University of Pretoria etd – Peer, E S (2005)

Chapter 3

Design Patterns

"A common mistake that people make when trying to design something completely foolproof is to underestimate the ingenuity of complete fools."
— Douglas Adams

Design patterns succinctly encapsulate the knowledge of experienced programmers by specifying proven solutions to commonly recurring software design scenarios. Patterns are not specifically invented or designed, rather, they are discovered by observing best practices and recurring design solutions that have proven to be useful, efficient, and extensible in existing software.

The Gang of Four [41], or GoF as the pioneers of the field are usually referred to, presented a catalogue identifying core design patterns which apply to Object Oriented Programming (OOP) in general. In addition, catalogues have since been compiled for the following:

- high level architectural patterns [19, 39];
- distributed systems and concurrency patterns [98];
- database programming patterns [86];
- language or framework specific patterns [4, 80].

Catalogues of design patterns enable software developers to draw upon documented experience instead of reinventing the wheel. Good design is difficult to accomplish, particularly for novice programmers, usually requiring a number of redesign iterations.

University of Pretoria etd – Peer, E S (2005)

Pattern catalogues consist of mature and successful designs that have been frequently found in software written by experienced programmers. In this way patterns capture the experience of experts, providing it in a concise and easy to digest form.

An entry in a design pattern catalogue consists of four essential components. Firstly, a short and descriptive pattern name. These names define a vocabulary for communicating about entire designs at a higher level of abstraction. Secondly, an outline of the problem and its context together specify when it is appropriate to apply the pattern. The most important element of any pattern is obviously the solution to this problem. Solutions are described in abstract terms, along with class structure diagrams, that can be applied as a template in many different concrete situations. Sample code demonstrating the usage of the pattern is often presented. Finally, the impact and known consequences of the pattern are listed.

Software implementing design patterns does not only benefit from the expert experience derived from the patterns. The patterns themselves serve as documentation for that software too. Scholars of design patterns should be able to understand the design of such software with little more documentation than a reference to the applicable pattern and a brief explanation of any unusual implementation details. Furthermore, programmers unfamiliar with design patterns can simply refer to the catalogue where the design is discussed in detail. The self documenting nature of code that uses patterns is an important reason for patterns being discussed in this work, otherwise the patterns that have been used in the implementation, although very useful in ensuring good design, may just as well have been considered an irrelevant implementation detail.

This chapter summarises those GoF patterns that are applicable to CILib and CiClops. The patterns are separated, based on their purpose, into three distinct categories: creational patterns, presented in Section 3.1; structural patterns, presented in Section 3.2; and behavioural patterns, presented in Section 3.3. The intention, describing the primary purpose of a pattern, is quoted directly from the GoF catalogue [41] as an introduction to each pattern. The patterns are summarised in a less rigid form than the GoF catalogue without many examples. Chapters 6 and 7 will serve as adequate examples where the implementations of these patterns are discussed. High level architectural and framework specific patterns are implicitly covered, as required, when platforms such as Java 2 Enterprise Edition (J2EE) are discussed in Chapter 5. This chapter concludes with a short discussion in Section 3.4

University of Pretoria etd – Peer, E S (2005)

3.1 Creational Patterns

The common theme amongst the creational patterns is delegating the details of object creation in a particular system, or client, to other classes external to the client that can vary independently. That is, there is a decoupling between the use of objects and their creation.

Section 3.1.1 presents the *Abstract Factory* pattern, where the instantiation of objects is delegated to a polymorphic interface. The *Builder* pattern, in Section 3.1.2, abstracts the process of instantiating a complex set of objects into a reusable unit that can be used to construct different representations using the same build process. Section 3.1.3 discusses a pattern for creating objects by cloning existing prototype objects. Finally, the *Singleton* pattern, in Section 3.1.4, limits the instances of a given class.

3.1.1 Abstract Factory

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes" — GoF

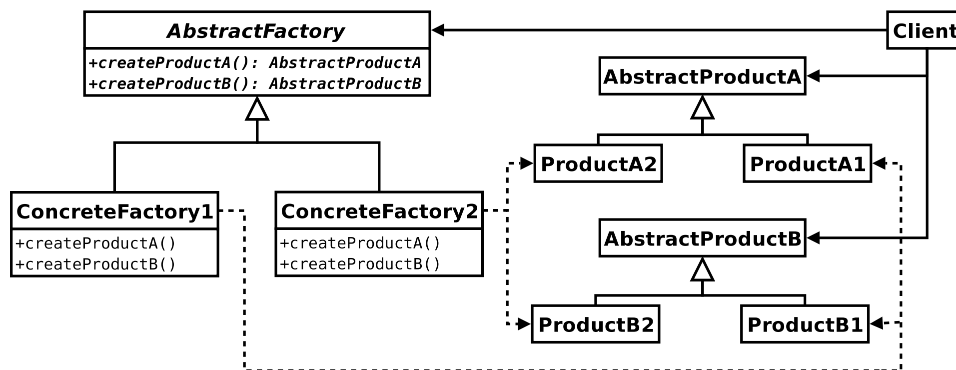

Figure 3.1: Abstract Factory

Figure 3.1 illustrates the design of the *Abstract Factory* pattern. The core participant in the pattern is the abstract factory interface which defines the contract that its client uses to instantiate objects. The most important aspect of the pattern is that the client is never exposed to the implementation details, including the class names, of the concrete factories or the classes that they create. Each concrete factory is responsible for producing its own

University of Pretoria etd – Peer, E S (2005)

family of concrete products with the only requirement being that the abstract interfaces are satisfied. Thus, if the client is written to conform to the abstract interfaces then the concrete factories, and by extension the products that they produce, may be interchanged without requiring changes to the client.

