This is to certify that the
thesis entitled

DESIGN PATTERNS FOR DEVELOPING DYNAMICALLY ADAPTIVE SYSTEMS

presented by

ANDRES J. RAMIREZ

has been accepted towards fulfillment
of the requirements for the

M.S. degree in COMPUTER SCIENCE

Betty H C Cheng
Major Professor's Signature

12/11/08
Date

DESIGN PATTERNS FOR DEVELOPING DYNAMICALLY ADAPTIVE SYSTEMS

By

ANDRES J. RAMIREZ

A THESIS

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Computer Science

2008

ABSTRACT

DESIGN PATTERNS FOR DEVELOPING DYNAMICALLY ADAPTIVE SYSTEMS

By

ANDRES J. RAMIREZ

As applications grow in size and complexity, and computing infrastructure continues to evolve, it becomes increasingly difficult to build a system that satisfies all requirements and constraints that might arise during its lifetime. As a result, there is an increasing need for the software to adapt to new requirements and environmental conditions after the software has been deployed. Due to their high complexity, adaptive programs are generally difficult to specify, design, verify, and validate. In addition, the current lack of reusable design expertise that can be leveraged from one adaptive system to another further exacerbates the problem. To address this problem, we have developed adaptation-focused design patterns to support monitoring, decision-making, and reconfiguration of adaptive systems where the patterns facilitate the separate development of the functional logic and the adaptive logic. We have also extended the template used by Gamma *et al.* [26] for describing design patterns with *Behavioral* and *Constraints* fields to uniformly present and capture each adaptation design pattern. In addition, the *Related Pattern* section is also used to indicate which adaptation design patterns are commonly used together in adaptive systems. We present these patterns in the context of a modeling-based development process, where we focus on supporting the design of adaptive systems. Furthermore, we provide support for specifying invariant properties of adaptive systems. This thesis describes each design pattern and illustrates how they can be used to construct adaptive and autonomic computing systems. We demonstrate this approach by re-engineering an adaptive news web server from scratch with our design patterns.

To my family, who has always supported me in every possible way.

Acknowledgments

First, I would like to thank my Master's advisor, Dr. Betty H.C. Cheng, for her guidance, support, and expertise. This thesis would not be complete today without her support and feedback.

I would like to take this opportunity to express my gratitude to my committee members: Dr. Betty H.C. Cheng (chairperson), Dr. Philip K. McKinley, and Dr. Sandeep Kulkarni.

I wish also to thank the SENS lab members, in particular Benjamin Beckmann, Eduardo Diaz, Heather Goldsby, and Farshad Samimi for their insightful comments and suggestions.

Lastly, I want to express my appreciation to Amanda Sloan, Ronald Southwick, John Cameron, Erinn Laimon-Thompson, Erick Nieves, and Nicholas Hobart for their constant support throughout this long process.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

As applications grow in size, complexity, and heterogeneity in response to growing computational needs, it is increasingly difficult to build a system that satisfies all requirements and design constraints that it will encounter during its lifetime. Many of these systems are required to run continuously, disallowing long downtimes where humans look for places to modify the code. As a result, it is important to be able to adapt an application's behavior at run time in response to changing requirements and environmental conditions [62]. Recently, IBM proposed autonomic computing [50] in which a system manages itself based on high-level objectives from a systems administrator that promotes properties such as self-management and self-reconfiguration. As a result of their high complexity, adaptive programs and autonomic systems are generally difficult to specify, design, verify, and validate [85]. In addition, the current lack of reusable design expertise that can be leveraged from one adaptive system to another further exacerbates the problem. To address this problem, we have identified design patterns for adaptive and autonomic systems. In order to facilitate their use, we constructed an adaptation design pattern template, much akin to the template used by Gamma *et al.* [26] for design patterns. This thesis describes the adaptation design patterns and how they can be used to construct adaptive and autonomic

systems.

Most adaptive systems, including autonomic systems, comprise three key elements: monitoring, decision-making, and reconfiguration. Monitoring enables an application to be aware of its environment and detect conditions warranting reconfiguration; decision-making determines what set of monitored conditions should trigger a specific reconfiguration response; and reconfiguration enables an application to change itself in order to fulfill its requirements. Not only must developers design and implement each of these elements correctly, they must also carefully determine their interactions. For instance, if the monitoring process fails to report a significant environmental change, then the decision-making process may incorrectly determine whether a reconfiguration is warranted or not. Unfortunately, until recently, most approaches have addressed adaptation in *ad hoc* manners [34]. To address these concerns, researchers provided adaptation-enabling frameworks [12, 21, 27], middleware [54, 64], and language-based support [23, 73]. These approaches, however, tend to be tightly coupled with specific domains or technologies, thus limiting their fitness with respect to the problem being addressed. Design patterns, on the other hand, work at the modeling level of abstraction, thereby possibly increasing the amount of design reuse when compared to other approaches.

Thesis Statement *Based on recurring problem-solution pairs, it is possible to develop adaptation-focused design patterns to support monitoring, decision-making, and reconfiguration of adaptive systems where the patterns facilitate the separate development of the functional logic and the adaptive logic.*

This thesis presents twelve adaptation-oriented design patterns to facilitate the reuse of adaptation expertise. In the spirit of the original design patterns by Gamma *et al.* [26], each of the adaptation-oriented patterns were developed by generalizing

several existing design solutions. For each design pattern presented in this thesis, we use platform-independent models to represent the solution. As a result, our approach does not depend on specific programming languages. In addition, our design patterns separate the adaptive logic from the functional logic by focusing on the recurring challenges fond in monitoring, decision-making, and reconfiguration activities. This separation of concerns facilitates reusing adaptation designs across multiple applications and domains. Similarly, we have observed recurring interactions between monitoring, decision-making, and reconfiguration processes while harvesting each design pattern. This information enables us to suggest which design patterns should be used together. Lastly, we extended the design pattern template introduced by Gamma *et al.* [26] with a constraints field to specify properties that must be satisfied once the design pattern is instantiated. Since our approach is compatible with the high assurance model-based development process previously introduced by Zhang and Cheng [85], automated verification techniques can be used to analyze the instantiated design patterns against safety critical properties.

Harvesting design patterns is a difficult and subjective task for two main reasons. First, it is impractical to examine all available systems and research projects associated with adaptation. Second, some of the surveyed systems had little to no documentation accompanying their design. To ensure that the design patterns harvested were sufficiently mature to aid developers in building adaptive systems, we performed two forms of validation in this work. First, we reviewed previously developed adaptive systems for similar instances of the design patterns. Information from the new instances enabled us to further generalize the solutions and refine the design patterns. Second, we re-engineered an adaptive news web server, originally presented in [16], from scratch using our design patterns. This case study enabled us to evaluate the usefulness of the design patterns in guiding the development of an adaptive system. In addition, this case study was used to compare and contrast

different development processes and final artifacts between our approach and other well-established framework-oriented approaches.

Organization of Thesis The remainder of this thesis is organized as follows. Chapter 2 presents background information for this work, including the different types of adaptation and their semantics, the key objectives of monitoring, decision-making, and reconfiguration within adaptive systems, an introduction to the Zhang-Cheng model-based development process [85], and a brief overview of design patterns. Chapter 3 overviews related work for building adaptive systems. Chapter 4 illustrates the research method used for harvesting and abstracting design patterns. Chapter 5 introduces the adaptation design pattern template, the classification scheme, and the set of design patterns harvested thus far. Chapter 6 expands the model-based development process by illustrating how these design patterns can be integrated into the development process. Chapter 7 presents a proof of concept case study that applies monitoring, decision-making, and reconfiguration design patterns in the development of an adaptive web server. Chapter 8 summarizes our main findings and discusses future directions of work.

Chapter 2

Background

This chapter provides background information on three topics central to the research. First, we overview adaptive systems. This includes a description of the different types of adaptations, the three most common adaptation semantics found in adaptive systems, and the objectives of monitoring, decision-making, and reconfiguration processes within adaptive systems. Second, we introduce the model-based development process previously introduced by Zhang and Cheng [85]. The key ideas, benefits, and steps of the model-based development process are briefly described. Third, we overview the area of software design patterns as well as introduce the design pattern template created by Gamma *et al.* [26].

2.1 Adaptation Overview

A system is considered to be adaptive if it can be reconfigured in response to changing requirements and environmental conditions [67]. Although many forms of adaptations are possible, most adaptive systems perform some form of introspection and intercession [62]. Introspection is the ability for an application to observe its own behavior. Intercession, on the other hand, is the ability for an application to reason about these observations and alter its execution. In some adaptive systems,

a systems administrator may perform either introspection or intercession functions. For instance, a system administrator might be responsible for selecting the appropriate reconfiguration based on the available information. While it is desirable for an adaptive system to automatically perform the tasks of introspection and intercession, it is not a requirement.

