CHAPTER 5

REQUIREMENTS ANALYSIS FROM THE SLOOP PERSPECTIVE

5.1 Introduction

When applying the SLOOP software development method, the requirements analysis phase has two important deliverables: the class diagram and a set of correctness properties describing the expected behaviour of the system, as well as a set of correctness properties for each individual class comprising the system. The class diagram is constructed by identifying classes and the relationships between them. The correctness properties are derived from the informal problem statement.

The purpose of the specification of the correctness properties is manifold:

- It encourages the designer to focus on correctness issues from the outset, i.e. this forms part of the **"constructive approach"** towards software development. As will be shown later on in this chapter, the designer is inclined to consider not only what the required behaviour should be, but also what it should not be. **Error conditions** are therefore identified at this early stage of system development.
- The identification of correctness properties encourages the software designer to perform a **careful analysis** of the informal problem statement. The checklist that is presented in Section 5.2.4 guides the designer to view the problem statement from many different perspectives and this exercise usually **reveals deficiencies** in the problem statement. These could be ambiguities, inconsistencies or omissions. The case study that is used in this chapter provides several examples of this aspect of the SLOOP method.
- Since the problem statement has to be studied from so many different perspectives, it aids the software designer in obtaining a **thorough understanding** of the requirements. A better understanding during the initial phases of software development is likely to result in fewer problems during the later phases.
- The specification of the behaviour of the system and its constituent classes via correctness properties aids the software designer in **identifying suitable frameworks and classes** in the repository of reusable artifacts during the design phase. The correctness properties of the system under development are compared with those of the reusable artifacts. It is not necessary to study the code of each reusable class in detail in order to understand its behaviour. It should suffice to study its correctness properties only.

In Chapter 1 it was stated that **scalability** needs to be addressed when devising a software development method. The case study that is used in this and other chapters was chosen specifically in order to illustrate this aspect of the SLOOP method. Especially in Section 5.4, it will be evident that the task of specifying correctness properties is not insignificant. Considerable time and effort is required in order to come up with meaningful properties. However, such an investment has the advantages as pointed out in the list above.

Furthermore, it will be noticed that the correctness properties listed in Section 5.4 are only specified **informally**. This is to avoid wasting time and effort translating informal properties into formal ones if this has already been done before, i.e. in the case where a similar system or parts of it can be found in a repository of reusable artifacts. Since the artifacts contained in a repository will have been implemented already, all aspects of their behaviour will have been specified in detail, both informally and formally.

The idea is therefore to specify enough correctness properties to capture all the requirements of the system under development. One of the aims of this step is to use the resulting specification to try and find **reusable** artifacts that might match these requirements in a repository. If inspection of the behaviour of those artifacts does not reveal any **conflicts** with the requirements of the system under development, it means that a great deal of **effort is saved**. This is because the formal specification of the behaviour of the matching artifacts can be reused in the new system. In addition, the refinements captured in the specification of these matching artifacts can also be reused if applicable. If a class of the system under development cannot be matched, its behaviour is specified formally during the design phase.

Understandability is another issue which receives attention in this chapter. A stated goal is to make the SLOOP method accessible to software engineers that are not necessarily trained in formal methods. This is one of the reasons for developing the **checklist** of correctness properties as presented in Section 5.2.4. It helps the software designer not to overlook some aspects of the behaviour of the system under development and is therefore an aid in avoiding **underspecification**.

Experience with the SLOOP method has shown that one of the difficult aspects of the specification of the properties at the analysis level is to ensure that the properties are formulated without implementation bias, i.e. the problem is to avoid **overspecification**. This topic is also covered in this chapter.

In Chapter 1 it was stated that the SLOOP method is a **"lightweight"** formal method. Thus, although no formal proofs are produced, the correctness properties have to be specified in a rigorous fashion. This is to ensure that these properties are **unambiguous** and **consistent**. Correctness arguments would not have much value if one could not at least rely on the unambiguity and consistency of the properties that are being reasoned about. During the requirements analysis phase the first steps towards achieving this are taken. The formalisation of the properties is performed during the design phase, when the final structure of the system has been determined (i.e. the classes that make up the system under development have been finalised after design level refinements and the incorporation of reusable frameworks, design patterns and/or classes).

It is not expected that the first version of the correctness properties will be the final one. The SLOOP method allows for multiple iterations of the steps of the requirements analysis phase and even iterations between the design and analysis phases. However, due to the fact that the designer is encouraged to analyse the problem statement very thoroughly during the analysis phase, the need to return to the analysis phase during the design phase tends to be reduced. The properties in the example shown in this chapter did **not** require iteration between the specification of analysis level properties and the development of SLOOP statements (a design phase activity).

During the analysis phase **multiple iterations** are usually caused by **deficiencies** in the initial informal problem statement and also as a result of the fact that the software designer gains **better insight** into the problem statement as he/she works through the checklist of correctness properties. Only the final iteration is shown in the example that is worked out in this chapter. However, wherever applicable the reasons for multiple iterations are pointed out.

The remainder of this chapter is organised as follows: First of all the correctness property definitions that were specified in terms of temporal logic in Chapter 2 are rewritten in terms of the SLOOP logical relations. This culminates in a checklist of useful correctness properties.

The definitions in this chapter are required in order to assist the software designer in formulating the correctness properties during the various development phases. Although the correctness properties are only formalised during the design phase, it is necessary to have a good understanding of what they are about when specifying them informally during the requirements analysis phase, hence the inclusion of their formal definitions in this chapter.

The latter part of this chapter demonstrates how the SLOOP method is applied during the requirements analysis phase. First of all the informal problem statement for the system that is used as running example in the remainder of this chapter as well as in the ensuing chapters, is presented. It is shown that the construction of the **class diagram** is an essential part of the requirements analysis phase, but any existing technique such as the one described in [RBPEL91] suffices to identify the classes and their relationships. The distinguishing aspect of the SLOOP method is its approach towards describing the **behaviour** of the system under development, which is the focus of the discussions in this chapter. In Section 5.4, the following issues are highlighted:

- It demonstrates how the different correctness property types are **interpreted** in terms of an informal problem statement.
- It shows how the specification of correctness properties can **reveal deficiencies** in an informal problem statement.
- It illustrates how the **"constructive approach"** results in a specification that does not only contain properties describing the desired behaviour of the system, but also properties describing the conditions that need to be satisfied in order to **prevent undesired behaviour**.
- It addresses the issue of **overspecification**.
- It discusses the need for **consolidation** after one has worked through the checklist of correctness properties, i.e. it highlights the need to check for and remove **inconsistencies** and **redundancies**.
- Finally, it is shown how the classes that make up the system under development have to **preserve** the correctness properties of the latter.

5.2 Useful correctness properties

In Chapter 2 it was stated that the correctness properties for concurrent systems are generally categorised as either safety or liveness properties. Manna and Pnueli [MaPn81a] define a third category, viz. precedence properties. A number of useful properties within these categories were then given, which serve as a basis for the **checklist** developed in this chapter.

In Chapter 2 the definitions of these properties were presented in terms of temporal logic. Location counters featured prominently in these definitions. The properties are now **rewritten** in terms of the logical relations used in the SLOOP method. Definitions of the UNITY logical relations **unless**, **ensures**, **leads-to**, **stable**, **invariant**, **detects**, **until** and **precedes** were given in Chapter 2, Section 2.5.5. Their SLOOP counterparts were described in Chapter 4, Section 4.3.4.4. The SLOOP **results-in** logical relation was also defined in the latter. All the SLOOP logical relations except the **results-in** logical relation are based on the SLOOP computational model.

Note that most of the definitions of the SLOOP logical relations refer only to the parallel statements of a SLOOP program. This is because the SLOOP parallel statements are the atomic units that are executed infinitely often, as required by the SLOOP computational model. (In the case of the **unless** logical relation used in the **class** *properties-section*, the definition also refers to

sequential methods, since the **safety** properties of a **class** must also be preserved by the sequential methods. However, the definition does not refer to individual sequential statements, since a sequential method is always executed as an atomic unit, usually as part of the execution of a parallel statement.)

As pointed out in Chapter 4, Section 4.3.4.3, the correctness of a sequential method is described via a total correctness property. The total correctness property of a sequential method is defined in terms of the **results-in** logical relation, which is based on the conventional model of control flow.

The following symbols are used in the definitions of the correctness properties:

$\varphi(\bar{x})$	The precondition that restricts the set of inputs $\bar{x}$ for which a program is supposed to be correct.
$\psi(\bar{x}, \bar{y})$	The statement of correctness, i.e. the relation that should hold between the input values $\bar{x}$ and the output values $\bar{y}$.
FP	Fixed point of the program, i.e. the execution of any parallel statement of the program does not change the state of the program.

In the case of UNITY, a fixed point of a program *G* is a predicate FP which is defined as:
$FP \equiv <\forall$ statements *s*: *s* in $G \wedge s$ is $X := E :: X = E>$
where *G* is a program, *X* is a variable and *E* is an expression [ChMi88]. Thus, if a fixed point exists, it is a program state such that the execution of any statement of program G leaves the state unchanged.

The above definition is not suitable for SLOOP programs, even if the phrase '∀ statements *s*' is replaced by '∀ **parallel** statements *s*' for the SLOOP case. This is because SLOOP statements are not necessarily of the form $X := E$. A SLOOP statement may also be made up of one or more message expressions only.

The SLOOP definition of a fixed point of a program *G* is therefore as follows:

$FP \equiv <\forall$ classes *C* **where** *C* in *G* ::
    <∀ parallel methods *PM* **where** *PM* in *C* ::
        <∀ statements *s* **where** *s* in *PM* :: $\alpha = \beta$
        >
    >
>

where α represents the program state prior to the execution of statement *s* and β represents the program state after the execution of statement *s*.

Thus, a fixed point of a SLOOP program leaves the program state unchanged after the execution of any parallel statement of any class in the SLOOP program. Note that a fixed point may or may not exist in a program.

One of the goals of the SLOOP method is follow a **unified** approach towards software development. Thus, the aim is to enable the software designer to use a single mechanism to specify any type of system, i.e. sequential, concurrent, terminating, non-terminating, etc. SLOOP programs are always non-terminating, since the parallel statements execute infinitely often. Terminating programs are viewed as a special case of a non-terminating program. More precisely, if a **stand-alone** SLOOP program G has reached a fixed point, then it does not matter

whether the execution continues or is terminated[1]. Correctness properties that are only meaningful for terminating **programs**, i.e. the partial and total correctness properties, are therefore formulated in terms of fixed points. (As noted above, the total correctness properties for sequential **methods** are defined in terms of the **results-in** logical relation.)

5.2.1 Safety properties

5.2.1.1 Partial correctness

Partial correctness is represented by the following relation:
$\varphi(\bar{x}) \wedge \neg$FP **unless** FP $\wedge\ \psi(\bar{x},\bar{y})$.

Thus, if a program reaches a fixed point, then the results of the program will be correct. The partial correctness property is only meaningful for a terminating program.

Since sequential methods are based on the conventional model of control flow, the partial correctness definition presented in Chapter 2, Section 2.4.4.1, can be used to describe the partial correctness of a sequential method:
$\varphi(\bar{x}) \supset\ \Box$ (at $\bar{l}_e \supset \psi(\bar{x},\bar{y})$)
where $\bar{l}_e$ is the vector of the exit locations of the sequential method. An exit location is one where the statement is preceded by the caret (^) symbol, or otherwise it is the last statement of the sequential method.

Note however that a SLOOP sequential method always returns a value, which is denoted by the special SLOOP variable `methodReturnValue` in the method correctness properties. As a result, it is not necessary to refer to explicit exit locations in the partial correctness properties of sequential methods. It suffices to refer to the required value of `methodReturnValue` in order to indicate that an exit location has been reached. The following definition of partial correctness is therefore used for SLOOP sequential methods:
$\varphi(\bar{x}) \supset\ \Box$ ((methodReturnValue = ρ) $\supset \psi(\bar{x},\bar{y})$)
where ρ represents the value returned by the method.

Some examples of the usage of the `methodReturnValue` variable are given in Section 5.2.2.1.

5.2.1.2 Clean behaviour

Clean behaviour comprises the conjunction of a number of cleanness conditions, which, if they are all satisfied by all statements of the program, will guarantee that no exception conditions will occur. Thus, this property specifies the conditions that should be satisfied in order to ensure that abnormal conditions will never occur. A cleanness condition α_{exc} is defined as

$\alpha_{exc} \equiv$ <$\forall$ statements s **where** s in G $\wedge$ s contains $MessageExpression_{exc}$ ::
$Condition_{exc}$
>

where *exc* is the exception condition, $MessageExpression_{exc}$ is the message expression that contains the construct that could result in the exception condition and $Condition_{exc}$ is the condition that needs to hold in order to prevent the exception condition from occurring.

[1] In the case where the SLOOP program G shares variables with other programs running concurrently with program G, then if program G has reached a fixed point, the state of program G can be changed only by statements outside program G.

Clean behaviour is therefore represented by

invariant $\bigwedge_{exc} \alpha_{exc}$

The conjunction is taken over all possible categories of exception conditions *exc* in the program.

For example:

$\alpha_1 \equiv$ <$\forall$ statements *s* **where** *s* in $G \wedge s$ contains *exp1*/*exp2* :: $exp2 \neq 0$
>

where *exp1* and *exp2* represent two different expressions.

"The exception condition is division by zero."

$\alpha_2 \equiv$ <$\forall$ statements *s* **where** *s* in $G \wedge$ s contains *receiver* at: *index* ::
$1 \leq index \wedge index \leq$ (*receiver* capacity)
> "The exception condition is an out of bounds subscript."

$\alpha_3 \equiv$ <$\forall$ statements *s* **where** *s* in $G \wedge$ s contains *receiver message* ::
receiver notNil $\wedge$ *receiver* respondsTo: (*message* selector)
> "The exception condition is a message sent to a non-existent receiver or one that does not support the message selector."

The above definition applies to the design phase of a system. However, clean behaviour at the analysis level can be viewed as the specification of those properties that will guarantee that certain abnormal conditions will never occur. This is explained further in Section 5.4.1.2, where examples of such properties are also given.

5.2.1.3 Global and local invariants

A global invariant is defined as:

$(\varphi(\bar{x}) \Rightarrow p) \wedge$ **stable** *p*

which can be written as

invariant *p*.

A global invariant can be defined for a class, in which case the invariant must hold before and after the execution of each parallel statement of the parallel methods of the class, as well as before and after the execution of each sequential method of the class. A global invariant can also be defined for a parallel method, in which case the invariant must hold before and after the execution of each parallel statement of that parallel method. A global invariant only becomes effective after instance creation has taken place, i.e. it has to hold after the execution of the last statement of an instance creation method, but it does not have to hold before that.

The concept of a local invariant is not used in a SLOOP program, since universal quantification is used in the definition of SLOOP safety properties. Invariants are not associated with specific locations in a SLOOP program.

5.2.1.4 Mutual exclusion

As described in Chapter 4, mutual exclusion is achieved in the SLOOP method by encapsulating all the statements that comprise a critical section into a single SLOOP parallel statement, since each SLOOP parallel statement is executed atomically. This is possible because of the expressive power of SLOOP parallel statements, as discussed in Chapter 4. For example, if the requirement is to remove the first two consecutive members from an ordered collection, then the steps to perform this action represent a critical section. It has to be ensured that no other object can remove or add members to the ordered collection between the first and second *removeFirst* messages.

As was shown in Chapter 4, Section 4.2.3, this is achieved as follows: the *if* clause of the SLOOP parallel statement contains the test whether there are at least two elements in the collection. If the condition evaluates to true, the corresponding *component-part* of that same statement then invokes a method which sends the *removeFirst* message to the collection twice in succession. Thus, all the actions are contained in a single parallel statement.

When the SLOOP software development method is used, **mutual exclusion** properties are therefore **not specified**. Instead, **safe liveness** properties are used to indicate which actions should be performed as a single atomic unit. Safe liveness properties are discussed in Section 5.2.3.1. At this stage it suffices to say that a safe liveness property can be defined as *p* **ensures** *q*, which means that if predicate *p* holds, then predicate *q* is achieved via a **single** parallel statement.

5.2.1.5 Deadlock freedom

The conditions for deadlock and how they apply to the SLOOP environment were discussed at length in Section 4.3.6.5 of the previous chapter. In summary, it is not considered necessary to define deadlock freedom correctness properties at the design level. This is because there is no concept of processes at the design level. There is therefore no concept of a process which can be blocked while waiting for a response from another process. At the design level one only needs to think in terms of the parallel statements of the SLOOP program. The semantics of the SLOOP parallel statements are defined such that two parallel statements that share objects may not execute simultaneously.

As described in Section 4.3.6.5, a SLOOP program could possibly contain circular conditional parallel statements. However, since such circular conditions could also be discovered via the correctness arguments of the progress properties of the program (as demonstrated in Section 4.3.6.5), the SLOOP method advocates the specification of the appropriate progress properties as a more pragmatic way to handle this problem.

For the reasons given above it is therefore not considered necessary to define deadlock freedom in terms of SLOOP parallel statements.

At the implementation level, the possibility of deadlock does exist. The deadlock prevention strategy used at this level was discussed in Section 4.3.6.5 of the previous chapter and more detail is provided in Chapter 8.

5.2.1.6 Generalised deadlock freedom

Since a SLOOP program does not contain waiting locations which contain looping instructions, this property does not apply to SLOOP programs.

5.2.1.7 Unless property

An **unless** property is defined as:
p **unless** *q*

Thus, if predicate *p* holds, then after the execution of parallel statement *s* (which could include the execution of sequential method *SM*), predicate *p* either continues to hold or predicate *q* holds.

5.2.2 Liveness properties

5.2.2.1 Total correctness

The total correctness property is particularly relevant for the specification of a terminating program using the SLOOP method. When referring to the parallel statements of the program, it is represented by the following statement:

$\varphi(\bar{x})$ **leads-to** FP $\wedge$ $\psi(\bar{x},\bar{y})$

Thus, a fixed point will be reached and when it is reached the results will be correct.

For sequential methods, total correctness is defined as follows:

$\varphi(\bar{x})$ **results-in** (methodReturnValue = ρ) $\wedge$ $\psi(\bar{x},\bar{y})$,

where ρ represents the value returned by the method.

The `methodReturnValue` variable was discussed in Section 5.2.1.1. Examples of its usage are now presented. In the first example it is used to represent the value returned by the method. The method below determines whether there is an idle service provider to which a service request can be assigned. The method returns true if it finds a service provider that can accept a new service request of the specified service category and false otherwise. (The candidate service providers are elements of the `spSubset` collection.) The total correctness property is as follows:

```
true results-in methodReturnValue =
      (spSubset detect:
      [:each | each canAcceptNextSR: serviceQCategory]
      ifNone: [nil] ) notNil
```

Although all sequential methods return values, the main purpose of some methods may be to modify some variables and there is no requirement to return a specific value. In that case the receiver is returned as a default return value. This is the second scenario in which the `methodReturnValue` variable can be used. The following total correctness property provides an example:

```
<∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
categoryIndex = x ∧
aServiceRequest notNil ∧
self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
      results-in
            methodReturnValue = self ∧
            serviceRequest = aServiceRequest ∧
            newEventRequired ∧
            categoryIndex = (x + 1) \\ nrOfCategoriesServed
      >
```

It is the total correctness property of the `processServiceRequest:` method of the ServiceProviderSimulator class. This method is executed when a new service request is assigned to a service provider simulator. Its purpose is to update the `serviceRequest`, `newEventRequired` and `categoryIndex` instance variables as indicated above. The return value is not used by the sender. The methods associated with the above properties are discussed further in the chapters to follow and full SLOOP specifications of these methods appear in Appendix B, Sections B.10 and B.13 respectively.

5.2.2.2 Intermittent assertions

An intermittent assertion specifies that if property *p* holds at some stage, then at some later stage property *q* will hold. Thus:

p **leads-to** *q*

The important issue here is that p is not required to hold until q holds. If that is a requirement, the **ensures** or **until** relations have to be used. They are discussed in the precedence property category.

5.2.2.3 Accessibility

This is represented by:
R_1 **until** C_1
where R_1 represents the **condition** that holds when concurrent object 1 wishes to enter its critical section and where C_1 represents the **condition** which holds when concurrent object 1 is inside its critical section. Thus, concurrent object 1 will eventually enter the critical section associated with C_1 if condition R_1 becomes true and R_1 continues to hold until object 1 enters the critical section.

Accessibility and mutual exclusion properties are complementary properties. The former specifies the liveness aspect of critical section handling, whereas the latter specifies the safety aspect. As explained in Section 5.2.1.4, mutual exclusion properties are not specified in SLOOP programs. Instead, safe liveness properties are used to indicate which actions should be executed atomically.

A safe liveness property can be defined as p **ensures** q[2]. It specifies that a **single** parallel statement establishes q, which takes care of the safety aspect of the critical section handling, since each parallel statement executes atomically. The **ensures** logical relation also guarantees progress, which takes care of the liveness aspect of the critical section handling. As a result it is not necessary to specify accessibility properties for SLOOP programs, provided safe liveness properties are specified to indicate which actions should be executed atomically.

5.2.2.4 Liveness (absence of individual starvation)

All parallel statements are executed infinitely often, therefore all concurrent objects will always progress from one statement to the next. However, since a statement may execute conditionally, it could happen that the condition is always false whenever the statement is scheduled. In that case it amounts to starvation, since the state changes specified by the statement can never take place.

A design heuristic which addresses this problem is to design the conditions in such a way that once they evaluate to true, they remain true until the associated statement has been executed. However, this is not always possible, therefore this is not a hard and fast rule, but merely a heuristic. In SLOOP programs the intermittent assertions (described in Section 5.2.2.2) and the safe liveness properties (described in Section 5.2.3.1) should ensure that each concurrent object makes the desired progress.

5.2.2.5 Responsiveness

This property states that if the predicate r_i representing some request i is true at some point, then eventually predicate g_i will become true, representing the granting of request i. Thus,
r_i **leads-to** g_i.

---

[2] If it is not important that a **single** parallel statement should establish q, then safe liveness is defined as p **until** q, as described in Section 5.2.3.1.

5.2.3 Precedence properties

5.2.3.1 Safe liveness

If property p has to hold at least until q holds and a single parallel statement establishes q, the **ensures** relation is used, i.e.
p **ensures** q.

If property p has to hold at least until q holds, but q is not established by a single parallel statement, the property is formulated as follows:
p **until** q,
where the **until** logical relation is defined as follows:
p **until** $q \equiv (p$ **unless** $q) \wedge (p$ **leads-to** $q)$.

5.2.3.2 Absence of unsolicited response

Absence of unsolicited response is formulated as follows:
p **precedes** q,
where the precedes logical relation is defined as follows:
p **precedes** $q \equiv \neg((\neg p)$ **until** $q)$.

Thus, if q ever becomes true, it will not become true until p becomes true first [MaPn81a].

5.2.3.3 Fair responsiveness

Fair responsiveness means that if the request represented by r_1 is received before the request represented by r_2, then the first request is granted before the second one, represented by g_1 and g_2 respectively. In addition, the individual responsiveness properties hold as well. Thus,
$((r_1$ **precedes** $r_2) \Rightarrow (g_1$ **precedes** $g_2))$
$\wedge$
$((r_1$ **leads-to** $g_1) \wedge (r_2$ **leads-to** $g_2))$.

5.2.4 SLOOP checklist of correctness properties

The list of correctness properties that are used in SLOOP specifications, is now presented, based on the discussions above. In a SLOOP specification each correctness property has a unique number associated with it. This number is used for cross-referencing purposes. The number has the following format: XYx-yy, where

X	the value A, D or I denotes that the property emanates from the analysis, design or implementation phase respectively,
Y	has the value S (safety), L (liveness) or P (precedence),
x	is the number of the property type within the safety, liveness or precedence category,
yy	distinguishes the property from others of the same type.

The X and yy generic symbols only receive values once an actual correctness property is specified for a system or class.

Safety properties:
XS1-yy: Partial correctness
XS2-yy: Clean behaviour
XS3-yy: Global invariants
XS4-yy: Unless property

Liveness properties:
XL1-yy: Total correctness
XL2-yy: Intermittent assertions
XL3-yy: Responsiveness

Precedence properties:
XP1-yy: Safe liveness
XP2-yy: Absence of unsolicited response
XP3-yy: Fair responsiveness

This checklist will be used in the example that is described in the remainder of this chapter.

5.3 Constructing the class diagram

In the previous chapter a number of aspects of the SLOOP method were illustrated via the call centre example. The requirements for a call centre system are now discussed in more detail. The call centre system serves as a case study of the application of the SLOOP method and is used in this and the remaining chapters to highlight various features of the method.

