduct> func) {
        _factory = factory;
        _func = func;
    }

    public TProduct Create() { // Route the Create invocation to the _factory delegate
        Contract.Ensures(Contract.Result<TProduct>() != null);
        return _factory();
    }

    public TResult Execute() { return _func(Create()); } // Route the Execute invocation to the
                                                         // _func delegate
}
```

A number of **`FuncCreator`** and **`FuncFactoryCreator`** components are also present in the APL library, each one catering for a certain set of arguments.

The last **`ActionCreator`** variant in the APL library is the **`ActionPrototypeCreator`** component. This component serves the same function as the **`ActionCreator`**, except that a Product instance is registered with the component during its construction:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionPrototypeCreator<TProduct> : IFactory<TProduct>, ICommand {

    private readonly TProduct _product;
    private readonly Action<TProduct> _action;

    protected ActionPrototypeCreator(TProduct product, Action<TProduct> action) {
        _product = product;
        _action = action;
    }

    public void Execute() {
        action(Create());
    }

    // Use the Prototype component in order to return a clone of the _product instance
    public TProduct Create() {
        Contract.Ensures(Contract.Result<TProduct>() != null);

        return _product.DeepCopy();
    }
}
```

The **`ActionPrototypeCreator`** component uses a Product instance in order to clone it in the **`Create`** method, instead of using the Product's default constructor or an injected factory delegate. It thus implements an *extension* (Dyson & Anderson, 1997) of the factory method where the prototype pattern

(Gamma, Helm, Johnson, & Vlissides, 1994) is used. The cloning mechanism uses the prototype APL component.

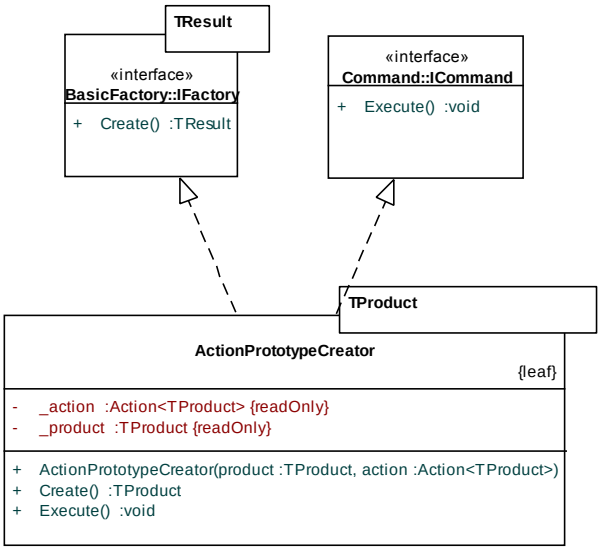

Figure 12. UML class diagram of the ActionPrototypeCreator APL component.

Figure 12 shows a UML class diagram of the **`ActionPrototypeCreator`** APL component. It shows that the component's constructor takes in a Product, which is cloned in the **`Create`** method. It also shows the component's implementation of the **`IFactory`** and **`ICommand`** APL interfaces.

Once again, multiple **`ActionPrototypeCreator`** and **`FuncPrototypeCreator`** components are defined in the APL library, depending on the number of desired arguments.

5.3 Theoretical Examples

The following example shows the usage of the **`ActionCreator`**, **`ActionPrototypeCreator`** and **`ActionFactoryCreator`** components. The code below shows the definitions of the **`ConcreteProduct1`** and the **`ConcreteProduct2`** classes, both of which implement the **`IProduct`** interface and are serializable:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IProduct { void Operation(); }

[Serializable]
class ConcreteProduct1 : IProduct {
    public void Operation() { Console.WriteLine("Calling operation on ConcreteProduct1 …"); }
}

[Serializable]
```

```
class ConcreteProduct2 : IProduct {
    public void Operation() { Console.WriteLine("Calling operation on ConcreteProduct2 …"); }
}

public static class OperationHelper {
    public static void AnOperation(IProduct product) { product.Operation(); }
}
```

In the above example code, some action logic resides in the **`OperationHelper`** that is registered with the **`ActionCreator`** components.

The **`concreteCreatorA`** instance, shown in the example code below, is created using the **`ActionCreator`** component together with the **`AnOperation`** method action logic that is defined on the **`OperationHelper`** static class. The **`concreteCreatorA`** instance is also created with the **`IProduct`** and **`ConcreteProduct1`** generic arguments, notifying the component of its Product and ConcreteProduct types. The **`concreteCreatorB`** instance is created using the **`ActionPrototypeCreator`** component, also with the **`AnOperation`** method action logic that is defined on the **`OperationHelper`** static class. The **`concreteCreatorB`** instance is also created with a **`ConcreteProduct2`** instance that the component will use as a Prototype. An instance of the **`ActionFactoryCreator`** component is created where both the creational logic and action logic are injected using lambda expressions (Samko, et al., 2006):

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
class FactoryMethodExample {
    static void Main() {
        var concreteCreatorA = new ActionCreator<IProduct, ConcreteProduct1>(OperationHelper.AnOperation);
        concreteCreatorA.Execute();

        var concreteCreatorB = new ActionPrototypeCreator<IProduct>(new ConcreteProduct2(),
                                                                    OperationHelper.AnOperation);
        concreteCreatorB.Execute();

        var concreteCreatorC = new ActionFactoryCreator<IProduct>(() => new ConcreteProduct2(),
                                                                  x => x.Operation() + "[More]");
        concreteCreatorC.Execute();


        Console.WriteLine("Press Enter to exit.");
        Console.ReadLine();
    }
}

/* Output
Calling operation on ConcreteProduct1 …
Calling operation on ConcreteProduct2 …
Calling operation on ConcreteProduct2 …[More]
*/
```

In the example code above, the ConcreteCreators are created and their specific **`Execute`** methods are invoked, thus executing the desired action on each created Product. It can be seen in the output that the **`Operation`** method on the correct ConcreteProduct is called successfully by all the ConcreteCreators.

5.4 Outcome

The componentization of the factory method design pattern is a partial success because it meets some of the requirements listed in section 1.4:

- **Completeness**: The factory method design pattern library components cover all cases described in the original core design pattern.

- **Usefulness**: A factory method design pattern implementation is largely structural and cannot be successfully componentized. With the `ActionCreator` group of components, a fully functional factory method can be implemented, which thus makes the component reusable. However, its usefulness is debatable, as there might be scenarios where a developer wishes to add multiple abstract factories to the same class. Furthermore, implementing a fully functional abstract factory by hand is a simple task and the reusable component might be an overhead in certain scenarios. Also, a factory method usually blends into an existing class in a system design, and is not a standalone element. For these three reasons, the componentization of the factory method design pattern can be regarded as only partially successful. Nevertheless, there are certain scenarios in which the `ActionCreator` is functionally adequate and can be regarded as useful. An instance of an `ActionCreator` realizes the `ICommand` pattern and can be used by the command patterns described later in this thesis.

- **Faithfulness**: Some elements of the implementation of the factory method pattern follow the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). The reusable `ActionFactoryCreator` component follows the original core pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994), except for the creational method as a constant name, namely `Execute`. The `ActionCreator`, however, is slightly different to the implementation mentioned in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) where the default constructor of a specific ConcreteProduct type is used for the ConcreteProduct creation. The `ActionPrototypeCreator` component, which uses a Prototype for ConcreteProduct creation, is mentioned in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). Using a Prototype for ConcreteProduct creation, however, does not form part of the core factory method pattern (Gamma, Helm, Johnson, & Vlissides, 1994).

- **Type-safety**: All of the library components are fully type-safe.

- **Extended applicability**: The factory method library components cover more cases than the original core factory method pattern in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). The `ActionPrototypeCreator` and `FuncPrototypeCreator` components use a Prototype in order to create the ConcreteProduct. The Prototype usage, as a variant implementation of the

factory method, is mentioned in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994), however not as the core implementation.

- **Performance**: The factory method components do not have a performance impact.

The following language features are fundamental to the implementation or usage of the reusable factory method design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Attributes (Nagel, Evjen, Glynn, & Watson, 2010), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010) and Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005).

Chapter 6

6 FLYWEIGHT

6.1 Introduction

The flyweight pattern is suitable wherever there is the possibility of a large number of instances of the same class, with some partial common state, of which the non-common part can be evaluated with arguments. The flyweight design pattern is thus used where a large number of fine grained objects are shared for maximum efficiency (Gamma, Helm, Johnson, & Vlissides, 1994).

6.1.1 Structure.

The following figure shows the formal structure of the flyweight design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

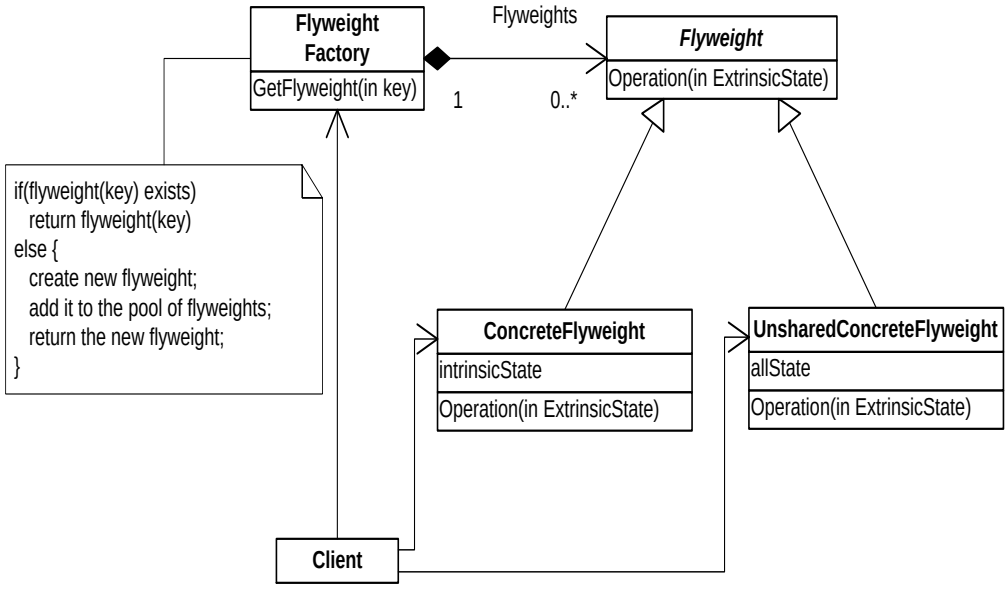

Figure 13. Flyweight structure.

6.1.2 Participants.

The classes and/or objects participating in the flyweight design pattern are:

- **Flyweight**

  A Flyweight defines an interface that flyweight objects can use in order to process messages with extrinsic state.

- **ConcreteFlyweight**

  The ConcreteFlyweight implements the operations of the Flyweight interface. It also stores the intrinsic state if it exists. The stored state must be intrinsic, which means that the state must not influence the ConcreteFlyweight object's functional context. All ConcreteFlyweight instances must be sharable.

- **UnsharedConcreteFlyweight**

  The Flyweight interface does not enforce sharing. A Flyweight subclass thus does not need to be shared. UnsharedConcreteFlyweight instances are usually concrete and hold a state that influences the object's functional context. An UnsharedConcreteFlyweight can have a child ConcreteFlyweight as a subclass.

- **FlyweightFactory**

  This is the class that instantiates, controls and manages flyweight objects. It enforces the sharing of flyweight objects through a common acquisition operation. On the demand of a client or user, the FlyweightFactory returns an existing flyweight or creates a new one if none exists. The FlyweightFactory thus returns an existing Flyweight, or creates a new one if none exists, on demand.

- **Client**

  The Client holds a reference to the Flyweights that were acquired by the FlyweightFactory. It also regulates and probably manages and stores, or has some control over, the extrinsic state of Flyweights.

6.2 Library Components

6.2.1 The FlyweightFactory component.

The `FlyweightFactory` APL component lies at the heart of the reusable flyweight pattern implementation. The component is defined as a Singleton (Gamma, Helm, Johnson, & Vlissides,

1994) that holds an internal Flyweight cache (Drepper, 2007). The code below shows the implementation of the **`FlyweightFactory`** APL component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class FlyweightFactory<TKey, TConcreteFlyweight> :
        Singleton<FlyweightFactory<TKey, TConcreteFlyweight>> { // The FlyweightFactory is a Singleton
    private IFlyweightCache<TKey, TConcreteFlyweight> _cache; // Internal Flyweight cache

    // … S N I P …

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_cache != null, "The cache cannot be null");
    }

    // Constructor is private because the FlyweightFactory is a Singleton
    private FlyweightFactory(DictionaryType type) { CreateCache(type); }

    // Constructor is private because the FlyweightFactory is a Singleton
    private FlyweightFactory() : this(DictionaryType.BinaryTree) { }

    public TConcreteFlyweight GetFlyweight(TKey key) {
        Contract.Requires<ArgumentNullException>(key!= null, "Argument key cannot be null");
        Contract.Ensures(Contract.Result<TConcreteFlyweight>() != null);
        TConcreteFlyweight flyweight;

        lock(this) {
            if(!GetFlyweight(key, out flyweight)) { // Get a Flyweight object for the given key
                Construct(key, out flyweight); // If the Flyweight does not exist create it…
                _cache.Add(key, flyweight); // … and add it into the internal cache
            }
        }

        return flyweight;
    }

    public TConcreteFlyweight this[TKey key] { get { return GetFlyweight(key); } }
    public int Count { get { return _cache.Count; } } // Get the number of Flyweight objects in the cache

    protected virtual void Construct(TKey key, out TConcreteFlyweight flyweight) {
        Contract.Requires<ArgumentNullException>(key!= null, "Argument key cannot be null");
        Contract.Ensures(flyweight != null);

        var args = new object[1];
        args[0] = key;

        flyweight = CreateHelper<TConcreteFlyweight>.CreateFromPrivateConstructor(args);
    }

    private void CreateCache(DictionaryType type) {
        Contract.Ensures(cache != null);

        var factory = new FlyweightCacheFactory<TKey, TConcreteFlyweight>();
        _cache = factory.Create(type);
    }

    private IFlyweightCache<TKey, TConcreteFlyweight> GetCache() { return _cache; }

    private bool GetFlyweight(TKey key, out TConcreteFlyweight flyweight) {
        Contract.Ensures(flyweight != null);

        return _cache.TryGetValue(key, out flyweight);
    }
}
```

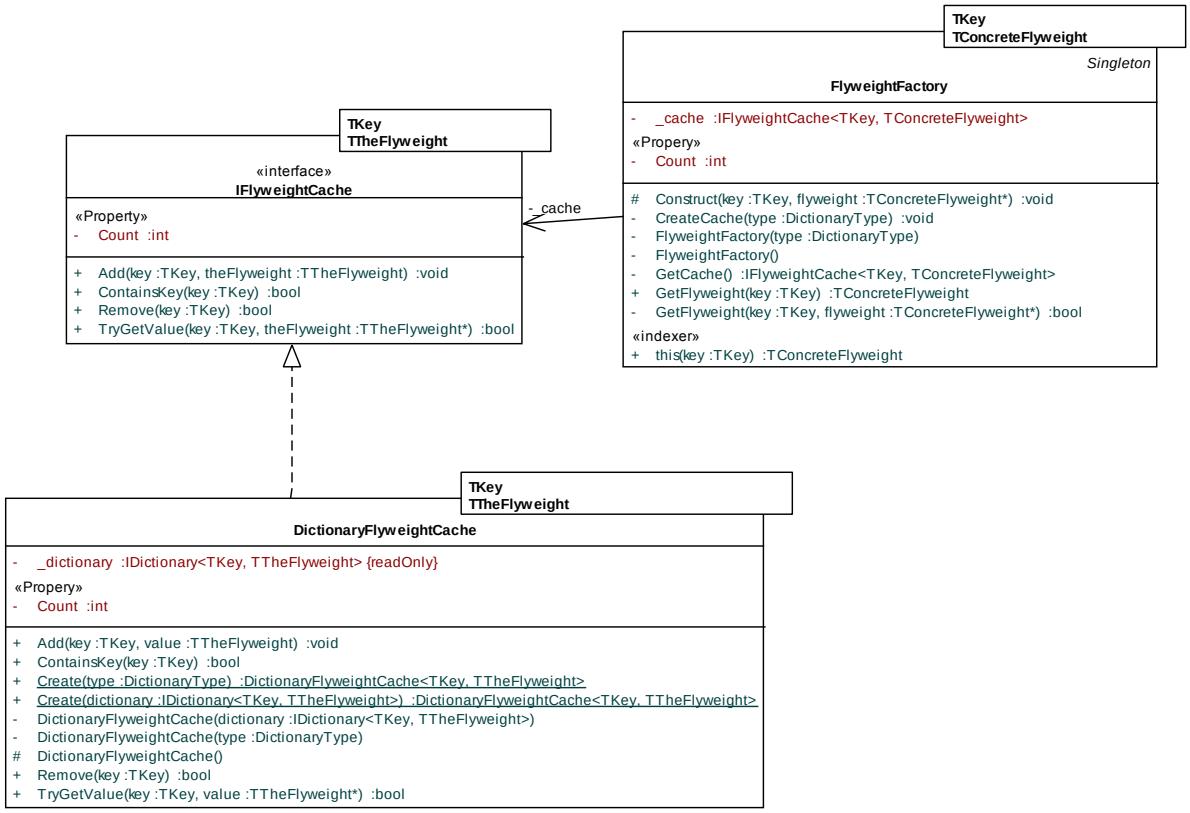

Figure 14. UML class diagram of the FlyweightFactory APL component.

Figure 14 shows a UML class diagram of the **`FlyweightFactory`**. This diagram shows how the **`FlyweightFactory`** is associated with the **`IFlyweightCache`** interface that is used to register and retrieve ConcreteFlyweight instances according to a certain key. The diagram also shows the **`DictionaryFlyweightCache`** that is a realization of the **`IFlyweightCache`** interface.

The reusable APL **`Singleton`** component is used to enforce the singleton nature of the **`FlyweightFactory`** component. The **`FlyweightFactory`** component takes two generic arguments **`TKey`** and **`TConcreteFlyweight`**. **`TKey`** defines the key type of the Flyweight where **`TConcreteFlyweight`** defines the actual instance of the Flyweight. The **`FlyweightFactory`** thus creates only one specific ConcreteFlyweight type, which is defined by the **`TConcreteFlyweight`** generic argument. The **`TKey`** generic argument defines the key type that is used to determine what specific ConcreteFlyweight instance must be returned.

The **`IFlyweightCache`** interface defines the contract of the internal flyweight cache. The actual implementation of the **`IFlyweightCache`** can be any desired data structure (Knuth, 1968) (Wirth, 1976),

such as a dictionary (Weiss, 1999), or any associative array with fast lookups to avoid performance implications.

The code below shows the implementation of the **`DictionaryFlyweightCache`** component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
class DictionaryFlyweightCache<TKey, TTheFlyweight> : IFlyweightCache<TKey, TTheFlyweight> {
    private readonly IDictionary<TKey, TTheFlyweight> _dictionary;

    public DictionaryFlyweightCache(IDictionary<TKey, TTheFlyweight> dictionary) {
        _dictionary = dictionary;
    }

    // … S N I P …
    public void Add(TKey key, TTheFlyweight value) { _dictionary.Add(key, value); }
    public bool ContainsKey(TKey key) { return _dictionary.ContainsKey(key); }

    public bool TryGetValue(TKey key, out TTheFlyweight value) {
        return _dictionary.TryGetValue(key, out value);
    }

    // … S N I P …
}
```

In the above code snippet, the **`_dictionary`** variable itself can be a standard C# runtime .NET **`SortedDictionary`** (Microsoft, 2010l) or a **`Dictionary`** (Microsoft, 2010f). In .NET a **`SortedDictionary`** is a red black binary tree (Leiserson, Rivest, & Stein, 2001) and a **`Dictionary`** is a hash table (Tenenbaum, Langsam, & Augenstein, 1990). The flyweight cache can hold any desired data structure, as long as it adheres to the **`IFlyweightCache`** contract.

The **`GetFlyweight`** public method or C# **`[]`** operator defined on the **`FlyweightFactory`** component is used to return a specific ConcreteFlyweight instance by supplying it with the **`key`**:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public TConcreteFlyweight this[TKey key] { get { return GetFlyweight(key); } }

public TConcreteFlyweight GetFlyweight(TKey key) {
    Contract.Requires<ArgumentNullException>(key!= null, "Argument key cannot be null");
    Contract.Ensures(Contract.Result<TConcreteFlyweight>() != null);
    TConcreteFlyweight flyweight;

    // Use double checked locking pattern
    if (!GetFlyweight(key, out flyweight)) {
        lock (GetCache()) {
            if (!GetFlyweight(key, out flyweight)) { // Get a Flyweight object for the given key
                Construct(key, out flyweight); // If the Flyweight does not exist create it…
                _cache.Add(key, flyweight); // …and add it to the internal cache
            }
        }
    }

    return flyweight;
}
```

The acquisition of the ConcreteFlyweight first checks whether the ConcreteFlyweight exists in the local cache. If the ConcreteFlyweight object does not exist, then a new ConcreteFlyweight is created. The key is passed to the ConcreteFlyweight's constructor, where it can be used in the construction logic. The newly created ConcreteFlyweight object is then added into the local cache. The **`FlyweightFactory`** component also has some value added public methods, such as **`Count`**, which returns the number of ConcreteFlyweight objects in the cache:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public int Count { get { return _cache.Count; } }
```

6.3 Theoretical Examples

The following theoretical example shows the usage of the FlyweightFactory component in the APL library:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
interface IFlyweight { void Operation(); }

class ConcreteFlyweight : IFlyweight {
    private readonly int _state;

    // The Key is an 'int' thus a private constructor that takes one argument of type 'int' must exist
    private ConcreteFlyweight(int state) { _state = state; }
    public override void Operation() { Console.WriteLine("ConcreteFlyweight: " + _state); }
}

class UnsharedConcreteFlyweight : IFlyweight {
    private readonly int _state;

    public UnsharedConcreteFlyweight(int state) { _state = state; }
    public override void Operation() { Console.WriteLine("UnsharedConcreteFlyweight: " + _state); }
}

class Program {
    static void Main() {
        // Creat an instance of a FlyweightFactory for a 'ConcreteFlyweight' with an 'int' key
        var factory = FlyweightFactory<int, ConcreteFlyweight>.Instance;

        Flyweight f1 = factory[1973]; // Get the Flyweight for instance for '1973'
        f1.Operation(); // Use the Flyweight

        Flyweight f2 = factory[1973]; // Get the Flyweight for instance for '1973'
        f2.Operation(); // Use the Flyweight

        // Check if the instances are the same
        if(f1 == f2) { Console.WriteLine("Objects are the same instance"); }

        Flyweight f3 = factory[2006]; // Get the Flyweight for instance for '2006'
        f3.Operation(); // Use the Flyweight

        var f4 = new UnsharedConcreteFlyweight(2009); // Create a UnsharedConcreteFlyweight
        f4.Operation();
    }
}

/*
ConcreteFlyweight: 1973
ConcreteFlyweight: 1973
```

```
Objects are the same instance
ConcreteFlyweight: 2006
UnsharedConcreteFlyweight: 2004
*/
```

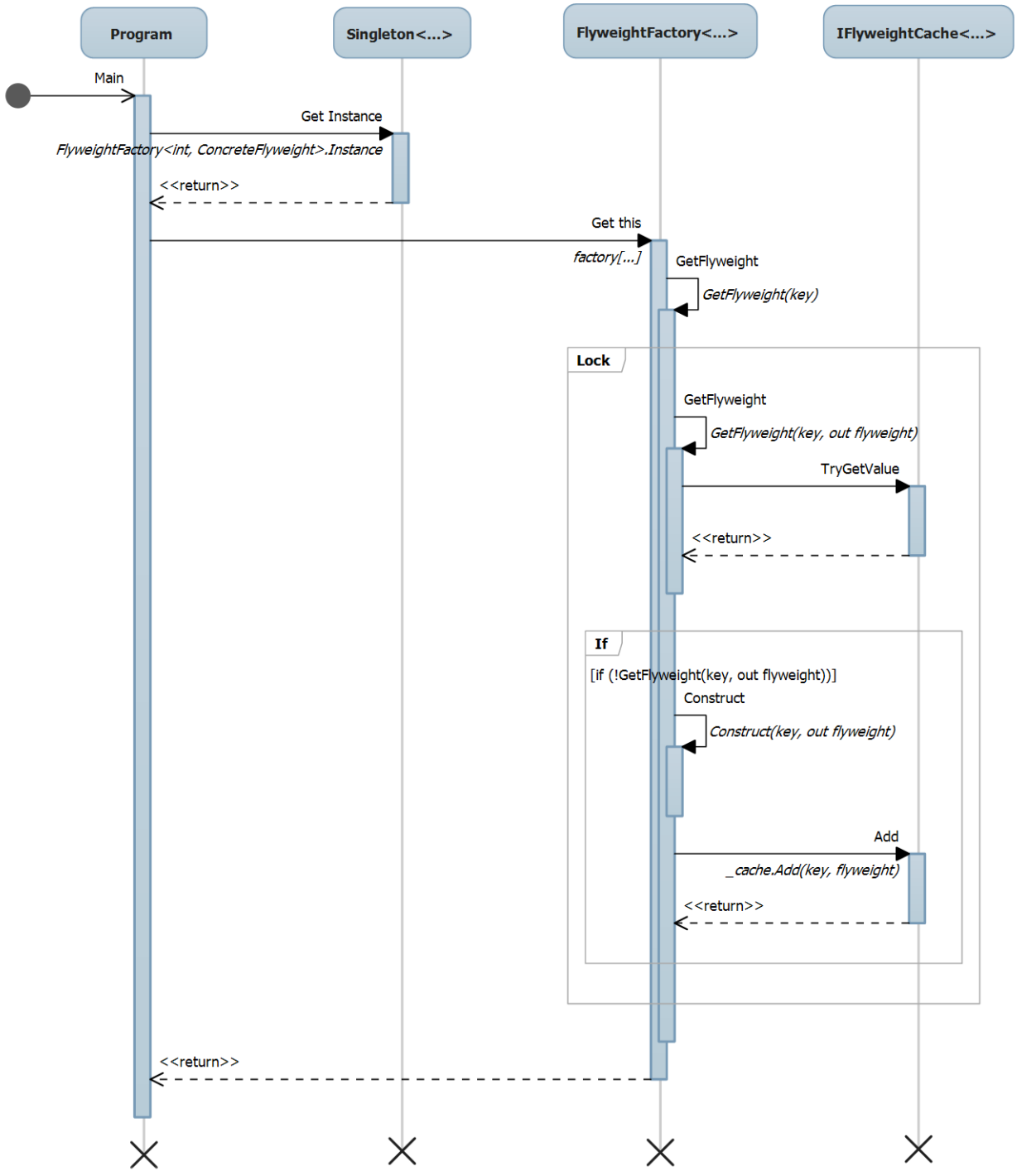

Figure 15. UML sequence diagram for the FlyweightFactory APL component example.

Figure 15 shows a UML sequence diagram for the flyweight theoretical example discussed in this section. It shows how the **`IFlyweightCache<TKey, TTheFlyweight>`** interface is used to register and retrieve ConcreteFlyweights instances via an instance of a **`FlyweightFactory`**.

The **`IFlyweight`** interface, shown in the example, defines the desired Flyweight contract. A ConcreteFlyweight **`ConcreteFlyweight`** is defined that implements the **`IFlyweight`** interface. A private constructor is defined on the ConcreteFlyweight **`ConcreteFlyweight`** that takes in the key as an argument.

In the example, the **`ConcreteFlyweight`** ConcreteFlyweight holds intrinsic state where the state is computed using the key received from the private constructor. A ConcreteFlyweight will not always hold intrinsic state. In the case where a ConcreteFlyweight does hold intrinsic state, then the state must be computed from the given key. There is always thus a direct coupling between the given key and a ConcreteFlyweight's intrinsic state. In the example the ConcreteFlyweights are created and managed with the reusable **`FlyweightFactory`** APL component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var flyweight1 = flyweightFactory[1973]; // Get the Flyweight for instance for '1973'
flyweight1.Operation(); // Use the Flyweight

var flyweight2 = flyweightFactory[1973]; // Get the Flyweight for instance for '1973'
flyweight2.Operation(); // Use the Flyweight
```

The **`FlyweightFactory`** is also a reusable generic singleton (Gamma, Helm, Johnson, & Vlissides, 1994) component. The first generic argument defines the key and the second argument defines the ConcreteFlyweight. A reference to the Singleton is acquired by supplying all the generic arguments and using the **`Instance`** property:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
// Creat an instance of a FlyweightFactory for a 'ConcreteFlyweight' with an 'int' key
var flyweightFactory = FlyweightFactory<int, ConcreteFlyweight>.Instance;
```

The **`flyweightFactory`** can then be used to acquire a desired Flyweight by passing it a specific key:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var flyweight1 = flyweightFactory[1973]; // Get the Flyweight for instance for '1973'
```

The example shows that the **`flyweight1`** and **`flyweight2`** Flyweights returned by the **`flyweightFactory`** are the same object instance and thus the **`FlyweightFactory`** component is working correctly.

An **`UnsharedConcreteFlyweight`** UnsharedConcreteFlyweight, that is not shared and thus not used by the flyweight pattern, is also defined in the example. The Flyweight interface does not enforce sharing, which is thus optional. A Flyweight subclass, therefore, does not need to be shared. UnsharedConcreteFlyweight instances are concrete Flyweights and hold a state that influences the object's functional context. It is possible that an UnsharedConcreteFlyweight might have child ConcreteFlyweights as subclasses.

6.4 Outcome

The componentization of the flyweight design pattern is a success, because it meets all the requirements listed in section 1.4:

- **Completeness**: The flyweight design pattern library component covers all cases described in the original core design pattern.

- **Usefulness**: The flyweight design pattern library component is useful, because it solves all of the flyweight scenarios desired by a developer and implement the pattern's defined intent. The flyweight library component is simple to understand and easy to use.

- **Faithfulness**: The implementation of the flyweight pattern follows the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994).

- **Type-safety**: All of the library components are fully type-safe.

- **Extended applicability**: The flyweight library component does not cover more cases than the original flyweight pattern.

- **Performance**: The flyweight component does not have a performance impact.

The flyweight pattern is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable flyweight components.

The following language features are fundamental in the implementation or usage of the reusable flyweight design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Attributes (Nagel, Evjen, Glynn, & Watson, 2010) and Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005).

Chapter 7

7 ADAPTER

7.1 Introduction

An interface is normally used to decouple the client from the implementation. It can happen that different interfaces exist for the same underlying functionality, usually in different frameworks. The adapter design pattern converts the contract and message flows from one interface to another.

The intent is thus to convert the interface of a class to an interface that clients expect. The adapter pattern makes it possible for classes to communicate with each other where it would otherwise not have been possible (Gamma, Helm, Johnson, & Vlissides, 1994).

7.1.1 Structure.

The following figure shows the formal structure of the adapter design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

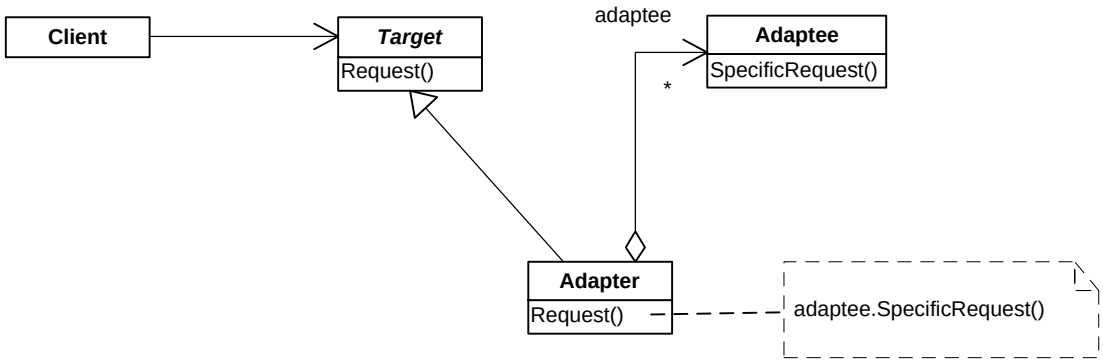

Figure 16. Adapter structure.

7.1.2 Participants.

The classes and/or objects participating in the adapter design pattern are:

- **Target**

  The Target declares the interface that the Client uses.

- **Adapter**

  The Adapter converts or adapts the interface of the Adaptee to the Target interface.

- **Adaptee**

  The Adaptee declares a current interface that needs adapting in order to be useful for the Client.

- **Client**

  The Client can use only objects implementing the Target interface.

7.2 Library Components

7.2.1 The AutoAdapter component.

The **AutoAdapter** APL component adapts registered **AdapterAction** and **AdapterFunc** delegates to methods available on a Target. It does so by dynamically routing a method invocation on an instance of the component to the appropriate method stored inside its internal dictionary (Weiss, 1999). The dictionary stores delegates associated with a certain method on the Target. The method behaviour is registered using the **RegisterAction** and **RegisterFunc** methods defined on the **AutoAdapter** component, as seen below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class AutoAdapter<TTarget, TAdaptee> : IDynamicInvoker // The IDynamicInvoker interface forces
                                                              // the implementation of the duck typing
                                                              // Invoke method
    where TTarget : class {
    private readonly Dictionary<DynamicMethod, Delegate> _operationDictionary =
        new Dictionary<DynamicMethod, Delegate>(); // Internal operation dictionary

    private TAdaptee _adaptee; // The adaptee instance
    private volatile TTarget _target; // Internal target cache

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_operationDictionary != null, "The operationDictionary cannot be null");
        Contract.Invariant(_adaptee != null, "The adaptee cannot be null");
    }

    // Constructor
    public AutoAdapter(TAdaptee adaptee) {
        _target = null;
        _adaptee = adaptee;
    }

    // Register an AdapterAction with no arguments
    public void RegisterAction(MethodInfo method, AdapterAction<TAdaptee> operation) { … }

    // Register an AdapterAction with no arguments
```

```
    public void RegisterAction(string methodName, AdapterAction<TAdaptee> operation) { … }

    // Register an AdapterAction with one argument
    public void RegisterAction(MethodInfo method, AdapterAction<TAdaptee, TArg1> operation) { … }

    // Register an AdapterAction with one argument
    public void RegisterAction<TArg1>(string methodName, AdapterAction<TAdaptee, TArg1> operation) { … }

    // … M O R E …

    // Register a AdapterFunc with no arguments
    public void RegisterFunc<TResult, TArg1>(MethodInfo method,
                                             AdapterFunc<TAdaptee, TArg1, TResult> operation) { … }

    // Register a AdapterFunc with no arguments
    public void RegisterFunc<TResult>(string methodName,
                                                  AdapterFunc<TAdaptee, TResult> operation) { … }

    // Register a AdapterFunc with one argument
    public void RegisterFunc<TResult, TArg1>(MethodInfo method,
                                             AdapterFunc<TAdaptee, TArg1, TResult> operation) { … }

    // Register a AdapterFunc with one argument
    public void RegisterFunc<TResult, TArg1>(string methodName,
                                             AdapterFunc<TAdaptee, TArg1, TResult> operation) { … }

    // … M O R E …

    // The following method, which is required by the IDynamicInvoker interface, maps the recieved
    // method signature to a delegate stored in the internal dictionary.
    // If a delegate is found with the same method signature then it is invoked and its result
    // is returned.
    public object Invoke(string methodName, object[] args) {
        Contract.Requires<ArgumentException>(!string.IsNullOrEmpty(path),
                                             "Argument methodName cannot be null");

        // Get a delegate from the internal dictionary with a matching method signature
        var operation = GetAdapterOperation(methodName, args);

        // Invoke the delegate and return its result
        if (operation != null)
            return operation.DynamicInvoke(_adaptee, args);

        throw new Exception("No adapter method found");
    }

    // Dynamicly create an instance during runtime that realizes the TTarget interface and return it
    // to the calling Client
    public TTarget Target {
        get {
            Contract.Ensures(Contract.Result<TTarget>() != null);

            return DoubleCheckedLock<TTarget>.Create(_target, this, () => this.AsIf<TTarget>(true));
        }
    }
}
```

The **`RegisterAction`** set of methods registers a specific **`AdapterAction`** against a certain method available on the Target. The method name must be passed through as a **`string`** or a C# reflection **`MethodInfo`** type, as seen in the example code below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
adapter.RegisterAction<string>("Request", (x, y) => x.TheRequest(); Console.WriteLine(y));
```

A number of **RegisterAction** methods are defined on the component, each specifying a specific number of arguments. A number of **AdapterAction** delegates also exist in the library, where each relates to the number of arguments needed:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate void AdapterAction<in TAdaptee>(TAdaptee adaptee);
public delegate void AdapterAction<in TAdaptee, in T>(TAdaptee adaptee, T arg);
public delegate void AdapterAction<in TAdaptee, in T1, in T2>(TAdaptee adaptee, T1 arg1, T2 arg2);
public delegate void AdapterAction<in TAdaptee, in T1, in T2, in T3>(TAdaptee adaptee,
                                                                     T1 arg1, T2 arg2, T3 arg3);

// … M O R E …
```

The first argument to all of the above **AdapterAction** delegates defines the Adaptee that is registered with the **AutoAdapter** component. The user thus has access to the registered Adaptee instance when formulating the adapter logic for a specific method. In the example above, the **x** variable in the lambda expression (KJärvi & Freeman, 2008) denotes the Adaptee instance and the **y** variable denotes the only argument available on the **Request** method. The injected adapter lambda expression **(x, y) => x.TheRequest(); Console.WriteLine(y)** thus first calls the **TheRequest** method on the Adaptee and then writes the contents of the **y** argument to the console.

The **RegisterFunc** set of methods does exactly the same as the **RegisterAction** methods, except that it registers **AdapterFunc** delegates. An **AdapterFunc** defines a return value and is thus used to adapt functions. A number of **RegisterFunc** methods are defined on the component, each specifying a specific number of arguments. A number of **RegisterFunc** delegates also exist in the library, where each relates to the number of arguments needed:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate TResult AdapterFunc<in TAdaptee, out TResult>(TAdaptee adaptee);
public delegate TResult AdapterFunc<in TAdaptee, in T, out TResult>(TAdaptee adaptee, T arg);
public delegate TResult AdapterFunc<in TAdaptee, in T1, in T2, out TResult>(TAdaptee adaptee, T1 arg1, T2
arg2);
public delegate TResult AdapterFunc<in TAdaptee, in T1, in T2, in T3, out TResult>(TAdaptee adaptee, T1
arg1, T2 arg2, T3 arg3);

// … M O R E …
```

The **Target** property returns an interface proxied by the dynamic *duck typing* (Koenig & Moo, 2005) engine:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
adapter.Target.Request();
```

Each call by the client to a certain method on the auto generated Target is intercepted by the **Invoke** method on the **AutoAdapter** component, as seen in the following code:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------------
public object Invoke(string methodName, object[] args) {
    // … C O N T R A C T S …

    // Get a delegate from the internal dictionary with a matching method signature
    var operation = GetAdapterOperation(methodName, args);

    // Invoke the delegate and return its result
    if (operation != null)
        return operation.DynamicInvoke(_adaptee, args);

    throw new Exception("No adapter method found");
}
```

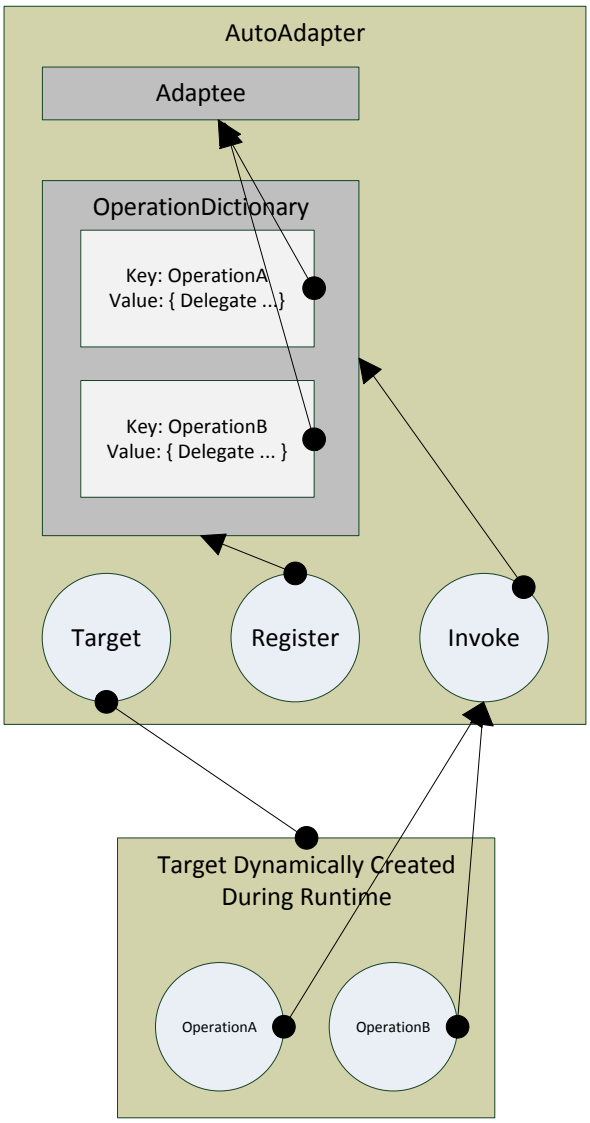

Figure 17. AutoAdapter APL component overview.

The **Invoke** method matches a registered Adapter method from the internal dictionary, with the method signature received from its arguments. If an Adapter method is found in the internal dictionary that matches the method signature, then the call is routed to it and the result returned.