Autonomic computing systems were proposed by IBM [50] to overcome the growing complexity of managing systems. While all autonomic systems are adaptive in nature, not all adaptive systems are autonomic. Specifically, in an autonomic computing system, every component is an autonomic element that is capable of introspection and intercession [41]. As a result, an autonomic computing system is self-managed, guided only by high-level objectives from a systems administrator. To accomplish these high-level goals, autonomic systems incorporate self-* properties such as self-configuration, self-healing, self-optimization, and self-protection. Self-configuration refers to the ability to reconfigure components and their interactions. Self-healing refers to the ability of automatically discovering and correcting faults. Self-optimization refers to the ability to optimize behavior based on requirements and constraints. Self-protection refers to ability to detect and fend-off attacks. Throughout this thesis, the term adaptive system is used to include autonomic computing systems unless otherwise noted.

2.1.1 Types of Adaptation

Two general approaches are used to implement software adaptation [62]. The first approach, parameter adaptation, involves adjusting and fine tuning variables and strategies to achieve optimal behavior. While parameter adaptation is relatively simple to implement, the possible range of adaptation scenarios supported by this approach is limited. Specifically, parameter adaptation can switch between existing strategies already built into the system but it may not adopt new strategies and com-

ponents after deployment. The second approach, compositional adaptation, involves adding, removing, and modifying algorithmic and structural components at run time. While compositional adaptation is difficult to implement, it provides greater flexibility in terms of reconfiguration.

The wide spectrum of adaptation techniques developed over the past several years can be classified as either static or dynamic composition [62]. Static composition takes place during development, compile, or load time. Development time composition hard codes adaptive behavior into an application, thereby forcing developers to manually modify the code to incorporate new adaptations. Compile-time composition adapts an application's behavior by recompiling or relinking different components to suit particular environments. Load time composition delays the decision of which component to load until run time. Dynamic composition, on the other hand, refers to tunable and mutable methods applied at run time to alter an application's behavior. Tunable reconfiguration supports the fine-tuning of crosscutting concerns in response to changing environmental conditions. Mutable reconfiguration, the most flexible form of adaptation, supports changes to the entire application, including its functional logic. Sometimes, static composition is referred to as closed-adaptive, and dynamic composition is referred to as open-adaptive [67]. Dynamic composition is more powerful than static composition because it can adopt new strategies at run time that were not available at design time. However, the added flexibility of dynamic composition increases the difficulty associated with ensuring a system's integrity across adaptations. While static composition is simpler to implement and verify than dynamic composition, it can only support adaptation strategies known at design time.

2.1.2 Adaptation Semantics

Three types of adaptive behavior are commonly seen in adaptive programs [84]: one-point adaptation, guided adaptation, and overlap adaptation. As Figure 2.1 illustrates, the key difference between the three types of adaptations is when adaptation can begin and terminate. Zhang and Cheng extended LTL to develop A-LTL (adaptoperator LTL) that precisely defines the semantics [84]. In one-point adaptation, a single transition transfers execution from the source program to the target program. As a result, at one state during the source program's execution, the source behavior terminates and the target behavior commences. In guided adaptation, the source program must first reach a state in which an adaptive transition can be applied without leaving the system in an inconsistent state. To reach such a state, also known as a quiescent state, the source program typically enters a restricted mode in which some features are disabled. Once the source program reaches a quiescent state, a one-point adaptation can be applied to transfer execution to the target program. In overlap adaptation, the source and target behavior may overlap. That is, during overlap adaptation the target behavior commences even though the source behavior has not yet terminated. Eventually the source behavior completes and only the target behavior is observable.

2.1.3 Monitoring

Two primary types of monitoring can be performed by an application. Internal monitoring refers to the measuring and information gathering of how the system itself is performing. External monitoring refers to the measuring and information gathering of how the environment that surrounds the system is behaving. In general, both internal and external monitoring are considered to be computationally expensive because they continuously intrude upon many different portions of an application. Various researchers [30, 43, 50, 59, 81] have proposed to externalize and orthogonalize moni-

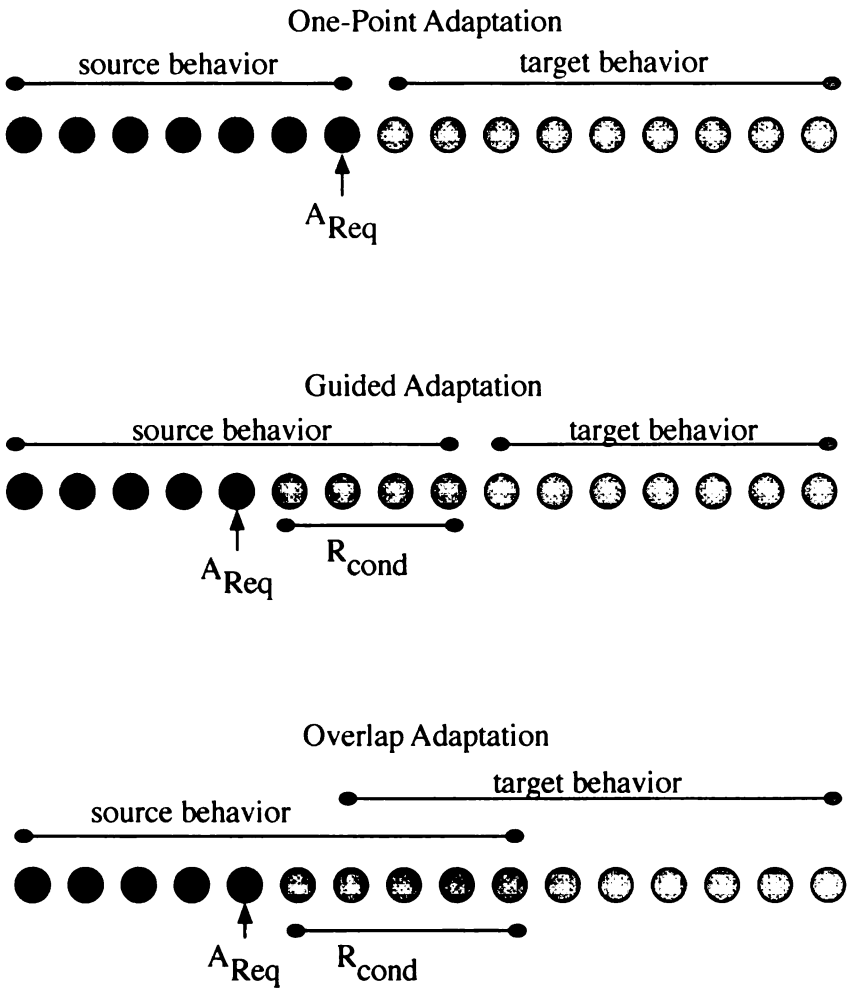

Figure 2.1: Adaptation Semantics.

toring tasks to reduce the cost of monitoring in an application. In these architectures, sensors can either actively probe for the desired information or passively wait for a notification that an event of interest has occurred.

As distributed and mobile applications gained interest, researchers began developing techniques for monitoring components across distributed infrastructures. Garlan *et al.* identified several reasons for why monitoring across a distributed infrastructure is difficult [27, 29, 30]. First, distributed systems comprise heterogeneous platforms. As a result, no specific standard exists for probing components running

on above different platforms. Second, the set of sensors deployed across a distributed infrastructure is likely to be developed by third parties. This can lead to possible interface conflicts between clients, sensors, and components to be monitored. Third, the set of sensors, components being monitored, and clients utilizing the monitored information may change dynamically at run-time. In addition, any distributed sensor is susceptible to security risks and network delays that can render the monitored information useless and possibly even adverse.

2.1.4 Decision-making

A decision-making process is typically responsible for performing two tasks within an adaptive system. First, decision-making processes must determine when the system is not behaving as expected based on the information gathered from the monitoring process [59]. Second, decision-making processes must determine which reconfiguration will yield the desired behavior [29]. Unfortunately, it is usually impossible to predict all possible reconfigurations that may be required ahead of time. As a result, designing decision-making processes that are reliable and correct at all times is a difficult task. For instance, if the monitoring information is delayed in reaching the decision-making process, then the decision-making process may issue an out of date or obsolete adaptation request. Fortunately, the artificial intelligence field has been studying decision-making for many years [70].

Decision-making processes can be classified according to how much knowledge they possess about the environment in which they execute [70]. Full knowledge decision-making processes know, ahead of time, every possible event that may occur. Partial knowledge decision-making processes know only a limited subset of every possible event that may occur. Since uncertainty is present in almost every software system, most decision-making processes fall into the partial knowledge category. This lack of complete information implies that in some situations the decision-making

process will not be able to correct a fault. To address this concern, decision-making processes are sometimes enabled with learning capabilities that enable the adoption of new strategies not known at design time.