This specific example was chosen because it is non-trivial and it therefore demonstrates how the SLOOP method addresses the **scalability** problem. At a high level of abstraction the functionality is not restricted to a call centre; any system which has to switch service requests to service providers based on certain criteria can be based on the framework of high level classes presented here. This **reusability** aspect is discussed in more detail in Chapter 9.

For the sake of brevity, the example is restricted to the first level of abstraction.

5.3.1 Initial informal problem statement

The problem statement below refers to a specific type of call centre, viz. one with anonymous service users. A typical application of such a call centre is the toll-free customer service offered by many large companies. The identity of the caller is not required in order to **switch**[3] the service request to the appropriate service provider. For example, a potential customer might just want to enquire where the company branch in a certain area is located. Another customer might want to order an item, in which case information about the customer is required for payment purposes, but this information is obtained by the service provider, not the call centre, since the latter does not require it for switching purposes.

There are other types of call centres that use the Automatic Number Identification (ANI) service provided by the Public Switched Telephone Network (PSTN) to identify the caller, which influences the way in which the call centre switches the service request. For example, if the caller is identified as an important customer, the service request might be enqueued in a high priority queue. However, since the **purpose** of this case study is to illustrate various aspects of the SLOOP method and not to provide a comprehensive framework for call centre systems, the problem statement below refers to anonymous service users only. In the next chapter (Section 6.7) it is illustrated how the approach that is followed during the design phase allows for relatively simple extensions to cater for other types of call centres. It therefore demonstrates how the goals of **reusability** and **extensibility** are addressed.

[3] The term 'switch' is used to refer to the actions performed by the call centre when it analyses the service request in order to determine which service provider (or set of service providers) would be most suitable to service the request, as well as all the subsequent actions taken by the call centre that culminate in a connection between the service user and a service provider. The service request is an object created by the call centre and it contains all the information required by the call centre about a particular request for service.

The initial informal problem statement is as follows:

A call centre is a non-terminating system which ensures that dial-in users are serviced in a specified order by the available service providers. This is accomplished as follows:

The service user places a call to a central (usually toll-free) number. Physically, this number may represent several lines (if line hunting is provided by the Public Switched Telephone Network), or it may represent a single high speed line which can carry multiple connections. The number of calls that can be handled **simultaneously** is therefore bounded. If the service user receives a ringing tone, it implies that a connection or line is available, otherwise a busy tone is signalled. The call centre is not responsible for controlling the ringing and busy tones. That is performed by the Public Switched Telephone Network (PSTN).

The line or connection that is occupied by a service user remains so for the duration of the call, i.e. until the user hangs up, the connection is terminated due to a problem or the service provider has finished serving the user. The identity of the service user is irrelevant, therefore there is no upper limit to the number of service users, but they cannot all connect to the call centre simultaneously. Since the call centre does not store any information about the service users for switching purposes, there are no physical constraints regarding the maximum number of service users that can be supported.

The call centre processes the calls on a first-come, first-served basis, i.e. as they are received from the PBX. If the call centre supports more than one service request category, it receives information about the type of service required by the service user together with the call. This information is obtained from the service user via an Interactive Voice Response (IVR) before the call reaches the call centre, as shown in Figure 5-1. The call centre uses this information to categorise the service required by the service user. If at least one service provider is active which provides a service of the specified category, the service request is accepted, otherwise it is rejected and the connection is terminated.

The call centre ensures that each service user is serviced on a first-come, first-served basis within each service request category. This is illustrated graphically in Figure 5-1. It shows that there are a number of queues (one for each service request category). Each service request is entered into one of these queues and each queue is serviced on a first-come, first-served basis.

Once the service request has been allocated to a service queue, the service user is informed at regular intervals of his/her progress towards being served. If there is only one service request category, there is no need to obtain information from the service user in order to categorise the service required.

The order in which the various service request categories are served is based on some application-dependent criteria. For example, there may be high and low priority categories and the criterium might be to serve the high priority category before the low priority category as long as there are high priority service requests pending. Alternatively, the criteria might merely bias service towards the high priority category in some way.

There are one or more service providers. The number of service providers is bounded. Each service provider may service one or more service request categories. For example, one type of service provider (say the elementary type) might service queries only. Another type of service provider (say the advanced type) might be able to service both queries and other transactions. There are therefore one or more service provider categories. Each service request category should be serviced by at least one service provider category.

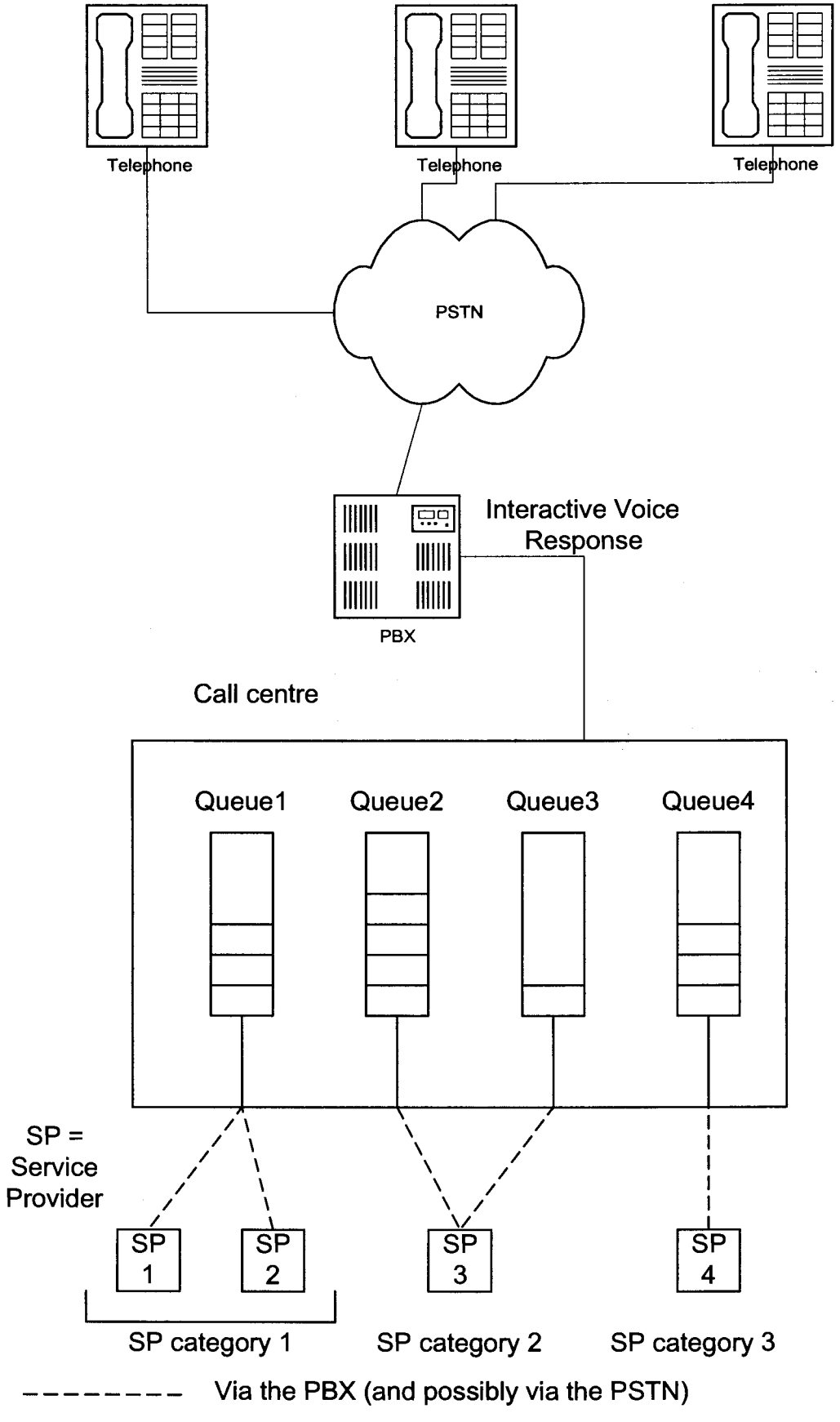

Figure 5-1. Architecture of a call centre and its environment at a high level of abstraction.

Figure 5-1 illustrates some of the allowed combinations as follows: Both Service Provider 1 (SP1) and SP2 service Queue 1. Queue 1 represents local directory queries. Queue 1 is serviced by service providers in the elementary category. Both SP1 and SP2 belong to the elementary service provider category. Queues 2 and 3 represent non-local national directory queries and international directory queries respectively. Both queues are serviced by service providers in the intermediate category. Only SP3 belongs to this category. Queue 4 represents other transactions and is serviced by service providers in the advanced category. SP4 belongs to that category.

The service request categories and the service provider categories referred to in this problem statement are devised in order to provide flexibility in the way in which service requests are allocated to service providers. These categories may therefore be independent of any other service request categorisation performed by the service providers once a service request is allocated to one of them.

Once a service request has been categorised and enqueued in a service queue, the call centre ensures that it is eventually assigned to a service provider, unless the service user hangs up beforehand. The nature of the service being provided is irrelevant to the call centre. The connections between the service providers and the call centre may be via the PSTN or they may be established internally via the local PBX as shown in Figure 5-1.

The initial product needs to be tested by simulating the actions of the service users, service providers and the communication provider.

System boundaries:
The service providers, service users and communication service form part of the environment of the call centre, i.e. the latter performs a queuing and switching function only. The service providers contain the software to process a service request. This software may vary from application to application and therefore does not form part of the call centre. The service providers may even be supplied by a third party. The software that needs to be designed is restricted to the part pertaining to the call centre and its interface with its environment. This is the part that remains unchanged between applications.

Assumptions:
At this level of abstraction a reliable communication medium is assumed between the service users and the call centre as well as between the call centre and the service providers.

The analysis phase is an **iterative** process as was indicated in Figure 4-9 in the previous chapter. While listing the properties of the system, it quite often happens that **deficiencies** in the problem statement are discovered. This is illustrated as the analysis for the above example is performed.

5.3.2 The class diagram

The first step towards creating a class diagram is the identification of the classes in the problem domain. The SLOOP method does not prescribe a specific technique towards accomplishing this, since the distinguishing aspect of the SLOOP method is its description of the **dynamic** behaviour of the system. Any of the usual techniques will suffice. For example, the nouns in the problem description can be extracted and used as a basis for identifying classes [RBPEL91]. This aspect of the analysis phase is not pursued further; only the results are given here.

A preliminary analysis of the problem statement yields the classes as shown in Table 5-1. It contains all the classes in the problem domain, i.e. those that form part of the system, (the call centre) as well as those in its environment. The simulation classes are included in addition to their real-world counterparts.

Class	Justification
ServiceUser	This class represents the service user. The service users are external to the call centre, i.e. form part of the environment of the call centre. There is no container class for the service users, which **models** the fact that there is **no upper limit** to the number of service users.
Connection	Although the problem statement refers to lines that are connected to the call centre and to calls that are placed, the crux of the matter is the fact that some or other mechanism is used to **connect** to the call centre. It is sufficient to model this at a higher level of abstraction via the Connection class. This ensures that the system is **not over-specified,** i.e. it leaves scope for any type of connection to the system. Once the connection is switched to a service provider, it represents the complete path from the service user to the service provider.
Connection Container	The aspect that needs to be modelled here is the fact that the number of **simultaneous** service user connections is **bounded**. ConnectionContainer, a container class with a maximum capacity, adequately describes this requirement. The ringing and busy tone objects do not yield new classes, because they refer to implementation detail of the communication service. At this level of **abstraction** it is only relevant to know whether a connection can be established or not, which is modelled by the states of the connections in the connection container.
Communication Provider	The CommunicationProvider class provides the communication service. It forms part of the environment of the call centre and could be the PSTN or in the case of service provider access, it could be private lines or the PSTN.
ServiceProvider	A service provider ultimately services the service request. The service providers form part of the environment of the call centre.
ServiceProvider Container	The ServiceProviderContainer class, which has a maximum capacity, **models** the physical **constraint** on the number of ServiceProvider instances that can be supported by the call centre. The very fact that service requests are queued implies the **boundedness** of the number of service providers. If there had been an unlimited number of service providers, each service request could have been assigned to a service provider as it was categorised. This is therefore an **analysis level** constraint and not a design level one.
ServiceCategory Allocator	The problem statement refers to the call centre when it describes the required behaviour of the system. Various aspects of the behaviour are modelled by various different classes. The ServiceCategoryAllocator class models that part of the call centre which allocates new service requests to the service queues of the appropriate service categories.
InputQueue	In this case there is no real-world object that needs to be modelled; instead the **abstract concept** of the **order** in which new service requests are processed needs to be reflected. This order is modelled by the properties of InputQueue, a container class which provides methods to manipulate its elements in a **First In First Out** manner. An alternative model which comes to mind is to use the ConnectionContainer class to represent this ordering. Thus, a Connection instance is added to the connection container in the order in which the connections are established. However, this also implies that the Connection instance has to be removed from the container as soon as the connection has been enqueued in a service queue, even though the

	connection is still in use. This means that the connection container no longer fulfils its other purpose, namely that of representing the boundedness of the number of users that may be connected to the system simultaneously. (The reason why the Connection instance has to be removed from the connection container once it has been enqueued in a service queue if there is no InputQueue class is to guarantee that the connection container class models the First In First Out ordering. Consider the following counter-example: If the Connection instance had remained in the connection container until the connection had been terminated, then the state of the Connection instance could have been used to indicate whether the associated service request had already been enqueued. However, since connections can be terminated in any order, new connections would have had to be entered into the first available slot in the connection container, in which case the elements of the connection container might no longer have been ordered in a First In First Out order. For example, slot 4 might have become available before slot 1.)
ServiceRequest	The ServiceRequest class represents all the information that the call centre requires from the service user. The ServiceRequest class is therefore an abstraction for identification information (in other call centre types), the type of service that is required and any other information required from the service user. The mechanism whereby this information is obtained is beyond the scope of the call centre (e.g. some of it may be supplied by the PSTN, other information may be provided via the IVR). A service request is associated with a connection.
ServiceCategory	This is an **aggregate** class. There is a ServiceCategory instance for each service request category. The number of service categories is also **bounded**, but that is a **design level constraint**. Nothing in the problem statement implies that there is an upper limit to the number of service categories. **At the analysis level there is therefore no ServiceCategoryContainer class.** The ServiceCategory has the following components: ServiceQueue, ServiceProviderSubset andServiceProviderCategories. The ServiceQueue instance contains service requests that are of the same category as that of the ServiceCategory instance. The ServiceProviderSubset instance contains references to the service providers that handle service requests of that particular category. The ServiceProviderCategories instance contains a set of service provider categories. These are the allowed categories of the service providers that may service the requests in the service queue belonging to this ServiceCategory instance. A ServiceCategory instance therefore has a service request category, an ordered collection of service requests, a non-empty collection of service providers and a non-empty collection of service provider categories associated with it. The service providers associated with a ServiceCategory instance do not all have to be of the same service provider category. The ServiceCategory instance monitors the status of the service providers referenced by the elements of the ServiceProviderContainer.

	The order in which the various service queues are serviced by the service providers is based on application-dependent criteria. Each ServiceCategory instance is responsible for determining whether the service requests in its service queue can be serviced and which service provider the service request should be allocated to.
ServiceQueue	The ServiceQueue class is a **component** of the ServiceCategory class. The service queue contains an ordered collection of service requests of the specified category. The elements of the collection are processed on a first-come first-served basis. Query and transaction are examples of service request categories.
ServiceProvider Subset	The ServiceProviderSubset class is a **component** of the ServiceCategory class. It contains a collection of references to all the service providers that handle service requests of the specified category.
ServiceProvider Categories	The ServiceProviderCategories class is a **component** of the ServiceCategory class. It contains a collection of all the service provider categories associated with the service request category.
TimerServices	When a service request is added to a service queue, a progress timer is started. When it expires, the service user is informed of the progress of the service request and the timer is restarted. Timers are also used by the simulation classes. The ServiceProviderSimulator uses timers to simulate the non-instantaneous nature of actions such as the servicing of a request. The CommsProviderSimulator uses timers to simulate the random periods that may elapse between receiving new calls. The TimerServices class handles all timer-related aspects.
CommsProvider Simulator	The system facilitates testing without actual service users, service providers and a communication provider by introducing classes to simulate their behaviour. The CommsProviderSimulator class **simulates** the actions of the communication provider.
ServiceProvider Simulator	The ServiceProviderSimulator class **simulates** the actions of a service provider.
ServiceProvider Simulator Container	The ServiceProviderSimulatorContainer class, which has a maximum capacity, models the physical constraint on the number of ServiceProviderSimulator instances that can be supported by the call centre.

Table 5-1. Classes yielded during the analysis phase.

Note: There is no need to create a ServiceUserSimulator class to simulate the behaviour of the service user. The reasons are as follows: Before the call is switched, the interaction with the service user occurs between the service user and the communication provider (e.g. via the IVR). After the call is switched, the interaction is between the service user and the service provider. There is therefore no direct interaction between the service user and the call centre at any stage.

Figure 5-2 shows the class diagram (static structure) of the problem domain. It does not contain the simulation classes. Figure 5-3 shows the class diagram where the simulation classes replace their real-world counterparts. The differences between the two diagrams are shown in bold.

The following is a very brief summary of the UML notation used in the diagrams. Classes are represented by rectangles. A solid line connecting two classes represents the association between the two classes. The small black solid triangle is the UML symbol which indicates in which direction to read an association name. The end where the association connects to a class is called an association role. An association role is adorned with a multiplicity string. The latter specifies the allowable cardinalities of a set. An unlimited upper bound is denoted

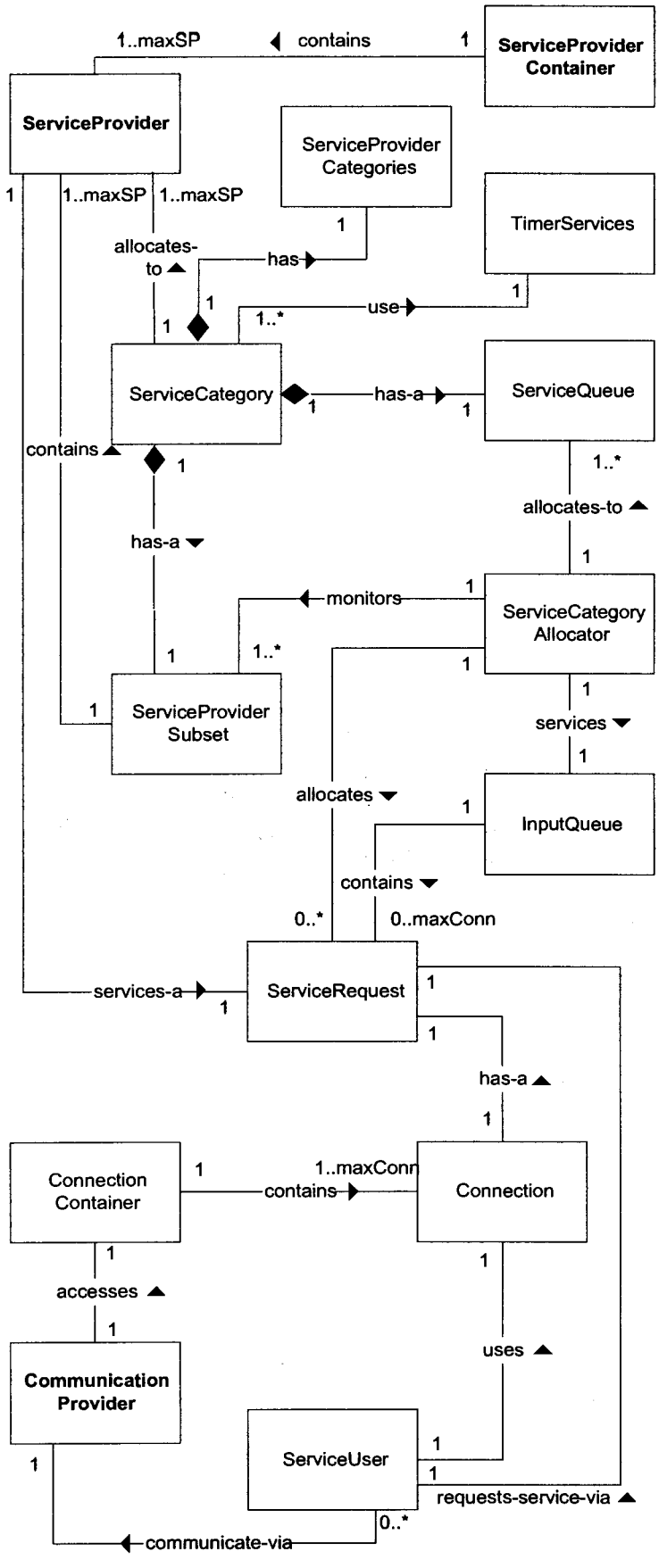

Figure 5-2. Class diagram of the call centre and its environment (without simulation classes).

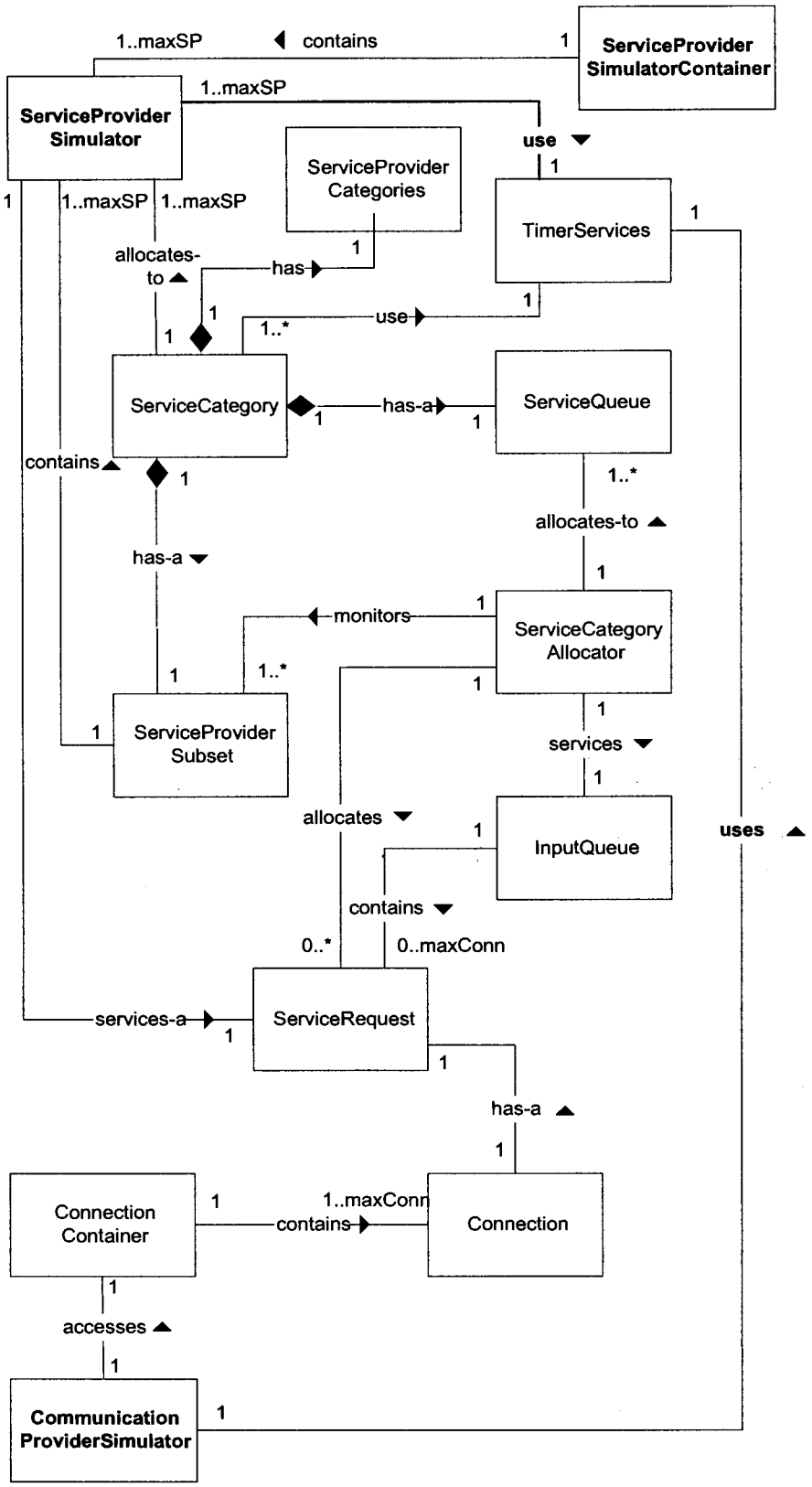

Figure 5-3. Class diagram of the call centre and its environment (with simulation classes).

by specifying an asterisk (*) for the upper bound value. If an association role is adorned with a diamond, it indicates aggregation. The diamond is attached to the aggregate class. If the diamond is filled, it indicates composition, the strong form of aggregation.