Figure 17 shows a graphical overview of the **AutoAdapter** component. It shows the three main contracts of an **AutoAdapter**, namely **Target**, **Register** and **Invoke**. The registration set of contracts is used to register Target methods that use the internal Adaptee instance together with Adapter logic in the body of the method. The figure also shows the **Target** contract, which is used to retrieve a dynamically created instance of a Target during runtime. Finally, Figure 17 shows the **Invoke** method used by the *duck typing* runtime (Koenig & Moo, 2005). The dynamically created Target forwards all local invocations to the invoke method on the **AutoAdapter** instance, from where the call is forwarded to the correct delegate in the dictionary. For example, a call to the **OperationA** method on the Target is forwarded to the **Invoke** method on the **AutoAdapter**. From there, a delegate that represents an **OperationA** is retrieved from the dictionary and is executed. The result, if any, is passed back to the caller.

The registration of Adapter operations against a certain method available on the Target interface can be improved by using C# dynamics or lambda expressions, as shown in Appendix I. The same mechanism can be used for all the components in this thesis that have to register a method that will be used in a *duck typing* (Koenig & Moo, 2005) environment.

7.3 Theoretical Examples

The following example shows the usage of the **AutoAdapter** component. An **AutoAdapter** instance is created with an **Adaptee** instance. The **Adaptee** instance has internal state, together with a **SpecificRequest** method that takes one **string** argument. The **ITarget** Target has one **Request** method that also takes in one **string** argument. In the following example a lambda expression **(x, y) => x.SpecificRequest(y)** is registered against the **Request** method available on the **ITarget** Target:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ITarget { void Request(string arg); }

public class Adaptee {
    private string _state;

    public Adaptee(string state) { _state = state; }

    public void SpecificRequest(string arg) {
        Console.WriteLine("Called SpecificRequest() : " + state "|" + arg);
    }
}

public static void Run() {
    var adaptee = new Adaptee("[State]");
    var adapter = new AutoAdapter<ITarget, Adaptee>(adaptee); // Creates an Adapter for Target ITarget
```

```
    // Register the lambda expression against the "Request" method on the Target interface
    adapter.RegisterAction<string>("Request", (x, y) => x.SpecificRequest(y));

    // Delegates the call to the injected AdapterAction
    adapter.Target.Request("[External Data]");
}

/* Output
Called SpecificRequest() : [State]|[External Data]
*/
```

In the above example the **`adapter.Target.Request`** invocation delegates the call to the injected **`AdapterAction`** lambda expression.

The next example shows another usage of the **`AutoAdapter`** and is almost identical to the previous example. In this example, however, the **`Request`** method on the Target **`ITarget`** returns a **`string`**. The example thus shows the registration and usage of an **`AdapterFunc`** delegate:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ITarget { string Request(string arg); }

public class Adaptee {
    private string _state;

    public Adaptee(string state) { _state = state; }

    public string SpecificRequest(string arg) {
        Console.WriteLine("Called SpecificRequest() : " + state "|" + arg);
        return "[" + arg + "]";
    }
}

public static void Run() {
    var adaptee = new Adaptee("[State]");
    var adapter = new AutoAdapter<ITarget, Adaptee>(adaptee); // Creates an Adapter for Target ITarget

    // Register the lambda expression against the "Request" method on the Target interface
    adapter.RegisterFunc<string, string>("Request", (x, y) => x.SpecificRequest(y));

    // Delegates the call to the injected AdapterFunc
    string ret = adapter.Target.Request("[External Data]");
    Console.WriteLine("Ret : " + ret);
}

/* Output
Called SpecificRequest() : [State]|[External Data]
Ret : [[External Data]]
*/
```

The next example shows how multiple Adapter methods, in this case using an **`AdapterAction`** and an **`AdapterFunc`** delegate, can be registered with the **`AutoAdapter`** component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ITarget {
    void Request1();
    string Request2(string arg1);
}
```

```
public class Adaptee {
    private string _state;

    public Adaptee(string state) {
        _state = state;
    }

    public void SpecificRequest1() { Console.WriteLine("Called SpecificRequest1()"); }

    public string SpecificRequest2(string arg) {
        Console.WriteLine("Called SpecificRequest2() : " + state "|" + arg);
        return "[" + arg + "]";
    }
}

public static void Run() {
    var adaptee = new Adaptee("[State]");
    var adapter = new AutoAdapter<ITarget, Adaptee>(adaptee); // Creates an Adapter for Target ITarget

    // Register the adaptee.SpecificRequest method against the "Request" method on the Target interface
    adapter.RegisterAction<string>("Request1", (x, y) => x.SpecificRequest(y));

    // Register the lambda expression against the "Request" method on the Target interface
    adapter.RegisterFunc<string, string>("Request2", (x, y) => x.SpecificRequest(y));

    // Delegates the call to the injected AdapterAction
    adapter.Target.Request1();

    // Delegates the call to the injected AdapterFunc
    string ret = adapter.Target.Request2("[External Data]");
    Console.WriteLine("Ret : " + ret);
}

/* Output
Called SpecificRequest1()
Called SpecificRequest2() : [State]|[External Data]
Ret : [[External Data]]
*/
```

The output shows that all of the Adapters were called successfully.

7.4 Outcome

The componentization of the adapter design pattern is a success, because it meets all the requirements listed in section 1.4:

- **Completeness**: The adapter design pattern library component cover all cases described in the original core design pattern.

- **Usefulness**: The adapter design pattern library component is useful, because it solves most of the adapter scenarios desired by a developer. The **`AutoAdapter`** component solves the standard criteria for an Adapter, and is not overly complex to use. A developer is only tasked with defining the Target interface and injecting the adapting methods. A developer thus does not have to implement the Adapter boiler plate code manually. However, in some scenarios, implementing an Adapter manually might still be appropriate, especially to maintain the

cohesion of the adapter algorithms. A manually implemented Adapter, in most cases, is also simple to implement and does not need much boiler plate code.

- **Faithfulness**: The implementation of the reusable `AutoAdapter` component differs from the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). In the original implementation the Adapter is hand coded with the corresponding methods that implement the Adaptee. With the reusable `AutoAdapter` component, the Adapter class is dynamically created during runtime using meta-programming (Perrotta, 2010). The methods are injected with the component using anonymous functions (Ierusalimschy, 2003) and lambda expressions (Michaelis, 2010). The outcome of the component is, however, the same as the original pattern and implements its defined intent.

- **Type-safety**: The string literals used when registering the adapter methods are not type-safe. Lambda expressions (expressions trees) (Albahari & Albahari, 2007, p. 317) however, can be used to solve the type-safe registration problem, as shown in Appendix I. Other than that, the library component is fully type-safe.

- **Extended applicability**: The adapter library component does not cover more cases than the original adapter pattern.

- **Performance**: The adapter library component does have a performance impact because of the usage of *duck typing* (Koenig & Moo, 2005). Appendix II shows the performance impact of *duck typing*. The performance impact is, however, acceptable in normal situations.

The adapter pattern is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable pattern component.

The following language features are fundamental to the implementation or usage of the reusable adapter design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010), Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005), Duck Typing (Koenig & Moo, 2005) and Meta-programming (Perrotta, 2010).

Chapter 8

8 DECORATOR

8.1 Introduction

The decorator design pattern bestows additional behaviour on an object dynamically during runtime. It thus provides a flexible alternative to sub-classing for extending object behaviour (Gamma, Helm, Johnson, & Vlissides, 1994).

8.1.1 Structure.

The following figure shows the formal structure of the decorator design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

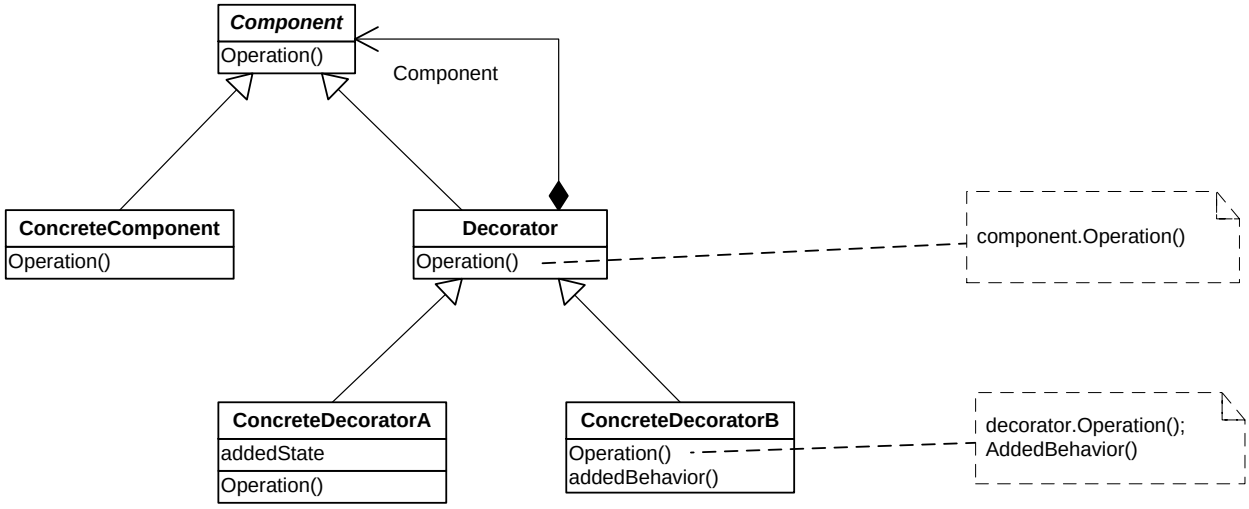

Figure 18. Decorator structure.

8.1.2 Participants.

The classes and/or objects participating in the decorator design pattern are:

- **Component**

  A Component declares the interface for Decorator instances. The operations declared in the interface will thus have behaviour dynamically added during runtime.

- **ConcreteComponent**

  A ConcreteComponent declares an instance that implements the Component interface.

- **Decorator**

  A Decorator holds and manages an association to a Component instance. A Decorator also implements the Component interface.

- **ConcreteDecorator**

  A ConcreteDecorator also implements the operations defined by its Component's interface. A decorated operation combines the behaviour of the Decorator and the Component instance in order to add functionality dynamically. A ConcreteDecorator thus adds new behaviour to the Component.

8.2 Library Components

8.2.1 The AutoDecorator component.

The **`AutoDecorator`** APL component maps registered delegates to methods available on the Component interface. The **`AutoDecorator`** holds a dictionary of delegates with a corresponding **`DynamicMethod`** instance as the key. The **`AutoDecorator`** also inherits from the **`Decorator`** APL component, which stores an internal reference to a certain Component and realizes the dynamic *duck typing* (Koenig & Moo, 2005) **`IDynamicInvoker`** interface. On the **`Decorator`** APL component, the internally stored Component type is defined by the **`TComponent`** generic argument. The **`Decorator`** component is abstract, with an abstract **`Invoke`** method, which is used by the *duck typing* (Koenig & Moo, 2005) engine. The abstract **`Invoke`** method must be overridden by a base class. The code snippet below shows the implementation of the abstract **`Decorator`** component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class Decorator<TComponent> : // TComponent defines the component participant
        IDynamicInvoker // Used for duck typing and forces the implementation
                        // of the Invoke method
    where TComponent : class {

    // Internal reference to a component participant
    protected TComponent Component;

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(Component != null, "The internal component cannot be null");
    }

    // Constructor that takes a component
```

```
    protected Decorator(TComponent component) {
        Component = component;
    }

    // … S N I P …

    // Used for duck typing
    public abstract object Invoke(string methodName, object[] args);
}
```

The **`AutoDecorator`** APL component implements the overridden **`Invoke`** method, which takes in as arguments the method name of a certain invocation and its arguments. The **`Invoke`** method checks whether a delegate for the certain method signature exists in the internal dictionary. If one does exist, it is executed and the result returned:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class AutoDecorator<TComponent> : Decorator<TComponent> // Uses the Decorator component
    where TComponent : class {
    private readonly Dictionary<DynamicMethod, Delegate> _operationDictionary =
            new Dictionary<DynamicMethod, Delegate>(); // Internal method dictionary.
                                                       // A delegate stored in this dictionary
                                                       // is mapped to a method on the TComponent
                                                       // contract.
                                                       // The Invoke method executes the appropriate
                                                       // delegate stored in the dictionary by
                                                       // matching the method names.

    private volatile TComponent _targetCache; // Dynamicly generated TComponent instance that
                                              // is generated during runtime

    public AutoDecorator(TComponent component) : base(component) {
        // … S N I P …
    }

    // Register methods.
    // Four different type of delegates can be registered:
    // Action                 : .Net Action
    // Func                   : .Net Func (function)
    // ActionDecoratorStrategy : APL action decorator strategy delegate
    // FuncDecoratorStrategy   : APL function decorator strategy delegate

    // Register an Action with no arguments
    public void (MethodInfo method, Action operation) { … }

    // Register an Action with no arguments
    public void RegisterAction RegisterAction(string methodName, Action operation) { … }

    // Register an Action with one argument
    public void RegisterAction<TArg1>(MethodInfo method, Action<TArg1> operation) { … }

    // Register an Action with one argument
    public void RegisterAction<TArg1>(string methodName, Action<TArg1> operation) { … }

    // … M O R E …

    // Register a Func with no arguments
    public void RegisterFunc<TResult>(MethodInfo method, Func<TTResult> operation) { … }

    // Register a Func with no arguments
    public void RegisterFunc<TResult>(string methodName, Func<TResult> operation) { … }

    // Register a Func with one argument
    public void RegisterFunc<TArg1, TResult>(MethodInfo method, Func<TArg1, TResult> operation) { … }
```

```
    // Register a Func with one argument
    public void RegisterFunc<TArg1, TResult>(string methodName, Func<TArg1, TResult> operation) { … }

    // … M O R E …

    // Register an ActionDecoratorStrategy delegate with no arguments
    public void RegisterStrategy(MethodInfo method, ActionDecoratorStrategy decoratorStrategy) { … }

    // … M O R E …

    // Register a FuncDecoratorStrategy delegate with no arguments
    public void RegisterStrategy<TResult>(MethodInfo method,
                                          FuncDecoratorStrategy<TResult> decoratorStrategy) { … }

    // … M O R E …

    // The Invoke method routes an invocation on the dynamicly created TComponent instance
    // returned by the Target property to an appropriate delegate stored in the internal dictionary
    public override object Invoke(string methodName, object[] args) {
        Contract.Requires<ArgumentException>(!string.IsNullOrEmpty(methodName),
                                             "Argument methodName cannot be null");

        // Call the decorator strategy that can be an ActionDecoratorStrategy or FuncDecoratorStrategy
        var decoratorStrategy = GetDecoratorStrategy(methodName, args);
        if(decoratorStrategy != null) {
            var internalComponentOperation = GetInternalComponentOperation(methodName, decoratorStrategy);
            return InvokeDecoratorStrategy(decoratorStrategy, internalComponentOperation, args);
        }

        // Else - just call both the component method and
        //        the registered method normally as an Action or Func
        var componentMethod = GetComponentMethod(methodName, args);
        var registeredMethod = GetRegisteredMethod(methodName, args);
        object ret = null;

        if(componentMethod != null) { ret = componentMethod.DynamicInvoke(args); }
        if(registeredMethod != null) { ret = registeredMethod.DynamicInvoke(args); }

        If(componentMethod == null && registeredMethod == null) {
            throw new Exception("No method found to invoke.");
        }

        // If it is a Func, the registered method's return value
        // takes precedence over the component method's return value
        return ret;
    }

    public TComponent Target {
        get {
            Contract.Ensures(Contract.Result<TComponent>() != null);
            return DoubleCheckedLock<TComponent>.Create(_targetCache, this,
                      () => this.AsIf<TComponent>(true));
        }
    }

    //… M O R E …
}
```

Four different types of delegates can be registered with the **`AutoDecorator<TComponent>`** component against a specific method available on the **`TComponent`** Component. These delegates are a **`Func`** (Microsoft, 2010h), **`Action`** (Microsoft, 2010a), **`FuncDecoratorStrategy`** and **`ActionDecoratorStrategy`**. The **`FuncDecoratorStrategy`** and **`ActionDecoratorStrategy`** are APL delegates.

The **ActionDecoratorStrategy** delegate takes in an **Action** as its first argument. The rest of the arguments in an **ActionDecoratorStrategy** define the number of arguments on the underlying **Operation** that is decorated:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate void ActionDecoratorStrategy(Action decoratorOperation);
public delegate void ActionDecoratorStrategy<TArg>(Action<TArg> decoratorOperation, TArg args);
public delegate void ActionDecoratorStrategy<TArg1, TArg2>(Action<TArg1, TArg2> decoratorOperation,
                                                           TArg1 arg1, TArg2 arg2);
//… M O R E …
```

The following example shows the usage of the **ActionDecoratorStrategy** delegate:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
// ActionDecoratorStrategy example:
// x is an Action representing the method being decorated on the internal TComponent Component
// y is a string that represents the argument of the decorated method
//
// Thus decorator.Target.Foo("Hello World") does:
//
//   x(y) - - - > this.Component("Hello World");
//   Console.WriteLine("More" + y) - - - > Console.WriteLine("More" + "Hello World");

decorator.RegisterStrategy<string>("Foo", (x, y) => { x(y); Console.WriteLine("More" + y); });
```

The example above shows how a lambda expression is used to register an **ActionDecoratorStrategy** with an **AutoDecorator** instance. In the lambda expression, the **x** argument represents a specific method on the internal Component instance referenced by the **AutoDecorator**. The **y** argument represents the argument type of the specific method on the Component instance. The **string** template argument thus tells the **ActionDecoratorStrategy** that the method being decorated on the Component interface has one argument, which is of type **string**. Thus, in this case, the method is **void Foo(string str)**. The **Foo** method does not return any value; it is thus not a function. The injected lambda expression first makes a call to **x(y)**. The **x(y)** call is translated into an invocation on the **Foo(string str)** method on the internally stored Component instance, with **y** as the string argument. The second decorative part of the lambda expression writes a comment to the console, using the **y** string argument.

The **FuncDecoratorStrategy** is almost exactly the same as the **ActionDecoratorStrategy**, except that it takes in a **Func** as its first argument and not an **Action**, as shown below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate TResult DecoratorStrategy<TResult>(Func<TResult> decoratorOperation);
public delegate TResult DecoratorStrategy<TArg, TResult>(Func<TArg, TResult> decoratorOperation,
                                                         TArg arg);
public delegate TResult DecoratorStrategy<TArg1, TArg2, TResult>(
        Func<TArg1, TArg2, TResult> decoratorOperation, TArg1 arg1, TArg2 arg2);
public delegate TResult DecoratorStrategy<TArg1, TArg2, TArg3, TResult>(
        Func<TArg1, TArg2, TArg3, TResult> decoratorOperation, TArg1 arg1, TArg2 arg2, TArg3 arg3);
```

An **`ActionDecoratorStrategy`** represents a decorative expression for a method on the Component interface that does not return anything. A **`FuncDecoratorStrategy`**, on the other hand, represents a decorative expression for a method on the Component interface that does return something. The first argument for both the **`ActionDecoratorStrategy`** and **`FuncDecoratorStrategy`** delegates represents the same operation on the Component which is being decorated, where the operation is available on the Component reference that is stored internally on an **`AutoDecorator`** instance. These delegates make it possible to write advanced decorator algorithms that can be registered with an **`AutoDecorator`** instance.

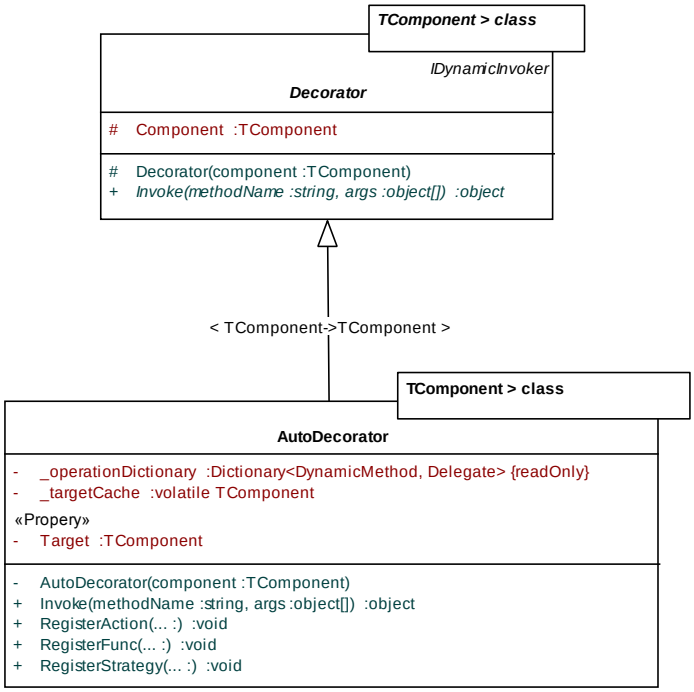

Figure 19. UML class diagram of the AutoDecorator APL component.

Figure 19 shows a UML class diagram of the APL **`AutoDecorator`** component. It shows all the different types of registration methods available on the **`AutoDecorator`**. Figure 19 also shows the **`Target`** property, which is used to acquire, during runtime, an auto generated instance that realizes the Component interface. Finally, the figure shows the **`Invoke`** method that is used by the *duck typing* (Koenig & Moo, 2005) runtime. All method invocations on the auto generated Component instance are routed to the **`Invoke`** method. The **`Invoke`** method then routes the invocation to the appropriate registered operation.

Figure 20 shows an overview of the **`AutoDecorator`** APL component. The register set of methods registers a new decoration operation into the internal dictionary. Each registered operation is associated with one method defined on the Component interface. The **`Target`** property returns a

runtime generated instance that realizes the Component interface. Each invocation on the instance is routed through to the **`Invoke`** method on the **`AutoDecorator`** instance. The **`Invoke`** method then routes the call to the appropriate operation stored in the internal operation dictionary.

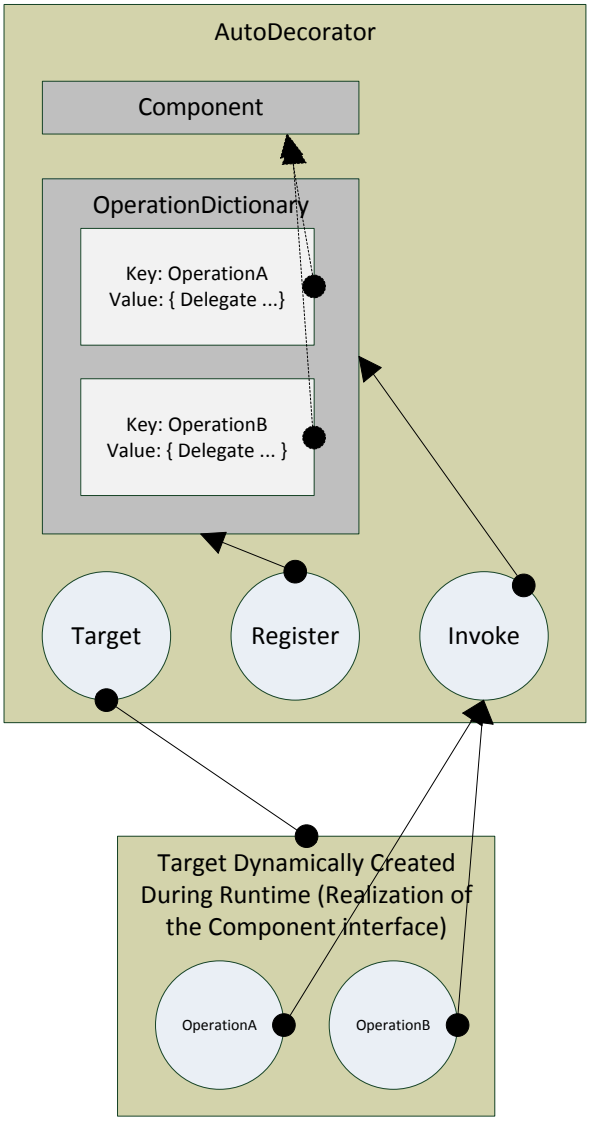

Figure 20. AutoDecorator APL component overview.

The registration of operations against a certain method available on the Component interface can be improved by using C# dynamics or lambda expressions, as shown in Appendix I. The same mechanism can be used for all the components in this thesis that have to register a method that will be used in a *duck typing* (Koenig & Moo, 2005) environment.

8.3 Theoretical Examples

The following example shows the usage of the **`AutoDecorator`** APL component. The **`IComponent`** interface defines the methods of the Component, some with arguments and others with return values:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IComponent {
    void Operation1(); // No arguments and no return value
    void Operation2(string arg); // One argument and no return value
    uint Operation3(); // No argument and one return value
    uint Operation4(string arg); // One argument and one return value
}
```

A ConcreteComponent is also defined, implementing the **`IComponent`** contract:

```
C#  (APL Example)
-----------------------------------------------------------------------------------------------------------
public class ConcreteComponent : IComponent {
    public void Operation1() { Console.Write("a"); }
    public void Operation2(string arg) { Console.Write("a" + arg); }
    public uint Operation3() { return 10; }
    public uint Operation4(string arg) { Console.Write("a"); return 10; }
}
```

The example creates a ConcreteDecorator **`decorator1`**, and injects a decorative algorithm for each method on the Component using the **`RegisterStrategy`** set of methods on the **`AutoDecorator`** instance. Each decorative algorithm is injected using a lambda expression. An instance of the ConcreteComponent is used to construct the **`decorator1`** object, as seen below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
static public void Main() {
    var concreteComponent = new ConcreteComponent();
    var decorator1 = new AutoDecorator<IComponent>(concreteComponent); // Create a decorator

    // Register a decorative expression for "Operation1" (no direct decoration in this case)
    decorator1.RegisterStrategy("Operation1", x => x());

    // Register a decorative expression for "Operation2"
    decorator1.RegisterStrategy<string>(concreteComponent.Operation2,
                                        (x, y) => { x(y); Console.Write("b" + y); });

    // Register a decorative expression for "Operation3"
    decorator1.RegisterOperation("Operation3", x => x() + 2);

    // Register a decorative expression for "Operation4"
    decorator1.RegisterOperation<string, uint>("Operation4",
        (x, y) => { Console.Write("b" + y); return x(y) + 2; });

    // Use decorator1
    Console.WriteLine("Decorator 1:");
    Console.Write("Operation1: "); decorator1.Target.Operation1(); Console.WriteLine();
    Console.Write("Operation2: "); decorator1.Target.Operation2("c"); Console.WriteLine();
    Console.Write("Operation3: "); Console.Write(decorator1.Target.Operation3()); Console.WriteLine();
    Console.Write("Operation4: "); Console.Write(decorator1.Target.Operation4("c")); Console.WriteLine();

    // Use decorator2
    var decorator2 = new AutoDecorator<IComponent>(decorator1.Target); // Link to decorator1
    Console.WriteLine(); Console.WriteLine("Decorator 2:");
```

```
    decorator2.RegisterOperation(concreteComponent.Operation3, x => x() * 4);
    Console.Write("Operation3: "); Console.Write(decorator2.Target.Operation3()); Console.WriteLine();
}

/* Output
Decorator 1:
Operation1: a
Operation2: acbc
Operation3: 12
Operation4: bca12

Decorator 2:
Operation3: 48
*/
```

Each method on the ConcreteDecorator **`decorator1`** in the example is called, some with passed-in arguments. The **`Target`** property on the **`AutoDecorator`** is used to acquire a dynamically generated instance that implements the **`IComponent`** contract. All the requests made on the instance are forwarded to an instance of the **`AutoDecorator`** component, where they are processed. The output shows that all the methods on the **`decorator1`** object were processed correctly.

The example also creates a ConcreteDecorator **`decorator2`** using the **`decorator1`** instance, and injects a decorative algorithm using a lambda expression for **`Operation3`**. The output shows that **`Operation3`** on the **`decorator2`** object was processed correctly.

8.4 Outcome

The componentization of the decorator design pattern is a success because it meets all of the requirements listed in section 1.4:

- **Completeness**: The decorator design pattern library components cover all cases described in the original core design pattern.

- **Usefulness**: The decorator design pattern library components are useful because they solve all of the decorator scenarios desired by a developer. The components serve the same functionality as a hand written decorator; however, a developer does not have to write the decorator boiler plate code by hand. With the **`AutoDecorator`** group of components, a developer is only responsible for implementing the Component and hooking up the decorative algorithms. The **`AutoDecorator`** group of components are relatively simple and easy to use.

- **Faithfulness**: The implementation of the **`AutoDecorator`** group of components deviates from the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). The implementation makes use of dynamic *duck typing* (Koenig & Moo, 2005) and meta-programming (Perrotta, 2010) in order to hook up decorative algorithms within an

`AutoDecorator` instance which, in return, auto generates a ConcreteDecorator instance. The end result and intent of the decorator library components are, however, the same.

- **Type-safety**: The `Register` methods on the `AutoDecorator` component use non type-safe string literals for the specification of the method names. Lambda expressions (expressions trees) (Albahari & Albahari, 2007, p. 317) however, can be used to solve the type-safe registration problem, as shown in Appendix I. Other than that, all the library components are fully type-safe.

- **Extended applicability**: The decorator library components do not cover more cases than the original decorator pattern.

- **Performance**: The decorator library components do have a performance impact because of the usage of *duck typing* (Koenig & Moo, 2005). Appendix II shows the performance impact of *duck typing*. The performance impact is, however, acceptable in normal situations.

The decorator pattern is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable pattern components.

The following language features are fundamental to the implementation or usage of the reusable decorator design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010), Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005), Duck Typing (Koenig & Moo, 2005) and Meta-programming (Perrotta, 2010).

Chapter 9

9 COMPOSITE

9.1 Introduction

The composite design pattern distinguishes objects in a certain tree-like structure to represent a part-whole hierarchy. The recursive tree-like structure allows single objects and compositions of objects to be treated uniformly by a client or user (Gamma, Helm, Johnson, & Vlissides, 1994).

9.1.1 Structure.

The following figure shows the formal structure of the composite design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

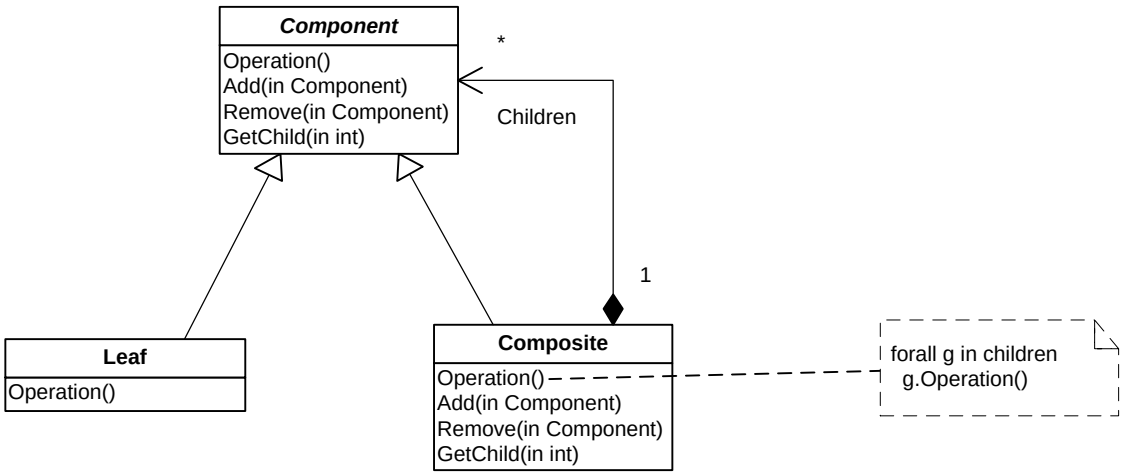

Figure 21. Composite structure.

9.1.2 Participants.

The classes and/or objects participating in the composite pattern are:

- **Component**

  A Component defines the interface for every instance used in the composition. It also defines the interface for retrieving, using and controlling each one of its child components. It might also implement the default behaviour for the operations defined in the desired Component

contract. A Component might also declare an interface and the implementation for retrieving, using and controlling the component's parent recursively.

- **Leaf**

  A Leaf is an instance that implements the behaviour of the interface defined in the Component, but it has no children. It is thus known as a primitive instance in the composition.

- **Composite**

  A Composite is an instance that implements the behaviour of the interface defined in the Component. It also holds references to child Components.

- **Client**

  A Client holds a reference to a Component interface through which it uses the Composite's functionality.

9.2 Library Components

9.2.1 The AutoComposite component.

The `AutoComposite` reusable component implements a Composite for a specific Component that is represented by a generic argument `TComponent`. At the heart of the `AutoComposite` is a list of child Components, which is of type `List<IComponent<TComponent>>`. The `IComponent` interface is part of the APL library. The `AutoComposite` also holds a dictionary of composite strategy function delegates, composite function delegates and normal operations for methods that are present on the `TComponent` interface. Composite strategy function delegates and composite functions delegates are used to register algorithms that participate in the composite pattern, against methods present on the `TComponent` Component. Note that only the methods on the `TComponent` interface that participate in the composite pattern and are tagged with the `CompositeMethodAttribute` APL attribute can be registered on the `AutoComposite` component with a composite strategy function delegate or composite function delegate. The composite function delegates stored in the internal dictionary use a special `CompositeFunc` APL delegate. The composite strategy delegates stored in the internal dictionary use a special `CompositeStrategy` APL delegate. A certain method on the `TComponent` Component that participates in the composite pattern can be registered by means of either a `CompositeFunc` delegate or a `CompositeStrategy` delegate or as a C# `Action` or `Func`. The method must, however, be tagged with the `CompositeMethodAttribute` APL attribute. C# `Action` or `Func` registered methods are invoked as normal actions or functions by the `AutoComposite` on each Component stored in the internal list. A method on

the **TComponent** Component that is attributed with the **CompositeMethodAttribute** APL attribute will thus always join the composite pattern.

It is possible that a certain Component, where the Component is defined by the **TComponent** generic argument, might have methods that should not be part of the composite pattern. Methods on the **TComponent** Component that do not participate in the composite pattern must be registered as C# **Actions** or **Funcs** and these methods must not be tagged with the **CompositeMethodAttribute** APL attribute on the Component.

An exception is thrown in the **Invoke** method of the **AutoComposite** if no registered implementation is found for a method on the **TComponent**. Furthermore, at least one of the methods on the **TComponent** generic argument must be registered with an **AutoComposite** instance as a composite method. Thus, at least one of the methods on the **TComponent** Component must be tagged with the **CompositeMethodAttribute** APL attribute. The code below shows the implementation of the **AutoComposite** in the APL library:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IComponent<T> {
    IList<IComponent<T>> GetList();
    T GetInterface();
}

public class AutoComposite<TComponent> : IDynamicInvoker, IComponent<TComponent>
    where TComponent : class {
    private readonly List<IComponent<TComponent>> _components;
    private readonly Dictionary<DynamicMethod, Delegate> _operationDictionary;
    private volatile TComponent _target; // Target cache

    [ContractInvariantMethod]
    private void ContractInvariant() {
        Contract.Invariant(_components != null, "The components list cannot be null");
        Contract.Invariant(_operationDictionary != null, "The operationDictionary cannot be null");
    }

    public AutoComposite() {
        _components = new List<IComponent<TComponent>>();
        _operationDictionary = new Dictionary<DynamicMethod, Delegate>();
    }

    public AutoComposite(List<IComponent<TComponent>> components) : this() { … }

    // Register methods.
    // Four different type of delegates can be registered:
    // Action                   : .Net Action
    // Func                     : .Net Func (function)
    // CompositeStrategy        : APL composite strategy delegate
    // CompositeFunc            : APL composite func delegate

    // Register an Action with no arguments
    public void RegisterAction(MethodInfo method, Action operation) { … }

    // Register an Action with no arguments
    public void RegisterAction(string methodName, Action operation) { … }

    // Register an Action with one argument
```

```
public void RegisterAction<TArg1>(MethodInfo method, Action<TArg1> operation) { … }

// Register an Action with one argument
public void RegisterAction<TArg1>(string methodName, Action<TArg1> operation) { … }

// … M O R E …
// Register a Func with no arguments
public void RegisterFunc<TResult>(MethodInfo method, Func<TTResult> operation) { … }

// Register a Func with no arguments
public void RegisterFunc<TResult>(string methodName, Func<TResult> operation) { … }

// Register a Func with one argument
public void RegisterFunc<TArg1, TResult>(MethodInfo method, Func<TArg1, TResult> operation) { … }

// Register a Func with one argument
public void RegisterFunc<TArg1, TResult>(string methodName, Func<TArg1, TResult> operation) { … }

// … M O R E …
// Register a CompositeStrategy with no arguments
public void RegisterStrategy<TResult>(MethodInfo method,
                                      CompositeStrategy<TResult> compositeStrategy) { … }

// Register a CompositeStrategy with no arguments
public void RegisterStrategy<TResult>(string methodName,
                                      CompositeStrategy<TResult> compositeStrategy) { … }

// Register a CompositeStrategy with one argument
public void RegisterStrategy<TArg1, TResult>(MethodInfo method,
                                CompositeStrategy<TArg1, TResult> compositeStrategy) { … }

// Register a CompositeStrategy with one argument
public void RegisterStrategy<TArg1, TResult>(string methodName,
                                CompositeStrategy<TArg1, TResult> compositeStrategy) { … }

// … M O R E …
// Register a CompositeFunc with no arguments
public void RegisterCompositeFunc<TResult>(MethodInfo operation,
                                           CompositeFunc<TComponent, TResult> compositeFunc) { … }

// Register a CompositeFunc with no arguments
public void RegisterCompositeFunc<TResult>(string operation,
                                           CompositeFunc<TComponent, TResult> compositeFunc) { … }

// … M O R E …
// Register a CompositeStrategy with one argument
public void RegisterCompositeFunc<TArg1, TResult>(MethodInfo operation,
                                     CompositeFunc<TComponent, TArg1, TResult> compositeFunc) { … }

// Register a CompositeStrategy with one argument
public void RegisterCompositeFunc<TArg1, TResult>(string operation,
                                     CompositeFunc<TComponent, TArg1, TResult> compositeFunc) { … }

// … M O R E …
public object Invoke(string methodName, object[] args) {
    Contract.Requires<ArgumentException>(!string.IsNullOrEmpty(methodName),
                                         "Argument methodName cannot be null");

    // Step 1 : Are there any registered component methods?
    if(HasComponentMethodToInvoke(methodName, args)) {
        // If it is a CompositeStrategy registered method, then execute it…
        var strategy = GetCompositeStrategy(methodName, args);
        if(strategy != null) {
            object ret = null;
            _components.ForEach(x => { ret = InvokeStrategy(x, methodName, args, strategy, ret); });
            return ret;
        }
```

```
            // Step 2 : Or if it is a CompositeFunc registered method, then execute it…
            var func = GetCompositeFunc(methodName, args);
            if(func != null) { return func.DynamicInvoke(GetFuncArguments(args)); }

            // Step 3 : Or just a Func or Action but must still participate in the composite pattern
            var method = GetNormalComponentMethod(methodName, args);
            if (method != null) { // Call it on each method in the list, ignore the return value if Func
               _components.ForEach(x => { method.DynamicInvoke(args);});
               return null;
            }
        }

        // Are there any non registered component methods?
        if(HasNonComponentMethodToInvoke(methodName, args)) {
            // If it is not a component method, just execute it normally…
            var method = GetNonComponentMethod(methodName, args);
            if (method != null) { return method.DynamicInvoke(args); }
        }

        throw new Exception("The method " + methodName + " is not registered.");
    }

    public TComponent Target {
        get {
            Contract.Ensures(Contract.Result<TComponent>() != null);
            _target = DoubleCheckedLock<TComponent>.Create(_target, this,
                () => this.AsIf<TComponent>(true));
            return _target;
        }
    }

    public IList<IComponent<TComponent>> GetList() { return _components; }
    public TComponent GetInterface() { return Target; }

    // … S N I P …
}
```

The **`IComponent<T>`** APL interface, shown on page 96, defines a method that returns the list of Components and a method that returns the Component interface. The **`IComponent<T>`** interface also injects extension methods with the **`ComponentExtend`** APL class as shown below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public static class ComponentExtend {
    public static int GetCount<T>(this IComponent<T> component) { … }
    public static void Add<T>(this IComponent<T> composite, IComponent<T> element) { … }
    public static void Remove<T>(this IComponent<T> component, T obj) { … }
    public static IComponent<T> Remove<T, TArg>(this IComponent<T> component,
                                                RemoveCompare<T, TArg> removeCompare, TArg arg) { … }
    public static void ForEach<T>(this IComponent<T> composite, Action<T> action)  { … }
    public static IEnumerator GetEnumerator<T>(this IComponent<T> component) { … }
    public static IComponent<T> Find<T>(this IComponent<T> component, T obj) { … }
    public static IComponent<T> Find<T, TArg>(this IComponent<T> component,
                                              FindCompare<T, TArg> findCompare, TArg arg) { … }
}
```

The **`AutoComposite<TComponent>`** component thus offers the above composite extension methods because it realizes the **`IComponent<TComponent>`** interface.

The **`RegisterStrategy`** set of methods, defined on the **`AutoComposite<TComponent>`** component, is used to register a **`CompositeStrategy`** delegate that is associated with a certain method on the Component.

The code below shows the implementation of the delegate, where the **T** generic argument denotes the return type of the composite method. Multiple **CompositeStrategy** delegates exist in the APL library, where each delegate caters for a different set of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate T CompositeStrategy<T>(T leftValue, T rightValue);
public delegate T CompositeStrategy<TArg1, T>(TArg1 arg1, T leftValue, T rightValue);
public delegate T CompositeStrategy<TArg1, TArg2, T>(TArg1 arg1, TArg2 arg2, T leftValue, T rightValue);
// … M O R E …
```

The example code below shows how a summation lambda expression **(l, r) => l + r** is registered on an instance of the **AutoComposite** component against the **Operation** method. The **Operation** method, which returns an **int**, is declared on the example **ITheComponent** Component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var composite1 = new AutoComposite<ITheComponent>();
composite1.RegisterStrategy<int>("Operation", (l, r) => l + r);
```

The **AutoComposite** component will thus apply the injected **CompositeStrategy** algorithm to all of the Components in its internal list for the specific registered method. The registered method using the **CompositeStrategy** delegate will thus always participate in the composite pattern. It is important to note that the **CompositeStrategy** delegate can only be applied to functions. The usage of the **(l, r) => l + r** expression by the **AutoComposite** component can be explain as follows: The **l** value is the value at which the Component iteration is currently. The **r** value is what the method invocation for the current Component in the iteration has returned. The expression, which in this case is a summation, is evaluated and its result will either be the **l** value for the next iteration or the overall return value.