2.1.5 Dynamic Reconfiguration

Dynamic reconfiguration involves adding, removing, and modifying components at run time. These components may be localized within a single system or may be distributed across a heterogeneous platform of computing devices. One of the key enabling technologies for realizing dynamic reconfiguration is the concept of component-based design [62]. Component-based design facilitates dynamic reconfiguration in two ways. First, third parties can independently develop, deploy, and compose components. As a result, this increases the number of components available that can be integrated into a system to either augment or replace functionality. Second, component-based design encapsulates a component by exposing only its interface. Thus, different components are interchangeable as long as they provide the same interface.

2.2 Model-based Development Process

Zhang and Cheng [85] previously introduced a model-based development process with the objective of guiding the rigorous development of adaptive programs. Their process separates the adaptive behavior and the non-adaptive behavior specifications of adaptive programs. By doing so, the respective models are easier to specify and more amenable to automated analysis and visual inspection. As Figure 2.2 illustrates, the process starts with high-level goals (G) and progresses through design models (M_i, M_j) to code. The focus of the process was the specification of key properties at each of the major development phases. While the original work used Petri-nets to

illustrate the process, the process itself is compatible with other state-based modeling approaches such as the Unified Modeling Language.

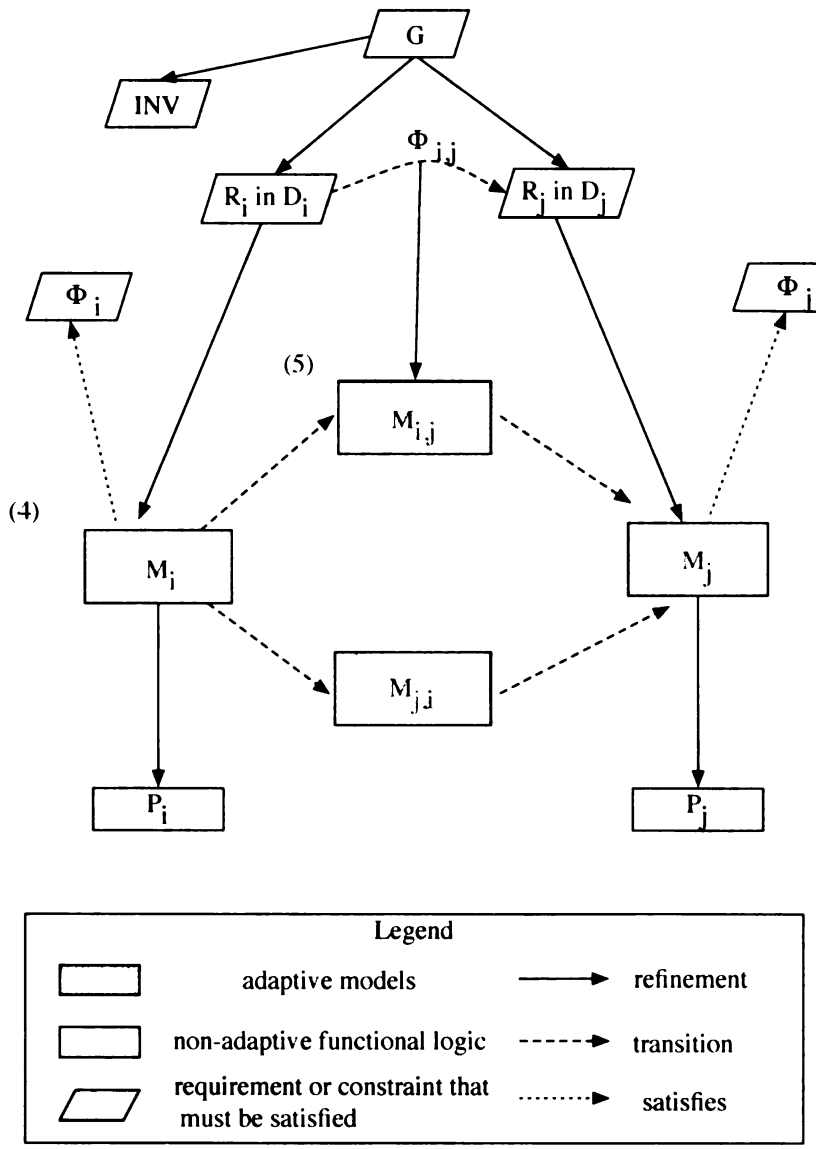

Figure 2.2: Model-based Development Process.

The model-based development process comprises six key steps (Figure 2.2):

1. Specify global properties, INV, using a high-level specification language such as temporal logic.

2. Identify the different domains, D_i, or environmental conditions under which a program with requirements R_i will execute.

3. Using a high-level specification language, specify local properties, Φ_i, for each domain identified in step (2).

4. Build state-based models (M_i and M_j) of the non-adaptive programs in each domain. Simulations and verifications can be applied to verify and validate the models against both the local (Φ_i, Φ_j) and global properties (INV) previously specified.

5. Identify the possible scenarios in which dynamic changes may occur. Build adaptive models, $M_{i,j}$ and $M_{j,i}$, to safely transfer execution from a source program to a target program. Specify transitional properties, $\Phi_{i,j}$ and $\Phi_{j,i}$, to indicate the properties that must be satisfied *during* the adaptation process. As with step (4), simulations and verifications can be applied to verify and validate the adaptive models against global and transitional properties.

6. The state-based models can be used to either generate rapid prototypes or to guide the development of adaptive programs [85].

2.3 Design Patterns Overview

A design pattern is a general and reusable solution to a commonly recurring problem in design [26]. Although Christopher Alexander proposed the idea of design patterns for buildings and towns [2], Gamma *et al.* were able to extend those principles and apply them to the design of object-oriented software. A software design pattern is not a finished design in the sense that it does not provide code nor can it be directly transformed into code. Instead, a design pattern names, abstracts, and

identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design [26]. It is important to note that the purpose of a design pattern is to facilitate the reuse of *successful* designs among developers, not to propose new and innovative approaches that have not been applied in practice. As a result, the main contribution from Gamma *et al.* was capturing proven designs in a new and accessible format, as a catalog of design patterns having a consistent format.

Although Gamma *et al.* did not include any domain-specific design patterns in their pattern catalog [26], they anticipated the need for domain-specific designs that could be reused. Each domain tends to be characterized by specific contexts and requirements, most of which are learned through experience. Domain-specific design patterns can leverage and reuse the experience gained from designing and building similar applications. In recent years, researchers have cataloged design patterns for a wide range of domains including software architectures [11], resource management [53], concurrent and distributed systems [10], embedded systems [55], and so forth.

2.3.1 Template Description

A design pattern has four essential elements [26]. First, a *pattern name* is a handle that can be used to describe a design pattern, its solutions, and consequences. The pattern name should be as descriptive as possible and ideally limited to one or two words. Second, the *problem* describes when to apply the pattern. It provides a detailed description of the design problem being addressed and its context. Third, the *solution* describes the elements that make up the design, their relationships, responsibilities, and collaborations. The *solution* should be sufficiently abstract to be applicable to different situations. Fourth, the *consequences* are the results and tradeoffs of applying the design pattern. This field is essential for evaluating design alternatives and determining whether it is beneficial to apply the design pattern

or not. The design pattern template *fields* will contain and organize the relevant information to describe each of these four essential elements.

The original design pattern template proposed by Gamma *et al.*comprises fourteen different fields given below with brief descriptions of each field:

1. **Pattern Name:** Serves as a unique handle to identify the design pattern.
2. **Classification:** Facilitates the organization of design patterns based on their level of abstraction and purpose. Some possible classifications include structural, behavioral, and creational.
3. **Intent:** Provides a brief description of what the design pattern does.
4. **Also Known As:** Other well-known name identifiers for the pattern.
5. **Motivation:** Presents a scenario that illustrates a design problem and how the class and object structures in the design pattern solve the problem.
6. **Applicability:** Defines the context under which the design pattern can be applied.
7. **Structure:** Provides a graphical representation of the classes and their relationships in the design pattern.
8. **Participants:** Describes the responsibilities for each class and object.
9. **Collaborations:** Presents how the participants collaborate to accomplish their responsibilities.
10. **Consequences:** Lists the known advantages and disadvantages of applying the design pattern.
11. **Implementation:** Indicates any known pitfalls, hints, or techniques that a developer should be aware of when instantiating the design pattern.

12. **Sample Code:** Presents code fragments to illustrate how the design pattern might be implemented.

13. **Known Uses:** Lists examples of the design pattern found in real systems.

14. **Related Patterns:** Lists other design patterns that are closely related, as well as other design patterns that should be used in conjunction with the current pattern.