The classes in the class diagram represent **real-world objects** as well as **abstract concepts** that are evident from the analysis of the problem statement. The suitability of these classes will be re-inforced during the specification of the system properties. If necessary, the class diagram is modified if it is found that it is inadequate at that stage. This iterative process was reflected in Figure 4-9, which showed a graphical representation of the requirements analysis phase.

As stated above, the purpose of this chapter is to illustrate how the **behaviour** of the system is specified via correctness properties during the analysis phase. The class diagram is therefore not elaborated any further. For example, a data dictionary is not provided here and the attributes of the various classes are not described. Attributes that are used in the specification of the correctness properties below will be explained where necessary.

5.4 Specifying system behaviour informally

The next step is to list the desired properties of the system informally. These properties result from an inspection of the informal problem statement using a checklist of correctness properties such as the one given in Section 5.2.4. Initially the focus is on global system properties as opposed to individual class properties. These include, inter alia, global invariants as well as properties that describe the nature of the **interactions between** the objects. This step yields some of the methods of the classes defined earlier.

By listing the system properties it can be determined what is required of the constituent classes. Subsequently, the properties displayed by each individual class are defined. Note that the SLOOP syntax does not have a special construct for the specification of system properties. This is because the system itself can also be viewed as a class and the properties can therefore be included in the class properties section. This approach is exemplified in Chapter 6.

5.4.1 Safety properties

The four safety (invariance) properties listed in Section 5.2.4 are now considered in turn. The properties that are identified here emanate from the requirements analysis phase, therefore all the property identifiers start with the letter 'A', followed by the letter 'S' to indicate that they are safety properties. Four groups are considered in turn, viz. partial correctness (AS1-yy), clean behaviour (AS2-yy), global invariants (AS3-yy) and unless properties (AS4-yy).

5.4.1.1 AS1-yy (partial correctness)

The system is cyclic (does not terminate), therefore partial correctness does not apply.

5.4.1.2 AS2-yy (clean behaviour)

At the **design level**, the clean behaviour properties ensure that **no exceptions** will be raised as described in Section 5.2.1.2. At the **analysis level** clean behaviour properties specify the conditions that should always be satisfied in order to **prevent violations of the specified system limits from occurring**. For example, the informal problem statement specifies that the number of simultaneous connections supported by the system is bounded. The call centre system should therefore not accept a new connection if the maximum number of connections are already established. This requirement results in the specification of the following clean behaviour property:

AS2-01. The connection container has a maximum capacity of maxConn connections, where maxConn is a positive integer.

Property *AS2-01* confirms the choice of a ConnectionContainer class in the class diagram; it is used to model the **capacity restrictions** of the call centre. The number of elements in a connection container (its size) may never exceed its capacity. The communication provider has to ensure that this property holds when it allows a connection to be established, otherwise the connection has to be refused. The system has to support at least one connection in order to provide any service at all, therefore `maxConn` has to be a positive integer.

The phrase "where maxConn is a positive integer" naturally leads to the following observation: The informal problem statement seems to assume that at least one connection will be supported, but there is no explicit reference to the **minimum requirements** of the system. This **reveals a deficiency** in the informal problem statement. It also leads to another question: What other collection classes in the class diagram are affected by the specification of a minimum capacity? This also reveals that the informal problem statement does not address the issue of the behaviour of the call centre system in the case of **resource problems**.

As a result of the above analysis of the problem statement, consultation with the client is required. An improved informal problem statement is produced. The following requirements are added:

"The call centre has a required capacity of maxConn connections, where maxConn is a positive integer. Once a connection has been accepted by the call centre, the associated service request may only be rejected if none of the active service providers have the capability to service the request. The service request may not be rejected due to any other resource problems.

The call centre has to support maxCategories of service request categories. It has to support maxSP of service providers."

As a result of the above modification to the original problem statement, the class diagram is modified to include a ServiceCategoryContainer class. Thus, now there is an analysis level requirement to model the fact that a certain number of ServiceCategory instances have to be supported. Property *AS2-01* is modified as follows:

*AS2-01. The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer.*

The following additional clean behaviour properties are specified:

AS2-02. The minimum capacity of the input queue is equal to the capacity of the connection container.

The InputQueue class models the order in which the service requests are received. It is therefore essential to be able to represent a service request in the input queue if it is possible to represent its associated connection in the connection container. Note that the property specifies that the capacity has to be greater than or equal to that of the connection container (a **minimum** capacity is required). This ensures that the property is not overspecified. There is no requirement for the input queue to be larger than the minimum, but there is also no analysis level reason why it should be restricted to have exactly the same capacity as the connection container. This enables one to consider multiple design options, as will be seen in Chapter 6, Section 6.2.1.

AS2-03. The minimum capacity of each service queue is equal to the capacity of the connection container.

The above property specifies that there will be space for a service request in a service queue if its associated connection is represented in the connection container. Thus, even if the previous `maxConn-1` service requests are of the **same** category and have not been allocated to a service provider yet, the capacity of the service queue will be sufficient to hold another service request.

This property ensures that the processing of a connection and the placement of its associated service request into a service queue are **independent** of any other service requests in the system. The following scenario therefore **cannot** occur: Suppose there is a high priority and a low priority service request category. The maximum capacity of each queue is `maxConn/2`. The low priority queue contains `maxConn/2` service requests and the high priority queue contains none. Service request A, which is of the low priority, arrives and is entered into the input queue. Since the low priority service queue is full, service request A has to remain in the input queue until space becomes available in the low priority service queue. If, however, one of the `maxConn/2` previous service requests had been a high priority service request, this service request would have been enqueued immediately.

The above scenario also illustrates that a service request can be delayed by a service request from a different category if property *AS2-03* is **not** specified. If, for example, service request A in the above scenario is followed by a high priority service request B in the input queue, then the latter cannot be processed until space becomes available in the low priority service queue and service request A can be removed from the input queue.

If property *AS2-03* is satisfied, then the above scenarios are not possible. Note that this is **not** a **design level** correctness property. It **does not prescribe how** the service queues should be implemented; it merely specifies that each service queue should be allowed to reach the size of `maxConn` in order to ensure that the service requests in the input queue can be serviced in a First In First Out order while at the same time ensuring that one service request cannot delay another while they are both in the input queue. This is reflected via the following additional sentence in the informal problem statement:
"The call centre ensures that service requests of one category are not affected by inadequate resources allocated to service requests of another category."

In order to reason about the minimum capacities of the input queue and the service queues, properties *AS2-04* and *AS2-05* as given below are also required. These properties ensure that obsolete entries are not allowed in these queues. If these properties are not satisfied, then it cannot be guaranteed that the system will be able to process a service request associated with a connection that is represented in the connection container.

AS2-04. If a connection is terminated, it implies that its associated service request is not present in the input queue.

AS2-05. If a connection is terminated, it implies that no service queue contains a service request associated with the connection.

The property below ensures that no obsolete service requests are associated with a service provider.

AS2-06. If a connection is terminated, it implies that no service provider is servicing a service request associated with the connection.

The next two properties demonstrate how the need to formulate correctness properties results in the clarification of ambiguous statements in the informal problem statement. Consider the following two sentences that were added to the informal problem statement earlier in this section: **"The call centre has to support maxCategories of service request categories. It has to support maxSP of service providers."**

It is not clear from the above whether the word 'support' should be interpreted that the relevant container objects should merely **be able** to handle the specified maximum number of elements, or whether those elements should **always be present** after initialisation. It is important for the person formulating the correctness properties to know exactly how the word 'support' should be interpreted, since it determines whether the word 'size' or 'capacity' should be used in the correctness properties. (The **size** of a collection reflects the **number of elements** of a collection, whereas the **capacity** indicates the **largest** number of elements that the collection **may contain**.)

Again the informal problem statement needs to be updated to clarify this issue. The following sentence is therefore added:
"All supported service categories and service providers must exist after initialisation of the call centre."

Properties *AS2-07* and *AS2-08* can now be formulated to reflect this interpretation:

AS2-07. *The size of the service category container is equal to maxCategories, where maxCategories is a positive integer.*

AS2-08. *The size of the service provider container is equal to maxSP, where maxSP is a positive integer.*

Since the above two properties refer to the 'size' rather than the 'capacity' of the respective containers, they make it clear that the elements of the containers should exist once the system has been initialised. The properties related to the maximum number of connections do not require the existence of instances of the Connection or ServiceRequest classes. They merely require that the classes that may **contain** instances of the Connection or ServiceRequest classes have sufficient **capacity** to **hold** these instances, hence the use of the word 'capacity' in those properties.

Property *AS2-08* implies that the number of service providers supported is bounded. This exposes another shortcoming of the informal problem statement: although the problem statement refers to service provider simulators, it does not specify the number of service provider simulators that should be supported.

Another sentence therefore needs to be added to the informal problem statement to clarify the role of the service provider simulators:
"In the case where the service providers are simulated, the service providers are replaced by maxSP of service provider simulators."

As a result, property *AS2-09* is formulated:

AS2-09. *The service provider container contains either service providers or service provider simulators.*

Note: In the remainder of this chapter, the terms 'service providers' and 'service provider simulators' are used interchangeably, unless otherwise stated.

The last clean behaviour property is the following:

AS2-10. The service provider subset of each service category contains at least one (simulated) service provider instance.

Property *AS2-10* is an example of an invariant that can only be satisfied after system initialisation has been completed. If the ServiceProviderSubset instance has been created, but none of the service provider instances have been created yet, then this property does not yet hold. Due to dependencies such as this one, invariants are not required to hold until after system initialisation has been completed.

Property *AS2-10* does not require the service providers in the service provider subset to be **active**. If the service provider subset contains at least one element, it implies that at least one service provider exists which has the **capability** to process service requests of that service category. The status, i.e. whether it is active or not, is an attribute of the service provider and it indicates whether the latter is presently able to process service requests. Property *AP1-04* deals with the impact of the status of the service provider on the handling of a service request.

5.4.1.3 AS3-yy (global invariants)

Global invariants describe the properties that should always be satisfied in order to ensure that something bad will never happen. In many cases **undesired behaviour** is **not described explicitly** in an **informal problem statement**, since the focus there is to describe the desired functionality. However, in order to ensure that undesired behaviour is **prevented**, one has to consider what such behaviour could possibly be. This demonstrates one of the advantages of a software development approach which focuses on correctness properties: it forces one to consider the situations that should be prevented as well as those that should occur. Such a **"constructive" approach** results in a product that is more reliable from the outset, since many of the problems that are often only discovered during the testing phase are now considered during the earlier phases of development.

The global invariants presented next are identified based on the informal problem statement given in Section 5.3.1. In some cases the properties can be inferred directly from the problem statement, but in other cases the requirement is implicit, as will be pointed out where relevant.

AS3-01. The establishment of a new connection implies that the associated service request is appended to the input queue.

The InputQueue class is used to model the First In First Out order in which service requests are handled by the call centre. The above property is required in order to ensure that the order of the elements of the input queue does indeed reflect the order in which the connections are established.

AS3-02. The category of each service request in a service queue is the same as the service category associated with the service queue.

AS3-03. Each service category is unique.

Properties *AS3-02* and *AS3-03* describe the correctness properties that are required in order to ensure that a service request is always placed into the correct service queue. Property *AS3-04*, presented next, can be derived from the above two properties.

AS3-04. A service request is only present in a single service queue at a time.

Property *AS3-04* is one of the properties that are required in order to ensure that a service request is not serviced in **duplicate**. The other properties are *AS3-05* and *AS3-06*. Note that the possibility of duplicate service requests is not mentioned at all in the informal problem statement.

It is assumed that the system will behave correctly and that includes not generating duplicate service requests. The **advantage** of formulating correctness properties is the fact that these **assumptions** are now being made **explicit**.

AS3-05.A service request is only present in the input queue if it has not been allocated to a service queue.

Thus, once it is allocated to a service queue, it is removed from the input queue. This property also ensures that service requests are not serviced in duplicate, but at a different level. It guarantees that the service request will not be entered into a service queue more than once.

AS3-06. A service request is only present in a service queue if it has not been serviced, i.e. if it has not been assigned to an element of the service provider subset.

Thus, once it is allocated to a service provider, it is removed from the service queue. This property also ensures that service requests are not serviced in duplicate. Should a service request not be removed, then it could first be assigned to SP1 and then to SP2. Both service providers could then proceed to service the request concurrently, which is not the desired behaviour. The following sentence is added to the informal problem statement:

AS3-07. Adding a service request to a service queue implies that a progress timer is started for the service request.

AS3-08. The category of each element of the service provider subset has to match one of the elements of the ServiceProviderCategories instance associated with the service category.

Phrased differently, property *AS3-08* specifies that the category of each service provider that may service requests associated with a particular service category, must match one of the service provider categories contained in the ServiceProviderCategories instance associated with that particular service category.

The combination of properties *AS3-03* and *AS3-08* indicates that a service provider category does not necessarily correspond with a service request category. This implies that the service providers can be designed with a categorisation that is independent from the categorisation of the service requests. The call centre is responsible for mapping the two types of categorisations. Associating a collection of service provider categories with each service category facilitates this and property *AS3-08* ensures that the mapping is performed correctly.

AS3-09. A service provider / service provider simulator services a single service request at a time.

Whereas properties *AS3-04* to *AS3-06* ensure that service requests are not serviced in duplicate, this property ensures that a service request **does not get lost**. Thus, once a service request is allocated to an element of a service provider subset, no other service request can be allocated to that element until the latter has completed servicing the first service request. It is therefore not possible to overwrite the first service request. This is another aspect of the call centre behaviour that is not described explicitly in the informal problem statement. The following sentence is therefore added to it:
"The call centre ensures that no service request is duplicated or lost."

The next invariant specifies that a service request may only be assigned to a service provider that can handle requests of the service category to which the service request belongs.

AS3-10. A service request may only be assigned to an element of the service provider subset associated with the service request category that matches the category of the service request.

The last invariant deals with the fact that a service provider can service multiple service categories.

AS3-11. A service provider / service provider simulator may be an element of multiple service provider subsets simultaneously.

The above global invariants were derived by considering every sentence of the informal problem statement and deciding whether it implied invariant behaviour. This included specifying desirable behaviour as well as specifying properties to prevent undesirable behaviour. It cannot be claimed that the list is complete (this is difficult when using a logic-based method [JiZh96] as was discussed in Section 2.4.3 of Chapter 2), but at least it provides a basis for reasoning about the correctness of the system in terms of those properties that have been specified. It also encourages the software designer to consider more than just those aspects that are explicitly mentioned in the problem statement, as was demonstrated by several of the properties listed above.

5.4.1.4 AS4-yy (unless properties)

The issue of the availability of the service providers is raised when one considers the implications of the following sentence in the problem statement: "Once a service request has been categorised and enqueued in a service queue, the call centre ensures that it is eventually assigned to a service provider, unless the service user hangs up beforehand." Thus, once a service request has been accepted by the call centre and enqueued in a service queue, it is guaranteed to be serviced by a service provider. The availability of a service provider therefore has to be guaranteed from that point onwards. Again, the initial problem statement is inadequate, since it does not specify explicitly whether service providers may go out of service and under what conditions. It is therefore updated as follows:

"It is assumed that if a service provider is available at the time when a service request is allocated to a service queue, but subsequently needs to go out of service, it remains available in a restricted fashion until that service request has been processed. The restriction is that it will not process any service requests that were added to the service queue after the service provider had made its intention to go out of service known. A service provider that operates in such a restricted mode may go out of service before the specified service request has been serviced only if at least one other service provider that can provide the specified service is available."

Careful analysis of this requirement yields a third service provider state, viz. restricted-active. This state is introduced to enable a service provider to indicate its intention of going out of service, while still honouring its commitments regarding service requests **already enqueued** in the relevant service queues. Thus, a service provider can either be active, non-active or active but operating in a restricted mode.

This requirement results in the following unless property:

AS4-01. If the number of active or restricted-active elements of a service provider subset is greater than zero, then it remains greater than zero unless the service queue associated with that service category is empty.

Thus, if at least one element of a service provider subset is active or restricted-active, then a minimum of one element must remain active or restricted-active for at least as long as there are elements in the service queue associated with that service category.

Note that it is not specified that an active / restricted-active element of the service provider subset has to remain active / restricted-active unless the service queue associated with the service category is empty. Such a property would be too restrictive. The way the property is formulated now, leaves room for many implementations. For example, one option is to implement the service provider in such a way that it may only go out of service if the service queues that it is servicing are empty. Another possibility is to allow a service provider to go out of service even if the affected service queues are non-empty, provided another service provider is active and able to service those requests. Yet another option is to allow the service provider to go out of service only if all the service requests that had been enqueued in the relevant service queues prior to its changing to the restricted-active state, have been serviced.

The unless property discussed in this section illustrates the need to consider the impact of **overspecification**. Since the analysis level properties should still hold once the design and implementation phases are reached, the analysis level properties should be sufficiently generic in order to allow for multiple design and implementation options.

5.4.2 Liveness properties

The three liveness (eventuality) properties listed in Section 5.2.4 are now considered, viz.
AL1-yy (total correctness),
AL2-yy (intermittent assertions) and
AL3-yy (responsiveness).

It is important that the software designer should keep the **SLOOP computational model** in mind throughout the specification of correctness properties, but the impact of this model is particularly evident in the **style** of the liveness and precedence properties. There is **no reference to flow of control**. The properties do not apply to specific program locations; once a SLOOP program is derived, they are quantified over all the statements in that program.

5.4.2.1 AL1-yy (total correctness)

The call centre software is cyclic (does not terminate), therefore total correctness does not apply to the system. Total correctness properties are defined for the sequential methods of the individual classes as shown in Section 5.4.5.1.

5.4.2.2 AL2-yy (intermittent assertions)

Intermittent assertion properties are specifically applicable to cyclic systems. It is used to specify that predicate p will eventually be followed by predicate q. In order to identify the relevant liveness properties, the informal problem statement is analysed in terms of its dynamic requirements. The general approach is to consider each class appearing in the class diagram and to scrutinise the problem statement for descriptions of progress related to that class. As will be evident from Section 5.4.3.1, most of the liveness properties in the call centre example are classified as safe liveness properties. In the case of a safe liveness property it is important that the predicate p should hold until predicate q holds.

AL2-01. Once a timer has been started, it will eventually expire or it will be stopped.

AL2-02. When a simulation event is required, a simulation timer is eventually started .

AL2-03. When a simulation timer expires, the simulation object eventually generates an event .

5.4.2.3 AL3-yy (responsiveness)

AL3-01. Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of the service provider container, or the service category allocator will reject the service request or the service user will hang up beforehand.

This property reveals a shortcoming in the informal problem statement. The latter specifies that an application may base the order in which the various service categories are served on some application-dependent criteria. It gives an example of a system which may support high and low priority categories and which might have a criterium to serve the high priority category before the low priority category as long as there are high priority service requests pending. Such a selection criterium would violate property *AL3-01*, since a continuous stream of high priority service requests could cause starvation of the low priority service requests. It is therefore important to modify the informal problem statement as follows:

"The order in which the various service request categories are served is based on some application-dependent criteria. For example, there may be high and low priority categories and the criterium might be to bias service towards the high priority category. It is, however, essential that the service category selection criteria should ensure that no service request category will be ignored for ever. "

In addition to responsiveness, the call centre example also requires fair responsiveness as indicated in Section 5.4.3.3.

Property *AL3-01* has implications regarding the **reliability** requirements of the system. It implies that if the service request is not aborted by the user and not rejected by the service category allocator, then the service request will be serviced by a service provider. Thus, when the service category allocator assigns a service request to a service queue, it implies that there is at least one active service provider that services service requests of that particular category (from properties *AP1-03* and *AP1-04*). Property *AL3-01* implies that in such a case, at least one of those service providers has to be active or restricted-active at the time when that service request reaches the head of the service queue and is assigned to a service provider. Thus, once a service request is allocated to a service queue (i.e. not rejected by the service queue allocator), the service request will be serviced. It is the function of the service category allocator to check whether at least one element of the appropriate service provider subset is active before it allocates the service request to the service queue.

5.4.3 Precedence properties

The three precedence (until) properties listed in Section 5.2.4 are now considered.

5.4.3.1 AP1-yy (safe liveness)

A safe liveness property deals with a predicate p that needs to hold **continuously** until the corresponding predicate q holds. The informal problem statement is therefore inspected to identify all the requirements that are of this format. The resulting safe liveness properties are as specified below.

Note that the word 'until' in the informal specification of the correctness properties in this section does not imply that the **until** logical relation should be used in the corresponding formal specification of the property. The same applies to the usage of the phrase 'it is ensured' and the **ensures** logical relation. The decision whether the **until** or **ensures** logical relation is more

appropriate is only made when the property is formalised during the design phase, since it is only during the design phase that the SLOOP statements are considered. At that stage it can therefore be decided whether the predicate *q* should be achieved via a single or multiple SLOOP parallel statements.

Property *AP1-01* ensures that the relationship between a connection and a service request remains unchanged while a service request is being processed by the call centre. The remaining properties deal with the path followed by a service request while being processed by the call centre. Properties *AP1-02* to *AP1-04* specify the actions that take place once the service request reaches the head of the input queue. These properties specify that the service request remains in that position in the queue until all the actions as specified by these properties have been performed.

Property *AP1-05* ensures that a service request remains in the input queue until it is assigned to a service queue. In turn, property *AP1-06* ensures that the service request remains in a service queue until it is assigned to a service provider. Finally, property *AP1-07* specifies that the service request remains assigned to a service provider until the required service has been performed.

Properties *AP1-08* and *AP1-09* deal with the safe liveness aspects of progress timers.

AP1-01.A connection is dedicated to a specific service request from the time that the connection is accepted until the connection is terminated by the service category allocator, the user hangs up or the service provider / service provider simulator has finished serving the user.

AP1-02. If the category of the first service request in the input queue has not yet been determined, then it remains uncategorised until its category is determined or the service user hangs up and the service request is removed from the input queue.

AP1-03. If the category of the first service request in the input queue has been determined and there is no active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is rejected by the service category allocator, removed from the input queue and the associated connection is terminated, or the service user hangs up and the service request is removed from the input queue.

AP1-04. If the category of the first service request in the input queue has been determined and there is an active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is removed from the input queue and appended to the service queue associated with that service request category, or the service user hangs up and the service request is removed from the input queue.

AP1-05.A service request remains in the input queue until it is assigned to a service queue, the associated connection is terminated by the service category allocator or until the service user hangs up.

AP1-06.A service request remains in a service queue until it is allocated to an element of the service provider container or until the service user hangs up.

AP1-07. A service request remains assigned to an element of the service provider container until the service provider completes the service and terminates the connection or until the service user hangs up.

AP1-08.If a progress timer has expired and the associated service request is still present in a service queue, then it is ensured that the timer is restarted and a service request progress update is sent to the corresponding service user.

AP1-09.If a progress timer has expired and the associated service request is no longer present in a service queue, then it is ensured that the timer is not restarted and no progress update is sent.

A number of the above properties, e.g. *AP1-01* and *AP1-07*, convey some of the **reliability** aspects of the system.

Property *AP1-01* implies that the communication provider provides a reliable communication service, since there is no option which specifies that the connection can be terminated by the communication service, i.e. it can only be terminated by the service user, the service category allocator or the service provider / service provider simulator.

Property *AP1-07* implies that a service provider will always service a request to completion, unless the service user aborts the request.

5.4.3.2 AP2-yy (absence of unsolicited response)

The correctness properties in this section provide another example of how the use of a checklist such as the SLOOP checklist of property types could lead to the explicit specification of many aspects of the behaviour of a system. In this case the problem statement indicates that the service requests in the service queues are serviced by service providers. Thus, it specifies what **should** happen. This behaviour is described by property *AP1-06*. However, in addition to this, property *AP2-01* ensures that service requests that are not enqueued in service queues cannot be assigned to service providers. Thus, it **also specifies what should not happen**. Property *AP2-02* has a similar purpose.

AP2-01.A service request is assigned to an element of the service provider container only if the service request has been enqueued in a service queue and has remained in the queue until it was assigned to the service provider container element.

