Pattern-based Software System Modelling and Performance Requirement Verification

Master Thesis

26 August 2005

Jan Hendrik Boelens

Department of Computing Science, University of Groningen

Supervisors:

University of Groningen — Ir. S. Achterop
Thales Nederland — Dr. Ir. J. Skowronek
Ir. M. Glandrup

- Unclassified -

Abstract

Software design patterns provide proven and well-understood solutions to reoccurring design problems. They are often aimed at realizing a very specific combination of non-functional software characteristics in a design. For these reasons, it would be desirable to use such design patterns as building blocks in the process of developing software system designs. In order for this to be achieved, good tool-support for patterns is required. Hence, this master thesis presents a process for creating pattern-based software system designs in the context of tool-support. We also aim to leverage these design patterns in order to predict performance characteristics of the resulting software design. Finally, we illustrate the process using a fire fighting system case study and present the pattern-based software tool that was developed during this project.

Table of Contents

1 Introduction 5
1.1 Purpose of the Thesis 5
1.2 Industrial Background 5
1.2.1 An introduction to Thales 5
1.2.2 An introduction to the Business Unit Combat Systems 5
1.2.3 An introduction to SPLICE and DDS 6
2 Problem Statement & Research Goals 7
2.1 An analogy 7
2.2 Problem Statement 7
2.3 Research Goals 8
2.4 Problem Solving Approach and Solution Overview 8
2.5 Introduction to the Case Study 9
3 Theoretical Background Information 11
3.1 Patterns 11
3.2 Software System Modelling and Meta-Modelling 12
3.3 Non-functional Software Properties 13
4 Software System Modelling 15
4.1 Analysis and Requirements Definition 15
4.2 The Software System Meta-model 17
4.2.1 Modelling the Hardware Configuration 17
4.2.2 Modelling the Software Entities 18
4.2.3 Modelling the Software Behaviour 19
4.2.4 Modelling the Deployment of Software Entities 20
4.2.5 Completing the Meta-Model 21
4.3 Example: A Fire fighting System Software Model 22
4.4 Tool-support for Software System Modelling 28
4.5 Summary 29
5 Modelling Component Interaction Using Patterns 30
5.1 The Observer Pattern 30
5.2 Analysis and Requirements Definition 31
5.3 The Pattern Meta-model 33
5.3.1 Participants & Structure 33
5.3.2 Collaborative Behaviour 33
5.3.3 Summary of the Pattern Meta-model 36
5.4 Example: A Model of the Observer Pattern 36
5.5 Designing Software Systems Based on Patterns 38
5.6 Extending the Software System Meta-model 39
5.7 A Visual Representation for Pattern-based Designs 42
5.8 Example: A Pattern-based Fire fighting System Software Model 43
5.9 Tool-support for Pattern-based Software System Design 45
5.10 Summary 48

6 Verifying the Non-functional Performance Requirements on a Pattern-Based Software System Design 49
6.1 Analysis and Requirements Definition 49
6.2 Pattern-based Performance Verification 50
6.2.1 Patterns and Non-functional Characteristics 50
6.2.2 Quantification of Pattern Performance Characteristics 50
6.2.3 An Alternative for Pattern-based Verification 51
6.3 A Process for Verifying Performance Requirements 51
6.3.1 The Software System Model Revisited 51
6.3.2 Defining the Performance Requirements 52
6.3.3 Modelling the Hardware Configuration 54
6.3.4 Modelling the Software Configuration 57
6.3.5 Calculating the Performance Characteristics of the System 60
6.3.6 Verifying the Performance Requirements 63
6.4 Tool-support for Performance Requirement Verification 65
6.5 Summary and Discussion 66
7 Developing a Pattern-Based Design Tool 67
7.1 Open Source: A Starting Point 67
7.2 Design of the User Interface 67
7.3 The Tool Software Architecture 69
7.3.1 The development process 69
7.3.2 Modular Decomposition of the Design 69
7.3.3 Detailed Discussion of Modules 71
7.4 Summary 72
8 Presentation of Research Results & Future Work 73
9 List of definitions 74
10 Bibliography 75

1 Introduction

1.1 Purpose of the Thesis

The purpose of this thesis is to research the application of design patterns to software designs in the context of tool-support. Although the area of design patterns has been explored for years and many design patterns have been published, tool-support for such patterns has not yet reached a satisfactory state.
For the purpose of this thesis, our scope is limited to the high-level design of software systems, where patterns may be used for describing the interaction between the coarse-grained software entities that together constitute the software system.
The aim is to address both the creation of a "pattern-based" software design and the verification of performance requirements with respect to such a design.
Chapter 2 provides a problem statement and defines the research goals more formally. Furthermore, it provides a detailed overview of the way in which subsequent chapters are related. Chapter 3 introduces the reader to some theoretical concepts and refers to a number of related publications. Chapter 4 introduces a process for modelling software system designs, whereas chapter 5 adapts this process to include patterns.
A model for verifying performance requirements on a pattern-based software design will be introduced in chapter 6.
Finally, chapter 7 addresses the issue of tool-support specifically and provides the design of the tool that was developed to demonstrate the concepts described in this thesis.

1.2 Industrial Background

The topic for this master thesis was provided by Thales Nederland, where the thesis-related research was performed. Therefore, a brief overview will be provided of the company in general as well as the business unit that provided the topic for the master thesis. Finally, a few of the business unit's products that are related to this research will be introduced.

1.2.1 An introduction to Thales

Thales (formerly known as Thomson CSF) is an international group of companies operating in three main markets: aerospace, defence and IT-solutions & services. It currently employs 60.000 people worldwide, most of whom work in France where the company's main headquarters is situated.
Furthermore, the company operates in several businesses, each consisting of a specific product range. These businesses are: aerospace systems, air systems, land & joint systems, naval systems, security systems and finally the services business.
In the Netherlands, the Thales Group is represented by the Thales Nederland holding, which is particularly active in the naval domain. Although Thales Nederland has several locations throughout the country, most employees are based in Hengelo.

1.2.2 An introduction to the Business Unit Combat Systems

The naval division is divided into several business units, which are spread over several countries. This master thesis was conducted within the Business Unit Combat Systems (BU-CS), which develops and integrates so-called combat management systems for naval vessels. The purpose of such a system is to integrate all sensors, weapon systems and other equipment in order to have a single command & control system for operating the entire ship: a *system of systems*.
The combat management system, currently marketed under the name *Tacticos*, consists of a collection of interacting applications that are loosely coupled by a custom-developed middleware product, which is known as *SPLICE*.

1.2.3 An introduction to SPLICE and DDS

Conceptually, SPLICE is a data-centric middleware product that allows applications to share data and pass it between them without knowing of each other's existence. Practically, this means that an application produces some kind of data and just hands it off to the middleware (SPLICE). It is not concerned with who consumes the data, or where the data will be sent and stored. If a second application now needs to access or update the data produced by the first one, it may request SPLICE to either provide it with the previously produced data or it may itself produce an update invalidating the previous data.

The data in the system is organized into so-called *topics*. A topic identifies a particular kind of data and its (design-time) definition tells an application how that kind of data is to be interpreted.

Topics enable a *publish-subscribe architecture*, in which applications can simply publish data of a particular topic in the entire system, while every other application may subscribe to that topic. Such an architecture allows very loose coupling of individual applications.

Of course, this loose coupling comes at a price: Consumers of a topic may become dependent on the existence of at least one producer of such a topic. In large systems, many such interdependencies may develop over time, making the system harder to maintain. Moreover, as it is hard to keep track of what data is contained in which topic, unplanned redundancy may start to occur, making it more difficult to maintain a globally consistent state.

Furthermore, SPLICE is a publish-subscribe middleware infrastructure, providing features such as distributed data-storage and redundancy.

Finally it should be noted that Thales has worked on standardizing (parts of) its middleware infrastructure by actively participating in the specification of the Object Management Group's (OMG) standard for "data distribution services" (DDS). This standard [Omg04] was released in 2004 and is largely based on SPLICE.

2 Problem Statement & Research Goals

This chapter introduces the problem domain that this thesis aims to address. After a brief introduction in the first section, section 2.2 defines the specific problems that led to the research goals defined in section 2.3. Given the research goals, section 2.4 provides an analysis of the problem domain in order to determine the approach for finding a set of satisfactory solutions. It then provides the reader with an overview of the different parts of the solution and how they relate to one another. Finally, this chapter concludes by describing a scenario for a case study that is used throughout this thesis in order to illustrate the concepts developed in each chapter.

2.1 An analogy

Consider the following analogy: When a new car is designed, engineers do not start from scratch by designing a mechanism for transferring power from the engine to the wheels. Neither do they occupy themselves with the question of how the car should be operated by the driver. How is this possible?
The reason for this is that the people designing the car make use of experience that was gained by others in the past. They already know efficient solutions to the problems mentioned above and have a common vocabulary to identify them: the crankshaft and the steering wheel respectively. Not only does every engineer understand these concepts, but a steering wheel can be added to a car-design as such: a car designer does not have to explain to a mechanic what the steering wheel's purpose is and no one would try to use it for anything other than steering the car.
Although this example may seem trivial, the important fact to notice is that standard solutions to reoccurring problems were formulated. Such standard solutions simplify the design of complex systems, because designers may simply "apply" a well-known solution to their design problem, avoiding the effort and risk involved in pioneering a new solution, while achieving predictable results at the same time. These problem-solution pairs exist in every engineering discipline and are generally referred to as "patterns".

2.2 Problem Statement

The point of the analogy presented in section 2.1 is that patterns in disciplines other than software engineering are well developed and can be identified as such in designs. In software design, patterns are available as well, but they have been far less standardized and formalized, nor is there any well-defined way of integrating them into software designs.
This is considered to be a problem, because patterns that are not explicitly identified are difficult for a software engineer to recognise. This makes it more likely that one software engineer might misinterpret a design made by another and (unintentionally) "break" some of the original design philosophies. Ultimately, this means that patterns may disappear over time as subsequent changes to the design are made. As the pattern disappears, the non-functional properties it displays may also be affected.
With appropriate tool-support, it might be possible to explicitly instantiate patterns in software designs, while the tool might warn a designer if any pattern were about to be "broken".
Once a software design based on patterns (a so-called pattern-based design) is available, it might be possible to leverage knowledge about the patterns contained in the design to estimate performance characteristics of the resulting software product. This would be very valuable in evaluating software designs prior to implementation, which might lead to higher quality designs and thus, ultimately, to lower development cost due to less "performance correction effort" after and during the implementation phase. Obviously, such pattern-based design evaluation procedures are a good candidate for tool-support.
Summarizing, we would like to have a well-defined process for creating pattern-based designs as well as a process for estimating performance properties of such designs. This thesis focuses on two specific performance properties: *end-to-end time* and *throughput*. End-to-end time is defined as the time that a message takes to "travel" between two specified "points" in a software system
Throughput is the amount of data being sent from one software entity to another in a given period of time. A software system has to be able to function correctly in a given throughput-scenario. Such a scenario could

thus be specified as a performance requirement on the design. For our requirement verification process, our interest is mainly in a worst-case system analysis.
Finally, the processes that result from the research should be suitable for tool-support.

2.3 Research Goals

The research goals derived from the problem statement as specified in the previous section are as follows:

1. Development of a process for expressing pattern-based software system designs.
2. Development of a process for pattern-based estimation of performance characteristics (end-to-end time and throughput) of a pattern-based software system design with the aim of verifying requirements with respect to these characteristics.
3. Development of a software tool that implements the processes mentioned under 1 and 2 in order to demonstrate that these processes are suitable in the context of tool-support.

2.4 Problem Solving Approach and Solution Overview

This section explains the steps involved in progressing from the specified research goals to a satisfactory set of solutions. Then, an overview of the envisioned solutions is provided.
First, consider the primary research goal: In order to develop a process for expressing pattern-based designs, we should first define what we mean by "a design". A design is, per definition, a model of the envisioned system, meaning that you have to specify limits to the level of detail that will be expressed within the design. Chapter 4 discusses the software system modelling approach used throughout this thesis. Considering the first research goal once more, we need to augment or alter the software system design approach in a way that will enable the designs to contain patterns. This requires a prior analysis of which patterns should be expressible, since the requirement of being able to "represent any possible pattern" may very well be too large for this thesis. Furthermore, embedding patterns in designs might be a big leap, if we do not yet have an approach for modelling patterns in a stand-alone fashion at our disposal. Therefore, chapter 5 starts by developing a process for modelling patterns individually, before it continues to explore an approach for integrating them into a software system model.
Obviously, the research goal of facilitating tool-support should be considered in an early stage. The issue is addressed throughout the thesis, yet a complete tool design is first presented in chapter 7.
Finally, the research goal of developing a process for pattern-based estimation of performance characteristics is addressed in chapter 6. One of the first issues to analyse in that context is to what extent pattern characteristics could be leveraged for this estimation process. After this analysis, we need to determine the input that would be required for constructing a realistic evaluation scenario, which should allow the development of a formal estimation process.
The dependencies between the topics discussed in each chapter are illustrated in Figure 2-1.

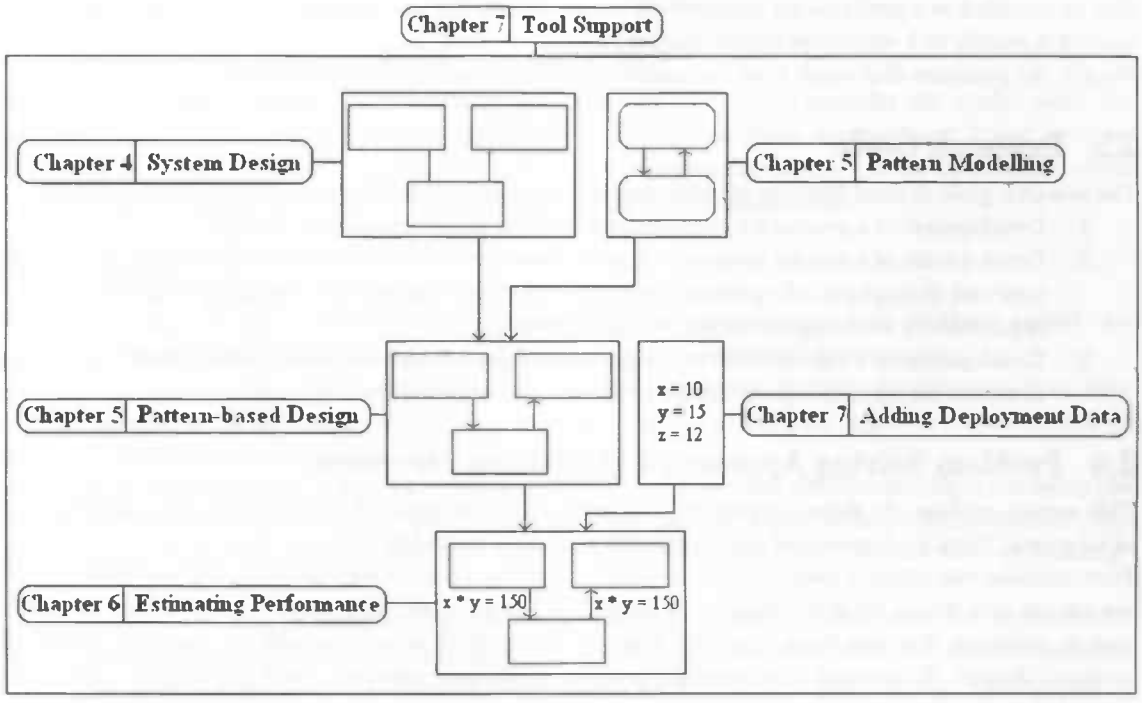

Figure 2-1 Solution overview

2.5 Introduction to the Case Study

This section introduces the case used throughout this thesis to illustrate the theory developed in each chapter. The case consists of a fire fighting-support software system, the design of which will be developed in the rest of this thesis. Although the design has been kept rather trivial, the scenario should contain most elements of a typical software system.
The following scenario describes the system to be designed:

A new tunnel is to be built, which is to be equipped with a fire fighting support system. The role of the system is to automatically detect a fire as soon as it occurs and to determine if hazardous chemical particles are present in the tunnel. For these purposes, the system is to be equipped with smoke- and chemical detectors.
Automatic threat assessment software will process the sensor inputs in order to compile an overall threat assessment, based on the level of smoke, the detected chemicals and possibly other criteria. Another software application, the fire fighting planning system, will process threat assessments to determine the course of action. Additionally, it also uses input from the chemical detectors for decision making, since some chemicals might pose a grave risk to fire fighters, making it undesirable to deploy such units.
The planning system can trigger any number of fire fighting measures, two of which we will consider: sprinkler installations installed inside the tunnel and mobile fire brigade units that may be dispatched from a nearby fire brigade station.
During the fire fighting procedure, the system continuously updates the threat evaluation and fire fighting plans using the detectors mentioned above, as well as feedback received from fire fighters on the scene.

The dataflow described in the case above is given by the image below:

Figure 2-2 Fire-fighting system

A pattern-based design for this system will be developed in subsequent chapters. Furthermore, some performance requirements (end-to-end time and throughput) will also be specified for the system, allowing the demonstration of the requirement verification process developed in this thesis.

Before we elaborate on the system design however, we will first present some related work in order to provide the required theoretic basis for the rest of the thesis.

3 Theoretical Background Information

This chapter describes relevant work that has been done in the past. It provides the reader with the necessary theoretical knowledge to fully comprehend the rest of this thesis. Furthermore, it also provides some useful references to related literature. The first section elaborates on the history of patterns, while subsequent sections deal with system modelling and non-functional requirements.

3.1 Patterns

Patterns originate from the world of building architecture. Christopher Alexander, an architect, argued that existing architectural methods failed to meet individual, social and ecological requirements.
Moreover, he recognized certain similarities between several different building architectures that were generally considered "good" architectures. Thus, he started analysing this issue, captured the similarities in such "good" architectures and documented them systematically.
Each similarity was described by first providing a reoccurring architectural problem and then providing the solution that had been applied in the "good" architectures. Alexander called these problem-solution pairs *patterns* and published the ones he had described in one of his books: "The Timeless Way of Building" [Ale79].
Whereas Alexander was a building architect, some people realised that his ideas were also applicable to software design. Thus, in 1995 Erich Gamma et al. [Gam95] published their work on software patterns, which they called *design patterns*. Quoting their work, software design patterns are "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." Their book contains descriptions of many common patterns used in software design. Much like Alexander, they use a very structured approach to describe their patterns, always providing the following information for each pattern:

- **Intent** - A brief description of what the pattern intended to accomplish
- **Motivation** – A more detailed description of when to use this design pattern
- **Applicability** – Describes in which context the pattern could be applied
- **Structure** – A visual representation of the classes and relations that constitute the pattern
- **Participants** – Specifies the purpose of each class in the pattern
- **Collaborations** – Describes the details of the collaboration between the participating classes

They even went one step further, by structuring the patterns into three categories: structural, behavioural and creational patterns. Even though this pattern catalogue was very structured and had an enormous impact on the way patterns were expressed, it was not suitable for automatic processing, since it relied mainly on textual, human-readable descriptions.

Soon after Gamma's book appeared, Riehle [Rie96A] and other researchers noted that the class-diagrams used for describing design patterns were not really describing general solutions to a particular problem. Rather, a class-diagram presented an efficient implementation of such a solution. These researchers proposed an alternative pattern model based on role-diagrams. The main difference is, that class-diagrams define the relationships and interaction between classes in the software design. A role-based model of a pattern simply defines several roles that participate in a pattern, not specifying how these roles should be mapped onto the classes of a design. This leaves a designer more freedom to assign several roles to one class when this fits better into his design.
In the Netherlands, Florijn et al. [Flo96] [Flo98] performed some research in the field of tool-support for pattern-based design. They developed the "fragment model" for expressing patterns such designs. One of the key features of this model is, that it does not stop at representing the roles and relationships participating in a pattern. Instead, it also assigns roles to methods, attributes, super-classes and other modelling entities. This allows them to map a pattern even more flexibly onto a software design, since the relationships between the roles do not have to be obvious in the design.

Finally, Buschmann et al. [Bus96] describe a more coarse-grained form of software patterns: architectural patterns. To quote their book: "Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of rules and guidelines for organizing the relationships between them."

3.2 Software System Modelling and Meta-Modelling

This section first explores the concepts of modelling and meta-modelling, after which it discusses the application of these concepts to the software domain. Although this may seem trivial, we will first consider what constitutes a model: A model is "an abstraction of phenomena" in the real world. Therefore a model is, per definition, a simplification of reality. Furthermore, a model is usually created with a particular context in mind, which means that the applicability of the model will be limited to this particular context.
Thus, modelling is the practice of creating appropriate abstractions of the real world in order to describe those properties and relationships that we are interested in.
A higher level of modelling is called *meta*-modelling. Whereas a model describes phenomena in the real world, a meta-model defines the "constructs" and "rules" needed to create such a model (i.e. a meta-model can be considered a language for constructing models). In short, a meta-model is essentially a model of a model: It defines what a model should "look like" (i.e. a template). The importance of a meta-model lays in the fact that it provides the basis for interpreting a model. For example, if two software engineers were to draw a model of the same software system, both might draw a collection of boxes and lines. Yet, the meaning of the modelling element "box" could still be very different in either diagram: In one it might mean "class", while in the other it might mean "interface". Both engineers apparently use the same modelling elements (i.e. boxes and lines), but the meaning of these modelling elements is still a matter of interpretation. This is the kind of problem that a meta-model solves, by defining all valid modelling elements, their relationships and their meaning (semantics). A brief and informal introduction to meta-modelling may be found in [Met02].
Obviously, we should still discuss the relevance of (meta-)modelling to software design. Software systems are usually highly complex systems, which is why software engineers often draw models abstracting only those "phenomena" that they are interested in (i.e. a view). One model might describe control-flow, while another might describe class-hierarchies. Creating such models is a common activity in design. Yet, as stated in the example above, a problem occurs when several engineers use different notations for the same things. Solving this problem has been the driving force behind standardization of software modelling languages. Note that such modelling languages are actually software meta-models, since they describe the constructs and rules necessary to create a software model. At present, the Universal Modeling Language (UML) is the most common meta-model for describing software systems [Omg03]. Interestingly, UML itself adheres to a higher-level meta-model for meta-models: The Meta-Object Facility (MOF) Specification [Omg00]. For completeness, it is mentioned that such a meta-model for meta-models is called a *metameta*-model. The relationships between these layers of models are depicted in Figure 3-1 below.

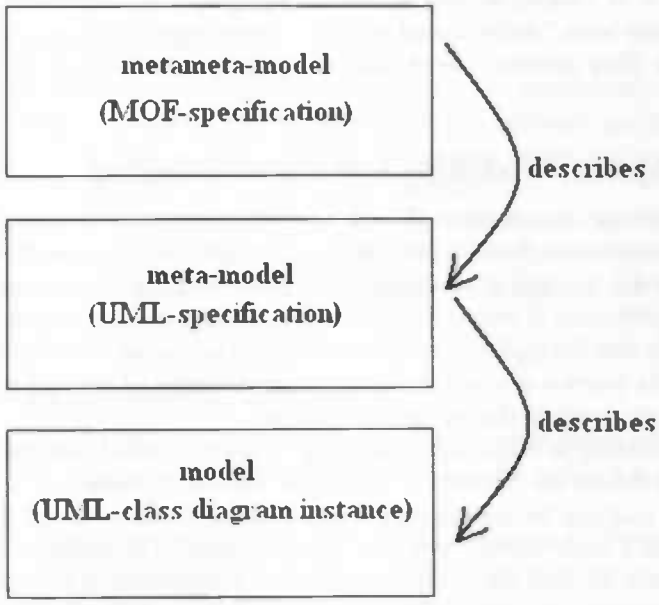

Figure 3-1 Meta-modelling concepts

3.3 Non-functional Software Properties

Very often, software development is mainly concerned with creating software that satisfies the functional requirements of the tasks it was designed to do (i.e. it does the work that it is supposed to do). Yet, this approach neglects the non-functional requirements on a design, which are often referred to as the *quality requirements*. Quality in this context, refers not to whether the software performs the required task, but to the way in which the software performs these tasks. For example, in a fire fighting software system, the fact that the system reports fires (functional) is a prerequisite. Yet, it is at least as important that the system reports the fire within an acceptable time (performance), that it reports every fire under all possible circumstances (reliability) and that it does its job 24 hours a day (availability). Such requirements are considered non-functional and are the topic of many research projects.

In order to achieve a common vocabulary for referring to non-functional characteristics of software, the International Organization for Standardization (ISO) published a definition of the most common characteristics in the ISO 9126-standard [Iso01]. The ISO-defined non-functional characteristics are:

- **reliability**-related: maturity, fault tolerance, recoverability, availability, degradability
- **usability**-related: understandability, learnability, operability, explicitness, customisability, attractivity, clarity, helpfulness, user-friendliness
- **efficiency**-related (a.k.a. performance): time behaviour, resource behaviour
- **maintainability**-related: analysability, changeability, stability, testability, manageability, reuseability
- **portability**-related: adaptability, installability, conformance, replaceability

Elaborating on all these characteristics is beyond the scope of this brief introduction. We therefore refer the interested reader to the official ISO-document for a more thorough discussion.

The issue of considering non-functional requirements during high-level system design is discussed in detail by Bas et al. in their book on software architecture [Bas03].

This research focuses on two specific non-functional characteristics of a software system: throughput and end-to-end time. These characteristics are both performance-related (ISO uses the term *efficiency*) and will be discussed in more detail in a subsequent chapter.

Finally, Bas et al. define software performance as follows:

> *Performance is about timing. Events occur, and the system must respond to them. [...] Basically, performance is concerned with how long it takes the system to respond when an event occurs. [...] A response might be the number of transactions that can be processed in a minute, [...], or a response might be the variation in the response time.*

4 Software System Modelling

Chapter 2 introduced the scenario that we use for system modelling and verification in the rest of this thesis. In this chapter, we develop a so-called *meta-model* for modelling software systems (meta-modelling is discussed in section 3.2). The aim is to provide a software system meta-model that we will use as a basis for pattern-based system models and verification in subsequent chapters.

The first section of this chapter specifies the requirements and constraints that the meta-model will be subject to. After that, the meta-model will be developed in a step-by-step fashion. Along the way, it will be demonstrated how the meta-model might be instantiated using our fire fighting case study in order to come to an overall system model at the end of this chapter.

4.1 Analysis and Requirements Definition

In order to develop a satisfactory meta-model, it will be necessary to first define what will be required of our system model. The fire fighting scenario from section 2.5 will be analysed in order to derive these requirements.