The **RegisterCompositeFunc** method is used to register **CompositeFunc** delegates that are also associated with a certain method on the Component that participates in the composite pattern. The **CompositeFunc** set of delegates takes in the **IComponent<TComponent>** as its first argument and the rest of the arguments are determined by the number of arguments on the Component method itself. The **IComponent<TComponent>** interface has a **GetInterface** method, from where the instance of the **TComponent** contract can be acquired. The **CompositeFunc** delegate thus gives the user the ability to inject a powerful composite algorithm that can utilise the contract of a full Component. The code below shows some of the **CompositeFunc** delegates that are available in the APL library, where each one caters for a certain number of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate TResult CompositeFunc<TComponent, out TResult>(IComponent<TComponent> component); // None
public delegate TResult CompositeFunc<TComponent, in T, out TResult>( // One
    IComponent<TComponent> component, T arg);
public delegate TResult CompositeFunc<TComponent, in T1, in T2, out TResult>( // Two
    IComponent<TComponent> component, T1 arg1, T2 arg2);
// … M O R E …
```

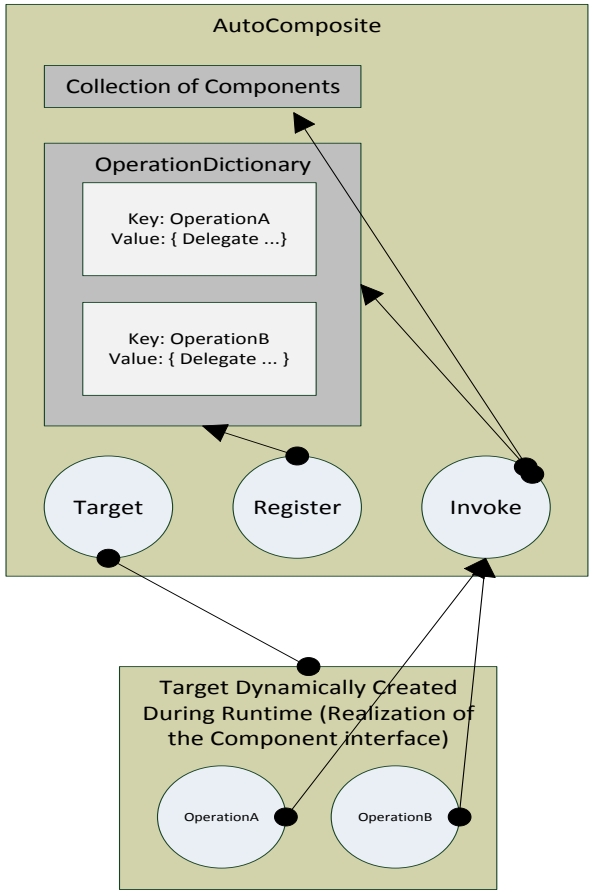

Figure 22. AutoComposite APL component overview.

Figure 22 shows an overview of the **`AutoComposite`** APL component. The register set of methods registers a new operation on the internal dictionary. Each registered operation has an association with one, and only one, method defined on the Component interface. The **`Target`** property returns a runtime generated instance that realizes the Component interface. Each invocation on the instance is routed through to the **`Invoke`** method on the **`AutoComposite`** instance. The **`Invoke`** method instance then routes the call to the appropriate operation stored in the internal operation dictionary.

The following code shows how the **`CompositeFunc`** can be used in order to inject a composite algorithm with an instance of the **`AutoComposite`** component. The code shows the registration of a lambda expression that must be used for the **`Operation`** method that is defined on a certain Component interface:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
// In the lambda expression below c defines the Component and
// arg defines the argument of the "Operation" method
// The first template argument – int – is the type of the single argument on the "Operation" method
// The second template argument – string – is the return type of the "Operation" method
// The "Operation" method is avaliable on the Component
```

```
composite.RegisterCompositeFunc<int, string>("Operation", (c, arg) => {
                  var stringBuilder = GetStringBuilder(arg, c.GetInterface().Name, c.GetCount());
                  c.ForEach(x => stringBuilder.Append(x.GetInterface().Operation(arg + 2)));
                  return stringBuilder.ToString(); });
```

In the example above, the **Operation** method has one argument of type **int** and it returns a value of type **string**. The injected lambda expression has access to the Composite instance that is passed in as the first argument. The user can thus inject complex composite algorithms without writing the necessary composite pattern plumping code.

The **Target** property on the **AutoComposite** returns an instance of a dynamically created class that implements the **TComponent** contract. All calls on the instance are first intercepted by the **Invoke** method which receives the runtime name of the method and the runtime arguments. The **Invoke** method first tests to see whether the received method must participate in the Composite pattern, by looking for a **CompositeStrategy** delegate in the internal dictionary with the same method signature. If a delegate is found, it is invoked together with all the Composite's registered Components, as shown in the code snippet below found within the **Invoke** method:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
// If it is a CompositeStrategy registered method, then execute it…
var strategy = GetCompositeStrategy(methodName, args);
if(strategy != null) {
    object ret = null;
    _components.ForEach(x => { ret = InvokeStrategy(x, methodName, args, strategy, ret); });
    return ret;
}
```

If no strategy is found, then the **Invoke** method determines whether a relevant **CompositeFunc** delegate is available for the given method in the internal dictionary:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
// If it is a CompositeFunc registered method, then execute it…
var func = GetCompositeFunc(methodName, args);
if(func != null) { return func.DynamicInvoke(GetFuncArguments(args)); }
```

If no **CompositeStrategy** or **CompositeFunc** is found, then the relevant method on all of the internally stored Components is invoked:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
// Or just a Func or Action but must still participate in the composite pattern
var method = GetNormalComponentMethod(methodName, args);
if (method != null) { // Call it on each method in the list, ignore the return value if Func
    components.ForEach(x => { method.DynamicInvoke(args);});
    return null;
}
```

A registered method on the internally stored Composite doesn't participate in the composite pattern if it is not attributed with the **`CompositeMethodAttribute`** attribute. In this case, the **`Invoke`** method just invokes the registered method normally and the internal list of Components is thus ignored:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
// Are there any non registered component methods?
if(HasNonComponentMethodToInvoke(methodName, args)) {
    // If it is not a component method, just execute it normally…
    var method = GetNonComponentMethod(methodName, args);
    if (method != null) { return method.DynamicInvoke(args); }
}
```

9.2.2 The Composite component.

The **`Composite`** APL component is a simple component that is used in a *curiously recurring template pattern* (Coplien, 1995) environment. It takes in one generic argument that defines the underlying user coded Component. It also implements the **`IComponent<T>`** APL interface, giving it access to the large set of Component extension methods.

The code snippet below shows the implementation of the **`Composite`** APL component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class Composite<T> : IComponent<T> {
    protected List<IComponent<T>> List = new List<IComponent<T>>();
    protected void SetComposite(T composite) { Target = composite; }
    public IList<IComponent<T>> GetList() { return List; }
    public T GetInterface() { return Target; }
    public T Target { get; private set; }
    public int GetCount() { return List.Count; }
}
```

The **`Composite<T>`** component stores the list of Components internally. A developer now has access to a large number of standard Composite functionalities, including an enumerator to a list of Components. Component methods can now be added to the base hand coded Composite by a developer.

Developers need only concentrate on the algorithms of the methods defined in the user coded concrete Composite, and thus do not have to implement the entire pattern structure by hand:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface {
    public string Operation(int depth)
}

public class TheComposite : Composite<ITheComponent>, ITheComponent { // Using CRTP
    public TheComposite(string name) {
        Name = name;
        SetComposite(this);
    }

    public string Name { get; set; }
```

```
    // Implementation of the 'Operation' method that is defined on the ITheComponent interface
    public string Operation(int depth) {
        var stringBuilder = new StringBuilder(new String('-', depth));
        stringBuilder.Append("Set " + Name + " length :" + GetCount() + "\n");
        this.ForEach(x => stringBuilder.Append(x.Display(depth + 2)));
        return stringBuilder.ToString();
    }
}
```

A Leaf participant may also be added by using the **`Leaf`** APL component, where it implements the **`IComponent<T>`** interface for a certain Component contract **`T`**:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class Leaf<T> : IComponent<T> {
    private T _component;

    protected void SetComponent(T component) {
        Contract.Requires<ArgumentNullException>(component != null, "Argument component cannot be null");
        _component = component;
    }

    public IList<IComponent<T>> GetList() { return new List<IComponent<T>>(); } // Return an empty list
    public T GetInterface() { return _component; }
    public T Target { get { return _component;  } }
    public int GetCount() { return 0; }
}
```

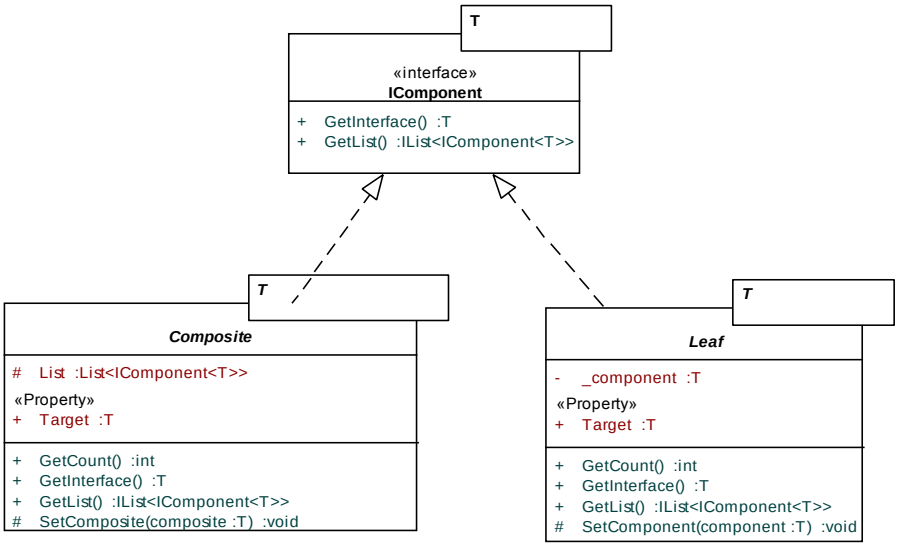

Figure 23. UML class diagram of the Composite APL component.

Figure 23 shows a UML class diagram of the **`Composite`** APL component. It shows the **`Composite`** and **`Leaf`** components and also their realization of the **`IComponent`** APL interface. A Leaf can easily be created using the **`Leaf`** APL component together with the *curiously recurring template pattern* (G´eraud &

Duret-Lutz, 2000). The methods of the **T** contract must, however, be implemented manually, as seen in the code below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class TheLeaf : Leaf<ITheComponent>, ITheComponent {
    public TheLeaf () { SetComponent(this); }
    public string NonCompositeOperation() { … }
}
```

9.3 Theoretical Examples

The first theoretical example shows the usage of the **AutoComposite** APL component. The Component contract is implemented with the **ITheComponent** interface, which implements the **IComponent<T>** APL interface. The **ITheComponent** Component has one **Operation** method that returns an **int**. The **Operation** method participates in the composite pattern. A Component can also have methods that are not used in the Composite pattern; however, this is not shown in this example. Only the methods that participate in the composite pattern should be attributed with the **CompositeMethodAttribute** attribute. A Leaf is also defined, which inherits from the **Leaf<T>** APL component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ITheComponent : IComponent<ITheComponent> {
    [CompositeMethod]
    int Operation();
}

public class ConcreteLeaf : Leaf<ITheComponent>, ITheComponent {
    private readonly int _value;

    public ConcreteLeaf(int value) {
        SetComponent(this);
        _value = value;
    }

    public int Operation() { return _value; }
}

class Program {
    static void Main() {
        var composite1 = new AutoComposite<ITheComponent>();
        composite1.RegisterStrategy<int>("Operation", (l, r) => l + r);
        var leaf1 = new ConcreteLeaf(10);
        composite1.Add(leaf1);
        var leaf2 = new ConcreteLeaf(12);
        composite1.Add(leaf2);

        var composite2 = new AutoComposite<ITheComponent>();
        composite2.RegisterStrategy<int>("Operation", (l, r) => l + r);
        var leaf3 = new ConcreteLeaf(18);
        composite2.Add(leaf3);
        var leaf4 = new ConcreteLeaf(22);
        composite2.Add(leaf4);

        // Add a composite2 to a composite1, which creates a tree-like structure
        composite1.Add(composite2.Target);
        var leaf5 = new ConcreteLeaf(45);
        composite1.Add(leaf5);
```

```
        // Add and remove a leaf
        var leaf6 = new ConcreteLeaf(9);
        composite1.Add(leaf6);
        composite1.Remove(leaf6);

        // Calculate the value
        int value = composite1.Target.Operation();
        Console.WriteLine("Value = " + value);
    }
}

/* Output
Value = 116
*/
```

In the above example, no Composites are hand coded. Both the **composite1** and **composite2** instances are implemented by new instances of the **AutoComposite** component. A **CompositeStrategy** is injected on the **Operation** method, using a lambda expression, on each Composite instance:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var composite1 = new AutoComposite<ITheComponent>();
composite1.RegisterStrategy<int>("Operation", (l, r) => l + r);

var composite2 = new AutoComposite<ITheComponent>();
composite2.RegisterStrategy<int>("Operation", (l, r) => l + r);
```

A couple of Leaf instances are also registered on the **composite1** and **composite2** Components. The **composite2** instance is also added to the **composite1** instance, as seen below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
composite1.Add(composite2.Target);
```

Finally, the **Operation** method is called on the **composite1** Composite that runs through all of the added Components. The **Target** property on the **AutoComposite** uses *duck typing* (Koenig & Moo, 2005) in order to map the **Operation** method to the injected **CompositeStrategy**, which in this case is the lambda expression **(l, r) => l + r**:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
int value = composite1.Target.Operation();
```

The output shows that the correct value was calculated and returned by the **Operation** invocation.

The final example shows the usage of the **Composite** APL component. It is almost the same as the previous example, except that the **Composite** component is used instead of the **AutoComposite** component. The **ConcreteComposite** inherits from the **Composite** component and is thus able to reuse most of the component's Composite functionality. The Composite only has to implement the **Operation** method:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ITheComponent : IComponent<ITheComponent> {
    [CompositeMethod]
    int Operation();
}

public class ConcreteComposite : Composite<ITheComponent>, ITheComponent {
    public ConcreteComposite() { SetComposite(this); }

    public int Operation() {
        int sum = 0;
        this.ForEach(x => sum = sum + x.Operation());
        return sum;
    }
}

public class ConcreteLeaf : Leaf<ITheComponent>, ITheComponent {
    private readonly int _value;

    public ConcreteLeaf(int value) {
        _value = value;
        SetComponent(this);
    }

    public int Operation() { return _value; }
}

class Program {
    static void Main() {
        var composite1 = new ConcreteComposite();
        var leaf1 = new ConcreteLeaf(10);
        composite1.Add(leaf1);
        var leaf2 = new ConcreteLeaf(12);
        composite1.Add(leaf2);

        var composite2 = new ConcreteComposite();
        var leaf3 = new ConcreteLeaf(18);
        composite2.Add(leaf3);
        var leaf4 = new ConcreteLeaf(22);
        composite2.Add(leaf4);

        // Add a composite2 to a composite1, which creates a tree-like structure
        composite1.Add(composite2.Target);
        var leaf5 = new ConcreteLeaf(45);
        composite1.Add(leaf5);

        // Add and remove a leaf
        var leaf6 = new ConcreteLeaf(9);
        composite1.Add(leaf6);
        composite1.Remove(leaf6);

        // Recursively display tree
        int value = composite1.Operation();
        Console.WriteLine("Value = " + value);
        Console.ReadLine();
    }
}

/* Output
Value = 116
*/
```

In the above example, a developer has access to the Component list inside the user coded **`Operation`** method. The **`Operation`** method iterates through all of the registered Components and calculates their sum and returns the value:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public int Operation() {
    int sum = 0;
    this.ForEach(x => sum = sum + x.Operation());
    return sum;
}
```

In the last full example, instances of the Composite **`ConcreteComposite`** class are created normally and Leaves are added using the inherited **`Add`** method, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var composite1 = new ConcreteComposite();
var leaf1 = new ConcreteLeaf(10);
composite1.Add(leaf1);
```

The output of the example is the same as the previous one, showing that the **`composite1.Operation()`** invocation was successful.

9.4 Outcome

The componentization of the composite design pattern is a success because it meets all the requirements listed in section 1.4:

- **Completeness**: The composite design pattern library components cover all cases described in the original design pattern.

- **Usefulness**: The composite design pattern library components are useful because they solve all of the composite scenarios desired by a developer. The components serve the same functionality as a hand written composite, without a developer having to write the composite boiler plate code by hand. With the **`AutoComposite`** component a developer is responsible only for implementing the Component and Leaf participants and hooking up the composite algorithms. With the **`Composite`** component a developer is also responsible only for implementing the Component and Leaf participants and implementing the composite methods. Both of the components are relatively simple and easy to use.

- **Faithfulness**: The **`AutoComposite`** reusable pattern component follows an implementation that differs from the original core pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). With the **`AutoComposite`** component, the Composite is generated during runtime using *duck typing* (Koenig & Moo, 2005) and meta-programming (Perrotta, 2010). The

`Composite` component implementation follows the same implementation as the original pattern described in *Design Patterns*.

- **Type-safety**: The registration methods on the `AutoComposite` component use non type-safe string literals for the specification of the method names. Lambda expressions trees (Albahari & Albahari, 2007, p. 317) however, can be used to solve the type-safe registration problem, as shown in Appendix I. Other than that, all the library components are fully type-safe.

- **Extended applicability**: The composite library components do not cover more cases than the original composite pattern.

- **Performance**: The composite library components do have a performance impact because of the usage of *duck typing* (Koenig & Moo, 2005). Appendix II shows the performance impact of *duck typing*. The performance impact is, however, acceptable in normal situations.

The composite pattern is fully componentizable, because the developer is not tasked with implementing any boiler plate code when using the reusable pattern components.

The following language features are fundamental to the implementation or usage of the reusable composite design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Attributes (Nagel, Evjen, Glynn, & Watson, 2010), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010), Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005), Duck Typing (Koenig & Moo, 2005) and Meta-programming (Perrotta, 2010).

Chapter 10

10 STATE

10.1 Introduction

The state design pattern (Gamma, Helm, Johnson, & Vlissides, 1994) tackles the challenge of how an object implements an interface differently according to the state it is in. This problem is sometimes incorrectly implemented using conditional statements such as **`if`** and **`switch`**. The pattern adheres to the refactoring rule of *Replace Conditional with Polymorphism* (Fowler, Beck, Brant, Opdyke, & Roberts, 1999, pp. 255-259) that states "*Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract*". The state design pattern shows an elegant object-oriented solution that is closed to change, yet open to extension (Meyer, 2000).

The pattern permits an object to change its functionality according to its internal state. It will thus appear as though the object has changed its class (Gamma, Helm, Johnson, & Vlissides, 1994). The intent is, therefore, to offer an unsophisticated and adaptable mechanism for an object to delegate messages to different concrete implementations depending on the state of the underlying object.

10.1.1 Structure.

The following figure shows the formal structure of the state design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

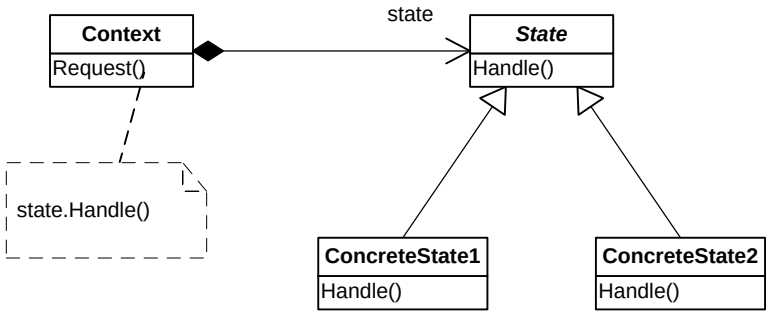

Figure 24. State structure.

10.1.2 Participants.

The classes and/or objects participating in the state design pattern are:

- **Context**

  The Context declares the interface that will be used by clients or users. It also holds, manages and uses an instance of the subclass of a ConcreteState that controls the present desired state.

- **State**

  A State declares an interface that implements the operations in a ConcreteState, which is associated with a distinctive state of the Context.

- **ConcreteState**

  A ConcreteState implements the operations declared in the State interface. It holds the actual state linked with a Context instance.

10.2 Library Components

10.2.1 The State component.

The APL **`IState`** interface defines a standard reusable contract for a State. The interface defines useful methods such as setting and getting the underlying State instance, as seen below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IState<TState> : IAutoState<TState> {
    void SetState(IState<TState> state); // Sets the state to the new IState state instance
    void SetState<TConcreteState>() where TConcreteState : TState; // Sets the state to TConcreteState
    void SetStateContext(IStateContext<TState> stateContext); // Sets a new Context
    TState GetTarget(); // Get a Target
    void SetTarget(TState state); // Set the Target
}
```

The APL **`IState`** interface also implements the **`IAutoState<TState>`** APL interface in order for an implementer of the **`IState`** interface to make use of the extension methods made available by the **`IAutoState<TState>`** APL interface. The implementer of the **`IState<TState>`** interface must implement all of the methods defined on the interface. The implementation can, however, delegate the processing to the relevant extension method that is made available on the **`IAutoState<TState>`** interface.

The **`State<TState>`** APL component implements a part of a ConcreteState that must be used in a *curiously recurring template pattern* (CRTP) (Coplien, 1995) setting. Figure 25 shows a UML class diagram of the **`State`** APL group of components, which illustrates the following four items: First, it illustrates the implementation hierarchy of the **`IState`** interface and the **`State`** component; secondly, the **`StateContext`** component's usage of the **`State`** component (discussed later in this chapter) and the **`StateContext`** component's implementation of the **`IStateContext`** interface; thirdly, the **`State`**

component's usage of the **`IState`** interface, which it uses to change the state of a **`StateContext`** instance and, fourthly, the dominance of the state getter and setter methods on the components.

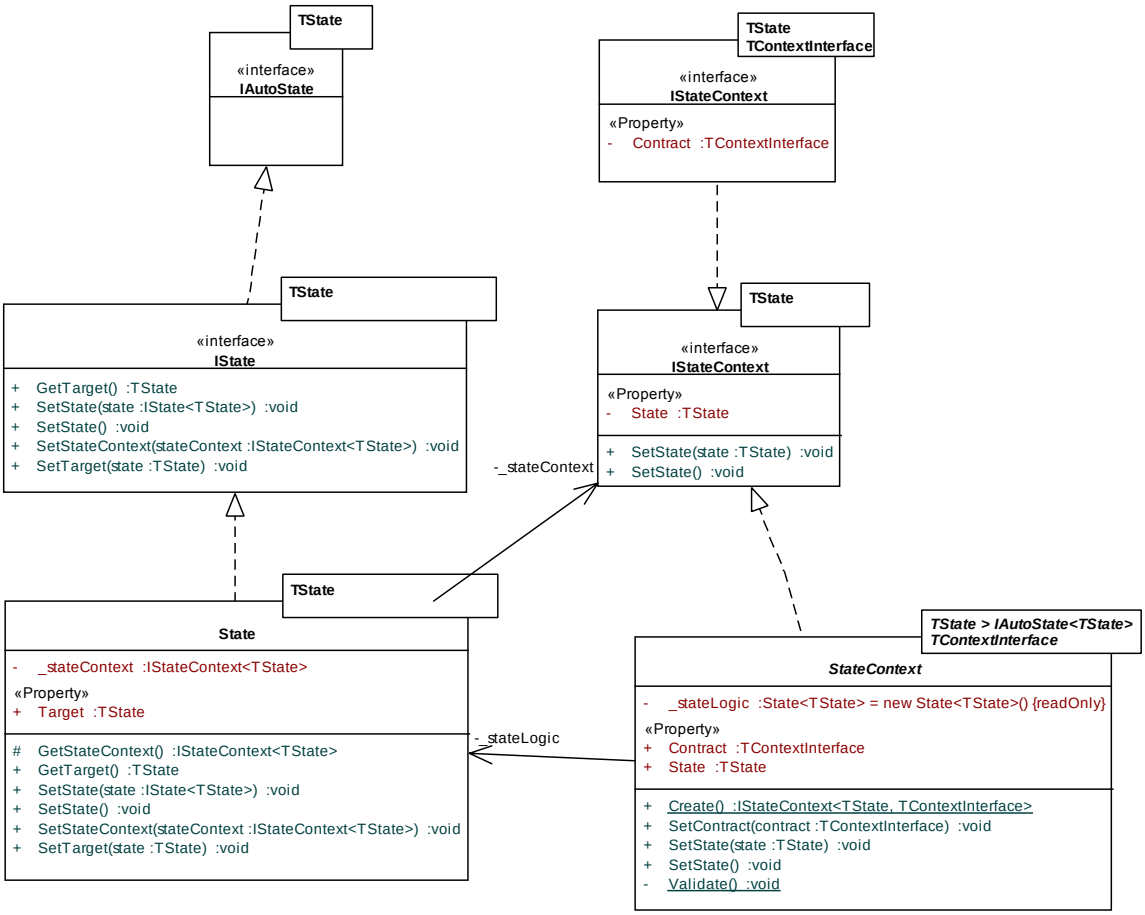

Figure 25. UML class diagram of the State APL component.

The **`State<TState>`** component, which realizes the **`IState<TState>`** APL interface, implements a certain *extension* of the state design pattern (Dyson & Anderson, 1997) where it holds an internal reference to the State's Context through an **`IStateContext<TState>`** APL interface, as illustrated in Figure 25. A State instance thus holds a reference back to its Context instance. This makes it possible for a ConcreteState to change the state, delegating the state change request to its holding Context:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class State<TState> : IState<TState> {
    private IStateContext<TState> _stateContext;

    public void SetStateContext(IStateContext<TState> stateContext) {
        Contract.Requires<ArgumentNullException>(stateContext!= null,
                                                 "Argument stateContext cannot be null");
        _stateContext = stateContext;
```

```
        Validate();
    }

    public IStateContext<TState> GetStateContext() {
        Contract.Ensures(Contract.Result<IStateContext<TState>>() != null);
        return _stateContext;
    }

    public void SetState(IState<TState> state) {
        Contract.Requires<ArgumentNullException>(state != null,
                                                 "Argument state cannot be null");
        Contract.Requires<ArgumentNullException>(_stateContext!= null,
                                                 "The internal stateContext cannot be null");
        _stateContext.SetState(state.GetTarget());
    }

    public void SetState<TConcreteState>()
        where TConcreteState : TState {
        Contract.Requires<ArgumentNullException>(_stateContext!= null,
                                                 "The internal stateContext cannot be null");
        _stateContext.SetState<TConcreteState>();
    }

    // … S N I P …

    public TState Target { get; set; }

    public TState GetTarget() {
        Contract.Ensures(Contract.Result<TState>() != null);
        return Target;
    }

    public void SetTarget(TState state) {
        Contract.Requires<ArgumentNullException>(state != null,
                                                 "Argument state cannot be null");
        Target = state;
    }
}
```

The **SetState** method changes the Context's state to the new desired state, where the state can be passed in with either a generic argument or as an **IState<TState>** APL interface. The **TState** generic argument of the **State<TState>** component defines the State participant of the state pattern. The **TConcreteState** generic argument that is passed to the **SetState** defines a ConcreteState participant and must be of type **TState**. The **SetStateContext** method changes the Context reference of the ConcreteState. This can only be done after the Context's state has been set to reference the State, as the **SetStateContext** will validate this rule. The **SetTarget** method registers the **TState** State whereas the **GetTarget** method retrieves it. The **SetTarget** and **GetTarget** methods are used only in a scenario where the creator and user of the component are separated and the user does not have access to the **TState**. The user can thus gain access to the **TState** by using the **GetTarget** method.

In the example below, the example **ConcreteState** class is defined in a *curiously recurring template pattern* (Coplien, 1995) setting. The **ConcreteState** class implements the user defined **ITheState** interface, which define the **HandleState1** and **HandleState2** methods as its contract. The **ConcreteState** class can now use the inherited **SetState** method in order to change the state on the Context:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class ConcreteState : State<ITheState>, ITheState {
    public void HandleState1() {
        // … S N I P …
        SetState<MyConcreteStateB>(); // Change the state to MyConcreteStateB
        // … S N I P …
    }

    public void HandleState2() { … }
}
```

The **`IStateContext<TState>`** APL interface defines the contract for a standard Context. It defines methods whereby the state of the context can be changed. The **`TState`** generic argument defines the State participant and it can be retrieved with the **`State`** property:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IStateContext<TState> {
    void SetState(TState state); // Sets the State using a TState instance
    void SetState<TConcreteState>() where TConcreteState : TState; // Sets the State using TConcreteState
    TState State { get; } // Gets the State
}
```

The **`StateContext<TState, TContextInterface>`** APL component defines a standard Context. It is defined with two generic arguments **`TState`** and **`TContextInterface`**. The **`TState`** generic argument must be a specific State implementation, and the **`TContextInterface`** must be a Context interface. The **`StateContext<TState, TContextInterface>`** component realizes the **`IStateContext<TState, TContextInterface>`** APL interface. The **`TState`** generic argument must be of type **`IAutoState<TState>`** because it must have the standard injected State functionality:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class StateContext<TState, TContextInterface> : IStateContext<TState, TContextInterface>
    where TState : IAutoState<TState> {
    public TState State { get; private set; }

    public void SetState(TState state) {
        System.Diagnostics.Contracts.Contract.Requires<ArgumentNullException>(state != null,
                                              "Argument state cannot be null");

        // … S N I P …
        State = state;
        state.SetStateContext(this); // After setting the state on the context,
                                     // set the context on the state
    }

    public void SetState<TConcreteState>() where TConcreteState : TState {
        SetState(StateFactory<TConcreteState, TState>.Create());
    }

    public void SetContract(TContextInterface contract) {
        System.Diagnostics.Contracts.Contract.Requires<ArgumentNullException>(state != null,
                                              "Argument state cannot be null");

        Contract = contract;
```

```
    }

    public TContextInterface Contract { get; private set; }


    static public IStateContext<TState, TContextInterface> Create<TStateContext, TConcreteState>()
        where TStateContext : StateContext<TState, TContextInterface>, TContextInterface
        where TConcreteState : State<TState>, TState {
        System.Diagnostics.Contracts.Contract.Ensures(
            Contract.Result<IStateContext<TState, TContextInterface>>() != null);

        Validate<TConcreteState>();
        return StateContextFactory<TStateContext, TConcreteState, TState, TContextInterface>.Create();
    }

    // … S N I P …
}
```

The **State** auto property, as seen in the implementation code above, holds the current State instance of the Context. The internal state cannot be set with the **State** property; it must be set with the public **SetState** method. The **SetState** method sets the state of the Context instance and also sets the Context on the State instance. A State instance holds a reference back to its Context, in order for the state to be changed.

The **StateFactory** APL component, which is used in the **SetState<TConcreteState>** method on the **StateContext** component, is used to create a normal instance of the ConcreteState. Different types of ConcreteState creational strategies exist in the APL library, such as **Normal**, **Singleton** and **Flyweight**. The **StateFactory** APL component just creates a normal instance of a certain ConcreteState. The **SingletonStateFactory** APL component creates a singleton instance of a certain ConcreteState and the **FlyweightStateFactory** creates ConcreteState instances in a flyweight pattern setting (Gamma, Helm, Johnson, & Vlissides, 1994). The ConcreteState creational strategies adhere to the creational patterns discussed in *Design Patterns* with regard to the state pattern (Gamma, Helm, Johnson, & Vlissides, 1994).

The different creational strategies available are **Normal**, **Singleton** and **Flyweight** and are defined on the **StateCreationStyle** enumerator:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public enum StateCreationStyle {
    Normal, // The State class is a normal instance and holds a reference back to the context
    Singleton, // The state class is a Singleton and doesn't hold any context reference
    Flyweight // The state class is a Flyweight and doesn't hold any context reference
}
```

The **StateCreationStyle** enumerator is used when the **StateAttribute** is tagged on a State interface, as shown later in this section.

The **StateContext** component is used in a *curiously recurring template pattern* (Coplien, 1995) setting, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface ITheContextInterface { void Request(); }

public class Context : StateContext<ITheState, IContextInterface>, ITheContextInterface { // Using CRTP
    // … S N I P …

    public void Request() { // Implementation of the 'Request' method on the ITheContextInterface

        // … S N I P …
        SetState<ConcreteState1>; // Switch to state ConcreteState1
        // … S N I P …
        State.HandleState1();     // Invoke the HandleState1 method on the state instance
        // … S N I P …
        SetState<ConcreteState2>; // Switch to state ConcreteState2
        // … S N I P …
        State.HandleState3();     // Invoke the HandleState3 method on the state instance
        // … S N I P …
        State.HandleState2();     // Invoke the HandleState2 method on the state instance
        // … S N I P …
        SetState<ConcreteState3>; // Switch to state ConcreteState3
        // … S N I P …
    }
}
```

The above example shows the usage of the **`StateContext<TState, TContextInterface>`** component. The user must supply the State interface and the Context interface through generic arguments when using the **`StateContext`** component. In this example, the **`ITheState`** interface defines the State contract and the **`TContextInterface`** interface defines the Context contract. The user defined **`Context`** class must implement the Context interface. The Context concrete instance, thus, must implement the method of the **`ITheContextInterface`**, which in this case is **`Request`**. The **`Request`** method implements the necessary state transitions using the **`State`** property inherited from the **`StateContext`** component.

A **`SingletonStateContext`** component also exists in the APL library. It performs exactly the same functionality as the **`StateContext`** except that it uses a **`SingletonStateFactory`** to create an instance of a certain ConcreteState:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public void SetState<TConcreteState>() where TConcreteState : TState {
    Validate<TConcreteState>();
    SetState(SingletonStateFactory<TConcreteState, TState>.Create());
}
```

The **`SetState`** method validates that the **`TConcreteState`** is indeed a singleton (Gamma, Helm, Johnson, & Vlissides, 1994) by checking some standard singleton rules. The developer of the **`TConcreteState`** does not have to implement a full singleton by hand. The **`TConcreteState`** must only be implemented in such a way that the **`SingletonStateFactory`** can use it as a singleton:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class ConcreteStateA : ITheState {
    // Must be set to private in order to pass validations
    private ConcreteStateA() { }
```

```
    public void HandleState1(IStateContext<IState> context) {
        // … S N I P …

        // Switches the state to a Singleton ConcreteStateB using the context argument
        context.SetState<ConcreteStateB>();
    }

    public void HandleState2(IStateContext<IState> context) { … }
}
```

In the above example, **`ConcreteStateA`** is set to **`private`** in order to prohibit intermittent instance creation of the **`ConcreteStateA`** class. The validation in the **`SetState`** method on the **`SingletonStateContext`** component fails if its **`TConcreteState's`** constructor is not private. The **`ConcreteStateA`** implementation can also no longer inherit from the **`State`** APL component. This is because the **`State`** component holds a reference back to a certain Context. An instance of the **`State`** component thus holds its own internal state, which makes a **`State`** component instance impossible to share with multiple Context instances; therefore it cannot be a Singleton. A Singleton ConcreteState must be shareable and must hold no state. The Context must, therefore, be passed to the ConcreteState through the arguments of the handler methods, as shown in the example below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Singleton)] // The state interface is used as a singleton
public interface ITheState {
    // State handle…
    void HandleState1();

    // Another state handle…
    void HandleState2(IStateContext<IState> context); // Pass in the Context's state
}
```

It is the developer's responsibility to code the State contract and the ConcreteState implementation of that contract. The developer must also define the creational style for the State with the **`StateAttribute`** APL attribute. The **`SingletonStateContext`** will, however, validate if the ConcreteState was implemented correctly. The **`SingletonStateContext`** will thus validate that the ConcreteState holds only one constructor that is private, with no arguments, and that it does not hold any state.

The **`FlyweightStateContext`** APL component performs the same functionality as the **`StateContext`** and **`SingletonStateContext`**. It does, however, ensure that the ConcreteStates are also Flyweights (Gamma, Helm, Johnson, & Vlissides, 1994). The **`FlyweightStateContext`** forces the ConcreteStates to be Flyweights through its **`SetState`** method:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
// Set the state using a flyweightkey
public void SetState<TConcreteState, TFlyweightKey>(TFlyweightKey flyweightKey)
    where TConcreteState : TState {
    Validate(flyweightKey); // Validate if the new state is possible…
    // Sets the new state using the flyweightKey argument
    SetState(FlyweightStateFactory<TConcreteState, TFlyweightKey>.Create(flyweightKey));
}
```

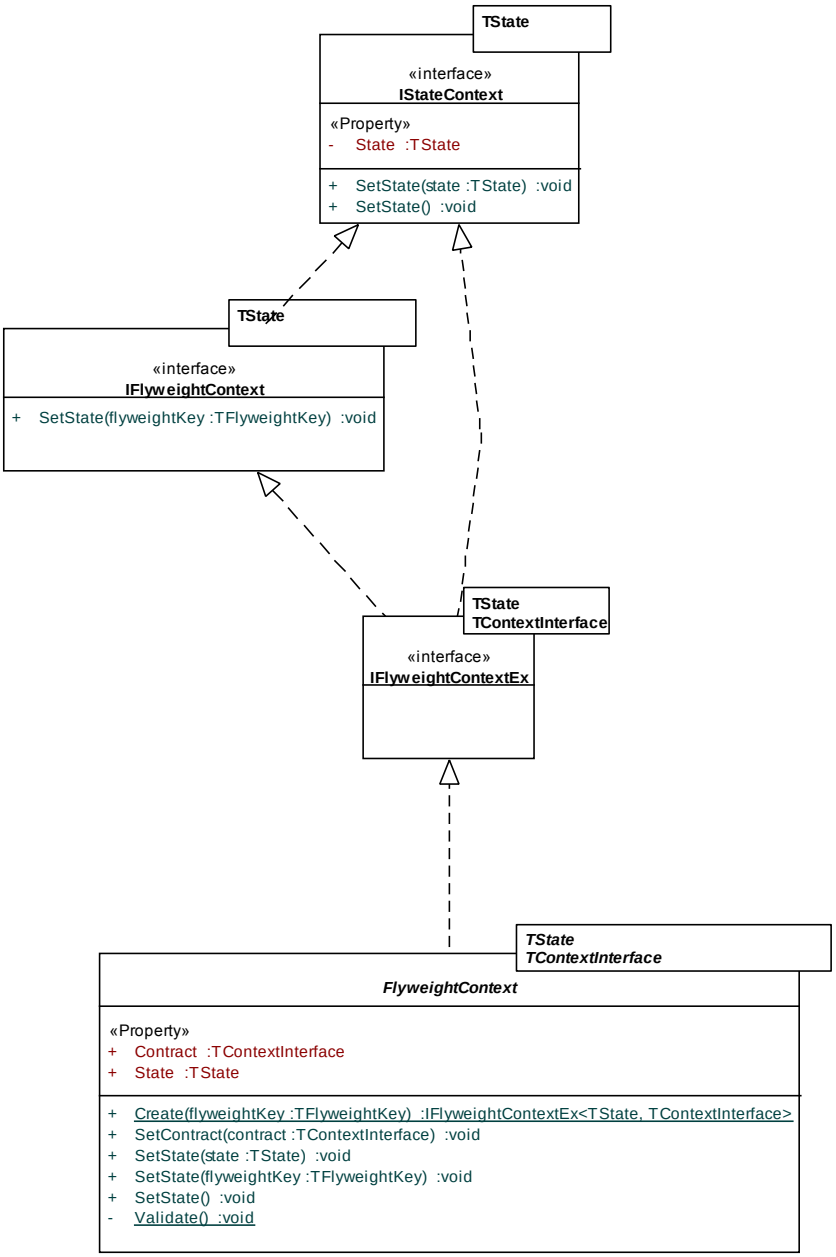

Figure 26. UML class diagram of the FlyweightContext APL component.

Figure 26 shows a UML class diagram of the **`FlyweightContext`** APL component and the APL interfaces it implements.