AP2-02.A service request is allocated to a service queue only if the service request has been enqueued in the input queue and has remained in the latter until it was allocated to the service queue.

The informal problem statement is modified as follows:
"The call centre ensures that each service user is serviced on a first-come, first-served basis within each service request category. **This is the only way in which service requests are serviced, i.e. a service request is never serviced out of turn."**

5.4.3.3 AP3-yy (fair responsiveness)

AP3-01.Service requests are added and removed from the input queue on a First In First Out basis, except in the case where the user aborts a service request, in which case the service request may be removed from anywhere within the input queue.

Property *AP3-01* confirms the choice of the InputQueue class to model the order in which service requests are processed. Service requests are added to the end of the queue and removed from the head of the queue, except where the user aborts a service request, in which case the service request may be removed from anywhere within the input queue. The above property also implies that the relative ordering of the elements of the input queue always remains the same.

AP3-02.For each service queue, service requests are added and removed on a First In First Out basis, except in the case where the user aborts a service request, in which case the service request may be removed from anywhere within the service queue.

The service queues model the fact that service requests are processed on a first-come, first-served basis **within each category**. Service requests are added to the end of a service queue and removed from the head of the queue, except where the user aborts a service request, in which case the service request may be removed from anywhere within the service queue. The above property also implies that the relative ordering of the elements of a service queue always remains the same.

5.4.4 Consolidation

The informal problem statement is updated to rectify omissions and errors as identified during the problem analysis.

The problem statement is now rewritten in a tabular format, showing how the various properties listed above describe the requirements as given in the problem statement. This step forms part of the SLOOP method and it is performed in order to **cross-check** that all aspects of the problem statement are covered and also to verify that each specified property serves a useful purpose. The properties are checked for **inconsistencies** and **redundant** properties are removed.

For example, an earlier iteration[4] of the above properties contained the following global invariant:

AS3-12. A service request is not allocated to a service queue if the service provider subset associated with that service category does not contain at least one active element.

However, while performing the consolidation step, it was noticed that the safe liveness properties *AP1-03* and *AP1-04* include the safety aspect covered by property *AS3-12*. The two safe liveness properties as listed below are therefore sufficient to describe these particular requirements of the system.

AP1-03. If the category of the first service request in the input queue has been determined and there is no active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is rejected by the service category allocator, removed from the input queue and the associated connection is terminated, or the service user hangs up and the service request is removed from the input queue.

AP1-04. If the category of the first service request in the input queue has been determined and there is an active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is removed from the input queue and appended to the service queue associated with that service request category, or the service user hangs up and the service request is removed from the input queue.

The updated parts of the informal problem statement are shown in bold in the table below.

Note that the terms 'service provider' and 'service provider simulator' are used interchangeably in the table below, unless stated otherwise.

[4] For brevity earlier iterations are not shown .

Informal problem statement	**Correctness properties**
A call centre is a non-terminating system which ensures that dial-in users are serviced in a specified order by the available service providers.	*AL3-01*, *AP3-01* and *AP3-02*. The system is non-terminating, therefore no partial or total correctness properties are defined. Instead, responsiveness and fair responsiveness properties describe the way in which the system responds to events.
The service user places a call to a central (usually toll-free) number. Physically, this number may represent several lines (if line hunting is provided by the Public Switched Telephone Network), or it may represent a single high speed line which can carry multiple connections. The number of calls that can be handled **simultaneously** is therefore **bounded**. If the service user receives a ringing tone, it implies that a connection or line is available, otherwise a busy tone is signalled. The call centre is not responsible for controlling the ringing and busy tones. That is performed by the Public Switched Telephone Network (PSTN). **The call centre has a required capacity of maxConn connections, where maxConn is a positive integer. Once a connection has been accepted by the call centre, the associated service request may only be rejected if none of the active service providers have the capability to service the request. The service request may not be rejected due to any other resource problems.**	*AS2-01* to *AS2-03*. These clean behaviour properties model the restrictions on the number of simultaneous connections imposed by the communication provider. They also specify explicitly that the call centre itself imposes no **further** restrictions on the number of simultaneous connections (due to lack of resources) by specifying the capacity requirements of those modelling elements that are affected. *AS2-04* to *AS2-05*. These clean behaviour properties ensure that obsolete entries (i.e. service requests of connections that have been terminated) do not remain in the input queue or service queues. Thus, an entry in one of these queues is always associated with an active connection. *AL3-01*. This property specifies that once a connection is accepted, its associated service request may only be rejected by the service category allocator. *AP1-03*. The conditions for rejecting a service request once the connection has been accepted, are described in *AP1-03*.
The line or connection that is occupied by a service user remains so for the duration of the call, i.e. until the user hangs up, the connection is terminated due to a problem or the service provider has finished serving the user.	*AP1-01*, *AP1-03*, *AP1-07*. *AP1-01* describes the association between a connection and a service request. *AP1-03* specifies that a connection is terminated if the service category allocator rejects the service request. *AP1-07* ensures that a service provider eventually finishes servicing the service request.
The identity of the service user is irrelevant, therefore there is no upper limit to the number of service users, but they cannot all connect to the call centre simultaneously. Since the call centre does not store any information about the service users for switching purposes, there are no physical constraints regarding the maximum number of service users that can be supported.	This is modelled by the fact that there is no container class for the service users.
The call centre processes the calls on a first-come, first-served basis, i.e. as they are received from the PBX.	*AS3-01*, *AP3-01*. *AS3-01* specifies that the establishment of a new connection implies that the associated service request is appended to the input queue. *AP3-01* specifies that this queue is processed on a

	first-come, first-served basis.
If the call centre supports more than one service request category, it receives information about the type of service required by the service user together with the call. This information is obtained from the service user via an Interactive Voice Response (IVR) before the call reaches the call centre. The call centre uses this information to categorise the service required by the service user. If at least one service provider is active which provides a service of the specified category, the service request is accepted, otherwise it is rejected and the connection is terminated.	*AP1-02* to *AP1-05*. *AP1-02* deals with the categorisation of the service request, while *AP1-03* to *AP1-05* describe the requirements for the allocation of a service request to a service queue.
The call centre ensures that each service user is serviced on a first-come, first-served basis within each service request category. **This is the only way in which service requests are serviced, i.e. a service request is never serviced out of turn.**	*AS3-01*, *AP1-05*, *AP3-01*, *AS3-02* to *AS3-04*, *AP3-02*, *AS3-10*, *AP2-01* and *AP2-02*. When a connection is established, the associated service request is entered into the input queue (*AS3-01*), where it remains until it is allocated to a service queue (*AP1-05*). The input queue is a FIFO queue (*AP3-01*). *AS3-02* to *AS3-04* ensure that each service request is allocated to the correct service queue. *AP3-02* specifies that the service requests in each service queue are allocated to service providers on a first-come, first-served basis. *AS3-10* ensures that service requests are assigned to the correct service providers. *AP2-01* and *AP2-02* ensure that service requests are never serviced out of turn.
There are a number of queues (one for each service request category). **The call centre has to support maxCategories of service request categories. All supported service categories must exist after initialisation of the call centre.**	*AS3-02*, *AS3-03*, *AS2-07*. *AS3-02* states that there is a service category associated with each service queue and *AS3-03* specifies that each service category is unique. *AS2-07* is concerned with the number of service request categories that have to be supported.
Each service request is entered into one of these service queues and each queue is serviced on a first-come, first-served basis. **The call centre ensures that no service request is duplicated or lost.**	*AS3-04* to *AS3-06*, *AS3-09*, *AP2-02*, *AP3-02*. *AS3-04* to *AS3-06* deal with the prevention of duplicate service requests, while *AS3-09* is concerned with the possibility of service requests that could get lost. Since the previous four properties are based on the assumption that service requests reach service providers via service queues, two additional properties are required, viz. *AP2-02* and *AP3-02*. The former ensures that a service request cannot reach a service provider unless it has been enqueued in a service queue and AP3-02 ensures that the service requests in each service queue are allocated to service

	providers on a first-come, first-served basis.
The call centre ensures that service requests of one category are not affected by inadequate resources allocated to service requests of another category.	*AS2-03*.
Once the service request has been allocated to a service queue, the service user is informed at regular intervals of his/her progress towards being served.	*AS3-07*, *AL2-01*, *AP1-08* and *AP1-09*. *AS3-07* ensures that the progress timer is started when the service request is entered into a service queue. *AL2-01* states that a timer which has been started will eventually expire or it will be stopped. *AP1-08* specifies that the service user receives a progress update and the timer is restarted if the service request is still present in the service queue when the progress timer expires. *AP1-09* describes the behaviour if the service request is no longer present in the service queue when the timer expires.
If there is only one service request category, there is no need to obtain information from the service user in order to categorise the service required.	Since this information is obtained from the service user before the connection is processed by the call centre, no properties need to be specified to reflect this behaviour.
The order in which the various service request categories are served is based on some application-dependent criteria. **For example, there may be high and low priority categories and the criterium might be to bias service towards the high priority category. It is, however, essential that the service category selection criteria should ensure that no service request category will be ignored for ever.**	*AL3-01*. This property specifies that a service request will eventually be serviced, unless it is rejected or aborted.
There are one or more service providers. The number of service providers is bounded. **The call centre has to support maxSP of service providers. All supported service service providers must exist after initialisation of the call centre. In the case where the service providers are simulated, the service providers are replaced by maxSP of service provider simulators.**	*AS2-08* and *AS2-09*. These properties specify the number of service providers / service provider simulators that need to be supported. The fact that the number of service providers is bounded, implies that a service request may not necessarily be serviced immediately once it has been categorised. This is modelled by the fact that it is enqueued in a service queue and each service queue is required to have space for at least `maxConn` service requests (*AS2-03*).
Each service provider may service one or more service request categories. For example, one type of service provider (say the elementary type) may service queries only. Another type of service provider (say the advanced type) may be able to service both queries and other transactions. There are therefore one or more service provider categories.	*AS3-08*, *AS3-11*. The fact that a ServiceProviderCategories instance could contain multiple elements (*AS3-08*) implies that there can be one or more service provider categories. Each service category has a service provider subset associated with it. Property *AS3-11* states that a service provider may be an element

	of multiple service provider subsets simultaneously, which implies that a service provider may service one or more service request categories.
Each service request category should be serviced by at least one service provider category.	*AS2-10*, *AS3-08*. The combination of these two properties implies this requirement.
The service request categories and the service provider categories referred to in this problem statement are devised in order to provide flexibility in the way in which service requests are allocated to service providers. These categories may therefore be independent of any other service request categorisation performed by the service providers once a service request is allocated to one of them.	The categorisation that may be performed by the service providers is beyond the scope of the call centre and is not specified here.
Once a service request has been categorised and enqueued in a service queue, the call centre ensures that it is eventually allocated to a service provider, unless the service user hangs up beforehand. The nature of the service being provided is irrelevant to the call centre.	*AP1-06*, *AP1-07*, *AS2-06*. *AP1-06* specifies that once a service request is entered into a service queue, it remains there until the connection is terminated or until it is assigned to a service provider. *AP1-07* ensures that a service provider eventually finishes servicing the service request. *AS2-06* ensures that a service provider becomes free once it has finished servicing a service request.
The connections between the service providers and the call centre may be via the PSTN or they may be established internally via the local PBX.	The communication mechanism between the call centre and the service providers forms part of the environment of the call centre and need not be specified here.
The initial product needs to be tested by simulating the actions of the service users, service providers and the communication provider.	*AL2-02*, *AL2-03*. The simulation of the communication provider and service providers rely on simulation timers to trigger events.
System boundaries: The service providers, service users and communication service form part of the environment of the call centre, i.e. the latter performs a queuing and switching function only. The service providers contain the software to process a service request. This software may vary from application to application and therefore does not form part of the call centre. The service providers may even be supplied by a third party. The software that needs to be designed is restricted to the part pertaining to the call centre and its interface with its environment. This is the part that remains unchanged between applications.	*AS2-01*, *AS3-01*, *AS3-08*, *AS3-09*, *AS3-11*, *AS4-01*, *AP1-07*. These properties specify the requirements that have to be satisfied by the objects in the environment of the call centre. *AS2-01* specifies the requirement that the communication provider should ensure that the maximum number of connections supported by the call centre is not exceeded. *AS3-01* states that the service request associated with a new connection has to be appended to the input queue at the time when the new connection is established. *AS3-08* ensures that each element of the service provider subset belongs to one of the categories in the ServiceProviderCategories instance associated with that service category. *AS3-09* requires that a service provider should not service more than one service request at a time. *AS3-11* specifies that a service provider may be an element of

	multiple service provider subsets simultaneously. *AS4-01* implies that an active service provider subset element may not become inactive if it is the only active element in that subset and the associated service queue is non-empty. *AP1-07* specifies that a service provider will complete the service requested by a service request once the latter has been assigned to it. It also specifies that it will terminate the connection once it has completed the service.
Assumptions: At this level of abstraction a reliable communication medium is assumed between the service users and the call centre as well as between the call centre and the service providers.	*AP1-01.* Failure of the communication medium is not listed as a reason for terminating a connection. This is based on the assumption that the communication medium is reliable.
It is assumed that if a service provider is available at the time when a service request is allocated to a service queue, but subsequently needs to go out of service, it remains available in a restricted fashion until that service request has been processed. The restriction is that it will not process any service requests that were added to the service queue after the service provider had made its intention to go out of service known. A service provider that operates in such a restricted mode may go out of service before the specified service request has been serviced only if at least one other service provider that can provide the specified service is available.	*AS4-01*, *AL3-01.* Property *AS4-01* specifies the required behaviour of the service provider, while property *AL3-01* describes the guarantee that a service request that has been enqueued in a service queue will be serviced unless the service user aborts the request.

The correctness properties specified in Sections 5.4.1 to 5.4.4 describe the desired behaviour of the system as a whole. Some of these properties result from the collaboration of a number of classes, whereas others can be derived from the correctness properties of an individual class. For example, property *AL3-01* requires the collaboration of a number of classes. It states:
Once a service user has connected to the call centre, the associated service request will eventually be serviced by an element of a service provider subset, or the service category allocator will reject the service request or the service user will hang up beforehand.

This property requires the ServiceCategoryAllocator instance to process a ServiceRequest instance. It also requires a ServiceCategory instance to assign the ServiceRequest instance to the appropriate ServiceProvider instance eventually.

On the other hand, property *AS3-03* pertains to the ServiceCategory class only. It states:
Each service category is unique.

The next section deals with the specification of the correctness properties of individual classes that form part of the system under development.

5.4.5 Informal specification of correctness properties of individual classes

After the correctness properties of the system as a whole have been specified, the next step of the analysis phase is to identify the correctness properties of the individual classes. The properties of the CommsProviderSimulator and ServiceProviderSimulator classes are shown here as examples. These particular classes were chosen as examples, because they have some properties in common. During the **design** phase these **common properties** are used to determine whether the design can be improved by extracting a **common superclass** from these classes. This aspect of the SLOOP method is discussed in Chapter 6, Section 6.2.3.

The class properties eventually result in the **identification** of the **methods** of these classes during the design phase. During the design phase the properties of the individual methods are also specified. Thus, the specification of class properties deals with the population of the *properties-section* within each class definition. The purpose of the **class properties** is to specify the behaviour of the **class and its instances**. It describes the collective behaviour of the methods of the class and its instances at a high level of abstraction. Once the individual methods are defined during the design phase, their individual correctness properties are specified in the *properties-section* of each method. The only method correctness properties that are specified during the analysis phase, are those of the instance creation methods. This is to facilitate the specification of the initial state of an instance.

The **numbering scheme** for class properties is the same as for system properties, except that the property identification numbers are suffixed with the class name in brackets. The class name may be omitted if it is clear from the context which class the properties refer to. The numbers used within each class are independent of the numbers used for the system properties. The numbers of the correctness properties that are specified within the *properties-section* of each method of a class are unique within that class, not within each method. Thus, for each new method that is specified for a class, the correctness property number follows on from the number of the last property in the same category that was specified for that class or one of its methods.

If a correctness property is **overridden** in a subclass, then the modified property in the subclass is identified by the same number and class name suffix as the property in the superclass. For example, if property *DL1-05(CC_Activation)* of the CC_Activation class is overridden in its subclass, CC_SimulationActivation, then the property number in the CC_SimulationActivation subclass remains *DL1-05(CC_Activation)*. The fact that the suffix of a property refers to a superclass indicates to the reader that the property is overriding a superclass property. Any correctness property that is inherited **unmodified** by a subclass is not repeated in the list of correctness properties defined for the subclass.

If a class is used as a building block in a larger system, then it must be ensured that the class properties are **not in conflict** with any of the correctness properties of the larger system. However, it must be possible to determine the behaviour of a class and its instances solely by inspecting the correctness properties of the class and its methods, i.e. it should not be necessary to refer to the properties of any system in which the class might be used in order to understand the behaviour of the class. For example, when the TimerServices class definition is inspected, it should be clear from its correctness properties what services its instances will provide and what would be expected from its clients. This makes it possible to reuse the TimerServices class in many different systems, not only in a call centre system.

The correctness properties that are specified during the **analysis** phase are formulated in terms of the logical relations that are defined for the SLOOP computational model. The only exception is the **results-in** logical relation, which is used in the total correctness properties of the sequential instance creation methods. These methods are used to define the initial state of an object after instance creation.

5.4.5.1 The CommsProviderSimulator class

The purpose of the CommsProviderSimulator class is to simulate events from the communication provider. These events are the connection attempts by service users and they may occur at random intervals. Thus, from the time that a CommsProviderSimulator instance is created, it needs to try and establish new connections to the call centre at random intervals. The random intervals are modelled by starting a random timer after each event has been generated and the generation of the next event when the timer expires.

The behaviour of the simulator **should not violate any of the correctness properties specified for the call centre as a whole**. In particular, it has to ensure that the capacity restrictions are not violated (property *AS2-01*) and that a service request entry is made into the input queue when a new connection is established (property *AS3-01*). This is evident from the table in Section 5.4.4, which shows that properties *AS2-01* and *AS3-01* apply to the CommsProviderSimulator class in particular. System properties *AL2-02* and *AL2-03* refer to simulation timers and simulation events, therefore they are also applicable to the CommsProviderSimulator class.

The correctness properties below describe the behaviour of the CommsProviderSimulator class. As stated previously, the behaviour of each class should be fully specified via its correctness properties. Thus, it should not be necessary to refer to the system properties for the specification of certain aspects of the behaviour of a class. For this reason it might be necessary to duplicate some of the properties specified for the call centre system in the list of properties specified for the class. That would be case when a property specification in the class definition does not need to add further detail (e.g. properties *AL2-01* and *AL2-02* of the CommsProviderSimulator class). In other cases new properties (such as *AP1-01* and *AP1-02* of the CommsProviderSimulator class) are specified that describe the specific role of the CommsProviderSimulator with respect to the behaviour specified for the system as a whole.

AL2-01 (CommsProviderSimulator). When a simulation event is required, a simulation timer is eventually started .

AL2-02 (CommsProviderSimulator). When a simulation timer expires, the simulation object eventually generates an event .

AP1-01 (CommsProviderSimulator). If an event has to be generated and the maximum number of connections have not yet been established, the communication provider simulator ensures that a new connection is established, the associated service request is appended to the input queue and a new communication provider simulator event is again required.

AP1-02 (CommsProviderSimulator). If an event has to be generated and the maximum number of connections have already been established, the communication provider simulator ensures that the event is cancelled and a new communication provider simulator event is again required.

The following total correctness property is specified for the instance creation method of the CommsProviderSimulator class:

AL1-01(CommsProviderSimulator). Instance creation results in the initialization of the instance variables of the CommsProviderSimulator class and its superclasses and in a communication provider simulation event being required.

The CommsProviderSimulator class ensures that system property *AS2-01* is satisfied by adhering to property *AP1-02* (CommsProviderSimulator), while system property *AS3-01* is preserved via property *AP1-01* (CommsProviderSimulator).

Properties *AL1-01*, *AL2-01*, *AL2-02*, *AP1-01* and *AP1-02* of the CommsProviderSimulator class ensure that the cyclic nature of the system is achieved.

5.4.5.2 The ServiceProviderSimulator class

The purpose of the ServiceProviderSimulator class is to simulate the actions of the service provider. The latter has to process the service requests from the service users. The time it takes to process a service request may vary. Thus, once a service request has been assigned to a service provider simulator, it needs to simulate the random period it takes to service the request. The random period is modelled by starting a random timer when the service provider simulator detects that a service request has been assigned to it. Once the timer expires, the connection is terminated.

The behaviour of the ServiceProviderSimulator class should not violate the correctness properties specified for the call centre as a whole. The table in Section 5.4.4 shows that properties *AS3-08*, *AS3-09*, *AS3-11*, *AS4-01* and *AP1-07* apply to the ServiceProviderSimulator class in particular. Properties *AL2-02* and *AL2-03* refer to simulation timers and events, therefore they are also applicable to the ServiceProviderSimulator class. For brevity, the term 'simulator' will be used in the remainder of this section when referring to a ServiceProviderSimulator instance.

AS3-08 ensures that each simulator in a service provider subset belongs to one of the categories in the ServiceProviderCategories instance associated with that service category. *AS3-09* requires that a simulator should not service more than one service request at a time. *AS3-11* allows a simulator to be an element of multiple service provider subsets simultaneously. *AS4-01* implies that an active / restricted-active simulator may not become inactive if it is the only active / restricted-active element in a service provider subset and the associated service queue is non-empty. *AP1-07* specifies that a simulator will complete the service requested by a service request once the latter has been assigned to it. It also specifies that it will terminate the connection once it has completed the service. Property *AL2-02* states that a simulation timer is started when a simulation event is required. Upon expiry of the simulation timer, a simulation event is generated, as specified in property *AL2-03*.

The properties that describe the behaviour of the ServiceProviderSimulator class are now listed, followed by a description of how these properties relate to the previously mentioned properties of the call centre as a whole, viz. *AS3-08*, *AS3-09*, *AS3-12*, *AS4-01*, *AL2-01*, *AL2-02* and *AP1-07*.

AS3-01(ServiceProviderSimulator). A service provider simulator services a single service request at a time.

AS4-01 (ServiceProviderSimulator). If the service provider simulator is active or restricted-active, then it remains active or restricted-active unless an empty service queue is associated with each service category which has this service provider simulator as an element of its service provider subset or the service provider subsets of each of these service categories contain other service provider simulators that are active or restricted-active.

AL2-01 (ServiceProviderSimulator). When a simulation event is required, a simulation timer is eventually started .

AL2-02 (ServiceProviderSimulator). When a simulation timer expires, the simulation object eventually generates an event .

AP1-01 (ServiceProviderSimulator). The assignment of a new service request to the service provider simulator ensures that a new service provider simulator event will be required.

AP1-02 (ServiceProviderSimulator). If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request.

AP1-03 (ServiceProviderSimulator). A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection or until the service user hangs up.

The following total correctness property is specified for the instance creation method of the ServiceProviderSimulator class:
AL1-01(ServiceProviderSimulator). Instance creation results in the initialization of the instance variables of the ServiceProviderSimulator class and its superclasses and it also results in the registration of the service provider simulator with the relevant service categories.

Property *AS3-01(ServiceProviderSimulator)* ensures that property *AS3-09* of the call centre system is satisfied. Property *AS4-01* is an example of a system property that is now rewritten from the perspective of one of the classes involved. Property *AL1-01 (ServiceProviderSimulator)* describes the initialization of the simulator. System properties *AS3-08* and *AS3-11* are preserved via property *AL1-01 (ServiceProviderSimulator)*.

Properties *AS3-01*, *AL2-01*, *AL2-02*, *AP1-01* and *AP1-02* of the ServiceProviderSimulator together ensure that property *AP1-03 (ServiceProviderSimulator)* is satisfied. The latter is the equivalent of system property *AP1-07*, but with specific reference to the ServiceProviderSimulator class.

In Chapter 6 it is shown how the properties of the above two classes are used to identify a class in the repository with similar properties. The latter is eventually used as a parent class of the above two classes.

5.4.6 Summary of the requirements analysis phase steps

The steps that are performed during the analysis phase are summarised below:

- Analyse the informal problem statement.
- Create the class diagram.
- Determine the system boundaries.
- Specify the required behaviour of the system via informal correctness properties. This is done by considering each type of property in the SLOOP checklist of correctness properties. Inspect the informal problem statement in order to specify correctness properties of each property type in turn.
- Update the informal problem statement if deficiencies are found.
- Consolidate the property specifications. This time it is not the checklist of property types that is used as basis for the inspection of the property specifications. Instead, each sentence of the problem statement is checked to make sure that each aspect is sufficiently covered by the specified correctness properties. Remove redundant properties and add additional properties if necessary.
- Specify the analysis level properties for the classes comprising the system. Ensure that they do not violate any system properties and have sufficient functionality to provide what is required by system.