Imagine the smoke detector from our scenario: it is a device that is mounted somewhere on the inside of the tunnel. Now imagine that it is connected to a computer that records and processes its smoke measurements. This computer is connected to a network of computers all running part of the tunnel safety system. One of these computers on the network, in a centralized control room, reads the input from all smoke sensors and compiles a threat assessment report of the current situation in the tunnel.

Periodically, the smoke sensor provides updates, which are processed by a program on the corresponding computer. The processed measurements are then sent over the network to the threat evaluator. This software entity uses several sub-systems to process the smoke reports just received. Figure 4-1 illustrates this scenario.

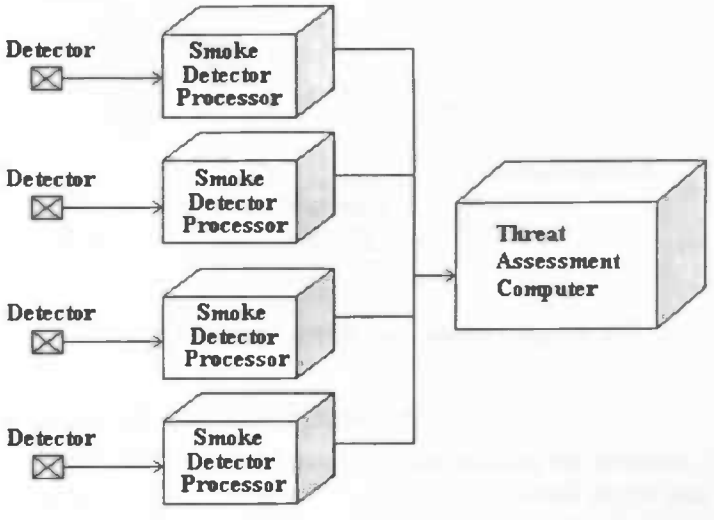

Figure 4-1 Part of an example hardware configuration

The information provided in this brief description is the kind of information we would like to capture in our system models. Therefore, the following requirements were derived:

1. The meta-model shall enable the designer to model the participation of multiple processing nodes in the system. It shall be possible to extend the meta-model to add node-specific properties (e.g. processing speed).

2. The meta-model shall enable the designer to model the physical or logical network connections interconnecting the nodes in the system. This meta-model shall also be extensible with respect to network properties (e.g. maximum throughput).
3. The meta-model shall enable the designer to model the coarse-grained software entities that constitute the software system. The collection consisting of all these entities shall provide a complete coarse-grained overview of the software system. Again, the meta-model shall be extensible with respect to software entity-specific properties (e.g. required run-time memory).
4. The meta-model shall enable the designer to provide details about the inner-workings of coarse-grained software entities by specifying finer-grained sub-entities. Such sub-entities shall represent a specific type of processing (e.g. an algorithm for processing smoke measurements). The meta-model shall also be extensible with respect to sub-entity specific properties (e.g. run-time behaviour with respect to input-size).
5. The meta-model shall enable the designer to define the collaborations between sub-entities, by specifying both the messages being sent between sub-entities and the required sequencing of such messages. The meta-model shall be extensible with respect to dataflow-specific properties (e.g. average message-size).
6. The meta-model shall enable the designer to specify how the software (sub-)entities are mapped onto the available processing nodes. The meta-model shall also allow the designer to specify how dataflow is mapped onto the available network connections.
7. The meta-model shall be suitable for implementation in a software tool (e.g. by defining a UML-mapping of the meta-model).

4.2 The Software System Meta-model

Now that we have defined exactly what the meta-model should be able to express, we need to determine how to represent the modelling elements required to fulfil the requirements. For a better understanding by the reader, we start by providing an overview of the completed software system meta-model. After that, we will argument how this meta-model was created. The UML class diagram in Figure 4-2 shows the meta-model we develop in the rest of this section.

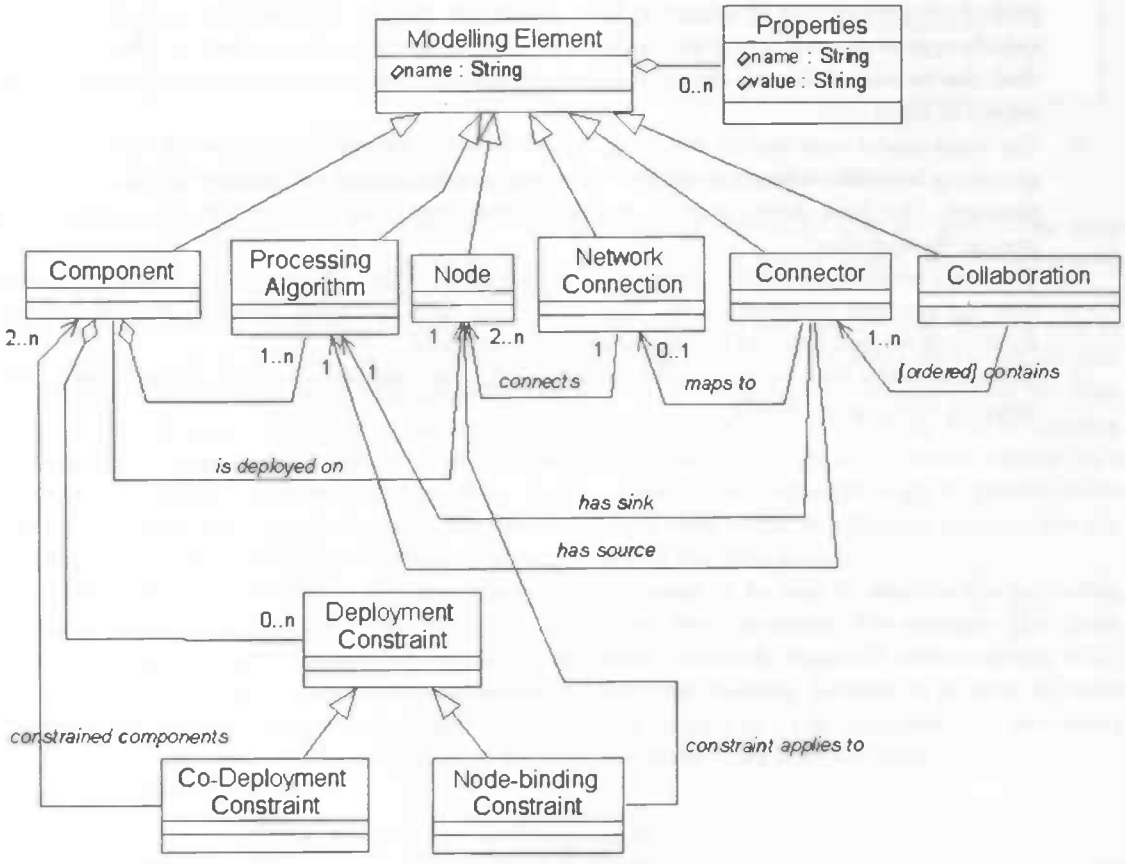

Figure 4-2 Software system meta-model (diagram 1/2)

4.2.1 Modelling the Hardware Configuration

To start simply, we need to fulfil the first requirement of representing the presence of multiple processing nodes. First, a processing node will be defined as any physical or virtual device that is able to execute a computer program. Obviously, a software system will always need at least one processing node. Each node will be assigned a unique name, which will allow us to refer to the node in an unambiguous way. Finally, a processing node may display a range of properties depending on what system characteristics need to be modelled. These specific properties our outside the scope of this chapter and will be discussed in chapter 6.

If two processing nodes wish to communicate, they need to be interconnected by a network connection. Requirement 2 specifies that the meta-model should include these network connections. Note that a network connection is not necessarily a dedicated connection between two nodes: In most cases it is based on a "shared bus-architecture". Our meta-model will thus define networks and allow nodes to be "attached" to that network. Some more constraints are applicable: If a network connection is to be useful, at least two nodes need to be attached. Therefore, a valid model will always have at least two nodes attached to a network (ad-hoc networking is thus not supported).

Just like the node, a network will have a unique name as well. Figure 4-3 provides an example hardware configuration consisting of four nodes, attached to two separate networks.

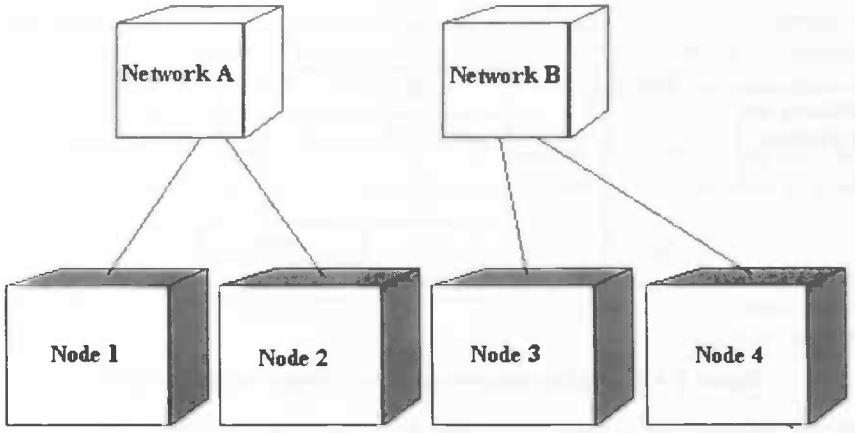

Figure 4-3 Example hardware configuration

4.2.2 Modelling the Software Entities

Now that our physical environment is established, we need to model the software entities that reside in it. Requirement 3 states that we should define coarse-grained software entities: these will be referred to as "components" in the rest of this document. The term "component" is used a lot in software engineering, yet we choose to use the definition provided by the OMG in [Omg03]:

> *A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component is typically specified by one or more classifiers that reside on the component. A subset of these classifiers explicitly define the component's external interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component. A component may be implemented by one or more artefacts, such as binary, executable, or script files. A component may be deployed on a node.*

From this definition, it follows that a component is solely an encapsulating entity, which contains finer-grained artefacts. In order to describe a system's behaviour we shall need to describe these artefacts.
On the source-code level, such artefacts might be composed of classes, subroutines or other logical units of code. However, we are mainly concerned with system-level modelling, meaning that much of the code (or even the detailed design) might not yet exist at this stage. Therefore, we would like to enable the designer to model the system without knowing such specifics.
Our solution is to abstract from objects and other such fine-grained constructs. Note that, given a component's adherence to its interfaces, we need not be concerned with its actual internal structure. We know that all interaction with the component will (by definition) involve its interfaces. Therefore, for our interaction-modelling purposes, modelling a component's interfaces would be sufficient.
Chapter 6, however, addresses the modelling of run-time behaviour in terms of execution time and other such properties. From this perspective, it would not be sufficient to model interfaces alone.
Our solution is to introduce the "processing algorithm" (or simply "algorithm"), which represents one or more run-time artefacts that reside on the component and collaborate to implement some functionality. Such an algorithm might contain part of the component's interface if it interacts with artefacts outside the component. If however, it only interacts with other artefacts on the same component it will not contain any part of the component's interface. Figure 4-4 illustrates the concepts discussed above.

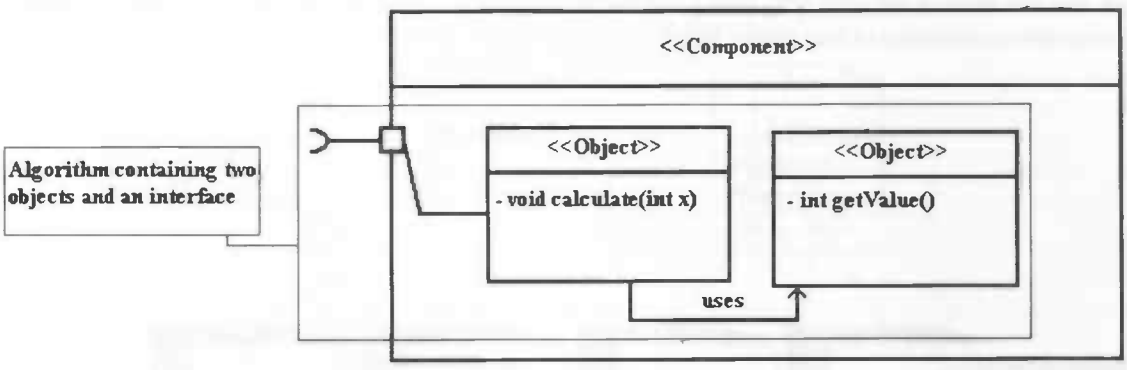

Figure 4-4 Algorithm consisting of two objects and an interface

It follows from the definition of a component, that all processing algorithms that are part of the same component shall be deployed on the same node. Since a component will always require processing capacity to run, it should be assigned to one of the available processing nodes.

Finally, it is important to realize that whereas object-oriented modelling provides different terms to refer to either a run-time instance (object) or its design-time source-code representation (class), this is not the case for components. Therefore, we should explicitly define whether we refer to a component as run-time instance or source-code representation. In our system model, we are obviously dealing with a run-time configuration of components. Hence, we shall use the term component to refer to a run-time instance of a component. For completeness, several instances of the same source-code component might be present in the system at the same time (e.g. multiple smoke detectors might exist in the fire fighting system). For our purposes however, there is no need to include this relationship in the meta-model.

Further elaborating on the details of a component, we also want to be able to describe the processing algorithms themselves, since each algorithm may display its own behaviour. For example, the threat evaluation component may contain a sub-routine for processing the smoke detector's measurements, while it has another sub-routine for processing measurements from the chemical detector. It is these different behaviours that we will want to analyse in a later stage (see chapter 6). The containment of processing algorithms in components, which are themselves mapped onto nodes is depicted in Figure 4-5.

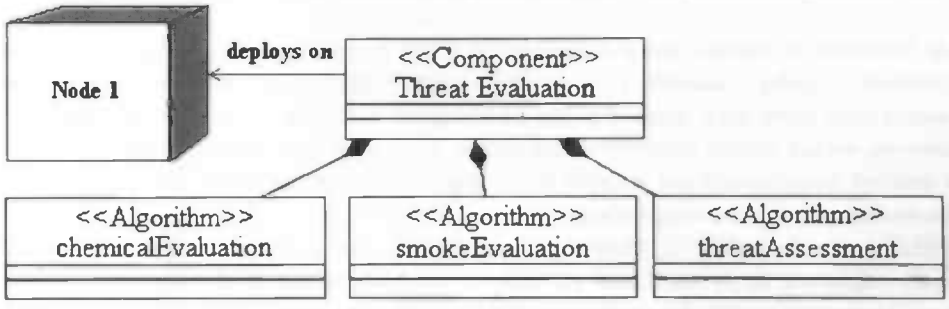

Figure 4-5 Example algorithms contained in a deployed component

4.2.3 Modelling the Software Behaviour

We are now left with modelling the data sent between processing algorithms. First, note that processing algorithms may both "produce" data and "consume" data produced by other processing algorithms. For convenience, we will assume that each processing algorithm at least produces or consumes data, since otherwise it would be isolated and not constitute an interacting part of the modelled system.

It follows that one algorithm may be a producer of a particular kind of data, whereas another one may consume this data. This relationship implies a dependency from the consumer on the producer and is manifested in the system by the flow of data. This kind of relationship will be referred to as a "connector".

For example, the smoke measurements are produced by the smoke detector and consumed by the threat evaluator. Therefore the smoke measurements are considered to be a connector connecting a processing algorithm within the smoke detector-component to one within the threat evaluation-component.
Formally, connectors always connect exactly two processing algorithms, one being the source and the other being the sink. Data always flows from source to sink, so two-way connectors are not allowed. Instead, use two one-way connectors when modelling a two-way connection.
Algorithms, on the other hand, are not restricted to using just one connector: They may be connected to many input- and output connectors, where each connector represents a connection to exactly one other algorithm.
Figure 4-6 illustrates how a connector might connect two interacting algorithms.

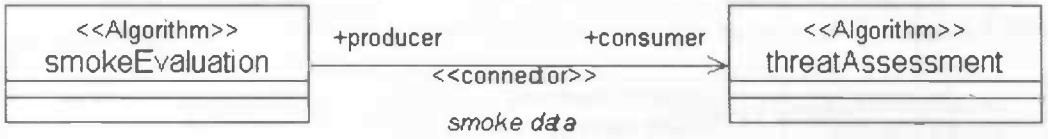

Figure 4-6 Example of a connector connecting two algorithms

Whereas connectors represent data being sent between two algorithms, in many cases we would like to model more complex interactions in which several messages are required to be sent in a predefined sequence between two or more processing algorithms. To include these complex interactions into the system meta-model, we will introduce the "collaboration". A collaboration consists of an ordered sequence of connectors, which together represent part of the system behaviour. All collaborations contained in a system model should define the complete system behaviour (with respect to dataflow between algorithms). Please note that one connector may occur in more than one collaboration, yet this will always represent the same logical connection between two algorithms.

4.2.4 Modelling the Deployment of Software Entities

In the paragraph concerning components, we mentioned that a component is a unit of deployment in that all its parts should always be deployed on a single node. To finalize our model of a component, we need to take one additional concern into consideration: Some components may be subject to special deployment constraints. A component might be required to be deployed on a specific (group of) node(s) (e.g. a computationally intensive component might be assigned to a high-performance node), whereas another component might be subject to a restriction specifying that it should never be deployed on a specific (group of) node(s). In the same way, two components might have similar deployment constraints, specifying that they should always/never be co-located on the same node (e.g. a redundant component should never be deployed on the same node as its original, since this would defeat the redundancy purpose). Figure 4-7 illustrates the concepts of deployment constraints.

Whereas components are deployed onto nodes, connectors need to be deployed onto the available network connections. This is necessary, since there might exist multiple network connections between any two nodes. When data is sent from one algorithm to another, it is either transmitted over a network connection or it is sent via another mechanism whenever the algorithms reside on the same processing node. It was decided that this should be modelled by letting the connector refer to the network across which it is transmitted. If the data is sent within the processing node, the network-reference is omitted.

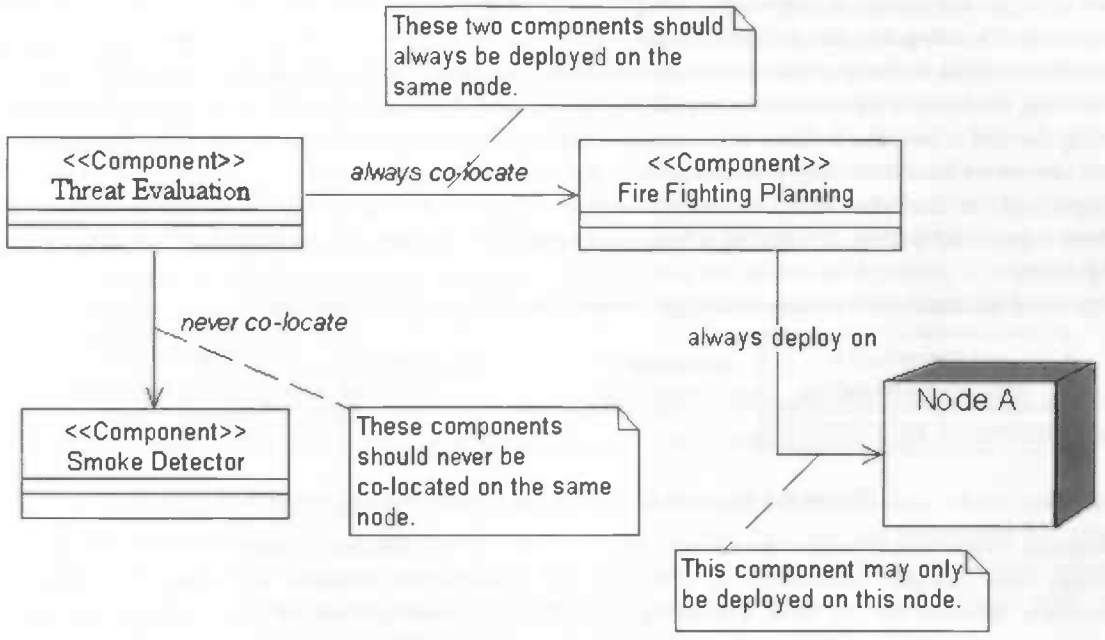

Figure 4-7 Example deployment constraints

4.2.5 Completing the Meta-Model

At this point, we have fulfilled the first six meta-model requirements listed at the beginning of this chapter. The result is the meta-model that was presented at the beginning of this section (Figure 4-2). Yet, this figure does not specify any multiplicity-constraints imposed on the main modelling elements. In order to make the diagram more understandable and to separate concerns, we decided to include these final additions to the meta-model in a separate diagram, which is given below in Figure 4-8. Furthermore, due to the specification in UML, it should be easy to provide software tool-support for this meta-model. This will be demonstrated in chapter 7 and should satisfy the final requirement from section 4.1.

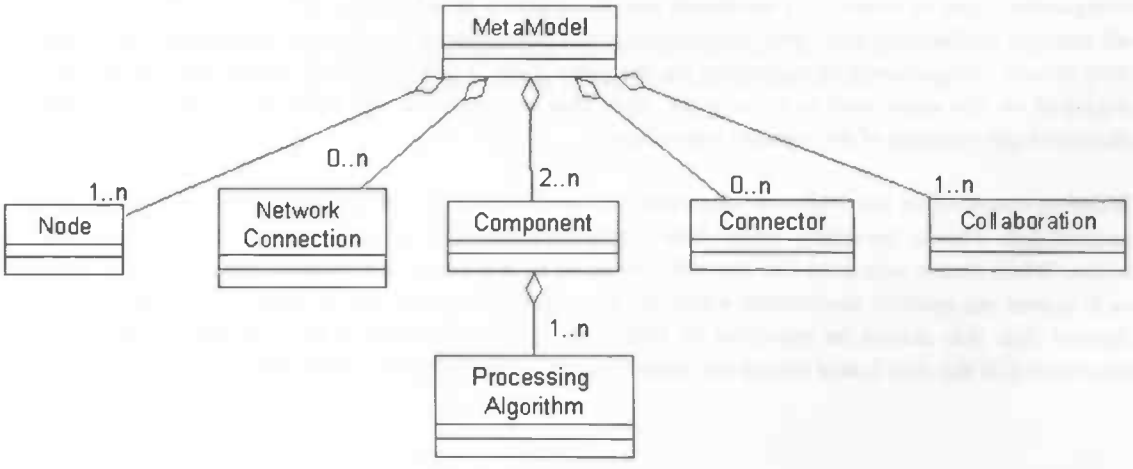

Figure 4-8 Software system meta-model (diagram 2/2)

4.3 Example: A Fire fighting System Software Model

After the rather theoretical process of developing the meta-model, this section provides a hands-on example by using the meta-model to create a system model for the fire fighting system from section 2.5.

First, we represent each of the entities identified in Figure 2-2 by a component. Since we want to keep the example simple (as opposed to realistic), we will include just one smoke detector and one chemical detector to the system.

Each of the components will be assigned a number of processing algorithms, which correspond to the input and output they have to process according to Figure 2-2. Moreover, to make the example slightly more interesting, some components will be assigned additional algorithms connecting some of the other algorithms within the components.

To provide a clear visual representation, a non-UML diagram will first be provided in Figure 4-9, followed by a more formal specification in UML.

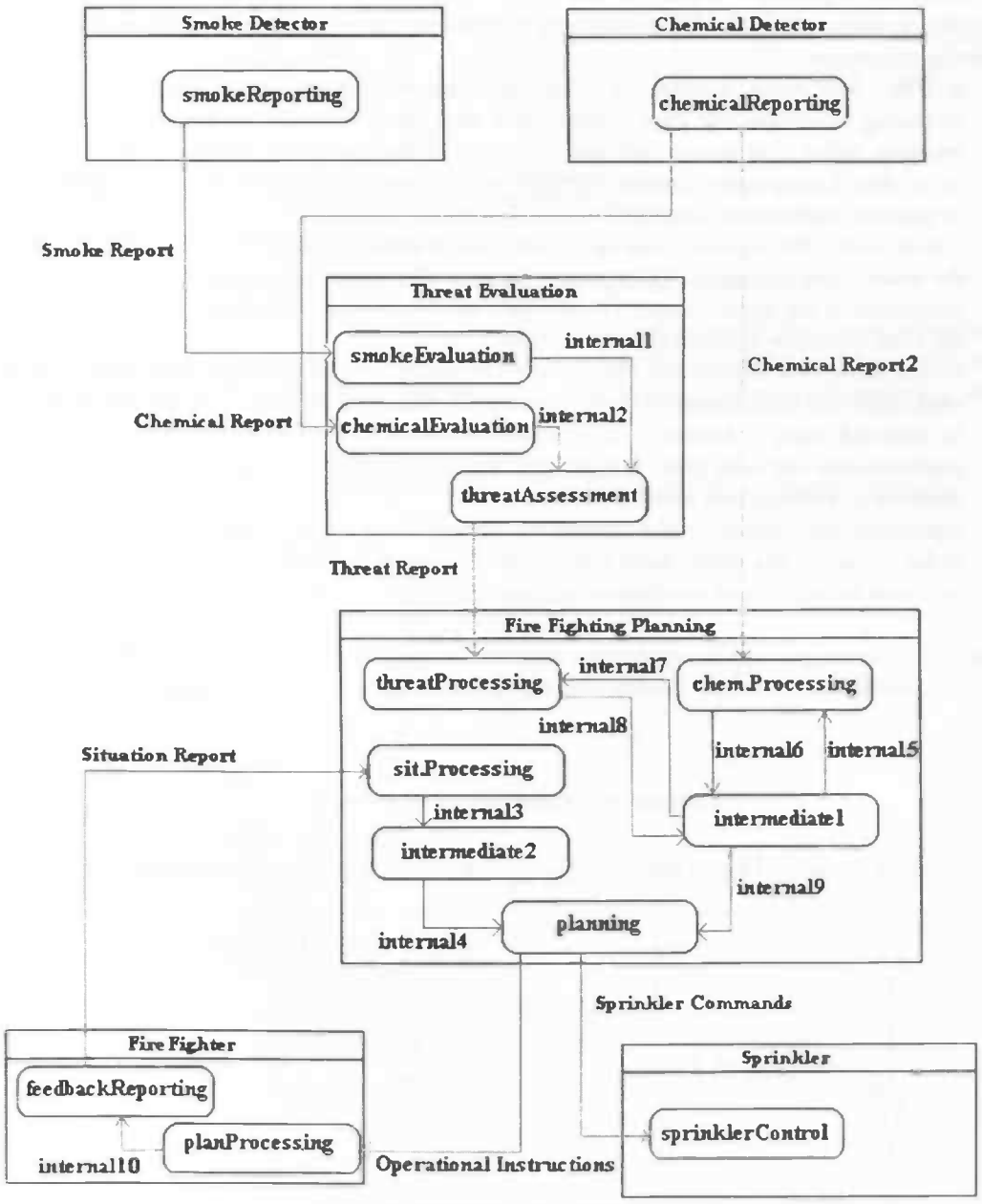

Figure 4-9 Fire fighting system model (logical view)

As can be seen in Figure 4-9, components have been modelled as rectangles, connectors as arrows and algorithms as rounded rectangles. To illustrate the process of interpreting this diagram, consider the following: the smoke detector component uses the *smokeReporting*-algorithm to produce its measurement report, called "smoke report". This smoke report is sent to the threat evaluation component, which processes the report using the *smokeEvaluation*-algorithm. The output produced by *smokeEvaluation* is handed to the *threatAssessment*-algorithm, which also receives the processed chemical reports. *threatAssessment* then produces a threat report, which is sent to the fire fighting planning component, where the *threatProcessing*-algorithm receives and processes the reports. Algorithm *intermediate1* fuses the processed threat reports with some elements of the processed chemical reports and enables the *planning*-algorithm to create a fire fighting plan that involves the sprinklers and/or fire fighters. The *planning*-algorithm then sends control-messages to these actuators.