The ConcreteStates must be implemented by the developer in order for them to be used by the APL flyweight components. This means that the developed ConcreteStates must follow the same kind of rules as the singleton ConcreteStates. The ConcreteStates can have a state, but the state must be

related to the key of the Flyweight, because their instances must be shareable. A ConcreteState's constructor must also be private, in order to protect the creation of the class. The ConcreteState's private constructor must, however, take one argument that represents the key of the Flyweight. The APL flyweight components use this keyed private constructor in order to create a unique instance of a ConcreteState that is related to the key:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class MyConcreteStateA : ITheState {
    private Setting Setting = Setting.SettingA;

    // Private constructor with a Flyweight key.
    private MyConcreteStateA(Setting setting) { Setting = setting; }
    public override void HandleState1(IFlyweightContext<IMyState> context) { … }
    public override void HandleState2(IFlyweightContext<IMyState> context) { … }
}
```

An **IFlyweightContext<TState>** interface must be passed to a handler in order for it to make state changes. The **IFlyweightContext<TState>** interface adds an extra **SetState** method to the **IStateContext<TState>** interface, which also supplies the Flyweight key as a generic argument:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IFlyweightContext<TState> : IStateContext<TState> {
    void SetState<TConcreteState, TFlyweightKey>(TFlyweightKey flyweightKey)
        where TConcreteState : TState;
}
```

It is now possible for the user or ConcreteState to change the state by using the Context:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
context.SetState<ConcreteState, Key>(Key.Value1);
```

In the above example the key is an enumerator on which **Value1** is a variable.

A ConcreteState can also be defined without having to inherit from the **State<TState>** APL component. C# does not allow multiple inheritance (Balagurusamy, 2008). A C# class can thus only inherit from one base class. This limits the possibilities for applying multiple patterns on a certain class if only the *curiously recurring template pattern* (Coplien, 1995) is available. The state pattern in the APL library gives the developer the option to implement the **IAutoState<TState>** interface on a ConcreteState instead of inheriting from the **State<TState>** APL component. The **IAutoState<TState>** interface injects the same standard State functionality as the **State<TState>** component, by using C# extension methods (Esterbrook, 2001) (Jesse & Xie, 2008):

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class ConcreteState : IAutoState<ITheState>, ITheState {
    public void HandleState1() {
        // … S N I P …
```

```
        this.SetState<MyConcreteStateB, IMyState>();
        // … S N I P …
    }

    public void HandleState2() { … }
}
```

The extension methods are injected with the **`DynamicStateEx`** static APL class, whereby they implement all the methods on the **`IState<TState>`** APL interface.

The code below shows the implementation of the **`IAutoState<TState>`** interface:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IAutoState<TState> { } // Just an empty interface
```

The **`IAutoState<TState>`** APL interface is empty because all the methods it injects are defined in the extension methods. Figure 27 shows a UML class diagram of the **`DynamicStateEx`** APL static class. The **`IAutoState<TState>`** interface thus allows for the automatic inclusion of those state pattern methods on a certain State implementation without using inheritance.

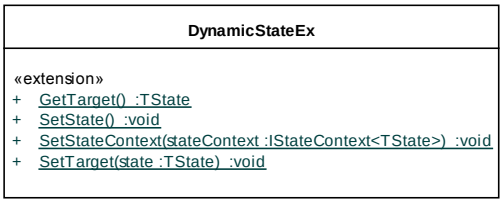

Figure 27. UML class diagram of the DynamicStateEx APL component.

The code below shows the implementation of the **`DynamicStateEx`** extension method in the APL library:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public static class DynamicStateEx {
    public static TState GetTarget<TState>(this IAutoState<TState> obj) { … }

    public static void SetState<TConcreteState, TState>(this IAutoState<TState> obj)
        where TConcreteState : TState { … }

    public static void SetStateContext<TState>(this IAutoState<TState> obj,
                                               IStateContext<TState> stateContext) { … }

    public static void SetTarget<TState>(this IAutoState<TState> obj, TState state) { … }
}
```

A State implemented with the **`IAutoState<TState>`** APL interface can be used as a participant in another design pattern using APL components, because multiple interface implementations are

allowed in C#. A State implemented using the *curiously recurring template pattern* (Coplien, 1995) cannot be a combined with another design pattern using inheritance only APL components. This is because the *curiously recurring template pattern* uses inheritance, and multiple inheritance is not allowed in C#. For example, a State can be defined as a Decorator and a Composite, as the following code snippet shows:

```
C# (Example)
-----------------------------------------------------------------------------------------------------
public class MyState : Composite<ITheComponent<T>>, ITheComponent<T>, // APL class using CRTP
                       IAutoState<TState>, // APL interface
                       IAutoDecorator { // APL interface
    // … S N I P …
}
```

In the above example, the **`MyState`** class is made a State and a Decorator by using APL interfaces. The **`MyState`** class is made a decorator by the **`IAutoDecorator`** APL interface, which is not discussed in this thesis. The APL interfaces use C# extension methods in order to inject the desired boiler plate reusable pattern code. The **`MyState`** class is also made a Composite by using an APL composite component using CRTP (Coplien, 1995). The implementation would not have been possible if more than one pattern injected its boiler plate code using the *curiously recurring template pattern* (Coplien, 1995), as seen below:

```
C# (Error Example)
-----------------------------------------------------------------------------------------------------
public class MyState : Composite<ITheComponent<T>>, ITheComponent<T>, // APL class using CRTP
                       State<TState>, // APL class using CRTP (error)
                       IAutoDecorator { // APL interface
    // … S N I P …
}
```

In the above code, the **`MyState`** class inherits from both the **`Composite`** and **`State`** APL classes. This scenario is not allowed, because multiple inheritance is illegal in C# (Balagurusamy, 2008).

An **`IAutoStateContext<TState>`** also exists in the APL library, which allows the use of an interface instead of the **`StateContect<TState>`** component. The **`DynamicStateContextEx`** static class injects the necessary standard Context functionality by using C# extension methods, as shown below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------
public interface IAutoStateContext<TState> { // None } // Just an empty interface

public static class DynamicStateContextEx {
    public static TState GetState<TState>(this IAutoStateContext<TState> autoStateContext) {
        // … S N I P …
    }

    public static void SetState<TState>(this IAutoStateContext<TState> autoStateContext, TState state) {
        // … S N I P …
    }

    public static void SetState<TState, TConcreteState>(this IAutoStateContext<TState> autoStateContext)
        where TConcreteState : TState { … }
```

```
    private static IStateContext<TState> GetStateContext<TState>(
        IAutoStateContext<TState> autoStateContext) { … }
}
```

The following example shows how the **`IAutoStateContext`** can be used when creating a Context:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class Context : IAutoStateContext<ITheState>, IContextInterface { // No CRTP
    public void Request() {
        // … S N I P …
        this.GetState().HandleState1(); // The GetState method is auto injected
        // … S N I P …
        this.SetState<IMyState, MyConcreteStateA>(); // The SetState method is auto injected
        // … S N I P …
        this.GetState().HandleState2(); // The GetState method is auto injected
        // … S N I P …
        this.GetState().HandleState3();// The GetState method is auto injected
        // … S N I P …
    }
}
```

The example above shows how the Context has access to State functionalities such as **`SetState`** and **`GetState`**, which are auto injected by the **`IAutoStateContext<TState>`** APL interface.

Different creational strategies can also be used when using the **`IAutoState`** and **`IAutoStateContext`** interfaces, by using the **`AutoStateContextFactory`** APL component and other library factories. When using the auto state interfaces, the state creational strategy must be supplied by using the **`StateCreationStyle`** property on the **`State`** attribute:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Singleton)] // The state instance will be a singleton
public interface ITheState : IAutoState<ITheState> {
    void HandleState1(IAutoStateContext<IMyState> context);
    void HandleState2(IAutoStateContext<IMyState> context);
}
```

Note, however, that the Context must be supplied to the state handler when using the singleton or flyweight (Gamma, Helm, Johnson, & Vlissides, 1994) pattern, because the ConcreteState itself cannot hold any state. The **`ITheState`** State defined above can thus be used to implement ConcreteState participants, as shown in the example below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class ConcreteStateA : ITheState {
    public void HandleState1(IAutoStateContext<IMyState> context) {
        // … S N I P …
        context.SetState<IMyState, MyConcreteStateB>(); // This will create a singleton MyConcreteStateB
    }

    public void HandleState2(IAutoStateContext<IMyState> context) { … }
}
```

The ConcreteState can now be used by the state factories in order to create it:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var contextFactory = new AutoStateContextFactory<MyConcreteStateA, IMyState>();
var context = contextFactory.Create<Context, IContextInterface>();
```

The **SetState** used in the **HandleState1** handler on the ConcreteState in the example on the previous page also uses the internal APL state factories. In this case the **ITheState** State is defined as a Singleton. The **SetState** method will thus return a Singleton instance.

A flyweight (Gamma, Helm, Johnson, & Vlissides, 1994) pattern can also be used as a creational strategy for the ConcreteStates in the example on the previous page. The **StateCreationStyle** on the **ITheState** handler must be changed to **Flyweight** and the ConcreteState must be given a private constructor that takes in one Flyweight key:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Flyweight)] // The state instance will be a flyweight
public interface ITheState : IAutoState<ITheState> {
    void HandleState1(IAutoFlyweightContext<IMyState> context);
    void HandleState2(IAutoFlyweightContext<IMyState> context);
}
```

The **IAutoFlyweightContext** APL interface must also be used instead of the **IFlyweightContext**, because the **FlyweightFactory** APL component needs a key in order to create a Flyweight. This key is used by the private constructor of the ConcreteFlyweight during its construction.

10.3 Theoretical Examples

The following example shows the usage of the **State<TState>** APL component. First, the State implementation must be defined with an appropriate creational style which, in this case, is **StateCreationStyle.Normal**. A creational style of **Normal** means that the ConcreteState instance is created normally and thus will not be shared. The State interface in this example has two handlers or methods **HandleState1** and **HandleState2**. The State interface must also implement the **IState<TState>** APL interface, which realizes standard state functionality:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Normal)] // Defines a Normal State
public interface IMyState : IState<IMyState> {
    void HandleState1();
    void HandleState2();
}
```

Two ConcreteState classes are defined in the example. Both implement the user defined **IMyState** State interface. The two ConcreteState classes also inherit from the **State<TState>** APL component, which itself implements the methods on the **IState<TState>** APL interface. In the example on the next page both the **MyConcreteStateA** class and **MyConcreteStateB** class are defined in a *curiously recurring template pattern* (Coplien, 1995) setting:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class MyConcreteStateA : State<IMyState>, IMyState { // The state is implemented using CRTP
    public void HandleState1() {
        Console.WriteLine("Calling HandleState1 from state A");
        SetState<MyConcreteStateB>(); // Set the State to MyConcreteStateB
    }

    public void HandleState2() { Console.WriteLine("Calling HandleState2 from state A"); }
}

public class MyConcreteStateB : State<IMyState>, IMyState {
    public void HandleState1() { Console.WriteLine("Calling HandleState1 from state B"); }
    public void HandleState2() { Console.WriteLine("Calling HandleState2 from state B"); }
}
```

In the code above, the **`HandleState1`** handler on the **`MyConcreteStateA`** State class changes the state on the Context to **`MyConcreteStateB`**. The handler is able to perform a state change task because it has access to the Context through the inherited **`State<TState>`** component.

A Context class **`Context`** is also implemented. The Context class inherits from the **`StateContext<TState, TContext>`** APL component, which injects standard Context functionality. In the example below the **`Context`** class is thus defined in a *curiously recurring template pattern* (Coplien, 1995) setting:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IContextInterface : IContext { void Request(); }

public class Context : StateContext<IMyState, IContextInterface>, IContextInterface { // Using CRTP
    private Context() { }

    public void Request() {
        State.HandleState2();
        State.HandleState1();
        State.HandleState2();
    }
}
```

An instance of the **`Context`** class is then created using a factory on the **`StateContext<TState, TContext>`** component. The factory must be supplied with the Context type and the ConcreteState type, which are used to set the initial state of the Context instance:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var context = Context.Create<Context, MyConcreteStateA>(); // Create context instance using a factory
                                                           // with an initial state of MyConcreteStateA
// Invoke 'Request' on the context instance
context.Contract.Request();

// Change the state of the context to MyConcreteStateA
context.SetState<MyConcreteStateA>();

// Invoke 'Request' on the context instance
context.Contract.Request();

Console.Write("Press any key to exit.");
Console.Read();
```

```
/* Output
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
*/
```

In the code above, a Context instance is created with an initial state of **`MyConcreteStateA`**. The **`Request`** method on the **`context`** instance is then called, whereupon the state is changed to **`MyConcreteStateB`** by one of the handlers. The state is changed to **`MyConcreteStateA`** again and the **`Request`** method on the **`context`** instance is called for the last time. From the output it can be seen that the state processing was handled correctly.

The next example is almost exactly the same as the previous one, except that a singleton (Gamma, Helm, Johnson, & Vlissides, 1994) creational style is used for the ConcreteStates. In this example the **`StateCreationStyle`** enumerator on the **`IState`** State interface is set to **`Singleton`**, informing the internal factories of the APL library that they should treat the ConcreteStates as singletons. The ConcreteState instances now will not hold intrinsic state to the Context any more:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Singleton)] // The state interface is used as a singleton
public interface IState {
    void HandleState1(IStateContext<IState> context); // The context is passed in as an argument
    void HandleState2(IStateContext<IState> context); // The context is passed in as an argument
}

public class ConcreteStateA : IState {
    private ConcreteStateA() { }
    public void HandleState1(IStateContext<IState> context) {
        Console.WriteLine("Calling HandleState1 from state A");
        context.SetState<ConcreteStateB>();
    }

    public void HandleState2(IStateContext<IState> context) {
        Console.WriteLine("Calling HandleState2 from state A");
    }
}

public class ConcreteStateB : IState {
    private ConcreteStateB() { }
    public void HandleState1(IStateContext<IState> context) {
        Console.WriteLine("Calling HandleState1 from state B");
    }

    public void HandleState2(IStateContext<IState> context) {
        Console.WriteLine("Calling HandleState2 from state B");
    }
}
```

The ConcreteStates no longer inherit from the **`State<TState>`** APL component, because they must be shareable and thus cannot hold any intrinsic state, such as the Context. The Context is thus passed to the handler through a method as an argument by means of the **`IStateContext<TState>`** APL interface.

The Context implementation in this example does almost exactly the same as in the previous example, except that the Context instance must be passed to the handlers:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IContextInterface { void Request(); }

public class Context : SingletonStateContext<IState, IContextInterface>, IContextInterface {
    private Context() { }

    public void Request() {
        State.HandleState2(this); // The context is passed as an argument
        State.HandleState1(this); // The context is passed as an argument
        State.HandleState1(this); // The context is passed as an argument
    }
}
```

The client performs the same steps as in the previous example. From the output it can be seen that the handlers were processed correctly:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var context = Context.Create<Context, ConcreteStateA>(); // Create context instance using a factory
                                                         // with an initial state of MyConcreteStateA
context.Contract.Request(); // Invoke 'Request' on the context instance
context.SetState<ConcreteStateA>(); // Change the state of the context to MyConcreteStateB
context.Contract.Request(); // Invoke 'Request' on the context instance
Console.Write("Press any key to exit.");
Console.Read();

/* Output:
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
Calling Handlestate2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
*/
```

The next example shows how the state design pattern can be implemented using a **`Flyweight`** creational style. Once again a State contract is defined, this time with the **`StateCreationStyle`** set to **`Flyweight`**. The Context instance that is passed to the handlers must also be of type **`IFlyweightContext<TState>`**. The **`IFlyweightContext<TState>`** APL interface must be passed to the handlers as an argument, because the interface holds a state transition contract that uses the flyweight pattern:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Flyweight)]
public interface IMyState {
    void HandleState1(IFlyweightContext<IMyState> context);
    void HandleState2(IFlyweightContext<IMyState> context);
}
```

The key used for the Flyweights in the example is an **`enum`** holding five items, as seen below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public enum Setting {
    SettingA,
    SettingB,
    SettingC,
    SettingD,
    SettingE
}
```

A base class **`BaseConcreteState`** is defined that implements the **`IMyState`** State. The **`BaseConcreteState`** class also holds the intrinsic state of the Flyweight, which in this case is exactly the same as the Flyweight key. The **`BaseConcreteState`** base class also implements the non-public constructor that is used by the Flyweight factory:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public abstract class BaseConcreteState : IMyState {
    protected Setting Setting = Setting.SettingA;

    protected BaseConcreteState(Setting setting) { Setting = setting; }
    public abstract void HandleState1(IFlyweightContext<IMyState> context);
    public abstract void HandleState2(IFlyweightContext<IMyState> context);
}
```

Two ConcreteStates are also defined that inherit from the **`BaseConcreteState`** and implement the **`IMyState`** State:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class MyConcreteStateA : BaseConcreteState {
    private MyConcreteStateA(Setting setting) : base(setting) { }

    public override void Handlestate1(IFlyweightContext<IMyState> context) {
        Console.WriteLine("Calling HandleState1 from state A");
        context.SetState<MyConcreteStateB, Setting>(Setting.SettingA);
    }

    public override void HandleState2(IFlyweightContext<IMyState> context) {
        Console.WriteLine("Calling HandleState2 from state A");
    }
}

public class MyConcreteStateB : BaseConcreteState {
    private MyConcreteStateB(Setting setting) : base(setting) { }

    public override void HandleState1(IFlyweightContext<IMyState> context) {
        Console.WriteLine("Calling HandleState1 from state B");
    }

    public override void HandleState2(IFlyweightContext<IMyState> context) {
        Console.WriteLine("Calling HandleState2 from state B");
    }
}
```

The Context class is implemented in the same way as in the two previous examples, except that the Context class inherits from the **`FlyweightContext<TState, TContext>`** APL component, which adds the necessary Flyweight functionality:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IContextInterface { void Request(); }

public class Context : FlyweightContext<IMyState, IContextInterface>, IContextInterface {
    private Context() { }

    public void Request() {
        State.HandleState1(this);
        State.HandleState2(this);
        State.HandleState1(this);
    }
}
```

The client performs the same steps as in the previous two examples. From the output it can be seen that the state handlers were processed correctly:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var context = Context.Create<Context, MyConcreteStateA, Setting>(Setting.SettingD);
context.Contract.Request();
context.SetState<MyConcreteStateA, Setting>(Setting.SettingA);
context.Contract.Request();
Console.Write("Press any key to exit.");
Console.Read();

/* Output
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
*/
```

The next and final example shows the usage of the **`IAutoState<TState>`** APL interface. A State contract is defined that implements the **`IAutoState<TState>`** interface, which is configured with a **`Normal`** creational style. The **`IAutoState<TState>`** APL interface injects a standard set of State functionality with the help of C# extension methods. The ConcreteState implementations of the **`IMyState`** interface thus do not have to inherit from the **`State<TState>`** APL component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[State(StateCreationStyle = StateCreationStyle.Normal]
public interface IMyState : IAutoState<IMyState> { // IAutoState injects state functionality
    void HandleState1();
    void HandleState2();
}
```

Both the ConcreteState implementations, **`MyConcreteStateA`** and **`MyConcreteStateB`**, thus only have to implement the State contract:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public class MyConcreteStateA : IMyState {
    public void HandleState1() {
        Console.WriteLine("Calling HandleState1 from state A");
        this.SetState<MyConcreteStateB, IMyState>();
```

```
    }

    public void HandleState2() {
        Console.WriteLine("Calling HandleState2 from state A");
    }
}

public class MyConcreteStateB : IMyState {
    public void HandleState1() {
        Console.WriteLine("Calling HandleState1 from state B");
    }

    public void HandleState2() {
        Console.WriteLine("Calling HandleState2 from state B");
    }
}
```

The Context in this example does not inherit from a Context APL class. Instead, it implements the **`IAutoStateContext<IState>`** interface, which injects the necessary Context functionality with C# extension methods. This allows the Context class to be combined with other reusable pattern components:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public interface IContextInterface { void Request(); }

public class Context : IAutoStateContext<IMyState>, IContextInterface { // IAutoStateContext injects
                                                                        // context functionality
    public void Request() {
        this.GetState().HandleState2();
        this.GetState().HandleState1();
        this.GetState().HandleState2);
    }
}
```

The Context class in the above example code can also be made a Singleton, where the **`Singleton`** component is used in a *curiously recurring template pattern* (Coplien, 1995) setting:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------

public class Context : Singleton<Context>, IAutoStateContext<IMyState>, IContextInterface {
    private Context() { }
    public void Request() { … }
}
```

The client performs the same steps in this final example as in the previous examples. The client does, however, use the **`AutoStateContextFactory`** APL component in order to create an instance of the Context class.

From the output it can be seen that the handlers were processed correctly:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var contextFactory = new AutoStateContextFactory<MyConcreteStateA, IMyState>();
var context = contextFactory.Create<Context, IContextInterface>();
context.Contract.Request();
context.SetState<MyConcreteStateA>();
context.Contract.Request();
Console.Write("Press any key to exit.");
Console.Read();

/* Output
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
Calling HandleState2 from state A
Calling HandleState1 from state A
Calling HandleState2 from state B
*/
```

10.4 Outcome

The componentization of the state design pattern is a success because it meets all the requirements listed in section 1.4:

- **Completeness**: The state design pattern library components cover all cases described in the original design pattern.

- **Usefulness**: The state design pattern library components are useful because they solve most of the state scenarios desired by a developer. The developer is free to define the state interface as he sees fit and can then use it with the reusable state components. The state plumbing functionality is reusable; a developer is only tasked with implementing the state specific structures and algorithms. The state design pattern library components are relatively easy to understand and to implement.

- **Faithfulness**: The implementation of the state pattern follows the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994).

- **Type-safety**: All of the library components are fully type-safe.

- **Extended applicability**: The state library components cover more cases than the original core state pattern in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994), whereby a State participant can be implemented as a flyweight or a singleton.

- **Performance**: Using the state components does not have a performance impact.

Dyson and Anderson have shown that the state design pattern can be broken up into the following extensions or refinements: state object, state member, pure state, exposed state, state-driven,

transitions owner-driven, transitions and default state (Dyson & Anderson, 1997). The state reusable component can be used to implement all of the above mentioned extensions or refinements, except for the exposed state pattern. The exposed state pattern, however, is a special state pattern where the state interface changes according to the state the Context is in. The exposed state pattern thus should best be solved with dynamic language features. At the heart of the rest of the patterns discusses by *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) and by Dyson and Anderson (Dyson & Anderson, 1997) is a rigid State interface.

The state pattern is fully componentizable because the developer is not tasked with implementing any boiler-plate code when using the reusable pattern component.

The following language features are fundamental to the implementation or usage of the reusable state design pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Attributes (Nagel, Evjen, Glynn, & Watson, 2010), Mixins (Extension Methods) (Esterbrook, 2001) (Jesse & Xie, 2008) and Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005).

Chapter 11

11 COMMAND

11.1 Introduction

The command design pattern decouples strongly related clients from particular behaviours. It makes changes to the participant relationships easier and lessens the complexity of the interfaces.

The command design pattern packages a client request in an object called a command. This allows for different requests for the same command contract. The command objects can be queued or logged and may support undoable operations (Gamma, Helm, Johnson, & Vlissides, 1994).

11.1.1 Structure.

The following figure shows the formal structure of the command design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

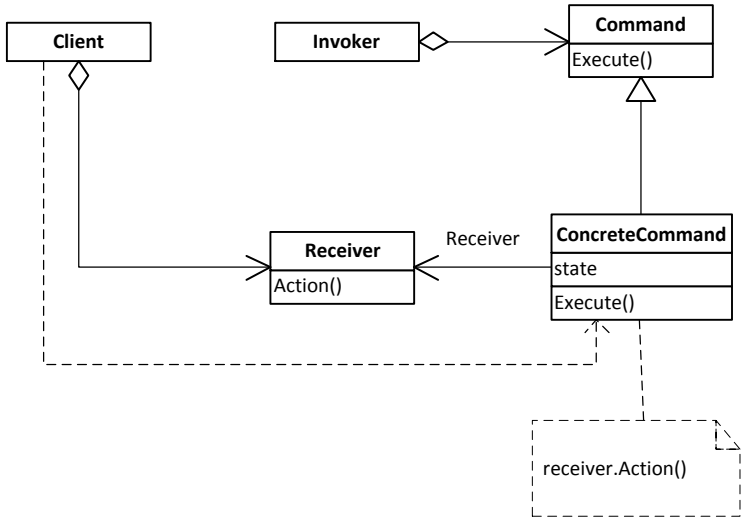

Figure 28. Command structure.

11.1.2 Participants.

The classes and/or objects participating in the command pattern are:

- **Command**

  The Command defines an interface for the command operations or actions.

- **ConcreteCommand**

  A ConcreteCommand implements the operations defined in the Command and is the link between a Receiver object and a command action.

- **Client**

  A Client or user creates, holds and manages a ConcreteCommand object and passes it to a Receiver.

- **Invoker**

  The Invoker directs a Command or queue of Commands to execute a certain action in their interface.

- **Receiver**

  Operations in a ConcreteCommand might delegate all or some of the command actions to an associated Receiver.

11.2 Library Components

11.2.1 The ActionCommand component.

At the heart of the reusable Command component is the **`ICommand`** interface. The APL library defines a number of **`ICommand`** interfaces as seen in the code snippet below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface ICommand { void Execute(); } // No arguments

public interface ICommand<in TArgument> { void Execute(TArgument arg); } // One argument

public interface ICommand<in TArgument1, in TArgument2> { // Two arguments
    void Execute(TArgument1 arg1, TArgument2 arg2);
}

public interface ICommand<in TArgument1, in TArgument2, in TArgument3> { // Three arguments
    void Execute(TArgument1 arg1, TArgument2 arg2, TArgument3 arg3);
}

// … M O R E …
```

Each **ICommand** interface has an **Execute** method that represents the action of the command. Different **ICommand** interfaces are defined with a unique set of arguments that can be passed to its **Execute** method.

Interfaces are also defined in the APL library for undoable Commands, macro Commands and macro undoable Commands (Gamma, Helm, Johnson, & Vlissides, 1994), each with its unique set of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IUndoableCommand : ICommand { void Undo(); }

public interface IUndoableCommand<in TArgument1> : ICommand<TArgument1> {
    void Undo(TArgument1 arg1);
}

public interface IUndoableCommand<in TArgument1, in TArgument2> : ICommand<TArgument1, TArgument2> {
    void Undo(TArgument1 arg1, TArgument2 arg2);
}

// … M O R E …

public interface IMacroCommand : ICommand, IComponent<ICommand> {
    [CompositeMethod]
    new void Execute();
}

public interface IMacroCommand<in TArgument1> : ICommand,
                                                IComponent<ICommand< TArgument1>> {
    [CompositeMethod]
    new void Execute(TArgument1 arg1);
}

// … M O R E …

public interface IMacroUndoableCommand : IUndoableCommand,
                                         IComponent<IUndoableCommand> {
    [CompositeMethod]
    new void Undo();
}

public interface IMacroUndoableCommand<in TArgument1> :
        IUndoableCommand< TArgument1>,
        IComponent<IUndoableCommand< TArgument1>> {
    [CompositeMethod]
    new void Undo();
}

// … M O R E …
```

The macro Commands implement the **IComponent** interface because they use the APL reusable composite components.

The **ActionCommand** APL component is used to create ConcreteCommand instances. The logic of the Receiver that is invoked inside the **Execute** method of a ConcreteCommand is injected with a C# **Action** (Microsoft, 2010a). Multiple reusable implementations for an **ActionCommand** exist in the APL library, one for each corresponding APL **ICommand** interface:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class ActionCommand : ICommand { // No arguments
    protected Action ExecuteReceiver;

    public ActionCommand() { }

    public ActionCommand(Action executeReceiver) { ExecuteReceiver = executeReceiver; }

    public void Execute() {
        if(ExecuteReceiver == null) return;

        ExecuteReceiver();
    }
}

public class ActionCommand<T1> : ICommand<T1> { // One argument
    protected Action<T1> ExecuteReceiver;

    public ActionCommand() { }

    public ActionCommand(Action<T1> executeReceiver) { ExecuteReceiver = executeReceiver; }

    public void Execute(T1 arg1) {
        if(ExecuteReceiver == null) return;

        ExecuteReceiver(arg1);
    }
}

// … M O R E …

public class ActionUndoableCommand : ActionCommand, IUndoableCommand { // No arguments
    protected Action UndoReceiver;

    public ActionUndoableCommand() { }

    public ActionUndoableCommand(Action executeReceiver, Action undoReceiver) : base(executeReceiver) {
        UndoReceiver = undoReceiver;
    }

    public void Undo() {
        if(UndoReceiver == null) return;

        UndoReceiver();
    }
}

public class ActionUndoableCommand<T1> : ActionCommand<T1>, IUndoableCommand<T1> { // One argument
    protected Action<T1> UndoReceiver;

    public ActionUndoableCommand() { }

    public ActionUndoableCommand(Action<T1> executeReceiver, Action<T1> undoReceiver)
        : base(executeReceiver) {
        UndoReceiver = undoReceiver;
    }

    public void Undo(T1 arg1) {
        if(UndoReceiver == null) return;

        UndoReceiver(arg1);
    }
}

// … M O R E …
```

Figure 29 shows a UML class diagram of the **`ActionCommand`** and **`ActionUndoableCommand`** APL components; this figure also depicts the hierarchy and available methods.

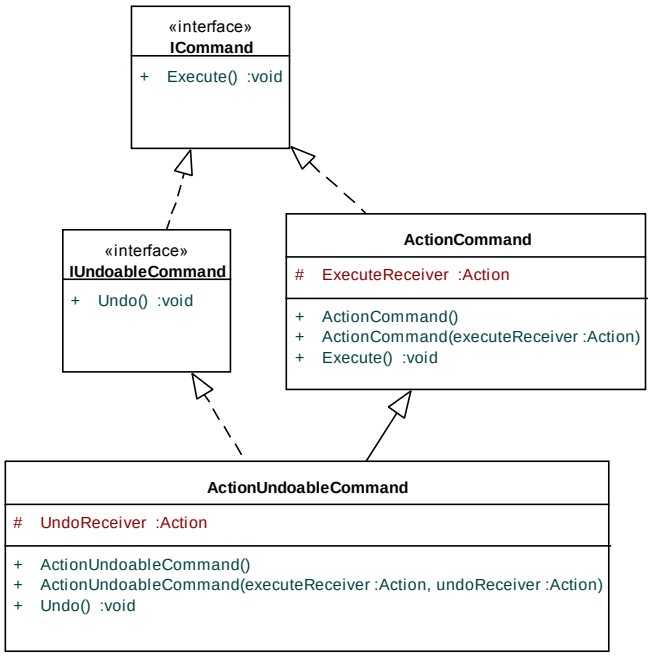

Figure 29. UML class diagram of the ActionCommand and ActionUndoableCommand APL components.

The usage of the **`ActionCommand`** is relatively simple. An **`Action`** that represents the Receiver is supplied during the construction of an **`ActionCommand`**. The **`ActionCommand`** ConcreteCommand instance is then ready to be processed by an Invoker. In the following code snippet, given as an example, the **`Action`** is injected with a lambda expression (Michaelis, 2010). The **`ActionCommand`** ConcreteCommand instance is then processed by an Invoker:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
var concreteCommand = new ActionCommand(() => Console.WriteLine("The command was invoked!"));
invoker.Process(concreteCommand);
```

The usage of the **`ActionUndoableCommand`** APL component takes on an extra undo **`Action`**. The undo **`Action`** tells an instance of the **`ActionUndoableCommand`** component what action to perform when the Command must be undone:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
var concreteCommand = new ActionUndoableCommand(() => ServerSingleton.Instance.ClientConnections++,
                                                () => ServerSingleton.Instance.ClientConnections--);
invoker.Process(concreteCommand);
invoker.Undo(concreteCommand);
```

In the above undoable example the **concreteCommand** is used to increase the number of connections on a hypothetical server. The **concreteCommand** can also be undone because it implements the **IUndoableCommand** APL interface. An undo **Action** is injected with a lambda expression (Michaelis, 2010) during its construction. Calling the **Undo** method on the Invoker invokes the undo injected action logic on the **concreteCommand**.

ActionMacroCommand and **ActionMacroUndoableCommand** components also exist in the APL library, which realize the **IMacroCommand** and **IMacroUndoableCommand** APL interfaces. These action macro Command components are almost exactly the same as the above-mentioned action Command components, except that they allow the ConcreteCommands to exist in a composite environment, as defined in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). The composite pattern is applied on the macro Commands using APL composite components.

11.2.2 The Command component.

The **Command** group of APL components differs from the **ActionCommand** in that the **Command** components are abstract. The components define an abstract **Execute** method that must be overridden in the derived class. The Receiver is also not a C# **Action** (Microsoft, 2010a) but is an **IReceiver** APL interface, as seen in the code below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class Command : ICommand { // No arguments
    protected IReceiver Receiver; // Internal receiver

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(_Receiver!= null, "The receiver  cannot be null");
    };

    protected Command(IReceiver receiver) { Receiver = receiver; } // Constructor with a receiver
    public abstract void Execute(); // Execute the command
}

public abstract class Command<TArgument> : ICommand<TArgument> { // One argument
    protected IReceiver<TArgument> Receiver;
    [ContractInvariantMethod]
    private void ObjectInvariant() { … };
    protected Command(IReceiver<TArgument> receiver) { Receiver = receiver; }
    public abstract void Execute(TArgument arg); // Execute the command
}

public abstract class Command<TArgument1, TArgument2> :
        ICommand<TArgument1, TArgument2> { // Two arguments
    protected IReceiver<TArgument1, TArgument2> Receiver;
    [ContractInvariantMethod]
    private void ObjectInvariant() { … };
    protected Command(IReceiver<TArgument1, TArgument2> receiver) { Receiver = receiver; }
    public abstract void Execute(TArgument1 arg1, TArgument2 arg2); // Execute the command
}

public abstract class Command<TArgument1, TArgument2, TArgument3> :
        ICommand<TArgument1, TArgument2, TArgument3> { // Three arguments
    protected IReceiver<TArgument1, TArgument2, TArgument3> Receiver;
    [ContractInvariantMethod]
```

```
    private void ObjectInvariant() { … };
    protected Command(IReceiver<TArgument1, TArgument2, TArgument3> receiver) {
        Receiver = receiver;
    }

    public abstract void Execute(TArgument1 arg1, TArgument2 arg2, TArgument3 arg3);
}

// … M O R E …
```

A number of abstract **Command** components exist, each with a unique set of arguments. The arguments define the information that must be passed to the **Execute** command method. The **Command** components are more flexible than the **ActionCommand** components, because a developer is free to inject logic in the overridden **Execute** method that has access to the custom state of the Command instance.

The **IReceiver** interface defines the contract of the Receiver. It has an **Action** method that abstracts the action that must be performed by the Receiver. Multiple **IReceiver** interfaces exist, each according to the number of arguments required:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IReceiver { void Action(); } // No arguments

public interface IReceiver<in TArgument> { void Action(TArgument arg); } // One argument

public interface IReceiver<in TArgument1, in TArgument2> { // Two arguments
    void Action(TArgument1 arg1, TArgument2 arg2);
}

public interface IReceiver<in TArgument1, in TArgument2, in TArgument3> { // Three arguments
    void Action(TArgument1 arg1, TArgument2 arg2, TArgument3 arg3);
}

// … M O R E …
```

A number of APL **AutoCommand** components also exist in the APL library, where each one inherits from an abstract APL **Command** component. The **AutoCommand** components are used to define a specific ConcreteCommand. An **AutoCommand** must be constructed with an **IReceiver** interface. The code below shows the implementation of the **AutoCommand** APL component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class AutoCommand : Command { // One argument
    public AutoCommand(IReceiver receiver) : base(receiver) { } // Construction using the receiver
    public override void Execute() { Receiver.Action(); } // Invoking the receiver instance
}

public sealed class AutoCommand<TArgument> : Command<TArgument> { // One arguments
    public AutoCommand(IReceiver<TArgument> receiver) : base(receiver) { }
    public override void Execute(TArgument arg) { Receiver.Action(arg); }
}

public sealed class AutoCommand<TArgument1, TArgument2> :
        Command<TArgument1, TArgument2> { // Two arguments
    public AutoCommand(IReceiver<TArgument1, TArgument2> receiver) : base(receiver) { }
```

```
    public override void Execute(TArgument1 arg1, TArgument2 arg2) { Receiver.Action(arg1, arg2); }
}

public sealed class AutoCommand<TArgument1, TArgument2, TArgument3> :
    Command<TArgument1, TArgument2, TArgument3> { // Three arguments
    public AutoCommand(IReceiver<TArgument1, TArgument2, TArgument3> receiver) : base(receiver) { }

    public override void Execute(TArgument1 arg1, TArgument2 arg2, TArgument3 arg3) {
        Receiver.Action(arg1, arg2, arg3);
    }
}
```

The above code shows that the **Execute** method on the **AutoCommand** delegates its processing to the internal Receiver instance.

AutoUndoableCommand, **AutoMacroCommand** and **AutoUndoableMacroCommand** ConcreteCommand components also exist in the APL library. The **AutoUndoableCommand** component is implemented in the same way as the **AutoCommand** component, except that it also allows for the undoing of commands by realizing the **IUndoableCommand** APL interface. The **AutoMacroCommand** and **AutoUndoableMacroCommand** components can be used with any **ICommand** or **IUndoableCommand** interface respectively. The **AutoMacroCommand** and **AutoUndoableMacroCommand** components reuse the APL composite components by inheriting from the **Composite** APL component.

The code snippet below shows the implementation of the **AutoMacroCommand** and the **AutoUndoableMacroCommand** implementations in the APL library:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class AutoMacroCommand : Composite<ICommand>, IMacroCommand {
    public void Execute() {
        foreach(var component in List) {
            component.GetInterface().Execute();
        }
    }
}

public class AutoUndoableMacroCommand : Composite<IUndoableCommand>, IMacroUndoableCommand {
    public void Execute() {
        foreach(var component in List) {
            component.GetInterface().Execute();
        }
    }

    public void Undo() {
        var commandsReversed = List.ToArray();

        Array.Reverse(commandsReversed);

        foreach(var command in commandsReversed) {
            command.GetInterface().Undo();
        }
    }
}
```

Figure 30 shows a UML class diagram of the **AutoMacroCommand** APL component and its inheritance hierarchy. It shows how the component inherits from the **Component** APL component and realizes the

APL **`IMacroCommand`** interface. Figure 30 also shows that the **`IMacroCommand`** itself realizes the **`ICommand`** APL interface:

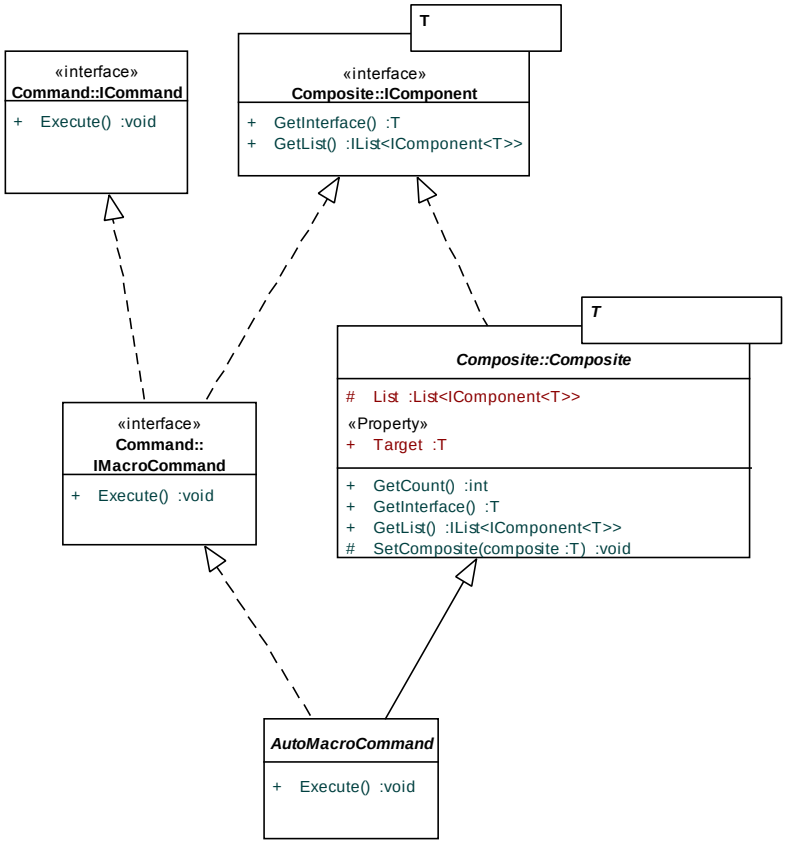

Figure 30. UML class diagram of the AutoMacroCommand APL component.

11.2.3 The Invoker component.

The APL library also has reusable Invokers. It defines an **`ICommandInvoker`** with a contract that is common to most Invokers, as seen below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface ICommandInvoker {
    void Store(ICommand command); // Stores or queues the command in the invoker
    bool Process(); // Processes the next command on the queue
    int Count(); // Returns the number of commands in the queue
    ICommand Peek(); // Returns the next command in the queue without popping it from the queue
    int GetProcessedCount(); // Returns the number of commands processed
}
```

The **`Store`** method registers a Command with the Invoker. The **`Process`** method invokes the next unprocessed command stored in the Invoker. The rest of the methods deliver value added

functionality. For example, the **Peek** method shows what Command will be invoked next, without actually invoking it.