Chapter 3

Related Work

This chapter presents work that is related to building adaptive systems. First, we describe some of the earliest approaches and attempts to build adaptive systems. Second, we present some of the efforts by the system's community at creating middleware to facilitate the design and construction of adaptive systems. Third, we overview several software engineering efforts to efficiently build and manage adaptive systems. Some of these approaches include architectural description languages, adaptation frameworks, and aspect-oriented techniques. Finally, we present a few language-based approaches that support the construction of adaptive systems.

3.1 Early Approaches

Adaptive computing systems have steadily gained attention throughout the past several years. Nonetheless, the concept of dynamic reconfiguration has existed since the earliest days of computing. Some of the first attempts at self-modifying code supported run-time program optimization and explicit management of physical memory [62]. These programs were frequently complex and difficult to understand because developers lacked the proper support to abstract the low-level details of dealing with adaptation. New approaches and techniques for building adaptive systems eventually

began to emerge. For instance, developers began applying error detection and error handling capabilities to render systems self-adaptive [32]. While these types of approaches helped demonstrate that adaptation was both possible and powerful, they were tightly coupled with source code, application-specific, and typically applied in an *ad hoc* fashion. As a result, the first generation of adaptive programs were considered difficult to write, debug, and maintain.

3.2 Systems Approach

The first generation of techniques and tools created to enable both static and dynamic adaptive behavior in applications mostly focused on the implementation level. This strategy proved particularly problematic for adaptive systems in terms of development and maintenance. For example, with these approaches, building an adaptive system required identifying all the corresponding places where a system might need to reconfigure and manually introducing the changes. Likewise, correcting errors entailed identifying where the problem occurred, what caused it, which changes were required, and where modifications needed to be performed. These first-generation *ad hoc* approaches were not well-suited for efficiently building and maintaining complex adaptive systems. As a result, research on adaptive systems gradually shifted towards developing more efficient adaptation schemes that reduced the burden on developers.

3.2.1 Middleware

Recent research by the systems community has focused on extending middleware approaches to provide adaptation services [18, 64, 36]. Middleware refers to the various layers of services that separate applications from operating systems and network protocols [62]. Schmidt [74] decomposed middleware into four layers comprising a host-infrastructure layer, a distribution layer, a common layer, and a domain-specific

layer. In its most basic form, the different service layers of adaptive middleware serve as a level of indirection by intercepting and modifying messages as needed. One benefit of middleware-based adaptation is that it shields developers from dealing with resource distribution and platform heterogeneity, thus alleviating tasks previously relegated to developers. However, middleware tends to be highly domain-specific, and as a result, may not be readily available for many application domains.

The Mobility and ADaptability enAbling Middleware (MADAM) [31, 64] project provides a general component model and middleware infrastructure that supports various adaptation styles for mobile applications. Adaptation occurs seamlessly and without user intervention in reaction to context changes. The MADAM middleware infrastructure supports three types of functionalities. First, it monitors, detects, and reasons about context changes. Second, it decides which adaptation to perform in response through a utility theory approach. Third, it implements the adaptation choices through dynamic composition. To support these functionalities, MADAM operates on an architectural model of the application at run time. This provides the adaptation middleware information about the application structure, its constraints, and the various context and resource dependencies that exist.

Sadjadi *et al.* developed the Adaptive CORBA Template (ACT) to enable run-time improvements to CORBA applications in response to changing requirements and environmental conditions [72]. ACT transparently weaves adaptive code into an object request broker (ORB) at run time. The woven code intercepts and modifies the requests, replies, and exceptions that pass through the ORBs. One of the benefits of ACT is that it is language and ORB independent. Thus, developers can use ACT to build an object-oriented framework in any language that supports dynamic loading of code. Although the ACT infrastructure introduced a slight overhead, experimental results showed it was insignificant when compared to the highly flexible adaptations it offered.

3.3 Software Engineering-Based Approaches

As developers gained experience with these initial approaches they realized that building adaptive systems from scratch was impractical. The second generation of tools and techniques for building adaptive systems would have to address the following requirements. First, the specifics of adapting a system should be as transparent as possible to developers. Second, adaptation mechanisms should be reused whenever possible. Third, the adaptive logic should be minimally invasive upon the functional logic. Finally, these approaches should be applicable to both new systems, as well as legacy systems. Ideally, developers would be able to create efficient adaptive applications without explicitly implementing all the required adaptation mechanisms. Based on these requirements, researchers provided architectures, frameworks, and language-based support for systematically building adaptive systems.

3.3.1 Architectural-based Techniques

Separating the adaptive logic from the functional logic simplified the development and maintenance of adaptive systems while promoting software reuse. Researchers presented several architectures for cleanly separating concerns in adaptive systems [6, 27, 42, 67]. In particular, Oreizy [67] proposed an infrastructure that supported two simultaneous processes in self-adaptive software. While the first process dealt with the evolution of the system, the second process dealt with the cycle of detecting changing circumstances and planning responsive modifications. Meanwhile, other researchers [22, 30, 32] explored the tasks of monitoring and decision-making and how they interacted within adaptive systems. Garlan and Shaw [27, 75] further subdivided the architecture of adaptive systems by applying control theory approaches to adaptive systems. More recently, new approaches [6] have further extended these architectures by decentralizing each process across distributed infrastructures. As

a result, the most common architecture found in adaptive systems today comprises monitoring, decision-making, and reconfiguration processes.

Another area of software engineering adaptive research has focused on using architectural description languages (ADL) to capture and manage system evolution and system adaptation. Architecture-based approaches for self-adaptive software usually view systems as networks of concurrent components bound together by connectors [67]. Architectural-based representations of a system shift focus away from source code to coarse-grained components and their interconnections. In these representations, a component is responsible for implementing application behavior and maintaining state information. Connectors, on the other hand, offer transport and routing services for messages or objects. In architectural-based approaches, dynamic reconfiguration involves not only adding, removing, or modifying components and their connections, but also managing the evolution of the system and the consistency of the component-connector representations. Recently, Kramer and Magee proposed a three-layer architecture-based model for self-adaptive systems [58]. The lowest layer, the component control layer, is responsible for the creation, interconnection, and deletion of components. The change management layer comprises a predefined set of reconfiguration plans that can be applied to repair the application at run time. The highest layer, the goal management layer, creates new change management plans as needed, thus facilitating the overall evolution of the system and its reconfiguration mechanisms.

Three examples of architectural-based approaches at self-adaptive software include Taylor *et al.*'s C2 [67], Gorlick's Weave [35], and Garlan *et al.*'s Rainbow [29]. C2 [67] composes systems as a hierarchy of concurrent components bound together by connectors such that a component within the hierarchy can only be aware of components residing at the same level or beneath it. Weaves, on the other hand, is a dynamic, object-flow-centric architecture targeted towards applications with large

volumes of data flow and real-time constraints [35]. One interesting characteristic of Weaves is that no component in a network knows the sources of its input objects or the destination of its output objects. While this approach provides a large degree of flexibility, it is also susceptible to security risks. For instance, a component may not be able to authenticate the origins of a particular message if it does not know the source of its inputs. Lastly, Rainbow [29] is an adaptation framework that uses models not only to represent the system's architecture, but also to select which reconfiguration will yield the desired behavior.

3.3.2 Frameworks

Adaptive software research has also focused on creating and using frameworks for building adaptive systems [12, 27, 49]. A framework is a set of cooperating classes that make up a reusable design for a specific class of software [26]. Among other things, the framework dictates the overall architecture of the application and its thread of control. This often leads to an inversion of control in which developers write code that gets called by the framework. One of the major benefits of frameworks is that it provides large amounts of reusable code, thereby enabling developers to build applications faster. Nevertheless, some creative freedom is lost because many design decisions have already been made by the framework developers [26]. Additionally, framework-based applications are sensitive to changes in the framework's interface.

Garlan *et al.*'s Rainbow is an architecture-based self-adaptation framework with reusable infrastructure [27]. Their approach uses external adaptation mechanisms for two reasons. First, it facilitates the application of their reusable infrastructure to legacy applications without being invasive upon the functional logic. Second, it allows developers to specify and reuse adaptation strategies for multiple system concerns. Rainbow supports distributed component monitoring, probe and gauge deployment, architectural-based system representation and adaptation strategies, and effectors to

reconfigure the system.

Rainbow's adaptation infrastructure incorporates control theory concepts [75]. First, probes monitor the system and report values to gauges and gauge consumers. These values are then related to properties of the architectural model. Each time an architecture property is updated, the architecture is analyzed to ensure no constraint is violated. If a constraint has been violated, then the architecture must be reconfigured. Rainbow uses utility theory-based approaches to select a reconfiguration plan. Finally, through Rainbow's infrastructure and effectors, the reconfiguration is executed on the system's architecture.