Iteration between any of the above steps is possible. The procedures described here have been demonstrated in detail in the previous subsections.

5.5 Summary

This chapter has sought to illustrate the feasibility of object-oriented analysis driven by the correctness properties of the system. The static structure of the system is obtained using any of the conventional methods. However, when the **behaviour** is specified, it is described in terms of **correctness properties**.

A case study was used to illustrate the benefits of such an approach. The following advantages were noted:

- **Deficiencies** in the problem statement can be **revealed**.
- The focus is not only on what the behaviour should be, but also on what it should not be. **Error conditions** are therefore considered from the outset.
- It assists the software designer in gaining a **thorough understanding** of the requirements.

The above benefits are evident during the requirements analysis phase, but there are additional advantages that only become apparent during the design phase. These aspects will be discussed in the next chapter, but a brief summary is given below:

- The correctness properties specifying the behaviour of the system under development and its constituent classes are used to **identify** possible matching frameworks / design patterns / classes in the repository of **reusable** SLOOP artifacts.
- Although the specification of the informal analysis phase correctness properties is not trivial, the effort required to translate these properties into formal ones, as well as to formalise the design level refinements, can be saved if appropriate reusable artifacts can be found in the repository. The **correctness properties** are therefore **reused** along with the classes themselves. The **scalability** problem is therefore addressed by taking advantage of the reusability aspect of object-orientation.
- The approach followed during the analysis phase encourages the software designer to think in terms of **properties** that are **universally** or **existentially** quantified over **all statements** in the program instead of focussing on **loci of control**. This paves the way for the design phase, where these properties are refined and the **SLOOP statements** are derived. It prepares the designer to write statements that are based on the computational model described in Chapter 4, thereby promoting **seamlessness** between the analysis and design phases.

The **understandability** problem was also addressed in this chapter. The case study was used as a vehicle to explain how the various correctness types are interpreted and to provide examples of their application. The purpose of the checklist developed in Section 5.2.4 was to assist the software designer in avoiding **underspecification**. The issue of **overspecification** was also addressed. The importance of maintaining a **sufficient level of abstraction** was pointed out. That ensures that multiple design options can be considered during the design phase.

The implications of using the SLOOP method during the design phase is the topic of the next chapter.

CHAPTER 6

DESIGN FROM THE SLOOP PERSPECTIVE

6.1 Introduction

The design phase comprises a number of important steps, viz.

- identifying reusable artifacts in the repository (these may be frameworks, design patterns and/or classes),
- refining the specification,
- identifying class and instance methods based on the correctness properties,
- formally specifying correctness properties,
- deriving SLOOP statements,
- reasoning about the correctness of the SLOOP program,
- possibly creating prototypes and
- improving the design using design patterns.

This chapter covers the first five aspects listed above, while Chapter 7 deals with correctness reasoning. Chapter 8 shows how a SLOOP program, one of the deliverables of the design phase, can be implemented and Chapter 9 elaborates further on the topic of incorporating design patterns.

Multiple iterations over all these steps may be required. In the sections below the issues discussed for each step do not necessarily belong to the same iteration. For example, Section 6.2 describes the identification of reusable artifacts. Some reusable classes may already be discovered during the first iteration, whereas it may be necessary first to perform some design level refinements (the topic covered in Section 6.3) in order to discover some of the other reusable classes. The actions described below are therefore not described in chronological order, but rather according to the topics of identifying reusable artifacts, refining the specification, formally specifying correctness properties and deriving SLOOP statements.

One of the issues that are highlighted in this chapter is the fact that the **reuse** of artifacts stored in a repository **influences** the way in which the properties of design level refinements are reconciled with the analysis level properties. In the SLOOP method this step in the software development process is called the **mapping** of problem domain objects onto solution domain objects, in order to reflect the fact that it is not purely a refinement step.

It will be argued that the design process is not merely a matter of refinement and proving that the refinement preserves the higher level properties. In some cases the reusable component may have additional properties that are not required by the analysis level class, but also do not violate any of the properties of the system under development. In that case the class from the repository can still be used to represent the analysis level class. Furthermore, the reusable class description might contain methods that are not required by the new system and which may or may not

violate the properties of the latter. The SLOOP approach under these circumstances is described in Section 6.2.1.

That section also highlights the impact of placing such emphasis on reusability during the design phase. While trying to map analysis level classes onto classes from the repository, the responsibilities allocated to the various analysis level classes might have to be adjusted in order to take advantage of the reusable classes. This often results in a more elegant solution. Whereas the aim in conventional refinement methods usually is to merely add more detail at each refinement level [BaWr89, Back89, ShLa89], the focus in the SLOOP method is to identify reusable artifacts while adding more detail. As a result adjustments could be made to earlier levels of refinement, provided the system still meets its requirements. This characteristic of the SLOOP method is discussed further in Section 6.2.1.

The **level of formalisation** of correctness properties is another topic that receives attention in this chapter. This is particularly relevant in view of the stated goal of making the SLOOP method **usable by practising software designers.**

Finally, it is shown how SLOOP statements are arrived at and how a SLOOP program is constructed from its constituent classes.

Note that although the discussions in this and other chapters of this thesis usually refer to the actions taken by "the software designer", that does not imply that the SLOOP method restricts one to a single-person approach to software engineering. Since the SLOOP method is an object-oriented method, a system is designed as a set of classes. The modularity of such a system makes it particularly suitable for a **work-sharing multi-person approach** towards software development.

6.2 Identifying reusable artifacts

This chapter continues with the call centre case study. The first step is to try and find one or more frameworks that match one or more subsystems. The class diagram and correctness properties identified during the analysis phase are compared with those of the frameworks in the repository. As noted in Chapter 5, the format of the correctness properties yielded as a deliverable of the analysis phase is informal. The properties of the artifacts in the repository are recorded both formally and informally.

The software designer needs to compare the informal property specifications of the system under development with those of the artifacts in the repository and decide whether they convey the same meaning. The formal specification of the property in the repository is used to confirm that its associated informal description is interpreted correctly.

The question arises whether it would not have been more efficient to have compared **formal** descriptions of properties. There are two reasons for rejecting this option. The first was pointed out in the previous chapter, viz. if a match for the informal property is found, then the **translation** from the informal to the formal format of the property is **reused**.

The second is the fact that even formal descriptions of the same property might **not** be **identical**. For example, different names might be used for the classes, instances and methods referred to by the property. Furthermore, the properties might not be specified at exactly the same level of abstraction. This is particularly true of the property specifications of frameworks, which are likely to be at a higher level of abstraction than that of the specific application being developed. It is the task of the software designer to recognise such differences in levels of abstraction and to take advantage of any reusable artifacts.

This particular problem, viz. finding components in a repository and recognising their applicability, is also discussed in [SiCh97]. It is stated there that "a specification describing a component can be expressed in many equivalent ways". It is also claimed in [SiCh97] that it is **not tractable** to establish the applicability of a component in the repository **automatically**. It is the task of the software designer to determine whether a component in the repository is equivalent to or a refinement of the component defined for the system under development.

As indicated by the flowchart in Figure 4-10(a), if a framework exists which describes the complete system, it is instantiated, i.e. object composition and subclassing techniques are used to adapt the framework to the system being developed. In very large systems, framework(s) might exist for part(s) of the system. In that case the relevant framework(s) are instantiated and the remainder of the system is developed as described below.

If no suitable frameworks exist in the SLOOP repository, the individual classes are considered. The properties that were identified for each class during the analysis phase are used to determine whether appropriate reusable classes can be found in the repository. The class descriptions in the repository are in the SLOOP format. For the purposes of this example it is **assumed** that the **repository** contains a SLOOP description of the `EventSimulator` abstract class, as well as SLOOP descriptions of the Smalltalk library classes.

6.2.1 Mapping problem domain objects onto solution domain objects

It is found that some of the existing generic solution domain classes in the repository, such as the Smalltalk `OrderedCollection` and `Array` classes, have sufficient functionality to represent some of the new classes defined during the analysis phase. For example, the `InputQueue` class could be replaced by `OrderedCollection`, since all the properties identified for the `InputQueue` class during the analysis phase are supported by the `OrderedCollection` class. Before presenting the motivation for this decision, the following must be noted regarding the discussion that follows:

- The discussion of this example is at a detailed level for illustrative purposes.
- In the discussion below the specifications of the properties of the `InputQueue` and `OrderedCollection` classes correspond almost *verbatim*. They only differ when a particular issue is illustrated by the difference. In practice the properties of two classes specified by different authors are likely to be much further apart from each other, both in number and in wording. However, since the purpose of this discussion is to illustrate certain aspects of the task of finding appropriate reusable classes, the examples are written in such a way so as to focus on the issues being highlighted. The discussion is illustrative of the task in broad terms.

Although the identification of the analysis level correctness properties of the `InputQueue` class formed part of the actions discussed in the previous chapter, they were not discussed (for brevity, only the correctness properties of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes were covered there). The analysis level correctness properties of the `InputQueue` class are now described in order to provide the necessary background to the present discussion.

Listed below are all the analysis level call centre correctness properties that **refer** to an `InputQueue` instance. This list is used as the basis for the identification of the analysis level correctness properties of the `InputQueue` class.

AS2-02. The minimum capacity of the input queue is equal to the capacity of the connection container.

AS2-04. If a connection is terminated, it implies that its associated service request is not present in the input queue.

AS3-01. The establishment of a new connection implies that the associated service request is appended to the input queue.

AS3-05. A service request is only present in the input queue if it has not been allocated to a service queue.

AP1-02. If the category of the first service request in the input queue has not yet been determined, then it remains uncategorised until its category is determined or the service user hangs up and the service request is removed from the input queue.

AP1-03. If the category of the first service request in the input queue has been determined and there is no active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is rejected by the service category allocator, removed from the input queue and the associated connection is terminated, or the service user hangs up and the service request is removed from the input queue.

AP1-04. If the category of the first service request in the input queue has been determined and there is an active service provider / service provider simulator servicing that particular service request category, then it is ensured that the service request is removed from the input queue and appended to the service queue associated with that service request category, or the service user hangs up and the service request is removed from the input queue.

AP1-05. A service request remains in the input queue until it is assigned to a service queue, the associated connection is terminated by the service category allocator or until the service user hangs up.

AP2-02. A service request is allocated to a service queue only if the service request has been enqueued in the input queue and has remained in the latter until it was allocated to the service queue.

AP3-01. Service requests are added and removed from the input queue on a First In First Out basis, except in the case where the user aborts a service request, in which case the service request may be removed from anywhere within the input queue.

When the above properties are analysed, it is clear that the only actions that are performed on the input queue itself, are to **add, remove and access** elements. Many of the properties specify **additional** issues, such as the fact that a new element is appended to the queue **when a connection is established** (*AS3-01*) and the fact that when a service request is removed from the input queue, then **it is appended to a service queue** if the service request is not rejected or aborted (*AP1-05*) . However, this additional information is not relevant to the characteristics of the input queue itself.

The above properties therefore yield the following information regarding the characteristics of the input queue itself :

- The input queue has a **minimum capacity**, which must be equal to the capacity of the connection container (AS2-02).
- Elements are **appended** to the input queue (AS3-01, AP3-01).
- Elements are **removed** from the head of the queue (AS2-04, AP1-02, AP1-03, AP1-04, AP3-01), as well as from any other position in the input queue (AS2-04, AP1-05, AP2-02, AP3-01).
- Elements are **accessed** for the purpose of categorisation (AP1-02).

It is clear that the input queue is responsible for maintaining the **relative** ordering of its elements. However, when deriving the correctness properties of the `InputQueue` class, one has to determine whether the FIFO order in which elements are **added** or **deleted** from an `InputQueue` instance should be the responsibility of the instance or of its clients. Analysis of property *AP3-01* yields the insight that although an `InputQueue` instance could force new elements to be added to the tail of the queue at all times, it should not enforce removal of

elements to be from the head of the queue only. This is because it might be necessary to remove an element from an arbitrary position in the queue if the connection is aborted.

This implies that the responsibility for ensuring that the elements of the queue are processed in a FIFO order should lie with the clients of the input queue. Thus, the `InputQueue` instance should provide methods to add an element to the tail of the queue, remove an element from the head of the queue and remove an element from any position in the queue. However, it is the responsibility of the client to invoke these methods in such a way that elements are added and deleted in a FIFO order under normal circumstances (i.e. when a service request is not aborted). Since the `InputQueue` instance will not enforce the FIFO order, it will therefore not present a problem if the `InputQueue` instance provides a method to add an element to any position in the queue, as will be seen below.

The correctness properties of the `InputQueue` class resulting from the analysis phase should be consistent with the foregoing properties that relate to the characteristics of the input queue. They are identified as follows:

AS3-01 (InputQueue). The elements of the input queue are always in contiguous positions.
AS4-01 (InputQueue). The relative ordering of any two elements of the input queue remains the same unless one of the elements is removed.
AS4-02 (InputQueue). An object that is not an element of the input queue remains not an element unless it becomes an element of the input queue at a specified position.
*AS4-03 (InputQueue). An object that is an element of the input queue remains an element unless the **object** is specified and it is removed from the input queue or the **position** of the object is specified and the corresponding object is removed from the input queue.*
AS4-04 (InputQueue). The capacity of the input queue is non-decreasing (i.e. the capacity of the input queue remains the same unless it grows).

The following is the total correctness property of the `setup` instance creation method of the `InputQueue` class:
AL1-01 (InputQueue). Instance creation results in the capacity of the instance being equal to the capacity of the connection container.

The *AS3-01*, *AS4-01*, *AS4-02* and *AS4-03* `InputQueue` properties together describe the role of the `InputQueue` class in the preservation of property *AP3-01* of the call centre system. Note that the *AS4-02* and *AS4-03* `InputQueue` properties merely allow the client to add and delete elements at specified positions; the values of those positions are the responsibility of the client. The *AL1-01* and *AS4-04* `InputQueue` properties are used to preserve property *AS2-02* of the call centre system.

The above `InputQueue` properties are used during the search for a matching class in the SLOOP repository. The relevant correctness properties of the Smalltalk `OrderedCollection` class are as follows:
AS3-01 (OrderedCollection). The elements of anOrderedCollection are always in contiguous positions.
AS4-01 (OrderedCollection). The relative ordering of any two elements of anOrderedCollection remains the same unless one of the elements is removed.
AS4-02 (OrderedCollection). An object that is not an element of anOrderedCollection remains not an element unless it becomes the last element of anOrderedCollection if its position is unspecified or it becomes an element of anOrderedCollection at a specified position.
*AS4-03 (OrderedCollection). An object that is an element of anOrderedCollection remains an element unless the **object** is specified and it is removed from anOrderedCollection or the **position** of the object is specified and the corresponding object is removed from anOrderedCollection.*

AS4-04 (OrderedCollection). The capacity of anOrderedCollection is non-decreasing (i.e. the capacity of anOrderedCollection remains the same unless it grows).

The `OrderedCollection` class has multiple instance creation methods (e.g. the `new` method creates an ordered collection that has a default capacity of 10, while the `with:` method does the same, but also enters the argument of the method as an element of the collection). The instance creation method that is relevant to the `InputQueue` class is the `new:` method. It has the following total correctness property:
AL1-01 (OrderedCollection). If the capacity specified as the argument of the method is greater than zero, then it results in a new instance that has a capacity equal to the value of the argument.

Properties *AS3-01*, *AS4-01* and *AS4-03* of the `InputQueue` and `OrderedCollection` classes are the same, except for the names of the classes. Property *AS4-02 (OrderedCollection)* is an example of a case where the correctness property of the class found in the repository is **not exactly the same** as that of its counterpart in the system under development. However, there is nothing in the `OrderedCollection` property which violates the requirements of the call centre system. More specifically, it does not violate property *AP3-01* of the call centre system. Even though the postcondition of the `OrderedCollection` property is weaker than that of the `InputQueue` property (the former has an additional *or* clause), this property is still acceptable because it does not violate the system requirement to add elements to the tail of the queue.

Property *AL1-01* of the `OrderedCollection` class guarantees that an instance will have a certain specified capacity immediately after its creation. The corresponding `InputQueue` correctness property specifies that the capacity of an instance will be the same as the capacity of the connection container (in the call centre system). Property *AL1-01 (InputQueue)* implies that it is the responsibility of the `InputQueue` class to ensure that the capacity of an `InputQueue` instance is the same as that of the connection container (in the call centre system). By explicitly referring to the capacity of the connection container in the property of the `InputQueue` class, it immediately rules out a design which reuses a generic collection class such as the Smalltalk `OrderedCollection` class. This suggests that the **distribution** of the **responsibilities** allocated to the various classes during the analysis phase should be modified.

Note that such a modification is not required because the analysis level properties are incorrect, but because a **reusable** class was found which represented **a more elegant solution** to the problem. Note also that there is no modification to the system properties of the analysis phase. It merely represents a different distribution of responsibilities amongst the **constituent** classes of the call centre system.

Property *AL1-01 (InputQueue)* is modified as indicated below in order to reflect the shift in responsibilities:
AL1-01 (InputQueue). Instance creation of the InputQueue class results in the capacity of the instance being equal to the specified capacity.

The modification to this analysis level correctness property of the `InputQueue` class now enables one to map the problem domain `InputQueue` class successfully onto the solution domain `OrderedCollection` class that is found in the SLOOP repository of reusable artifacts. This example demonstrates how the **emphasis on reusability influences the refinements** that are performed at each stage of the software development process. Thus, it is not merely a matter of taking the analysis level correctness properties and adding more detail during the design phase. Instead, it is a case of determining whether the reusable classes found in the repository could possibly provide some of the required functionality and then adjusting the distribution of responsibilities amongst the classes of the call centre system accordingly **if necessary**. It is of course possible that reusable classes could be found that add design level detail only, which would make it a case of pure refinement. In the other case, i.e. where the match is not exact, the

reusable class still represents a refinement, since it also contains design level detail such as the algorithms to implement its methods.

This example highlights the fact that the reusable class may contain many methods that are not necessarily required by the system under development. Some of the **individual** methods may contain correctness properties that are in conflict with the correctness properties of the system. For example, the `new` method of the `OrderedCollection` class creates an instance that has a capacity of 10 elements. This implies that the client which sends the `new` message to the `OrderedCollection` will not be able to ensure that the capacity is equal to that of the connection container. However, the important issue here is that **none** of the properties that are defined at the **class/instance level** of the reusable class should be in conflict with the correctness properties of the system. It is the responsibility of the system designer to ensure that any **method** that is used always satisfies the system properties.

There are several ways of addressing this last issue. One option is to create a subclass of the reusable class and to prohibit the use of any of the methods that do not satisfy the system properties. However, this requires a great deal of effort to be spent on studying the properties of methods that might be totally irrelevant to the system under development. The approach that is followed in the SLOOP method is to place the emphasis on the methods that **are** to be used. This is done by explicitly listing all the methods that are being used by the new system in the *partial-class-methods-section* of the appropriate *partial-class-description* of the SLOOP program. It suffices to ensure that the correctness properties of these methods do not violate the system requirements.

The `InputQueue` example has illustrated why this step of the SLOOP method is not called a refinement step. It is better categorised as a **mapping** of problem domain classes onto solution domain classes, since it is not restricted to refinement only.

The other container classes in the call centre system are also mapped onto Smalltalk-80 library classes found in the SLOOP repository. The way in which one arrives at each of those mappings is similar to the procedures followed for the `InputQueue` class as discussed above. The table below summarises these mappings and their justifications.

Analysis level class	**Design level class**	**Justification for mapping**
`ConnectionContainer`	`Array`	The purpose of the `Connection= Container` class is to model the capacity constraints of the call centre system. It is used to indicate that the call centre can handle a maximum of `maxConn` simultaneous connections. At the same time it has to be guaranteed that a minimum of `maxConn` simultaneous connections can be processed. Property *AS2-01* captures these constraints by specifying that the the capacity of the connection container is **equal** to `maxConn`, where `maxConn` is a positive integer. The capacity is therefore a **fixed** value which may neither be decreased nor increased. Unlike the `OrderedCollection` class,

		which increases its capacity if a new element has to be added and it has already reached its full capacity, the capacity of an array is fixed upon instance creation. . The `Array` class is therefore the most appropriate mapping in this situation.
`ServiceCategory= Container`	`Array`	System property *AS2-07* specifies that the size (i.e. number of elements) of a service category container is equal to `maxCategories`, where the latter is a positive integer. The Smalltalk `Array` class is again the most appropriate, since the capacity of an `Array` instance is fixed.
`ServiceProvider= Container`	`Array`	System property *AS2-08* states that the size of the service provider container is equal to `maxSP`, where the latter is a positive integer. The `Service= ProviderContainer` class is mapped to the `Array` class because the capacity of an `Array` instance is fixed.
`InputQueue`	`Ordered= Collection`	As discussed above, the properties of the `InputQueue` class specify that the relative ordering of the elements of an `InputQueue` instance remains the same and that the positions of its elements are contiguous. Its capacity is non-decreasing (i.e. a minimum capacity is specified). The order in which elements are added to and removed from an input queue is the responsibility of its clients. The Smalltalk `OrderedCollection` class satisfies these requirements.
`ServiceQueue`	`Ordered= Collection`	The requirements are similar to those of the `InputQueue` class. Again, the Smalltalk `OrderedCollection` class is an appropriate mapping.
`ServiceProvider= Categories`	`Set`	As in the case of the `InputQueue` and `ServiceQueue` classes, most of the call centre system properties that refer to the `ServiceProviderCategories` instances deal with the behaviour of the elements of these instances. As far as the behaviour of the container class is concerned, there are no requirements for ordering. There are no correctness properties dealing with the capacity of the `ServiceProviderCategories` instances. This is because these instances are populated at start-up and there is no need to cater for dynamic insertion of elements. The Smalltalk `Set` class has the required functionality.

`ServiceProviderSubset`	`Set`	The justification for choosing the Smalltalk `Set` class as substitute for the `ServiceProviderSubset` class is similar to the one given for the `ServiceProviderCategories` class.

Table 6-1. Mappings of problem domain classes onto solution domain classes.

Figure 6-1 shows the class diagram resulting from the mappings in Table 6-1. The class diagram contains the reusable classes that represent initial mappings of analysis level classes onto design level classes. However, these mappings are not necessarily final, as will be evident from Section 6.7. The mapping of the `ServiceProviderSubset` class is revisited when the flexibility and extensibility of the design are considered. Reasons are given in that section for replacing the Smalltalk `Set` class with the Smalltalk `OrderedCollection` class in the `ServiceProviderSubset` mapping.