The decoupling of a system from how its products are created provides immense flexibility, to the extent that the entire behaviour of the system can be altered by simply changing the factory used to create the objects that it uses. Furthermore, dependencies between a family of products can be enforced, since a single concrete factory is responsible for all the different products at any given time. Unfortunately, a drawback of the design is that adding new products is difficult, since it entails a modification of the abstract factory interface. Such an interface change translates into changes to all existing concrete factory implementations to support the new product which, in turn, is likely to require new product implementations to be defined as well.

3.1.2 Builder

"Separate the construction of a complex object from its representation so that the same construction process can create different representations" — GoF

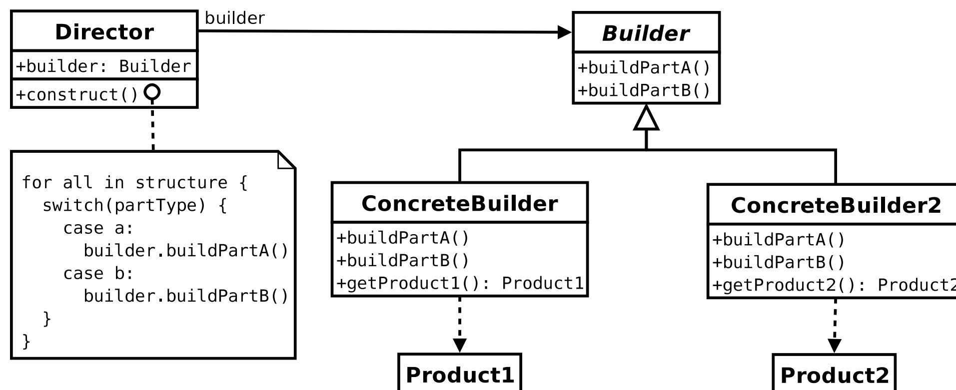

Figure 3.2: Builder

The *Builder* pattern, depicted in Figure 3.2, assembles complex objects in a piecemeal fashion, building them part by part. A director class controls the construction process while delegating the creation and assembly of parts of the product to an abstract builder

University of Pretoria etd – Peer, E S (2005)

interface. Thus, a concrete builder has jurisdiction over the implementation details of the parts as well as how they are assembled to create a larger complex product. Typically, the functioning of the director is dictated by the traversal of some data structure or document. The builder interface exposes the set of operations that may be utilised by a director to construct a product according the structure it traverses.

Products produced by a given concrete builder implementation need not conform to any given interface. Thus, it is possible for two different concrete builders to create two very different products using the same construction process, as specified by the director. Alternatively, different directors may use the same builder interface permitting different structures to be rendered into the same product representation. In addition, the director provides finer control over the construction process than the *Abstract Factory* which creates each of its products in a single shot.

3.1.3 Prototype

> *"Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype"* — *GoF*

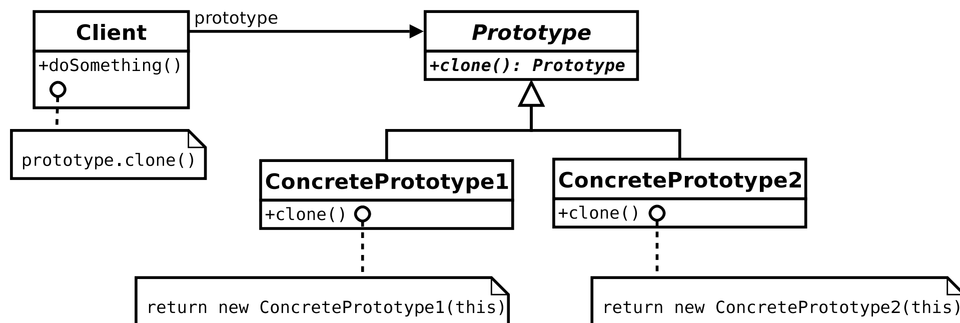

Figure 3.3: Prototype

The *Prototype* pattern creates new objects by copying, or cloning, existing objects. Importantly, the client making a clone of an object need not know the type of object it is dealing with, only the fact that the object implements the prototype interface. The responsibility of making the copy falls on the object being cloned, as shown in Figure 3.3.

One of the key benefits of prototypes is that they enable a client to instantiate objects that have been configured at run time. That is, objects with different run time state or

University of Pretoria etd – Peer, E S (2005)

object structures that have been composed together in different ways at run time may conceptually be considered to be instances of different classes. The *Prototype* allows these different run time configurations of objects to be treated as new classes that can be instantiated like any other class. Thus, an application can be configured with new classes dynamically.

When used in conjunction with the *Abstract Factory*, the *Prototype* pattern can mitigate the need to create concrete factories for every product. Instead, a single factory can simply be configured with different prototype instances as products.

The clone operation typically performs a deep copy which has an obvious caveat pertaining to circular references. Prototypes containing any circular references need to take appropriate measures to prevent infinite looping.