The APL library also defines an **IUndoableCommandInvoker** that is an interface for an Invoker that can perform command rollbacks. The **Undo** method undoes the methods in the same sequence as they were called by the Invoker. The **Redo** method reverses the **Undo** command in the same sequence as they were rolled back. The rest of the methods, once again, define value added functionality:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IUndoableCommandInvoker {
    void Store(IUndoableCommand command);
    bool Process(); // Processes the next command on the queue
    void Undo();    // Undo the next command on the undo stack
    void Redo();    // Redo the next command on the redo stack
    int Count();
    IUndoableCommand Peek();
    int UndoCount();
    IUndoableCommand UndoPeek();
    int RedoCount();
    IUndoableCommand RedoPeek();
}
```

Multiple **ICommandInvoker** and **IUndoableCommandInvoker** interfaces exist, which accommodate the argument needs of the client, as shown below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface ICommandInvoker<TArgument> { // One argument
    void Store(ICommand<TArgument> command);
    bool Process(TArgument arg);
    // … S N I P …
}

public interface ICommandInvoker<TArgument1, TArgument2> { // Two arguments
    void Store(ICommand<TArgument1, TArgument2> command);
    bool Process(TArgument1 arg1, TArgument2 arg2);
    // … S N I P …
}

public interface ICommandInvoker<TArgument1, TArgument2, TArgument3> { // Three arguments
    void Store(ICommand<TArgument1, TArgument2, TArgument3> command);
    bool Process(TArgument1 arg1, TArgument2 arg2, TArgument3 arg3);
    // … S N I P …
}

// … M O R E …

public interface IUndoableCommandInvoker<TArgument> { … } // One argument
public interface IUndoableCommandInvoker<TArgument1, TArgument2> { … } // Two arguments

// … M O R E …
```

Reusable abstract Invokers exist in the APL library, from where most of the concrete Invokers are derived. The **BaseInvoker** and **BaseUndoableInvoker** Invokers define abstract Invokers which implement the basic functionally of most Invokers:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class BaseInvoker : ICommandInvoker {
    protected int ProcessedCount = 0;
    protected IQueue<ICommand> Queue; // Internal command queue

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(_Queue != null, "The queue cannot be null");
    }

    protected BaseInvoker(IQueue<ICommand> queue) { // Constructor
        Queue = queue;
    }

    protected BaseInvoker() { // Constructor that adapts a .Net queue to an IQueue
        Queue = new QueueAdapter<ICommand>();
    }

    public abstract void Store(ICommand command);
    public abstract bool Process();
    public int Count() { return Queue.Count; }
    public ICommand Peek() { return Queue.Peek(); }
    public int GetProcessedCount() { return ProcessedCount; }
}

public abstract class BaseUndoableInvoker : IUndoableCommandInvoker {
    protected int ProcessedCount;
    protected IQueue<IUndoableCommand> Queue; // Internal command queue
    protected Stack<IUndoableCommand> RedoStack; // Redo stack
    protected Stack<IUndoableCommand> UndoStack; // Undo stack

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(Queue != null, "The Queue cannot be null");
        Contract.Invariant(RedoStack!= null, "The RedoStack cannot be null");
        Contract.Invariant(UndoStack!= null, "The RedoStack cannot be null");
    }

    protected BaseUndoableInvoker(IQueue<IUndoableCommand> queue) {
        Queue = queue;
        UndoStack = new Stack<IUndoableCommand>();
        RedoStack = new Stack<IUndoableCommand>();
    }

    public abstract void Store(IUndoableCommand command);
    public abstract bool Process();
    public abstract void Undo();
    public abstract void Redo();
    public int Count() { return Queue.Count; }
    public IUndoableCommand Peek() { return Queue.Peek(); }
    public int UndoCount() { return UndoStack.Count; }
    public IUndoableCommand UndoPeek() { return UndoStack.Peek(); }
    public int RedoCount() { return RedoStack.Count; }
    public IUndoableCommand RedoPeek() { return RedoStack.Peek(); }
    public int GetProcessedCount() { return ProcessedCount; }
}
```

Multiple **`BaseInvoker`** components exist that cater for the number of arguments required by the user:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public abstract class BaseInvoker<TArgument> : // One argument
    ICommandInvoker<TArgument> { … }

public abstract class BaseInvoker<TArgument1, TArgument2> : // Two arguments
```

```
    ICommandInvoker<TArgument1, TArgument2> { … }

public abstract class BaseInvoker<TArgument1, TArgument2, TArgument3> : // Three arguments
    ICommandInvoker<TArgument1, TArgument2, TArgument3> { … }
```

A number of concrete Invokers are defined within the APL library that inherits from the **`BaseInvoker`** component, such as the **`SimpleInvoker`** and **`SimpleUndoableInvoker`** components:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class SimpleInvoker : BaseInvoker { // Reuse from the BaseInvoker
    public SimpleInvoker() { }

    public override bool Process() { // Executes the next command in the queue
        ICommand command = null;

        lock(this) {
            if(Queue.Count > 0) command = Queue.Dequeue();
        }

        if(command != null) {
            command.Execute(); // Execute the command
            return true;
        }

        return false;
    }

    public override void Store(ICommand command) {
        lock(this) {
            Queue.Enqueue(command);
        }
    }
}

public sealed class SimpleUndoableInvoker : BaseUndoableInvoker { // Reuse from the BaseUndoableInvoker
    public SimpleUndoableInvoker() { }

    public override bool Process() { // Executes the next command in the queue
        ICommand command;

        lock(this) {
            command = Queue.Dequeue();
        }

        if(command != null) {
            command.Execute();       // Execute the command
            UndoStack.Push(command); // Push the command unto the UndoStack
            ProcessedCount++;
            return true;
        }

        return false;
    }

    public override void Store(IUndoableCommand command) {
        Contract.Requires<ArgumentNullException>(command != null, "Argument command cannot be null");

        lock(this) {
            Queue.Enqueue(command);
        }
    }

    public override void Undo() {
        if (UndoStack.Count <= 0) return;
```

```
        var command = UndoStack.Pop(); // Pop the command from the UndoStack
        RedoStack.Push(command);       // Push the command unto the RedoStack
        command.Undo();                // Undo the command

        ProcessedCount--;
    }

    public override void Redo() {
        var command = RedoStack.Pop(); // Pop the command from the RedoStack
        UndoStack.Push(command);       // Push the command unto the UndoStack
        command.Execute();             // Execute the command

        ProcessedCount++;
    }
}
```

The above code shows how the simple invokers implement the very basics needed for an Invoker. At the core of the **`SimpleInvoker`** and **`SimpleUndoableInvoker`** components is the **`Process`** method. It retrieves the next Command from the internal queue and invokes the command by using the **`Execute`** method.

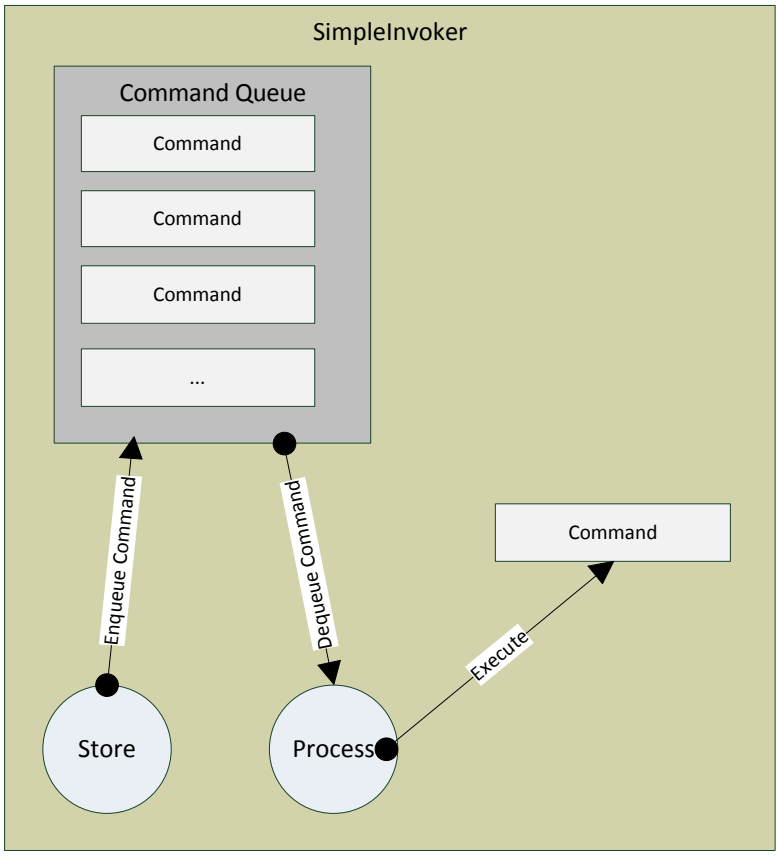

Figure 31. Diagram overviewing a SimpleInvoker APL component.

Figure 31 shows an overview of the **`SimpleInvoker`** APL component. It shows the **`Store`** public method pushing a Command instance into the internal queue. It also shows the **`Process`** public method popping the next Command instance from the internal queue and invoking the **`Execute`** method on it.

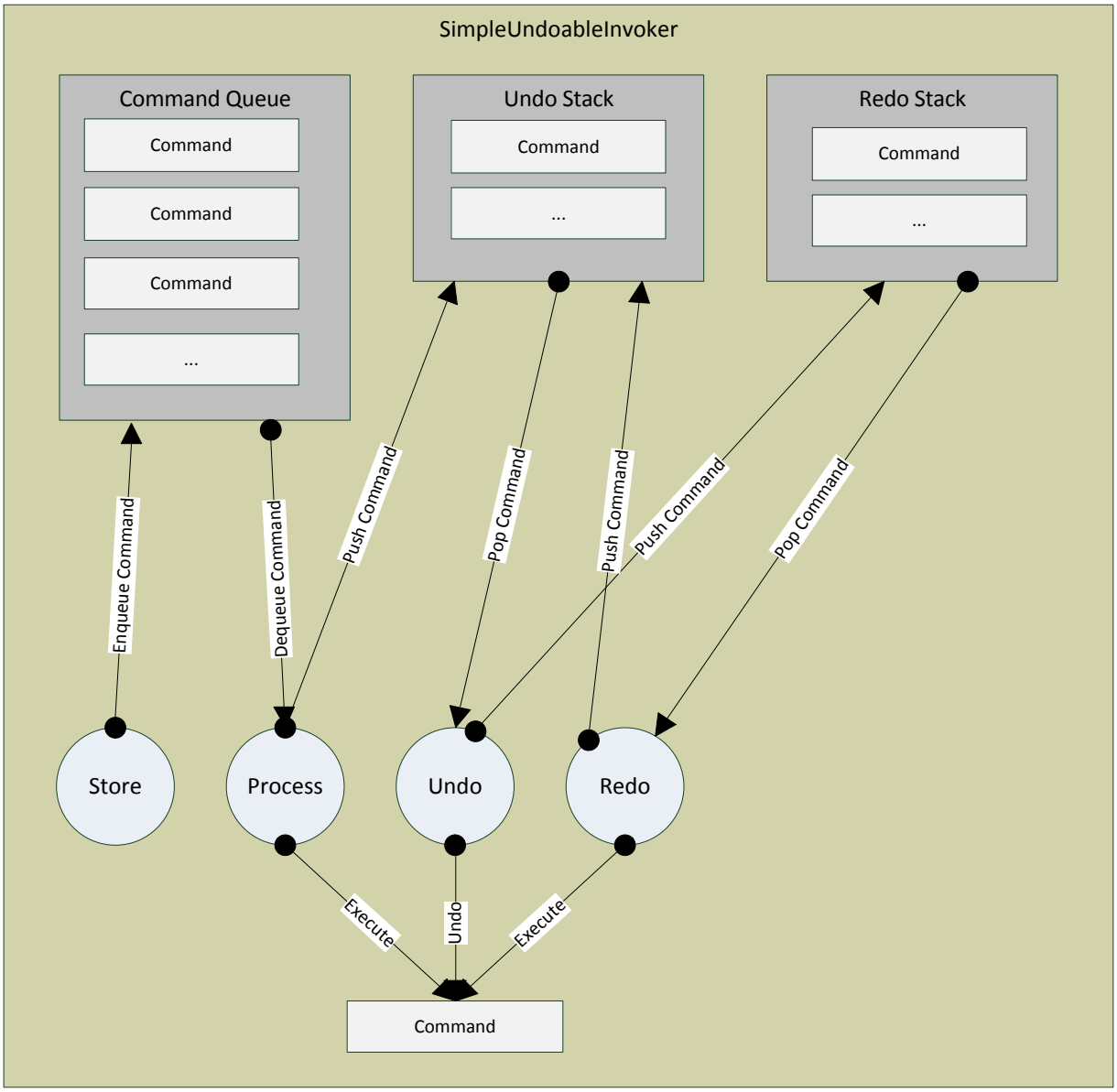

Figure 32. Diagram overviewing a SimpleUndoableInvoker APL component.

The **`SimpleUndoableInvoker`** also has implementations for the **`Undo`** and **`Redo`** methods, using two stacks at the core of its logic. Every time a Command is invoked, it is pushed onto an undo stack. The **`Undo`** method pops the next executed Command from the undo stack and invokes the **`Undo`** contract on the Command, undoing its original Command. The undo method also pushes the Command being

undone onto the redo stack. The **Redo** method pops the next Command from the redo stack and re-executes the Command using the **Execute** method of the Command. The **Redo** method also pushes the executed Command onto the undo stack, in case the client decides to undo the last executed Command. Figure 32 shows an overview of the **SimpleUndoableInvoker** APL component.

Multiple **SimpleInvoker** and **SimpleUndoableInvoker** components exist that implement a different number of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class SimpleInvoker<Arg> :
    BaseInvoker<Arg> { … } // One argument

public sealed class SimpleInvoker<Arg1, Arg2> :
    BaseInvoker<Arg1, Arg2> { … } // Two arguments
// … M O R E …
```

More advanced Invokers also exist in the APL library such as a **BlockingInvoker** and an **AsyncInvoker**. The **BlockingInvoker** uses a blocking queue that implements the producer/consumer pattern (Schmidt & Huston, 2002) (Lea, 1999). The client thus blocks on the **Process** method implemented in the **BlockingInvoker** component. With the **AsyncInvoker**, the **Process** method is invoked asynchronously. The call to the **Process** method thus returns immediately where the invocation on the Command is performed on a background thread, which was allocated from a thread pool.

Auto Invokers also exist in the APL library. With the Invokers discussed so far, the **Process** method must be controlled by the client. The auto Invokers on the other hand take complete control of how and when the **Process** method is invoked. The auto Invokers can be seen as an Invoker server or service. Behind the scene, the auto Invoker implementations use the Invoker components such as the **SimpleInvoker** component or the **BlockingInvoker** component.

The contract of the auto Invoker is simple, focusing on the server or service methodology:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IAutoCommandInvoker { // No arguments
    void Store(ICommand command); // Store a command
    void Run();                   // Start processing commands
    void Stop();                  // Stop processing commands
    int GetProcessedCount();      // Return the number of commands that was processed so far
}

public interface IAutoUndoableCommandInvoker { // No arguments
    void Store(IUndoableCommand command); // Store a command
    void Undo(); // Undo an executed command
    void Redo(); // Redo a rolled back command
    void Run();  // Start processing commands
    void Stop(); // Stop processing commands
    int GetProcessedCount(); // Return the number of commands that was processed so far
}

public interface IAutoCommandInvoker<out TArgument> { // One argument
```

```
    void Store(ICommand<TArgument> command); // Store a command
    void Run();                              // Start processing commands
    void Stop();                             // Stop processing commands
    int GetProcessedCount(); // Return the number of commands that was processed so far
}

public interface IAutoCommandInvoker<out TArgument1, out TArgument2> { // Two arguments
    void Store(ICommand<TArgument1, TArgument2> command); // Store a command
    void Run();              // Start processing commands
    void Stop();             // Stop processing commands
    int GetProcessedCount(); // Return the number of commands that was processed so far
}

// … M O R E …

public interface IAutoUndoableCommandInvoker<out TArgument>  { // One argument
    void Store(IUndoableCommand<TArgument>  command); // Store a command
    void Undo(); // Undo an executed command
    void Redo(); // Redo a rolled back command
    void Run();  // Start processing commands
    void Stop(); // Stop processing commands
    int GetProcessedCount(); // Return the number of commands that where processed so far
}

public interface IAutoUndoableCommandInvoker<out TArgument1, out TArgument2>  { // Two arguments
    void Store(IUndoableCommand<TArgument1, TArgument2>  command); // Store a command
    void Undo();             // Undo an executed command
    void Redo();             // Redo a rolled back command
    void Run();              // Start processing commands
    void Stop();             // Stop processing commands
    int GetProcessedCount(); // Return the number of commands that where processed so far
}

// … M O R E …
```

Once an auto Invoker is started using the **`Run`** method, it processes all Commands automatically until the client decides to stop the processing. Commands stored on a stopped auto Invoker are not processed. The **`AutoInvoker`** APL component is a concrete auto Invoker that realizes the **`IAutoCommandInvoker`** interface. The **`AutoUndoableCommandInvoker`** APL component is a concrete auto Invoker that realizes the **`IAutoUndoableCommandInvoker`** interface.

11.3 Theoretical Examples

The following theoretical example shows all the different permutations in which the Command components can be used:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
class Program {
    static void Main() {

        Console.WriteLine("Normal invoker…:");
        var invoker = new SimpleInvoker();

        // Store a user defined concrete command that implements the ICommand interface
        invoker.Store(new ConcreteCommand1());

        // Store a user defined concrete command that inherits from the Command component
        invoker.Store(new ConcreteCommand2(new Receiver()));

        // Store an AutoCommand with a Receiver
```

```
        invoker.Store(new AutoCommand(new Receiver()));

        // Store an ActionCommand with a lambda expression
        invoker.Store(new ActionCommand(() => Console.WriteLine("Called ActionCommand.Execute()")));
        while(invoker.Process()) { }

        Console.WriteLine();
        Console.WriteLine("Auto invoker…:");
        var autoInvoker = new AutoInvoker();

        // Invoke all three commands using an AutoInvoker
        autoInvoker.Store(new ConcreteCommand1());
        autoInvoker.Store(new ConcreteCommand2(new Receiver()));
        autoInvoker.Store(new AutoCommand(new Receiver()));
        autoInvoker.Store(new ActionCommand(() => Console.WriteLine("Called ActionCommand.Execute()")));
        autoInvoker.Run(); // Runs indefinitely until the autoInvoker is stopped…

        Console.ReadKey();
    }
}

class ConcreteCommand1 :
        ICommand { // User defined concrete command that implements the ICommand interface
    public void Execute() { Console.WriteLine("Called ConcreteCommand1.Execute()"); }
}

class ConcreteCommand2 :
        Command { // Store a user defined concrete command that inherits from the Command component
    public ConcreteCommand2(IReceiver receiver) : base(receiver) { } //
    public override void Execute() {
        Receiver.Action();
        Console.WriteLine("Called ConcreteCommand2.Execute()");
    }
}

class Receiver : IReceiver {
    public void Action() { Console.WriteLine("Called Receiver.Action()"); }
}

/* Output
Normal invoker…:
Called ConcreteCommand1.Execute()
Called ConcreteCommand2.Execute()
Called Receiver.Action()
Called Receiver.Action()
Called ActionCommand.Execute()

Auto invoker…:
Called ConcreteCommand1.Execute()
Called ConcreteCommand2.Execute()
Called Receiver.Action()
Called Receiver.Action()
Called ActionCommand.Execute()
*/
```

In the example above, a **`SimpleInvoker`** is instantiated. It then stores a custom **`ConcreteCommand1`** instance with the Invoker. The **`ConcreteCommand1`** is implemented using the **`ICommand`** APL interface. The **`ConcreteCommand1`** implementation does not use a Receiver. The next Command that is stored on the **`invoker`** is a **`ConcreteCommand2`** instance, which is implemented using the **`Command`** APL component. The **`ConcreteCommand2`** overrides the **`Execute`** method from where it calls the injected **`Receiver`**. Next, an **`AutoCommand`** instance is stored on the Invoker. An instance of **`Receiver`** is registered with the **`AutoCommand`**. Finally, an **`ActionCommand`** is stored on the **`invoker`**. A lambda expression is passed to the

ActionCommand constructor that represents the action of the Command. The **invoker** is then processed in a while loop until all of the Commands have been executed, as seen in the code snippet below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
while(invoker.Process()) { }
```

The next part of the example does basically the same as the first part, except that an **AutoInvoker** is used. The **AutoInvoker** instance is started after all the Commands have been stored on it with the **Run** method, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
autoInvoker.Run(); // Runs indefinitely until the autoInvoker is stopped…
```

The **autoInvoker** blocks indefinitely on the **Run** method until the client stops it in another thread.

The output of the example shows that all the Command instances were invoked successfully.

11.4 Outcome

The componentization of the command design pattern is a success, because it meets all the requirements listed in section 1.4:

- **Completeness**: The command design pattern library components cover all cases described in the original design pattern.

- **Usefulness**: The command design pattern library components are useful because they solve most of the command scenarios desired by a developer. A slight drawback is the fact that the command interface has only one method with a fixed naming convention, namely **"Execute"**. It is thus not possible to use the reusable command pattern if multiple command methods are desired. This situation is, however, rare. A carefully designed command pattern should most often be able to use only one command method (excluding the **undo** and **redo** methods). It is also beneficial to extract the command method into a decoupled interface, because it promotes the decoupling of the ConcreteCommand from the Receiver. A developer has a large choice of implementation combinations from which to choose within the APL library. The **ICommand** interfaces can be used individually in order to create custom ConcreteCommands. The **ActionCommand** and **Command** components can also be used, which gives out-of-the-box ConcreteCommand solutions. Any ConcreteCommand that was created with an **ICommand** can be used with the **Invoker** group of components. The command design pattern library components are easy to understand and simple to use.

- **Faithfulness**: The implementation of the command pattern library components follows the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). The only difference is that, when using the command components, only one command pattern method, namely **`"Execute"`**, is available.

- **Type-safety**: All the library components are fully type-safe.

- **Extended applicability**: The command library components cover more cases than the original core command pattern. The library supplies interfaces such as **`IUndoableCommand`**, **`IMacroCommand`** and **`IMacroUndoableCommand`**. Although macro and undoable commands are discussed in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994), they do not form part of the core pattern. These interfaces are used to implement ConcreteCommand classes that are used in undoable command and macro command scenarios (Gamma, Helm, Johnson, & Vlissides, 1994). ConcreteCommands also exist in the library for these scenarios, such as the **`AutoUndoableCommand`**, **`AutoMacroCommand`** and **`AutoUndoableMacroCommand`** components. Advanced invokers also exist such as the **`SimpleUndoableInvoker`**, **`BlockingInvoker`**, **`AsyncInvoker`**, **`AutoInvoker`** and the **`AutoUndoableCommandInvoker`**.

- **Performance**: Using the command components does not have a performance impact.

The command pattern is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable pattern component.

The following language features are fundamental to the implementation or usage of the reusable command pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003) and Lambda Expressions (Michaelis, 2010)

Chapter 12

12 CHAIN OF RESPONSIBILITY

12.1 Introduction

The chain of responsibility design pattern allows for a certain request to be passed along a chain of related objects or handlers, all implementing the same interface, yet with different behaviours. Each one of the handlers in the chain can either process the request or pass it to the next handler in the chain. The handlers can be added to the chain dynamically during runtime.

The chain of responsibility design pattern decouples the originator of a request from its receiver by giving multiple objects the opportunity of handling a request. A specific request is propagated along the dynamic chain of handlers until one accepts and processes it (Gamma, Helm, Johnson, & Vlissides, 1994).

12.1.1 Structure.

The following figure shows the formal structure of the chain of responsibility design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

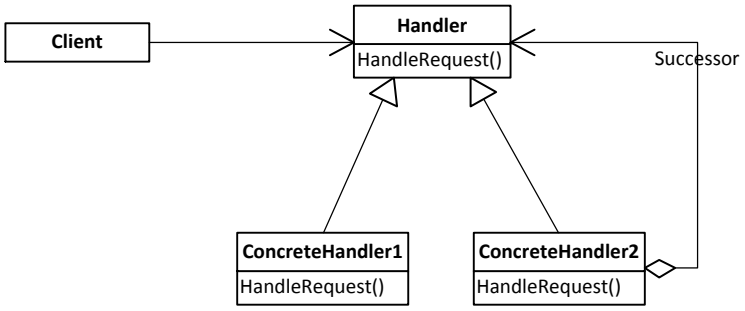

Figure 33. Chain of responsibility structure.

12.1.2 Participants.

The classes and/or objects participating in the chain of responsibility pattern are:

- **Handler**

  The Handler declares an interface for the desired request operations. It might implement the link to the next successor in the chain.

- **ConcreteHandler**

  The ConcreteHandler intercepts requests passed to it through the chain and handles those it is responsible for. If it is not responsible for that request, it forwards the request to its successor. It also holds a reference to the next successor in the chain.

- **Client**

  The Client sends the request to a ConcreteHandler object to which it has a reference.

12.2 Library Components

12.2.1 The DynamicChainOfResponsibility component.

The **`DynamicChainOfResponsibility`** component uses the built in dynamic C# language features. It inherits from the **`DynamicObject`** .NET class (Microsoft, 2011b) (Nagel, Evjen, Glynn, & Watson, 2010), which is a base class for specifying dynamic behaviour (Tratt, 2009) during runtime. The **`DynamicObject`** class enables one to define what operations can be performed on dynamic objects and how to perform those operations. One cannot directly create an instance of the **`DynamicObject`** class because it is abstract (Musser & Stepanov, 1989). To implement the dynamic behaviour, one can inherit from the **`DynamicObject`** class and override necessary methods. For example, if only operations for setting and getting properties are needed, one can override just the **`TrySetMember`** and **`TryGetMember`** methods. The following code shows the implementation of the **`DynamicChainOfResponsibility`** APL component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class DynamicChainOfResponsibility : DynamicObject {
    private readonly Dictionary<string, object> _members = new Dictionary<string, object>();
    private DynamicChainOfResponsibility _successor;

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(_members != null, "The members cannot be null");
    }

    public DynamicChainOfResponsibility() { }

    public DynamicChainOfResponsibility(DynamicChainOfResponsibility successor) {
        _successor = successor;
    }
```

```
    public void SetSuccessor(DynamicChainOfResponsibility successor) { _successor = successor; }

    public override bool TrySetMember(SetMemberBinder binder, object value) {
        Contract.Requires<ArgumentNullException>(binder != null, "Argument binder cannot be null");
        Contract.Requires<ArgumentNullException>(value!= null, "Argument value cannot be null");

        if(!_members.ContainsKey(binder.Name))
            _members.Add(binder.Name, value);
        else
            _members[binder.Name] = value;

        return true;
    }

    public override bool TryGetMember(GetMemberBinder binder, out object result) {
        Contract.Requires<ArgumentNullException>(binder != null, "Argument binder cannot be null");
        if(_members.ContainsKey(binder.Name)) {
            result = _members[binder.Name];
            return true;
        }

        return base.TryGetMember(binder, out result);
    }

    public override bool TryInvokeMember(InvokeMemberBinder binder,
                                         object[] args,
                                         out object result) {
        Contract.Requires<ArgumentNullException>(binder != null, "Argument binder cannot be null");

        ChainOfResponsibilityEx.Handled = false;
        var executedHandler = false;
        result = null;

        // Invoke the handler if it is present
        if(_members.ContainsKey(binder.Name) &&
            _members[binder.Name] is Delegate) {
            result = ((Delegate)_members[binder.Name]).DynamicInvoke(args);
            executedHandler = true;
        }

        // If no handler exists or the handler did not handle the request,
        // then pass it on to the successor if it exists
        if(!ChainOfResponsibilityEx.Handled && _successor != null) {
            return _successor.TryInvokeMember(binder, args, out result);
        }

        return executedHandler;
    }

    public override IEnumerable<string> GetDynamicMemberNames() { return _members.Keys; }
}
```

The **DynamicChainOfResponsibility** component uses the dynamic language features of C# (Microsoft, 2011b). Thus any Handler method can be registered with an instance of **DynamicChainOfResponsibility**, as shown in the example below:

```
C# (APL Example)
-------------------------------------------------------------------------------------------------------
handler1.HandleChar = new Action<char>(x => { // Dynamically add the 'HandleChar'
                                              // method to the handler1 instance
    if(x != 'X') return;
    Console.WriteLine("I am X");
    ChainOfResponsibilityEx.SetHandled();
});
```

In the example above, a new method **HandleChar**, with one argument, is registered dynamically with the **handler1** instance, which is an instance of **DynamicChainOfResponsibility**. The **SetHandled** method on the static **ChainOfResponsibilityEx** APL static helper class is used to notify the **DynamicChainOfResponsibility** component that the request was handled successfully. The **SetHandled** method uses a thread static flag, which is defined by the **_handled** field. The **ThreadStatic** (Microsoft, 2010n) attribute on the **_handled** field tells the runtime that a unique instance of the field must exist per thread:

```
static public class ChainOfResponsibilityEx {
    [ThreadStatic]
    private static bool _handled;
    public static bool Handled { get { return _handled; } set { _handled = value; } }
    public static void SetHandled() { Handled = true; }
}
```

The **_handled** field can thus be safely used by the internals of the **DynamicChainOfResponsibility** component in order to check whether the Handler handled the request:

```
// If no handler exists or the handler did not handle the request,
// then pass it on to the successor if it exists
if(!ChainOfResponsibilityEx.Handled && _successor != null) {
    return _successor.TryInvokeMember(binder, args, out result);
}
```

The new **HandleChar** method can now be used by the client, as shown below:

```
C# (APL Example)
-------------------------------------------------------------------------------------------------------
handler1.HandleChar('C'); // Invoke the 'HandleChar' method, which was dynamicly added…
```

In the example above, the **DynamicChainOfResponsibility** component checks whether the **HandleChar** method actually exists. If it does, then it is invoked. If it is not found, or if it was not handled, then the successor is invoked, which is also an instance of **DynamicChainOfResponsibility**. All this logic is processed in the **TryInvokeMember** method implemented on the **DynamicChainOfResponsibility** component, which is an abstract method in the **DynamicObject** base class (Nagel, Evjen, Glynn, & Watson, 2010). The **TryInvokeMember** method routes the invocation to the Handler, if it is present. The **TryInvokeMember** method tests whether the Handler is present by searching for the method in the internal **_members** dictionary. If no Handler exists or the Handler did not handle the request, then the request is passed to a successor, if one exists. The successor itself is just another instance of the **DynamicChainOfResponsibility** component that can be registered with its constructor or with the **SetSuccessor** method, as seen below:

```
C# (APL)
-------------------------------------------------------------------------------------------------------
public class DynamicChainOfResponsibility : DynamicObject {
    // … S N I P …
    public DynamicChainOfResponsibility(DynamicChainOfResponsibility successor) {
        _successor = successor;
```

```
    }

    public void SetSuccessor(DynamicChainOfResponsibility successor) {
        _successor = successor;
    }
    // … S N I P …
}
```

12.3 Theoretical Examples

The following example shows the usage of the **`DynamicChainOfResponsibility`** component. The example uses the **`DynamicChainOfResponsibilityFactory`** APL component with which to create **`DynamicChainOfResponsibility`** instances. The **`DynamicChainOfResponsibilityFactory`** component creates **`DynamicChainOfResponsibility`** instances with a registered default Handler:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
dynamic defaultHandler = new DynamicChainOfResponsibility(); // Create an instance of the
                                                             // DynamicChainOfResponsibility component
                                                             // Note the 'dynamic' C# keyword

// Dynamically add a new 'HandleRequest' method to the defaultHandler object
defaultHandler.HandleRequest = new Action<int>(x => {
                Console.WriteLine("Default.");
                ChainOfResponsibilityEx.SetHandled(); });

// Create a factory with the above default handler
var factory = new DynamicChainOfResponsibilityFactory(defaultHandler);

// Use the factory to create a handler
// Dynamically add a new 'HandleRequest' method to the defaultHandler object
dynamic handler1 = factory.Create();
handler1.HandleRequest = new Action<int>(x => {
    if(x < 0 || x >= 10) return;
    Console.WriteLine("h1 handled request {0}", x);
    ChainOfResponsibilityEx.SetHandled(); });

// Use the factory to create a handler
// Dynamically add a new 'HandleRequest' method to the defaultHandler object
dynamic handler2 = factory.Create();
handler2.HandleRequest = new Action<int>(x => {
    if(x >= 10 && x < 20) return;
    Console.WriteLine("h2 handled request {0}", x);
    ChainOfResponsibilityEx.SetHandled(); });

// Use the factory to create a handler
// Dynamically add a new 'HandleRequest' method to the defaultHandler object
dynamic handler3 = factory.Create();
handler3.HandleRequest = new Action<int>(x => {
    if(x >= 20 && x < 30) return;
    Console.WriteLine("h3 handled request {0}", x);
    ChainOfResponsibilityEx.SetHandled(); });

// Set some successors
handler1.SetSuccessor(handler2);
handler2.SetSuccessor(handler3);

// Process the request
int[] requests = { 2, 5, 14, 22, 18, 3, 27, 20 };
foreach(int request in requests) { handler1.HandleRequest(request); }

/* Output
h1 handled request 2
h1 handled request 5
```

```
h3 handled request 14
h2 handled request 22
h3 handled request 18
h1 handled request 3
h2 handled request 27
h2 handled request 20
*/
```

Handlers are registered with the component instances during runtime, using C# dynamics (Microsoft, 2011b). The following code shows how a **`HandleRequest`** method with one argument is registered:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
// Dynamicaly add a new 'HandleRequest' method to the defaultHandler object
handler1.HandleRequest = new Action<int>(x => {
    if(x < 0 || x >= 10) return; // Return if not handled
    Console.WriteLine("h1 handled request {0}", x); // Handle the request
    ChainOfResponsibilityEx.SetHandled(); // Notify that the request was handled
});
```

The logic of the Handler method is injected using lambda expressions (Microsoft, 2010i). The client can call the **`HandleRequest`** like any other method after registering the method, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
handler1.HandleRequest(request); // Invoke the 'HandleRequest' just like any other method that
                                 // is avaliable on the handler1 object.
                                 // The 'HandleRequest' method was added to the handler1 object during
                                 // runtime
```

When a request is sent to an instance of **`DynamicChainOfResponsibility`**, it determines whether that specific method was registered with it. If not, it passes the request on to the successor.

The output of the example shows that the correct Handler was called for each request by the client.

12.4 Outcome

The componentization of the chain of responsibility design pattern is a success because it meets all the requirements listed in section 1.4:

- **Completeness**: The chain of responsibility design pattern library component cover all cases described in the original design pattern.

- **Usefulness**: The chain of responsibility design pattern library component is useful because it solves exactly the same chain of responsibility design pattern intent as an implementation written by hand. With the **`DynamicChainOfResponsibility`**, ConcreteHandler algorithms are hooked up with the component using C# 4.0 dynamic language features. With the reusable component, a developer is not tasked with writing any chain of responsibility boiler plate code. The component is easy to use and easy to understand.

- **Faithfulness**: The `DynamicChainOfResponsibility` component follows a certain chain of responsibility variant described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994), where the usage of dynamic language features is mentioned. The `DynamicChainOfResponsibility` component, however, solves the same intent as that of the chain of responsibility pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994).

- **Type-safety**: The `DynamicChainOfResponsibility` component is not type-safe. It is, however, the explicit intent of the component to be dynamic, in order to implement a chain of responsibility pattern solution.

- **Extended applicability**: The chain of responsibility library component does cover more cases than the original core chain of responsibility pattern. The `DynamicChainOfResponsibility` component is a special implementation of the pattern in which dynamic language features are used. The chain of responsibility library component, however, does not follow the original core chain of responsibility implementation. The `DynamicChainOfResponsibility` component, however, solves the same intent as the pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994).

- **Performance**: Using the chain of responsibility component does have a performance impact. Using dynamically typed features is typically slower than using statically typed implementations. Appendix III shows that the `TryInvokeMember` method defined on the `DynamicChainOfResponsibility` component will incur a large performance penalty. There is, however, successful research into making dynamically typed implementations as fast as statically typed implementations (Cuni, Ancona, & Rigo, 2009). Nevertheless, the performance penalty incurred when using the `DynamicChainOfResponsibility` component is acceptable in normal situations.

The chain of responsibility pattern is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable pattern component.

The following language features are fundamental to the implementation or usage of the reusable chain of responsibility pattern components: Inheritance (Mitchell, Mitchell, & Krzysztof, 2003), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010), Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005) and Dynamic Typing (Tratt, 2009).

Chapter 13

13 MEMENTO

13.1 Introduction

In certain situations there is a need to store the internal state of an object in order for it to be restored back to a previous state by a user or client.

The memento design pattern extracts and externally stores an object's internal state in order to restore it back to its original state sometime in the future, without violating encapsulation (Gamma, Helm, Johnson, & Vlissides, 1994).

13.1.1 Structure.

The following figure shows the formal structure of the memento design pattern (Gamma, Helm, Johnson, & Vlissides, 1994):

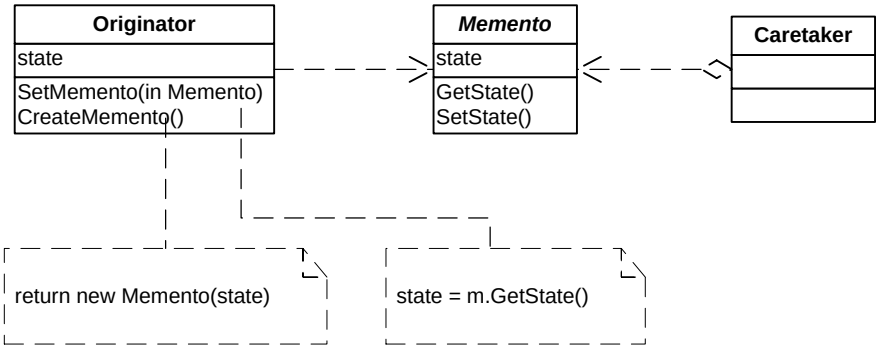

Figure 34. Memento structure.

13.1.2 Participants.

The classes and/or objects participating in the memento pattern are:

- **Memento**

  The Memento extracts the internal state of the Originator object and stores it locally. The Memento may store the entire internal state of the Originator or only a subset thereof. The stored internal state is protected against access by foreign objects and only the Originator can access it. Mementos have two interfaces. The Caretaker sees a narrow interface to the

Memento. The Originator, in contrast, sees a wide interface to the Memento that lets it access all the data necessary to restore itself to its previous state. If possible, only the Originator that creates the Memento has access to the Memento's internal state.

- **Originator**

  The Originator instantiates a Memento instance by encapsulating a copy of its own recent internal state. The Originator is also capable of restoring its internal state using the Memento.

- **Caretaker**

  The Caretaker has custody over the Memento's existence. However, it will not use or read the contents of a Memento or use any of its functionality.

13.2 Library Components

13.2.1 The Memento group of components.

The **`Originator<TOriginator>`** generic APL component is a reusable Originator that takes in the hand coded part of the Originator as a generic argument. It uses the hand coded Originator to make a copy of its internal state and pass it on to the Memento:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class Originator<TOriginator> : IOriginator<TOriginator> {
    private readonly TOriginator _originator;
    private readonly MementoRestore<TOriginator> _restore;

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(_originator != null, "The originator cannot be null");
        Contract.Invariant(_restore != null, "The restore cannot be null");
    }

    public Originator(TOriginator originator, MementoRestore<TOriginator> restore)  {
        _originator = originator;
        _restore = restore;
    }

    public IMemento<TOriginator> CreateMemento()  {
        Contract.Ensures(Contract.Result<IMemento<TOriginator>>() != null);
        var memento = GetMemento();
        memento.SnapshotState = _originator.DeepCopy(); // Make a copy
        return memento;
    }

    private Memento<TOriginator> GetMemento() {
        Contract.Ensures(Contract.Result<IMemento<TOriginator>>() != null);
        return new Memento<TOriginator>(_restore);
    }

    public void SetState(IMemento<TOriginator> memento) {
        Contract.Requires<ArgumentNullException>(memento != null, "Argument memento cannot be null");
        memento.RestoreState(_originator);
```

```
    }

    public static Originator<TOriginator> Create(TOriginator originator,
                                                 MementoRestore<TOriginator> set) {
        Contract.Requires<ArgumentNullException>(originator != null,
                                                 "Argument originator cannot be null");
        Contract.Requires<ArgumentNullException>(set != null, "Argument set cannot be null");
        Contract.Ensures(Contract.Result<Originator<TOriginator>>() != null);
        return new Originator<TOriginator>(originator, set);
    }
}
```

The APL prototype (Gamma, Helm, Johnson, & Vlissides, 1994) reusable component is used to make a copy of the Originator's internal state, as seen in the **`CreateMemento`** method:

```
C# (APL)
-------------------------------------------------------------------------------------------------------------
memento.SnapshotState = _originator.DeepCopy(); // Make a copy
```

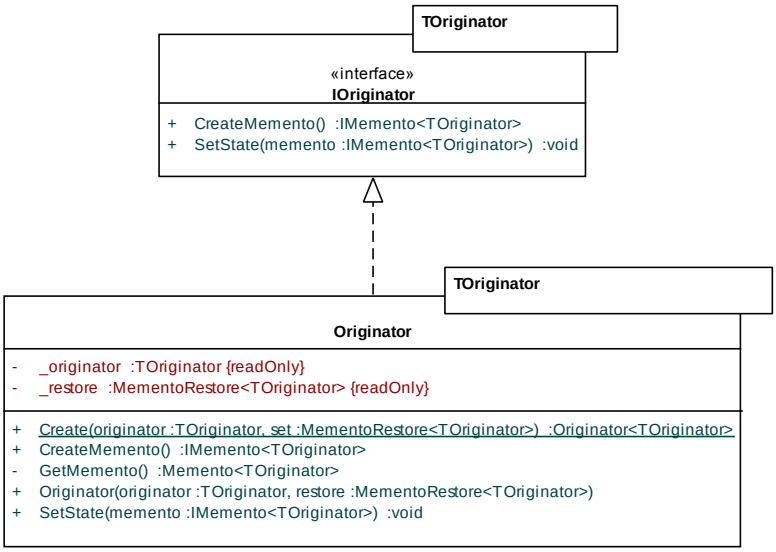

Figure 35. UML class diagram of the Originator APL component.

Figure 35 shows a UML class diagram of the **`Originator`** APL component. It shows the component's realization of the **`IOriginator`** APL interface and also the available methods on the **`Originator`** component. Figure 35 also shows the internal state of the **`Originator`** component where it holds an instance of a **`TOriginator`** and a **`MementoRestore`** delegate.

The **`Originator<TOriginator>`** component implements the **`IOriginator<TOriginator>`** interface that defines a contract for a standard Originator:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IOriginator<TOriginator> {
    IMemento<TOriginator> CreateMemento();
    void SetState(IMemento<TOriginator> memento);
}
```

The **CreateMemento** method creates a new Memento in which to store a snapshot of the Originator's internal state. The Memento is represented as an **IMemento<TOriginator>** interface that defines methods which manipulate the state of an Originator:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IMemento<in TOriginator> {
    TOriginator SnapshotState { set; }
    void RestoreState(TOriginator originator);
}
```

The **GetMemento** private method on the **Originator<TOriginator>** is a simple factory that returns a new instance of the **Memento<TOriginator>** component. The **Memento<TOriginator>** class is a generic reusable APL component that implements the **IMemento<in TOriginator>** interface. The Memento is used to set the state of the Originator back to its original state. The state of the Originator is probably not publicly accessible. In order for the **Memento<TOriginator>** component to set the state back, a generic delegate **MementoRestore** instance is passed to the **Originator<TOriginator>** component, which in turn passes it to the **Memento<TOriginator>** component. An instance of the **MementoRestore** delegate must have access to the internal state of the Originator. The **MementoRestore** delegate has two arguments which are the original Originator and a snapshot of the Originator:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate void MementoRestore<in TOriginator>(TOriginator originator, TOriginator snapshot);
```

In the code below, the **Restore** method on the **ClientOriginator** example class is an example implementation for the **MementoRestore** delegate and is used to set the state of the Originator back to its original state:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
public static void Restore(ClientOriginator originator, ClientOriginator snapshot) {
    Contract.Requires<ArgumentNullException>(originator != null, "Argument originator cannot be null");
    Contract.Requires<ArgumentNullException>(snapshot != null, "Argument snapshot cannot be null");
    originator._state = snapshot._state;
}
```

In the above example the **Restore** method is defined on the **ClientOriginator** class and thus has access to its own private state. An instance of the **MementoRestore** delegate on the other hand has access only to the specific restore method to which it is linked and thus has no access to the Originator's private state.

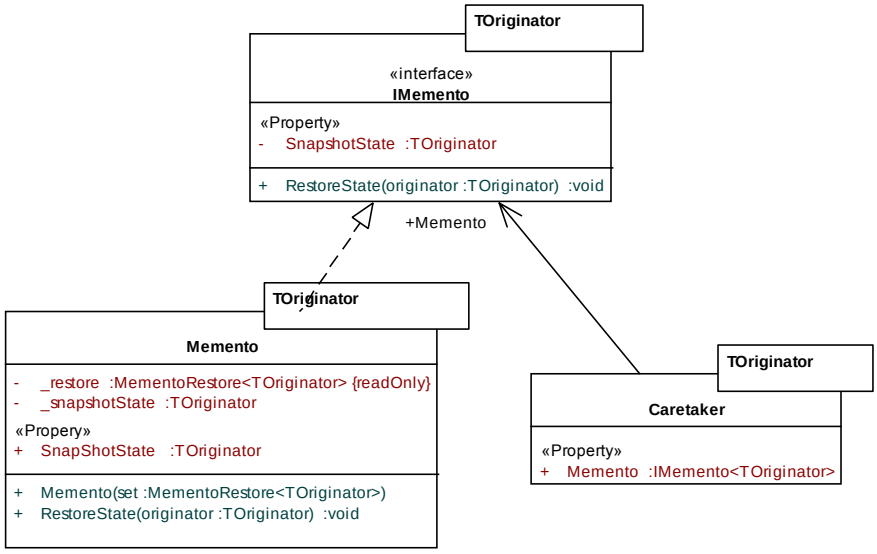

Figure 36. UML class diagram of the Memento APL component.

Figure 36 shows a UML class diagram of the **`Memento`** and **`Caretaker`** APL components. It shows the **`Memento's`** realization of the **`IMemento`** interface and the **`Caretaker's`** reference to, and usage of, an **`IMemento`**.

The **`SetState`** method implemented on the **`Originator`** component sets the state of the Originator back to its original state, using the supplied Memento component instance:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public void SetState(IMemento<TOriginator> memento) {
    Contract.Requires<ArgumentNullException>(memento != null, "Argument memento cannot be null");
    memento.RestoreState(_originator);
}
```

The **`Create`** method on the **`Originator`** component is a basic factory (Freeman, Robson, Bates, & Sierra, 2004), which is used to create a new instance of the **`Originator`** component using the given arguments.