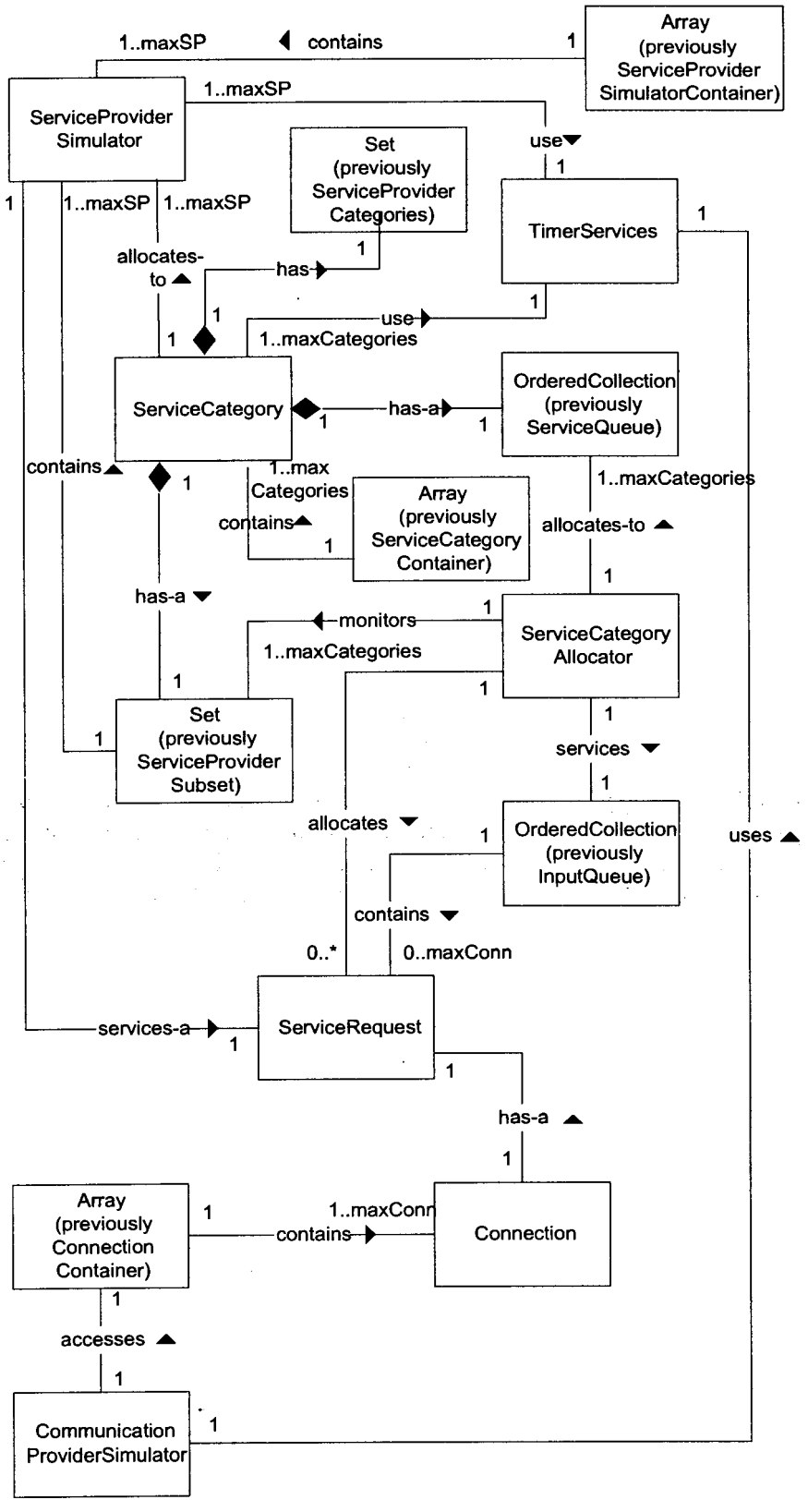

Figure 6-1. Mapping problem domain classes onto solution domain classes.

6.2.2 Formal versus informal specification of class properties

Another issue which is highlighted by the `InputQueue` example in the previous section is the problem of the formalisation of correctness properties at the class level. The problem relates to the fact that some of these properties might refer to aspects that are not represented by relationships between values of attributes of the class, but rather to aspects that are implied by method names or method arguments. The following property of the `OrderedCollection` class illustrates the problem:

AS4-03 (OrderedCollection). An object that is an element of anOrderedCollection remains an element unless the ***object*** *is specified and it is removed from anOrderedCollection or the* ***position*** *of the object is specified and the corresponding object is removed from anOrderedCollection.*

In this property there are references to the specification of an object or a position of an object. Typically, this information would be provided by the client via the argument of the relevant method. However, the argument of a method is not an attribute of the class, i.e. it is not a class or instance variable. The scope of the *pseudo-variable* representing the argument of a method is local to the method itself. It is therefore difficult to write a **class** level property such as *AS4-03(OrderedCollection)* in formal terms. However, the total correctness properties of the individual **methods** can easily be formalised. That is because the *pseudo-variables* are within the scope of the method and can therefore be referenced in the correctness properties of the method.

Similar problems occur when the correctness property of the class refers to aspects that are implied by method names. Again the `OrderedCollection` class serves as an example. It is implied by property *AS4-03* that an element can be selected for removal by specifying its position. However, this position can be implied by the name of the method. For example, the `removeFirst` method implies that the first element has to be removed. Again it is easy to indicate this requirement formally as part of the total correctness property of the `removeFirst` method. However, it is far more difficult to indicate formally in a class level property that clients may specify the position of removal via the name of a method.

It is for this reason that the SLOOP method requires the correctness properties of the individual methods to be specified both formally and informally, while formal specifications in the *properties-section* of the class are desirable, but not mandatory. Examples of formal specifications of method properties are given in Section 6.4.

6.2.3 Reusing existing classes through inheritance

In Chapter 5 the correctness properties of the two simulator classes were given. When searching for a matching class in the SLOOP repository, one first tries to find a reusable class which satisfies **all** the correctness properties of the given class. If such a reusable class is not found, it is still possible that the search might yield one or more classes that satisfy a **subset** of the given correctness properties.

For the purposes of this discussion it is assumed that the SLOOP repository does not contain classes that satisfy all the properties of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes. However, it is further assumed that the SLOOP repository does contain an `EventSimulator` class which satisfies the following correctness properties:

AL2-01(EventSimulator). When a simulation event is required, a simulation timer is eventually started .

AL2-02(EventSimulator). When a simulation timer expires, the simulation object eventually generates an event.

The `EventSimulator` class is an abstract class which provides the functionality to start a timer and to monitor it for its expiry. The conditions that result in an event being required and the actual event that is generated are unspecified, i.e. these aspects are the responsibility of the subclasses. Some additional design level properties regarding the random nature of the timers are also specified.

However, properties *AL2-01* and *AL2-02* of the `EventSimulator` class are the ones that draw the attention of the software designer, because they match properties *AL2-01* and *AL2-02* of the `CommsProviderSimulator` and the `ServiceProviderSimulator` classes. (As in the example in Section 6.2.1, it is unlikely that the properties of the new classes being specified and those of a class in the repository will match almost *verbatim*. The point that is made here, is that these properties are equivalent.)

By making the `CommsProviderSimulator` and `ServiceProviderSimulator` classes subclasses of the `EventSimulator` class, the functionality of the latter is reused. This has the advantage that the software designer only needs to redefine those methods that are left as responsibility of the subclasses.

6.2.4 Discovering suitable existing classes after design level refinements

During the analysis phase it was determined that a `TimerServices` class was necessary in order to provide the timer functionality of the system. While performing the **design level refinements** of the `TimerServices` class, one needs to decide how the timers should be represented and also how the clients of the TimerServices instance should be informed of the expiry of timers. In this example, each timer is represented by an instance of the `TimeoutElement` class. The instance variables (attributes) contain information about the timer, such as a reference to the requestor of the timer, an identifier which uniquely identifies the timer with respect to the requestor and the duration of the timer.

There are several ways in which timeouts can be indicated to the requestors of the timers. One option is to send a message to the requestor (i.e. invoke one of the requestor's methods) indicating that the requested timer has expired. Alternatively, the relevant `TimeoutElement` instance can be entered into a queue of expired timers. Each requestor inspects this queue on a regular basis and if it finds an element representing one of the requestor's timer requests, it removes it from the queue and takes the necessary action.

In this example, the second option is chosen, because it is a more loosely coupled solution. Consider the alternative where the `TimerServices` instance indicates a timeout event by invoking one of the requestor's methods. In that case the `TimerServices` instance is tied up until this object returns control to it if synchronous invocation [Vino97] is used. This is undesirable, because this object may choose to perform several actions as a result of the timeout. Since the timers should be as accurate as possible, the aim of the design is to reduce the influence of other objects on the `TimerServices` instance as much as possible.

As a result of the above design decision, **a new object is introduced**, viz. a queue of expired timers. The correctness properties of this queue are now determined in order to facilitate **comparison** with the classes in the SLOOP repository. The elements of the queue have to be ordered, since the requestors should be able to process their timeout events in the order of expiry. The elements are therefore always added to the end of the queue. However, elements are not necessarily removed from the queue in a FIFO order. For example, consider the case where a requestor is not inspecting the queue for a while (the requestor might be taken out of service

temporarily). When a timeout event for this particular requestor reaches the head of the queue, it will not be removed until the requestor becomes active again. This should not prevent the other requestors from obtaining their timeout information, therefore all requestors are allowed to inspect all elements of the queue and to remove them from any position in the queue.

The properties of the Smalltalk `OrderedCollection` class, first presented in Section 6.2.1, satisfy all of these requirements, therefore this class is reused for the queue of expired timers. This example illustrates how design level refinements could yield new classes, which could then be mapped onto **reusable** classes in the SLOOP repository.

6.3 Refining the specification

The aim of the previous section was to demonstrate how correctness properties are used during the **search for reusable artifacts** in the SLOOP repository. This section deals with the role played by correctness properties during **design level refinements**. It also discusses the impact of the computational model on design level decisions.

6.3.1 The role of correctness properties during design level refinements

As stated in Chapter 5, abstraction during the analysis phase means that design level details are not included. It does not mean that relevant **analysis level details** are omitted. Thus, although the deliverables for this phase may go through a series of refinements, the final refinement of the analysis phase reflects the **complete functionality** of the system. The '**what**' has to be fully specified, while the '**how**' is postponed to the design phase. For example, the first level of refinement might specify normal behaviour only. Error conditions are added in a subsequent refinement. Functionality that is nice to have, but not essential, could be added in yet another refinement. The final analysis phase refinement yields the deliverables for that phase.

During the design phase it is decided **how** the functionality specified during the analysis phase is going to be realised. Again, there may be multiple levels of refinement. In the call centre example the first level of refinement does not include any error conditions (the call centre may not reject a service request and the service user does not abort it). The functionality that is not essential is also not included, i.e. there are no progress timers and the service user is not informed of his/her progress towards being served. The `CommsProviderSimulator` and `ServiceProviderSimulator` classes are included in the design instead of the `CommunicationProvider`, `ServiceUser` and `ServiceProvider` classes.

Design detail is now added for this level of functionality. Two of the important **design** level goals are **reusability** and **flexibility**. The clean behaviour properties in Chapter 5, Section 5.4.1.2, refer to the number of connections, service request categories and service providers that need to be supported by the call centre system. By introducing a `Configuration` class to configure all of the above, the above goals are achieved, since it allows the capacity of the system to differ from application to application, without having to redesign anything for the various applications.

The introduction of a `Configuration` class may not violate any of the properties specified for the system. Part of the behaviour of the `Configuration` class is derived by inspecting the list of system properties. Further design level refinements provide the basis of the remaining part of its behaviour. Listed below are the system properties relevant to the `Configuration` class. Note that these are not properties of the `Configuration` class. They are relevant to the `Configuration` class, because they enable the software designer to determine which aspects of the call centre system should be configurable and what the properties of these configurable items should be.

*AS2-01. The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer.*

AS2-07. The size of the service category container is equal to maxCategories, where maxCategories is a positive integer.

AS2-08. The size of the service provider container is equal to maxSP, where maxSP is a positive integer.

AS3-02. The category of each service request in a service queue is the same as the service category associated with the service queue.

AS3-03. Each service category is unique.

AS3-08. The category of each element of the service provider subset has to match one of the elements of the ServiceProviderCategories instance associated with the service category.

The following design level clean behaviour property results from properties *AS2-01*, *AS2-07* and *AS2-08*:

DS2-01 (Configuration). The values of each of the maximumConnections, maximum= ServiceCategories and maximumServiceProviders[1] *instance variables are invariant and always greater than zero.*

Thus, once the `Configuration` class has been instantiated, the values of the above-mentioned instance variables will be greater than zero and remain constant. The `Configuration` class therefore has to ensure that values greater than zero are assigned to these attributes during system initialization and thereafter the values may be queried, but not modified.

Property *AS3-02* implies that each service request has a category associated with it and this category has to match the category associated with the service queue that it is assigned to. Since these service category names are likely to differ from application to application, it is appropriate to make them configurable items. The `Configuration` class also has to ensure that property *AS3-03* is preserved, i.e. the service category names that are configured for the call centre system have to be unique. The following properties describe the responsibilities of the `Configuration` class regarding the service category names.

DS3-01 (Configuration). The number of service request category names that are configured is equal to maximumServiceCategories.

DS3-02 (Configuration). Each configured service request category name is unique.

Property *AS3-08* implies that a set containing one or more service provider categories is associated with a service category. Again it would be prudent to make the service provider category names configurable items. The `Configuration` class therefore has the following property:

DS3-03 (Configuration). At least one service provider category name is configured.

[1] The names of these instance variables differ from the names used in the system properties. This is because the system properties cannot refer directly to the attributes of one of its classes; it always has to use the methods provided by that class to obtain the values of those attributes. In this example, the maxConn, maxCategories and maxSP macros defined at the system level represent the messages sent to the Configuration instance to obtain the values of the maximumConnections, maximumServiceCategories and maximumServiceProviders attributes respectively.

Note that there is no requirement in the problem statement that each service provider should belong to a unique service provider category. The number of service provider categories that are configured therefore has to be greater than zero, but it does not have to be equal to `maximumServiceProviders`.

The above discussion shows how the `Configuration` class is added during the design phase in order to produce a **flexible** and **reusable** implementation. Although no management functionality is required of the system (based on the the outcome of the analysis phase), this does not preclude a design which makes provision for such functionality, provided the additional functionality does not violate any of the system properties.

When the behaviour of the `Configuration` class is determined, the first step is to inspect the analysis level correctness properties. Subsequently, design level aspects are also considered. In particular, when design level detail is added to the `TimerServices` class, it becomes evident that a maximum allowable timeout value needs to be specified.

This is because the timeout values are represented by positions in a **circular** array[2] and clearly such an array is bounded. Correctness property *DS2-01 (Configuration)* is therefore extended to read as follows:

DS2-01 (Configuration). The values of each of the maximumConnections, maximum= ServiceCategories, maximumServiceProviders and maximumAllowableTimeout instance variables are invariant and always greater than zero.

The `Configuration` class example has served to illustrate how design level refinements can result in the specification of an additional class. In turn, the behaviour of the additional class is determined from both analysis level and design level requirements of the system.

6.3.2 The impact of the computational model on design level decisions

The next topic deals with the impact of the computational model on design level considerations. In the previous chapter it was stated that the purpose of the `ConnectionContainer` class was to model the **capacity constraints** of the call centre. Thus, this class is used to indicate that the call centre can handle a maximum of `maxConn` simultaneous connections. At the same time it also guarantees that a minimum of `maxConn` simultaneous connections can be processed. Property *AS2-01* captures these constraints as follows:

*AS2-01. The capacity of the connection container is **equal** to maxConn, where maxConn is a positive integer.*

The above analysis level property does not prescribe whether the elements of the connection container (i.e. the connections) should always be present or whether they should be added and removed as required. It therefore allows for multiple design possibilities regarding the way in which the establishment and termination of connections can be modelled.

One design possibility is to create a `Connection` instance and enter it into the connection container whenever a new connection is established and to destroy it when the connection is

[2] Each position in the array represents a time unit. The TimerServices instance maintains an index into the array. This index is advanced every time unit, taking into account that it is a circular array. Each entry in the array is an ordered collection of TimeoutElement instances. The collection identified by the current value of the index represents timers that have expired during the last time unit. When a new timer needs to be started, a new instance of TimeoutElement is created and it is appended to the collection that will be reached after x time units, where x is the timeout value specified for the new timer. Thus, the appropriate collection is identified by using the value of the index into the array, the size of the array and the requested timeout value in the calculation. For a detailed description of the design of the TimerServices class the interested reader is referred to Section B.11 of Appendix B.

terminated. Another option is to create the maximum number of `Connection` instances upon startup and to use an instance variable (attribute) of the `Connection` instance to model whether a connection is established or terminated.

The SLOOP computational model influences this decision at the design level. When using the SLOOP method, the design is in terms of statements that execute infinitely often. If parallel statements are used to represent the actions of a `Connection` instance, then such an instance has to be present all the time, otherwise one cannot reason in terms of statements that execute infinitely often[3]. For this reason, the design decision is taken to create the `Connection` instances upon startup and to represent the state of the connection via an instance variable.

The SLOOP computational model has another important effect on the design level refinements of a system, viz. the fact that the concept of blocking statements does not form part of a SLOOP design.

For example, the `wait` method of the `Delay` Smalltalk-80 library class causes the active process to be suspended for a specified period. A statement which includes such a message would therefore block for that period. A SLOOP design postpones decisions about assignment of statements to processes to a much later stage of the software development. There is therefore no concept of a suspended process. For this reason the Smalltalk-80 library methods that cause the active process to be suspended, are not used during the design phase.

It is now shown how system services such as timers can be designed within the paradigm of statements that are all executed infinitely often and without using these blocking messages.

One way of implementing timer services using conventional methods is to create a timer services object which suspends itself for a specified period. The value of this period depends on the granularity of the timeouts. Whenever the object resumes execution, it indicates that a time unit has expired by advancing the index into the circular array (the positions in this array represent the timeout values, as described in the previous section). The list of `TimeoutElement` instances at the new position in the array represent the timers that have expired as a result of the expiry of the last time unit. The object then informs the relevant objects. It repeats the cycle of suspending itself and informing other objects of expired timers *ad infinitum*.

In order to make the timeouts more accurate, the above functionality could be distributed over a number of objects, i.e. the object which suspends itself could merely inform one or more other objects that the specified period has elapsed before suspending itself again. The other auxilliary objects could then be responsible for determining whether any timers have expired and inform the relevant objects. This approach would be appropriate if many timers expired simultaneously, resulting in a situation where the time spent between suspended states was no longer negligible.

In the SLOOP environment there is no concept of an object which suspends itself for a specified period. Instead, the `TimerServices` instance contains a parallel method which checks whether the specified period has elapsed and if it has, it advances the index into the circular array and invokes the methods that check whether any timers have expired.

The relevant parts of the parallel statements of the `p_runTimer:` method are shown below (upon instance creation, both `currentTime` and `lastTime` are set to the same value):

```
currentTime := SmalltalkLibPkg:::Time now asSeconds
[] lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 "... and other boolean conditions..."
```

[3] There are exceptions to this rule, as will be discussed in Chapter 9, Section 9.3.2.

The value of `difference` is calculated by evaluating the following *macro-expression*:

```
difference ≡ currentTime - lastTime
                 if (currentTime - lastTime) ≥ 0 ~
               currentTime + (86400 - lastTime)
                 if (currentTime - lastTime) < 0
```

Thus, whenever the statement

```
currentTime := SmalltalkLibPkg:::Time now asSeconds
```

is scheduled for execution, the `currentTime` variable is updated with the latest value of `Time now asSeconds`. The latter provides the number of seconds that have elapsed since midnight.

At some point the other parallel statement, viz.

```
lastTime := currentTime \+
currentTick := (currentTick + 1) \\ (timeoutCollection size)
      if difference ≥ 1 "... and other boolean conditions..."
```

is executed. It calculates whether more than one second has elapsed since the variable `lastTime` has been updated. If it has, the variable `lastTime` is updated with the current value of `currentTime`. The calculation of `difference` takes the rollover at midnight into account. If more than one second has elapsed, the index into the circular array is advanced and the parallel statement that checks whether any timeouts have occurred becomes effective. (This statement is not shown here, but is given in Appendix B, Section B.11. That section contains a detailed description of the internal design of the `TimerServices` class.)

It is clear from the above that SLOOP statements have sufficient expressive power even though blocking messages are not allowed.

6.4 Formalising correctness properties

Once the classes for a particular level of refinement have been identified, it has to be ensured that all the required class and instance methods are defined. At the **method** level it is mandatory to specify the correctness properties formally. This has the advantage of providing an **unambiguous** and **concise** description of the behaviour of the method. The total correctness properties of the `configure` method of the `Configuration` class is now shown as an example. (The purpose of the variables listed below that have not been discussed before, will be given shortly.)

```
"Total correctness property"
"Upon completion of the configure method, the
maximumConnections, maximumServiceCategories, maximumService=
Providers and maximumAllowableTimeout instance variables will
each have a value greater than zero, the srCategoryNames and
spCategoryNames collections will have been created, the number
of elements in the srCategoryNames collection will be equal to
maximumServiceCategories, there will be at least one element in
the spCategoryNames collection, the srToSpCategoryMap will have
been created, the number of mappings in this collection will be
equal to maximumServiceCategories and the categoriesAssigned
variable will have the value zero."
<∀ ( t, u, v, w) where
t > 0 ∧ u > 0 ∧ v > 0 ∧ w > 0 ::
true results-in
      maximumConnections = t ∧
      maximumServiceCategories = u ∧
      maximumServiceProviders = v ∧
```

```
        maximumAllowableTimeout = w ∧
        srCategoryNames notNil ∧ spCategoryNames notNil ∧
        srCategoryNames size = u ∧
        ¬ spCategoryNames isEmpty ∧
        srToSpCategoryMap notNil ∧
        srToSpCategoryMap size = u ∧
        categoriesAssigned = 0
>                                        "DL1-02 (Configuration)"
```

When the properties of a method are specified, one always has to guard against **overspecification**. In this particular example, default values are assigned to all the instance variables. However, the actual values of some of these default values are not presented in the total correctness property, since that would prevent subclasses from assigning other values to those variables. For example, the value of the instance variable `maximumConnections` is set to 8 in the SLOOP statements of this method, while the correctness property merely specifies that it is set to any value `t`, where `t` is greater than zero. This ensures that subclasses can set it to different hard-coded values, or even obtain the values from other sources, while still adhering to the above specification.

The `Configuration` class is a composite class and when it is instantiated, it creates the `srCategoryNames` and `spCategoryNames` objects. These are collections that contain the configured service request and service provider category names respectively. Note that the `srCategoryNames` collection has to contain the maximum number of service category names (indicated by `srCategoryNames size = u`, where `u = maximumServiceCategories`), whereas `spCategoryNames` only has to be non-empty. This is because each service category has to be unique, whereas multiple service providers may belong to the same service provider category.

The `Configuration` class also creates the `srToSpCategoryMap` object. This is a table which maps a service request category name to a collection of service provider category names. This table is interrogated when a `ServiceCategory` instance is created. At that point the `ServiceCategory` instance has to record which service provider categories will be associated with it. The `categoriesAssigned` instance variable is used to keep track of the number of service request categories that have already been assigned to `ServiceCategory` instances. It ensures that each service request category is assigned only once, thereby ensuring the uniqueness of the service request categories.

When the informal and formal versions of the above property are compared, the **concise** and **exact** nature of the formal version becomes apparent. Furthermore, it should be noted that the correctness properties of a method have to contain **all the information** that is required in order to enable the software designer to understand the **impact** of the execution of this method on the **state of the object**. The `configure` method initialises all the instance variables of the Configuration object and this has to be reflected in the correctness property. The ultimate aim of the specification of the method properties is that the software designer should be able to obtain an **exact understanding** of the behaviour of a method without having to study a single program statement.

6.5 Deriving SLOOP statements

Once the classes comprising a system have been identified and correctness properties have been specified for each class, both at the analysis level and the design level, the methods required for each class need to be identified. The class properties are inspected to determine what methods

need to be created. The next step is to specify the correctness properties for each method. Thereafter the SLOOP statements for the individual methods are derived.

The main purpose of this section is to provide an example of how the statements of SLOOP classes are derived. The `ServiceProviderSimulator` class, which is used in this example, also provides a convenient vehicle for highlighting some other issues, viz.

- It is an example of a class which provides its required functionality by simply **complementing** the parallel statements inherited from its superclass with its own parallel statements.
- If there needs to be a relationship at all between the parallel statements of a class and its superclass then that relationship could be via variables rather than via parallel methods that need to be overridden. It is explained why such a design maximises **flexibility** and **extensibility**.
- Criteria for dividing the functionality of a class between its parallel and sequential methods are described.
- The rationale for grouping certain parallel statements into a single rather than into multiple parallel methods is discussed.
- The advantages of using abstract preconditions in SLOOP methods are described.

The `ServiceProviderSimulator` class and its superclass, the `EventSimulator` class, are used as the examples in this section. SLOOP statements are derived for one sequential and one parallel method for each of these classes. However, in order to place the functionality of these methods in context, brief descriptions of the functionality of all the other methods identified for these two classes are required. Summaries of the methods of the `EventSimulator` and `ServiceProviderSimulator` classes are therefore presented in Tables 6-2 and 6-3 respectively[4]. It is beyond the scope of the present discussion to show how these methods were arrived at; only the results of the design phase refinements are shown here.

Note that **abnormal conditions**, such as the user aborting a service request and a service provider simulator going out of service, **are not included at this level of refinement**.

Thus, the abstract `EventSimulator` class provides the common functionality shared by many different types of simulators. It is driven by its parallel method, `p_simulate:timeoutEventsIn:`. The latter sets the `newEventRequired` instance variable to false and invokes `startRandomTimer: withMaximum:`, a sequential `EventSimulator` method, if the `newEventRequired` instance variable is true when the statement starts executing. The `startRandomTimer:withMaximum:` method invokes the relevant `TimerServices` instance method to start a timer.

Message pattern	Discussion
`initialize`	The `initialize` method initializes the instance variables defined for the `Event= Simulator` class.
`startRandomTimer: aTimerServices withMaximum: maximumValue`	This method requests `aTimerServices` to start a timer. The value requested is a random value between 1 and `maximumValue`. (The `startRandom= Timer:withMaximum:` method invokes the `nextRandomNumber:` method to obtain the next random number.)