3.1.4 Singleton

> *"Ensure a class only has one instance, and provide a global point of access to it" — GoF*

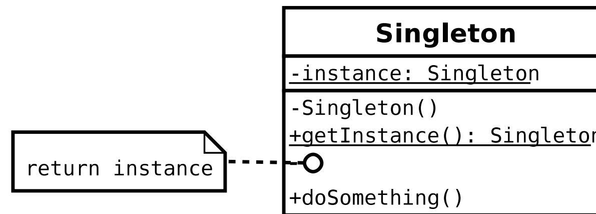

Figure 3.4: Singleton

The *Singleton* pattern, illustrated in Figure 3.4, is characterised by three properties. Firstly, any constructors are inaccessible so that clients can not arbitrarily create instances of the class. Secondly, the only existing instance is a static field, also known as a class scoped field, which is also not directly accessible to clients. Finally, a publicly accessible static method provides clients with access to the single instance. The single instance may be statically initialised or it may be initialised in a lazy fashion by the public accessor the first time it is called.

The purpose of the *Singleton* is to prevent a shared object from being instantiated by multiple clients. Limiting the number of instances not only saves memory, but more

University of Pretoria etd – Peer, E S (2005)

importantly, it prevents difficult to detect programming errors from occurring, where an object which is supposed to be shared is not being shared properly. Further, a singleton can be used as a namespace to store global application context cleanly, without resorting to global variables. Moreover, instead of restricting clients to a single instance, it is trivial to extend the pattern so that the implementation maintains a limited pool of objects for applications that require it.

3.2 Structural Patterns

Structural patterns describe methods to compose classes to form larger useful structures. That is, they illustrate flexible methods of interaction between classes by specifying how classes should be combined and used together.

The *Adapter* pattern, in Section 3.2.1, demonstrates how incompatible classes can be made compatible and used together. Section 3.2.2, the *Composite*, discusses a pattern that enables hierarchies of objects and individual objects to be treated in a uniform fashion. The *Decorator* pattern, which can be used to dynamically associate additional behaviour with objects, is discussed in Section 3.2.3. Complex systems of classes can be simplified into a single interface using the *Facade* in Section 3.2.4. Finally, the *Proxy* pattern provides a way to facilitate or control access to the objects which it stands in for.

3.2.1 Adapter

> *"Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces" — GoF*

Figure 3.5 illustrates the most common form of the *Adapter* pattern, particularly in languages that only support single inheritance. The adapter class maintains a reference to the object which it is adapting, the adaptee, while conforming to the target interface expected by the client. Another form of adapter inherits both the target and adaptee interfaces which may not always be possible in languages that do not support multiple inheritance. The multiple inheritance version has the advantage of being able to triv-

University of Pretoria etd – Peer, E S (2005)

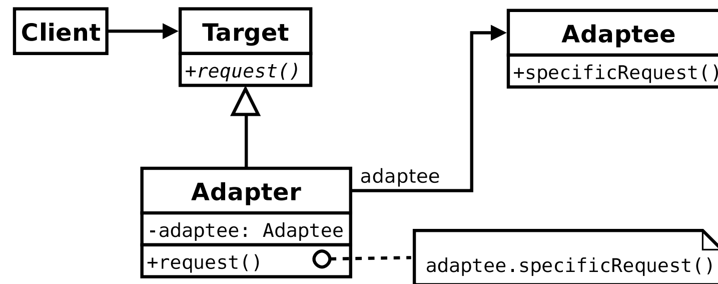

Figure 3.5: Adapter

ially override any operations belonging to the adaptee, if necessary, whereas the version presented here requires an auxiliary class to override adaptee operations.

The amount of work that needs to be done by the adapter is application specific and depends on how much the target interface differs from that of the adaptee. In some cases, particularly when reusing legacy classes in a new framework, all that may be required is changing the the name of an operation or converting the types of its arguments. In more extreme cases, the interface may be totally different, requiring more work to make the adaptee conform to the target interface expected in the context of the client.

3.2.2 Composite

> *"Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and the compositions of objects uniformly." — GoF*

The *Composite* pattern, depicted in Figure 3.6, represents hierarchical structures of objects in such a way that clients can treat the individual objects in exactly the same way as they treat the entire composite. Operations on leaf nodes in a composite structure behave according to the type of node that the operation is being executed on, whereas composite nodes typically delegate the requested operation to each of their child nodes. Hierarchies can be built recursively, since a composite node is itself a component which in turn contains components.