The **`Memento<TOriginator>`** generic APL component stores a snapshot of an instance of type **`TOriginator`**. It also holds an instance of the **`MementoRestore`** delegate which is used to set the Originator back to its original state. The **`MementoRestore`** instance must be supplied on construction with the **`Memento<TOriginator>`** component:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class Memento<TOriginator> : IMemento<TOriginator> {
    private TOriginator _snapshotState;
    private readonly MementoRestore<TOriginator> _restore;

    [ContractInvariantMethod]
    private void ObjectInvariant() {
        Contract.Invariant(_restore != null, "The restore cannot be null");
    }

    public Memento(MementoRestore<TOriginator> restore) { _restore = restore; }
    public TOriginator SnapshotState { `
        set {
            Contract.Requires<ArgumentNullException>(value != null,
                                                     "Argument SnapshotState cannot be null");
            _snapshotState = value;
        }
    }

    public void RestoreState(TOriginator originator) {
        Contract.Requires<ArgumentNullException>(originator!= null,
                                                  "Argument originator cannot be null");
        _ restore(originator, _snapshotState); // Restore from the snapshot using the
                                               // MementoRestore<TOriginator> delegate that
                                               // was supplied by the user
    }
}
```

The **`RestoreState`** public method on the **`Memento`** component is used to restore the state of the Originator back to the original snap shot. The method must be supplied with the current Originator, which is passed to an instance of the **`MementoRestore`** delegate together with the snap shot of the Originator's previous state. The **`MementoRestore`** delegate will restore the state of the Originator back to the state of the snap shot.

13.3 Theoretical Examples

The following example shows a theoretical usage of the memento reusable component. First, a **`ClientOriginator`** class is defined; this is the actual object whose state is going to be stored and then eventually restored:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
[Serializable] // Must be Serializable in order to perform the deep copy
class ClientOriginator {
    private string _state;

    public void SwitchOff() { _state = "On"; }
    public void SwitchOn() { _state = "Off"; }

    public void PrintState() { Console.WriteLine(_state);}

    // Restore the state back to the snapshot state
    public static void Restore(ClientOriginator originator, ClientOriginator snapshot) {
        originator._state = snapshot._state;
    }
}
```

The **ClientOriginator** has no concept of a Memento. This is managed by the reusable APL **Originator** component. The **Originator** component makes a copy of the **ClientOriginator** instance's state, using the APL **Memento** component:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
var clientOriginator = new ClientOriginator();
var originator = new Originator<ClientOriginator>(clientOriginator, ClientOriginator.Restore);
clientOriginator.SwitchOn();
clientOriginator.PrintState();

// (1) Store state

// Create a new Caretaker
var caretaker = new Caretaker<ClientOriginator>();

// Set the Memento on the Caretaker using the Originator
caretaker.Memento = originator.CreateMemento();

// (2) Change state

// Call SwitchOff changing the state on the ClientOriginator instance
clientOriginator.SwitchOff();

// Show the new state
clientOriginator.PrintState();

// (3) Restore state
originator.SetState(caretaker.Memento);

// Show the state
clientOriginator.PrintState();

/* Output
Off
On
Off
*/
```

The **Restore** static method defined on the **ClientOriginator** class is registered through the **Originator** constructor in order for the Originator instance to make a copy of the internal state of a **clientOriginator** instance. The **Restore** method must follow the contract of the **MementoRestore<TOriginator>** delegate defined in the APL library. The **Restore** static method thus manages how the internal state of the **ClientOriginator** is restored.

From the output, it can be seen that the final state of the **clientOriginator** object is restored back to its original state via the **originator** object, which is an instance of the APL **Originator** component.

13.4 Outcome

The componentization of the memento pattern is a success, because it meets all the requirements listed in section 1.4:

- **Completeness**: The memento library components cover all cases described in the original memento design pattern.

- **Usefulness**: The memento library components are useful because they solve exactly the same intent as a memento implementation written by hand. The components are easy to use and easy to understand.

- **Faithfulness**: The implementation of the memento pattern is slightly different from the original pattern described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). A delegate is used to access and set the internal state of an Originator. Bishop has shown that patterns can be implemented in different ways depending upon the available language features (Bishop, 2007). Thus, even if the memento pattern were to be implemented by hand in C# 4.0, using a delegate to access the internal state of an Originator is acceptable.

- **Type-safety**: The memento library components are fully type-safe.

- **Extended applicability**: The memento library components do not cover more cases than the original memento pattern.

- **Performance**: The memento components use the prototype component in order to make a clone of the hand written Originator. Internally the prototype component uses serialization for cloning, which will always be slower than a hand developed algorithm. Serialization is however widely used in APIs such as WCF and ORM tools, within the context of transactional applications, where its performance overhead is deemed to be acceptable.

The memento pattern is fully componentizable because the developer is not tasked with implementing any boiler plate code when using the reusable pattern component.

The following language features are fundamental to the implementation or usage of the reusable memento pattern components: Interfaces (Pattison & Box, 2000), Generics (Jagger, Perry, & Sestoft, 2007), Design by Contract™ (Mitchell & McKim, 2001), Method References (Microsoft, 2010e), Anonymous Functions (Ierusalimschy, 2003), Lambda Expressions (Michaelis, 2010) and Reflection (Sobel & Friedman, 1996) (Forman & Forman, 2005).

Chapter 14

14 EXISTING REUSABLE PATTERN LIBRARIES

14.1 Prototype

Static programming languages (Pierce, 2002) such as Java, C++, C# and Delphi are quite rigid. With static programming languages the behaviour of an object is defined by a class and that behaviour can be changed only by sub-classing. Prototype-based programming languages such as Self (Chambers, 1992) and JavaScript (David, 2006) do not introduce the concept of a class. Instead they supply only objects, but enable one to add services and attributes dynamically during run-time. In Self, new objects are created solely from cloning. Thus, the root object is cloned and that clone can evolve over time, generating further clones with different services and attributes.

As an example, the following Self code makes a copy of the **`account`** object and sends it a message to put 5000 into the slot called **`value`**:

```
Self
----------------------------------------------------------------------------------------------------
account copy value: 5000
```

The .NET framework does supply a cloning operation **`MemberwiseClone`** (Microsoft, 2010p) on all objects. The **`MemberwiseClone`** implements a shallow copy on the calling object. It thus does not implement a full prototype of an object if that object references other non-primitive types (Binder, 1999). The .NET framework also supplies an **`IClonable`** interface (Microsoft, 2010q). A class implementing the **`IClonable`** interface must implement a **`Clone`** method that returns an object. A drawback of the **`IClonable`** interface is that it is not clear whether the **`Clone`** method will do a shallow or deep copy of the current object and it is thus ambiguous (Abrams, 2004).

Copyable is a dedicated framework (Stranden, 2011), written in C#, which offers a reusable C# prototype component for cloning .NET objects. A major advantage with the *Copyable* framework is that the cloned .NET objects do not have to be attributed with the serializable attribute (Albahari & Albahari, 2007).

In Python, the library's **`copy`** module provides a **`deepcopy`** method (van Rossum, 2008) that returns a clone of the current object. Developers may define a special method, **`__deepcopy__`**, on an object in order to provide a custom cloning implementation.

In Smalltalk the **`Object`** class has a method, **`deepCopy`**, which is available to all objects via inheritance. The **`deepCopy`** method makes a deep copy, and thus a clone, of the current object (Alpert, Brown, & Woolf, 1998).

In Eiffel a **`deep_clone`** method, which is defined in the Kernel Library, is available to all classes (Thomas & Weedon, 1997), where it performs a deep clone on the current object. The prototype design pattern is thus part of the Eiffel language where it is implemented in the standard library (Arnout, 2004).

14.2 Singleton

Arnout has shown that it is not possible to create a reusable singleton in Eiffel (Arnout & Bezault, 2004) because of the lack of certain language features.

The *Unity* dependency injection container framework (Microsoft, 2010o), which is part of the *Patterns & Practices* project from Microsoft, has a mechanism for acquiring a single instance for a registered type, as do virtually all dependency injection (Fowler, 2004) containers. With dependency injection (Fowler, 2004) an independent object, which is usually called an assembler, populates the state in a certain instance of a class with an appropriate predefined implementation for the interfaces referenced in that class.

Windows Communication Foundation (WCF) also offers a singleton service (Lowy, 2007). WCF offers an integrated development environment for building service-oriented systems that communicate over the web and the enterprise (Bustamante, 2007). WCF is part of the .NET Framework. When a service is set as a singleton, all client messages are channelled to that same single instance. The singleton service lives indefinitely; it is only destroyed once the host process is killed. The singleton service instance is created only once, when the host is created.

Schmidt has created a generic class that implements the singleton design pattern in the ACE (Adaptive Communication Environment) C++ library (Schmidt, Stal, Rohnert, & Buschmann, 2000). The reusable singleton C++ class uses generics in order to turn ordinary C++ classes into singletons optimised with the double-checked locking optimisation pattern (Schmidt & Harrison, 1996). A similar, yet simpler, reusable C++ singleton is made available by the *TSingleton* project from Google Code (Anilao, 2010), as seen in the code below:

```
C++
-----------------------------------------------------------------------------------------------------------
template<typename type> class Singleton {
public:
    // Get the instance of this singleton.
    static type &GetInstance() {

        // Assumes template type has a default constructor.
```

```
        static type *pTheInstance;

        // Check to see if the auto pointer is empty or not.
        if(pTheInstance == NULL) {
            pTheInstance = new type();
            // This will cause the pTheInstance to be deleted when
            // program execution has ended.
            static std::auto_ptr<type> theInstance(pTheInstance);
        }

        // Return a reference of our singleton.
        return *(pTheInstance);
    }

protected:
    // Constructor hidden.
    Singleton() {}

    // Copy constructor hidden.
    Singleton(Singleton const & orig) {}

    // Assignment operator hidden.
    Singleton & operator=(Singleton const & rhs) {}

    // Destructor.
    virtual ~Singleton() {}
}
```

It can be seen from the above code that the **`GetInstance`** method will always return a singleton instance for the given template type.

The *Loki* library has a reusable singleton template called a **`SingletonHolder`** (Alexandrescu, 2001). This template class lets one create a singleton instance from any C++ **`struct`** or **`class`** using a constructor that takes no arguments. The *Loki* singleton provides template guidelines that allow for the specification of how the singleton must be created, how it is terminated, and what threading model it must use (such as single threaded or multi-threaded).

There is also a project under Google Code, called *DesignByContract* (Fraiteur, 2010), which uses *PostSharp* (Fraiteur, 2008) in order to weave in special code into a class that is configured with a **`Singleton`** attribute. The example below, from the *DesignByContract* (Fraiteur, 2010) project, shows the implementation and usage of their singleton:

```
C#
-----------------------------------------------------------------------------------------------------------
[Singleton]
public class MySingletonCandidate {
    // Default constructor
    public MySingletonCandidate() { … }
    … S N I P …
}

MySingletonCandidate obj1 = new MySingletonCandidate(); // Just use the new keyword…
// Or
MySingletonCandidate obj1 = MySingletonCandidate.Instance; // Use the injected Instance static property
```

In the code above, the **new** C# operator is replaced by code that ensures that only one instance of the attributed class ever exists. An **Instance** static property is also added to the class in cases where developers do not want to use the **new** operator.

Scala, a type-safe functional language, allows one to instantiate singleton objects using the **object** (Odersky, Spoon, & Venners, 2011) keyword. A singleton object thus cannot be instantiated with the **new** keyword. A Scala singleton object is automatically instantiated the first time it is used and there is only ever one instance per process (Odersky, Spoon, & Venners, 2011):

```
Scala
-------------------------------------------------------------------------------------------------------------
// In WorldlyGreeter.scala

// The WorldlyGreeter class
class WorldlyGreeter(greeting: String) {
    def greet() = {
        val worldlyGreeting = WorldlyGreeter.worldify(greeting)
        println(worldlyGreeting)
    }
}

// The WorldlyGreeter companion object
object WorldlyGreeter {
    def worldify(s: String) = s + ", world!"
}

// In WorldlyApp.scala
// A singleton object with a main method that allows
// this singleton object to be run as an application
object WorldlyApp {
    def main(args: Array[String]) {
        val wg = new WorldlyGreeter("Hello")
        wg.greet()
    }
}
```

In the paper, *Construction with Factories* (Cohen & Gil, 2007), Cohen and Gil show how the Java programming language can be extend with the **new** keyword on a constructor in order to implement a singleton:

```
Java
-------------------------------------------------------------------------------------------------------------
class STemplate {
    private static STemplate instance = null;
    public static new() { // Extension for the new keyword
        if (instance == null)
            instance = this();

        return instance;
    }

    STemplate() { … }
}

// … S N I P …
STemplate sTemplate = new STemplate() // Will always return the same single instance
```

Groovy's meta-programming features allow notions or idioms such as the singleton pattern to be defined in a more focal way, as shown in the article *Groovy Singleton Pattern* (Groovy, 2011). The Groovy singleton example below shows functionality that keeps track of the total number of calculations that a calculator performs. This can be achieved by using a singleton for the calculator class where a counter is defined in the class that holds the counting state (Groovy, 2011).

First a base class **`Calculator`** is defined, which performs the calculations and records the sum of the number of calculations that was performed. A **`Client`** class is also defined, that acts as a facade to the calculator singleton (Groovy, 2011):

```
Groovy
-------------------------------------------------------------------------------------------------------------
class Calculator {
    private total = 0
    def add(a, b) { total++; a + b }
    def getTotalCalculations() { 'Total Calculations: ' + total }
    String toString() { 'Calc: ' + hashCode() }
}

class Client {
    def calc = new Calculator()
    def executeCalc(a, b) { calc.add(a, b) }
    String toString() { 'Client: ' + hashCode() }
}
```

Next a *MetaClass* that intercepts all attempts to create a **`Calculator`** object is defined. The defined **`CalculatorMetaClass`** *MetaClass* will always provide a pre-created instance. The **`CalculatorMetaClass`** is then registered with the Groovy system:

```
Groovy
-------------------------------------------------------------------------------------------------------------
class CalculatorMetaClass extends MetaClassImpl {
    private final static INSTANCE = new Calculator()
    CalculatorMetaClass() { super(Calculator) }
    def invokeConstructor(Object[] arguments) { return INSTANCE }
}

def registry = GroovySystem.metaClassRegistry
registry.setMetaClass(Calculator, new CalculatorMetaClass())
```

One can now use instances of the **`Client`** class from within a Groovy script as shown below. A request to create a new **`Calculator`** class, in this case through the **`Client`** class's constructor, will always return the singleton:

```
Groovy
-------------------------------------------------------------------------------------------------------------
def client = new Client()
assert 3 == client.executeCalc(1, 2)
println "$client, $client.calc, $client.calc.totalCalculations"

client = new Client()
assert 4 == client.executeCalc(2, 2)
println "$client, $client.calc, $client.calc.totalCalculations"
```

```
/* Output
Client: 7306473, Calc: 24230857, Total Calculations: 1
Client: 31436753, Calc: 24230857, Total Calculations: 2
*/
```

The Boo language has an assembly called Boo.Lang.Useful that is filled with useful classes, but which is not yet core to the standard of the language itself (de Oliveira, 2005). Boo is a statically typed, object-oriented, general-purpose programming language with a Python-inspired syntax (de Oliveira, 2008). Boo has a key focus on language and compiler extensibility. The Boo language has features such as interfaces, multimethods, generators, type inference, duck typing, closures, currying, macros and first-class functions (Rahien, 2010). The **`Singleton`** attribute, which is defined in the Boo.Lang.Useful library, automates or mechanises the implementation of the singleton design pattern (Ionescu, 2005). Attaching the singleton attribute to a Boo structure or a Boo class auto generates code that protects all constructors on that class. The attribute also implements a single property called **`Instance`** on the class that will always return a single instance of the class.

The example below, from the article *Useful things about Boo* (Quesnel, 2005), shows a simple example for the usage of the **`Singleton`** attribute in Boo:

```
Boo
-------------------------------------------------------------------------------------------------------
"""
Hey, hey, what do you say?
"""

import Useful.Attributes from "Boo.Lang.Useful"

[Singleton]
class SingletonExample:
    [property(Variable)]
    _var as string

    def constructor():
        Variable = "Hey, hey, what do you say?"

print SingletonExample.Instance.Variable // Instance will always return a single instance
```

14.3 Abstract Factory

Arnout shows that the abstract factory pattern can be fully componentized in Eiffel (Arnout, 2004). A slight drawback of the reusable component is that no AbstractFactory exists that holds the contracts of the creational operations that are defined on the AbstractProducts.

It is also possible to register an AbstractProduct with its corresponding Product using the *Unity* dependency injection container framework (Microsoft, 2010o). The following example shows how *Unity* can be used in order to implement the abstract factory design pattern:

```
C#
-----------------------------------------------------------------------------------------------------------
public interface IAbstractProductA { void Bar(); }
public interface IAbstractProductB { void Foo(IAbstractProductA a); }

public class ProductA1 : IAbstractProductA {
    public void Bar() { Console.WriteLine("ProductA1: Called Bar"); }
}

public class ProductB1 : IAbstractProductB {
    public void Bar() {
        Console.WriteLine("ProductB1: Called Bar"); }
        public void Foo(IAbstractProductA a) {
        Console.WriteLine("ProductB1: Called Foo - uses " + a.GetType().Name);
    }
}

IUnityContainer container = new UnityContainer();
container.RegisterType<IAbstractProductA, ProductA>();
container.RegisterType<IAbstractProductB, ProductB>();

// … S N I P
var productA = container.Resolve<IAbstractProductA>();
productA.Bar();

var productB = container.Resolve<IAbstractProductB>();
productB.Foo(productA);

/* Output
ProductA1: Called Bar
ProductB1: Called Foo - uses ProductA1
*/
```

In the above code the **`IAbstractProductA`** and **`IAbstractProductB`** AbstractProducts are registered with the *Unity* framework using the **`RegisterType`** method. Each AbstractProduct is registered with its corresponding Product. For example, the **`IAbstractProductA`** interface is registered against the **`ProductA`** concrete class. The *Unity* framework can then be used to create a new Product instance by invoking the **`Resolve`** method on the container and providing it with the AbstractProduct as a generic argument. In the above example, no AbstractFactory and ConcreteFactory participants exist. An AbstractFactory defines an interface for creational operations that instantiates an AbstractProduct. A ConcreteFactory implements the creational operations with which to instantiate Product objects. The abstract factory design pattern offers an interface for creating families of related objects that assist in decoupling applications from the concrete implementation of an entire framework or library (Gamma, Helm, Johnson, & Vlissides, 1994) (McConnell, 1993). In the example, the *Unity* container fulfils the role of the AbstractFactory and ConcreteFactory participants. The *Unity* container (Microsoft, 2010o) satisfies the original intent and functionality of the abstract factory design pattern (Gamma, Helm, Johnson, & Vlissides, 1994). The output of the above example shows that the *Unity* container works as expected.

14.4 Factory Method

Arnout has shown that the factory method pattern is fully componentizable in Eiffel (Arnout, 2004). She correctly argues that the factory method is just a special case of an abstract factory using only one creational method for a Product.

The paper *Better Construction with Factories* (Cohen & Gil, 2007) shows how the factory method pattern, which is also known as a virtual constructor (Gamma, Helm, Johnson, & Vlissides, 1994), can be made more explicit in Java by proposing the ability for the **new** keyword to be manually overridden, as seen in the example below:

```
Java
-------------------------------------------------------------------------------------------------------------
abstract class Application {
    List<Document> docs;
    protected abstract new Document();

    public void newDocument() {
        // Handles the File|New menu option
        doc = new Document();
        docs.add(doc);
        doc.open();
    }

    // … S N I P …
}

class MyApplication extends Application {
    protected new Document() { // Note the new keyword
        return new MyDocumentType(); // A concrete subtype
    }

    // … S N I P …
}
```

The above code shows an implementation of the factory method pattern with dynamically bound factories. Dynamically bound means, as the name suggests, without the **static** keyword. Syntactically, the invocation of a dynamically bound factory defined in the **Application** class for objects of class **Document** is written as **application.new Document(…)**, where **application** is an instance of class **Application**. The prefix **application** can be dropped from inside the **Document** class, where it should be replaced with **this**.

14.5 Flyweight

Arnout has shown that the flyweight pattern is fully componentizable in Eiffel without any drawbacks (Arnout, 2004), relying mostly on the unconstrained genericity language feature in Eiffel.

The Boost Flyweight Library (López Muñoz, 2008), which is part of the Boost Library, implements powerful reusable C++ flyweight components. The aim of the Boost Flyweight Library is to simplify the usage of the design pattern by providing the class template **flyweight<T>,** which acts as a drop-in replacement for **const** T:

```
C++
-------------------------------------------------------------------------------------------------------------
flyweight<std::string> name1; name1 = "aaa"
flyweight<std::string> name2; name2 = "aaa"
flyweight<std::string> name3; name3 = "bbb"
std::out << name1;
```

The flyweights defined above are copy-able and assignable and will never store duplicate string instances adhering to the flyweight design pattern requirements. The `flyweight<std::string>` offers the use of common operators such as `==`, `!=`, `<`, `>`, `<=`, `>=` with the same semantics as those of a C++ `std::string`. The `flyweight<std::string>` value is immutable; however, a flyweight object can be assigned a different value. The Boost Flyweight Library `flyweight` component, in fact, is a special type of flyweight pattern adaption called a value object (Evans, 2003) (Nilsson, 2006). A value object is simply a flyweight where the key that defines the flyweight and the value of the flyweight itself are the same.

The Boost Flyweight Library also implements a key-value flyweight pattern (López Muñoz, 2008), which is the more traditional flyweight pattern, where the key and value are different.

14.6 Adapter

Arnout has shown that the adapter pattern cannot be componentized in Eiffel (Arnout, 2004).

The *PerfectJPattern* library has a reusable component implementation for the adapter pattern (Garcia, 2009a). The component allows for the auto adaption of methods between the Adaptee and Target, using different adaption strategies. The *PerfectJPattern*'s adapter implementation thus has configurable strategies to adapt Target interfaces to Adaptee implementations. The available strategy implementations offered are `ExactMatchAdaptingStrategy` and `NameMatchAdaptingStrategy`. The default `ExactMatchAdaptingStrategy` strategy verifies and resolves Target methods that have precise method name and signature matches on the Adapter and Adaptee. The `NameMatchAdaptingStrategy`, on the other hand, uses a user defined mapping of Adaptee method names to Target interface method names, where unregistered method names are defaulted to the `ExactMatchAdaptingStrategy` implementation.

14.7 Decorator

Arnout has shown that the decorator pattern cannot be componentized in Eiffel (Arnout, 2004).

The *PerfectJPattern* library has a reusable component implementation `AbstractDecorator` for the decorator pattern (Garcia, 2009b). The `AbstractDecorator` component, which has a large number of features, auto decorates given interfaces. Component methods not expressed by the Decorator are automatically passed on to the Component. Developers are thus expected to offer implementations only for those additional methods.

The following code snippet from Redpath shows how a reusable decorator component can be implemented in Ruby (Redpath, 2009), using the language's dynamic features:

```
Ruby
-----------------------------------------------------------------------------------------------------------
module Decorator
  def initialize(decorated)
    @decorated = decorated
  end

  def method_missing(method, *args)
    args.empty? ? @decorated.send(method) : @decorated.send(method, args)
  end
end
```

The Ruby **Decorator** in the code above defines a constructor that takes in a Decorator. This allows for the decorative chaining of **Decorator** instances.

The following example from Redpath (Redpath, 2009) shows the use of the Ruby **Decorator** that refers to an example shown in the book *Head First Design Patterns* (Freeman, Robson, Bates, & Sierra, 2004). The example shows the calculation for a cup of coffee. There is a **Coffee** class that defines and implements a **cost** method. For the purposes of this example the value is hard coded:

```
Ruby
-----------------------------------------------------------------------------------------------------------
class Coffee
  def cost
    2
  end
end
```

Next a **WhiteCoffee** class is defined in order to define the cost for a coffee with milk:

```
Ruby
-----------------------------------------------------------------------------------------------------------
class WhiteCoffee
  def cost
    2.4
  end
end
```

Decorator classes **Milk**, **Whip** and **Sprinkles** are defined that add their price to the coffee. An instance of the decorators will thus decorate the **cost** method with the price of the extras:

```
Ruby
-----------------------------------------------------------------------------------------------------------
class Milk
  include Decorator

  def cost
    @decorated.cost + 0.4
  end
end

class Whip
  include Decorator
```

```
  def cost
    @decorated.cost + 0.2
  end
end

class Sprinkles
  include Decorator

  def cost
    @decorated.cost + 0.3
  end
end
```

The decorators can then be used to cost the price of a cup of coffee with extras such milk, sprinkles and whip added in any combination:

```
Ruby
-------------------------------------------------------------------------------------------------------
Whip.new(Coffee.new).cost
#=> 2.2
Sprinkles.new(Whip.new(Milk.new(Coffee.new))).cost
#=> 2.9
```

14.8 Composite

Arnout has shown that the composite pattern is fully componentizable in Eiffel (Arnout, 2004). She has shown that the componentization was possible mostly because of the generics language feature in Eiffel.

The reusable Java composite component implementation in the *PerfectJPattern* Java library (Garcia, 2009d) also uses generics, as the following example shows:

```
Java
-------------------------------------------------------------------------------------------------------
public interface IGraphic {
    public void
    draw();
}

public class Line implements IGraphic {
    public void draw() { theLogger.debug("Drawing a Line"); }
    protected static void setLogger(Logger aLogger) { theLogger = aLogger; }
    private static Logger theLogger = LoggerFactory.getLogger(Line.class);
}

public class Rectangle implements IGraphic {
    public void draw() { theLogger.debug("Drawing a Rectangle"); }
    protected static void setLogger(Logger aLogger) { theLogger = aLogger; }
    private static Logger theLogger = LoggerFactory.getLogger(Rectangle.class);
}

public class Text implements IGraphic {
    public void draw() { theLogger.debug("Drawing a Text"); }
    protected static void setLogger(Logger aLogger) { theLogger = aLogger; }
    private static Logger theLogger = LoggerFactory.getLogger(Text.class);
}

public final class Example {
    public static void main(String[] anArguments) {
        //-------------------------------------------------------------
```

```
        // Create composition using the reusable Composite implementation
        //-------------------------------------------------------------
        IComposite<IGraphic> myNestedComposite = new Composite<IGraphic>(IGraphic.class);
        myNestedComposite.add(new Rectangle());
        myNestedComposite.add(new Line());
        myNestedComposite.add(new Line());

        IComposite<IGraphic> myComposite = new Composite<IGraphic>(IGraphic.class);
        myComposite.add(new Rectangle());
        myComposite.add(new Text());
        myComposite.add(new Text());
        myComposite.add(myNestedComposite.getComponent());

        //-------------------------------------------------------------
        // Acquire reference to an IGraphic view of the Composite and call
        // business methods on it
        //-------------------------------------------------------------
        IGraphic myGraphic = myComposite.getComponent();
        myGraphic.draw();
    }
}
```

In the above code the **`Composite`** component is used to create two Composite instances, **`myNestedComposite`** and **`myComposite`**, for the **`IGraphic`** Component instance. Three Leafs are also implemented, namely a **`Line`**, **`Rectangle`** and a **`Text`**. Leaf instances are then added to both Composite instances. The Component part of the **`myNestedComposite`** Composite is then added to the **`myComposite`** instance, demonstrating nested composites. The **`draw`** method is then called on the Component part of the **`myComposite`** instance, rendering all of the Leaf instances, including the ones added to the **`myNestedComposite`** instance.

14.9 State

Arnout has shown that it is possible to implement a reusable state pattern component in Eiffel (Arnout, 2004). She argues, however, that the implementation is not comprehensive because the component does not cater for all the seven state pattern implementation variants described by Dyson and Anderson (Dyson & Anderson, 1997).

14.10 Command

Arnout has shown that the command pattern is fully componentizable in Eiffel (Arnout, 2004). She has shown that the main reason componentization is possible is because generics is a language feature in Eiffel.

The simplest form of the command pattern (Evans, 2003) is built into C# because of the availability of delegates and additional language features such as anonymous methods and lambda expressions:

```
C#
-----------------------------------------------------------------------------------------------------------
Action<string> debitAccount = x => Console.WriteLine("Debiting account number…:" + x);

// Pass the action around and invoke it later…
debitAccount("404938393");
```

It is clear from the above example that the C# **`Action`** (Microsoft, 2010a) delegate allows for the implementation of a simple command pattern. The command pattern implemented in an object-oriented language, however, is more useful if the commands can hold a certain cohesive state. Furthermore, the above command pattern is not user extendable, because multiple methods cannot be associated with the command, it can only perform one action. A command class, rather than a command action, is thus a more advanced and a more extendable command implementation because a class can hold state and it can hold multiple cohesive methods. In the last example shown on the previous page, no state is stored with the command. With a more advanced command implementation it is possible for the command instance to hold some kind of state, which can then be used when the command is executed. It is, however, possible to hold some kind of state on a command instance created as an **`Action`** in C#, because the language does support closures (J'arvi, Freeman, & Crowl, 2007), as seen in the code snippet below:

```
C#
-----------------------------------------------------------------------------------------------------
DateTime dateTime = DateTime.Now; // The state stored and used by the command instance
Action<string> debitAccount = x => Console.WriteLine("Debiting account number " + x + " on " + dateTime);

// Pass the action around and invoke it later. The invoker of the command
// does not know of it's internal state.
debitAccount("404938393");
```

The date and time of the debit command is given to the **`Action`** instance on creation and only used when the **`Action`** is invoked. The example above thus implements a more advanced command than the previous example. The state of the command instance, however, is not encapsulated with the command and is thus not cohesive (Miller, 2008) with the command, because the **`dateTime`** is not explicitly coupled with the action. It would thus be better to create a debit command class that holds a **`dateTime`** state in order to make the state more cohesive and more tightly coupled with the command instance. Furthermore, because the command is implemented as a command class, different command methods can be added to the class, where each method performs a different action for the same command state.

The Lua object-oriented programming language offers fully featured closures (Ierusalimschy, 2003). One can write generators (functions that create functions) in Lua using functions, which are first-class values, and use them to create commands; as shown in the following example (Ierusalimschy, 2003):

```
Lua
-----------------------------------------------------------------------------------------------------
function newDebitCommand ()
    local dateTime = print(os.date("%x %X", 906000490))
        return function ()
            return "Debiting account number " + x + " on " + dateTime
        end
end

command = newDebitCommand()
print(command ()) --> "404099282"
```

The *PerfectJPattern* Java library (Garcia, 2009e) also has a reusable component solution for the command pattern, as the following example from the project shows:

```
Java
-----------------------------------------------------------------------------------------------------------
public class Open extends AbstractReceiver<NullParameter, NullResult> {
    public void execute() {
        theLogger.debug("Asking user for location of document ...");
        theLogger.debug("Opening document");
    }

    protected static void setLogger(Logger aLogger) { theLogger = aLogger; }
    private static Logger theLogger = LoggerFactory.getLogger(Open.class);
}

public class Paste extends AbstractReceiver<NullParameter, NullResult> {
    public void execute() { theLogger.debug("Pasting an object into the document"); }

    protected static void setLogger(Logger aLogger) { theLogger = aLogger; }
    private static Logger theLogger = LoggerFactory.getLogger(Open.class);
}

public final class Example {
    public static void main(String[] anArguments) {
        //-------------------------------------------------------------
        // Create simple use-cases with Open and Paste commands
        //-------------------------------------------------------------
        IParameterlessInvoker myOpenInvoker = new ParameterlessInvoker();
        myOpenInvoker.setCommand(new ParameterlessCommand(new Open()));
        myOpenInvoker.invoke();

        IParameterlessInvoker myPasteInvoker = new ParameterlessInvoker();
        myPasteInvoker.setCommand(new ParameterlessCommand(new Paste()));
        myPasteInvoker.invoke();

        //-------------------------------------------------------------
        // Create macro use-case with multiple Open and Paste commands
        // i.e. a Composite Command
        //-------------------------------------------------------------
        IComposite<IParameterlessCommand> myComposite = new Composite<
            IParameterlessCommand>(IParameterlessCommand.class);
        myComposite.add(new ParameterlessCommand(new Open()));
        myComposite.add(new ParameterlessCommand(new Paste()));
        myComposite.add(new ParameterlessCommand(new Open()));
        myComposite.add(new ParameterlessCommand(new Paste()));

        IParameterlessCommand myMacroCommand = myComposite.getComponent();

        //-------------------------------------------------------------
        // note how Invoker is agnostic of the underlying Composite
        // Macro Command
        //-------------------------------------------------------------
        IParameterlessInvoker myMacroInvoker = new ParameterlessInvoker();
        myMacroInvoker.setCommand(myMacroCommand);
        myMacroInvoker.invoke();
    }
}
```

The above example shows the creation of two Receivers, **`Open`** and **`Paste`** using the **`AbstractReceiver`** component. Instances of the **`Open`** and **`Paste`** Receivers are then registered using a **`ParameterlessCommand`** component with a reusable **`ParameterlessInvoker`** Invoker. The Commands are then executed using the Invoker instances. The Composite component is also used to create macro

Commands. In the above example four **ParameterlessCommand** instances, representing the **Open** and **Paste** Receivers, are registered with the **Composite** instance. The **Composite** instance, **myMacroCommand**, is then executed using a **ParameterlessInvoker**.

The **Functor** class template defined inside the C++ *Loki* library (Alexandrescu, 2001) encapsulates any object and member function of that object, including the set of arguments belonging to that member function. A f*unctor* is thus a delayed invocation to a function, another *functor*, or a member function. It stores the original function and overrides the C++ **operator()**. An instance of a **Functor** can be executed just like any other function in C++ because of the overriding of the **operator()**.

A *Loki* **Functor** object is very useful when implementing the command pattern in C++. Alexandrescu (Alexandrescu, 2001) argues that hand coded command patterns do not scale well. Alexandrescu explains that with a hand coded command pattern lots of small concrete command classes must be implemented. He states that a generic **Functor** that forwards invocations to any member function of any object reduces the amount of boiler plate code that must be coded. The *Loki* generic **Functor** can also sequence multiple actions or assemble multiple actions and execute them in a specific order, such as the macro command (Gamma, Helm, Johnson, & Vlissides, 1994), eliminating the need for developing these features by hand.

The *Loki* **Functor** C++ component is a template that allows for function calls with up to 15 arguments. The first template argument of the **Functor** is the return type. The second template argument of the **Functor** is a **typelist** holding the argument types. The third template argument defines the threading model of the allocator that is used by the **Functor**.

The following example from Alexandrescu shows a simple usage of a *Loki* **Functor**. A **Functor** is instantiated that is defined to act as a function that takes in two arguments, an **int** and a **double**, and return a **void** (Alexandrescu, 2001):

```
C++
-----------------------------------------------------------------------------------------------------------
#include "Functor.h"
#include <iostream>
using namespace std;

// Define a test function
void TestFunction(int i, double d) {
    cout << "TestFunction(" << I << ", " << d << ") called." << endl;
}

int main() {
    Functor<void, TYPELIST_2(int, double)> cmd(TestFunction);
    cmd(4, 4.5); // will print: "TestFunction(4, 4.5) called."
}
```

The **Functor** instance in the above example is invoked just like a normal function.

Multiple **Functors** can also be chained together in a single **Functor** instance by using the **Chain** function, as shown in the example below by Alexandrescu (Alexandrescu, 2001):

```
C++
-----------------------------------------------------------------------------------------------------------
void f() {
    Functor<> cmd1(something);
    Functor<> cmd2(somethingElse);
    // Chain cmd1 and cmd2 as the container
    Functor<> cmd3(Chain(cmd1, cmd2));
    // Equivalent to cmd1(); cmd2();
    cmd3();
}
```

In the above example, calling the **cmd3 Functor** instance will also call the **Functor** instances that were registered with it. This allows for the usage of a macro command (Gamma, Helm, Johnson, & Vlissides, 1994) without implementing the boiler plate code by hand.

The **Functor** also has support for argument binding. A call to **BindFirst** binds the first argument to a certain constant value, as shown in the example below by Alexandrescu (Alexandrescu, 2001):

```
C++
-----------------------------------------------------------------------------------------------------------
void f() {
    // Define a Functor of three arguments
    Functor<void, TYPELIST_3(int, int, double)> cmd1(someEntity); // Bind the first argument to 10
    Functor<void, TYPELIST_2(int, double)> cmd2(BindFirst(cmd1, 10)); // Same as cmd1(10, 20, 5.6)
    cmd2(20, 5.6);
}
```

Stevens's article in *Dr. Dobb's Journal* shows how generic implementations (Stevens, 1998) of undo and redo can be used with *functors* in order to implement the same functionality as described in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) with regard to undo and redo features on the command pattern.

In the article *GoF patterns in Ruby* Tanguay-Carel (Tanguay-Carel, 2007) shows how a reusable Command component can be implemented in Ruby:

```
Ruby
-----------------------------------------------------------------------------------------------------------
class Command
    attr_accessor : receiver
    def initialize receiver
        @receiver=receiver
        @commands=[]
    end

    def register_command *command
        @commands.push *command
    end

    def execute
        @commands.each{|cmd| cmd.save }
        @commands.each{|cmd| cmd._execute }
```

```
        save
        _execute
    end

    def undo
        @commands.each{|cmd| cmd.undo }
    end

    #implement the following methods in the subclasses
    protected
    def save
    end

    def _execute
    end
end
```

In the above code the reusable **Command** component is initialised with a Receiver. Commands can also be grouped together by registering children commands to a macro command.

The reusable **Invoker** below is a simple Invoker that just invokes the **execute** or **undo** method of a Command instance (Tanguay-Carel, 2007):

```
Ruby
-------------------------------------------------------------------------------------------------------------
module Invoker
    attr_accessor :command

    def click
        @command.execute
    end

    def undo
        @command.undo
    end
end
```

The following code shows how the **Command** component is used to implement a command class. Note how the abstract **TextCommand** Command implements the **save** and **undo** methods. The concrete **UppercaseCommand** and **IndentCommand** Commands inherit from **TextCommand** and offer an implementation for the **_execute** method (Tanguay-Carel, 2007):