3.3.3 Aspect-oriented Programming

Another interesting approach for building adaptive systems is based on the aspect-oriented programming (AOP) paradigm introduced by Kiczales [51]. AOP provides abstraction techniques and language constructs to manage crosscutting concerns [62]. AOP defines an aspect as code that implements a crosscutting concern. Using an aspect weaver, AOP inserts aspects into specific code locations, called pointcuts, during compilation. As a result, not only does AOP decouple crosscutting concerns from the functional logic, it also localizes them. This separation facilitates the consistent maintenance of an application as it evolves.

Dynamic recomposition can exploit the AOP paradigm because most adaptations are crosscutting in nature. Yang *et al.* [82] introduced a systematic two-step process that defined where, when, and how adaptations would be incorporated into an application. First, aspects are used to extend a program with an adaptation infrastructure and entry points into the adaptation kernel. Then the adaptation kernel uses a rule-based engine to determine if an adaptation should be performed and executes the corresponding actions if necessary.

3.3.4 Software Reconfiguration Design Patterns

Gomaa *et al.* proposed several design patterns for reconfiguring software architectures at runtime [34]. The four design patterns they introduced specify the behavior required to dynamically reconfigure specific types of architectures. In particular, the design patterns describe the reconfiguration of master/slave, centralized, server/client, and decentralized architectures. For each design pattern, Gomaa *et al.* identify when it is safe to perform a reconfiguration and provide hierarchical UML state diagrams illustrating the necessary behavior. Although these reconfiguration design patterns are helpful to developers implementing dynamically adaptive systems from scratch, their contents are not organized in template format and they do not address safety and assurance.

3.4 Language-based Approach

Sadjadi *et al.* proposed a transparent reflective aspect programming (TRAP) technique for enabling adaptive behavior on legacy applications [73]. TRAP was designed around the four techniques of aspect-oriented programming, behavioral reflection, component-based design, and adaptive middleware. Briefly, TRAP works as follows. First, developers identify potential points of adaptation (hook points.) Adaptive infrastructure is then woven into the legacy system at the corresponding hook points. The hook points are then monitored for adaptation conditions. If a condition that requires an adaptation arises, then a rule-based decision-making process determines the appropriate code to swap in or out. As a result, TRAP could be used to enhance legacy code with adaptive behavior without explicitly altering the functional logic. TRAP/J was a specific incarnation of TRAP based on the Adaptive Java Language [48].

Chapter 4

Process Used for Developing Design Patterns

This chapter introduces the research methods designed to harvest, evaluate, and refine the adaptation design patterns presented throughout this thesis. First, we motivate why it is important to have a methodology that is able to systematically select, analyze, and abstract recurring solutions as design patterns. Second, we state the goals for this methodology and how they relate to the thesis statement. Third, we present the sequence of steps that were used to develop each adaptation design pattern. We describe and each of the key steps in this iterative process, and, if applicable, we also indicate any limitations they may have.

4.1 Motivation

The survey conducted on background and related work indicate two particular trends in the software engineering community with respect to adaptive and autonomic systems. First, the software engineering community has begun to address adaptation concerns by providing frameworks [27], middleware [64], and language-based support [73] for enabling applications with adaptive and autonomic behavior. The

majority of these approaches deal with adaptation at the implementation phase, thus making the assumption that the requirements, designs, and constraints are already understood. Second, with the exception of a few projects [10, 34], little attention has been given to reusing adaptation expertise at the modeling level through the use of design patterns. Compared to other approaches, design patterns promote creative freedom by imposing fewer initial constraints on design decisions [26].

This thesis combines the key ideas of specializing design patterns [34] and organizing their contents through templates [10, 26]. First, the design patterns presented by Gomaa [34] are focused solely on reconfiguring different types of software architectures at run time. Instead of providing design patterns for general problems in reconfiguration, their design patterns address very specific problems that arise during reconfiguration. Second, the design patterns presented in [10, 26] make explicit use of a design pattern template for structuring and presenting relevant information. Synthesizing concepts from both approaches, this thesis proposes to harvest design patterns that are specific to recurring problems encountered in adaptive applications and structuring their contents with the use of a template. As a result, these design patterns may be combined according to specific requirements and constraints in order to yield customized adaptive applications.

Harvesting design patterns is a difficult and subjective process because there is no existing set of metrics that quantify the quality of a design pattern. Those who harvest design patterns, for instance, must address many subjective questions while generalizing solutions to a recurring problem. For instance, exactly when does a problem get classified as being recurrent in a given domain? Which data sources should be used to find the desired design patterns? Are the solution models sufficiently abstract to be applicable to a wide range of systems yet, at the same time, specific enough to guide developers throughout the implementation phase? Since the answers to these questions will vary between developers, it is important to have an

iterative process that documents and justifies each decision taken while harvesting design patterns. The feedback loop will enable the refinement of both the results and the process itself.

4.2 Goals

Recall that the objective of this thesis is to investigate recurring problems encountered in adaptive and autonomic systems and promote the reuse of successful solutions. In order to make progress towards this objective and produce design patterns that are valuable to both experienced and inexperienced developers working with adaptive and autonomic systems, the process used for harvesting design patterns must:

- Look for good solutions and attempt to generalize them.
- Incorporate the use of a template to organize the information contained in the design pattern.
- Produce specific design patterns for monitoring, decision-making, and reconfiguration.
- Analyze interactions between monitoring, decision-making, and reconfiguration design patterns.
- Evaluate and gradually refine the resulting collection of design patterns.
- Facilitate the use of formal analysis tools for determining whether a solution model satisfies certain properties.

4.3 Harvesting Process

A data flow diagram of the iterative process used to harvest and refine the adaptation design patterns is shown in Figure 4.1. This data flow diagram illustrates how information is gradually processed and transformed into an adaptation design pattern. To start the process, developers must identify and define a recurring problem that is related to adaptation. One possible way to identify recurrent problems is to analyze research publications with common topics related to monitoring, decision-making, and reconfiguration. Based on the recurrent problem identified, developers must also determine what is the intent, context, and motivation for addressing the problem. A clear definition of these fields will narrow the search for existing solutions to the recurring problem.

Next, developers need to select the relevant data sources that will be analyzed and generalized into design patterns. Three types of data sources are available for this task: commercial applications, open-source implementations, and research projects. In general, some data sources are better suited than others for harvesting design patterns. For instance, commercial applications typically incur problems related to high costs and proprietary rights. Likewise, open-source implementations typically have little, if any, documentation. Research publications on the other hand are accessible, well documented, and peer reviewed for quality purposes. As a result, the solutions gathered from research publications typically bear more weight in the overall adaptation design patterns than commercial and open-source projects do.

Many research areas in computer science address issues related to monitoring, decision-making, and reconfiguration. For example, monitoring is frequently encountered in distributed systems and safety critical research communities. Likewise, decision-making techniques are practically ubiquitous throughout the artificial intelligence field. Additionally, reconfiguration techniques are now starting to emerge in new research communities that focus on safely adapting applications at run time.

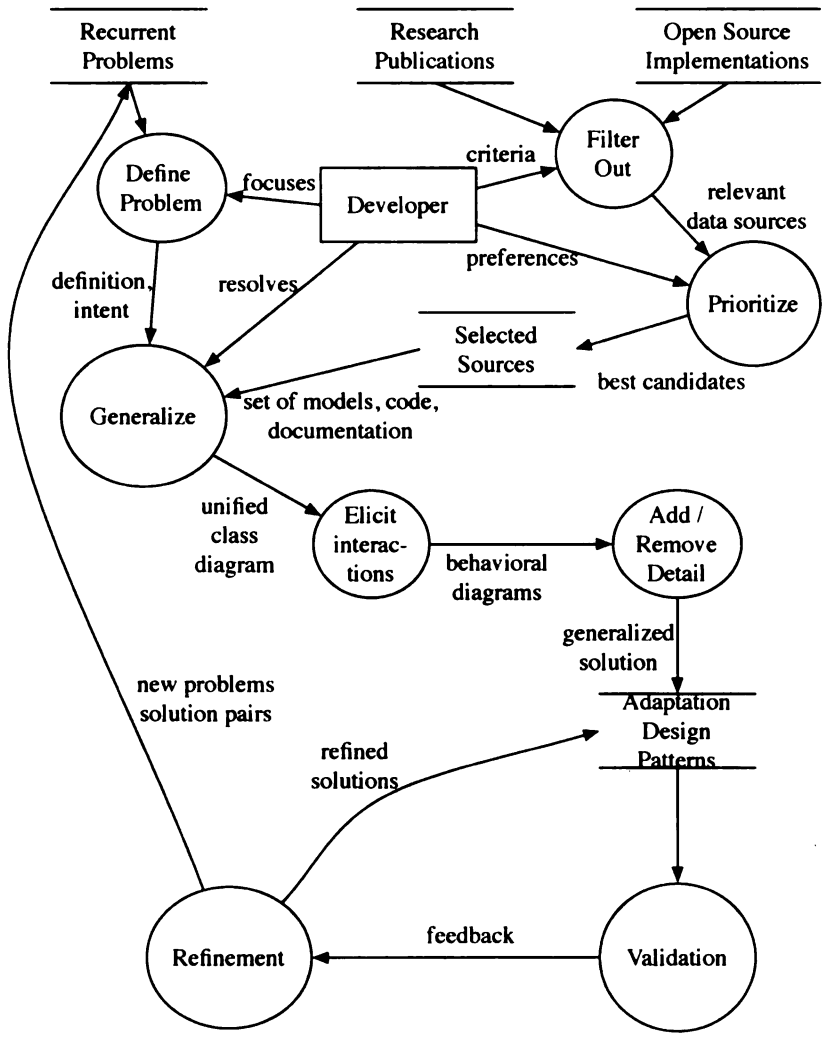

Figure 4.1: Harvesting process data flow diagram.