Since it would not be possible to capture all this system information in one single UML-diagram. Rather, we have decided to use a collection of UML-diagrams to represent our system models. Each model consists of the following diagrams:

1. A UML deployment diagram, in which all component-instances are listed, along with the processing algorithms that they contain. Note once more, that it is our aim to describe run-time instances rather than source-code entities, which is the reason we choose to use a deployment rather than a component-diagram. [Omg03] refers to such deployment diagrams without nodes as "degenerate deployment diagrams".
2. One or more UML sequence diagrams, where each sequence diagram describes one collaboration in the system (see paragraph 4.2.3). In the process, all collaborations together also define all the connectors in the system. Using a connector's unique name, it will always be possible to identify the same connector in different collaborations.
3. One or more UML deployment diagrams are also used to specify the deployment from components onto nodes and from connectors onto networks. In this case, the processing algorithms should not be included, since it follows from previous definitions that each algorithm in a component is deployed onto the same node. Finally, this diagram should also define all the networks that are available as well as which nodes they connect.
4. Optionally, the designer might include one or more (degenerate) UML deployment diagrams in order to specify the deployment constraints the system is subject to. These constraints may be indicated by stereotyped associations as illustrated in Figure 4-20 later in this section.

The rest of this section provides the appropriate diagrams for the fire fighting system described earlier. Figure 4-10 provides a deployment diagram defining all the software entities in the system.

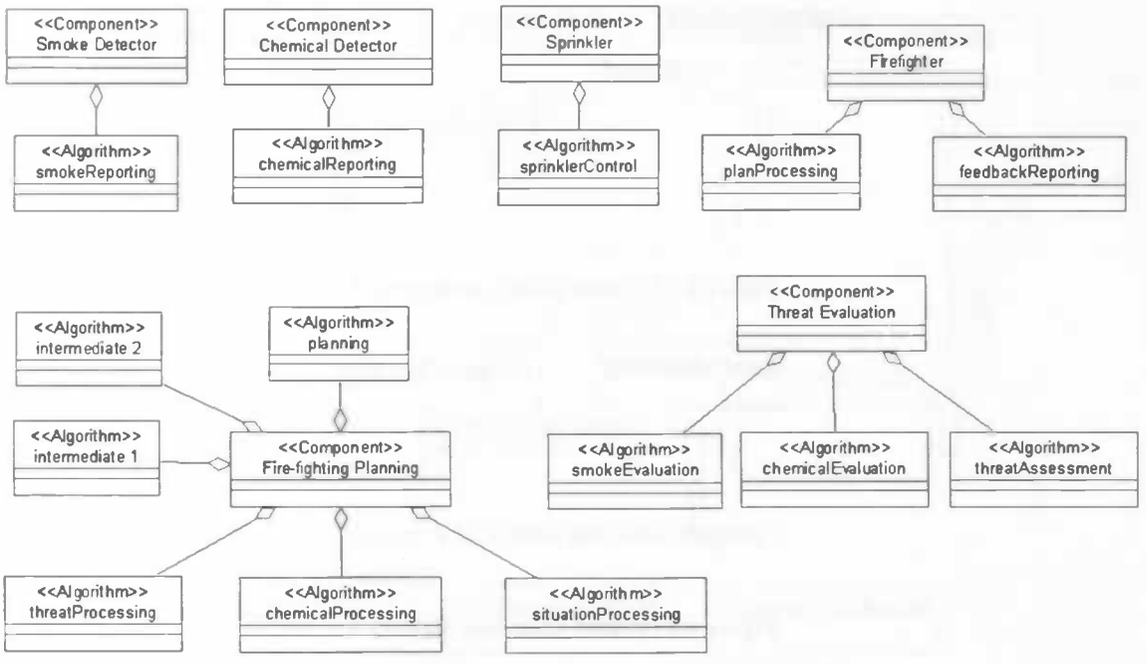

Figure 4-10 UML logical view of fire fighting system model

Now that all software entities have been identified, we shall need to model the system behaviour using multiple UML sequence diagrams. Note that the interaction specifics contained in the sequence diagrams were chosen arbitrarily. Thus, whether a particular interaction follows a request-reply pattern or any other mechanism is arbitrary at this stage.

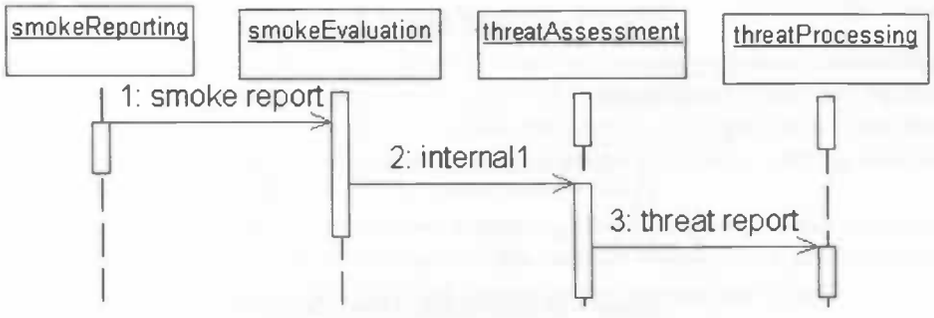

Figure 4-11 System behaviour diagram 1

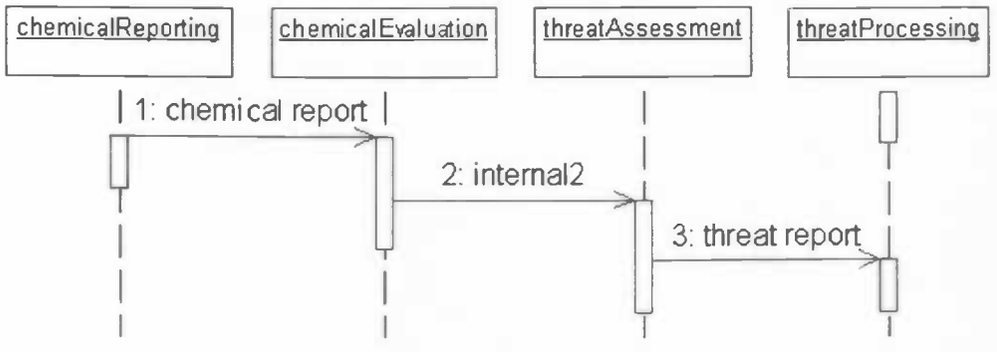

Figure 4-12 System behaviour diagram 2

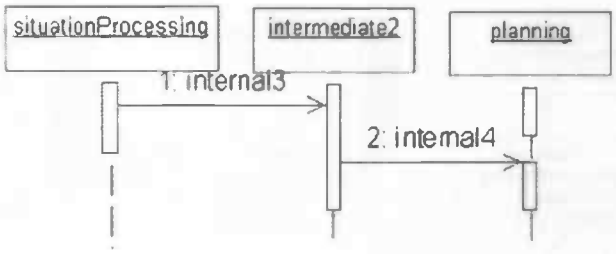

Figure 4-13 System behaviour diagram 3

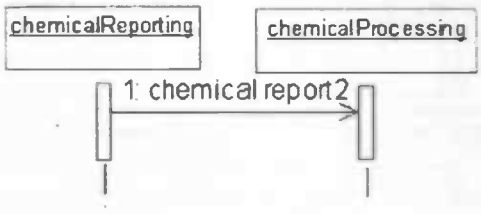

Figure 4-14 System behaviour diagram 4

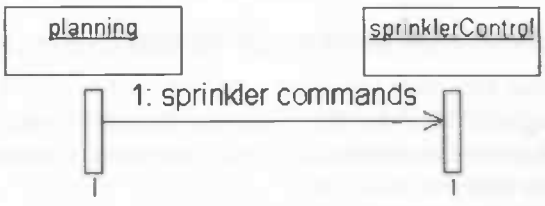

Figure 4-15 System behaviour diagram 5

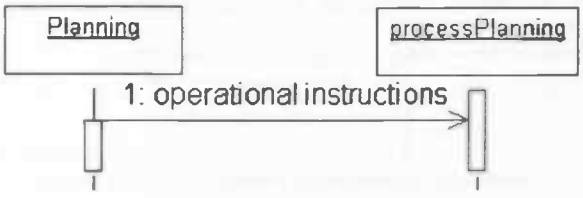

Figure 4-16 System behaviour diagram 6

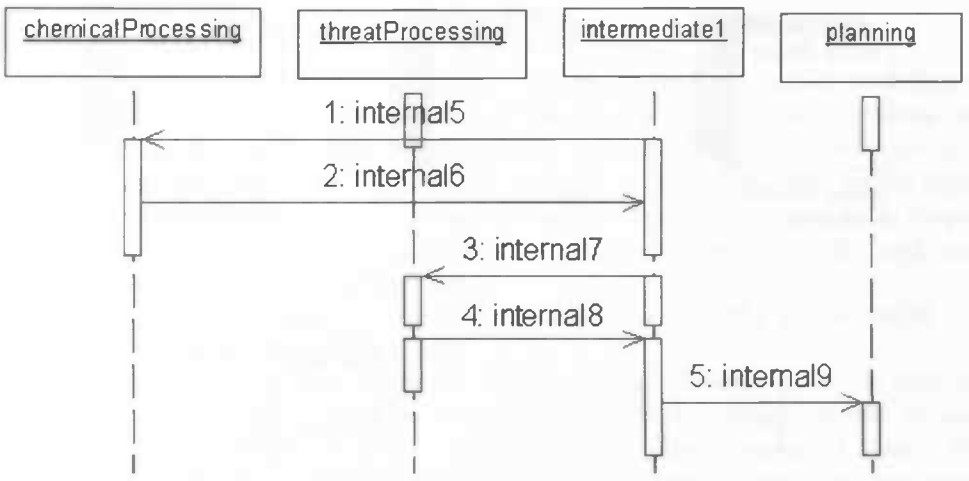

Figure 4-17 System behaviour diagram 7

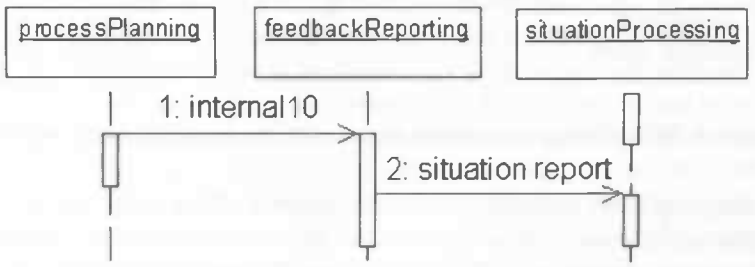

Figure 4-18 System behaviour diagram 8

For completeness, we shall provide a brief explanation of what these sequence diagrams express: Figure 4-11 and Figure 4-12 specify that data sent by both the chemical and smoke detectors is processed by the threat assessment component as soon as it receives the new input. Moreover, it will always send a new threat assessment to the fire fighting planning component as well (i.e. it does not wait for a timing event or for both measurements to become available).
Figure 4-13 through Figure 4-16 should be fairly self-explaining, whereas Figure 4-17 describes a more complex collaboration in which algorithm *intermediate1* first requests chemical data, after which it will request threat data in order to send a newly compiled message to the planning algorithm.
Figure 4-18 should be fairly straightforward.

We continue by providing a deployment diagram, in which all components are mapped onto nodes and all connectors are mapped onto networks.

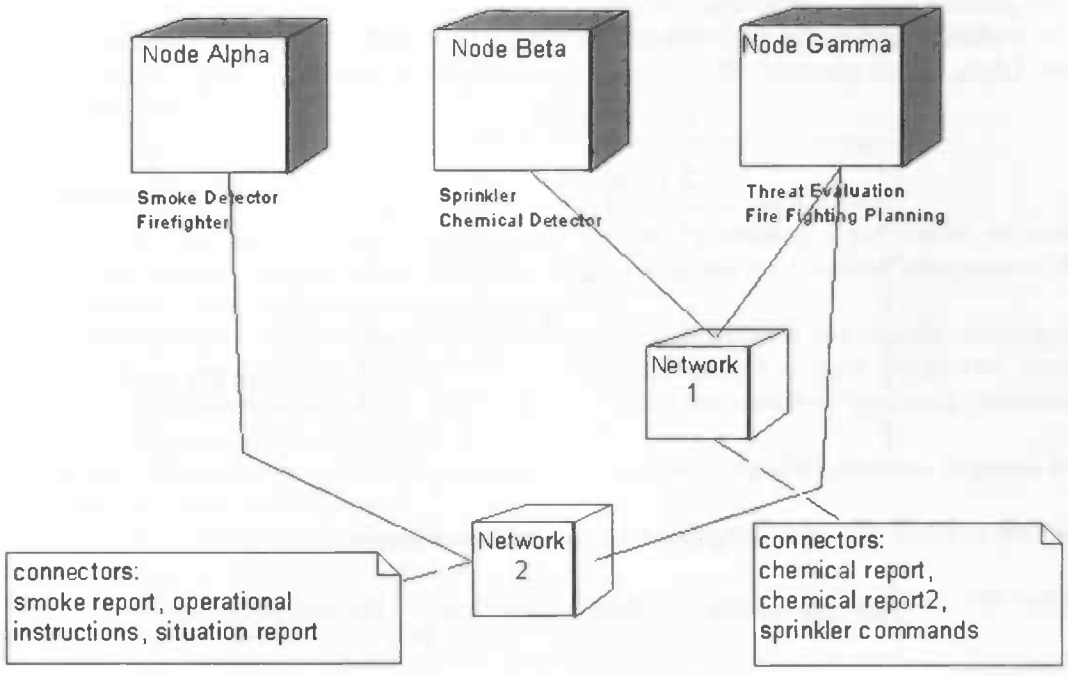

Figure 4-19 Deploying components onto nodes and connectors onto networks

Note that the mapping from components onto nodes is arbitrary in our example. It is easy to see that nodes Alpha and Gamma are now connected by network connection 2, whereas Beta and Gamma are connected by network connection 1. This implies that Alpha and Beta cannot directly communicate with one another: As far as our model is concerned, there is no network path between the two nodes.

To complete the system model, we shall also provide one deployment constraint in Figure 4-20.

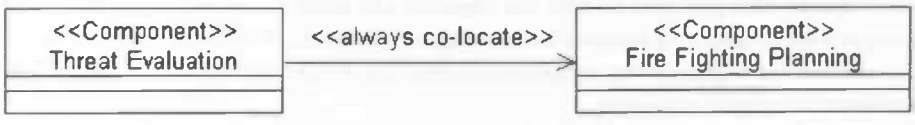

Figure 4-20 Co-location deployment constraint

The system model we have now created demonstrates the capabilities of the meta-model we developed in this chapter. Furthermore, this design will be used as the basis for the subsequent chapters.

4.4 Tool-support for Software System Modelling

During this research we developed a software tool that implements the system modelling processes described in this thesis. A more elaborate description of the tool is given in chapter 7, whereas this section solely describes how the concepts introduced in this chapter were implemented in the software tool.

The system modelling meta-model that was provided in section 4.2 formed the basis for the logical model underlying the software application. Although the logical model will be elaborated on in chapter 7, one might say that its implementation is a straightforward translation from the UML-based meta-model presented in this chapter (Figure 4-2 and Figure 4-8).

More importantly, the tool provides the user with several views upon the system design. The views correspond to the four diagrams listed in section 4.3:

1. The first view allows the user to define the components and processing algorithms contained in the design. At the time, this is done by selecting the appropriate design element (component or algorithm) from a palette of design elements. In the future however, a library of existing components may be integrated into the tool. This would allow the user to browse through these existing components and insert them into the design, facilitating the creation of a new design based on existing components (reuse).
2. The second view allows the user to specify the system's behaviour by drawing UML sequence diagrams (as demonstrated in section 4.3). The user selects some of the previously defined processing algorithms from a list of available algorithms, which automatically adds them to the diagram. The user may then define a collaboration by drawing associations between the algorithm-instances. The application automatically updates its internal logical model of the system.
3. The third view consists of a deployment diagram in which the user may define the nodes and networks that constitute the hardware configuration. Moreover, he may deploy components onto these nodes and connectors onto network connections. The nodes and networks are inserted by selecting these modelling elements from a palette of available modelling elements. Once a node- or network-instance is selected, component- or connector-instances may be deployed onto it using a context-menu.
4. A fourth diagram allows the user to specify deployment constraints. These can be used by the application to determine whether a deployment configuration is valid. Alternatively, the constraints might serve as input for a tool that automatically determines a viable deployment configuration. This however, is not the aim of our application. The application allows the user to select either components or nodes and to draw constraint-associations between them.

Figure 4-21 shows how the user defines the software entities in the first view.

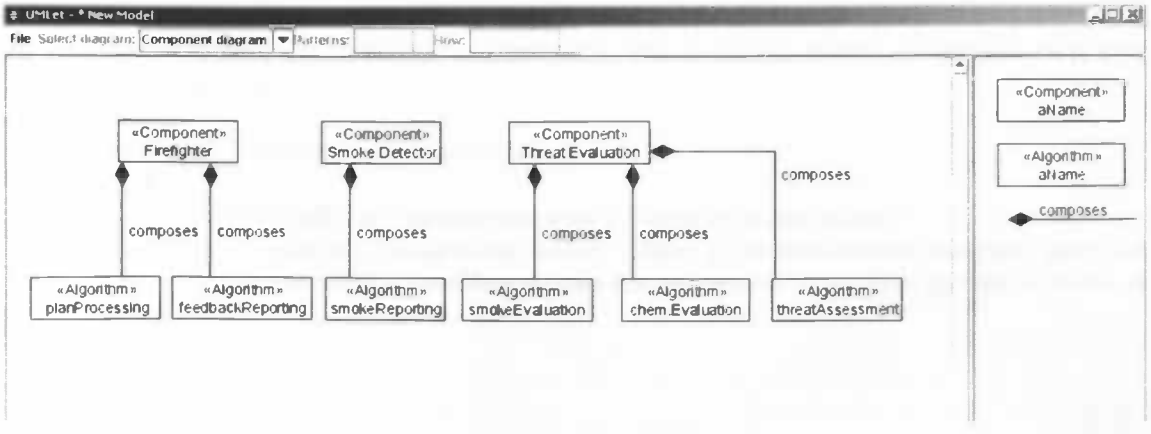

Figure 4-21 Screenshot of software system modelling tool

Finally, the software system model contains deployment constraint information. This information could be used for automatic generation of a deployment configuration. Since this is not a primary focus of this research it has not been implemented in our verification tool. In future versions though, such a feature could be implemented.

4.5 Summary

This chapter provided the foundation for subsequent chapters by defining a meta-model for creating software system models. The meta-model allows the designer to define the hardware configuration of the system, by including nodes and networks into the design.
The software entities that are contained in the system design are modelled at two separate, course-grained levels: Components represent modular, replaceable and deployable parts of a system that provide interfaces in order to communicate to other software entities. We introduced the concept of "processing algorithms" to abstract from component implementation details.
The behaviour of the software entities in the system was modelled using UML sequence diagrams, which each defined a so-called "collaboration".
We also provided a mechanism for describing the deployment of components onto nodes as well as the deployment of logical connectors onto network connections.
Finally, we described how we included the process described in this chapter into the software tool that we developed during this research. This also satisfies all requirements listed in section 4.1.

5 Modelling Component Interaction Using Patterns

The previous chapter provides us with a process for modelling software systems. However, one of the main goals of this project is to express the presence of patterns in such software designs. The first problem we face when trying to do this, is that there does not currently exist any consensus on how to express patterns, let alone express them in a design-context. This is the reason for dedicating this chapter to the problems of modelling patterns and expressing their presence in designs.

Section 5.1 will start by introducing a well-known pattern that will serve as an example throughout the rest of this chapter. After that, we will have all the prerequisite knowledge to start developing our own pattern meta-model in section 5.3. We continue by demonstrating the use of the meta-model in creating a model of our example pattern.

In section 5.5, we show how the pattern meta-model can be integrated with the previously developed system meta-model in order to create a meta-model for pattern-based software system designs. We also demonstrate the use of the meta-model as well as a way of visually expressing the presence of patterns in designs.

5.1 The Observer Pattern

Discussing an abstract topic like patterns will be much easier with a simple example pattern. In this section, the *Observer*-pattern is introduced. The Observer-pattern is widely used and taught nowadays. Although there exist several flavours of this pattern, this section describes the version published by Gamma et al. using the definition from their book [Gam95]:

Intent

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

Also known as

Publish-subscribe

Motivation

A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects. You don't want to achieve consistency by making the classes tightly coupled, because that reduces their reusability. [...]

Applicability

Use the Observer pattern in any of the following situations:

- When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
- When a change to one object requires changing others, and you don't know how many objects need to be changed.
- When an object should be able to notify other objects without making assumptions about who these objects are.

Participants

Subject, Observer, ConcreteSubject, ConcreteObserver

Collaborations

- ConcreteSubject notifies its observers whenever a change in its state occurs.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information. ConcreteObserver uses this information to reconcile its state with that of the subject.

Structure

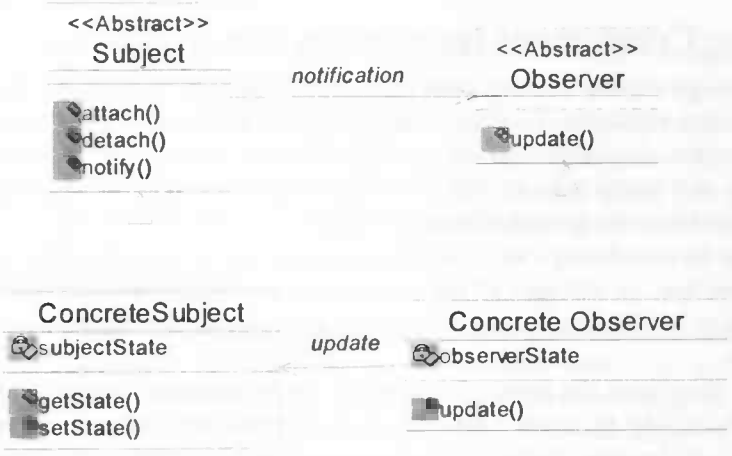

Figure 5-1 The Observer Pattern [Gam95]

5.2 Analysis and Requirements Definition

Since patterns are a very broad concept, covering many aspects, it would be difficult to simply provide "a process for representing patterns". As was discussed in chapter 3, Buschmann et al. introduced the concept of architectural patterns that describe solutions to design problems at the architectural level. In the case of architectural patterns however, we are mainly dealing with large, coarse-grained building blocks that need to be organised into a structurally coherent system.

In the case of design patterns, we are mainly concerned with individual classes and objects, which are very fine-grained entities compared to the architectural building blocks. Moreover, design patterns cover all design aspects from structuring a system to defining its behaviour.

In order to capture all of these aspects, a pattern meta-model would have to be usable with entities of any granularity and define their behavioural, life cycle and structural characteristics. This list is not necessarily complete, but it should be clear that it would be difficult to develop one monolithic meta-model that is able to capture all of these aspects.

Therefore, we decided to opt for the classic divide-and-conquer approach (note that divide-and-conquer is itself a problem-solving pattern), choosing to first model a specific subset of these pattern aspects.

Since we are concerned with system-level modelling, we have opted not to model patterns at the fine-grained class-level (design patterns), but rather to limit ourselves to coarse-grained architectural patterns as introduced by Buschmann et al. [Bus96].

Yet, we still need to focus more precisely before we can specify what kind of patterns our pattern meta-model should be able to represent. The pattern-classification criteria from [Gam95] will be used as a guideline: According to these criteria there exist structural-, behavioural- and creational patterns. Starting with creational patterns, it would not make much sense to model these at the architectural level, since coarse-grained software entities tend to have a very long life cycle. They are mostly created only once, when the system is started and are destroyed only when the system is halted.

Structural patterns are a very good candidate for modelling at the architectural level, since one of the main issues in developing a new software system is structuring the coarse-grained entities appropriately. Moreover, they are discussed in detail in [Bus96] and [Sch00].

Finally, we could choose to model behavioural patterns at the architectural level. Although, to the best of our knowledge, this has not been widely addressed in any previous research, it would be interesting and useful to examine behavioural aspects at this level.

To determine which of the above options to choose, it was examined what the influence on the rest of this thesis would be: With respect to calculating and verifying performance characteristics of a software design, structural patterns are definitely important for overall performance, but behaviour needs to be defined explicitly first if we are to be able to perform any calculations.

Therefore, behavioural patterns at the architectural level will be the subject of research in the remainder of this chapter. This is depicted in Figure 5-2, where we have listed the classification from Gamma et al. horizontally and the one from Buschmann et al. vertically. The red cross marks our area of interest.

	Behavioural pattern	Structural pattern	Creational pattern
Architectural pattern			
Design pattern			

Figure 5-2 Focussing the research area

Now that we have defined the specific area our research will be aimed at, it should be possible to specify the requirements that the pattern meta-model should fulfil. In order to determine what elements a good pattern-description should contain, let us once again look at the work of Gamma et al, which has been the basis for virtually every pattern meta-model.