The primary benefit of the *Composite* pattern is also its weakness. The fact that clients should not need to differentiate between operations on leaf nodes and operations on composite nodes means that the root component interface needs to support all of the

University of Pretoria etd – Peer, E S (2005)

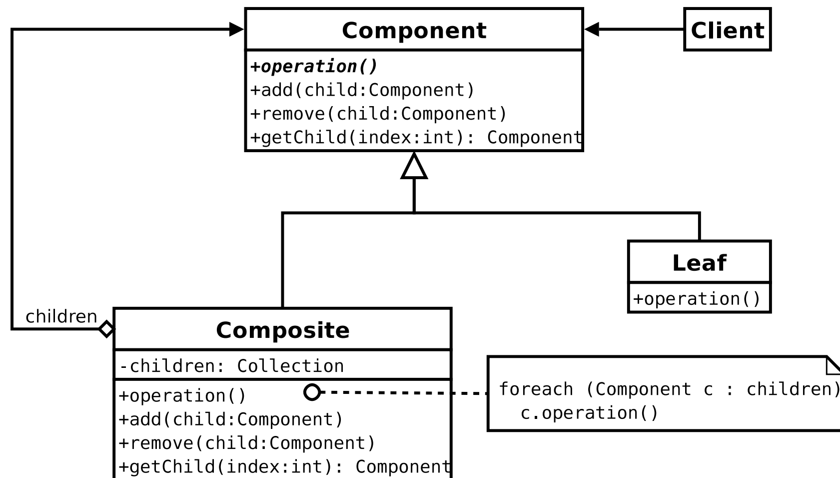

Figure 3.6: Composite

operations supported by any of the components, thus reducing type safety. For example, operations for maintaining the child nodes of a composite do not usually apply to leaf nodes, so these operations usually have an empty implementation in the root interface. Similarly, there may be operations specific to leaf nodes that do not make sense for composite nodes, or even other types of leaf node for that matter. Thus, even though all components must implement the same component interface by virtue of inheriting from it, some of them may have unexpected or default behaviours when certain operations are called.

3.2.3 Decorator

> *"Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality."* — *GoF*

Structurally, the *Decorator* pattern, in Figure 3.7, and the *Adapter* presented in Section 3.2.1 are similar. Both delegate operations prescribed by a target interface to another class which they reference, or wrap. In the case of the *Adapter*, the adaptee is an arbitrary class that must be made to conform to a target interface. The *Decorator*, however, delegates operations specified by the component interface to another class conforming to that same interface with the purpose of adding responsibilities to the original component, not to make the already compatible interfaces compatible with each other.

University of Pretoria etd – Peer, E S (2005)

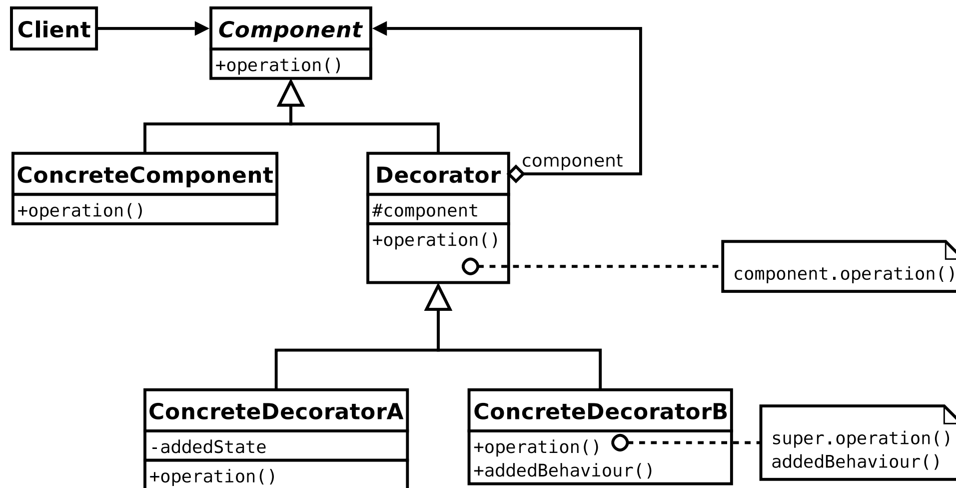

Figure 3.7: Decorator

Nevertheless, throughout design pattern literature, both the *Adapter* and the *Decorator* have been referred to by the same alternate name, namely the *Wrapper* pattern, probably owing to the fact that both have a similar structure.

Concrete decorator classes add a combination of additional state and behaviour to a target class without changing the interface that is exposed to the client. Typically, the base decorator class is simply an identity mapping for the operations defined by the component interface. That way, a concrete decorator need only override the operations necessary to achieve its goal. The primary benefit of the decorator is that these additional responsibilities can be dynamically added and removed from a component at run time, whereas extending the responsibilities of a class through normal inheritance is fixed at compile time and as such is less flexible. Concrete components need not implement seldom used functionality that can be added by decorators on an as needed basis. Unfortunately, decorators are not truly transparent, since clients cannot rely on the equivalence of decorators and their components based on their references.

3.2.4 Facade

> *"Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use"*
> — *GoF*

University of Pretoria etd – Peer, E S (2005)