Although this approach manually selects data sources, it is conceivable to automate some parts of this process by incorporating techniques from domains such as data mining.

Generalize Solutions. The process of abstracting and unifying different solutions into one representative design pattern is similar to the process of model fitting found in mathematics. Specifically, given a set of points, a mathematician derives a line that will best represent those points and, at the same time, predict where future points may lie with some degree of certainty. Similarly, recurring instances of the solution are scattered throughout different research projects and implementations. Design patterns are meant to generalize these instances while simultaneously guiding the development of future instances. As with model fitting in mathematics, some points may be of more interest than others. This results in two complimentary approaches for abstracting and unifying different solutions into a design pattern.

The conceptual difference between the two approaches can be seen in Figure 4.2. The first approach exploits the discovery of a particularly good solution to the problem being addressed. Specifically, a developer creates a preliminary draft of a design pattern based on a good solution [1] and then refines it as further instances are found. As a result, that solution bears more weight on the finalized design pattern than the other solutions do. The second approach, on the other hand, considers a suite of solutions all at once. Thus, while the first approach is biased towards one particular solution instance, the second approach weighs every solution more equally.

Regardless of which conceptual approach is undertaken, several steps must be performed to abstract and unify various solutions into a design pattern. First, developers must determine the similarities and differences between the different solutions being abstracted. That is, both structural and behavioral diagrams need to be analyzed to discover important classes, the types of associations between them, their

[1] We consider a good solution to be one that has been applied several times with positive results.

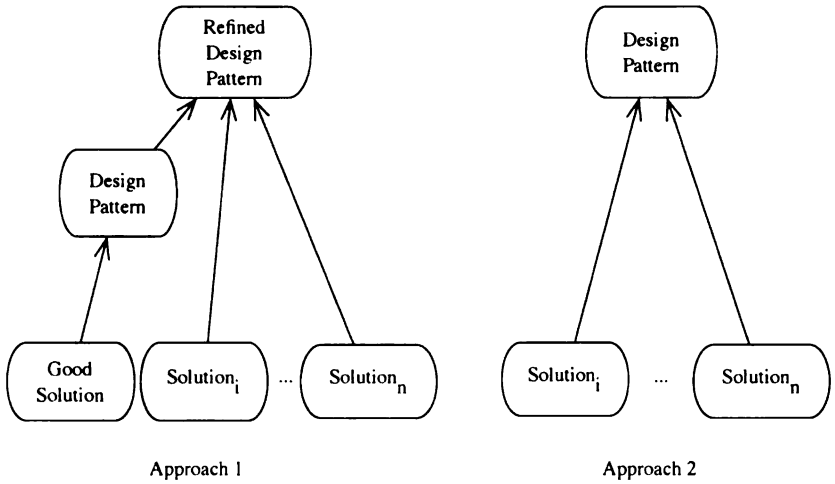

Figure 4.2: Abstraction Process Diagram

multiplicity, their responsibilities, how they interact, and what the constraints are. If these diagrams are not available, then developers can either manually derive them by studying the code or automatically generate them by applying reverse engineering techniques. The similarities and differences between these diagrams will help developers determine when to add or remove details in the design pattern.

Validation. Two forms of validation are used to estimate the quality of the resulting design patterns. The first form of validation consists of searching for additional instances of the design pattern in previously unexamined data sources. This validation is performed in the early stages of design pattern development. Each new instance encountered strengthens the validity of the solution as well as provides additional information for refining the design pattern. The second form of validation consists of applying matured design patterns to a case study application. As the design patterns are instantiated, the resulting models can be formally analyzed through tools such as Hydra [63] and the Spin model checker [47]. If errors are found, then the design pattern can be revised accordingly.

Lastly, the analysis gathered from the two forms of validation can be used to refine the process for harvesting design patterns. To facilitate the fine-tuning of this methodology and improve its results, there are three feedback points within the process. First, if the respective designs cannot be generalized to a common solution, it might indicate that the problem definition is too broad. As such, it may need to be narrowed towards a more specific recurring problem instead. Second, other research disciplines could be explored while searching for solutions. For instance, instead of focusing solely on intelligent systems, the scope of resources used to harvest a decision-making design pattern could be broadened to include biologically-inspired systems as well. Finally, case studies can refine the observations gathered with respect to how a set of design patterns interact with one another.

Chapter 5

Adaptation Design Patterns

This chapter introduces the template used to describe adaptation design patterns, enumerates the list of patterns, and presents a criteria for organizing, classifying, and using the patterns. In addition, this chapter contains the complete description of all adaptation design patterns identified thus far.

5.1 Adaptation Design Pattern Template

This thesis uses a template similar in style to that used by Gamma *et al.* [26] in order to facilitate the understanding and application of the adaptation design patterns. We have modified the original design pattern template in a few aspects to address the needs of adaptive systems. Table 5.1 overviews the adaptation design pattern template. First, the *Known As*, *Implementation*, and *Sample Code* sections have been removed. The *Known As* section is irrelevant as, to the best of our knowledge, the majority of the design patterns presented in this thesis have not been previously documented. The *Implementation* and *Sample Code* sections are too specific for the design patterns presented in this thesis. Second, the template has been extended with a *Behavior* and *Constraints* sections. The *Behavior* section presents either sequence and/or state diagrams that illustrate sample behavior. The *Constraints* section uses

Linear Temporal Logic (LTL) and A-LTL [84] and textual descriptions to specify properties that must be satisfied by the instantiated design patterns. Note that in some instances the LTL and A-LTL formula contain specific function invocations as "boolean operators". When these operators are referenced, they implicitly encode an predicate assertion that returns a boolean value corresponding to whether the function has been invoked or not. Lastly, although Gamma *et al.* used the Object Modeling Technique (OMT) [69] to represent structural and behavioral diagrams, we used the Unified Modeling Language (UML) to give structural and behavioral information about each design pattern. Specifically, structural diagrams are represented through UML class diagrams (for monitoring and decision-making patterns) and UML component diagrams (for reconfiguration patterns). Likewise, UML statecharts are used depict a pattern's behavior.

5.2 Adaptation Design Patterns Catalogue Overview

Table 5.2 gives an enumeration of the twelve adaptation design patterns harvested thus far along with their intentions. These design patterns have been identified from analyzing several adaptive systems and related projects. It is important to consider the following when evaluating these design patterns. First, these patterns capture only a fraction of what an adaptation expert might know. Other areas related to monitoring, decision-making, and reconfiguration could use design patterns as well. Second, this thesis includes only designs that have been applied more than once in different systems. Third, although most of the design patterns are applicable to a wide range of adaptive systems, some are applicable only to a specific set of adaptive systems. The *Intent* section can be used to determine the applicability of each design pattern. In addition, the *Related Pattern* section can be used to determine which

Pattern Name	The pattern name uniquely identifies and describes the pattern.
Classification:	The classification facilitates the organization of patterns based on the purpose of the pattern.
Intent:	A brief description of the problem(s) that the pattern addresses.
Context:	Describe the conditions and context in which the pattern may be applied.
Motivation:	A description of sample goals and objectives of a system that motivate the use of the pattern. Use-cases and use-case-diagrams describe goals of the pattern application.
Structure:	A representation of the classes and their relationships depicted in terms of UML class diagrams (for monitoring and decision-making patterns) and UML component diagrams (for reconfiguration patterns).
Participants:	Itemizes the classes and objects that are included in the adaptation design pattern and lists their responsibilities.
Behavior:	Provides an illustrative representation of scenarios for class and object interaction. Also gives a description of the behavior of the pattern by using sample or high-level, abstract UML state and sequence diagrams.
Consequences:	Describes how objectives are supported by a given pattern and gives the trade-offs and outcomes of the pattern application.
Constraints:	Contains LTL templates and a prose description of constraints that must be satisfied by a given design pattern implementation.
Related Patterns:	Additional design patterns that are commonly used in conjunction.
Known Uses:	Lists the sources where the design pattern was harvested from.