- Gamma et al. state that every pattern should have a clear name that will quickly remind the designer of its use.
- The intent, motivation and applicability of a pattern provide the designer with important context-depended information. Yet, this information is unstructured and not machine-readable. The nature of this information makes it impossible to capture it in a machine-readable way.
- The structure of an algorithm is relatively easy to capture using machine-readable representations, so our meta-model should definitely contain a structural component.
- Identifying the participants in a pattern is also easily accomplished and will have to be possible in our meta-model.
- For us, the most important part of a pattern description will probably be the part detailing the collaboration, since it may allow us to automatically generate the dataflow between entities in our software design. This is where the focus of this pattern meta-model should be.
- One aspect of patterns is not part of the model used by Gamma et al: each pattern may exist in several variants. Riehle discusses this issue [Rie96A] and suggests a partial solution for this problem. It would be a great gain, if our meta-model would be able to express the degrees of variability of a pattern.

The following requirements to our pattern meta-model follow from the above:

1. The pattern meta-model shall provide patterns with a unique name that clearly identifies the specific pattern.
2. The pattern meta-model shall provide the designer with information about how to map the pattern onto a logical software design.
3. The pattern meta-model shall identify the participants in the pattern.
4. The pattern meta-model shall be able to specify the collaborative behaviour of the participants in the pattern.
5. The pattern meta-model shall provide some degrees of freedom that will allow different variants of a pattern to be described by the same model.
6. The pattern meta-model shall be integrated with the software system meta-model from chapter 4 in order to form a pattern-based software system meta-model.
7. The pattern-based software system meta-model mentioned in requirement 6 shall allow patterns to be integrated with existing software system designs as described in chapter 4. Moreover, it shall enable the designer to describe collaborations between components at the architectural level.
8. A visual representation for integrating patterns into software system designs shall be provided.
9. The pattern meta-model and the pattern-based software system meta-model shall be suitable for easy use in automated software tools.

5.3 The Pattern Meta-model

In this section, we develop a process for expressing patterns in a way that is suitable for use in a pattern-based design process. This means that we do not need to develop a new pattern model from scratch. Instead, we will build upon the experience of others and use elements from their pattern expressing schemes.
The requirements in the previous section indicate that the innovative focus with respect to the pattern meta-model will have to be in enabling the representation of several pattern-variants in the same model (i.e. we have to build in degrees of freedom). This will be the guideline in the rest of this section.

5.3.1 Participants & Structure

From Gamma et al. we know that a behavioural pattern consists of several participants that collaborate in order to achieve a common goal. These participants will somehow have to be mapped onto design-time entities (e.g. classes) when we want to apply the pattern to a software design. In the original work of Gamma et al, the list of participants was tightly coupled to the structural design of the pattern, since the participants were represented by classes in the structural representation of the pattern. As Riehle and others pointed out, this means that only one particular instantiation of a more general pattern is described.
To solve this limitation, they suggest omitting classes and objects from the pattern meta-model. Rather than specifying all classes participating in a pattern, Riehle uses "roles" to model pattern participants. The difference is quite subtle, since a role may eventually correspond to a class in the software design, yet no such assumption is made within the pattern model. One class might just as well implement the roles of multiple participants: this decision is left to the designer.
For example: Consider the description of the Observer pattern from section 5.1. Gamma et al. describe the participants as four classes: ConcreteObserver, ConcreteSubject, Observer and Subject. A role-based pattern-model might list two participating roles: Observer and Subject. The software design might however, contain an entity called "Monolith" that implements both the Observer- and the Subject-role.

Adopting roles rather than classes in our pattern meta-model contributes to the goal of creating some degrees of freedom in representing patterns. Moreover, since roles may be mapped onto structural entities, using roles cancels the need for a structural element in our pattern meta-model.
Thus, participants will be represented by roles, whereas structural designs will be omitted.
In order for a pattern-model to be useful, it should contain at least two roles. Otherwise, no collaborative behaviour can be displayed. Figure 5-3 shows the pattern-role relationship for the aforementioned Observer pattern.

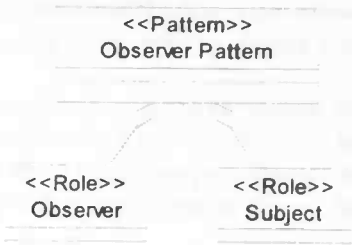

Figure 5-3 Pattern-instances contain roles to represent their participants

5.3.2 Collaborative Behaviour

The collaborative behaviour of participants can be specified as an exchange of messages. Since this sounds very abstract, it might be best to illustrate this idea with an example:
Recall the smoke detector and threat evaluator from our case study. The smoke detector collaborates with the evaluator in the simplest possible way: it sends periodic messages containing smoke level measurements.
A more sophisticated collaboration between these two components could be realized using the Observer pattern: The smoke detector might send the evaluator a notification whenever a new smoke measurement

were available. The evaluator might then reply by sending a request to retrieve the smoke report. In response, the smoke detector would send the smoke report. This process is illustrated in Figure 5-4 below.

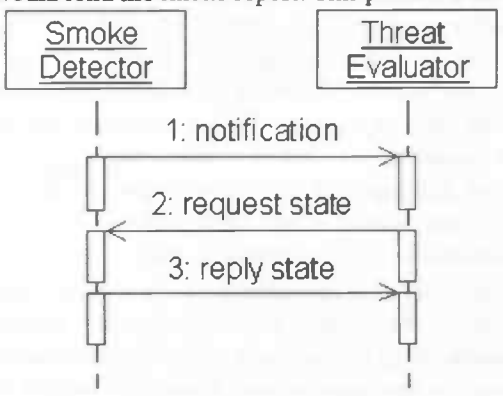

Figure 5-4 Collaboration in the Observer-pattern

As can be seen, this is an example of collaboration between two components. Moreover, it can truly be modelled as an exchange of messages and is easy to represent using UML sequence diagrams. This does not just hold for data-centric communication: Control-centric communication could be modelled just as well, since it is also generally based on exchanging messages to transfer control.

Furthermore, it should be noted that more complex patterns may contain more complex collaborations requiring multiple sequence diagrams each describing a valid "sequence of messages" for the particular pattern. Note that each participant in a pattern could (in principle) initiate a collaboration, so many complex interactions might be described by the same pattern. Multiple sequence diagrams are therefore required.

An example pattern-instance is depicted in Figure 5-5.

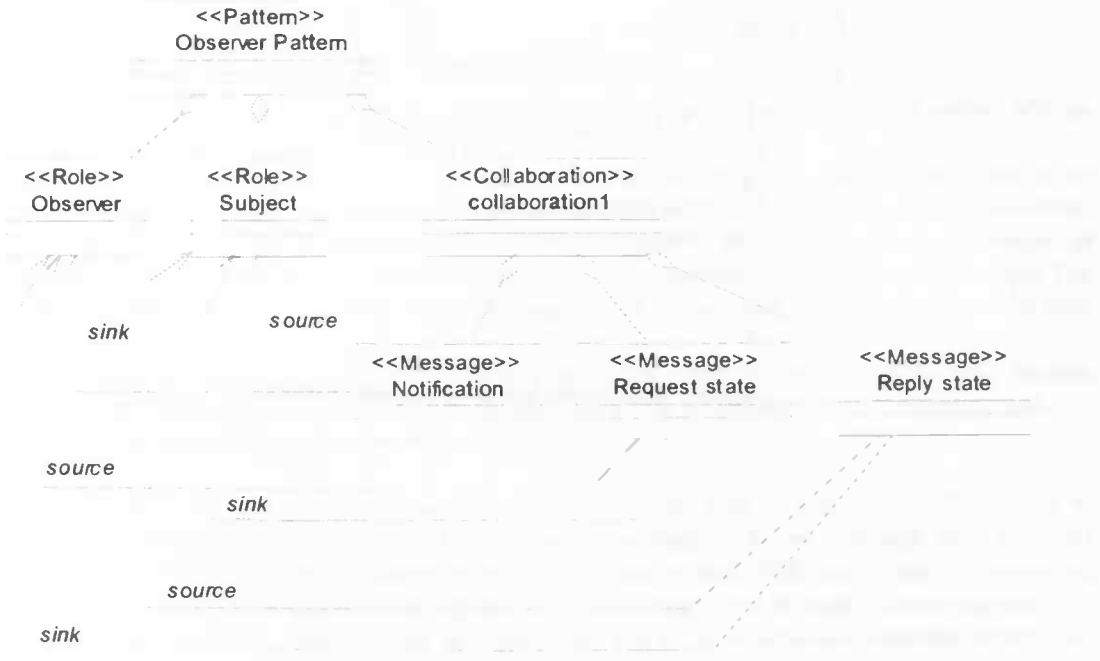

Figure 5-5 Pattern-instance contains collaborations to express valid behaviour

So far, our meta-model is rather flexible in that it allows several valid collaboration-schemes to be captured by a pattern model. Yet, it might be desirable to add another degree of freedom. To illustrate this need, consider (once again) the Observer pattern:

The original Observer pattern from [Gam95] states that the subject should notify its observers when a change in its internal state occurs. The observers should subsequently request this state information. What if the subject (e.g. smoke detector) were to send this state information along with the notification? Would this mean that we would no longer be implementing the Observer pattern? We would certainly still be observing the internal state of the subject. Therefore, it is our opinion that we are just referring to two slightly differing instances of the Observer-pattern. We are convinced that these kind of minor behavioural differences are an important reason for the fact that many variants of a single pattern exist. It would therefore be desirable to capture this degree of variability in our pattern meta-model. Moreover, capturing this variability in our pattern meta-model might be a stimulus for using pattern-based design, since designers would not feel so restricted by the "prescribed" patterns.

We developed a means for describing such variability in our pattern meta-model. The idea is relatively simple: One adds information to the pattern model, indicating whether a particular message in a collaboration is required for maintaining the pattern's integrity. Some messages might be strictly required, while others might be optional. In the example case above, the request for state information would be optional, whereas the notification would be a required part of the pattern. Note that the degree of variability allowed for a particular pattern is mainly in the eye of the beholder and therefore a topic not discussed in this thesis.

We could go one step further still, by also specifying messages that would be absolutely forbidden. For example, if we were to deliberately use a pattern that enables low coupling between the smoke detector and the threat evaluator, we certainly do not want the evaluator to send messages straight to the smoke detector, since it would defeat the goal of low-coupling. Instead, it might send messages through a "proxy" or the designer might have to limit the collaboration to one-way traffic. In either case, the low coupling would have to be made explicit in the low-coupling pattern. This could be done by adding information to the pattern-model, indicating that a particular message is not allowed or, as we shall call it, "disallowed". This situation is depicted in Figure 5-6.

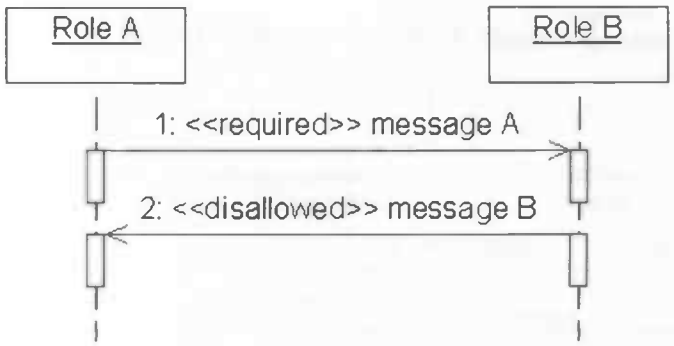

Figure 5-6 Enforcing low coupling with patterns

5.3.3 Summary of the Pattern Meta-model

The previous sections have laid the foundations for our pattern meta-model. This section presents the resulting meta-model using a UML class-diagram representation.

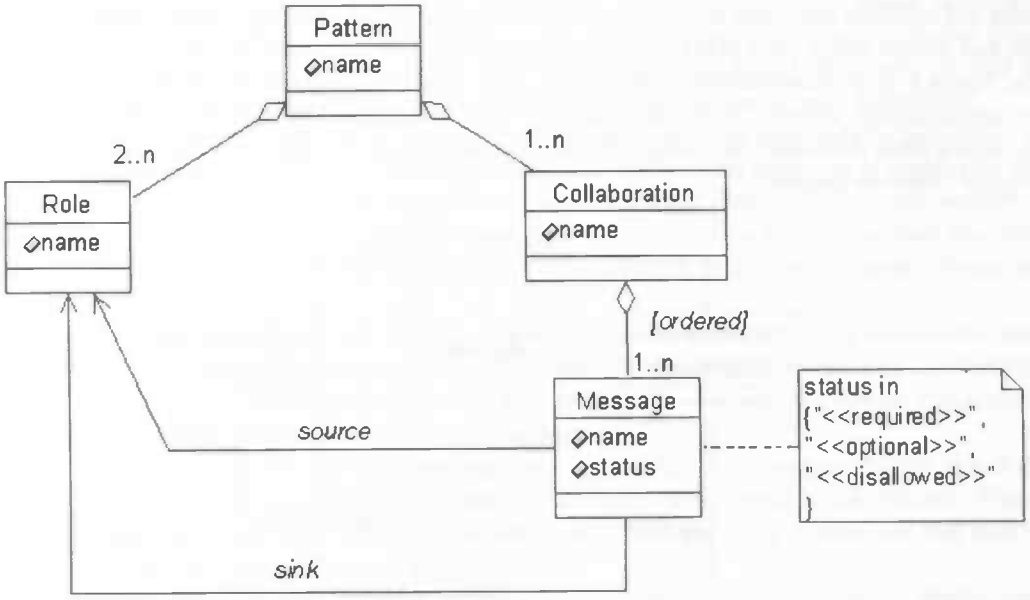

Figure 5-7 The pattern meta-model

5.4 Example: A Model of the Observer Pattern

The previous section discussed the development of a meta-model for pattern modelling. This section will use the meta-model to develop a model of the Observer pattern.

The first step in modelling the Observer pattern would consist of identifying the roles that participate in the pattern. If we look at [Gam95], we can identify two roles in this pattern: *subject* and *observer*. However, [Glaxx] identifies another role that plays a part in the Observer pattern: *notifier*. The idea is that instead of the subject notifying all of its observers, it just notifies the notifier whenever its internal state changes. The notifier then notifies all of the observers of that subject, which in turn may request the subject for state information directly.

This abstraction is, in a sense, cleaner than the original participant abstraction provide by [Gam95], because it allows the notification task to be performed by another entity (e.g. middleware) than the subject itself.

Hence, we identify three roles in the Observer pattern.

The next step will be to identify the messages that the participants send to one another. These may be deduced from the description above and will be provided in the diagrams below. Although there are many ways to express a model that would adhere to our pattern meta-model, UML was chosen for expressing patterns, since it provides the expressiveness required and is well-understood by most software engineers.

The collaboration element of a pattern may be captured by one or more sequence diagrams, where each diagram describes a valid sequence of messages.

Modelling roles is also possible in UML, since UML provides so-called role diagrams. In this case though, it might be wiser to apply UML a bit more creatively. Role diagrams are sufficient if we just want to models the roles participating in a pattern.

If, however, we were to use UML class diagrams, and interpret each class as a role in the pattern description, we could also model the messages in the same diagram. Essentially, it is just another view of the pattern than the one provided by a sequence diagram, yet the class diagram-view provides a much better overview

of which connectors are connected to which roles, whereas the sequence diagram stresses the sequence in which messages are to be sent.
These diagrams complement one another very well and provide a complete overview of the pattern elements we wish to model.
The Observer pattern model is given in Figure 5-8, Figure 5-9 and Figure 5-10 below. Figure 5-8 expresses the roles and relations contained in the Observer-pattern. Figure 5-9 provides a valid collaboration for this pattern and defines which parts of the collaboration are mandatory and which are not.
Finally, Figure 5-10 provides another collaboration. This collaboration however, specifies that a subject can never communicate directly with its observer. This represents the notion of low-coupling, the independence from the subject with respect to the observer, which was one of the fundamental strengths of the original Observer-pattern.

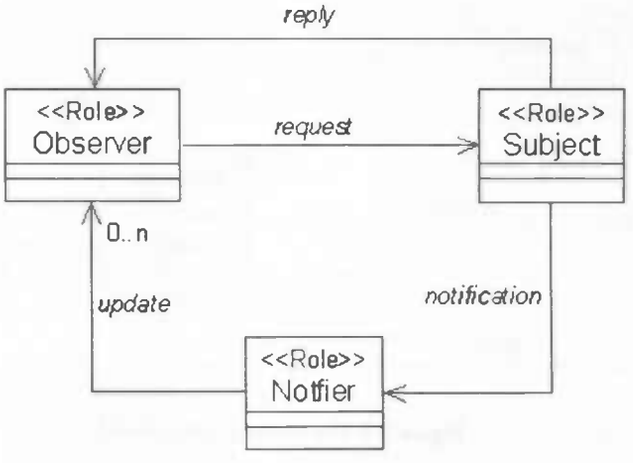

Figure 5-8 The Observer pattern: role view

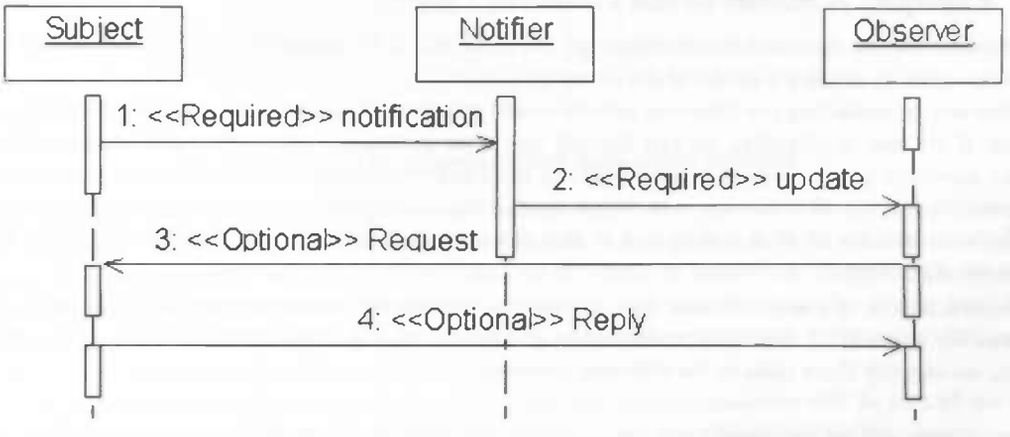

Figure 5-9 The Observer pattern: collaboration view

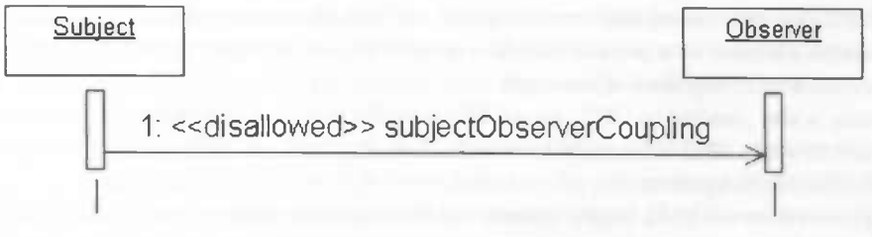

Figure 5-10 Expressing explicit de-coupling in a pattern model

5.5 Designing Software Systems Based on Patterns

In the previous sections, we developed a meta-model for expressing patterns in a stand-alone fashion. The reason we need such a meta-model is that we would like to use patterns as building blocks for creating software designs. In the ideal case we would like to have a tool that has access to a library of pattern definitions, which we could consult for finding appropriate patterns to solve our design problems. We would simply select patterns from the library, tell the tool to apply them to our design and after repeating this process several times we would have a purely pattern-based software design. Moreover, the tool would select a number of suitable patterns based on a description of the design-issue we were addressing.

Reality, however, is quite different. We are a long way from simply selecting and applying patterns in such a way. Although most popular UML-based software engineering tools provide some sort of support for design-patterns, including a pattern-library, this support is still fairly crude and certainly not context-sensitive.

This section explores the activity of applying patterns to a design, whereas the next section describes the necessary extensions to the system meta-model in order to transform it into a pattern-based software system meta-model.

First, we need to consider what it means to apply a pattern to a software design. If we consider the patterns described by Gamma et al, applying them in the purest sense is fairly straightforward: The *structural* part of the pattern-definition contains a class diagram that may be copied to the software design. The participants might be renamed, and the pattern would have been applied.

This approach has several drawbacks: The first was mentioned earlier and concerns the fact that there are many ways of mapping the participants in a pattern onto design-time artefacts (e.g. classes). Whether a mapping is appropriate or not, depends on the context in which the pattern is applied and may best be left to the creative judgement of the software engineer.

A second problem with this approach is that, afterwards, the classes *exist* in the software design, but they will be renamed and manipulated in ways that will make the patterns unrecognisable. As the patterns become less recognisable, a software engineer is more likely to break the pattern when modifying the design at a later stage (or at least: he will not be able to take advantage of the patterns that are present in the design).

For the reasons given above, we are convinced that patterns should not just be used as building blocks, but that these building blocks should also maintain a visible presence within the design. This may be achieved by adding a mapping from pattern-roles onto design artefacts to the design rather than just adding a collection of design artefacts. Through the presence of such a mapping, automated tools could easily verify the integrity of the pattern at any time. Moreover, visually oriented front-ends could display these mappings to the designer in a useful manner. Thus, while designing the software system, the designer would be aware of the patterns governing the interactions between components in the software design.

Section 5.7 will discuss the visual representation of patterns in such designs, whereas we will focus on including patterns into our meta-model in section 5.6.

5.6 Extending the Software System Meta-model

Figure 5-11 presents the software system meta-model from chapter 4 once more. It is this meta-model that we shall extend in this section in order to obtain a pattern-based software system meta-model.

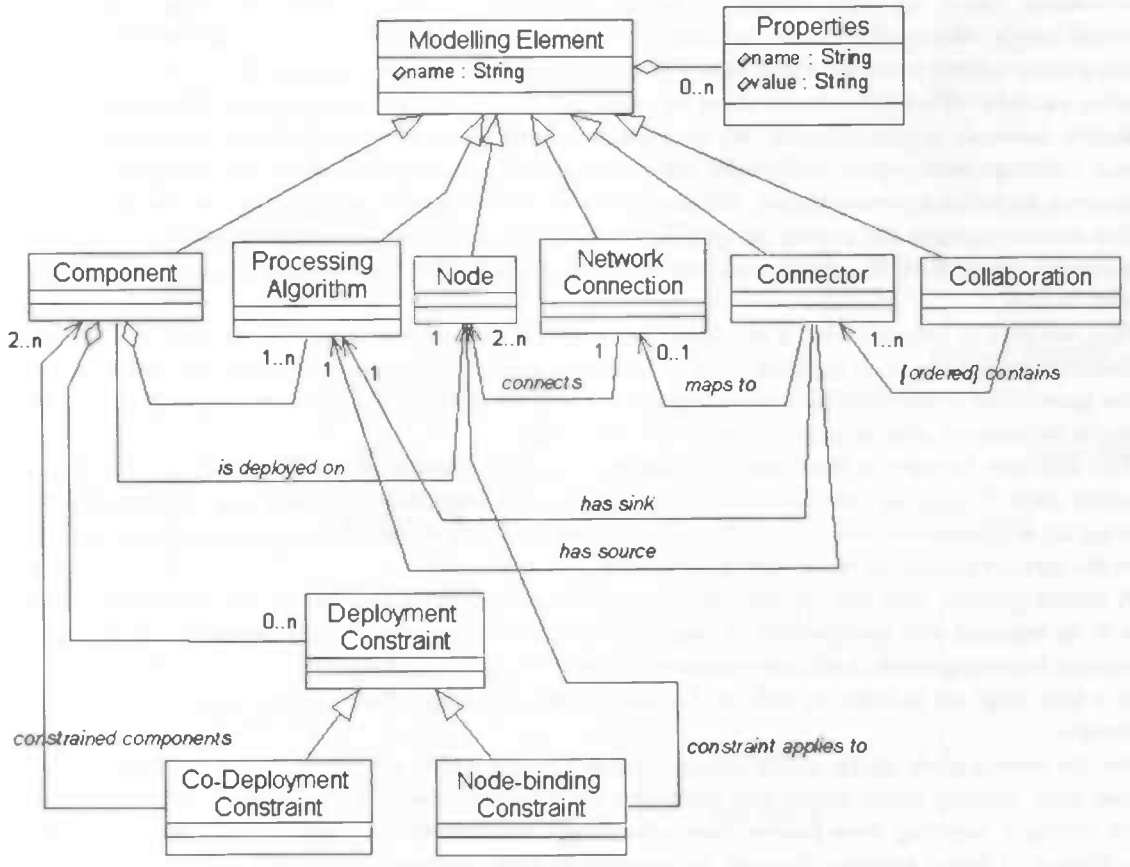

Figure 5-11 The software system meta-model revisited

Both components and processing algorithms may be considered coarse-grained architectural entities. Since we are concerned with coarse-grained architectural patterns, at first glance, both would seem to qualify for assuming the participant-role in a pattern. If we examine the situation somewhat closer though, we see that components are just containers for processing algorithms. The entities actually taking part in the collaborations within the system are the processing algorithms contained within the components. Therefore, we choose to map pattern-roles onto processing algorithms.

Within the context of our system meta-model, this means that pattern-instances will be part of the system model and that processing algorithms may have a "participant"-relationship with that pattern-instance. More precisely, an algorithm will assume one or more roles within the pattern-instance. An example instance is depicted in Figure 5-12.

Our pattern meta-model defines messages that are sent from one role to another. Obviously, applying a pattern also implies that these messages will have to be mapped onto the system design. In our basic system meta-model, this translates to mapping the messages from the pattern-model onto connectors in the system meta-model. Again, an example is depicted in Figure 5-12.

The stereotypes (the words between "<<" and ">>") indicate the names of the connectors in the pattern model. Using this notation allows us to include these names in the system design to identify the connectors as part of a particular pattern. These stereotypes will remain unchanged after the software engineer changes the name of the connector and thus form a more permanent means of including pattern meta-information into designs. Moreover, they may be used as labels for automatic processing by software tools.