```
Ruby
-------------------------------------------------------------------------------------------------------------
class TextCommand < Command
    def save
        @last_state ||= Marshal.load(Marshal.dump(@receiver.text))
        super
    end

    def undo
        @receiver.text= @last_state
        @last_state=nil
        super
    end
end

class UppercaseCommand < TextCommand
    def _execute
```

```
        @receiver.text.upcase!
        super
    end
end

class IndentCommand < TextCommand
    def _execute
        @receiver.text="\t" + @receiver.text
        super
    end
end
```

The code below shows how the above **UppercaseCommand** and **IndentCommand** implementations can be used (Tanguay-Carel, 2007):

```
Ruby
-----------------------------------------------------------------------------------------------------------
class Document
    attr _accessor :text

    def initialize text
        @text = text
    end
end

if__FILE__==$0
    text="This is a test"
    doc= Document.new text
    upcase_cmd = UppercaseCommand.new doc
    button = Object.new.extend(Invoker)
    button.command = upcase_cmd

    puts"before anything"
    putsdoc.text
    button.click
    puts"after click"

    putsdoc.text
    button.undo
    puts"after undo"
    putsdoc.text

    puts"\nNow a macro command"
    allCmds= Command.new doc
    indent_cmd= IndentCommand.new doc
    allCmds.register_command upcase_cmd, indent_cmd

    big_button= Object.new.extend(Invoker)
    big_button.command= allCmds
    puts"before anything"
    putsdoc.text
    big_button.click
    puts"after click"
    putsdoc.text

    big_button.undo
    puts"after undo"
    putsdoc.text
end
```

The above example also shows how the **Command** component can be used in order to implement a macro command (Gamma, Helm, Johnson, & Vlissides, 1994) solution without it being necessary to write the boiler plate code.

14.11 Chain of Responsibility

Arnout has shown that the chain of responsibility pattern is fully componentizable in Eiffel without any drawbacks (Arnout, 2004), relying mostly on the unconstrained genericity language feature in Eiffel.

The *PerfectJPattern* Java library also implements a comprehensive reusable component for the chain of responsibility pattern (Garcia, 2009c).

The *Commons Chain* project, which is part of the *Apache Commons Java* framework, is another reusable component for the chain of responsibility pattern (O'Brien, 2004).

Chain.NET or *.NChain* is a generic and reusable implementation of a chain of responsibility design pattern developed in C# (Stasiak, 2008). The *Chain.NET* library is based on the *Apache Commons Chain* (O'Brien, 2004) library for Java, which is mentioned above. The *Chain.NET* library merges the standard chain of responsibility design pattern with the command design pattern (Gamma, Helm, Johnson, & Vlissides, 1994) in order to implement a powerful action processing solution.

14.12 Memento

Arnout has shown that the memento pattern is fully componentizable in Eiffel without any drawbacks (Arnout, 2004), relying mostly on the unconstrained genericity language and agent features in Eiffel.

Chapter 15

15 PATTERNS, ACTIONS AND FUNCTIONS

Some of the patterns discussed in this thesis could also be transformed into reusable components where the solution focuses on one well known action or function.

For example, the command reusable pattern transformation defines an **ICommand** interface with a well known **Execute** method. The description of the command thus does not transpire in the command method name, but in the name of the command implementation itself. Thus, realizing from the **ICommand** interface, a command instance can only be implemented to perform one special task or command. A drawback of using the **ICommand** interface is thus that a user cannot cohesively combine methods in one command interface, which is the same drawback discussed in the previous chapter when using the **Action** delegate as a Command.

A benefit, however, of using a well-known method is that it makes the implementation and usage of a reusable pattern simple. Furthermore, most command implementations perform only one special task and thus naturally map to one well known method name. It is thus rare to find a command interface with multiple cohesive (Miller, 2008) command methods. Alexandrescu has also argued that hand written commands are not scalable and that reusable Commands reduce the amount of boiler plate code that must be written (Alexandrescu, 2001). The reusable command component in the APL library is thus simple, scalable and useful, but it is not easily adaptable to special user requirements.

A close relationship exists between an **Action** C# delegate and the **ICommand** interface. A C# **Action** can thus easily be converted into an **ICommand** interface, by using the **ActionCommand** component (as shown in the Chapter 11):

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
var concreteCommand = new ActionCommand(() => Console.WriteLine("The command was invoked!"));
invoker.Process(concreteCommand);
```

An **Action** C# delegate, converted into an **ICommand** interface, can make use of all the advanced command features available in the APL library. An **ICommand** interface can also easily be converted into an **Action**, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------
Action action = concreteCommand.Execute;
```

The above mentioned conversion is also useful because a Command can now be used where an **`Action`** is expected. This is especially useful with other reusable components that expect an **`Action`**.

In fact, there are more APL components where the major functionality of the pattern uses only one well known method, which can be an action or a function.

An **`ActionDecorator`** exists in the APL library. The **`ActionDecorator`** holds an internal **`Action`** delegate, which represents the Component. The **`Action`** delegate must be registered with the **`ActionDecorator`** in its constructor. In this case, however, the Component has only one well known **`Execute`** method. Furthermore, an **`ActionDecoratorStrategy`**, which implements the decoration algorithm, must also be registered with the reusable component via a constructor. The **`ActionDecorator`** also realizes the **`ICommand`** APL interface, which enables it to be used as a Command:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class ActionDecorator : ICommand {
    private readonly Action _component;
    private readonly ActionDecoratorStrategy _decoratorStrategy;

    // … C O N T R A C T S …

    private ActionDecorator(Action component) { _component = component; }
    public ActionDecorator(Action component, ActionDecoratorStrategy decoratorStrategy)
        : this(component) {
        // … C O N T R A C T S …
        _decoratorStrategy = decoratorStrategy;
    }

    public ActionDecorator(ICommand decorator, ActionDecoratorStrategy decoratorStrategy)
        : this(decorator.Execute) {
       // … C O N T R A C T S …
        _decoratorStrategy = decoratorStrategy;
    }

    public void Execute() {
        // … C O N T R A C T S …
        _decoratorStrategy(_component);
    }
}
```

The **`ActionDecorator`** does not realize any Component interface. In this case, the **`ICommand`** interface represents the Component. The **`ActionDecorator`** has only one well known **`Execute`** method, which performs the desired decoration action. The **`Execute`** method on the **`ActionDecorator`** sends the invocation request to the internal **`ActionDecoratorStrategy`** delegate, which receives the internal **`Action _component`** through its first argument:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate void ActionDecoratorStrategy(Action decoratorOperation);
public delegate void ActionDecoratorStrategy<TArg>(Action<TArg> decoratorOperation, TArg args);
public delegate void ActionDecoratorStrategy<TArg1, TArg2>(Action<TArg1, TArg2> decoratorOperation,
                                                           TArg1 arg1, TArg2 arg2);
// … M O R E …
```

An **ActionComposite** also exists in the APL library. The **ActionComposite** APL component implements a Composite with only one well known **Execute** method. The logic of each Component is injected on an instance of the **ActionComposite** component by means of an **Action** C# delegate. The **ActionComposite** component realizes the **IComponent<Action>** interface that defines the contract for a standard Component. The **ActionComposite** also realizes the **ICommand** APL interface, which enables it to be used as a Command:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionComposite : IComponent<Action>, ICommand {
    // List of Components
    private readonly List<IComponent<Action>> _components = new List<IComponent<Action>>();

    // … C O N T R A C T S …

    public ActionComposite() { }

    // Constructor that takes in an enumeration of Components
    public ActionComposite(IEnumerable<IComponent<Action>> components) {
        if(components == null) return;
        foreach(var item in components) { _components.Add(item); }
    }

    // Constructor that takes in an enumeration of Actions
    public ActionComposite(IEnumerable<Action> components) {
        if(components == null) return;
        foreach(var item in components) { _components.Add(new ActionComponent(item)); }
    }

    // Constructor that takes in a Composite
    public ActionComposite(ActionComposite composite) {
        if(composite == null) return;
        _components.Add(new ActionComponent(composite.Execute));
    }

    // Adds a Component to the Composite
    public void Add(Action component) {
        // … C O N T R A C T S …
        _components.Add(new ActionComponent(component));
    }

    public void Remove(Action component) {
        // … C O N T R A C T S …
        _components.Remove(new ActionComponent(component));
    }

    internal class ActionComponent : IComponent<Action> {
        public ActionComponent(Action action) { Target = action; }
        public IList<IComponent<Action>> GetList() { return null; }
        public Action GetInterface() { return Target; }
        public Action Target { get; private set; }
    }

    // Executes the Composite by iterating through all the Components and invoking them
    public void Execute() {
        // … C O N T R A C T S …
        _components.ForEach(x => x.GetInterface()());
    }

    // Returns the list of Components stored in the Composite
    public IList<IComponent<Action>> GetList() { return _components; }

    // Returns the Action of this Composite instance
```

```
    public Action GetInterface() {
        // … C O N T R A C T S …
        return Target;
    }

    // Returns the Target of this Composite instance, which is just the Execute method
    public Action Target {
        get {
            // … C O N T R A C T S …
            return Execute;
        }
    }
}
```

At the heart of the **`ActionComposite`** component is the list of Components of type **`IComponent<Action>`** that is used in the composite **`Execute`** method:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
private readonly List<IComponent<Action>> _components = new List<IComponent<Action>>();

// … S N I P …

// Executes the Composite by iterating through all the Components and invoking them
public void Execute() {
    // … C O N T R A C T S …
    _components.ForEach(x => x.GetInterface()());
}
```

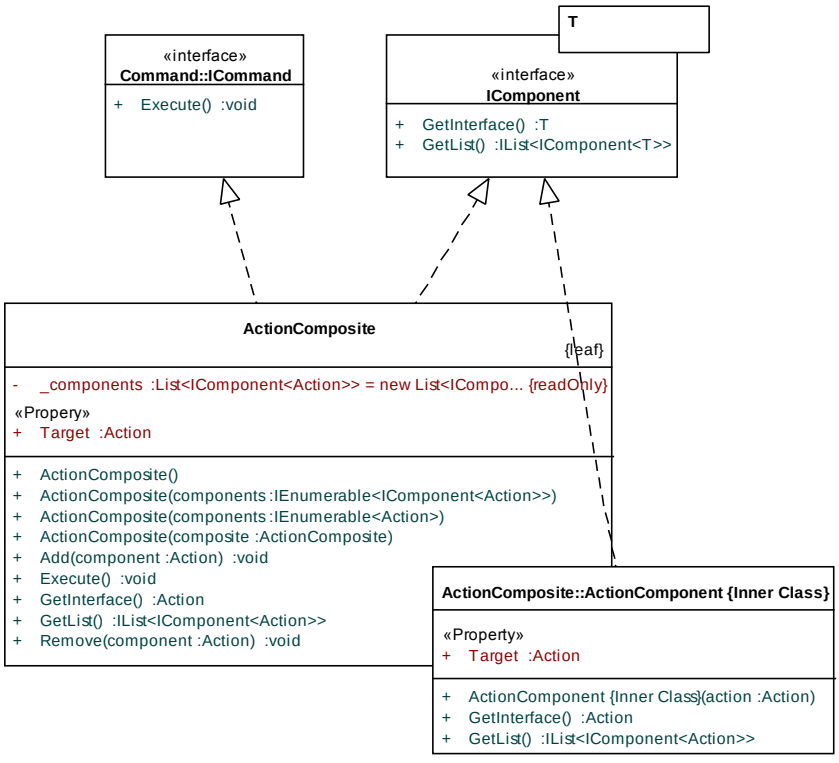

Figure 37: UML class diagram of the ActionComposite APL component.

Figure 37 shows a UML class diagram of the **`ActionComposite`** APL component, which shows the public methods of an **`ActionComposite`** component and that it realizes the **`ICommand`** and **`IComponent`** APL interfaces.

An **`Action`** can be added to an instance of the **`ActionComposite`** component with the **`Add`** method, which is also an implementation of the **`IComponent<Action>`** interface. Internally, the **`Action`** is converted into a Component using the internal **`ActionComponent`** inner class:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public void Add(Action component) {
    _components.Add(new ActionComponent(component)); // Convert the Action to a Component and add it to
                                                     // the internal list of Components
}
```

The **`Execute`** method iterates through the list of internal Components and executes the **`Action`** on each one of them.

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public void Execute() { _components.ForEach(x => x.GetInterface()()); }
```

The **`ActionComposite`** thus implements a standard Composite with only one operation that represents a basic composition algorithm of iterating through the list and invoking each Component.

An **`ActionChainOfResponsibility`** also exists in the APL library. The **`ActionChainOfResponsibility`** APL component is a simple implementation of the chain of responsibility pattern. It allows the client to inject a C# **`Action`** delegate for the Handler and also for the successor, which itself is also a Handler. The **`ActionChainOfResponsibility`** also realizes the **`ICommand`** APL interface, which enables it to be used as a Command:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class ActionChainOfResponsibility : ICommand { // The handler is also a command
    private readonly Action _successor; // Successor defined as an Action delegate
    private readonly Action _handler;   // Handler defined as an Action delegate

    // … C O N T R A C T S …

    public ActionChainOfResponsibility(Action handler, Action successor) : this(handler) {
        _successor = successor;
    }

    public ActionChainOfResponsibility(ICommand handler, ICommand successor) : this(handler.Execute) {
        _successor = successor.Execute;
    }

    public ActionChainOfResponsibility(Action handler, ICommand successor) { … }
    public ActionChainOfResponsibility(ICommand handler, Action successor) { … }

    public ActionChainOfResponsibility(Action handler) { _handler = handler; }
    public ActionChainOfResponsibility(ICommand handler) { _handler = handler.Execute; }
```

```
    public void Execute() { // Execute the handler
        // … C O N T R A C T S …

        ChainOfResponsibilityEx.Handled = false;
        if(_handler == null)
            throw new Exception("The chain of responsibility handler cannot be null");

        _handler(); // Invoke the handler as it is just a .NET action

        if(ChainOfResponsibilityEx.Handled) return; // Return of the hanlder handled the request
        if(_successor != null) _successor(); // Invoke the successor (if necessary)
    }
}
```

The injected Handler uses the APL **SetHandled** extension method, described in the Chapter 12, in order to tell the **ActionChainOfResponsibility** whether the request was handled or not. The reusable component has an **Execute** method that serves as the Handler method. The **Execute** method first determines whether a Handler was injected with the component, and throws an exception if not. It then calls the Handler, which is just an **Action**. If the Handler did not process the request, then the **Successor** is called, which is also just an **Action**.

The **ActionFactoryCreator** component, as discussed in the Chapter 5, realizes both the **IFactory** and **ICommand** interfaces. The **Create** method on the component, which is an implementation of the **IFactory** interface, is thus the Creator, a name that is well known. The **Execute** method on the reusable component, which is an implementation of the **ICommand** interface, is the well-known method that uses the Creator.

Reusable patterns also exist in the APL library that use C# **Func** delegates rather than C# **Actions**.

A **SimpleGenericAbstractFactory** component also exists in the APL library. The component has only one creational method, which must be registered with a **Factory** delegate or an **IFactory** interface. Multiple implementations of the **SimpleGenericAbstractFactory** component exist for each corresponding **Factory** delegate or **IFactory** interface, where each holds a certain number of arguments. The **SimpleGenericAbstractFactory**, which is a singleton (Gamma, Helm, Johnson, & Vlissides, 1994), also implements the **IFactory** APL interface, as shown below:

```
C# (APL)
-------------------------------------------------------------------------------------------------------------
public class SimpleGenericAbstractFactory<TAbstractProduct> :
        Singleton<SimpleGenericAbstractFactory<TAbstractProduct>>,
        IFactory<TAbstractProduct> {
    private readonly Factory<TAbstractProduct> _factory;
    // … C O N T R A C T S …

    private SimpleGenericAbstractFactory() {}
    public Register(Factory<TAbstractProduct> factory) { _factory = factory; }
    public Register(IFactory<TAbstractProduct> factory) { _factory = factory.Create; }

    // Convert a Func to a Factory…
    public Register(Func<TAbstractProduct> factory) { _factory = () => factory(); }
```

```
    public TAbstractProduct Create() {
        // … C O N T R A C T S …
        return _factory();
    }
}
```

The creational **`Factory`** delegates and **`IFactory`** interfaces are implemented in the APL library, as shown in Chapter 4. Both the **`Factory`** delegates and the **`IFactory`** interfaces define method contracts that should return a newly created instance. A number of **`Factory`** delegates exist in the APL library, each with a different set of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public delegate TResult Factory<out TResult>();
public delegate TResult Factory<out TResult, in T>(T arg);
public delegate TResult Factory<out TResult, in T1, in T2>(T1 arg1, T2 arg2);
// … M O R E …
```

A number of **`IFactory`** interfaces also exist in the APL library, again each with a different set of arguments:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public interface IFactory<out TResult> { TResult Create(); }
public interface IFactory<out TResult, in T> { TResult Create(T arg); }
public interface IFactory<out TResult, in T1, in T2> { TResult Create(T1 arg1, T2 arg2); }
// … M O R E …
```

The **`Create`** method on the **`SimpleGenericAbstractFactory`**, on the creational **`Factory`** delegates and on **`IFactory`** interface, returns a Product and thus represents a function that can be converted into a C# **`Func`**. The abstract factory design pattern offers an interface for creating families of related objects that assist in decoupling applications from the concrete implementation of an entire framework or library (Gamma, Helm, Johnson, & Vlissides, 1994) (McConnell, 1993). An abstract factory pattern implementation using the **`SimpleGenericAbstractFactory`** component has no AbstractFactory and ConcreteFactory participants. When using the **`SimpleGenericAbstractFactory`**, the family of related creational methods is replaced by a family of related registered AbstractProduct types. Thus, instead of creating new Products using the creational methods available on the AbstractFactory, new Products are created using the AbstractProduct type as a generic argument:

```
C# (Example)
-----------------------------------------------------------------------------------------------------------
// ++++++++++  1) Traditional Abstract Factory ++++++++++
public interface IMyAbstractFactory { // AbstractFactory
    IFoo CreateFoo(); // Creational method that return an AbstractProduct
    IBar CreateBar(); // Creational method that return an AbstractProduct
}

public MyAbstractFactory : IMyAbstractFactory { // ConcreteFactory
    IFoo CreateFoo() { return new Foo(); }
    IBar CreateBar() { return new Bar(); }
}
```

```
// Register the MyAbstractFactory ConcreteFactory with a Singleton in order for the instance to be
// avaliable system wide…
MyAbstractFactorySingleton.Instance.Register(new MyAbstractFactory());
IMyAbstractFactory factory = MyAbstractFactorySingleton.Instance.Get // Retrieve the ConcreteFactory from
                                                                    // the Singleton…

// Use the ConcreteFactory in order to create new Products, using the creational methods
// on the AbstractFactory
IFoo foo = factory.CreateFoo(); // Create a Product that realize IFoo
IBar bar = factory.CreateBar(); // Create a Product that realize IBar

// ++++++++++ 2) SimpleGenericAbstractFactory Abstract Factory ++++++++++
SimpleGenericAbstractFactory<IFoo>.Instance.Register(() => return new Foo()); // Register a Foo instance
SimpleGenericAbstractFactory<IBar>.Instance.Register(() => return new Bar()); // Register a Bar instance

// Use the SimpleGenericAbstractFactory in order to create new Products, using the AbstractProduct
// type generic argument
IFoo foo = SimpleGenericAbstractFactory<IFoo>.Instance.Create(); // Create a Product that realize IFoo
IBar bar = SimpleGenericAbstractFactory<IBar>.Instance.Create(); // Create a Product that realize IBar
```

In the above example, the **SimpleGenericAbstractFactory** reusable component fulfils the role of the AbstractFactory and ConcreteFactory participants. The **SimpleGenericAbstractFactory** component thus adheres to the original intent and functionality of the abstract factory pattern (Gamma, Helm, Johnson, & Vlissides, 1994). An abstract factory implementation using the **SimpleGenericAbstractFactory** follows the same concept when implementing an abstract factory with a dependency injection container framework (Fowler, 2004), as shown with the *Unity* container (Microsoft, 2010o) in Chapter 14.

The **FuncFactoryCreator** component, as discussed in the Chapter 5, is exactly the same as the **ActionFactoryCreator** component, except that the **Execute** method is a function and not an action.

A **FuncDecorator** also exists in the APL library. The **FuncDecorator** is a simple APL component that applies the decorator pattern to a well-known **Execute** method on the reusable component. A **Func** delegate is stored by the **FuncDecorator** APL component that represents the algorithm of the decorative method. The **Execute** method routes the call to a registered **FuncDecoratorStrategy** APL delegate and passes the internal **Func** delegate to it:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class FuncDecorator<TResult> {
    private readonly Func<TResult> _component;
    private readonly DecoratorStrategy<TResult> _decoratorStrategy;

    // … C O N T R A C T S …

    private FuncDecorator(Func<TResult> component) {
        _component = component;
    }

    public FuncDecorator(Func<TResult> component, DecoratorStrategy<TResult> decoratorStrategy)
        : this(component) {
        _decoratorStrategy = decoratorStrategy;
    }

    public FuncDecorator(FuncDecorator decorator, DecoratorStrategy<TResult> decoratorStrategy)
```

```
        : this(component.Execute) {
        _decoratorStrategy = decoratorStrategy;
    }