Table 5.1: Adaptation Design Pattern Template

design patterns are commonly used together. More details for each design pattern is provided in Section 5.5.

Name	Description
***Sensor-Factory (41)*:**	Deploy sensors across a distributed infrastructure and probe components.
***Reflective Monitoring (50)*:**	Perform introspection on a component and dynamically alter a sensor's behavior.
***Content-based Routing (59)*:**	Route monitoring information based on the content of the message.
***Case-based Reasoning (68)*:**	Rule-based approach to selecting a reconfiguration plan.
***Divide and Conquer (78)*:**	Systematically decompose a complex reconfiguration plan into simpler reconfiguration plans.
***Adaptation Detector (88)*:**	Interpret monitoring data and determine when an adaptation is required.
***Architecture-Based (97)*:**	Provide an architecture-based approach for selecting reconfiguration plans.
***TradeOff-Based (106)*:**	Systematically select a reconfiguration plan that best balances multiple objectives.
***Component Insertion (115)*:**	Safely insert and initialize a component at run time.
***Component Removal (125)*:**	Safely remove a component at run time.
***Server Reconfiguration (135)*:**	Safely reconfigure a server - client component architecture at run time.
***Decentralized Reconfiguration (145)*:**	Safely insert and remove components from a decentralized component architecture at run time.

Table 5.2: Current list of adaptation design patterns

5.3 Classifying Adaptation Design Patterns

It is important to classify and organize design patterns in order to facilitate their use. Our adaptation design patterns can be classified using two orthogonal classification schemes. The first option is to classify the patterns according to their

purpose: creational, structural, or behavioral [26]. Creational patterns focus on object creation. Structural patterns focus on describing the composition of classes or objects. Behavioral patterns depict the method of interaction and distribute the responsibility of classes or objects. Thus far, we have only identified structural and behavioral adaptation design patterns.

The second option is to classify the patterns according to their adaptation functions: monitoring, decision-making, and reconfiguration. Monitoring patterns focus on probing components and distributing the information across a network to interested clients. Decision-making patterns focus on identifying when a reconfiguration is needed and selecting a reconfiguration plan that will yield the desired behavior. Reconfiguration patterns focus on safely adding, removing, or modifying components at run time to adapt a program. Thus far, we have identified several design patterns for each area.

Monitoring and decision-making patterns are what we consider to be adaptation-enabling design patterns. These design patterns provide the necessary infrastructure to perform introspection and intercession. Although monitoring and decision-making design patterns do not reconfigure an application, without these, a developer would have to manually perform these tasks at run time. Therefore, reconfiguration patterns depend upon monitoring and decision-making design patterns.

These design patterns can also be used to aid in the design and construction of autonomic computing systems comprising some number of autonomous elements. Each autonomous element is instrumented with monitoring, decision-making, and reconfiguration processes (see Figure 5.1.) To build an autonomic element, at least one design pattern from each category must be applied.

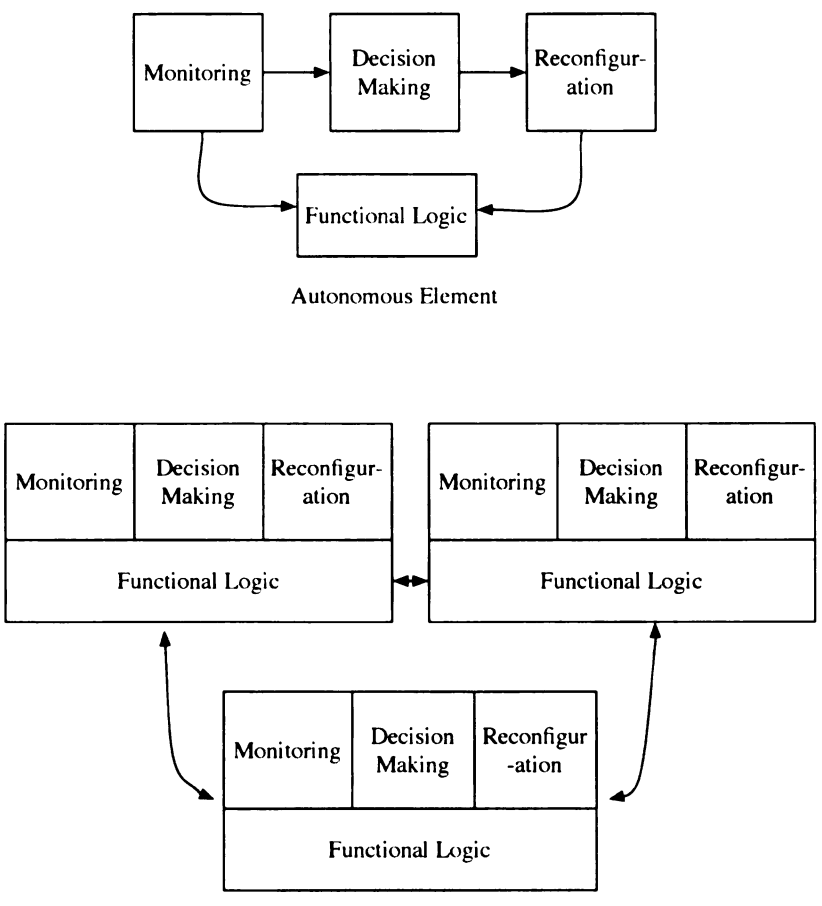

Figure 5.1: Applying Patterns to Self-Adaptive and Autonomic Systems

5.4 Adaptation Design Pattern Roadmap

In order to develop an adaptive system using our approach, developers must carefully integrate several of the design patterns presented in this thesis. We have observed how different patterns interact together while harvesting the individual design patterns. Based on these observations, we recommend certain sets of design patterns to be used together. To determine which design patterns work well together, developers can refer to either the *Related Patterns* section of each design pattern or to Figure 5.2. For instance, all monitoring and decision-making design patterns use the *Adaptation Detector (88)* pattern to interpret the data and determine when a reconfiguration is warranted. Likewise, every decision-making design pattern can use any of the reconfiguration design patterns presented in this thesis.

5.5 Adaptation Design Pattern Repository

This section gives a detailed description of the adaptation design patterns discovered thus far. The names of the design patterns are denoted in *italics*, and the fields of each design pattern are given in a san serif font. Method names and messages are denoted in *italics*.

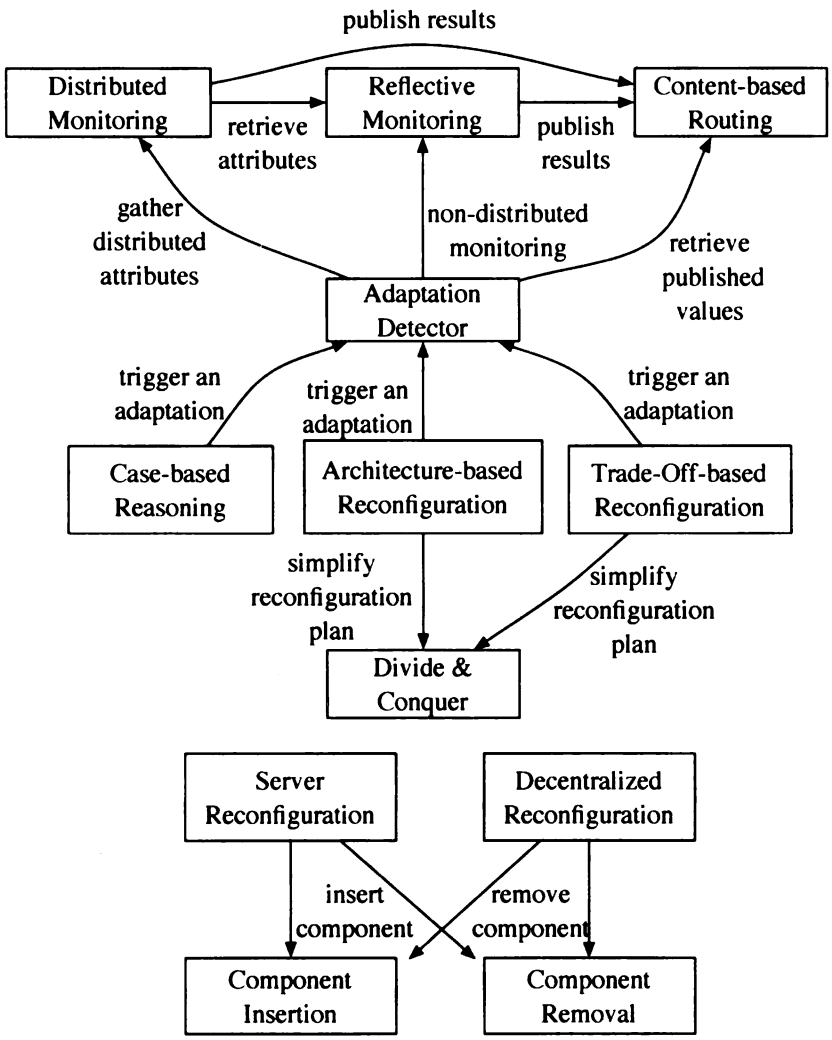

Figure 5.2: Adaptation Design Patterns Commonly Used Together.