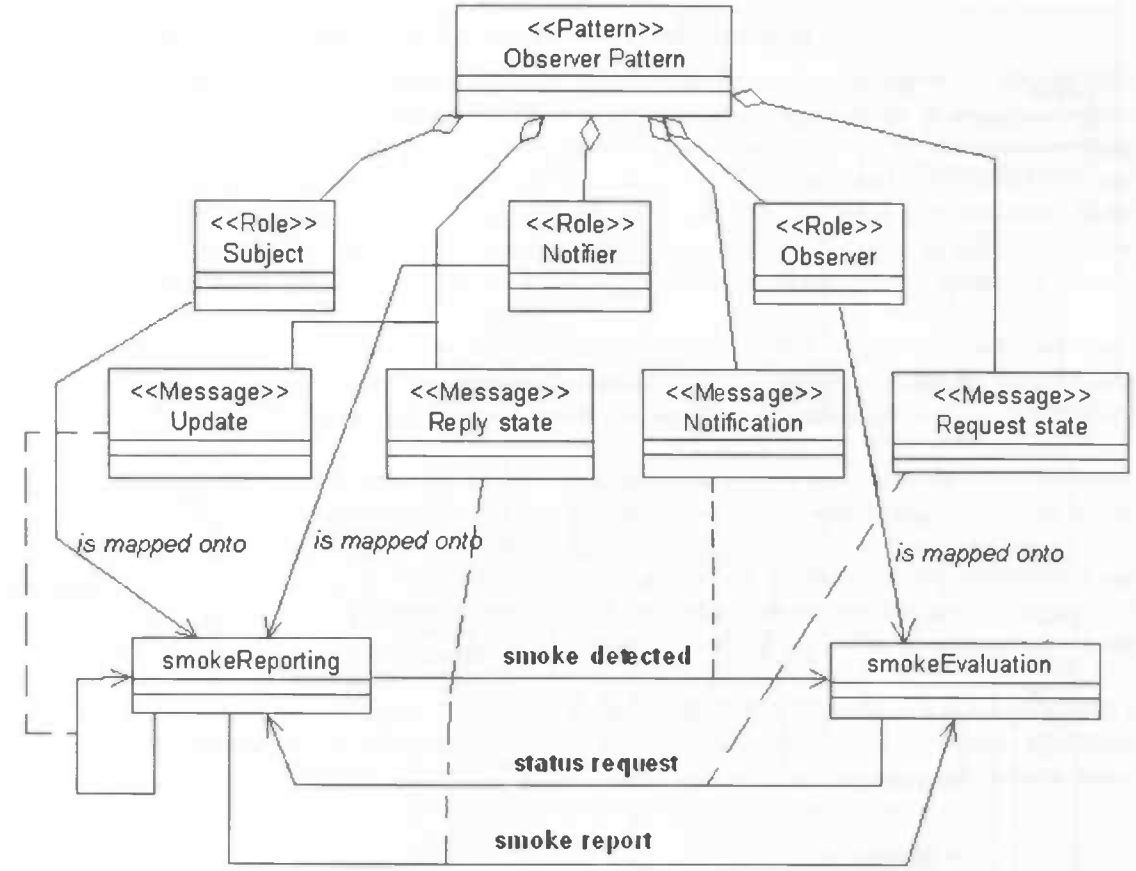

Figure 5-12 An Observer-pattern instantiated within a design

Finally, we extend the software system meta-model with the required elements to obtain a pattern-based software system meta-model. We integrate the pattern meta-model from the beginning of this chapter with the system meta-model from chapter 4, augmented with some additional associations for creating the required mappings. Figure 5-13 and Figure 5-14 show the resulting meta-model using UML class-diagrams.

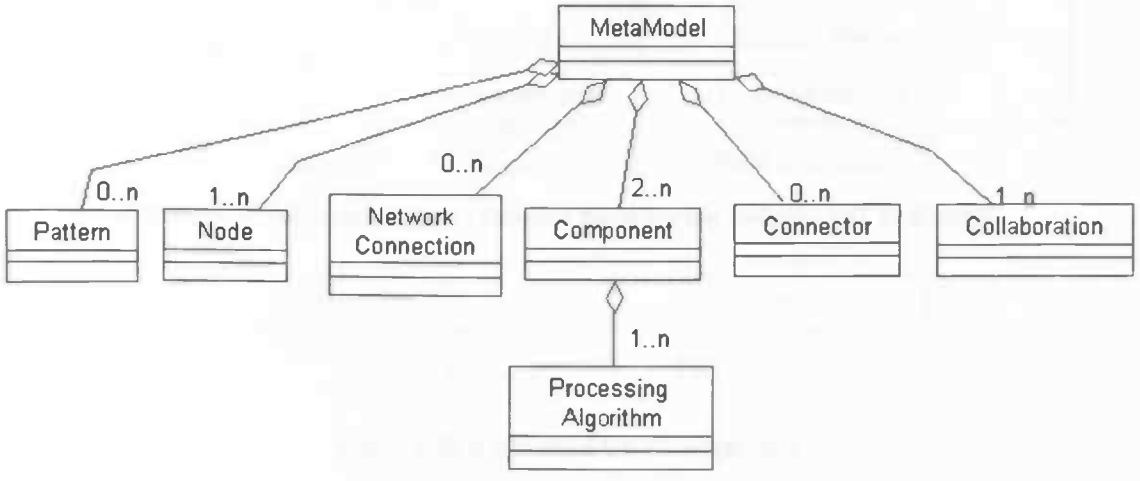

Figure 5-13 The pattern-based software system meta-model (diagram 1/2)

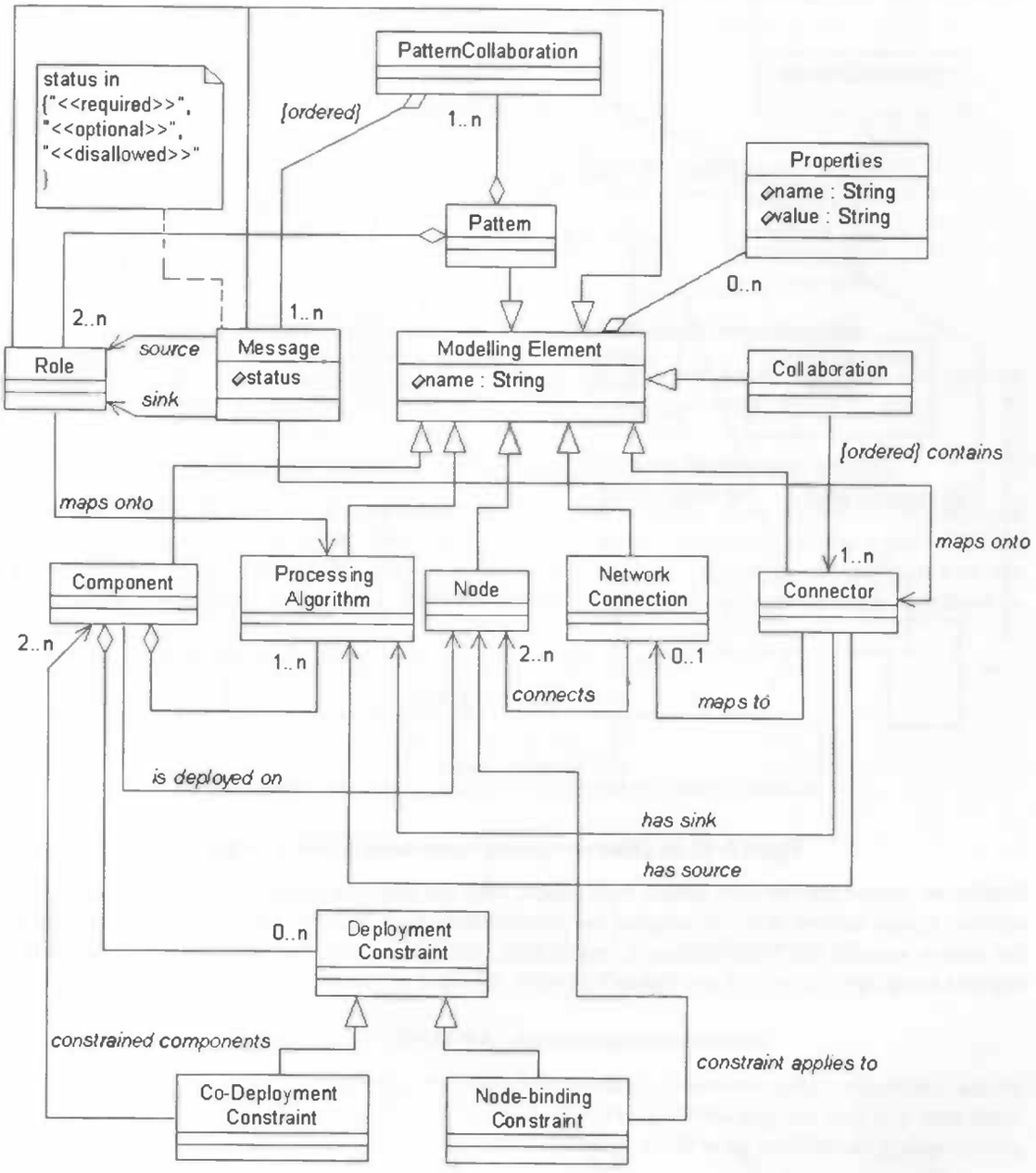

Figure 5-14 The complete pattern-based software system meta-model (diagram 2/2)

5.7 A Visual Representation for Pattern-based Designs

The previous section presented the meta-model we developed for expressing pattern-based designs. This meta-model however can not be deployed for visualizing the presence of patterns in software designs. Therefore, this section introduces a process for supporting pattern-based design by visually expressing pattern-information. Since software designs are often expressed in UML, our goal is to ensure that our pattern-visualization process will also adhere to UML. We shall build upon the software system meta-model we developed in chapter 4 and thus, assume all the UML-based views we discussed in section 4.3 to be present in the software design. Moreover, section 5.6 shows how this visual support was integrated into our software tool.

The meta-model in Figure 5-14 shows that the patterns are related to other design artefacts in two ways: Pattern-roles are mapped onto processing algorithms in the design and pattern-messages are mapped onto connectors in the design. Therefore, it follows that we should be able to visually express these relationships in our software designs.

First, the roles of a specific pattern-instance need to be mapped onto a set of processing algorithm-instances. Since we are dealing exclusively with instances, we choose to use a UML object-diagram (similar to the class-diagram, but representing run-time instead of design-time artefacts) to represent the mapping.

First, a pattern-instance is represented as an object, containing other object-instances, each representing one role in the pattern model. These role-instances should be assigned a "maps to"-association connecting it to an algorithm-instance in the design. This clearly visualizes the role the algorithm is assigned within the pattern-instance.

Obviously, this construct occupies a lot of space within the UML-design. Therefore, it is suggested that a separate pattern-oriented view is used to define these pattern-mappings. In other views though, algorithms might be assigned stereotypes containing the pattern-name and role. In this way, the pattern meta-information will still be present, yet only elaborated within the pattern-oriented view.

This pattern-oriented view is depicted in Figure 5-15.

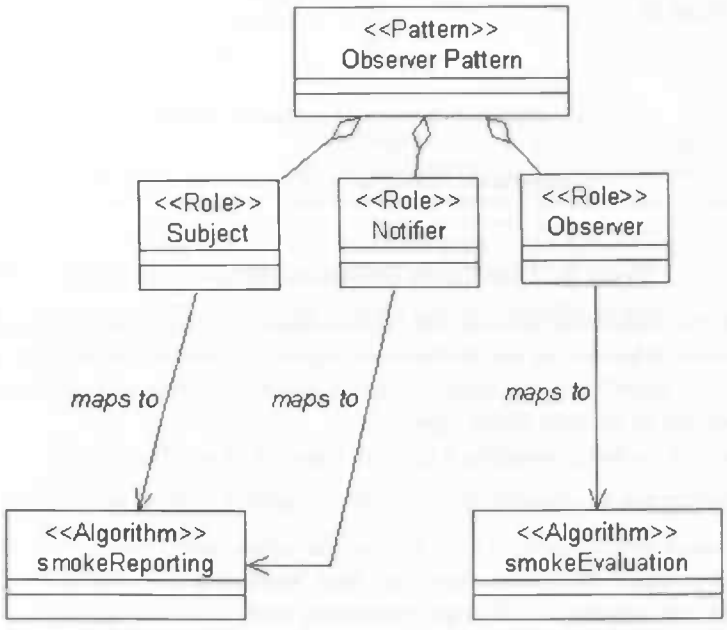

Figure 5-15 A pattern-oriented design-view

Obviously, we also need a means for visually expressing the mapping from pattern-messages onto connector-instances. Connectors are represented by UML-associations in virtually any kind of diagram. Also, in any diagram we have the possibility of using stereotypes to add meta-information to the model. Thus, we exploit this by expressing pattern-related meta-information in stereotypes. In this case, the

stereotype should contain the name of the pattern and the name of the message that the connector represents. This concept is illustrated in Figure 5-16.

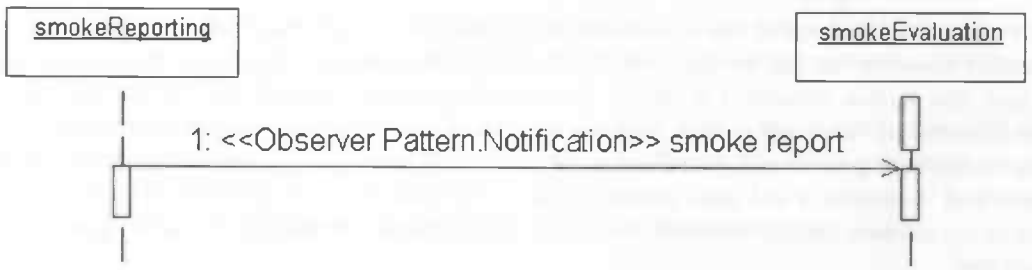

Figure 5-16 Expressing the mapping from pattern-messages onto connectors

This finalizes our work on visualizing patterns in designs. The next section will illustrate the concepts developed in this chapter, by applying them to the fire fighting system case study.

5.8 Example: A Pattern-based Fire fighting System Software Model

In this section, we extend the software system model developed in chapter 4, by applying a pattern to the existing design. It is outside the scope of this thesis to show a realistic example containing a great diversity of patterns and a sizeable design. We therefore limit ourselves to demonstrating the application of the Observer-pattern to the aforementioned design. Note that the Observer-pattern model was defined in section 5.4.

Recall the definition of the Threat Evaluation-component from chapter 4:

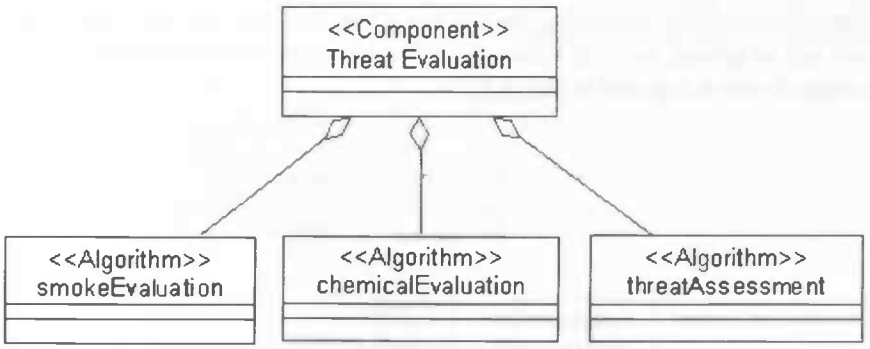

Figure 5-17 The Threat Evaluation-component revisited

We choose to make the interaction between the *threatAssessment* and *smokeEvaluation* algorithms adhere to the Observer-pattern, whereby the *smokeEvaluation* algorithm contains the relevant state (e.g. data about the smoke levels in the tunnel) and the *threatAssessment* algorithm needs to be informed of changes to this state in order to generate an accurate threat report.

A corresponding pattern-to-design mapping is given in Figure 5-18 and Figure 5-19.

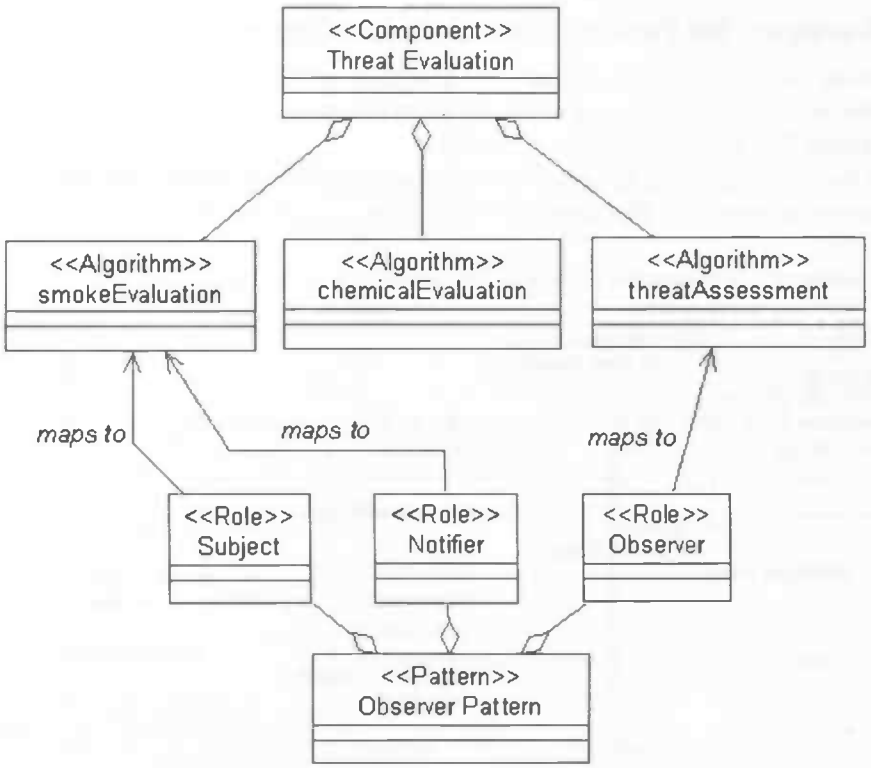

Figure 5-18 Mapping pattern-roles onto algorithm-instances in the example system

Figure 5-19 Mapping pattern-messages onto connector-instances in the example system

It should be noted that we chose to instantiate all parts of the Observer-pattern model. If we had omitted some of the optional connectors (e.g. *request state* and *reply state*), the mapping would still be valid. Moreover, we might have introduced a third processing algorithm to assume the role of *notifier* in the Observer-pattern.

The next section presents how the pattern-based design process was implemented in our software tool.

5.9 Tool-support for Pattern-based Software System Design

Before describing the tool support for pattern-based design we implemented as part of this research, we briefly describe the tool-support that is available in two of the more common commercially available tools: *Rational RSA* from IBM and *Together* from Borland.
It should be noted though, that neither of the two applications was at our disposal when we started designing our own software tool. Therefore, our results were not influenced by either application.

The visual representations of patterns in Rational RSA is shown in the figure below.

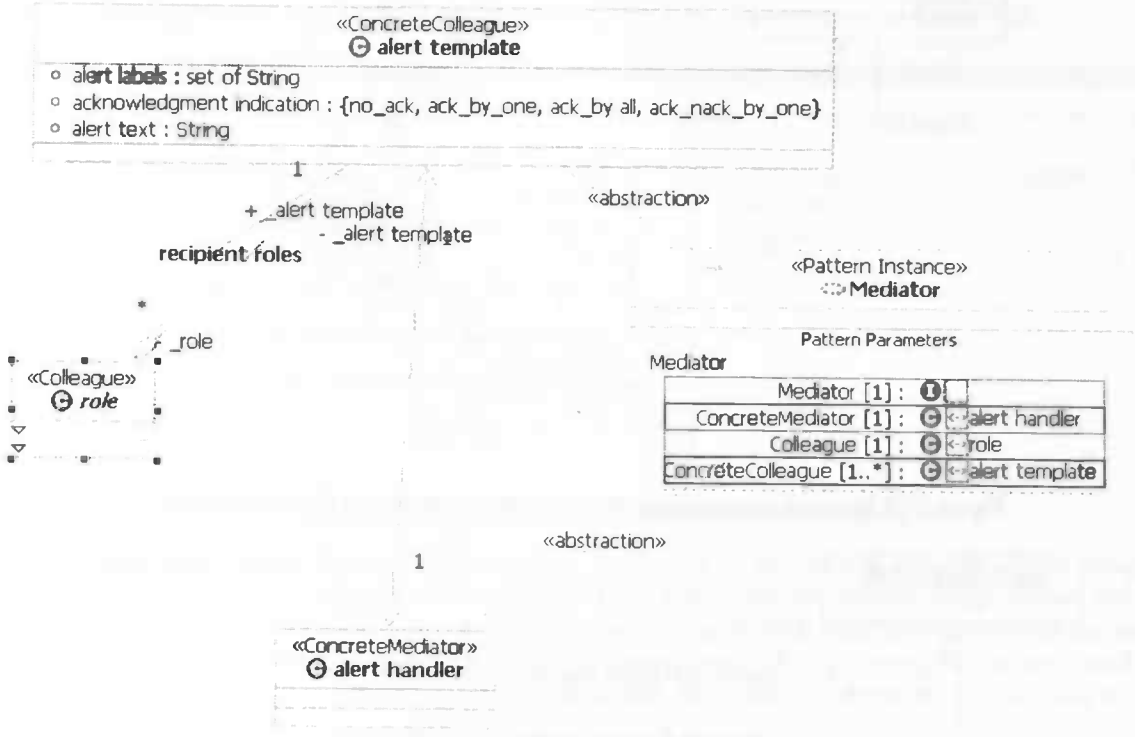

Figure 5-20 Patterns in Rational RSA

Compared to our own solution presented in this chapter, RSA uses a similar approach in that it adds a pattern-instance to the software design and adds stereotypes containing meta-information to the classes involved in the pattern-instance. Unlike our solution though, RSA uses a pattern-representation, which does not conform to standard UML.
Finally, RSA also provides a pattern-library that allows the designer to browse the available patterns. The library is structured according to the criteria identified by Gamma et al. [Gof95] and thus contains behavioural, structural as well as creational patterns.

On the other hand, Borland Together also provides a great variety of patterns to choose from, most of these are technology-specific though (e.g. J2EE-patterns or Oracle-patterns). Unlike Rational RSA, Together does not insert any non-UML artefacts in the software design. Moreover, it does not insert any instance of a pattern as such. Rather, it uses the approach used by Gamma et al. in that it simply allows the user to insert a fixed class-hierarchy into the software design. This approach does not provide much freedom and makes it difficult to maintain patterns over time.

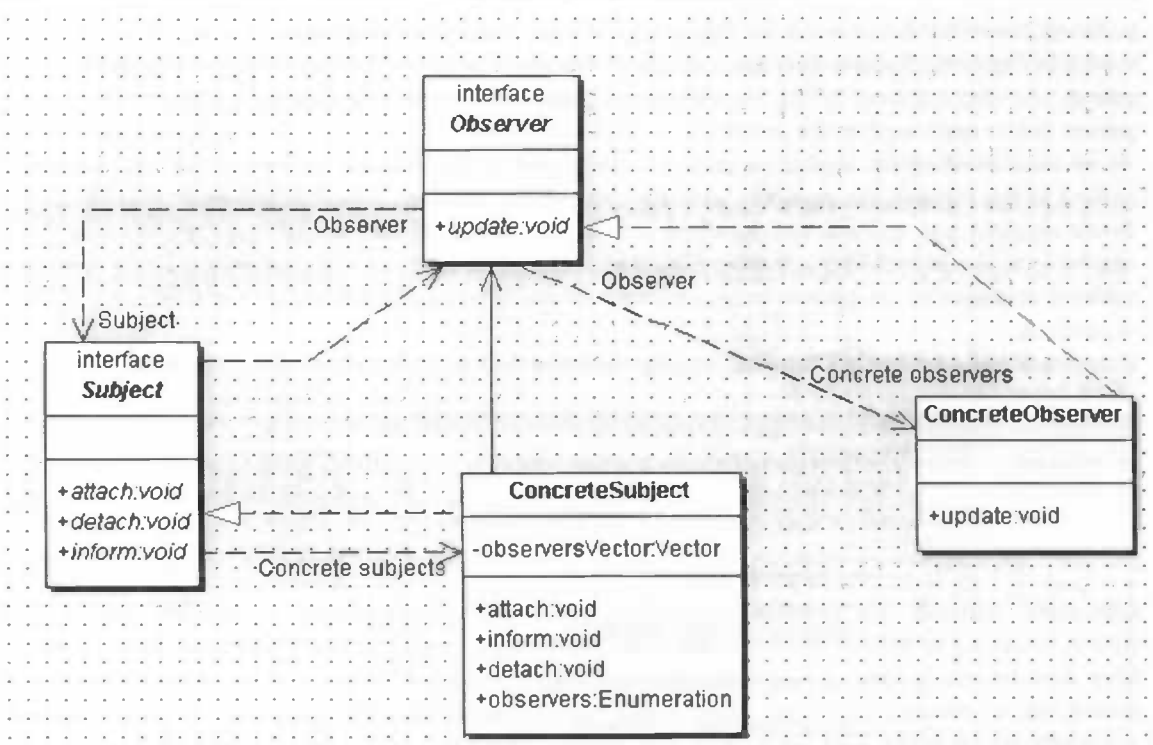

Figure 5-21 Patterns in Borland Together

In the rest of this section, we present the support for pattern-based design that we included into our own software tool during this research project. The support consists of the following elements:

- Patterns are visually expressed in the design-views
- A pattern-library is available for browsing pre-defined patterns
- Patterns are automatically applied upon selection
- Pattern-integrity is verified

The visual representation of patterns is implemented according to the constructs explained in section 5.7. Therefore, we do not reiterate what was presented before. Instead, we trust that a screenshot of the application shall be sufficient (Figure 5-22).