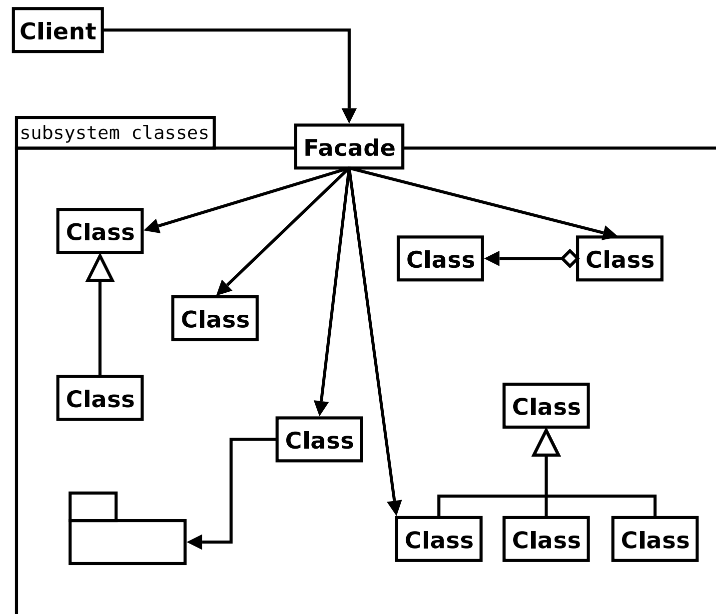

Figure 3.8: Facade

The *Facade* pattern, illustrated in Figure 3.8, decouples a complex system from its clients by providing a high level interface to access the system in a simplified way. The extra flexibility and extensibility that other design patterns bring to the table often has the net result of making a system of classes more complex. For example, a client may be able to configure a well designed system to better suit its needs by extending some of the classes that make up that system. The *Facade* provides a mechanism to counteract some of this complexity in the cases when a client does not need to alter the default behaviour of a system.

Structurally, the *Facade* is also similar to the *Adapter*, presented in Section 3.2.1, except that the facade typically maintains references to many objects within the system instead of only adapting the interface for a single class. In effect, the facade adapts the interfaces provided by an entire system and presents them as a single simplified interface to clients.

The most important feature, with respect to making a system more maintainable, is that the facade decouples the client from the system so that changes to the internals of the system do not affect clients. Further, the facade interface may be polymorphic so that the entire system implementation can be switched without the client's knowledge

University of Pretoria etd – Peer, E S (2005)

by simply changing the instance of the facade being used. The decoupling provided by the facade can also be extended to the interface between between different layers in a multi-layer framework. The refined interface reduces the communication between layers and thus reduces their dependency on one another while improving performance, particularly if the layers are implemented in different address spaces.

While the facade provides a simpler interface to the system, there is typically nothing preventing a client from accessing system classes directly. In fact, the facade interface may require the client to do so by accepting as arguments or returning system specific classes. Further, the client may need to use some complex features of the system that the facade does not provide access to. Obviously, the more that a client directly relies on the system classes, the tighter the coupling and harder it is to modify the system without affecting its clients.

3.2.5 Proxy

> *"Provide a surrogate or placeholder [sic] for another object to control access to it." — GoF*

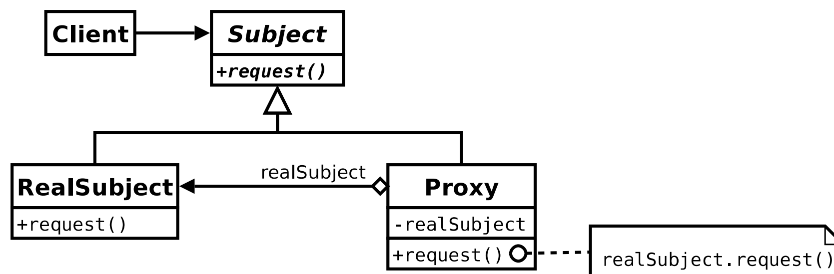

Figure 3.9: Proxy

According to Figure 3.9, the *Proxy* pattern is very similar to the *Decorator*, presented in Section 3.2.3. In fact, in certain cases, a proxy can also be considered to be attaching additional responsibilities to the object for which it stands proxy. The difference lies in the intent of the pattern, even though they are structurally very similar. The responsibilities associated with a proxy are typically more behind the scenes or house-keeping in nature than actually adding application specific behaviour to objects.

University of Pretoria etd – Peer, E S (2005)

There are four primary types of proxy. The first, a remote proxy, is a local representative that provides access to a complementary object in another address space. An example of this is a stub object, typically automatically generated, that implements calls to the same object on a remote machine via Remote Procedure Call (RPC). Secondly, virtual proxies are place holders, used to create and destroy their objects on demand, that are usually used to optimise memory or initial start up cost. Third, protection proxies prevent unauthorised client access to methods by implementing access control before delegating the method call to the real subject. Finally, smart references can be used to implement reference counting, locking or copy-on-write semantics.

3.3 Behavioural Patterns

Behavioural patterns model the flow of control and algorithmic interaction between objects. They specify how responsibility should be assigned to various classes to achieve communication between objects in the most flexible manner.

The *Interpreter* pattern, in Section 3.3.1, describes a method to represent a grammar as objects and use those objects to interpret the language. Section 3.3.2 discusses the well known *Iterator* pattern which specifies how objects in a collection should be traversed. Section 3.3.3 defines the *Observer* pattern which implements a flexible event model. The *Strategy* pattern, in Section 3.3.4, decouples a client from the algorithms it uses so that the algorithms can be varied independently. The *Template Method* pattern, discussed in Section 3.3.5, permits an algorithm to be defined in terms of abstract operations that are provided by subclasses. Finally, operations on collections or object structures can be encapsulated using the *Visitor* pattern, as discussed in Section 3.3.6.