[4] The detailed SLOOP specifications of these methods are given in Appendix B, Sections B.5 and B.13 respectively.

`nextRandomNumber: maximumValue`	This method returns the next random number within the specified range of 1 to `maximumValue`.
`timerExpired: timerEventQ`	The timerExpired: method returns true if `timerEventQ` contains `aTimeoutElement` representing the expiry of a timer requested by the receiver, otherwise it returns false.
`resetTimerExpired: timerEventQ`	This method removes the `timerEventQ` element representing the expiry of a timer requested by the receiver.
`p_simulate: aTimerServices timeoutEventsIn: timerEventQ`	If `newEventRequired` is true, it invokes the `startRandomTimer:withMaximum:` method and sets `newEventRequired` to false. If the `timerExpired` method returns true, it invokes the `resetTimerExpired:` method and it sets `generatingEvent` to true.

Table 6-2. Methods of the `EventSimulator` class after design phase refinements.

It is the responsibility of the subclass to set the `newEventRequired` instance variable to true. The conditions that result in a new event being required are provided by the subclass, as seen in the `processServiceRequest:` method in Table 6-3 below.

The `p_simulate:timeoutEventsIn:` parallel method also monitors the `timerEventQ` and sets the `generatingEvent` instance variable to true if a timer requested by the receiver has expired and an event therefore has to be generated. The actual event that is generated is again the responsibility of the subclass. The subclass adds its own parallel method to monitor the value of the `generatingEvent` instance variable. This allows for **maximum flexibility** regarding the type of event that is generated by the subclass, since there is no `EventSimulator` method which needs to be overridden by the subclass. The superclass therefore does not even prescribe the name of the method which generates the event, nor does it prescribe the number of arguments that should be passed to it. This aspect of the design is illustrated by the code fragments given below and is discussed further at the end of this section.

The methods of the `ServiceProviderSimulator` are now listed:

Method	**Discussion**
`startSimulation: scContainer using: aConfiguration`	This method creates a `ServiceProvider=Simulator` instance and invokes the `moreInit:using:` method, which performs initialization that is additional to that executed in the `initialize` method of the superclass.
`moreInit: scContainer using: aConfiguration`	The `moreInit:using:` method initializes the instance variables defined in the `Service=ProviderSimulator` class. It also registers the `ServiceProviderSimulator` instance with all the service categories that are serviced by service providers of this category.
`serviceProviderCategory`	This method returns the category of the receiver.
`serviceRequest`	The `serviceRequest` method returns the service request currently being serviced.

`p_generateEvent`	If `generatingEvent` is true, it initiates the termination of the connection associated with the service request, sets `serviceRequest` to nil and sets `generatingEvent` to false.
`registerServiceProvider: scContainer using: aConfiguration`	The simulator registers itself with each service category that requires service from service providers of this service provider category. The simulator keeps a record of all the service categories that it registers itself with. The resulting collection is used to ensure that the simulator does not ignore any of these service categories for ever. This is achieved by maintaining an index into this collection. The index is updated in such a way that the service categories are serviced in a round robin fashion by this simulator.
`processServiceRequest: aServiceRequest`	This method ensures that `newEventRequired` is set to true when a service request is assigned to the simulator. It sets `serviceRequest` to `aService= Request` and it also updates the current index into the collection of service categories being serviced by this simulator.
`canAcceptNextSR: requestingServiceCatgory`	The `canAcceptNextSR:` method returns false if the receiver is busy servicing a request or if the `requestingServiceCategory` does not match the service category that should be serviced next by this simulator, otherwise it returns true.
`p_updateCategoryIndex: scContainer`	This method updates the current index into the collection of service categories serviced by this simulator if the service queue associated with the current service category is empty. This ensures that an empty service queue will not prevent the simulator from servicing the service queues of other service categories.

Table 6-3. Methods of the `ServiceProviderSimulator` class after design phase refinements.

The next step is to **generate the SLOOP statements** of each method. In this example, the SLOOP statements of one sequential and one parallel method from each of the `EventSimulator` and `ServiceProviderSimulator` classes are presented. The interested reader is referred to Appendix B, Sections B.5 and B.13 respectively, for full specifications of these classes. In order to provide the necessary contextual information for these methods, all the class and instance variables, as well as all the class properties are included below.

Note that some of the properties, such as *AL2-01* and *AL2-02*, are only specified informally at the class level. This is because they refer to *pseudo-variables*, a topic which was discussed in Section 6.2.2. However, at the method level all properties are specified formally.

```
class EventSimulator
superclass Object from SmalltalkLibRepository

instance variable names
rand
      "This variable refers to an instance of the Random class from
      the Smalltalk library.  The instance is created when the
```

EventSimulator subclass is instantiated. The instance of the Random class maintains a seed from which the next random number is generated. The random number is used to start a timer with a random value."

newEventRequired

"When the value is equal to true it means that a new event is **required**. Once the variable has been set to true, a random timer will be started at some point afterwards. When the timer is started, newEventRequired is set to false. It is the responsibility of the subclass to set this variable to true when a new event is required, since each subclass will have its own conditions for requiring a new event. Once the timer expires, an event will be generated, as will be described in the comments section of the **generatingEvent** variable."

currentRandomTimeoutValue

"This variable contains the value of the random timeout currently being requested. The purpose of this variable is to provide a mechanism for referencing the current timeout value in the correctness arguments. Note that the SLOOP statements could therefore have been rewritten without this variable while still providing the same functionality. However, in that case it would not have been possible to formalise certain correctness properties (such as DL1-04)."

generatingEvent

"The value is equal to true if the timer has expired and an event has to be **generated**, otherwise it is equal to false. The subclass sets this variable to false at the time when the event is generated. The actual event that is generated is also the reponsibility of the subclass, since each subclass will generate a different type of event."

timerOutstanding

"This variable is set to true when a timer is started and it is set to false when a timeoutElement is removed from the timerEventQ (i.e. when an expired timer has been processed). The purpose of this variable is to provide a mechanism for reasoning about the uniqueness of outstanding timers in the EventSimulator class. In this class only one timer requested by the EventSimulator may be outstanding at a time. The timerOutstanding variable is used in the preconditions of the startRandomTimer:withMaximum: method as well as in the postconditions of the resetTimerExpired: method. If subclasses need to support multiple simultaneous timers, then the preconditions of the startRandomTimer:withMaximum: method need to be weakened and the postconditions of the resetTimerExpired: method need to be strengthened. Since the purpose of the timerOutstanding variable is to facilitate correctness reasoning, the SLOOP statements could have been rewritten without this variable while still providing the same functionality."

timerId

"This variable contains the identifier of the timer currently being requested.

class properties

"Liveness"

"When a simulation event is required, a simulation timer is eventually started."

"AL2-01"

"Liveness"

"If a simulator timer expires, the simulator eventually has to generate an event."

"AL2-02"

"Clean behaviour"
<∀ anObject ::
invariant anObject class ~~ EventSimulator
> **"DS2-01"**
"The EventSimulator class is an abstract class and should not be instantiated"

"Clean behaviour"
invariant rand notNil ∧ rand class = Random **"DS2-02"**
"Once rand has been initialized to refer to an instance of the Random class, it is never set to nil while the instance of the EventSimulator subclass exists. "
"It is therefore possible for the `EventSimulator` subclass instance to send messages to `rand` at any stage after initialization."

"Clean behaviour"
"The currentRandomTimeout value is always within the range specified by the precondition of the start:id:for: method of the TimerServices class."
"DS2-03"

"Global invariant"
"All outstanding timers requested by an EventSimulator subclass instance are identified uniquely with respect to the requestor."
"DS3-01"
"Thus, all the timers requested by this requestor that are currently running or that are in the timerEventQ are uniquely identified with respect to the requestor."

instance methods

category modifying
message pattern startRandomTimer: aTimerServices
withMaximum: maximumValue
"Start a timer with a random value within the range between 1 and maximumValue. When the resulting start:id:for: message is sent to the TimerServices instance, a reference to the requestor (in this case the EventSimulator subclass instance) as well as an identifier are passed as parameters. The combination of the reference to the requestor and the identifier ensures that each timer request can be identified uniquely within the system. This facilitates the correlation of the subsequent timeout notifications with the timer requests.

In the EventSimulator class only one timer is outstanding at a time for a specific requestor, i.e. ¬timerOutstanding is a precondition for starting a new timer for a specific instance of an EventSimulator subclass. Since the timers initiated by a specific EventSimulator subclass instance do not run concurrently, these timers can all have an identifier of 1.

If a subclass requires multiple concurrent timers, unique values must be allocated to the corresponding identifiers. The startRandomTimer:withMaximum: method therefore needs to be overridden in order to achieve this. The total correctness property of the modified method also needs to be updated, viz. a **disjunction** needs to be added to the precondition to state that the proposed identifier of any new timer requested by that EventSimulator subclass instance should not match any identifier of any other outstanding timer requested by that EventSimulator subclass instance. Thus, the precondition has to be **weakened.**

```
        In that case the value of timerOutstanding will no longer be
        relevant."

        method properties
        "Total correctness"
        ¬timerOutstanding results-in methodReturnValue = self ∧
              self postconditions: (#nextRandomNumber:)
              withArguments: #(maximumValue) ∧
              aTimerServices postconditions: (#start:id:for:)
              withArguments: #(self timerId currentRandomTimeoutValue) ∧
              timerOutstanding                                    "DL1-04"
        sequential
        currentRandomTimeoutValue :=
              (self nextRandomNumber: maximumValue)
        [] timerId := 1
        [] aTimerServices start: self id: timerId for:
              currentRandomTimeoutValue
        [] timerOutstanding := true
        end-sequential

category cyclic
        message pattern p_simulate: aTimerServices timeoutEventsIn:
        timerEventQ
        "If a new event is required, start a random timer, the expiry of
        which will cause an event to be initiated."
        method properties
        "This method ensures that properties DS2-03, AL2-01 and AL2-02
        are satisfied by the EventSimulator class."

        "Clean behaviour"
        invariant   currentRandomTimeoutValue > 0 ∧
        currentRandomTimeoutValue ≤ aTimerServices maximumTimeout
                                                                    "DS2-03"
        "A timeout requested by the EventSimulator subclass instance is always within the range
        that ensures that the precondition of the start:id:for: method of the TimerServices class
        is met when the EventSimulator subclass instance invokes that method."

        "Precedence"
        newEventRequired ensures
              self postconditions: (#startRandomTimer:withMaximum:)
              withArguments:
              #(aTimerServices (aTimerServices maximumTimeout))
              ∧ ¬newEventRequired                                   "DP1-01"
        "When newEventRequired is true, it ensures that a simulation timer is started and
        newEventRequired becomes false."

        "Precedence"
        self timerExpired: timerEventQ ensures
              generatingEvent ∧
              self postconditions: (#resetTimerExpired:)
              withArguments: #(timerEventQ)                         "DP1-02"
        "When a simulation timer expires, it ensures that generatingEvent becomes true."

        parallel
        self startRandomTimer: aTimerServices withMaximum:
        (aTimerServices maximumTimeout) \+
        newEventRequired := false
              if newEventRequired
```

```
        [] generatingEvent := true \+
        self resetTimerExpired: timerEventQ
              if self timerExpired: timerEventQ
        end-parallel
```

The `p_simulate: timeoutEventsIn:` method demonstrates very clearly how simple the derivation of the parallel statements is once the correctness properties of the method have been specified. In this case the first and second parallel statements correspond with properties *DP1-01* and *DP1-02* respectively.

The above two methods provide an example of **how the functionality of a class is divided amongst its parallel and sequential methods**. If the class has to react to events, then one or more parallel method(s) are used to monitor those events. The corresponding actions are usually captured in sequential methods. In the above example, either a change to the value of the `newEventRequired` variable or the expiry of a timer constitutes an event. These changes are monitored in the *if* clauses of the parallel statements.

The above example highlights another design issue, viz. the **structuring of parallel statements**. The parallel statements of the `EventSimulator` class are grouped into a single `p_simulate:timeoutEventsIn:` parallel method. Alternatively, each parallel statement could have been encapsulated in a separate parallel method. A single parallel method containing both statements has the advantage that fewer parallel methods need to be activated.

It also ensures that whenever the parallel statement which **starts** a timer is activated, then the corresponding statement which monitors the **expiry** of this timer is also activated. Thus, whenever there is a **dependency** between two parallel statements of the same class, in the sense that if the one is used in a program, then the other should also be used, then it is prudent to group those statements into a single parallel method. That ensures that one of the statements will not be omitted accidentally when the software designer of a new system decides to reuse this functionality[5].

The `ServiceProviderSimulator` class is presented next. The `newEventRequired` and `generatingEvent` instance variables inherited from its superclass are now interpreted with respect to the functionality of the `ServiceProviderSimulator` class. The timer that is started when a new event is required, represents the time it takes to service a service request. When the `generatingEvent` instance variable is set to true, it implies that the service has been completed and the connection should be terminated.

```
class ServiceProviderSimulator
superclass EventSimulator from ApplicationsRepository

instance variable names
serviceRequest
        "This variable refers to the service request currently being
        serviced by the service provider simulator.  Note that the
        reference to the ServiceRequest instance is passed to the
        simulator as a parameter, i.e. the ServiceRequest instance is
        not created by the ServiceProviderSimulator instance and
        therefore does not form part of it."
serviceProviderCategory
        "This variable contains the name of the service provider
        category to which the service provider simulator belongs."
```

[5] Recall that all the parallel methods defined for a class need not be activated when the class is reused. This feature of the SLOOP method was described in Chapter 4, Section 4.3.1.

categoriesServed
"This is an ordered collection containing the names of the service request categories serviced by this service provider. The purpose of this array is to facilitate a round robin servicing scheme of these categories. That prevents starvation of a specific service category."
nrOfCategoriesServed
"This variable contains the number of service request categories serviced by this service provider. It is used in the calculation when the categoryIndex is updated."
categoryIndex
"This variable is used as index into the categoriesServed collection. It is used to determine the next service request category to be serviced by this service provider. It is incremented modulo nrOfCategoriesServed. Its values range from 0 to nrOfCategoriesServed - 1"

class properties

<∀ categoryIndex **where** categoryIndex ≥ 0 ∧
categoryIndex ≤ nrCategoriesServed - 1 ::
invariant serviceRequest notNil ⇒ ¬self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1))
> **"AS3-01 (ServiceProviderSimulator)"**
"A service provider simulator services a single service request at a time."
"If a service request is currently assigned to the simulator, no other service request from any of the categories being served by this simulator will be served by the latter."

serviceRequest isNil ∧ ¬newEventRequired **unless**
serviceRequest notNil ∧ newEventRequired
"AS4-01 (ServiceProviderSimulator)"
"When a new service request is assigned to the service provider simulator then a new service provider simulator event is required."

"Note: The parent class, viz. EventSimulator, contains a parallel method which monitors the value of newEventRequired. If it detects that newEventRequired is true, it starts a timer and sets newEventRequired to false."

serviceRequest notNil ∧ ¬newEventRequired **unless**
serviceRequest isNil ∧ ¬newEventRequired
"AS4-02 (ServiceProviderSimulator)"
"If a service request has been assigned to the service provider simulator and newEventRequired is false, then newEventRequired remains false while the service request is still assigned to the service provider simulator."
"This has the effect that this simulator will not start another timer before the servicing of the current service request has been completed."

generatingEvent ∧ serviceRequest notNil **ensures**
(serviceRequest connection) postconditions: (#terminate:)
withArguments: #('completed') ∧ serviceRequest isNil ∧
¬generatingEvent **"AP1-01 (ServiceProviderSimulator)"**
"If a service provider simulator has to generate an event, it ensures that the service provider simulator terminates the connection currently associated with it and becomes available to service a new service request."

"Note: The parent class, viz. EventSimulator, contains a parallel method which sets generatingEvent to true when a timer has expired."

```
<∀ aServiceRequest where serviceRequest = aServiceRequest ::
 serviceRequest = aServiceRequest ensures
      (serviceRequest connection) postconditions: (#terminate:)
      withArguments: #('completed') ∧ serviceRequest isNil
>                          "AP1-02 (ServiceProviderSimulator)"
```

"A service request remains assigned to a service provider simulator until the latter completes the service and terminates the connection."

```
invariant categoryIndex ≥ 0 ∧
      categoryIndex < nrOfCategoriesServed
                           "DS2-01 (ServiceProviderSimulator)"
```

"The categoryIndex is always greater than or equal to zero and less than nrOfCategoriesServed."

```
"Clean behaviour"
invariant    categoriesServed notNil ∧
             categoriesServed class = OrderedCollection
                           "DS2-02 (ServiceProviderSimulator)"
```

"Once categoriesServed has been initialized to refer to an instance of the OrderedCollection class, it is never set to nil while the ServiceProviderSimulator instance exists."

```
<∀ categoryIndex where 0 ≤ categoryIndex ∧
categoryIndex < nrOfCategoriesServed ::
¬(self canAcceptNextSR:
(categoriesServed at: (categoryIndex + 1)))
leads-to
      self canAcceptNextSR:
      (categoriesServed at: (categoryIndex + 1))
>                          "DL2-01 (ServiceProviderSimulator)"
```

"For any service category serviced by the service provider simulator, the service provider simulator will eventually be able to service a request from that service category."

```
instance methods
category modifying
message pattern processServiceRequest: aServiceRequest
      method properties
      "A new simulation is required each time when a new service
      request is processed."
      "Total correctness"
      <∀ x where 0 ≤ x ∧ x < nrOfCategoriesServed ::
      categoryIndex = x ∧
      aServiceRequest notNil ∧
      self canAcceptNextSR:(aServiceRequest serviceRequestCategory)
      results-in
            methodReturnValue = self ∧
            serviceRequest = aServiceRequest ∧
            newEventRequired ∧
            categoryIndex = (x + 1) \\ nrOfCategoriesServed
      >                          "DL1-06 (ServiceProviderSimulator)"
      sequential
        newEventRequired := true
      [] serviceRequest := aServiceRequest
```

```
        [] categoryIndex := (categoryIndex + 1) \\ nrOfCategoriesServed
        end-sequential

category cyclic
        message pattern p_generateEvent
        method properties
        "The newEventRequired attribute is not updated here.  It is only
        set to true once a new service request has been received."

        generatingEvent ∧ serviceRequest notNil ensures
                (serviceRequest connection) postconditions: (#terminate:)
                withArguments: #('completed') ∧
                serviceRequest isNil ∧ ¬generatingEvent
                                "DP1-01 (ServiceProviderSimulator)"
        "If a service provider simulator has to generate an event, it
        ensures that the connection currently associated with the
        service request is terminated and that the service provider
        simulator becomes available to service a new service request."
        parallel
        (serviceRequest connection) terminate: 'completed' \+
        serviceRequest := nil \+
        generatingEvent := false
                if generatingEvent
        end-parallel
```

Correctness property *AS3-09* of the system specifies that a service provider / service provider simulator services a single service request at a time. Thus, a new service request should only be assigned to a service provider if it has finished servicing the previous one, i.e. the `processServiceRequest:` sequential method shown above should only be invoked if the `serviceRequest` attribute of the simulator is nil. However, in order to make provision for other conditions that may also play a role when deciding whether a service request can be assigned to a service provider simulator, clients do not interrogate the `serviceRequest` attribute of the service provider simulator, but rather invoke the `canAcceptNextSR:` method. The `canAcceptNextSR:` method facilitates the implementation of an abstract precondition, a concept described in [Meye97].

An abstract precondition is a construct which allows one to strenghten preconditions, without requiring the clients of the target class having to modify their code. Thus, the preconditions of a method are encapsulated in the postconditions of another method (in this example, the `canAcceptNextSR:` method). The client has to ensure that the latter returns true before invoking the `processServiceRequest:` method.

In the above example, the `canAcceptNextSR:` method returns true if the `serviceRequest` instance variable is equal to nil and the service category passed as parameter matches the next category to be served by this `ServiceProviderSimulator` instance. However, in subclasses of the `ServiceProviderSimulator` class, the postconditions of the `canAcceptNextSR:` method might be strenghtened. In this manner the clients of the `processServiceRequest:` method can be sure that the preconditions of the `processServiceRequest:` method will be satisified if they invoke it only if `canAcceptNextSR:` returns true. This remains true even if the **preconditions** of the `processServiceRequest:` method are strengthened (via the strengthening of the **postconditions** of the `canAcceptNextSR:` method). The rationale for including abstract preconditions in a design is to maximise the **extensibility** of the classes. This is an important design level consideration.

The `EventSimulator` and `ServiceProviderSimulator` examples also illustrate how the concept of parallel methods enables one to handle **inheritance** very elegantly in the SLOOP method. In this example the parent class merely sets the `generatingEvent` instance variable to true in the `p_simulate:timeoutEventsIn:` method. It does not invoke the `p_generateEvent` method. This enables the subclass to define any parallel method to act upon the setting of the `generatingEvent` instance variable. For example, the `ServiceProviderSimulator` subclass defines the `p_generateEvent` method, whereas the `CommunicationProviderSimulator` subclass defines the `p_generateEvent: target:` method. (The latter is described in detail in Appendix B, Section B.6.) This approach is possible because the parallel methods are executed infinitely often. The required action will eventually be performed, provided the `generatingEvent` variable is eventually set to the value true and all the necessary parallel methods are activated.

The alternative is to define a `p_generateEvent` method for the `EventSimulator` class and to specify that it is the responsibility of the subclass to implement it. The `p_simulate: timeoutEventsIn:` method invokes the `p_generateEvent` method directly. However, in that case all subclasses are restricted to the selector specified in the superclass. If one subclass requires parameters to be passed to the `p_generateEvent` method, it has to define an additional method and it has to override the `p_simulate:timeoutEventsIn:` method in the parent class to invoke the new method. The first alternative clearly allows the designer more freedom during inheritance.

This section has covered the derivation of the SLOOP statements contained in the SLOOP classes. It is evident from the examples above that **most of the effort** is spent on the specification of the **correctness properties**. Once that has been completed, the derivation of the SLOOP statements is almost automatic, because in most cases there is a very simple correspondence between the correctness properties of a method and the resulting SLOOP statements. Several other issues have also been discussed, all with the aim of highlighting how the SLOOP method aids the designer to generate SLOOP programs that are **modular**, **extensible** and **flexible**.

6.6 Constructing the SLOOP program

Once the classes comprising the system have been identified, the SLOOP program is constructed. The latter consists of an *activation-section* and one or more packages containing the classes used by the system. At this stage it is convenient to define a class which performs the activation function of the system. In the call centre example this class is called `CC_SimulationActivation`. It creates the relevant instances and ensures that the required parallel methods are activated. **The correctness properties of this class are therefore the correctness properties of the system.** (The `CallCentreSimulation` program structure was first presented in Chapter 4, Section 4.3.1, where it can be seen where the `CC_SimulationActivation` class fits into the program.)

The purpose of this section is to **summarise** the results of the actions described in the earlier sections of this chapter **and to show how this information is used to determine the contents of the `CC_SimulationActivation` and `CC_Activation` classes**. This section also demonstrates how an **alternative representation** of the actions performed by the objects of the system can be used to check that the system contains all the required parallel methods to provide the desired behaviour. This alternative representation is given in a format reminiscent of a spreadsheet and its purpose is also to aid **understandability**.

6.6.1 Using the results of the design phase refinements to determine the contents of the sequential methods of the activation classes

The input of the design phase comprises an object model of the problem domain classes and a set of correctness properties describing the required behaviour of the system under development. During the design phase the problem domain (analysis level) classes are mapped onto solution domain (design level) classes. Some of these solution domain classes might be the same as the problem domain classes and others might be different. This depends on various factors:

- **Suitable existing classes** might be found in the SLOOP repository of reusable artifacts (e.g. the `InputQueue` class is replaced by the `OrderedCollection` class).
- Design level refinements might result in the identification of **new classes** (e.g. the `Configuration` class).
- Design level refinements might result in the identification of **new objects**, but the classes of those objects might already exist (e.g. the `OrderedCollection` class can be reused for the `timerEventQ` object).
- Reusable classes could be found that provide functionality that is **common** to some of the analysis level classes. The classes are then restructured to take advantage of the reusable class found in the repository (e.g. the `CommsProviderSimulator` and `ServiceProviderSimulator` classes are modified to inherit their common functionality from the `EventSimulator` class)

The two tables below **summarise** the effects of the design level modifications on the call centre classes and objects. Table 6-4 shows how the design level classes comprising the call centre compare with the ones identified during the analysis phase. Table 6-5 provides a list of all the design level objects defined for the call centre system. The additional objects resulting from the design phase refinements are pointed out. This is followed by excerpts from the sequential methods of the activation classes, demonstrating how the information in these tables is used to determine the contents of these classes.