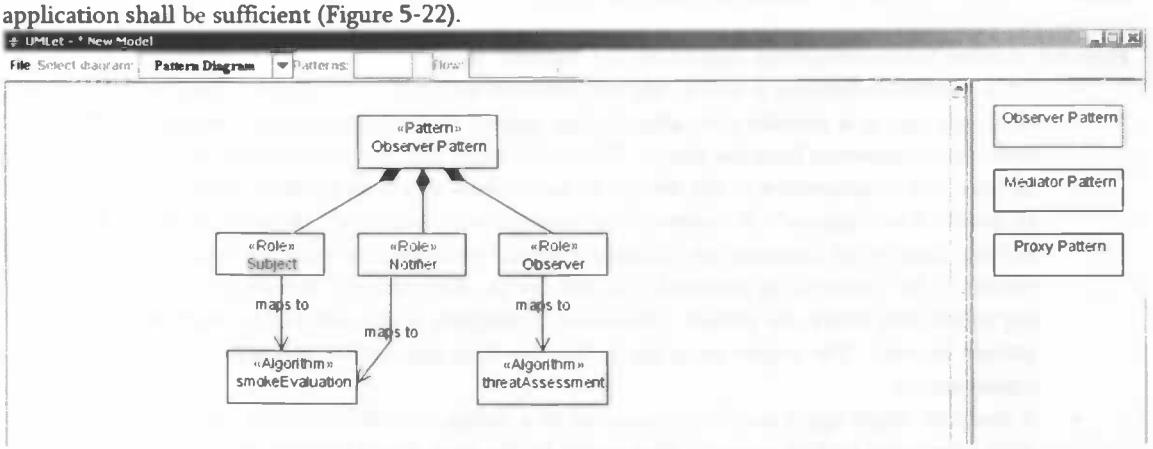

Figure 5-22 Pattern-visualization in our software tool

A good pattern-library would be one that is sorted according to at least one sorting key. Otherwise, looking for an appropriate pattern for our design-issue would not be practical. However, developing an appropriate pattern-library system was not within the scope of our research goals. Therefore, we implemented a very rudimentary and unsorted "library" and leave the implementation of a more sophisticated implementation

to be addressed by future work on this project. To be complete: Our pattern-library implementation is shown in Figure 5-22 above. It consists simply of the list of patterns listed on the right-most panel on the screen. Clicking a pattern in the list will open a pattern-viewer, which allows the designer to inspect a pattern before applying it to the design.
As we mentioned earlier, applying a pattern is performed as demonstrated in the figure above. Applying a pattern in this manner, automatically results in the addition of a new (set of) collaboration(s) to the design. Every required and optional message from the pattern model is automatically inserted into the design according to the user-defined mapping. The user is then free to remove the connectors corresponding to optional messages in the pattern or to otherwise manually adjust the interaction between the processing algorithms.
An automatically applied collaboration, corresponding to the figure above is given in Figure 5-23.

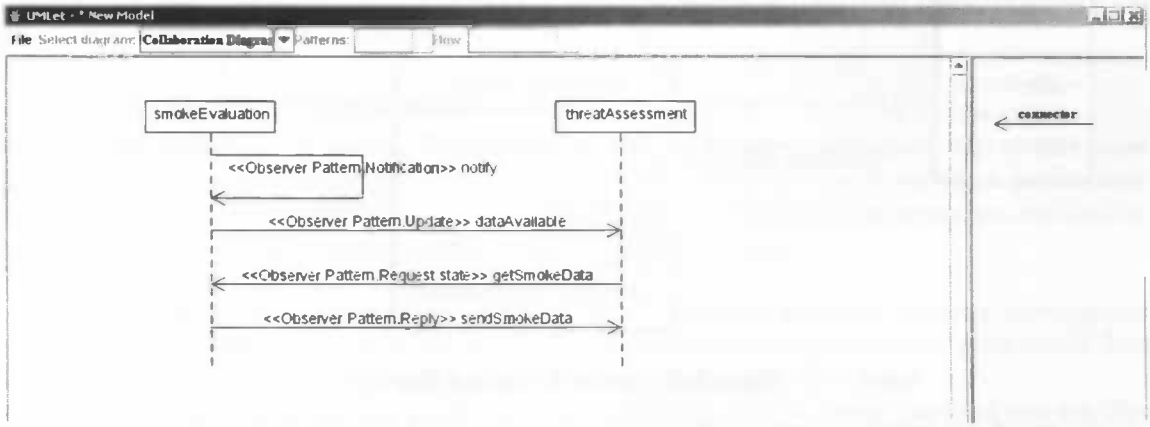

Figure 5-23 Automatic collaboration-instantiation in our software tool

Finally, our software tool also supports automatic verification of the pattern integrity in the design. Obviously, we did not have the resources to develop a fully-fledged and user-friendly design-support front-end for this verification mechanism, nor was this our goal. The application does, however provide the user with feedback once it detects a pattern's integrity is no longer intact. Ideally, the tool should detect such a problem before it occurs and provide suggestions for solving the problem, but this is left as an issue to be addressed by future work.
In the rest of this section, we discuss the concepts used for our pattern-integrity verification.

First, we examine the scenarios that might lead to a "broken" pattern:

- Since a software designer is free to edit the software design in a non-pattern-based way (i.e. normal UML-editing), it is possible that, after having applied several patterns, the designer might remove individual connectors from the design. This could mean that some messages from the pattern would no longer be implemented in the design. In some cases, this is no problem since the message might be qualified as "optional". If, however, the message were "required", the pattern would be "broken" and the interaction between the algorithms would no longer be valid. This situation requires the pattern to be "repaired" or removed from the design. Alternatively, the designer might opt to undo the action that broke the pattern. Obviously, a designer might also opt to purposefully break the pattern anyway. The important point is that he does this knowingly, while being aware of the consequences.
- A designer might apply conflicting patterns to a design, or otherwise add connectors that conflict with previously applied patterns. This might be the case, when patterns include the "disallowed"-attribute. One pattern might mandate a connector between two entities, while another prohibits this: such conflicts should be detected, yet are not easily solved, since they point out a design flaw. A designer cannot prohibit a connection between two algorithms from existing while instantiating such connections at the same time. This verification mechanism should provide support in detecting such flaws.

- Not all roles of a pattern might be mapped onto appropriate design elements (processing algorithms). In this case, the pattern will not have been completely mapped onto the design and hence will be "broken". This situation may be solved by mapping all pattern-roles onto processing algorithms.

In principle, checking for such inconsistencies is relatively easy:

1. Select all pattern-instances in the design.
2. For each pattern-instance, check whether all its roles are mapped onto the design.
3. For each pattern-instance, check that all its required messages are mapped onto connectors of the design.
4. For each pattern-instance, select all its disallowed messages and verify that the design contains no connectors breaking the "disallowed"-rule.

This process suffices to verify that none of the inconsistencies listed above will be present in the software design.

5.10 Summary

This chapter described a meta-model for expressing architectural patterns that describe interactions between system-level artefacts. The pattern meta-model states that a pattern consists of a number of roles that exchange messages in a predefined order. More complex patterns may contain several such valid message sequences. Moreover, the pattern meta-model allows some degrees of freedom in the pattern models, meaning that several variants of the same pattern may be captured by a single model. Moreover, through the use of suitable meta-information, the meta-model facilitates the automatic verification of a pattern's integrity.

The pattern meta-model has been integrated with the software system meta-model from chapter 4, resulting in a meta-model for pattern-based software system designs. We demonstrated how the meta-model might be used for creating pattern-based software designs and also suggested a means for visually representing patterns in such designs.

Finally, we presented the way in which we implemented the concepts from this chapter in our pattern-based software design tool.

6 Verifying the Non-functional Performance Requirements on a Pattern-Based Software System Design

In the previous chapters we described a process for pattern-based software system design. Moreover, we have been exclusively focussed on modelling such systems, without taking performance requirements into consideration. This chapter discusses a process for first calculating the performance characteristics of an existing design and then verifying whether these characteristics meet a set of pre-specified performance requirements. The performance characteristics are calculated based on a model of the hardware- and software behaviour, which was also developed during this thesis.
Finally, unlike in the previous chapters of this thesis, we chose not to dedicate a separate section to our fire fighting case study. Instead, we chose to discuss the theory along with the case study, since the concepts in this chapter would be too abstract without a tangible case.

6.1 Analysis and Requirements Definition

The aim of this chapter is to develop a process for verifying performance requirements on pattern-based software system designs. Clearly, this would not be possible without first specifying which performance requirements are to be verified. Section 2.2 first defined these requirements to be *throughput* and *end-to-end time*. Citing the definitions from the aforementioned section, we have:

> *Throughput is the amount of data being sent from one software entity to another in a given period of time. A software system has to be able to function correctly in a given throughput-scenario. Such a scenario could thus be specified as a performance requirement on the design.*
> *End-to-end time is defined as the time that a message takes to "travel" between two specified "points" in a software system.*

In the previous chapters we developed concepts that allow us to specify these terms more accurately. We thus redefine the terms as follows:

- Throughput is the amount of data per unit of time that is sent from one processing algorithm to another through exactly one connector. Throughput requirements are therefore specified as a number of bytes that need to be sent from one processing algorithm to another each second. An example throughput requirement would be: The *smokeReporting* algorithm shall be able to send at least 50 KB/s of data through the *smoke report*-connector to the *smokeEvaluation* algorithm.
- End-to-end time is the time it takes for a message to propagate through the software system from the sending processing algorithm through one or more intermediate processing algorithms to the receiving processing algorithm. More specifically, it is the time between the moment at which the sending algorithm starts processing the message and the moment at which the receiving algorithm has finished processing the message. Such a "logical path" through the software system may henceforth be referred to as a *functional flow*.

After having defined these concepts more accurately, we can use them to specify the requirements to our performance requirement verification process.

1. The process shall be able to approximate the system design's performance characteristics based on a model of the hardware behaviour and a model of the software behaviour.
2. The process shall use the approximated performance characteristics for verifying whether a specified set of throughput- and end-to-end time requirements will be met by the system design.
3. The process shall leverage pattern-specific characteristics for estimating performance characteristics.
4. The process shall be suitable for implementation in a software tool.

6.2 Pattern-based Performance Verification

The research goals in section 2.3 state that we aim to leverage the knowledge about patterns in a system design to estimate performance characteristics of that design. This section elaborates on this topic and discusses the results of our effort.

6.2.1 Patterns and Non-functional Characteristics

Since our interest is in estimating non-functional characteristics of a software design, this paragraph discusses the relation between patterns and non-functional characteristics.

Since patterns represent experience with the application of a particular solution to a reoccurring problem, non-functional characteristics of such solutions are generally known as well. For example: The Observer-pattern, which was introduced earlier in this thesis, is mainly used for reducing dependencies between several classes. This low coupling has a positive effect on the maintainability of the application, while it generally hurts its performance compared to a tightly coupled version of the same application.

Similar statements can be made about other patterns: They generally represent a solution aimed at achieving a set of desired software characteristics at the cost of one or more other characteristics. In this context it is interesting to note that Bas et al. [Bas03] point out that achieving any non-functional characteristics other than performance will generally come at the cost of performance. In our case, we are especially interested in performance-related characteristics (throughput and end-to-end time), which we shall discuss in more detail in the next section. Moreover, the next paragraph discusses the quantification of software characteristics within the context of patterns.

6.2.2 Quantification of Pattern Performance Characteristics

Knowing that a pattern increases the maintainability of an application is not enough. What is really required is a way of quantifying such pattern characteristics. For example: When applying the Observer-pattern to a system design, one does not just want to know that this will increase the maintainability of the system at the cost of performance. Rather, one needs to know how much the performance will be reduced and how much maintainability is gained as a result. This kind of information would enable the designer to make a balanced decision, knowing exactly how much performance is traded to achieve a known degree of maintainability.

Obviously, this scenario is somewhat idealistic: Although metrics for all kinds of software characteristics have been established, they are not able to measure a meaningful difference between a system design before and after the application of a single pattern. Trying to assign quantifiable metrics to individual patterns while not considering their implementation-context is even less fruitful, since many implementation-specific parameters (e.g. average message size or timing constraints) determine how the non-functional characteristics will be influenced.

A degenerate form of this approach does attempt to determine numerical values for the non-functional characteristics of patterns. Instead, it just specifies how a pattern affects relevant characteristics. For example: maintainability is increased and performance is reduced. Yet, this information cannot truly be leveraged for calculating a system's non-functional characteristics.

We conclude that metrics of non-functional characteristics cannot be associated with a pattern itself. The impact of applying a pattern can only be evaluated after it has been instantiated in a design context. Such a context instantiates design-specific parameters like average message size and average activation frequency. Moreover, a pattern will generally only be useful if it is applied within the right problem context (i.e. to solve a problem that it is meant to solve).

When looking at the non-functional characteristics that are of interest to our research, throughput and end-to-end time, we see that both require absolute values for an effective verification process. For a particular connector, the exact throughput should be estimated, not just a relative qualification like "good' or "reasonable". In the same way, one wants to estimate whether an end-to-end time is 15 or 20 milliseconds: a quantified answer.

As we mentioned, such quantified answers can only be determined within a design context. Since we cannot assign meaningful metrics to individual patterns, we need to evaluate the design after the pattern has been applied. Yet, in this context the pattern itself is not relevant to our verification process anymore. The behavioural patterns that we model in this thesis simply introduce a set of interaction rules and constraints into the design. It is these rules and constraints that need to be evaluated in the context of the entire design, not the pattern itself. The next paragraph will present our alternative approach.

6.2.3 An Alternative for Pattern-based Verification

Since we conclude that we cannot leverage patterns in a meaningful way in order to determine estimates of throughput and end-to-end time, we decided to develop an estimation process that does not rely on patterns. This deviates from the original research goals, yet we conclude that the research goal in its original form cannot be satisfied. Instead, our verification process shall be based on the interaction between components in the system design, where the interaction will be largely specified by pattern instances. Thus, indirectly we are still relying on pattern information, albeit not on pattern-specific metrics of non-functional characteristics. The next section details the estimation and verification process we have developed.

6.3 A Process for Verifying Performance Requirements

This section starts by discussing the specification of performance requirements, after which it continues by presenting a model of the hardware configuration characteristics. This model is required for approximating the hardware behaviour in the system. We then proceed by presenting a similar model for approximating the software behaviour, which will allow us to calculate the approximate performance characteristics of the system. Finally, this section concludes by using the performance calculations in order to verify the performance requirements.

6.3.1 The Software System Model Revisited

Although the fire fighting software system model was specified in chapter 4, it was extended in chapter 5. Because it forms the basis for the examples throughout this chapter, we provide the resulting model once more.

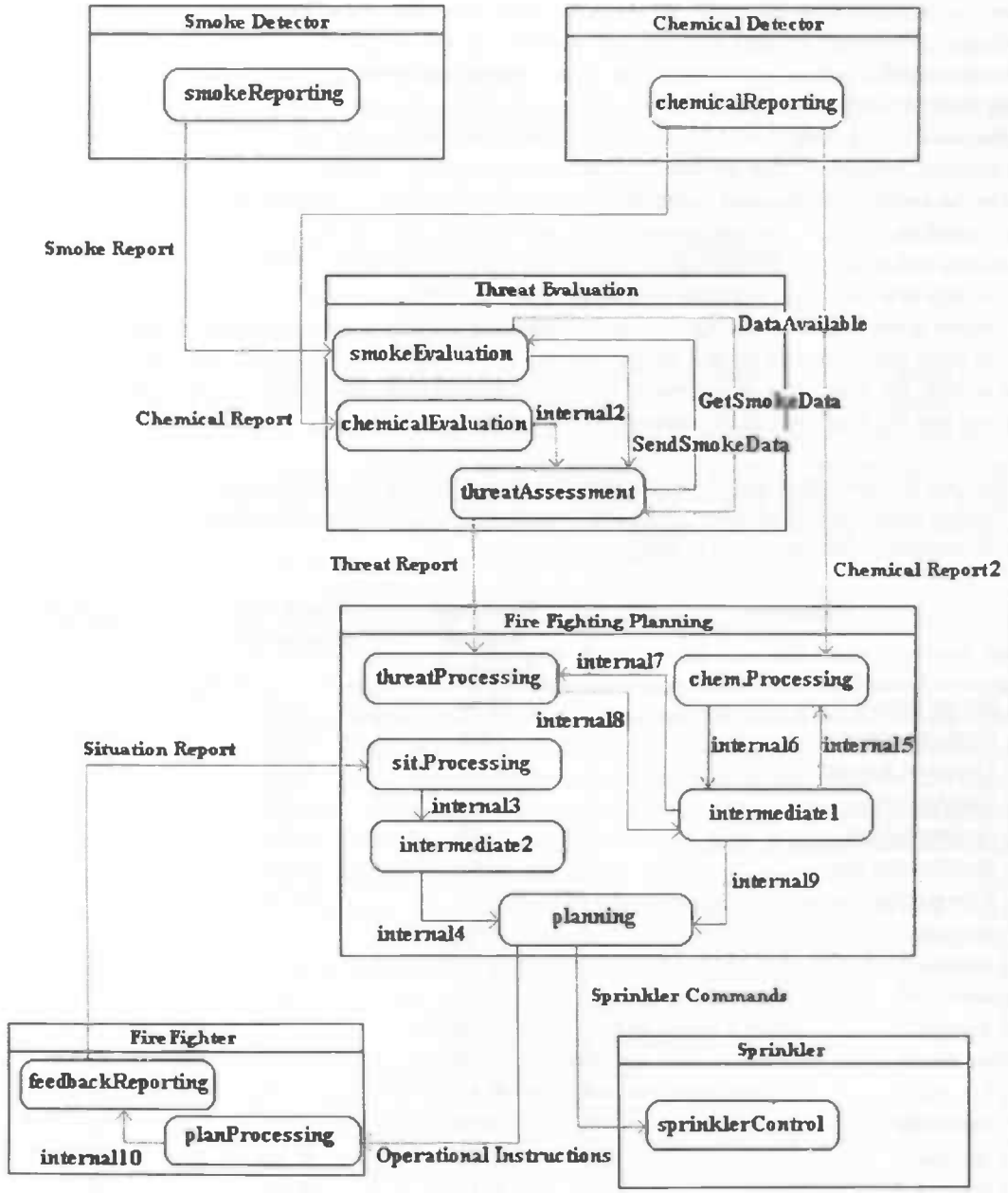

Figure 6-1 The fire fighting software system model revisited

6.3.2 Defining the Performance Requirements

As we mentioned earlier, the performance requirements that we aim to verify are throughput and end-to-end time. This paragraph discusses the specification of these requirements.

First, throughput requirements constitute input to the calculation process: We first define a throughput-scenario involving all connectors in the system and then calculate whether the system configuration will be able to cope with this scenario. It may be, for example, that the network- or processing resources in the system cannot handle the throughput specified by the scenario. Consequently, throughput requirements have the form of an input-scenario for the system. Usually, such a scenario will describe a worst-case system-load. Although throughput is defined as the number of bytes that flow through a connector each second, the input scenario should contain the two components from which throughput can be calculated:

worst-case message frequency and *worst-case message size*. Both these measures are used as input to our calculation process and are thus required to be provided by the designer.
As we mentioned earlier, end-to-end time requirements are specified with respect to a functional flow, which itself consists of an ordered sequence of one or more connectors. Unlike the throughput specification we discussed before, end-to-end time requirements do not necessarily take every connector in the system into account. Moreover, they are not required as input for the calculation process. Rather, they are only used in the verification stage and may therefore relate exclusively to those functional flows that the designer is interested in. As such, the end-to-end time requirements have the form of one or more functional flow definitions and an accompanying required maximum to the traversal time of that flow.
At this stage it is important to realize that deriving the performance requirements is not a trivial exercise. If the process is used for re-factoring an existing system, then many of the worst-case requirements may be derived from measurements on that system. On the other hand, if the designer is developing a new system from scratch, the worst-case requirements need to be derived from a thorough system analysis, based on the expected size and frequency of the messages in the system.

To illustrate this process of specifying performance requirements, the requirements relating to our example fire fighting system are listed below. Note that the connectors may be found in Figure 6-1.
The throughput-requirements are as follows:

	Connector	Worst-case message frequency	Worst-case message size	Throughput
1	Smoke Report	15 Hz	5 KB	75 KB/s
2	Chemical Report	5 Hz	8 KB	40 KB/s
3	Chemical Report2	5 Hz	8 KB	40 KB/s
4	DataAvailable	15 Hz	1 KB	15 KB/s
5	GetSmokeData	15 Hz	1 KB	15 KB/s
6	SendSmokeData	15Hz	3 KB	45 KB/s
7	Threat Report	20 Hz	5 KB	100 KB/s
8	Internal2	5 Hz	10 KB	50 KB/s
9	Internal3	10 Hz	2 KB	20 KB/s
10	Internal4	10 Hz	5 KB	50 KB/s
11	Internal5	10 Hz	1 KB	10 KB/s
12	Internal6	10 Hz	4 KB	40 KB/s
13	Internal7	10 Hz	1 KB	10 KB/s
14	Internal8	10 Hz	5 KB	50 KB/s
15	Internal9	10 Hz	45 KB	450 KB/s
16	Internal10	10 Hz	2 KB	20 KB/s
17	Sprinkler Commands	10 Hz	15 KB	150 KB/s
13	Operational Instructions	10 Hz	20 KB	200 KB/s
14	Situation Report	10 Hz	10 KB	100 KB/s

Table 6-1 Throughput requirement specification by using a scenario

The end-to-end time requirements are as follows:

	Functional flow	Worst-case end-to-end time
1	Smoke Report -> DataAvailable -> GetSmokeData -> SendSmokeData -> Threat Report	1 second
2	Chemical Report2 -> Internal5 -> Internal6 -> Internal9 -> Sprinkler Commands	1 second

Table 6-2 End-to-end time requirement specification

6.3.3 Modelling the Hardware Configuration

The hardware configuration of our fire fighting system was discussed in chapter 4. This paragraph presents a model for expressing the performance-related characteristics of such a hardware configuration. These characteristics correspond to the property-entities that are contained in our software system meta-model.
Moreover, the development of such a hardware simulation model is required for evaluating the run-time characteristics of the software system. In general, the performance of a software system cannot be accurately predicted without taking the hardware-related limitations into account. The modelling of processing nodes is discussed first, after which the modelling of networks is addressed.

The modelling of a processing node is a complex matter, since the performance of a processor in a particular application is dependent on many factors like processor speed, cache size and other processor specific features. Accurately modelling processing nodes is therefore not a goal of this research. Instead, we opt for a relatively simple model of processing nodes in which we define one specific type of node to be the reference node (e.g. Pentium 4 at 3 GHz). The idea is to run a series of benchmarking tests for this reference node, such that we can determine its performance characteristics (which are elaborated on below). Whenever a new type of node is to be deployed, one may execute the same series of benchmarking tests in order to determine the relative speed of the new node compared to the reference node. It is then possible to scale the performance characteristics in proportion to the relative processor speed.
To model a node's performance characteristics, we need to take into account that as the load on a node increases, it will need more time to do the same processing. So the execution times of an algorithm will increase (non-linearly) as the processor load of the node increases. To model this behaviour, we have chosen to determine the relation between CPU load and the resulting delay to the execution time of benchmarking code for each type of node. Obviously, a designer might just as well create a profile for a type of node that is not yet available: This allows the system designer to experiment with and specify node requirements in order to determine an appropriate type of node for the particular software system.
To illustrate the concept of node performance modelling, consider the processing delay-functions depicted in Figure 6-2 and Figure 6-3.

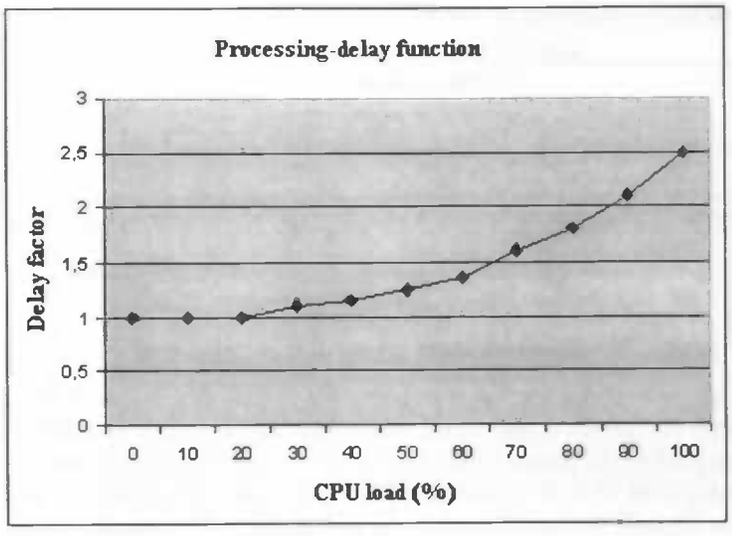

Figure 6-2 Example processing-delay function (1/2)

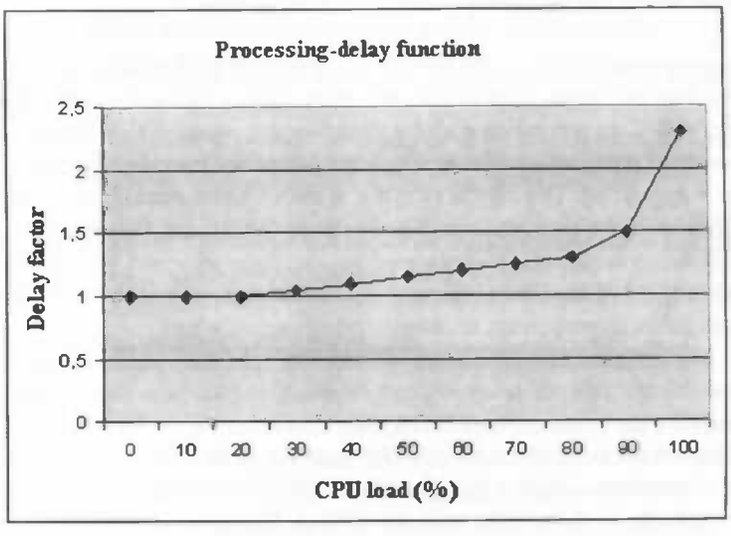

Figure 6-3 Example processing-delay function (2/2)

Finally, our system model from chapter 4 contains 3 nodes. For our calculation purposes, the characteristics of our nodes are summarized in Table 6-3.

	Node Alpha	Node Beta	Node Gamma
Relative speed	1x	2x	1.5x
Delay-function	Figure 6-2	Figure 6-3	Figure 6-2

Table 6-3 Fire fighting system node characteristics

Compared to the processing nodes, networks are relatively easy to model. The performance of a network is generally expressed as the amount of data it can transport per unit of time (usually bits per second is used as a measure). Therefore, it is easy to verify whether some throughput can be handled by a particular network. The timing characteristics involved in network modelling are somewhat more complex. In the ideal case, calculating the time it takes to transport an amount of data across a network with known network speed is fairly simple: One simply divides the message size by the network speed to acquire the time it takes to transmit the message.

For example: Transmitting 50 kilobytes over a 1-megabit connection would require

50,000 bytes/(1,000,000 bits/sec.) = 50,000 bytes/(125,000 bytes/sec.) = 0.4 sec.

Unfortunately, realistic networks are not ideal. They are subject to (non-linearly) increasing delays as the network load grows. Thus, we need to take this relationship into account by determining a delay-function that depends on the network load and is established for each type of networking hardware. Two example functions are provided in Figure 6-4 and Figure 6-5.
Finally, the data transmission time over a network is formally given by

actual_time = message_size/network_speed * delay(network_load)

Our fire fighting system contains two separate networks. The following table provides their performance characteristics.