    public TResult Execute() {
        // … C O N T R A C T S …
        return _decoratorStrategy(_component);
    }
}
```

Multiple **`FuncDecorator<TResult>`** components exist in the APL library, one for each of the corresponding **`Func`** delegates in the C# framework.

Reusable pattern components, implemented using actions or functions which represent the main functionality of the pattern, are simple to implement and easy to use and understand. The usage of the components can be abstracted to commands, actions or functions. The components can thus take advantage of powerful command functionality available in the APL library, or useful functional programming features available in C#. Alexandrescu has also argued that generic commands are scalable and that reusable Commands reduce the amount of boiler plate code that must be written (Alexandrescu, 2001). Reusable pattern components, implemented using actions or functions, are also scalable because of the small amount of boiler plate code that must be written when using the components. A drawback with the reusable components discussed in this chapter, however, is the fact that they are not extendible or adaptable. Thus, any advanced requirements desired by a user, especially the need for cohesive (Miller, 2008) contracts, would very likely make the use of a particular component impossible.

Some of the reusable pattern components shown in this chapter have not been discussed in their corresponding pattern chapter in this thesis, because a more extendible reusable pattern component has already been shown in that chapter for the pattern. Consequently, these patterns do not form part of the statistics shown in Chapter 16.

Chapter 16

16 CONCLUSION

This thesis has reviewed twelve patterns defined in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) and assessed their level of componentizability. Each pattern's reusable component or components, which are implemented in C# 4.0, is discussed in detail, including the success of the reusable component transformation.

All the design patterns reviewed in this thesis could be transformed into fully or partially reusable components, thus making their pattern implementation in C# 4.0 by a developer easier and more traceable. It thus stands to reason that object-oriented languages implementing the same language features as have been reviewed in this report should have the same level of success in transforming design patterns into reusable components. The following table shows a summary of the pattern componentization:

Table 3: Design pattern componentization summary.

Pattern category	Pattern	Complexity of reusable solution	Number of language features used	Success
Creational	Prototype	Simple	4	Partial Success
	Singleton	Moderate	5	Success
	Abstract Factory	Complex	10	Success
	Factory Method	Simple	9	Partial Success
Structural	Flyweight	Moderate	6	Success
	Adapter	Complex	10	Success
	Decorator	Complex	10	Success
	Composite	Complex	11	Success
Behavioural	State	Complex	7	Success
	Command	Moderate	7	Success
	Chain of Responsibility	Simple	8	Success
	Memento	Moderate	7	Success

From the above table it can be seen that the componentization success rate for the patterns discussed in this thesis is 83.33%. Not all of the patterns shown in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994), however, are discussed in this thesis.

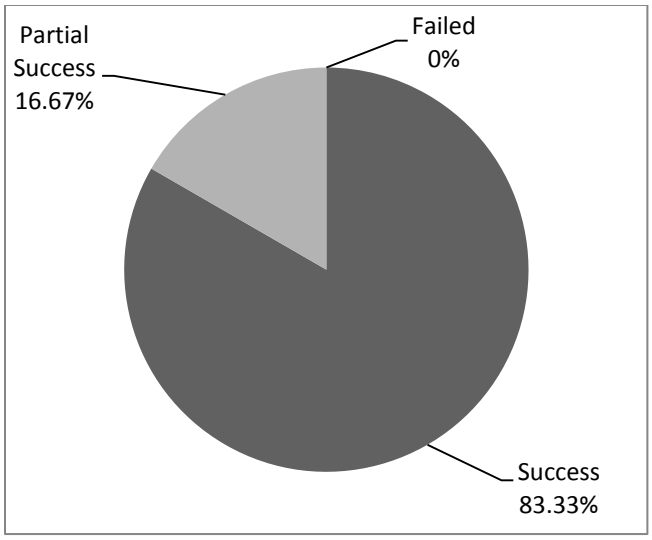

Figure 38. Componentization success rate of design patterns discussed in this thesis.

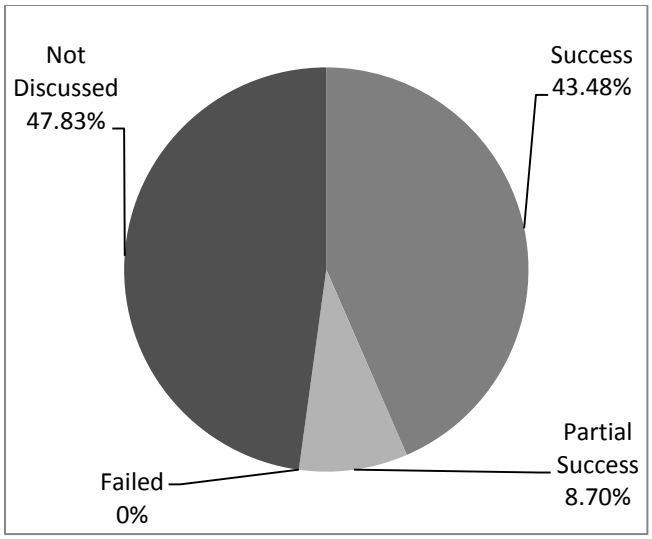

Figure 39. Componentization success rate against all of the patterns available in Design Patterns.

Figure 38 shows a pie chart of the componentization success rate of those design patterns discussed in this thesis. Figure 39 show the componentization success rate of design patterns discussed in this thesis against all of the patterns available in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994). Although most of the patterns that were chosen for this thesis could be converted into reusable

pattern components, the outcome of the pattern componentization of the rest of the patterns in *Design Patterns* (Gamma, Helm, Johnson, & Vlissides, 1994) remains unknown. That said, however, design patterns that have structural rules which may be implemented by the advanced language features available in C# 4.0 and also design patterns that are mostly behavioural, should be able to be componentized successfully. Arnout has shown that the decorator, adapter, template method, bridge, singleton, iterator, facade, and interpreter design patterns could not be componentized in Eiffel (Arnout, 2004). The template method, bridge and facade design patterns will also most probably not be componentizable in C# 4.0, chiefly because of their structural nature (Arnout, 2004). The iterator pattern is already built into C# 4.0. This thesis has shown that the decorator, adapter and singleton design patterns could be fully componentized using advanced language features in C# 4.0. Arnout has shown that the observer, mediator and visitor design patterns to be fully componentizable in Eiffel, and these patterns should also be fully componentizable in C# 4.0 because of the availability of more advanced language features. Arnout has also shown the memento reusable component not to be useful in Eiffel (Arnout, 2004), where this thesis has shown the memento component to be very useful in C# 4.0. She has also shown the state pattern to be componentizable in Eiffel, but not comprehensively, where this thesis has shown the state pattern to be fully componentizable in C# 4.0. That leaves the builder, proxy and strategy design patterns. Arnout shows that the builder and proxy design patterns are componentizable in Eiffel, but not comprehensively (Arnout, 2004). Her builder component supports only builders that need to construct no more than two-part or three-part products. Her proxy component does not cover all cases described in the original proxy pattern, because remote proxies, protection proxies and smart references are not supported. The builder and proxy design patterns should have a better componentization success probability in C# 4.0, because of the availability of more advanced language features. Proxy library components are already available in C#, such as the DynamicProxy library from CastleProject (CastleProject, 2011). A proxy library component in C# 4.0, however, would probably suffer from the same problems mentioned by Arnout, although to a lesser degree. No proxy library can cater for all the different types of proxies. With some effort, however, a comprehensive number of proxy components, where each one specialises in a certain area, such as remote proxies or protection proxies, can be built. She also shows that the strategy pattern is componentizable yet not faithful in Eiffel (Arnout, 2004), because of the exclusive usage of Eiffel agents (delegates). The strategy pattern can be implemented in C# 4.0 using the same techniques shown in this thesis, such as *duck typing* (Koenig & Moo, 2005) for an advanced component and `Action` and `Func` delegates and the `ICommand` APL interface for a simpler component that uses only one well known method. The strategy pattern, however, just like the adapter pattern, is easy to implement manually. There will thus always be situations in C# 4.0 where a manually implemented strategy pattern would be the best choice.

Figure 40, on the next page, shows a pie chart for the complexity break down of the pattern componentization effort.

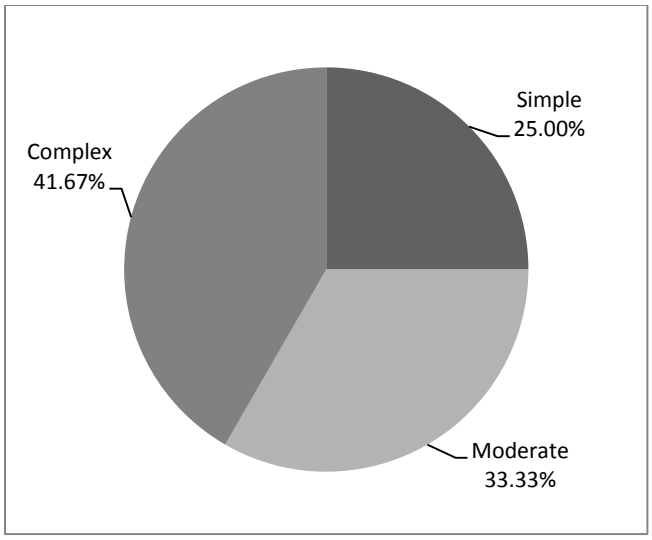

Figure 40. Reusable pattern implementation complexity.

For the reusable patterns implemented in the APL library, 25% of the implementations are simple, 33.33% are moderately complex and 41.67% are complex. Advanced language features thus do not guarantee that all reusable pattern implementations will necessarily be simple. On the contrary, the most complex reusable pattern implementations in this thesis use advanced language features available in C# 4.0.

The graph below shows the distribution of the language features used in the implementation of the reusable design pattern components in this thesis:

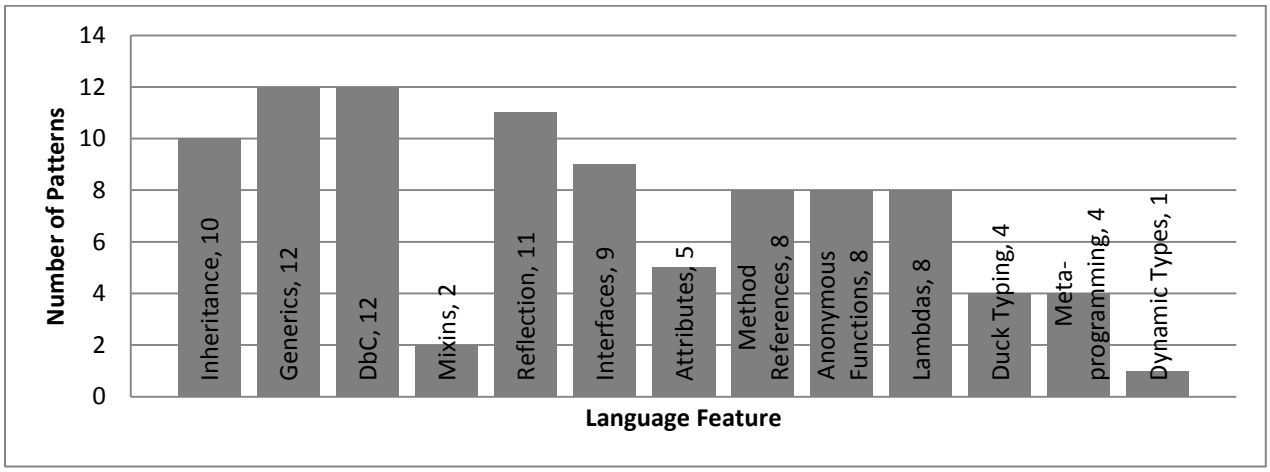

Figure 41. Distribution of language features used in pattern componentization.

Table 4 shows the language features that were used for the design pattern components described in each pattern chapter in this thesis:

Table 4: Language features used per pattern component.

Pattern category	Design pattern	Inheritance	Generics	Design by contract™	Mixins	Reflection	Interfaces	Attributes	Method references	Anonymous functions	Lambda expressions	Duck typing	Meta-programming	Dynamic types
Creational	Prototype		✓	✓	✓	✓								
	Singleton	✓	✓	✓		✓		✓						
	Abstract Factory	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	
	Factory Method	✓	✓	✓		✓	✓	✓	✓	✓	✓			
Structural	Flyweight	✓	✓	✓		✓	✓	✓						
	Adapter	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	
	Decorator	✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	
	Composite	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	
Behavioural	State	✓	✓	✓	✓	✓	✓	✓						
	Command	✓	✓	✓			✓		✓	✓	✓			
	Chain of Responsibility	✓	✓	✓		✓	✓		✓	✓	✓			✓
	Memento		✓	✓		✓			✓	✓	✓			

Not surprisingly, generics (Jagger, Perry, & Sestoft, 2007) and design by contract™ (Mitchell & McKim, 2001) are the most widely used language features. They are used in the componentization of all 12 reusable components. Reflection (Sobel & Friedman, 1996) is the next most widely used language feature and is used in 11 reusable components. Inheritance (Mitchell, Mitchell, & Krzysztof, 2003) is used in the implementation of 10 reusable components. Interfaces (Pattison & Box, 2000) are the next most widely used language feature and are used in 9 reusable components. Method references (Microsoft, 2010e), anonymous functions (Ierusalimschy, 2003) and lambda expressions (Michaelis, 2010) are used in 8 of the reusable components. Attributes (Nagel, Evjen, Glynn, & Watson, 2010) are used in 5 of the reusable component implementations. Duck typing (Koenig & Moo, 2005) and meta-programming (Perrotta, 2010) are used in 4 of the reusable component implementations. Mixins

(Esterbrook, 2001) are used in 2 of the reusable components. Finally, dynamic types (Tratt, 2009) are used in the implementation of just one of the reusable components.

Figure 42 shows the distribution of pattern components used in other pattern componentization implementations. The abstract factory and factory method components use the prototype component. The state component uses the flyweight and singleton components. The command component uses the composite component. Finally, the memento component uses the prototype component. The figure below thus shows that it is possible for the implementation of one pattern component to use another reusable design pattern component.

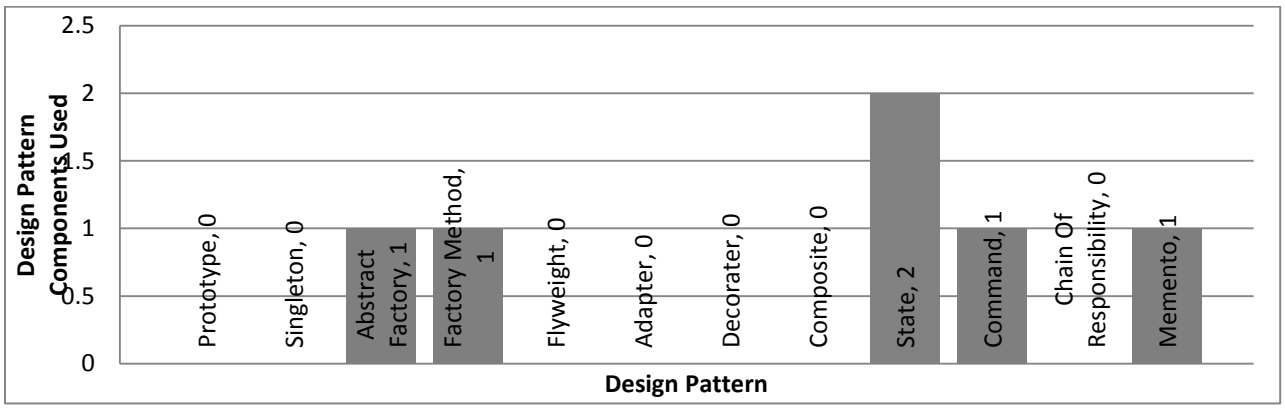

Figure 42. Distribution of pattern components used in other pattern componentization implementations.

Problems with design patterns include **traceability** in the implementation, the **implementation overhead** or **writability** (Bosch, 1998b) (Bosch, 1998a) and **maintainability** (Soukup, 1995), as discussed at the beginning of this thesis.

In the paper *Language features meet design patterns: raising the abstraction bar* it is argued that modern language features ameliorate all the above mentioned problems experienced in design pattern implementation (Bishop, 2008). This thesis has shown that modern language features also make it possible to improve the componentization of design patterns.

Reusable design pattern components solve the **traceability** problem, because the usage of a specific pattern library component clearly shows what pattern is being implemented. The physical implementation of a specific design pattern using reusable pattern components thus makes the pattern easy to identify and trace. Reusable design pattern components also solve the **reusability** problem. Design patterns are used in multiple places, and thus reused, in the design of a software system. With reusable components a developer is not forced to implement a design pattern repeatedly in a physical programming language. With reusable pattern components a developer can focus on re-implementing the outcome of a pattern and leave the plumbing and functional implementation of the pattern to the

library component. Reusable pattern components also solve the **implementation overhead** or **writability** problem. Traditionally, design patterns force a developer to implement several methods with trivial behaviour. When using reusable pattern components, however, most of these methods can simply be reused. Reusable pattern components also solve the **maintainability** problem. Reusable design pattern components do not force a developer to implement the behavioural and structural boiler plate code associated with a specific design pattern. This relieves the programming burden on the developer, which is exacerbated by the fact that traditional design pattern implementations cannot be reused.

Agerbo and Cornils have shown that there are design patterns that can be covered by a language construct in some, although not all, programming languages (Agerbo & Cornils, 1997). They categorise these design patterns as Language Dependant Design Patterns (LDDPs). As shown in the previous chapter, although the simplest form of the command pattern (Evans, 2003) is built into C#, it does not solve every possible user requirement. Furthermore, Agerbo and Cornils state that, when using a pattern as a Library Design Pattern (LDP), the design pattern implementation is fixed. It would thus not be possible to adapt the LDP in other ways desired by a user. The implementations of the pattern components in this thesis have shown this statement by Agerbo and Cornils to be partially incorrect. Most of the pattern components shown in this thesis are adaptable and should solve most of a user's requirements. A few, however, are more rigid. For example, with the command component, the user has access to only the `Execute`, `Undo` and `Redo` methods on a Command interface. If a user requires more methods to be available on the Command interface, then they must be built in manually.

Implementing design patterns as reusable library components is thus a step in the right direction for making design pattern implementations more traceable, more reusable and more productive. Design pattern transformations to reusable component artefacts should become more effective and simpler with the increase in advanced language features in main stream programming languages. Domain Specific Languages (DSL), functional and dynamic languages for example, open up an entire new dimension with regard to design pattern component transformation, as has been shown with the pattern components which use these language features in Chapter 14.

17 FUTURE WORK

More research must be done in the formalisation of design patterns in order for reusable design patterns to reach their full potential. *Design Patterns Formalization Techniques* (Taibi, 2007) and *Stepwise Refinement Validation of Design Patterns Formalized in TLA+ using the TLC Model Checker* (Taibi, Herranz, & Moreno-Navarro, 2009) show current trends in design pattern formalisation. The structural and behavioural rules of design patterns are currently described informally. The formalisation of patterns is an attempt to formalise the structural and behavioural rules that apply to a specific design pattern. The formalisation of design patterns will make pattern componentization easier, because any ambiguities in the pattern implementation will be eliminated. Figure 43 shows the formal specification of the bridge design pattern (Gamma, Helm, Johnson, & Vlissides, 1994) in LePUS3:

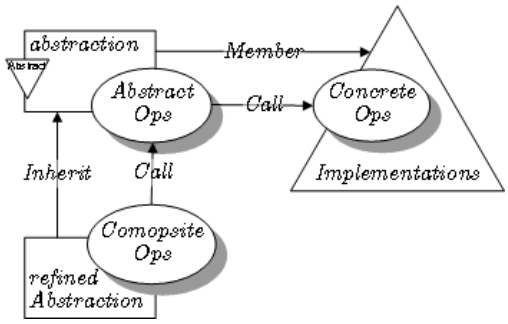

Figure 43. Bridge design pattern in LePUS3.

Gasparis, Nicholson and Eden define LePUS3 as "a visual object-oriented design description language: a notation for modelling and visualizing object-oriented programs at any level of abstraction" (Eden, Epameinondas, & Nicholson, 2011, para. 2). Appendix IV shows the basic set of symbols used in LePUS3.

In *Refactoring to patterns* Kerievsky shows a catalogue of refactoring rules for changing legacy code to use design patterns (Kerievsky, 2004). Reusable design pattern components should make these refactorings easier to implement and automate in advanced tools. For example, the *Replace State-Altering Conditionals with State* refactoring action could use the state reusable component, which should simplify the refactoring action. The *Replace Conditional Dispatcher with Command*, *Limit Instantiation with Singleton*, *Replace Constructors with Creation Methods* and *Replace One/Many Distinctions with Composite* refactoring actions (Kerievsky, 2004) could also be implemented and thus simplify using reusable pattern components.

REFERENCES

Abrams, B. (2004, May 3). *Should we obsolete ICloneable (The SLAR on System.ICloneable)*. Retrieved from blogs.msdn.com: http://blogs.msdn.com/b/brada/archive/2004/05/03/125427.aspx

Agerbo, E., & Cornils, A. (1997). *Theory of language support for design patterns.* Aarhus, Denmark: Aarhus University.

Agerbo, E., & Cornils, A. (1998). How to preserve the benefits of design patterns. *13th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications. 33*, pp. 134-143. Vancouver, Canada: ACM.

Albahari, J., & Albahari, B. (2007). *C# 3.0 in a nutshell.* Sebastopol, CA: O'Reilly Media.

Alexander, C., & Ishikawa, S. S. (1977). *A pattern language: Towns, buildings, construction.* Oxford, United Kingdom: Oxford University Press.

Alexandrescu, A. (2001). *Modern C++ design: Generic programming and design patterns applied.* Boston, MA: Addison-Wesley Professional.

Alpert, S., Brown, K., & Woolf, B. (1998). *The design patterns Smalltalk companion.* Boston, MA: Addison-Wesley Professional.

Anilao, C. B. (2010, November 3). *TSingleton: The templated singleton*. Retrieved from code.google.com: http://code.google.com/p/tsingleton/

Armstrong, J. (2007). *Programming Erlang: Software for a concurrent world.* Raleigh, NC: Pragmatic Bookshelf.

Arnout, K. (2004). *From patterns to components.* Zurich, Switzerland: Swiss Federal Institute of Technology.

Arnout, K., & Bezault, E. (2004, April). How to get a singleton in Eiffel? *Journal of Object Technology*, 75–95. Retrieved from http://www.jot.fm/issues/issue_2004_04/article5/

Avgeriou, P., & Zdun, U. (2005). Architectural patterns revisited — A pattern language. In R. Morrison, *10th European conference on pattern languages of programs (EuroPlop 2005)* (pp. 1-39). Paphos, Cyprus: Springer.

Balagurusamy, E. (2008). *Programming in C#.* New Delhi, India: Tata McGraw-Hill Education.

Barnett, M., Leino, R. K., & Schulte, W. (2005). The Spec# programming system: An overview. In G. Barthe, *Construction and analysis of safe, secure, and interoperable smart devices* (Vol. 3362, pp. 49-69). Marseille, France: Springer.

Binder, R. V. (1999). *Testing object-oriented systems: models, patterns, and tools.* Boston, MA: Addison-Wesley Professional.

Bishop, J. (2007). *C# 3.0 design patterns*. Sebastopol, CA: O'Reilly Media.

Bishop, J. (2008). Language features meet design patterns: Raising the abstraction bar. *Proceedings of the 2nd international workshop on the role of abstraction in software engineering*, 1-7.

Bishop, J., & Horspool, R. N. (2008). On the efficiency of design patterns implemented in C# 3.0. In R. F. Paige, *Objects, components, models and patterns (Lecture notes in business information processing)* (Vol. 11, pp. 356-372). Zurich, Switzerland: Springer.

Bosch, J. (1998a). Design patterns & frameworks: On the issue of language support. In Á. Frohner (Ed.), *Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP. 1357*, pp. 133-136. London, United Kingdom: Springer.

Bosch, J. (1998b). Design patterns as language constructs. *Journal of object-oriented programming, 11*(2), 18-32.

Bosch, J. (1998c). Specifying frameworks and design patterns as architectural fragments. *Proceedings: Technology of object-oriented languages (TOOLS 27). 0*, pp. 268-277. Beijing, China: IEEE Computer Society.

Bracha, G., & Cook, W. (1990). Mixin-based inheritance. *OOPSLA/ECOOP '90 Proceedings of the European conference on object-oriented programming systems, languages, and applications* (pp. 303-311). Ottawa, Canada: ACM.

Budinski, F., Finnie, M., Yu, P., & Vlissides, J. (1996). Automatic code generation from design patterns. *IBM Systems Journal, 35*(2), 151-171.

Burchall, L. (2009, August 13). *Multimethods in C# 4.0 with 'dynamic'*. Retrieved from blogs.msdn.com: http://blogs.msdn.com/b/laurionb/archive/2009/08/13/multimethods-in-c-4-0-with-dynamic.aspx

Bustamante, M. L. (2007). *Learning WCF: A hands-on guide*. Sebastopol, CA: O'Reilly Media.

Cantu, M. (2008). *Essential Pascal*. New York, NY: CreateSpace.

Cardelli, L., & Wegner, P. (1985). On understanding types, data abstraction, and polymorphism. *Computing surveys*, 471-522.

CastleProject. (2011). *DynamicProxy*. Retrieved from www.castleproject.org: http://www.castleproject.org/dynamicproxy/index.html

Chambers, C. (1992). *The design and implementation of the SELF Compiler, an optimizing compiler for object-oriented programming languages*. Palo Alto, CA: Stanford University.

Chambers, C., Harrison, B., & Vlissides, J. O. (2000). A debate on language and tool support for design patterns. *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on principles of programming languages* (pp. 277-289). Boston, MA: ACM.

Chaudhry, P. (2002, May 1). A per-thread singleton class. *Dr Dobbs*. Retrieved from http://drdobbs.com/184401516

Chris, R. (1989). *Elements of functional programming.* Wokingham, United Kingdom, U.K.: Addison-Wesley Longman.

Church, A. (1936, April). An unsolvable problem of elementary number theory. *American Journal of Mathematics, 58*, 345-363.

Clifton, C., Millstein, T., Leavens, G. T., & Chambers, C. (2006, May). MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst., 28*(3), 517-575.

Cohen, T., & Gil, J. (2007, July-August). Better construction with Factories. *Journal of Object Technology*, 103-123. Retrieved from http://www.jot.fm/issues/issue_2007_07/article3/

Coplien, J. O. (1995). Curiously recurring template patterns. *C++ Report*, 24-27.

Coplien, J. O., & Schmidt, D. (1995). *Patterns languages of program design.* Boston, MA: Addison-Wesley.

Cuni, A., Ancona, D., & Rigo, A. (2009). Faster than C#: Efficient implementation of dynamic languages on .NET. *ICOOOLPS'09.* (pp. 26-33). Genoa, Italy: ACM.

David, F. (2006). *JavaScript: The Definitive Guide.* Sebastopol, CA: O'Reilly & Associates. Retrieved from ISBN 0-596-10199-6

de Oliveira, R. B. (2005, October 2). *Boo.Lang.Useful.* Retrieved from docs.codehaus.org: http://docs.codehaus.org/display/BOO/Boo.Lang.Useful

de Oliveira, R. B. (2008, January 2). *http://docs.codehaus.org/display/BOO/Home.* Retrieved from docs.codehaus.org: http://docs.codehaus.org/display/BOO/Home

de Smet, B. (2008, November 10). *Introducing "The C# Ducktaper" – Bridging the dynamic world with the static world.* Retrieved from Bartdesmet: http://bartdesmet.net/blogs/bart/archive/2008/11/10/introducing-the-c-ducktaper-bridging-the-dynamic-world-with-the-static-world.aspx

Dehnert, J. C., & Stepanov, A. (2005). Fundamentals of generic programming. In M. Jazayeri, R. Loos, & D. R. Musser, *Generic programming* (Vol. 1766, pp. 1-11). Wadern, Germany: Springer.

DeMichiel, L. G., & Gabriel, R. P. (1987). The common lisp object system: an overview. *ECOOP. 276*, pp. 151-170. Paris, France: Springer.

Dijkstra, E. W. (1974, August). On the role of scientific thought. In E. W. Dijkstra, *Selected writings on computing: A personal perspective* (pp. 60-66). New York, NY: Springer-Verlag.

Drepper, U. (2007, November). *What every programmer should know about memory.* Retrieved from people.redhat.com: http://people.redhat.com/drepper/cpumemory.pdf

Driesen, K., & Hölzle, U. (1996). The direct cost of virtual function calls in C++. *SIGPLAN Not., 31*(10), 306-323.

Dyson, P., & Anderson, B. (1997). State patterns. In C. Martin R., D. Riehle, & F. Buschmann, *Pattern languages of program design 3* (pp. 271-294). Boston, MA: Addison-Wesley.

ECMA. (2006, June). *ECMA International: Standard ECMA-367 —Eiffel: Analysis, design and programming language*. Retrieved from www.ecma-international.org: http://www.ecma-international.org/publications/standards/Ecma-367.htm

Eric, L. (2007, January 10). *Lambda expressions vs. anonymous methods*. Retrieved from http://blogs.msdn.com: http://blogs.msdn.com/b/ericlippert/archive/2007/01/10/lambda-expressions-vs-anonymous-methods-part-one.aspx

Esterbrook, C. (2001, April 1). Using Mix-ins with Python. *Linux Journal*. Retrieved from http://www.linuxjournal.com/article/4540

Evans, E. (2003). *Domain-driven design: Tackling complexity in the heart of software*. Boston, MA: Addison-Wesley Professional.

Flanagan, D. (2011). *JavaScript: The definitive guide*. Sebastopol, CA: O'Reilly Media.

Forman, I. R., & Forman, N. (2005). *Java reflection in action*. Greenwich, CT: Manning.

Fowler, M. (2004, January 23). *Inversion of control containers and the dependency injection pattern*. Retrieved from martinfowler.com: http://martinfowler.com/articles/injection.html

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the design of existing code*. Boston, MA: Addison-Wesley Professional.

Fraiteur, G. (2008). User-friendly aspects with compile-time imperative semantics in .NET: An overview of PostSharp. *Seventh International Conference on Aspect-Oriented Software Development (AOSD)*. Brussels, Belgium: ACM.

Fraiteur, G. (2010, February 4). *DesignByContract*. Retrieved from code.google.com: http://code.google.com/p/postsharp-user-plugins/wiki/DesignByContract

Freeman, E., Robson, E., Bates, B., & Sierra, K. (2004). *Head first design patterns*. Sebastopol, CA: O'Reilly Media.

G´eraud, T., & Duret-Lutz, R. (2000). Generic programming redesign of patterns. *Proceedings of the 5th European conference on pattern languages of programs (EuroPLoP'2000)*. Irsee, Germany: Springer. Retrieved from http://www.coldewey.com/europlop2000

Gamma, E., Helm, R., Johnson, R., & Vlissides, R. (1994). *Design patterns, elements of reusable object-oriented software*. Boston, MA: Addison-Wesley.

Garcia, G. A. (2009a, February 28). *Componentized adapter pattern*. Retrieved from perfectjpattern.sourceforge.net: http://perfectjpattern.sourceforge.net/dp-adapter.html

Garcia, G. A. (2009b, February 28). *Componentized decorator pattern*. Retrieved from perfectjpattern.sourceforge.net: http://perfectjpattern.sourceforge.net/dp-decorator.html

Garcia, G. A. (2009c, February 28). *Componentized chain of responsibility pattern*. Retrieved from perfectjpattern.sourceforge.net: http://perfectjpattern.sourceforge.net/dp-chainofresponsibility.html

Garcia, G. A. (2009d, February 28). *Componentized composite pattern*. Retrieved from perfectjpattern.sourceforge.net: http://perfectjpattern.sourceforge.net/dp-composite.html

Garcia, G. A. (2009e, February 28). *Componentized command pattern*. Retrieved from perfectjpattern.sourceforge.net: http://perfectjpattern.sourceforge.net/dp-command.html

Gasiūnas, V., Satabin, L., Mezini, M., Núñez, A., & Noyé, J. (2010). *Declarative events for object-oriented programming*. Technical Report, INRIA. Retrieved from http://hal.inria.fr/inria-00494645/en/

Gasparis, E. N., & Eden, A. (2008). LePUS3: An object-oriented design description language. *Proceedings of 5th international conference on diagrammatic representation and inference (Diagrams'08)* (pp. 364-367). Herrsching, Germany: Springer.

Gil, J., & Lorenz, D. H. (1998, March). Design patterns vs. language design. *Computer, 31*(3), 118-120.

Groovy. (2011). *Singleton pattern*. Retrieved from groovy.codehaus.org: http://groovy.codehaus.org/Singleton+Pattern

Hannemann, J., & Kiczales, G. (2002, November). Design pattern implementation in Java and AspectJ. *SIGPLAN Not., 37*(11), 161-173.

Harrison, T., & Schmidt, D. C. (1997). *Thread-specific storage - An object behavioral pattern for efficiently accessing per-thread state*. Washington, DC: Washington University.

Hedin, G. (1997). *Language support for design patterns using attribute extensions*. Aarhus, Denmark: Aarhus University.

Hejlsberg, A., & Torgersen, M. (2007, March). *C# 3.0 new features*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/bb308966.aspx

Hejlsberg, A., Torgersen, M., Wiltamuth, S., & Golde, P. (2010). *C# programming language (Covering C# 4.0)* (4th ed.). Boston, MA: McGraw Hill Professional.

Hugunin, J. (2007, April 30). *A dynamic language runtime (DLR)*. Retrieved from http://blogs.msdn.com: http://blogs.msdn.com/b/hugunin/archive/2007/04/30/a-dynamic-language-runtime-dlr.aspx

Ichbiah, J. D., Krieg-Brueckner, B., Wichmann, B. A., Barnes, J. G., Roubine, O., & Heliard, J. (1979). Rationale for the design of the Ada® programming language. *SIGPLAN Not., 14*(6b), 1-261.

Ierusalimschy, R. (2003). *Programming in Lua*. Rio de Janeiro, Brazil: Lua.org.

Ionescu, S. (2005, May 26). *SingletonAttribute*. Retrieved from http://docs.codehaus.org: http://docs.codehaus.org/display/BOO/SingletonAttribute

Jagger, J., Perry, N., & Sestoft, P. (2007). *Annotated C# standard*. San Francisco, CA: Morgan Kaufmann.

J'arvi, J., Freeman, J., & Crowl, L. (2007). *Lambda expressions and closures: Wording for monomorphic lambdas*. The C++ Standards Committee. Retrieved from www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2413.pdf

Jesse, L., & Xie, D. (2008). *Programming C# 3.0.* Sebastopol, CA: O'Reilly Media.

Jézéquel, J. M., Train, M., & Mingins, C. (1999). *Design patterns and contracts.* Boston, MA: Addison-Wesley Longman.

Jon, S. (2010). *C# in depth.* Greenwich, CT: Manning.

Kennedy, A. (2006, November 6). *C# is a functional programming language.* Retrieved from http://fop.cs.nott.ac.uk/: http://fop.cs.nott.ac.uk/fun/nov-06/FunPm.pdf

Kerievsky, J. (2004). *Refactoring to patterns.* Boston, MA: Addison-Wesley Professional.

Kiczales, G., Lamping, J., Mehdhekar, A., Maeda, C., Lopes, C. V., Loingtier, J., & Irwin, J. (1997). Aspect-oriented programming. *ECOOP'97 — Object-oriented programming* (pp. 220-242). Jyväskylä, Finland: Springer.

KJärvi, J., & Freeman, J. (2008). Lambda functions for C++0x. *SAC '08* (pp. 178-183). Fortaleza, Brazil: ACM.

Klint, P. (1993). A meta-environment for generating programming environments. *ACM transactions on software engineering and methodology (TOSEM), 2*(2), 176-201.

Knuth, D. (1968). *The art of computer programming* (Vol. 1). Bonston, MA: Addison-Wesley.

Koenig, A., & Moo, B. E. (2005). Templates and duck typing. *Dr. Dobbs.* Retrieved from http://drdobbs.com/184401971

Landin, P. (1965). A correspondence between ALGOL 60 and Church's lambda-notation. *Communications of the ACM, 8*, 89–101.

Lea, D. (1999). *Concurrent programming in Java: Design principles and patterns.* Boston, MA: Addison-Wesley Longman.

Lee, A. H., & Zachary, J. L. (1995). Reflections on metaprogramming. *IEEE transactions on software engineering*, 883-893.

Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). *Introduction to algorithms.* Boston, MA: MIT Press.

Liberty, J. (2001). *Programming C#.* Sebastopol, CA: O'Reilly Media.

Lloyd, J. W. (1994). Practical advantages of declarative programming. *Joint conference on declarative programming (GULP-PRODE '94).* Peñiscola, Spain.

López Muñoz, J. M. (2008, August 13). *Boost.Flyweight Tutorial.* Retrieved from http://www.boost.org: http://www.boost.org/doc/libs/1_40_0/libs/flyweight/doc/tutorial/index.html

Lowy, J. (2007). *Programming WCF services.* Sebastopol, CA: O'Reilly Media.

Malenfant, J., Jacques, M., & Demers, F. N. (1996). A tutorial on behavioral reflection and its implementation. In G. Kiczales (Ed.), *Proceedings of the reflection '96 conference*, (pp. 1-20). Xerox Palo Alto Research Center, San Francisco, CA.

Mariani, J. (1999). Mix-Ins (Steve's ice cream, Boston, 1975). In J. Mariani, *The encyclopedia of american food and drink: With more than 500 recipes for american classics* (p. 166). New York, NY: Lebhar-Friedman.

Matsumoto, Y. (2001). *Ruby in a nutshell.* Sebastopol, CA: O'Reilly Media.

McConnell, S. (1993). *Code complete: A practical handbook of software construction.* Redmond, WA: Microsoft Press.

Mertes, T. (2011, October 14). *Seed7 - The extensible programming language.* Retrieved from seed7.sourceforge.net: http://seed7.sourceforge.net/manual/objects.htm#multiple_dispatch

Meyer, B. (1986). Genericity vs inheritance. *OOPSLA (First ACM conference on object-oriented programming systems, languages and applications)* (pp. 391–405). Portland, OR: ACM.

Meyer, B. (1991). *Eiffel : The language.* Upper Saddle River, NJ: Prentice Hall.

Meyer, B. (1992, October). Applying "Design by contract". *Computer (IEEE)*, 40–51.

Meyer, B. (2000). *Object-oriented software construction.* Upper Saddle River, NJ: Prentice Hall.

Meyer, B. (2001). Overloading vs object technology. *Journal of object-oriented programming (JOOP), 14*(4), 3-7.

Meyer, B., & Arnout, K. (2006). Componentization: the Visitor example. *Computer*, 23-30.

Michaelis, M. (2010). *Essential C# 4.0* (4th ed.). Sebastopol, CA: Addison-Wesley.

Microsoft. (2005). *What's new in the C# 2.0 language and compiler.* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/7cz8t42e(VS.80).aspx

Microsoft. (2007). *C# language specification*. Retrieved from http://msdn.microsoft.com: http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/CSharp%20Language%20Specification.doc

Microsoft. (2010a). *Action(of T) delegate.* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/018hxwa8.aspx

Microsoft. (2010b). *Anonymous methods (C# programming guide).* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/0yw3tz5k.aspx

Microsoft. (2010c). *Attribute class.* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/system.attribute.aspx

Microsoft. (2010d). *Attributes (C# and Visual Basic).* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/z0w1kczw(v=VS.100).aspx

Microsoft. (2010e). *Delegates (C# programming guide).* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/ms173171(v=VS.100).aspx

Microsoft. (2010f). *Dictionary(of TKey, TValue) class.* Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/xfhwa508.aspx

Microsoft. (2010g, October). *Extension methods (C# programming guide)*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/bb383977.aspx

Microsoft. (2010h). *Func(of T, TResult) delegate*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/bb549151.aspx

Microsoft. (2010i). *Lambda expressions (C# programming guide)*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/bb397687.aspx

Microsoft. (2010j). *MethodInfo class*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/system.reflection.methodinfo.aspx

Microsoft. (2010k). *SerializableAttribute Class*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx

Microsoft. (2010l). *SortedDictionary(Of TKey, TValue) Class*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/f7fta44c.aspx

Microsoft. (2010m). *The Obsolete attribute*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/aa664623(v=vs.71).aspx

Microsoft. (2010n). *ThreadStaticAttribute Class*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/system.threadstaticattribute(VS.71).aspx

Microsoft. (2010o, April). *Microsoft Unity 2.0*. Retrieved from msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/ff663144.aspx

Microsoft. (2010p, September). *Object.MemberwiseClone Method*. Retrieved from msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/system.object.memberwiseclone.aspx

Microsoft. (2010q, September). *ICloneable.Clone Method*. Retrieved from msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/system.icloneable.clone(v=VS.100).aspx

Microsoft. (2011a). *DevLabs: Code contracts*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx

Microsoft. (2011b). *Using Type dynamic (C# Programming Guide)*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/dd264736.aspx

Microsoft. (2011c). *Welcome to DevLabs*. Retrieved from http://msdn.microsoft.com: http://msdn.microsoft.com/en-us/devlabs/cc950527

Miller, J. (2008, October). *Patterns in practice: Cohesion and coupling*. Retrieved from msdn.microsoft.com: http://msdn.microsoft.com/en-us/magazine/cc947917.aspx

Mitchell, J. C., Mitchell, & Krzysztof, A. (2003). *Concepts in programming languages*. New York, NY: Cambridge University Press.

Mitchell, R., & McKim, J. (2001). Design by contract, by example. *Proceedings of the 39th international conference and exhibition on technology of object-oriented languages and systems (TOOLS39)* (p. 430). Washington, DC: IEEE Computer Society.

Moon, D. A. (1986). Object-oriented programming with Flavors. *Conference proceedings on object-oriented programming systems, languages and application (OOPLSA '86)* (pp. 1-8). Portland, OR: ACM.

Musser, D. R., & Stepanov, A. A. (1989). *The Ada generic library: Linear list processing packages.* New York, NY: Springer-Verlag.

Nagel, C., Evjen, B., Glynn, J., & Watson, K. (2010). *Professional C# 4.0 and .NET 4.* Birmingham, United Kingdom: Wrox Press.

Nierstrasz, O., Bergel, A., Denker, M., Ducasse, S., Gälli, M., & Wuyts, R. (2005). On the revival of dynamic languages. In *Software composition* (Vol. 3628, pp. 1-13). Berlin, Germany: Springer.

Nilsson, J. (2006). *Applying domain-driven design and patterns: With examples in C# and .NET.* Boston, MA: Addison-Wesley.

O'Brien, T. M. (2004). *Jakarta commons cookbook.* Sebastopol, CA: O'Reilly Media.

Odersky, M., Spoon, L., & Venners, B. (2011). *Programming in Scala: A comprehensive step-by-step.* Bury St Edmunds, United Kingdom: Artima.

Pattison, T., & Box, D. (2000). *Programming distributed applications with COM & Microsoft Visual Basic.* Redmond, WA: Microsoft Press.

Perrotta, P. (2010). *Metaprogramming Ruby: Program like the Ruby pros* (1st ed.). Raleigh, NC, and Dallas, TX: The Pragmatic Programmers, LLC.

Pierce, B. C. (2002). *Types and programming languages* . Boston, MA: MIT Press.

Pinto, M., Amor, M., Fuentes, L., & Troya, M. J. (2001). Run-time coordination of components: Design patterns vs. component & aspect based platforms. *European conference on object-oriented programming (ECOOP 2001 workshop on advanced separation of concerns).* Budapest, Hungary: Springer.

Pree, W. (1995). *Design patterns for object-oriented software development.* Boston, MA: Addison-Wesley.

Purdy, D., & Richter, J. (2002, January). *Exploring the observer design pattern.* Retrieved from msdn.microsoft.com: http://msdn.microsoft.com/en-us/library/ee817669.aspx

Python Software Foundation. (2011, October 30). *Python v2.7.2 documentation.* Retrieved from http://docs.python.org/: http://docs.python.org/

Quesnel, P. (2005, October 31). *Useful things about Boo.* Retrieved from docs.codehaus.org: http://docs.codehaus.org/display/BOO/Useful+things+about+Boo

Rahien, A. (2010). *DSLs in Boo: Domain specific languages in .NET.* Shelter Island, NY: Manning.

Reade, C. (1989). *Elements of functional programming.* Boston, MA: Addison-Wesley Longman.

Redpath, L. (2009, February 15). *http://lukeredpath.co.uk/blog/decorator-pattern-with-ruby-in-8-lines.html.* Retrieved from http://lukeredpath.co.uk: http://lukeredpath.co.uk/blog/decorator-pattern-with-ruby-in-8-lines.html

Ruby-Doc.Org. (2011, October 7). *Ruby-Doc.Org*. Retrieved from http://www.ruby-doc.org/: http://www.ruby-doc.org/

Samko, V., Willcock, J., Järvi, J., Gregor, D., Lumsdaine, A., & Stroustrup, B. (2006, September). Lambda expressions and closures for C++. *Sci. Comput. Program.*(9), 762-772. Retrieved from http://www.open-std.org: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n1968.pdf

Savitch, W. (1993). *Turbo Pascal 7.0*. Boston, MA: Addison Wesley.

Schmidt, D. C. (1995). Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM, 38*(10), 65-74.

Schmidt, D. C., & Harrison, T. (1996). Double-checked locking - An optimization pattern for efficiently initializing and accessing thread-safe objects. *3rd Annual Pattern Languages Design Conference*. Monticello, IL.

Schmidt, D. C., & Huston, S. D. (2002). *C++ network programming, Volume 2: Systematic reuse with ACE and frameworks* (1 edition ed.). Boston, MA: Addison-Wesley Professional.

Schmidt, D. C., Stal, M., Rohnert, H., & Buschmann, F. (2000). *Pattern-oriented software architecture: Patterns for concurrent and networked objects*. New York, NY: John Wiley & Sons.

Scott, M. L. (2009). *Programming language pragmatics*. San Francisco, CA: Morgan Kaufmann.

Seibel, P. (2004). *Practical common Lisp*. New York, NY: Apress.

Sobel, J. M., & Friedman, D. P. (1996). An introduction to reflection-oriented programming. In G. Kiczales (Ed.), *Proceedings of reflection'96*, (pp. 263-288). San Francisco, CA.

Soukup, J. (1995). Implementing patterns. In J. Coplien, & D. Schmidt, *Pattern languages of program design* (pp. 395-412). New York, NY: ACM Press/Addison-Wesley.

Stasiak, P. (2008). *Chain.NET*. Retrieved from nchain.sourceforge.net: http://nchain.sourceforge.net/license.html

Stein, D., & Shah, D. (1992). Implementing lightweight threads. *Summer '92 USENIX* (pp. 1-10). San Antonio, TX: USENIX.

Stepanov, A., & Lee, M. (1995). *The standard template library*. Palo Alto, CA: Hewlett Packard Laboratories.

Stevens, A. (1998). C++ class template library that implements undo operations of interactive programs. *Dr.Dobb's*. Retrieved from http://drdobbs.com/cpp/184410722

Stoyan, H. (1984). Early LISP history (1956-1959). *Proceedings of the 1984 ACM symposium on LISP and functional programming* (pp. 299-310). Austin, TX: ACM.

Stranden, H. (2011, 06 03). *Copyable: A framework for copying or cloning .NET objects*. Retrieved from Circles & Crosses: http://ox.no/software/copyable

Stroustrup, B. (1987). Multiple inheritance for C++. *Proc. European UNIX user's group*, (pp. 189-208). Helsinki, Finland.

Stroustrup, B. (1994). *The design and evolution of C++*. New York, NY: ACM Press/Addison-Wesley.

Stroustrup, B. (2007). Evolving a language in and for the real world: C++ 1991-2006. *Proceedings of the third ACM SIGPLAN conference on history of programming languages* (pp. 4-1, 4-59). San Diego, CA: ACM.

Szyperski, C. (2002). *Component software: Beyond object-oriented programming*. Boston, MA: Addison-Wesley Longman.

Taibi, T. (2007). *Design patterns formalization techniques*. Hershey, PA: IGI.

Taibi, T., Herranz, A., & Moreno-Navarro, J. J. (2009, March-April). Stepwise refinement validation of design patterns formalized in TLA+ using the TLC model checker. *Journal of object technology, 8*(2), 137-161.

Tanguay-Carel, M. (2007). *GoF patterns in Ruby*. Retrieved from www.scribd.com: http://www.scribd.com/doc/396559/gof-patterns-in-ruby

Tenenbaum, A. M., Langsam, Y., & Augenstein, M. J. (1990). *Data structures using C*. Upper Saddle River, NJ: Prentice-Hall.

Thomas, P. G., & Weedon, R. A. (1997). *Object-oriented programming in Eiffel*. Boston, MA: Addison-Wesley.

Thompson, S. (1999). *Haskell: The craft of functional programming*. Boston, MA: Addison-Wesley Longman.

Torgersen, M. (2007). Querying in C#: How language integrated query (LINQ) works. *OOPSLA '07 Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications companion* (pp. 852-853). Montreal, Quebec, Canada: ACM.

Torgersen, M. (2008). New features in C# 4.0. Microsoft. Retrieved from http://msdn.microsoft.com/en-us/vcsharp/ff628440

Tratt, L. (2009). Dynamically typed languages. *Advances in computers, 77*, 149-184.

van Rossum, G. (2008, February 21). *Python library reference*. Retrieved from docs.python.org: http://docs.python.org/release/2.5.2/lib/lib.html

Vandevoorde, D., & Josuttis, N. M. (2003). *C++ Templates: The complete guide*. Boston, MA: Addison-Wesley.

Wampler, D., & Payne, A. (2009). *Programming Scala: Scalability = functional programming + objects*. Boston, MA: O'Reilly Media.

Weiss, M. A. (1999). *Data structures and problem solving with C++*. Boston, MA: Addison-Wesley Longman.

Williams, C. (2006 , 10 31). *Patterns in Ruby: Singleton Pattern.* Retrieved from Ruby Buzz Forum: http://www.artima.com/forums/flat.jsp?forum=123&thread=182870

Wirth, N. (1976). *Algorithms + data structures = programs* (1St Edition edition ed.). New Delhi, India: Prentice Hall of India Pvt. Limited.

Zimmer, W. (1995). Relationships between design patterns. In *Pattern languages of program design* (pp. 345-364). New York, NY: ACM Press/Addison-Wesley.

APPENDIX I

In some of the reusable pattern components shown in this thesis, operations are registered against method contracts available on a certain interface. The registration is achieved by either passing in the method name as a **`string`** argument or using the **`MethodInfo`** .NET class. The code below is an extract from the **`AutoAdapter<TTarget, TAdaptee>`** component, showing the register methods available on it:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public void RegisterAction(string methodName, AdapterAction<TAdaptee> operation) { … }
public void RegisterAction(MethodInfo method, AdapterAction<TAdaptee> operation) { … }
// … M O R E …

public void RegisterFunc<TResult>(string methodName,
                                  AdapterFunc<TAdaptee, TResult> operation) { … }
public void RegisterFunc<TResult>(MethodInfo method,
                                  AdapterFunc<TAdaptee, TResult> operation) { … }
// … M O R E …
```

Adaptee operations can now be registered with an instance of the **`AutoAdapter<TTarget, TAdaptee>`** component. The example code below shows the registration of an Adaptee action against a **`Foo`** method available on the Target. In the example the **`Foo`** method is registered using the name of the method represented as a **`string`**:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
Adapter.RegisterAction("Foo", (x) => x("Hello World");
```

The **`Foo`** method can also be registered using a **`MethodInfo`** .NET class, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
Adapter.RegisterAction(typeof(TAdaptee).GetMethod("Foo"), (x) => x("Hello World");
```

Neither of the above registration methods is elegant and both rely on non type-safe and runtime reflection mechanisms.

C# dynamics can be used to implement a more elegant and declarative mechanism for method registration. A **`Register`** property can be added to the **`AutoAdapter<TTarget, TAdaptee>`** component, as seen below, that returns a **`dynamic`** type:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class AutoAdapter<TTarget, TAdaptee> : IDynamicInvoker
    where TTarget : class {
    private TAdaptee _adaptee;
    // … S N I P …

    public AutoAdapter(TAdaptee adaptee) { … }

    // … S N I P …

    public void RegisterAction(string methodName, AdapterAction<TAdaptee> operation) { … }
    public void RegisterAction(MethodInfo method, AdapterAction<TAdaptee> operation) { … }
    // … M O R E …

    public void RegisterFunc<TResult>(string methodName,
                                      AdapterFunc<TAdaptee, TResult> operation) { … }
    public void RegisterFunc<TResult>(MethodInfo method,
                                      AdapterFunc<TAdaptee, TResult> operation) { … }
    // … M O R E …

    // Register any adapter operation against a method contract on the TTarget
    public RegisterAny(MethodInfo method, MethodInfo operation) {
        // Validate that the operation has a valid AdapterAction or AdapterFunc and
        // that method is avaliable on the TTarget interface
        Validate(method, operation);

        // Add the operation to the internal dictionary against the method
        AddToDictionary(method, operation);
    }

    // Register any adapter operation against a method contract on the TTarget using
    // C# 4.0 dynamics
    public dynamic Register { return new AdapterMethodRegister<TTarget, TAdaptee>(this); }

    // … S N I P …

    public object Invoke(string methodName, object[] args) { … }
    public TTarget Target { … }
}
```

The **Register** property can now be used to register any valid operation on an instance of the component, as shown below:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
Adapter.Register.Foo = (x) => x("Hello World");
```

The **Register** property returns and instance of the **AdapterMethodRegister** class as a **dynamic** type. The **AdapterMethodRegister** APL class inherits from the C# **DynamicObject** class, located in the **System.Dynamic** .NET namespace, which makes it possible to inject new behaviour dynamically during runtime. The code snippet below shows the implementation of the **AdapterMethodRegister** class:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class AdapterMethodRegister<TTarget, TAdaptee>: DynamicObject {
    private readonly AutoAdapter<TTarget, TAdaptee> _adapter;

    public AutoAdapter(AutoAdapter<TTarget, TAdaptee> adapter) { _adapter = adapter; }

    public override bool TrySetMember(SetMemberBinder binder, object value) {
        // Validate that the method adhere to the signature of a AdapterAction or AdapterFunc delegate
        Validate(binder, value);

        // Register the adapter method with the adapter
        adapter.RegisterAny(GetContractMethod(binder, value), GetAdapterOperation(binder, value));
    }

    // … S N I P …
}
```

In the code above the **`TrySetMember`** method registers the received method on its internal instance of the **`AutoAdapter`**. The **`AdapterMethodRegister`** receives a method via the **`TrySetMember`** when a user tries to dynamically add a method on it during runtime, as seen in the previously shown **`Adapter.Register.Foo = (x) => x("Hello World")`** example. When called, the **`Register`** property always returns a new instance of the **`AdapterMethodRegister`** class, which was created with the underlying **`AutoAdapter`** instance:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
// Register any adapter operation against a method contract on the TTarget using
// C# 4.0 dynamics
public dynamic Register { return new AdapterMethodRegister<TTarget, TAdaptee>(this); }
```

Although this mechanism is more elegant than the original registration methods, it is still not type-safe. For example, if the **`Foo`** method on the Target interface is refactored to **`FooBar`**, then the **`Adapter.Register.Foo`** registration is not changed to **`FooBar`**. There is thus no direct type-safe relationship between the **`Foo`** method available on the Target interface and the **`Foo`** method used on the **`AutoAdapter`** registration. Unfortunately no language feature exists in C# whereby a user can reference the meta-information of a method available on an interface in a type-safe manner, as shown in the example code below:

```
C# (APL Conceptual Example)
-----------------------------------------------------------------------------------------------------------
Adapter.RegisterAction(ITarget.Foo, (x) => x("Hello World");
```

In the conceptual code above, a compile time error is generated if the **`Foo`** method on the **`ITarget`** interface is changed. Furthermore, if the **`Foo`** method is changed on the **`ITarget`** interface using

powerful refactoring tools, then the referenced **`Foo`** method in the **`RegisterAction`** method will also change.

Lambda expressions (expressions trees) (Albahari & Albahari, 2007, p. 317) can be used to solve the type-safe registration problem. The registration syntax may be a little convoluted, the solution, however, is fully type-safe:

```
C# (APL Example)
-----------------------------------------------------------------------------------------------------------
Adapter.RegisterAction("Foo", (x) => x("Hello World"); // Non type-safe
Adapter.RegisterAction(t => t.Foo, (x) => x("Hello World"); // Type-safe
```

The above registration technique is thus type safe at the cost of a slightly more convoluted syntax and of having to do a little decomposition of the expression tree to find the specific method name that is being referred to.

APPENDIX II

This appendix shows a performance test for *duck typing* (Koenig & Moo, 2005) used in this thesis. It specifically shows the performance of a method call on a dynamically created class. Each method call against the dynamically created class is routed to the **`Invoke`** method, which is enforced by the **`IDynamicInvoke`** interface, as seen below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public sealed class AutoAbstractFactory<TInterface> : IDynamicInvoke {
    // … S N I P …

    public object Invoke(string methodName, object[] args) {
        // … S N I P …
        var componentOperation = GetComponentOperation(methodName, args);
        if(componentOperation != null) {
            return componentOperation.DynamicInvoke(args);
        }

        return null;
    }
}
```

The test uses the **`AutoAbstractFactory<TInterface>`** reusable component. In the test two factory instances are created. One factory uses the **`AutoAbstractFactory<TInterface>`** component and the other factory instance is created normally. Each factory is used to create a Product from where a method is invoked on each Product instance. The method invocation on the Product created by the **`AutoAbstractFactory<TInterface>`** component will thus route its invocation to the **`Invoke`** method, which does create a performance overhead.

In Table 5 the times listed, shown in milliseconds and measured over 10000 traversals, compare the two method calls. The testing was done on an Intel® Core™ i5-2520M CPU @ 2.50GHz running Windows 7 Professional 64-bit with 6.00 GB of RAM. In order to reduce JIT influences from the timings, the test program executes one method call before starting the real test. All invoked methods are therefore JIT'ed (Bishop & Horspool, 2008) prior to the timing test:

Table 5: Duck typing performance test.

Test	Normal Invocation	Duck Typing Invocation
1	0.0307ms	0.2440ms
2	0.0307ms	0.2454ms
3	0.0311ms	0.2282ms
4	0.0311ms	0.2245ms
5	0.0311ms	0.2261ms
6	0.0307ms	0.2241ms

It is clear from the above table that, as expected, normal method invocations in C# are faster (on average by 7 times), than *duck typing* invocations.

APPENDIX III

The following shows a performance test for the **DynamicChainOfResponsibility** component discussed in this thesis. It specifically shows the performance of a handler method call invocation, where the method was dynamically added to an instance of the component during runtime. Each handler method call against a **DynamicChainOfResponsibility** instance is routed to the **TryInvokeMember** method, which is enforced by the **DynamicObject** abstract class, as seen below:

```
C# (APL)
-----------------------------------------------------------------------------------------------------------
public class DynamicChainOfResponsibility : DynamicObject {
    // … S N I P …

    public override bool TryInvokeMember(InvokeMemberBinder binder,
                                         object[] args,
                                         out object result) {
            ChainOfResponsibilityEx.Handled = false;
            var excecutedHandler = false;
            result = null;

            if(_members.ContainsKey(binder.Name) &&
                _members[binder.Name] is Delegate) {
                result = ((Delegate)_members[binder.Name]).DynamicInvoke(args); // Dynamic call
                excecutedHandler = true;
            }

            if(!ChainOfResponsibilityEx.Handled && _successor != null) {
                return _successor.TryInvokeMember(binder, args, out result);
            }

            return excecutedHandler;
        }

        public override IEnumerable<string> GetDynamicMemberNames() { return _members.Keys; }
    }

    // … S N I P …
}
```

In the test two handlers are created. One handler uses the **DynamicChainOfResponsibility** component and the other handler is created normally. The method invocation on the **DynamicChainOfResponsibility** handler will thus route its invocation to the **TryInvokeMember** method, which does create a performance overhead.

In Table 6 the times listed, shown in milliseconds and measured over 10000 traversals, compare the two handler method calls. The testing was done on an Intel® Core™ i5-2520M CPU @ 2.50GHz running Windows 7 Professional 64-bit with 6.00 GB of RAM. In order to reduce JIT influences from

the timings, the test program executes one method call before starting the real test. All invoked methods are therefore JIT'ed (Bishop & Horspool, 2008) prior to the timing test:

Table 6: DynamicChainOfResponsibility performance test.

Test	Normal Invocation	Dynamic Invocation
1	0.0032ms	24.5492ms
2	0.0036ms	23.7980ms
3	0.0024ms	23.0841ms
4	0.0032ms	24.0849ms
5	0.0032ms	24.2783ms
6	0.0041ms	24.1531ms

It is clear from the above table that normal method invocations in C# are much faster than dynamic method invocations defined on the `DynamicChainOfResponsibility` component.

APPENDIX IV

Gasparis, Nicholson and Eden show the basic set of symbols used in LePUS3 as illustrated below (Gasparis & Eden, 2008):

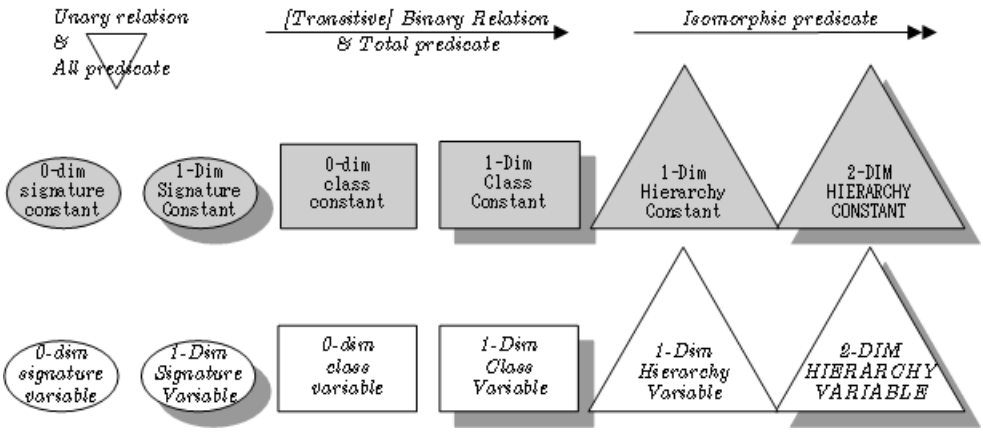

Figure 44. Basic set of symbols used in LePUS3.

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y