3.3.1 Interpreter

> *"Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language."*
> *— GoF*

Figure 3.10 shows the abstract structure of the *Interpreter* pattern, used to interpret sentences in a language defined by a given grammar. The dynamic, or run time, structure of the abstract syntax tree reflects a sentence in the language. Terminals in the language

University of Pretoria etd – Peer, E S (2005)

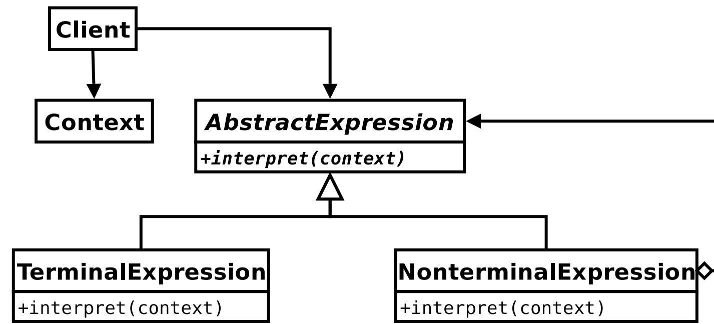

Figure 3.10: Interpreter

are represented by leaf nodes while non-terminals are represented by internal tree nodes. For arithmetic expressions, a separate non-terminal class would be defined for each of the arithmetic operators, while a single terminal expression class would suffice for representing constants. The value of the expression is then interpreted by simply calling the interpret method at the base of the tree, which is recursively propagated down the tree. Each operator is responsible for providing its own interpretation. For example, the interpret operation for an addition node would simply add the results of calling interpret for each of its children. The context is used to store global information, such as the current position in the sentence being interpreted.

The *Interpreter* pattern makes implementing and extending the grammar easy, since classes that represent the grammar have a one-to-one correspondence with its rules. Representing large grammars, however, requires many classes which becomes difficult to maintain. In addition, supporting a new interpretation of the grammar requires adding an operation to each of the expression classes which can become unwieldy if there are too many classes. Also, the *Interpreter* pattern does not address the process of parsing the language into its hierarchical representation, for which a traditional recursive descent or table-driver parser may be used.

3.3.2 Iterator

> *"Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." – GoF*

The *Iterator* pattern, demonstrated in Figure 3.11, provides a method to access elements

University of Pretoria etd – Peer, E S (2005)

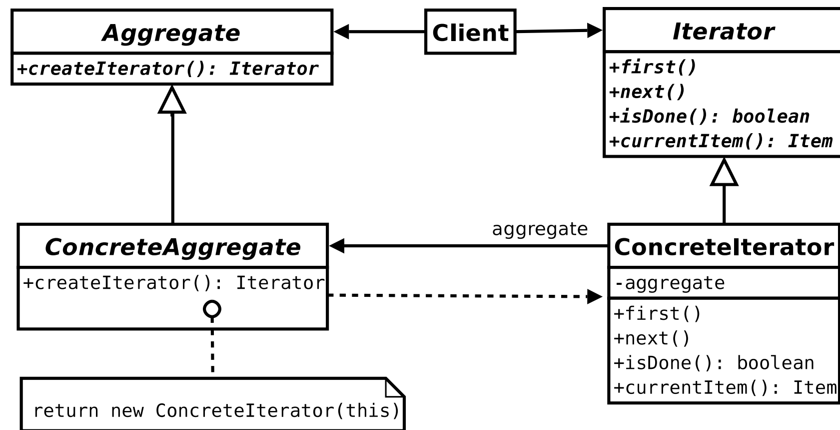

Figure 3.11: Iterator

of an aggregate object, or container, without exposing the client to the internal representation of the aggregate. The client obtains a reference to an iterator by calling an operation to create an iterator, a factory method, provided by the aggregate's interface. This operation returns an iterator that is specific to the concrete aggregate but which supports a well defined interface for performing the iteration. The iterator is responsible for keeping track of where it is in the traversal of the aggregate while providing operations for controlling the traversal. Using the iterator interface, the client can move the iterator to the start of the traversal, obtain the current element, move the iterator to the next element and determine whether there are any more elements left in the traversal. As long as all aggregates conform to the same interface, clients can access their elements in a uniform way.

The most important feature of the iterator is that it provides a standard mechanism for traversing aggregate structures. The interfaces of aggregates are kept clean, since new kinds of traversals can be implemented by simply replacing the iterator. Further, more than one traversal can be pending on the same aggregate because the iterator, and not the aggregate, is responsible for recording the state of the traversal.

University of Pretoria etd – Peer, E S (2005)