	Network 1	Network 2
Data rate	2.5 Mbps	5 Mbps
Delay-function	Figure 6-4	Figure 6-5

Table 6-4 Fire fighting system network characteristics

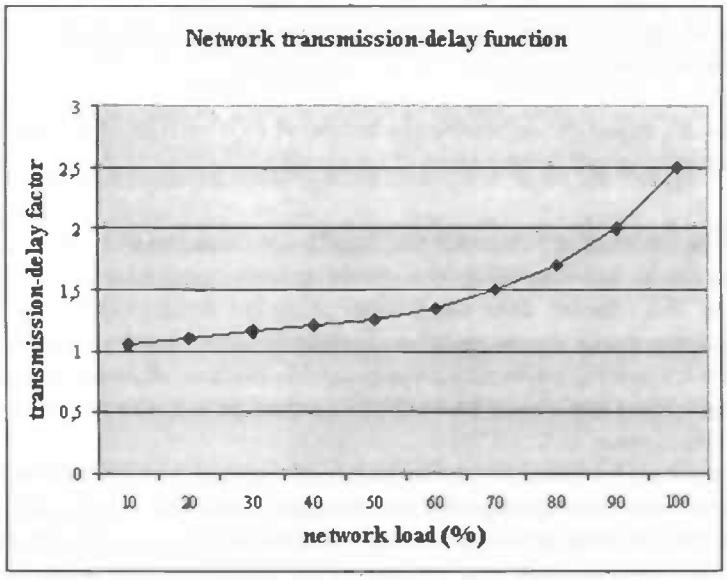

Figure 6-4 Network transmission-delay function (1/2)

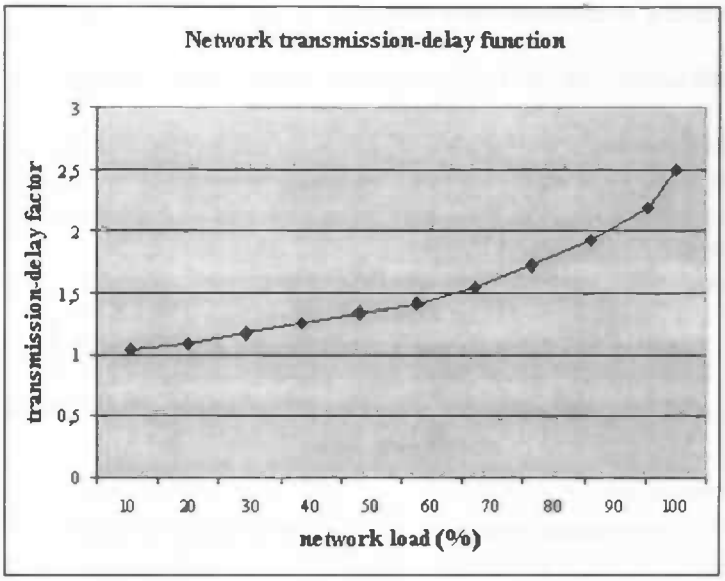

Figure 6-5 Network transmission-delay function (2/2)

Note that the hardware characteristics discussed above can be added to the system modelling meta-model by means of the so-called "properties" we defined within the meta-model. For example, the network speed is simply a property of the "Network Connection"-element of the meta-model, wheras the load-delay behaviour is another such property.

We have now defined a model for expressing the hardware configuration. The next paragraph presents a similar model for software-specific performance characteristics.

6.3.4 Modelling the Software Configuration

Chapter 4 addresses the issue of expressing the software entities contained within software designs, yet the model presented in that chapter does not provide a means for expressing the performance-related characteristics of the processing algorithms. The characteristics we require might be measured whenever existing components are used to construct a new system. In the case of a new development however, we might estimate these characteristics based on experience from previous projects or by creating prototypes and measuring their behaviour.

One of the most important performance characteristics of a processing algorithm is the time a processing node will require to execute it. Clearly, this characteristic cannot be determined without specifying a context. Therefore, the following approach is used: A reference node is used as the basis for measuring or predicting a so-called *base execution time*. Whenever the algorithm executes on a node other than the reference node, the execution time is scaled proportionally with the relative speed of the alternative node. The base execution time is defined as the time the algorithm takes to complete processing a worst-case input on the reference node under minimal system load.

The minimal system load is defined as the CPU load that the processing node displays when it is not running any tasks other than the operating system and other background processes (i.e. it is not performing any "work"). Note that using a worst-case input for determining the base execution time guarantees a correct worst-case analysis, while it may cause very pessimistic analysis results, especially for algorithms whose execution time is highly input-dependent.

Once the base execution time is known, it should be used for determining the actual execution time. The actual execution time is obtained by multiplying the base execution time with the node's execution-delay factor.

Another important software performance characteristic is given by the way in which the behaviour of an algorithm is influenced by its input- and output activity. As the algorithm receives and sends more messages

per time unit, its contribution to the CPU load increases. Moreover, as an algorithm's input or output grows in size (with a constant number of messages per time unit) it will again increase the algorithm's contribution to the CPU load. To take both these factors into account, while still aiming for easy measurability, an algorithm's CPU load contribution should be determined as a function of the input- or output-frequency and the message size.
Essentially, it does not matter whether one measures the activity with respect to input or output. Which option best describes a particular algorithm depends on the algorithm implementation and nature.

Finally, some algorithms might be *timed* while others are *un-timed*. Un-timed processing algorithms commence processing as soon as they receive an input message on one of their connectors (input-driven). This means that they are able to minimize the delay in forwarding messages through a functional flow.
In some cases an algorithm needs to provide output at a constant rate or a frequently activated algorithm might be prevented from being activated too often in order to avoid excessive processor context-switching. Such algorithms use a timed strategy, which implies an additional delay incurred by the functional flows that are dependent on such algorithms.
Obviously, this algorithm characteristic should be considered when calculating end-to-end times in a system.

In short, the following formulas summarize the concepts we have presented:

actual_execution_time = base_execution_time * processor_delay_factor(cpu_load) / node_relative_speed

processing_delay = actual_execution_time + timing_delay

timing_delay = 1 / activation_frequency

cpu_load_contribution = function(input_size, input_frequency)

The first formula state that the execution time of an algorithm can be determined by multiplying its base execution time with a delay-factor dependent on the node's CPU load, after which one should scale the result whenever a non-reference node is deployed.
The second formula states that the total time needed to process a message is higher than the actual execution time in timed algorithms. In these cases the worst-case amount of time that could be spent waiting in addition to the actual execution time is equal to one divided by the frequency at which the timed algorithm is activated (third formula).
The final formula specifies that an algorithm's contribution to CPU load is a function of the input size and the frequency at which the input is fed to the algorithm. Note that instead of input, output might be used as a basis for measurement. In our example, we specify whether an algorithm's CPU load contribution is dependent on its input or output.

Finally, we illustrate the concepts presented in this chapter by defining the characteristics of all processing algorithms contained in the fire fighting software system design. The calculation of actual execution times is postponed until the next paragraph. Figure 6-6 and Figure 6-7 provide functions relating an algorithm's total throughput (message size * frequency) to its CPU load-contribution.

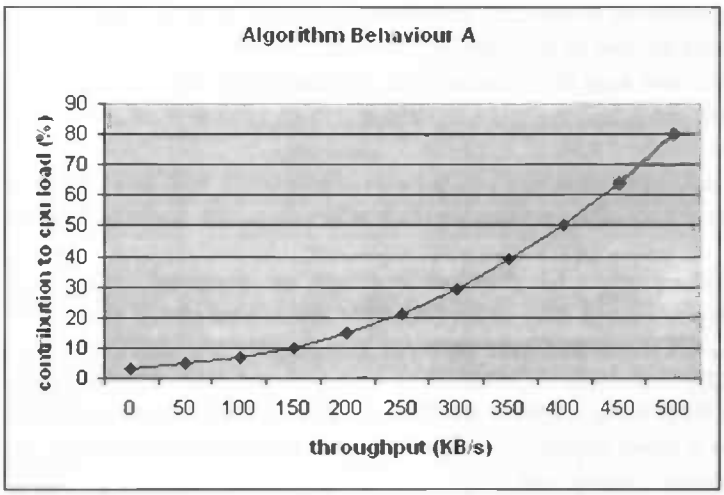

Figure 6-6 Algorithm contribution to load function (1/2)

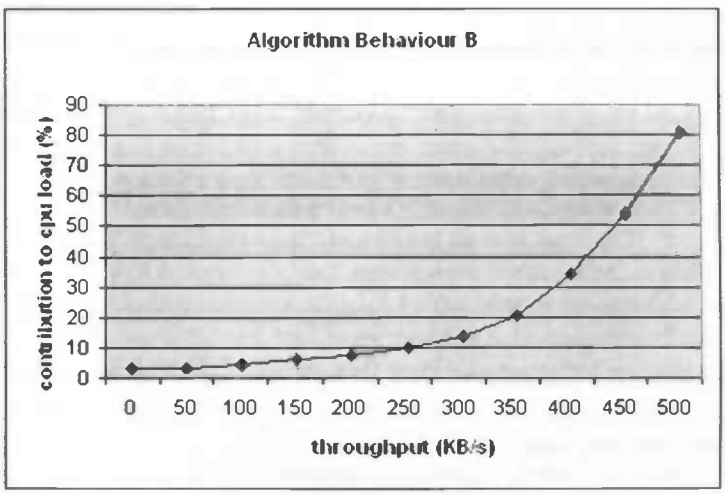

Figure 6-7 Algorithm contribution to load function (2/2)

The following table defines the relevant characteristics for all processing algorithms. Note that the characteristics have been arbitrarily assigned and would normally constitute input to the process.

	Base execution time (ms)	Timed behaviour (Yes/No)	Input/Output Dependent	Load function
smokeReporting	5	No	Output	Figure 6-6
chemicalReporting	8	No	Output	Figure 6-7
smokeEvaluation	10	No	Input	Figure 6-6
chemicalEvaluation	8	No	Input	Figure 6-6
threatAssessment	20	No	Input	Figure 6-7
Chem.Processing	10	No	Input	Figure 6-6
threatProcessing	15	No	Input	Figure 6-6
Sit.Processing	8	No	Input	Figure 6-7
intermediate1	6	Yes	Input	Figure 6-6
intermediate2	5	Yes	Input	Figure 6-7
planning	30	Yes	Input	Figure 6-7
sprinklerControl	5	No	Input	Figure 6-6
planProcessing	5	No	Input	Figure 6-7
feedbackReporting	12	Yes	Output	Figure 6-6

Table 6-5 Processing algorithm characteristics

All relevant algorithm characteristics have now been specified. The next paragraph demonstrates the process of calculating the appropriate performance measures based on the previously defined characteristics.

6.3.5 Calculating the Performance Characteristics of the System

This paragraph presents the process of applying the performance calculations presented in the previous paragraph. First, the contribution of each processing algorithm to the CPU load of their respective nodes is to be determined. These results are then used to calculate the actual execution times of the algorithms, which are then adjusted for timing-related delays.

In a subsequent step, the system's network loads are calculated on the basis of the deployment mapping and the throughput-scenario specified earlier in this section. Finally, the transmission times for all connectors are determined. This process is illustrated in the UML activity diagram in Figure 6-8.

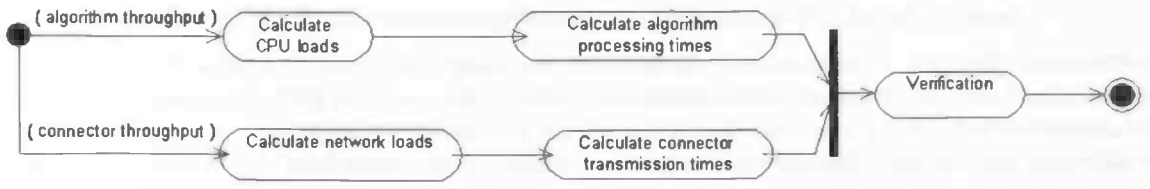

Figure 6-8 An overview of the calculation process

The following table contain the calculation for determining the CPU load of each node. To achieve this, the throughput of each algorithm is first determined (being the sum of all its input/output from connectors), which allows the determination of each algorithm's contribution to CPU load according to the functions given in Figure 6-6 and Figure 6-7. Chapter 4 provided a deployment mapping, on the basis of which the individual CPU load contributions are totalled for each node. Thus, the calculation input consists of a deployment mapping, a throughput-scenario and a load-contribution function for each algorithm.

Node	Algorithm	Throughput based on input-/output-connectors (KB/s)	CPU load contribution (%)	Cumulative CPU load on node (%)
Alpha	smokeReporting	75	6	6
	planProcessing	200	9	15
	feedbackReporting	100	9	24
Beta	chemicalReporting	40+40=80	5	5
	sprinklerControl	150	10	15
Gamma	smokeEvaluation	75+15=90	8	8
	chemicalEvaluation	40	4	12
	threatAssessment	50+15+45=110	6	18
	chem.Processing	100+10=110	9	27
	threatProcessing	40+10=50	5	32
	sit.Processing	100	5	37
	intermediate1	40+50=90	9	46
	intermediate2	20	3	49
	planning	50+450=500	80	129

Table 6-6 Calculating the CPU load for each node

The CPU loads calculated in Table 6-6 are calculated with respect to the reference node. In order to retrieve the actual CPU load values, the results should be adjusted for the relative speeds of the respective nodes they apply to. This adjustment is shown in Table 6-7 and uses the node definitions from Table 6-3 as input. Given the actual CPU load, the execution delay factor for each node may be looked up from Figure 6-2 and Figure 6-3.

Node	Relative speed	Reference CPU load (%)	Actual CPU load (%)	Execution delay factor
Alpha	1x	24	24 / 1 = 24	1.1
Beta	2x	15	15 / 2 = 7.5	1.0
Gamma	1.5x	129	129 / 1.5 = 86	1.9

Table 6-7 Actual CPU loads and the corresponding execution delay factor for each node

The actual CPU loads are required for calculating the processing times of the algorithms. This calculation is shown in Table 6-8. The input to the calculation consists of the execution delay factors from the previous calculation, the base execution times from Table 6-5 and the timing overhead, which is the period of worst-case delay when an algorithm has timed behaviour (discussed in paragraph 6.3.4). The actual execution time is then calculated by multiplying the base execution time with the node delay factor and then dividing it by the relative node speed, whereas the total processing delay is obtained by adding the timing-related overhead.

Algorithm	Execution delay factor	Base execution time (ms)	CPU speed	Actual execution time (ms)	Timing overhead (ms)	Total processing delay (ms)
smokeReporting	1.1	5	1	5.5	0	5.5
chem.Reporting	1.0	8	2	4.0	0	4.0
smokeEvaluation	1.9	10	1.5	12.7	0	12.7
chem.Evaluation	1.9	8	1.5	10.1	0	10.1
threatAssessment	1.9	20	1.5	25.3	0	25.3
chem.Processing	1.9	10	1.5	12.7	0	12.7
threatProcessing	1.9	15	1.5	19	0	19
sit.Processing	1.9	8	1.5	10.1	0	10.1
intermediate1	1.9	6	1.5	7.6	100	107.6
intermediate2	1.9	5	1.5	6.3	100	106.3
planning	1.9	30	1.5	38	100	138
sprinklerControl	1.0	5	2	2.5	0	2.5
planProcessing	1.1	5	1	5.5	0	5.5
feedbackReporting	1.1	12	1	13.2	100	113.2

Table 6-8 Calculating processing time for each algorithm

The performance characteristics relating to the processing algorithms have now been calculated. In the rest of this paragraph, network-related performance characteristics are calculated. In Table 6-9, the network load for each network in the fire fighting system is calculated based on the mapping from connectors onto network connections. This mapping results from deployment mapping provided in chapter 4 and is input to the calculation process.

Network	Connector	Throughput (KB/s)	Cumulative network throughput (KB/s)
Network 1	Chemical Report	40	40
	Chemical Report2	40	80
	Sprinkler Commands	150	230
Network 2	Smoke Report	75	75
	Operational Instructions	200	275
	Situation Report	100	375

Table 6-9 Calculating network throughput

Based on the cumulative network throughput calculated above, the relative network load can be determined. Moreover, using the transmission delay functions defined in Figure 6-4 and Figure 6-5 in paragraph 6.3.3 it is possible to determine the network's transmission delay factor. This factor will be required for determining transmission times in the next step of the process. Table 6-10 lists the relative network load and transmission delay factors.

	Cumulative throughput (KB/s)	Network Capacity (KB/s)	Relative network load (%)	Transmission delay factor
Network 1	230	312	73.7	1.6
Network 2	375	625	60	1.4

Table 6-10 Calculating network transmission delay factors

The final step in the calculation process is to calculate the network transmission time for each connector in the system. It should be noted though, that not every connector involves transmitting data cross a network

connection. Whenever two interacting algorithms are co-located on the same node, they may communicate through a virtual network connection, whose latency is very small compared to the transmission of data between nodes. It was therefore decided to represent the overhead related to node-internal communication by a fixed value. In the calculation below we shall set this fixed value to 2 ms.

Connector	Throughput (KB/s)	Network Capacity (KB/s)	Base transmission time (ms)	Transmission delay factor	Actual transmission time (ms)
Smoke Report	75	625	120	1.4	168
Chem. Report	40	312	128	1.6	205
Chem. Report2	40	312	128	1.6	205
DataAvailable	15	-	2	1	2
GtSmokeData	15	-	2	1	2
SndSmkData	45	-	2	1	2
Threat Report	100	-	2	1	2
Internal2	50	-	2	1	2
Internal3	20	-	2	1	2
Internal4	50	-	2	1	2
Internal5	10	-	2	1	2
Internal6	40	-	2	1	2
Internal7	10	-	2	1	2
Internal8	50	-	2	1	2
Internal9	450	-	2	1	2
Internal10	20	-	2	1	2
SprinklerCmds	150	312	480	1.6	768
Oper.Instruct.	200	625	320	1.4	448
Sit. Report	100	625	160	1.4	224

Table 6-11 Calculating network transmission time

6.3.6 Verifying the Performance Requirements

The previous paragraph presented the calculation of the required software performance characteristics. It is now possible to verify whether the specified throughput and end-to-end time requirements are met by the system.

First, throughput requirements have been specified as an input scenario for the system. Hence, if this scenario can be executed by the modelled system, the requirements are met. To verify whether this is possible, two conditions need to hold:

1. The processing and network resources must not be overloaded by the scenario. In theory, any value under 100 percent is not overloaded. In practice however, network transmission times and processing delays start increasing super-linearly at a much lower system load. We shall define this load to be 70 percent for both network and processing resources. Thus, if the requirements are to be met, the scenario should be not cause loads in excess of 70 percent.
2. The processing algorithms must be able to finish enough processing cycles in order to support the required throughput rate. If, for example, the algorithm needs to produce 50 updates per second, the processing for one update must be less than 1/50 seconds. In other words, the mathematical product of production frequency and actual execution time must not exceed 1 second. The production frequency can be derived by summing the frequencies from all outgoing connectors.

In the calculation below we verify whether the two conditions above hold for the fire fighting system.

Node	CPU load (%)	CPU load < 70%
Alpha	24	Yes
Beta	7.5	Yes
Gamma	86	No

Table 6-12 CPU load requirement verification

Network	Network load (%)	Network load < 70%
Network 1	73.7	No
Network 2	60	Yes

Table 6-13 Network load requirement verification

Table 6-12 and Table 6-13 demonstrate show that the processing resources are not sufficient to meet the required (worst-case) input scenario. Hence, at this stage the outcome of the verification process is known. For the sake of demonstration, we shall of course provide a complete verification.

The next step involves verifying whether the processing algorithms are capable of producing enough updates per second. The input consists of the production frequency of every algorithm (obtained by summing all outgoing connector frequencies) and the previously calculated actual execution times of the algorithms. The product of both inputs should be less than 1 second in order to satisfy the throughput requirement scenario.

Algorithm	Frequency (Hz)	Actual execution time (ms)	Execution time required per second (ms)	Required time < 1 second
smokeReporting	15	5.5	82.5	Yes
chem.Reporting	5+5=10	4.0	40	Yes
smokeEvaluation	15+15=30	12.7	381	Yes
chem.Evaluation	5	10.1	50.5	Yes
threatAssessment	15+20=35	25.3	885.5	Yes
chem.Processing	10	12.7	127	Yes
threatProcessing	10	19	190	Yes
sit.Processing	10	10.1	101	Yes
intermediate1	10	7.6	76	Yes
intermediate2	10	6.3	63	Yes
planning	20	38	760	Yes
sprinklerControl	0	2.5	0	Yes
planProcessing	10	5.5	55	Yes
feedbackReporting	10	13.2	132	Yes

Table 6-14 Processing time requirement verification

Table 6-14 shows that all processing algorithms satisfy the required throughput scenario.

The next step consists of verifying the end-to-end time requirements on the system. This is done by calculating the time a message would require to propagate through the system from the source of a functional flow to its sink. This verification is performed by summing all network transmission times and actual processing times that constitute the functional flow. Note that some algorithms may be contained in the same flow more than once. For the sake of simplicity, we shall assume that the appropriate calculations

are only performed once and therefore that the algorithm's execution time only needs to be added to the total once. The table below contains the calculation for the first functional flow.

Algorithm/Connector	Time overhead (ms)	Cumulatively required time (ms)
smokeReporting	5.5	5.5
Smoke Report	168	173.5
DataAvailable	2	175.5
threatAssessment	25.3	200.8
GetSmokeData	2	202.8
SendSmokeData	2	204.8
ThreatReport	2	206.8
threatProcessing	19	225.8

Table 6-15 Verifying end-to-end time for functional flow 1

The verification for the second functional flow is shown in the following table.

Algorithm/Connector	Time overhead (ms)	Cumulatively required time (ms)
chemicalReporting	4.0	4.0
Chemical Report2	205	209
chemicalProcessing	12.7	221.7
intermediate1	107.6	329.3
Internal5	2	331.3
Internal6	2	333.3
Internal9	2	335.3
planning	138	473.3
Sprinkler Commands	768	1241.3
sprinklerControl	2.5	1243.8

Table 6-16 Verifying end-to-end time for functional flow 2

The requirements specified in paragraph 6.3.2 state that the worst-case traversal time associated with either functional flow should be less than 1 second. In the case of the first functional flow, this requirement is satisfied. The second functional flow does not satisfy this requirement however, due to the high number of timed algorithms it contains.

This paragraph has demonstrated the process of verifying throughput and end-to-end time requirements. The limitation of this process and the calculation process are discussed in section 6.5 at the end of this chapter. The next section briefly presents the way in which we implemented tool-support for the verification process.

6.4 Tool-support for Performance Requirement Verification

The modelling tool we developed supports the verification process by allowing the designer to specify the characteristics of the software and hardware entities that are contained in the system model.

The functional flows are identified using normal UML sequence diagrams, which the user can compose of existing collaborations in the system. The required end-to-end time for the flow can then be specified using the property panel of the tool.

The throughput scenario consists of the collection of all connector throughput specifications as described in paragraph 6.3.2. Thus, the tool allows the user to specify the required throughput characteristics for every connector in the system. These values are entered through the property panel of the application. Figure 6-9 shows how a close-up of the property panel, while the user is entering the throughput requirements for the *Smoke Report*-connector.

Finally, the verification process is started by choosing the appropriate command from the tool's menu-bar. A pop-up window then lists the requirements, whether they are satisfied and the actual value of the throughput and end-to-end time metrics.

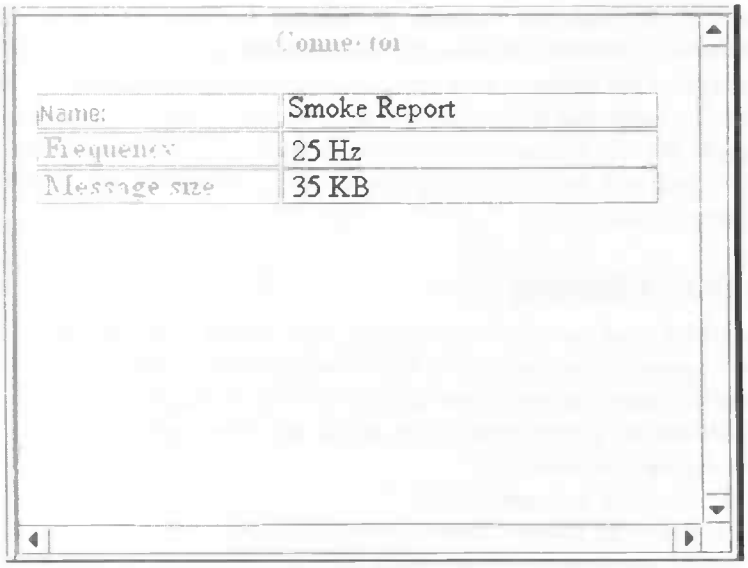

Figure 6-9 Specifying software characteristics and requirements through the property panel

6.5 Summary and Discussion

This chapter presented a process for calculating throughput and end-to-end time characteristics of a software system design. It also demonstrated how these calculations can be applied to verify requirements with respect to these characteristics.

Although we set out to leverage patterns in the process of verification, this goal could not be accomplished. We concluded that patterns could not be assigned any meaningful metrics, which could be leveraged in the verification process. We therefore proceeded by presenting a process that verifies the requirements on a pattern-based design, while not leveraging specific performance-related knowledge about the pattern instances contained in the design.

Although the process is complete and may be deployed using the software tool developed during this project, some issues remained unsolved. We decided to discuss them here, so they may be addressed by future research.

- Some algorithms may occur more than once in one functional flow. We solved this situation by only counting each algorithm once, since the most intensive computation are most likely only performed once and cached afterwards. In complex collaborations however, this may not be true. Therefore, the calculation process may need to be adjusted to take such collaborations into account.
- The calculation process does not yet support multi- or broadcasting, whereas this does occur in many systems. In such situations an algorithm would only need to perform one single calculation for a series of connectors, while networks may also need to transport just one message, rather than one for each connector.
- The process does not yet allow the designer to specify whether algorithms are real-time or non-real-time algorithms. The behaviour of non-real-time algorithms would be very different from that of hard real-time algorithms. Moreover, the calculation model could take real-time operating systems into account, meaning that non-real-time algorithms would be suspended under high system loads.
- The current calculation process implicitly assumes asynchronous communication between algorithms. Synchronous communication can thus not be modelled (i.e. when one algorithm suspends while calling another algorithm).