Design level class	Analysis level class(es)	Relevant repository class
`OrderedCollection`	`InputQueue` `ServiceQueue`	`OrderedCollection`
`Connection`	`Connection`	`Object (superclass)`

`Array`	`ConnectionContainer`
	`ServiceProviderSimulator=`
	`Container`
	`ServiceCategoryContainer`
		`Array`

`ServiceCategory=` `Allocator`	`ServiceCategoryAllocator`	`Object (superclass)`
`ServiceRequest`	`ServiceRequest`	`Object (superclass)`
`Set`	`ServiceProviderSubset` `ServiceProviderCategories`	`Set`
`ServiceCategory`	`ServiceCategory`	`Object (superclass)`
`Configuration`	-	`Object (superclass)`
`EventSimulator`	-	`Object (superclass)`
`CommsProvider=` `Simulator`	`CommsProviderSimulator`	`EventSimulator` `(superclass)`
`ServiceProvider=` `Simulator`	`ServiceProviderSimulator`	`EventSimulator` `(superclass)`
`TimerServices`	`TimerServices`	`Object (superclass)`
`TimeoutElement`	-	`Object (superclass)`
`CC_Simulation=` `Activation`	-	-
`CC_Activation`	-	`Object (superclass)`

Table 6-4. Relationship between the analysis and design level classes.

Table 6-5 shows the instances defined for the classes after the design level refinements have been made (an asterisk indicates that there are multiple instances). In order to ensure that the core classes do not have to be modified when the simulation classes are replaced with the actual interface classes for a specific application, the instance names do not contain any references to the word simulator. Instead, the more generic term "agent" is used.

Design level class	**Instance(s)**

`OrderedCollection`	`inputQ`
	`serviceQ*`
	`timerEventQ` (See *Note 1*)
	`categoriesServed*` (see *Note 2*)
	`timeoutCollection element*` (See *Note 1*)
	`srCategoryNames` (See *Note 3*)
	`spCategoryNames` (See *Note 3*)

`Connection`	the instances* are the elements of the `userConnections` array

`Array`	`userConnections`
	`scContainer`
	`spAgentContainer`
	`timeoutCollection` (See *Note 1*)

`ServiceCategoryAllocator`	`scAllocator`
`ServiceRequest`	`serviceRequest*`
`Set`	`spSubset*` `spCategories*`
`ServiceCategory`	the instances* are the elements of the `scContainer` array
`Configuration`	`config` (See *Note 3*)
`CommsProviderSimulator`	`commsAgent`
`ServiceProviderSimulator`	the instances* are the elements of the `spAgentContainer` array
`TimerServices`	`timer`
`TimeoutElement`	the instances* are the elements of the `timerEventQ`
`CC_SimulationActivation`	`aCCSimulationActivation`
`Dictionary`	`srToSpCategoryMap` (See *Note 3*)

Table 6-5. Classes and instances defined for the call centre case study.

Note 1: The design level refinements of the `TimerServices` class introduced several new objects. The classes of these objects are existing Smalltalk library classes.

Note 2: The `categoriesServed` instance variable of the `ServiceProviderSimulator` class was introduced during the design phase in order to ensure that none of the service queues serviced by a particular `ServiceProviderSimulator` instance would be ignored for ever. After the design level refinements the `ServiceProviderSimulator` class is therefore a composite class containing an instance of the `OrderedCollection` class in order to represent the service categories serviced by the `ServiceProviderSimulator` instance .

Note 3: The `Configuration` class was introduced during the design phase. Its internal design also introduced several new objects. The classes of these objects are existing Smalltalk library classes.

The above information is now used to determine the contents of the sequential methods of the activation classes. The `CC_Activation` class is an abstract superclass which performs the instantiation of all the call centre classes that do **not** form part of the interface, i.e. all the classes that remain **unchanged** whether a simulation or an actual system is running. The methods that instantiate the interface classes are left as the responsibility of the subclasses. The `CC_SimulationActivation` class instantiates the simulation classes.

When an instance of the `CC_SimulationActivation` class is created, the `initialize` method of the parent class, `CC_Activation`, is executed as part of the instantiation. The `initialize` method ensures that all the relevant classes are instantiated upon system startup. In cases where the subclasses should determine which classes should be instantiated, the `initialize` method merely invokes additional methods that can be overridden by its subclasses. In the code excerpt below, the `initCommsAgent` and `initSPAgent` methods are examples of methods that are overridden in the `CC_SimulationActivation` class.

The statements of the `initialize` method are as follows:

```
sequential
config := self initManagement                                    "S1"
[] commsAgent := self initCommsAgent                             "S2"
[] userConnections := SmalltalkLibPkg:::Array new: maxConn
                                                                 "S3"
[] < [] i where 1≤i≤maxConn :: userConnections at: i
      put: (self initConnection: i)
  >                                                              "S4"
[] inputQ := SmalltalkLibPkg:::OrderedCollection new:
maxConn                                                          "S5"
[] scAllocator := self initServiceCategoryAllocator              "S6"
[] scContainer := SmalltalkLibPkg:::Array
    new: maxCategories                                           "S7"
[] < [] j where 1≤j≤maxCategories :: scContainer at: j
    put: (CC_CorePkg:::ServiceCategory setup: config)
  >                                                              "S8"
[] spAgentContainer :=
      SmalltalkLibPkg:::Array new: maxSP                         "S9"
[] < [] k where 1≤k≤maxSP :: spAgentContainer at: k
    put: (self initSPAgent)
  >                                                              "S10"
[] timer:= SystemUtilitiesPkg:::TimerServices setup: config
                                                                 "S11"
[] timerEventQ := SmalltalkLibPkg:::OrderedCollection new
                                                                 "S12"
end-sequential
```

Table 6-6 is used to check that each instance listed in Table 6-5 is accounted for in the above method. In some cases the instances are not specified explicitly in the above statement. However, they are created as a result of classes that are instantiated via the above statements. This is true in the case of composite classes. Some classes need not be created at startup (e.g. the `TimeoutElement` instances). These exceptions are noted explicitly.

Instance(s)	Statement number
inputQ	S5
serviceQ	S8 (the setup: method of the ServiceCategory class results in the creation of a serviceQ object)
timerEventQ	S12
categoriesServed*	S10 (the initSPAgent method results in the creation of a categoriesServed object)
timeoutCollection element*	S11 (the setup: method of the TimerServices class results in the creation of the timeoutCollection elements)
srCategoryNames	S1 (the initManagement method results in the creation of the srCategoryNames object)
spCategoryNames	S1 (the initManagement method results in the creation of the spCategoryNames object)
userConnections element*	S4 (the initConnection: method results in the creation of a userConnections element)
userConnections	S3
scContainer	S7
spAgentContainer	S9
timeoutCollection	S11 (the setup: method of the TimerServices class results in the creation of the timeoutCollection object)
scAllocator	S6
serviceRequest*	S4 (the initConnection: method results in the creation of a serviceRequest object)
spSubset*	S8 (the setup: method of the ServiceCategory class results in the creation of an spSubset object)
spCategories*	S8 (the setup: method of the ServiceCategory class results in the creation of an spCategories object)
scContainer element*	S8
config	S1 (the initManagement method results in the creation of the config object)
commsAgent	S2 (the initCommsAgent method results in the creation of the commsAgent object)
spAgentContainer element*	S10 (the initSPAgent method results in the creation of an spAgentContainer element)
timer	S11
TimeoutElement instances	**These objects are created dynamically after initialization.**
srToSpCategoryMap	S1 (the initManagement method results in the creation of the spCategoryNames object)

Table 6-6. Table used to cross-check that all classes referenced by the system are instantiated appropriately.

As noted earlier, some of the instances listed above are created via methods that are overridden by subclasses of the CC_Activation class. For example, the CommsProviderSimulator class is instantiated in the initCommsAgent method. In the CC_Activation class the statements of this method are as given below:

```
sequential
^self subclassResponsibility
end-sequential
```

This method is redefined as follows in the CC_SimulationActivation subclass of the CC_Activation class:

```
sequential
^CC_SimulationInterfacesPkg:::CommsProviderSimulator
      startSimulation
end-sequential
```

The classes that are not created directly via the initialize method of the CC_Activation class are those that are **most likely to be subclassed**. This is an example of the application of the Factory Method design pattern [GHJV95], which will be discussed in detail in Chapter 9, Section 9.3.1. Thus, the application of this design pattern makes it possible to leave the initialize method intact during subclassing. Only methods such as initCommsAgent need to be overridden during subclassing. The interested reader is referred to Appendix B, Sections B.2 and B.3 for details of the other methods invoked by the initialize method (e.g. initManagement).

After initialization, all the class invariants of the instantiated classes have to hold. The progress and precedence properties are achieved via the execution of the selected parallel methods of the various classes in the system (and via the sequential methods invoked by the parallel methods). The next section deals with the topic of ensuring that all the relevant parallel methods of a system are activated.

6.6.2 Determining the contents of the parallel methods of the activation classes

After all the classes have been instantiated during the instance creation of the CC_SimulationActivation class, the parallel methods required by the system need to be activated. This is achieved via the p_activate method inherited from the CC_Activation class.

The Template Method design pattern [GHJV95] is evident in the p_activate method. The design pattern itself and the motivation for its application in the p_activate method will be described in more detail in Chapter 9, Section 9.5.3. At this stage it suffices to say that the main purpose of the p_activate method is to ensure that all the required parallel methods of the call centre system are activated without necessarily invoking those methods directly. As a result many of the statements of the p_activate method are encapsulated in other methods of the CC_Activation class. The statements that are most likely to be overridden are encapsulated. Subclasses may therefore selectively override some of these methods, while the p_activate method remains unchanged. The statements of the p_activate method are given next:

```
parallel
self p_executeCPAgent
"The parallel methods of the commsAgent are not invoked
directly, but rather via the p_executeCPAgent method of the
CC_Activation class."

[] timer p_runTimer: timerEventQ
"Activate the parallel methods of the timer object.  The timer
parallel statements have the following functionality: Whenever a
timeout occurs, the TimeoutElement instance representing the
timeout is added to the end of the timerEventQ, which indicates
to the requestor that the specified timer has expired."
```

```
[] self p_categoriseAndAllocate
"The parallel methods of the scAllocator object are invoked via
the p_categoriseAndAllocate method of the CC_Activation class.
The scAllocator parallel statements have the following
functionality: Once a service request has been categorised, it
is removed from the inputQ and appended to the appropriate
serviceQ."

[] < [] j where 1≤j≤maxCategories :: (scContainer at: j)
      p_execute
  >
"Activate the parallel methods of the ServiceCategory instances.
Their parallel statements have the following functionality: For
each service category the associated service queue and set of
service provider agents are monitored.  If the service queue is
not empty and a service provider agent in the spSubset
associated with the service category is available to process a
new service request, the first element of the service queue is
removed and assigned to a service provider agent."

[] < [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
  >
"The p_executeConnection method of the CC_Activation class is
executed for each Connection instance in order to invoke the
parallel methods of the latter.  The parallel statements of the
Connection instances have the following functionality: When a
connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection.  Once all the procedures have been completed to
terminate the connection, the connection and its associated
service request are reset to their initial states."

[] < [] k where 1≤k≤maxSP :: self p_executeSPAgent:
      (spAgentContainer at: k)
  >
"The parallel methods of the service provider agents are not
invoked directly, but rather by executing the p_executeSPAgent
method of the CC_Activation class for each of the service
provider agents."

end-parallel
```

The `p_runTimer:` and `p_execute` parallel methods of the `TimerServices` and `ServiceCategory` instances respectively are activated directly as can be seen from the statements above. In contrast, the parallel methods of the `ServiceCategoryAllocator`, `Connection`, `CommsProviderSimulator` and `ServiceProviderSimulator` classes are activated indirectly. The statements of the `p_categoriseAndAllocate` and `p_executeConnection:` methods of the `CC_Activation` class activate the parallel methods of the `ServiceCategoryAllocator` and `Connection` classes respectively, as can be seen below:

```
message pattern p_categoriseAndAllocate
method properties
"..."
```

```
parallel
scAllocator p_categorise: inputQ using: scContainer
"The scAllocator monitors the inputQ.  If it is not empty, it
enables the categorisation of the first element (a service
request)."
[] scAllocator p_allocate: scContainer from: inputQ
"Once the first service request has been categorised, the
scAllocator removes it from the inputQ and appends it to the
appropriate serviceQ."
end-parallel

message pattern p_executeConnection: aConnection
method properties
"..."
parallel
aConnection p_informCommsProvider: commsAgent
"When a connection has entered the 'TERMINATING' state, the
communication provider agent is requested to terminate the
connection."
[] aConnection p_doWrapUp
"Once all the procedures have been completed to terminate a
connection, the connection and its associated service request
are reset to their initial states."
end-parallel
```

The parallel methods of the `CommsProviderSimulator` and `ServiceProviderSimulator` classes are activated via the `p_executeCPAgent` and `p_executeSPAgent:` methods of the `CC_Activation` class, but these methods are the responsibility of the subclass, as illustrated by the code fragments presented next:

```
message pattern p_executeCPAgent
method properties
"..."
parallel
self subclassResponsibility
end-parallel

message pattern p_executeSPAgent: spAgent
method properties
"..."
parallel
self subclassResponsibility
end-parallel
```

The `CC_SimulationActivation` class redefines these methods as follows:

```
message pattern p_executeCPAgent
method properties
"..."
parallel
commsAgent p_simulate: timer timeoutEventsIn: timerEventQ
[] commsAgent p_generateEvent: userConnections target: inputQ
"The commsAgent simulates the establishment of new connections
at random intervals (within a configured range).  A simulation
timer is started after initialization and restarted each time
after the establishment of a connection has been simulated. The
latter is done by placing the service request associated with
the new connection into the input queue.  The commsAgent ensures
that the capacity of maxConn connections per call centre is not
exceeded, therefore a message is displayed indicating that all
```

```
connections are busy if the maximum number of connections are
currently assigned."
end-parallel

message pattern p_executeSPAgent: spAgent
method properties
"..."
parallel
spAgent p_simulate: timer timeoutEventsIn: timerEventQ
"When a service request has been assigned to a service provider
simulator, the latter simulates the time it takes to service the
service request by starting a random timer.  When this timer
expires, it represents the completion of the service."
[] spAgent p_generateEvent
"When the service provider has completed the service, it
indicates that the connection should be terminated."
[] spAgent p_updateCategoryIndex: scContainer
"Update the index into the categoriesServed collection if the
serviceQ of the current category being served by this spAgent is
empty."
end-parallel
```

Thus, the parallel methods that are specific to the interface classes are activated by methods redefined in the appropriate subclasses of the `CC_Activation` class.

Another way of viewing the dynamics of a SLOOP program is to compare it with the way in which a spreadsheet operates. There are a number of rules and when an event occurs, the rules that are affected are evaluated. This may result in another event, which again results in the evaluation of some rules. This process continues until the spreadsheet reaches a stable state, which is only disturbed if another event occurs.

A SLOOP program comprises a number of parallel statements that execute infinitely often. When an event occurs, the statement which is affected will eventually be executed. As a result of the execution of this statement, other events may be generated. The results of these events will be seen once the affected statements are executed. This process continues until a stable state is reached, i.e when the state remains unchanged no matter which statement is executed.

Table 6-7 below presents the parallel statements and events of the `CallCentreSimulation` program in the spirit of a spreadsheet. Only the parallel statements that appear in the methods of the `CC_Activation` and `CC_SimulationActivation` classes are shown.

Object ---------- Event	comms Agent	timer	scAllocator	scContainer at: j	spAgent Container at: k	user Connections at: i
new comms= Agent simulation timer must be started	p_simulate: timeoutEvents In:					
timer running for comms= Agent		p_runTimer:				
timer for comms= Agent expired	p_simulate: timeoutEvents In:					
comms= Agent must generate event	p_generate Event: target:					
inputQ is not empty and service request not yet categorised			p_categorise: using:			
first element of inputQ categorised, but not yet allocated to a serviceQ			p_allocate: from:			
serviceQ is not empty				p_execute		
service request assigned to a service provider simulator					p_simulate: timeoutEvents In:	
timer running for service provider simulator		p_runTimer:				
timer for service provider simulator expired					p_simulate: timeoutEvents In:	
timer for service provider simulator expired					p_generate Event	
service queue currently being serviced is empty					p_update Category Index	
connection needs to be terminated						p_inform Comms Provider:
connection termination needs to be wrapped up						p_doWrapUp

Table 6-7. Events and actions of the call centre in a tabular format.

The initial trigger is the new `commsAgent` event that must be simulated. It results in a timer being started for the `commsAgent` object (via the `p_simulate:timeoutEventsIn:` method). The `timer` objects monitors all timers via its `p_runTimer:` method. When the `commsAgent` timer expires (detected via a statement in the `p_simulate:TimeoutEventsIn:` method), the `commsAgent` inserts a service request into the `inputQ` (unless all the connections are busy) and indicates that a new event is again required. The whole process is repeated *ad infinitum*.

In the meantime, the `scAllocator` detects that the `inputQ` is not empty. It categorises the first element (via the `p_categorise:using:` method), assigns it to a `serviceQ` and removes it from the `inputQ` (via the `p_allocate:from:` method). This is done whenever the `inputQ` is not empty. Each `ServiceCategory` instance (each one is an element of the `scContainer`) checks the `serviceQ` associated with it. If it is not empty and a service provider can be found that is available and services requests of that category, it assigns the first element in the `serviceQ` to the service provider and removes it from the `serviceQ` (via the `p_execute` method).

The assignment of a service request to a service provider causes a new service provider simulation event to be required. As a result a new service provider simulation timer is started (via the `p_simulate:timeoutEventsIn:` method). The `timer` object monitors all timers (including this one) via the `p_runTimer:` method. When the timer expires, the service provider agent generates an event to terminate the connection (via the `p_generateEvent` method). When a connection has to be terminated, the statements to inform the `commsAgent` are executed (`p_informCommsProvider:`) and the connection is returned to its idle state (`p_doWrapUp`).

In addition to the parallel statements discussed above, the `ServiceProviderSimulator` instances also execute the `p_updateCategoryIndex:` parallel method. This method updates the `categoryIndex` instance variable if the service queue currently being serviced by the `ServiceProviderSimulator` instance is empty.

The sequence in which the statements are executed, is not important. For example, the statement which processes a service queue may be executed before the one that processes the `inputQ`. If there is no service request in the service queue, the former statement will have no effect until the `inputQ` has been processed.

In the case of the `p_activate` method, there are also several quantified statements, e.g.

```
< [] i where 1≤i≤maxConn :: self p_executeConnection:
      (userConnections at: i)
        >
```

Each instantiation of the above quantification represents a separate statement. Before a statement is selected for execution within a cyclic method, all the quantified statements are instantiated. Only a single statement is selected during an execution of the method. Thus, if a method consists of a quantified statement representing *n* statements only one of those *n* statements is executed when the method is invoked. If a method contains enumerated statements as well as quantified statements, each instantiation of the quantified statement competes on an equal footing with the enumerated statements for selection.

The above tabular representation of the parallel statements of the `CallCentreSimulation` program augments the description of **object interactions** in a SLOOP program. It aids the designer in **checking** that all the parallel statements that need to be executed by the system are indeed activated via the activation class and its ancestors. Once the contents of each class and instance method has been determined, an executable program can be derived. This is the topic discussed in Chapter 8.

6.7 Making the design more reusable

Once a design has reached the stage where it fulfills the functional requirements, it is time to consider improvements in order to make the design more reusable. This aspect of the design phase is covered in detail in Chapter 9, but a brief introduction is provided here.

One way of increasing the **extensibility** and **flexibility** of the design is by considering likely future requirements and by evaluating the ease with which such enhancements could be accommodated. For example, one likely future requirement of the call centre system is the capability to ensure that service requests are not always assigned to the same service provider if multiple service providers are available. This can be achieved quite easily if the `spSubset` of each service category is implemented as an instance of the `OrderedCollection` class rather than as an instance of the `Set` class. It is done as follows: A service request is always assigned to the first element of `spSubset` that is available to accept a new service request. When this assignment is performed, that element is removed from `spSubset` and added to its tail. This ensures that one service provider does not do all the work.

Thus, although the current requirements specification does not prescribe any ordering regarding the allocation of service requests to available service providers, it is prudent to make provision for some type of ordering. The `OrderedCollection` class does not restrict the design to a specific ordering; that depends on the way in which the clients add and remove elements from the `OrderedCollection` instance. However, it does guarantee the relative ordering of its elements while they form part of the collection.

Another way of improving the design is by incorporating design patterns [GHJV95]. An introductory description of an example from the call centre case study is presented next. As stated in Chapter 5, the identification of the service user is important in some types of call centres, because the identity of the service user may play a role in the allocation of the service request to the appropriate service queue. The calling telephone number may suffice as an identification, in which case the above parallel statements need not be changed. However, in other cases it may be necessary to extract some additional information about the service user from a database, using the calling telephone number as key into the database. For example, some service users may be paying an additional fee in order to ensure that all their service requests are treated as high priority requests. Such information may be contained in a database.

Such an extension to the system could easily be accommodated by replacing the algorithm used in the `p_categorise:using:` method of the `ServiceCategoryAllocator` class. Chapter 9, Section 9.5.4, shows in more detail how the Strategy design pattern [GHJV95] can be used to allow for the replacement of one algorithm by another.

6.8 Summary

The purpose of this chapter has been to demonstrate the **feasibility** and **advantages** of the SLOOP approach during the design phase of system development. Feasibility is an important concern, given the goal of making the method **usable by ordinary practising software designers**. The main concern is the amount of effort that is required when placing so much emphasis on correctness properties during the design phase. It was shown that a **pragmatic** approach is advocated in the SLOOP method. Only the **method** properties need to be specified formally. Informal property specifications suffice at the **class** level.

Other considerations also influence the direction taken regarding the level of formality. One aspect is the fact that a formal description of the classes in a system under development might not

match the formal specifications of artifacts in a repository, even though there could be enough similarities to warrant reuse. **Differences in terminology** is one reason for discrepancies.

Another factor is the **incomplete status** of the classes of the new system when the design phase is entered. Quite often design level refinements result in new insights which may even require updates to the requirements analysis phase deliverables. Examples of such situations were given in Section 6.2.1. It is therefore argued that the design process is not a mechanical procedure which could be automated easily, since there are too many factors that require human assessment while searching the repository for reusable artifacts[6] and also while performing the design level refinements. This is one of the reasons why the emphasis is on **informal** property specifications during these activities.

Another reason was the possibility of **reusing** the formal property specifications of the methods of the classes already present in the repository. Once a reusable class has been identified, all the effort of translating informal property specifications into formal ones can be saved by reusing the existing specifications.

Although the SLOOP method is not a formal method, correctness properties play a crucial role throughout the software development process. It was shown how these correctness properties were utilised during various stages of the design phase, e.g. during the identification of reusable artifacts and during the refinement of the design. The **derivation of the SLOOP statements** for each method become straightforward once their correctness properties have been determined. This was illustrated by some examples in Section 6.5. By concentrating on the correctness properties at all stages of the design, the likelihood of **a more correct result** increases.

The **impact** of the **SLOOP computational model** on the design phase was another issue which was investigated in this chapter. Although this model results in certain constraints during the design phase (e.g. blocking statements may not be used), these constraints do not restrict the applicability of the SLOOP method. It is still possible to provide a design for a class such as `TimerServices`, as was shown in Section 6.3.2. In fact, the resulting design has the advantage that issues such as the **assignment of statements to processes** are **postponed** to the implementation phase. The computational model also makes it easy to add new parallel statements during subclassing. As was shown in Section 6.2.3, the concept of **inheritance** fits in very neatly with the SLOOP approach.

The latter part of this chapter demonstrated how a SLOOP program was constructed. Appendix B contains a complete listing of the first level of refinement of the call centre classes. It serves to illustrate the applicability of the method to a **non-trivial** problem.

The next chapter is devoted to **reasoning** about correctness properties. It shows that the correctness properties can be **reused** along with all the other aspects of a class. The implementation phase is covered in Chapter 8, while Chapter 9 shows that the SLOOP design method is **amenable** to the usage of well-known **design patterns**, an issue which was touched upon briefly in Section 6.7.

[6] In [SiCh97] Sivilotti and Chandy also note that it is not tractable to automate the procedures for identifying matching components in the repositories of reusable components.