3.3.3 Observer

"Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically."
— GoF

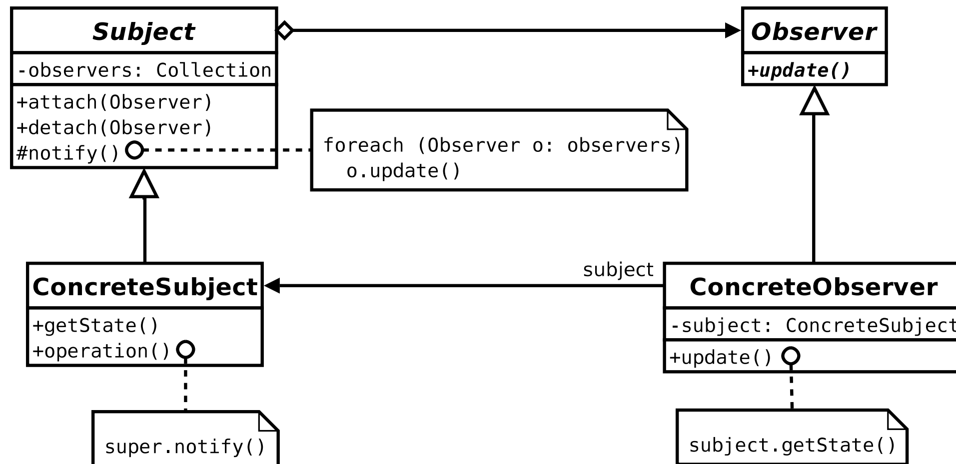

Figure 3.12: Observer

The *Observer* pattern, illustrated in Figure 3.12, models the dependency between a subject and its observers. Any number of observers may subscribe, by means of the attach operation, to be notified whenever the state of the subject changes. After detaching from a subject, an observer will no longer be notified of events. Upon being notified that the state of the subject has changed, an observer may query the state of the subject and take any appropriate actions.

The *Observer* promotes a very loose coupling between a subject and its observers. A subject knows nothing about its observers beyond that they conform to the observer interface. The observer interface presented here is fairly primitive, in that it does not provide any information about the change in state, other than the fact that some state change did occur on some subject. This means that an observer may have to expend considerable effort to determine exactly what state changed. A protocol that is more specific about any state changes would alleviate this problem. In addition, a single observer cannot differentiate between events from multiple subjects. Fortunately, the observer interface can be trivially extended to include a reference to the subject that

University of Pretoria etd – Peer, E S (2005)

originated the event, making many-to-many dependencies possible. Finally, observers have no knowledge about other observers attached to the same subject. This means they are blind to the cost of causing changes to the subject, which may cascade into more changes by other observers.

3.3.4 Strategy

> *"Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."* — *GoF*

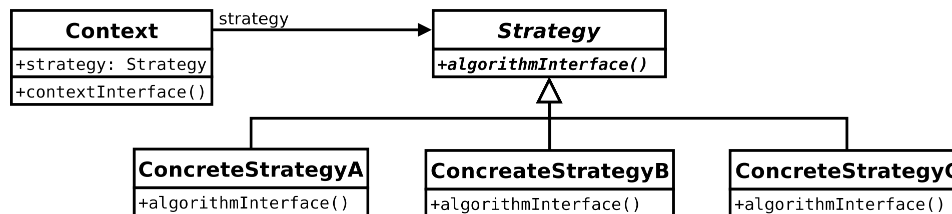

Figure 3.13: Strategy

Figure 3.13 shows the structure of the *Strategy* pattern. At first glance, it simply looks like a polymorphic class that implements multiple behaviours. The importance of the pattern, however, lies in the fact that it is the strategy interface which is polymorphic and not the context class itself. The context, which plays the role of the client, delegates the responsibility for a part of its implementation to an external strategy instance. Subclassing the context directly to provide the different behaviours would result in a less flexible design. By encapsulating the behaviour into a strategy, the context is simplified and different behaviours can be switched dynamically at run time. Also, the context can depend on multiple strategies, for different parts of its operation, simultaneously, which would be impossible to support by directly subclassing the context. For example, a client may rely on one hierarchy of strategies for one part of its implementation while maintaining an additional reference to another hierarchy of strategies for another. Subclassing the context directly would require a new subclass for each combination of the different strategies that can be independently interchanged.

University of Pretoria etd – Peer, E S (2005)

Another benefit of factoring the strategies into a separate hierarchy is that common functionality amongst a family of algorithms can be shared at the root of the strategy hierarchy without cluttering the context. Conditional statements in a client are prime candidates for factoring into a strategy, each branch is simply implemented as an additional concrete strategy, improving flexibility at the cost of increasing the number of classes in the system. The algorithm interface must provide access to the context data needed by any of the concrete strategies, which may create additional overheads for strategies requiring less context data. One possibility is to pass the context itself to the strategy and allow the strategy to query it directly.

3.3.5 Template Method

> *"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."* — *GoF*

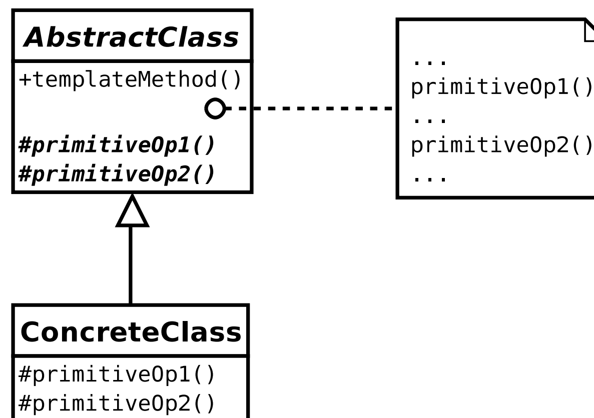

Figure 3.14: Template Method

The *Template Method* pattern, depicted in Figure 3.14, specifies the invariant parts of an algorithm in terms of primitive operations that may be overridden by subclasses. Primitive operations are usually abstract methods, however, they may also be empty methods or have default behaviours creating optional hooks that clients may choose to customise through subclassing. If any of the primitive operations are abstract then the template method is said to implement an abstract algorithm.