7 Developing a Pattern-Based Design Tool

The previous chapters have incrementally built-up theoretical concepts that were demonstrated with the help of a case study. In the respective chapters, we referred to a software tool that we developed as a demonstrator and showed how we implemented the theoretical concepts in this tool. This chapter elaborates on the design of the demonstration tool and discusses implementation specifics rather than the details of how specific concepts were implemented as these were discusses in the previous chapters.
The first section briefly presents the open-source tool that we used as the basis for our own software tool. We then continue by discussing the design of the user-interface, after which we present the tool's software architecture and design specifics.

7.1 Open Source: A Starting Point

The software application that we developed during this project was exclusively intended to be a demonstrator of the concept that we developed. Therefore, we tried to minimize the time required for the software implementation, since this would leave us more time for developing new concepts.
For this reason, we decided not to start from scratch, but to search for a suitable software tool that we would be able to extend for our own purposes.
Several options were considered, amongst which:

- Developing a plug-in for Rational Rose
- Using the 'Umlet' open-source software application as a code basis
- Using the 'ArgoUML' open-source software application as a code basis

We quickly ruled out the option of developing a Rational Rose plug-in, since Rose is not an open source application and only allows certain specific degrees of freedom to a plug-in developer. Moreover, plug-in development is a tedious procedure, which requires experience in the field of Microsoft COM-technology.
ArgoUML is a very rich open-source software tool, which made it a very good candidate. Unfortunately, its code base is very large, making the learning-process unnecessarily long and defeating the advantage of starting with an existing code base. Moreover, ArgoUML lacked support for some important UML diagram types (sequence diagrams), which was not acceptable.
Finally, UMLet consists of a rather rudimentary open-source application that has a very small code base and minimal functionality. This makes it possible to quickly understand the code and start developing custom extensions. On the other hand, UMLet does not support any specific type of UML diagram, rather it just provides a drawing pane and different palettes containing UML elements. Thus, it does not prevent the user from mixing different types of diagrams, nor does it build any logical model of what the user is drawing: it is a drawing application, rather than a modelling application.
Given the fact that UMLet is open-source, is written in easy-to-understand Java and has a very small code base, we opted to use this as the basis for our own software tool. The source code for UMLet may be downloaded from [Uml05].

7.2 Design of the User Interface

This section discusses the design of the user-interface for our tool. As we mentioned before, our aim was to minimize the time needed for tool-development, which led us to adopt much of the original UMLet user-interface. This user-interface is shown in Figure 7-1 below and consists of a main drawing area containing, a palette area, an editing area and a source code area. The palette area displays a selection of UML modelling elements, which are referred to as a palette. The user may select one palette from a range of available palettes.
In order to draw any of the elements in the palette in the main drawing area, the user simply double-clicks the chosen element in the palette area. Within the drawing area, modelling elements may be dragged, resized or even connected to each other. As soon as the user selects a modelling element in the drawing area, the editing area shows the modelling element's editable properties. Unfortunately, these properties are

displayed and edited as a text-string, meaning that the user has to use some special keywords and a specific grammar to edit the entities.
Finally, the source code area is meant for generating source code from the application. This is not relevant for our research.

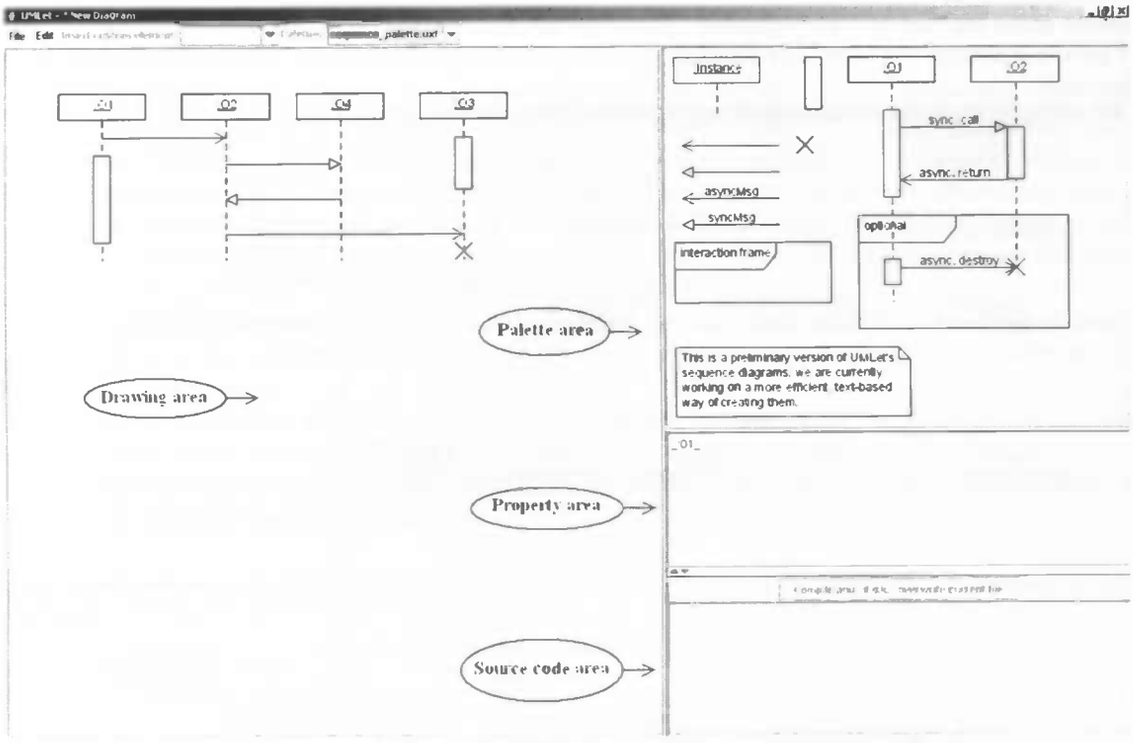

Figure 7-1 Screenshot of UMLet with highlighted functional areas

We now present the modifications that we made to this existing user-interface, as well as a rationale for doing so.
First, we decided to remove the source code area, since it would not be required for our purposes. Then, our modelling approach required the co-existence of several parallel diagrams that would have to be kept synchronized. Moreover, each diagram would only be allowed to contain a restricted subset of all UML modelling elements.
To solve this issue, we undertook the following design changes to the interface:

- We transformed the application from a single-diagram application to one that supports an arbitrary number of diagrams. In the new application, each diagram corresponds to a particular design-view of the software system. These views were defined and described throughout this thesis. To allow the designer some degrees of freedom, we decided that each view can consist of several diagrams. This makes it easier for the designer to separate concerns by drawing several diagrams that together constitute a view. Moreover, we added a dropdown menu to the user-interface, which allows the user to select the desired design view.
- We removed the possibility of freely choosing a desired palette. Instead, we assigned a fixed palette to each type of diagram (i.e. design-view). The appropriate palette is automatically set upon selection of a particular diagram.

As we mentioned before, the editing of modelling entity properties is rather crude in the original UMLet application. We replaced the text-based editing area by a graphical editing area that uses dropdown menus, text-boxes and other so-called *widgets* in order to ease editing. This cancels the need for the user to learn the required keywords and grammar used by UMLet. Moreover, it allows the user to input more complex data using standard widgets (e.g. a file-chooser).

We also decided to add a context-menu to each type of diagram in order to provide actions specific to each type of diagram.
Finally, we added a menu-command for performing the requirement verification process. Although this is certainly not the most elegant way of embedding this functionality in the user-interface, it achieves the goal of demonstrating the verification process without spending too much time designing the user-interface. The results of the verification are presented in a separate window.
Figure 7-2 depicts the new user-interface, which varies only slightly from the original UMLet user-interface.

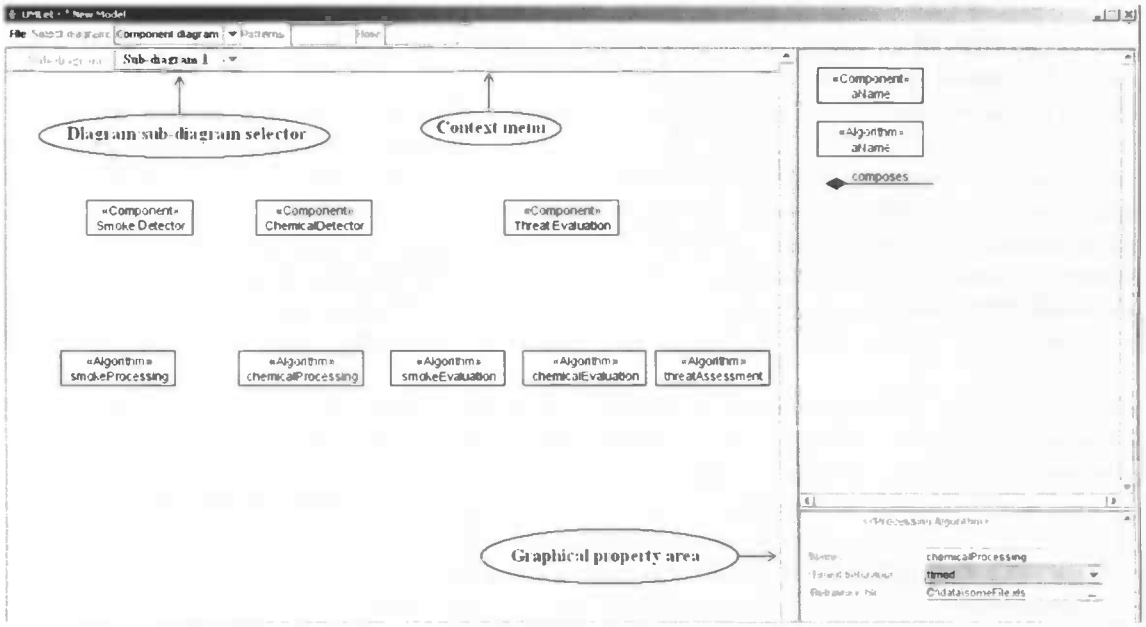

Figure 7-2 Screenshot of our tool with highlighted changes to UMLet

7.3 The Tool Software Architecture

As discussed in the previous sections, the tool is based around a graphical user-interface, which has been extended with a logical model and some performance calculation routines. This section discusses the design details of the software tool. First the development process is briefly presented, after which the modular decomposition of the design is discussed. Finally, the modules are discussed individually.

7.3.1 The development process

As was mentioned before, the development did not start from scratch. Instead, the basis was provided by the UMLet open-source tool. Our aim was to realize a first implementation of the tool relatively quickly, so we would be able to validate our concepts. Moreover, in the early stages of the design we did not yet know what would be required of our tool in the final phases of the project (i.e. we did not have an explicit list of requirements). We therefore decided to initially implement a first version using rapid-prototyping, which also gave us insight into the existing UMLet code and design principles.
After the first version was completed, we developed several subsequent increments, whereby the internal structure was overhauled more than once. Although this was a tedious process and cost more time than we had originally hoped, it allowed us to dynamically adjust the tool as our research progressed.

7.3.2 Modular Decomposition of the Design

The design has been decomposed into the following modules, where each module corresponds to a Java-package in the implementation:

- **com.umlet.core** is one of the original modules from UMLet. It contains the application's root-class as well as most of the code that is needed to initialise and compose the user-interface. It also contains the file input- and output-routines needed to save and load models from/ or to files on disk.
- **com.umlet.core.logicalmodel** was added to the design in the course of this project and contains all classes that constitute the logical model of the application as well as some of the model-related utility-classes.
- **com.umlet.element** is also part of the original design and contains all classes that represent modelling entities on the drawing panels of the application. Thus, if the logical model contains a component, the visual entity representing that component on a particular diagram is defined in this package. The package is divided into two sub-packages, where the **base**-sub-package contains a collection of classes that are used as the super-classes for all other visualization-entities in the application. The **custom**-sub-package on the other hand, defines classes that were derived from these super-classes and generally differ only in a minor way.
- **com.umlet.diagram** contains the classes that define the diagrams that represent the different design views in the application. It also contains the classes that implement the user-interaction behaviour of these diagrams.
- **com.umlet.palette** is similar to the com.umlet.diagram package, except that it defines the palettes belonging to each type of diagram.
- **com.umlet.propertypanel** defines a collection of views that represent parts of the logical model in the property panel of the application.

This decomposition is depicted in Figure 7-3 below. The next paragraph elaborates on the specifics of each package.

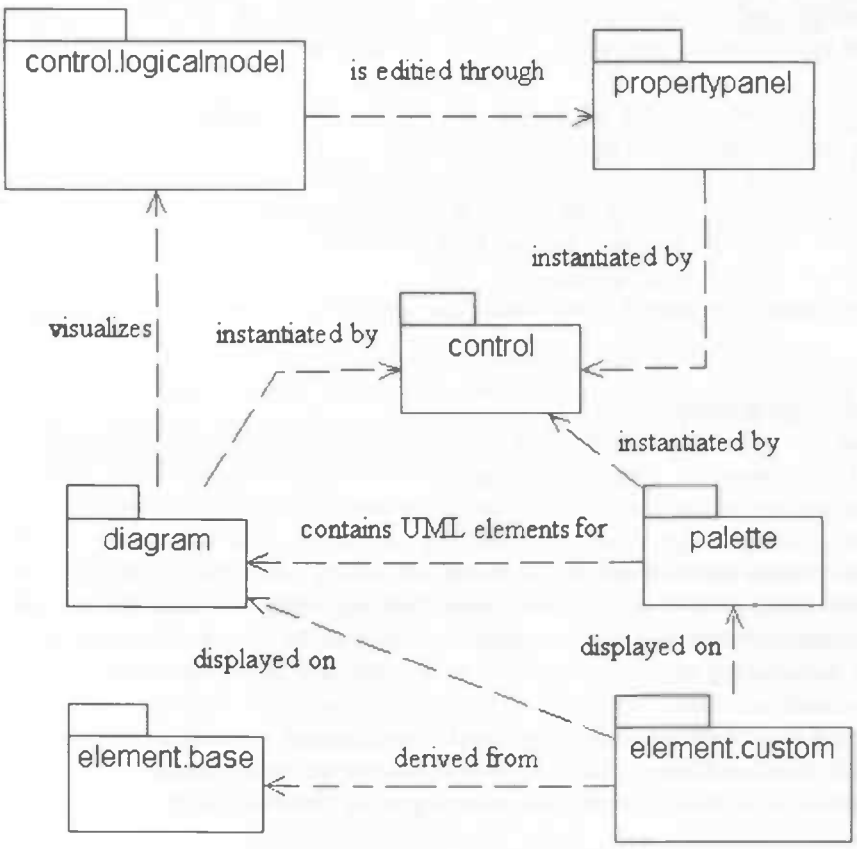

Figure 7-3 Modular decomposition of the software tool

7.3.3 Detailed Discussion of Modules

This paragraph provides a detailed discussion of all modules presented in the previous paragraph. The discussion is not meant as a complete design reference of the application, rather it should be seen as an introduction to the implementation specifics of the application.

com.umlet.control

The main class in this package is *Umlet*, which is the root-class of the application (i.e. it contains the *main()* method). Its main tasks are to instantiate the application window, all the user-interface elements and the logical model. Furthermore, the original Umlet-class had been defined as a static class and many parts of the application depend on this definition. We therefore decided to leave this unchanged, meaning that there can only ever be one instance of the UMLet-class (i.e. it is a singleton). The UMLet-instance is also the owner of the application's logical model.

In the original UMLet-application, all user-interface behaviour had been defined in the *UniversalListener*-class. This meant that this class was listening to all kinds of different events from different sources, making the behaviour very hard to trace and difficult to trace. Therefore, one of the major changes in the architecture was to remove most of the functionality from *UniversalListener* and move it into finer grained classes, each attached to only one event-source. This has made the application far more maintainable and less monolithic. Yet, some rudimentary behaviour, like menu-interaction, is still implemented by *UniversalListener*.

The *Command*-class is also one of the heritages from the organl UMLet-application. It encapsulates the notion of a user-command like an insertion, deletion or change of a modelling element on one of the drawing panels. It is still used in some of the code that has not been altered, but we refrained from using such commands, since it did not provide the flexibility we required. Many of the classes in this package are sub-classes of *Command*, where each of the subclasses represents a specific user-action.

The *CustomXMLHandler*-class is used when reading a model from an XML-file. Every token in the XML-file is handed to this class for processing. It is here, that the logical model and views are reconstructed from XML.

DrawPanel is the base-class for all panels that display UML modelling elements. It contains some rudimentary functionality that all of these panel share.

The *Inserter*-class was added to the original application to solve the problem of inserting a visual entity into an existing view. When performing this operation, one does not want the new entity to overlap any existing entity. Thus, the *Inserter* computes a location for a given new visual element on the current diagram, such that it will not overlap any existing element.

MainPanel represents the main drawing area of the application, where *PalettePanel* represents the palette area.

com.umlet.core.logicalmodel

This package was defined within the core-package, since its contents are considered to be part of the application's core classes (i.e. without it, the application would not be much use). The *LogicalModel*-class represents an instance of the logical model. The logical model constitutes a global data model, fusing all the data from the different design views. Thus, one diagram (design-view) represents only one view upon the logical model. It is the logical model that forms the basis for the calculation and verification process.

All the entities it may contain should inherit from *ModellingEntity*, which allows the model to be extended with new customized modelling elements without changing a lot of code. Moreover, *EntityFactory* should be used for instantiating any *ModellingEntity* as it takes care of all the details of instantiating specific entities.

If any class wishes to be informed of changed to the logical model, it should implement the *ModelObserver*-interface. This interface allows a class to register an interest for model changes and provides a well-defined callback mechanism for change notification (adhering to the Observer-pattern).

com.umlet.element.base

This package defines the super-classes of the visual entities used throughout the application. Every visual entity should inherit from the abstract *Entity*-class, which defines a lot of basic functionality as well as a common interface for all visual entities.
All visual entities that represent lines or arrows (i.e. associations) in the diagrams should not inherit from Entity directly. Instead, they should inherit from *Relation*, which itself is a subclass of *Entity* providing some common functionality for association-type entities.

com.umlet.element.custom
All classes defined in this package are sub-classes from the abstract *Entity*-class. They all represent a specific entity within the logical model and only complement the basic *Entity*-functionality where this is necessary. For example: The *Component*-class in this package describes a visual representation of the *Component*-class from the logicalmodel-package.

com.umlet.diagram
The purpose of this package is to group all classes that implement a specific type of UML diagram together in one place. The *Diagram*-class defined in this package provides an abstract super-class that should be inherited by all diagram implementations. Each diagram corresponds to a view upon the system design. These views have been discussed in detail throughout this thesis. One such a diagram is the *ComponentDiagram*, which shows all components and algorithms contained in the design. The user-interface behaviour of each diagram is defined in a separate class defined in the *listeners*-sub-package. For example, the behaviour with respect to mouse-input for *ComponentDiagram* is defined in the *ComponentDiagramListener*-class in the listener-sub-package. The diagrams may be instantiated using the *DiagramFactory*-class. Finally, a diagram is logically considered to be one large drawing area. Yet, in order to improve user-friendliness we decided that a diagram would be allowed to contain multiple drawing areas. Yet, this is only a visual separation, since logically, the diagram is still interpreted as one monolithic diagram. This is implemented by letting each diagram contain multiple *DrawPanel*-instances.

com.umlet.palette
This package is constructed in a similar way as the diagram-package. It contains a *Palette*-subclass, which defines a super-class for all palettes. Then, the package contains one palette-class for each diagram defined in the diagram-package. When the diagram is selected, the palette is automatically set by the application. The listener-sub-package defines the user-interface behaviour in a similar way as the listeners-sub-package of the diagram-package.

com.umlet.propertypanel
This package consists of a collection of classes that each represent a view of the properties of a particular modelling entity from the logical model. For example: When a component is selected in one of the diagrams, the property panel of the application should display all properties of that component (e.g. its name). Each view should inherit from the *PropertyPanel* abstract class.

7.4 Summary

This chapter discussed several alternatives that were considered for implementing a pattern-based design tool. The tool that we developed is based on the open-source modelling tool UMLet. The development of the software tool was achieved through incremental development, since the exact requirements were note known in advance.
Finally, this chapter presented the modular decomposition of the application, where implementation-details of each module were briefly discussed.

8 Presentation of Research Results & Future Work

This thesis presented a process for expressing behavioural architectural patterns and demonstrated a way of embedding these patterns into pattern-based software system designs. The process is based on a well-defined pattern-based software system meta-model as well as a meta-model for expressing pattern models. In addition, we also presented a way of visualizing the presence of patterns in such designs, using multiple UML-based views.

The goal of leveraging pattern-specific properties in the process of performance requirement verification on pattern-based software system design could not be achieved. We conclude that patterns do contribute to specific non-functional characteristics of a design, yet individual patterns cannot be assigned meaningful properties for estimating overall software system performance characteristics. This is due to the fact that patterns are instantiated using a set of implementation-specific parameters that influence the overall system characteristics. Thus, only instantiated patterns can be evaluated.

A process for performing a verification of throughput and end-to-end time requirements on a pattern-based design was presented in this thesis.

Since our goal was to deliver processes that would be suitable for tool-support, we developed a pattern-based software system design tool that implements the concepts demonstrated in this thesis. The design and usage of this tool were presented in this thesis.

As is often the case, the results of this research leave many opportunities for further research projects. Here, we present several feasible options that would be interesting to explore:

- The current software system meta-model and the verification process are not capable of dealing with multi- or broadcast networks. This constrains the usability of the developed processes for analysing the performance characteristics of realistic large-scale designs.
- The current software system meta-model and the verification process do not allow the modelling of a system that contains both real-time and non-real-time processes. Implementing this feature would be a valuable addition to the verification process.
- The current software system meta-model and the verification process do not allow the modelling of synchronous communication (i.e. when one thread suspends until a method-invocation returns). Implementing this feature would also provide a more powerful framework for expressing system designs.
- The pattern meta-model could be extended to express other kinds of patterns, including structural and creational patterns. Even with respect to behavioural patterns, it could be further enriched: In many cases, whether a message is "optional" depends on whether another message has been implemented. For example: A "reply" may only be omitted from the pattern instance if the corresponding "request" has also been omitted. If such relationships could be captured by the pattern meta-model, it would be capable of expressing more complex patterns as well as a higher degree of pattern flexibility.
- A pattern-language might be developed in order to systematically order all patterns in a pattern library. This would facilitate the process of finding an appropriate pattern to a particular design problem.
- A design-support system might be developed that would aid the designer in implementing a set of desired non-functional properties by exploiting knowledge about available patterns.
- The verification method used to approximate the non-functional properties of a design should be tested with actual software designs and components. Feedback might be used to improve the model in order to get more realistic estimates.
- The calculation and verification process might be extended to include a larger range of non-functional properties.

9 List of definitions

Term	Definition
architectural pattern	Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of rules and guidelines for organizing the relationships between them [Bus96].
component	A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component may be deployed on a node [Omg03].
connector	A relationship between two processing algorithms, where one produces data and the other consumes this data.
design pattern	Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [Gam95].
end-to-end time	End-to-end time is the time it takes for a message to propagate through the software system from the sending processing algorithm through one or more intermediate processing algorithms to the receiving processing algorithm.
functional flow	An ordered sequence of one or more connectors that represent a logical data path through the software system.
metameta-model	A metameta-model defines the constructs and rules needed to create a meta-model.
meta-model	A meta-model defines the constructs and rules needed to create a model.
model	An abstraction of phenomena in the real world
node	A device capable of executing a computer program (e.g. CPU).
non-functional characteristic	A non-functional characteristic of a software system is a characteristic that is not functional in nature. In other words, it is not concerned with what the software does, but how it does what it does. An example of a non-functional characteristic is the timing behaviour of a software entity.
non-functional requirement	A non-functional requirement is a requirement with respect to a non-functional characteristic. A software design generally needs to satisfy a set of non-functional requirements in addition to satisfying the basic functional requirements.
performance requirement	A performance requirement is a non-functional requirement relating to software performance aspects. Note that ISO relates to such requirements as efficiency requirements [Iso01].
processing algorithm	Represents one or more run-time artefacts that reside on one component and collaborate to implement some functionality.
throughput	Throughput is the amount of data per unit of time that is sent from one processing algorithm to another through exactly one connector.

10 Bibliography

[Rie96A] D. Riehle, 'Describing and composing patterns using role diagrams', 1996

[Rie96B] D. Riehle, 'The event notification pattern', 1996

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, 'Design Patterns – Elements of Reusable Object-Oriented Software', Addison-Wesley, Reading, MA, 1995

[Flo96] M. Meijers & G. Florijn, 'Tool Support for Object-Oriented Design Patterns', 1996

[Flo98] D. Gruijs & G. Florijn, 'A Framework of Concepts for Representing Object-Oriented Design & Design Patterns', 1998

[Pag96] B. Pagel, M. Winter, 'Towards Pattern-Based Tools', EuroPLoP '96, 1996

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, 'Pattern-Oriented Software Architecture: Volume1', 1996

[Sch00] D. Schmidt, 'Pattern-Oriented Software Architecture: Volume2', 2000

[Kim95] J. Kim & K. Benner, 'A Design Patterns Experience: Lessons Learned and Tool Support', ECOOP '95, 1995

[Glaxx] M. Glandrup, title still unknown, PhD thesis, (yet to be published)

[Omg00] Object Management Group, 'Meta-Object Facility (MOF) Specification', OMG Specification, http://www.omg.org, 2000

[Omg03] Object Management Group, 'Unified Modeling Language (1.5)', OMG Specification, http://www.omg.org, 2003

[Omg04] Object Management Group, 'Data Distribution Service for Real-time Systems Specification', OMG specification, http://www.omg.org, 2004

[Den03] W. Deng, M. Dwyer, J. Hatcliff, G. Jung, Robby, G. Singh, 'Model-checking Middleware-based Event-driven Real-time Embedded Software', Technical Report from Kansas State University, 2003

[Ale77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel, 'A Pattern Language', Oxford University Press, New York, 1977

[Ale79] C. Alexander, 'The Timeless Way of Building', Oxford University Press, New York, 1979

[Met02] 'What is meta-modeling?', http://www.metamodel.com/staticpages/index.php?page=20021010231056977, 2002

[Bas03] L. Bass, P. Clements, R. Kazman, 'Software Architecture in Practice', Addison-Wesley, Boston, 2003

[Iso01] International Organization for Standardization, 'ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model', ISO/IEC standard, http://www.iso.org

[Uml05] M. Auer, T. Tschurtschenthaler, L. Meyer, 'Umlet 5 – the fast and free UML tool', http://www.umlet.com