University of Pretoria etd – Peer, E S (2005)

The template method, particularly if it cannot be overridden, fixes a specific set of operations and their ordering for subclasses, promoting code reuse. Often, a subclass needs to perform some additional processing before or after a method in its parent class is called. A template method with an appropriate hook facilitates this kind of behaviour with the added benefit that the subclass cannot forget to call the original method which it would otherwise have overridden directly. Unfortunately, this approach can only be implemented one level deep without creating new names for the hook at each level of inheritance. Obviously, the template method doesn't restrict the placement of hooks to only the beginning and end of methods, giving a subclass far more flexibility in how it reuses the code in a parent class.

3.3.6 Visitor

"Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates." — *GoF*

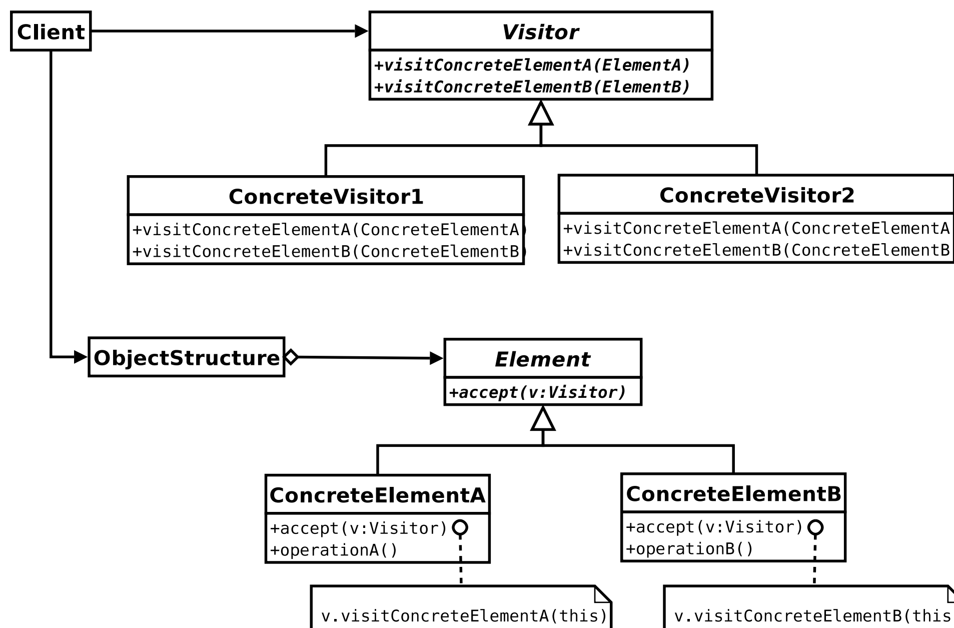

Figure 3.15: Visitor

Figure 3.15 illustrates the *Visitor* design pattern. The object structure can be any aggre-

University of Pretoria etd – Peer, E S (2005)

gate but is typically a tree structure such as an *Interpreter* hierarchy, as in Section 3.3.1, or a *Composite*, as in Section 3.2.2. A visitor encapsulates an operation which must be performed on each element of the object structure, while the accept method is responsible for traversing the object structure and calling the appropriate method for the type of element being visited. This calling strategy is known as double dispatch, since the method called to perform the operation is determined by both the type of the element in the object structure and the type of visitor.

Instead of spreading different parts of the same operation over multiple classes in a object structure, visitors enable related parts of an operation on multiple elements to be grouped into the same class. This clean encapsulation of an operation into a single class makes adding new operations easier, however, adding a new element type to the object structure requires changing all existing visitors to support it. Many of the special purpose methods in an *Interpreter* or *Composite* structure can be replaced with a single accept method for visitors that encapsulate those operations externally. Visitors also have the advantage of being able to accumulate state which may be difficult to distribute over multiple elements in an object structure. Unfortunately, because a visitor is external to the object structure, it may be necessary to provide a wider interface on the elements than would have otherwise been needed if the operations where supported internally within the structure. Thus, encapsulation for the elements may be adversely reduced so that visitors can perform their job.

3.4 Discussion

Design patterns are not an exact science. Patterns may be adapted and customised in the context in which they are being applied. Remember, design patterns are, for the most part, merely a way to encapsulate expert knowledge in an easy to digest form. They should be considered as guidelines for a good design rather than strict rules, since every situation is unique with its own set of constraints. Developers should still be free to be creative while building upon the knowledge gained from a study of patterns.

Patterns are also inter-related with certain patterns lending themselves to useful combinations. A few of these combinations have been hinted at in this chapter. Section 3.1.3 suggests that the *Prototype* can be used in conjunction with the *Abstract Factory* to alleviate the problem of parallel class hierarchies. The *Visitor* pattern, as discussed in

University of Pretoria etd – Peer, E S (2005)

Section 3.3.6, lends itself particularly well to a combination with the *Interpreter* or *Composite* patterns. Further, the *Abstract Factory* and *Facade* are often implemented as a *Singleton* when their implementations can be shared amongst multiple clients.

Finally, it should be noted that this chapter is not an exhaustive literary study of design patterns. There are more patterns presented in the GoF catalogue as well as many more ways that patterns are related to one another. Further, there are other catalogues that cover even more designs patterns, some of them specific to particular application domains. The content in this chapter is merely a terse summary of only those patterns that have been used in the implementation backing this work. Chapters 6 and 7 will refer back to the patterns presented in this chapter as appropriate.