5.5.1 *Sensor-Factory (41)* Pattern

Classification:

Structural - Monitoring.

Intent:

Systematically deploy software sensors across a network to probe distributed components.

Context:

The *Sensor-Factory (41)* Pattern may be used when:

- the components to be monitored are distributed.
- each component provides an interface that can be probed for the required information.

Motivation:

External adaptation mechanisms must effectively collect information about the running system to properly evaluate a system's operational status [30]. The objective of the *Sensor-Factory (41)* design pattern is to manage distributed sensors across a networked environment such that they may probe distributed components. The *Sensor-Factory (41)* design pattern captures the structural relationship between sensors, clients, and components. By decoupling sensors from clients and components, the monitoring infrastructure is flexible and more amenable to change.

Figure 5.3 shows a use-case diagram of the *Sensor-Factory (41)* Pattern. Two goals of this pattern are to deploy a software sensor across a network and to probe a distributed component.

Use-Case:	Request sensor
Actors:	Client
Description:	A client requests a sensor to monitor some component.
Includes:	Search registry, Add sensor

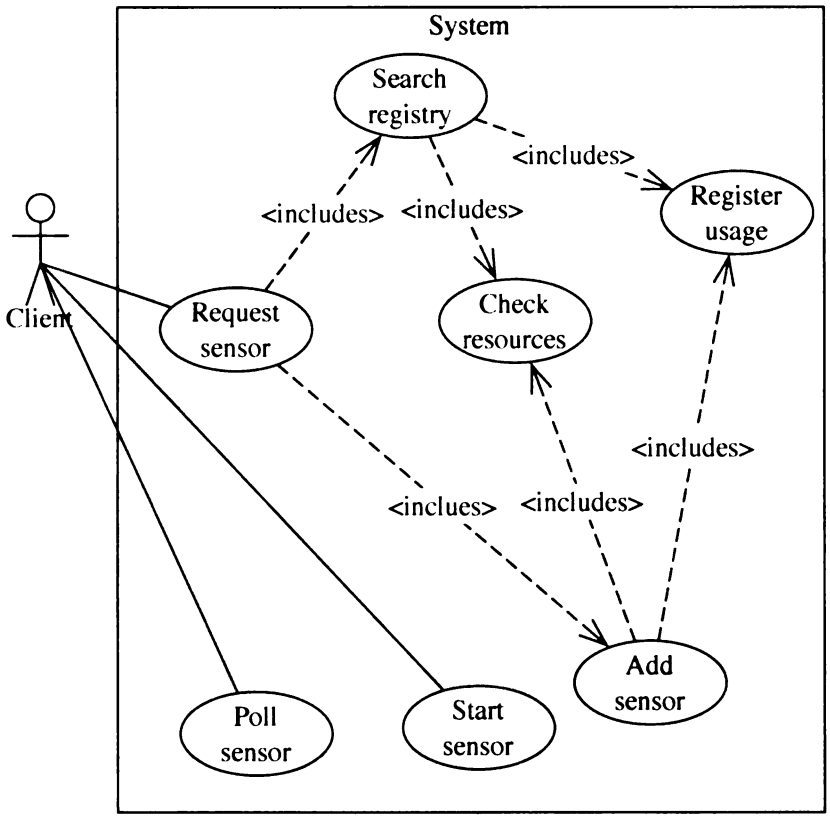

Figure 5.3: UML use-case diagram of the *Sensor-Factory (41)* Pattern

Use-Case:	Search registry
Actors:	-
Description:	Determine whether any deployed sensor already provides the needed information.
Includes:	Register usage, Check resources

Use-Case:	Check resources
Actors:	-
Description:	The system determines if an existing sensor can be shared between more than one client or whether a new sensor can be deployed across the network without violating any QoS constraint.
Includes:	-

Use-Case:	Register usage
Actors:	-
Description:	Records the relationship between a sensor, the component it is monitoring, and the clients it is servicing.
Includes:	-

Use-Case:	Add sensor
Actors:	-
Description:	Creates a new instance of a sensor in the network. Both a client and a component are assigned to this sensor.
Includes:	Register usage, Check resources.

Use-Case:	Start sensor
Actors:	Client
Description:	A sensor is initialized and activated before it begins transmitting data.
Includes:	-

Use-Case:	Receive data
Actors:	Client
Description:	A client receives data from the sensor. This service supports both push and pull actions on a sensor.
Includes:	-

Structure:

A UML class diagram for the *Sensor-Factory (41)* Pattern can be found in Figure 5.4.

There are two different types of sensors that can be found in this design pattern. Simple Sensors can handle booleans, integers, and real data types. Complex Sensors, on the other hand, are capable of either reporting more complex data types or of aggregating the outputs of a Simple Sensor. Regardless of their specific type, Simple Sensor and Complex Sensor both inherit the interface from the Abstract-Sensor abstract class. As a result, they should provide an interface with basic functionalities such as

pushing and polling for data.

Participants:

- Abstract-Sensor: Simple Sensor and Complex Sensor both inherit from this abstract class. As a result, these sensors share an interface to common operations such as pushing and pulling data.

- Client: This class is used to represent any component that needs to perform either internal or external monitoring.

- Complex-Sensor: This type of sensor contains greater computing resources on-board than a Simple Sensor does. As a result, a Complex Sensor is capable of reporting complex data types, aggregating various Simple Sensor data feeds, and performing on-board computations.

- Registry: This class is responsible for tracking deployed sensors across the network. Each entry should at least record the sensor name, the sensor type, the Client it is providing data to, and the component it is monitoring. Additionally, this class provides a search functionality based on the available fields.

- ResourceManager: This class has two responsibilities. First, it determines if an existing sensor can be shared with one or more clients. A sensor can be shared as long as it does not violate any existing constraint. Second, it determines if the system has enough resources to deploy a new sensor across the network.

- Sensor-Factory: Clients must interact with this class in order to gain access to a sensor. It regulates the dynamic access and management of sensors across a network.

- Simple-Sensor: The most basic sensor available. It is capable of reporting boolean, integer, and real data types. Additionally, it can be configured to

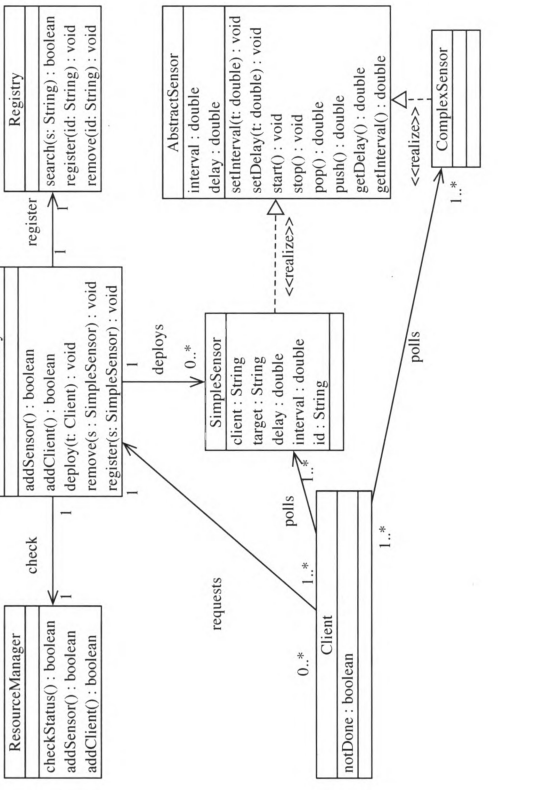

Figure 5.4: UML class diagram of the *Sensor-Factory (41)* Pattern

poll a component at different intervals and periods.

Behavior:

Figure 5.5 shows a UML sequence diagram for an example of the *Sensor-Factory (41)* Pattern in a distributed monitoring system. The Client requests a Simple Sensor (an active networked sensor) from the Sensor-Factory. The Sensor-Factory first determines whether an existing Simple Sensor is already providing the desired information. If not, then Sensor-Factory checks the Resource Manager to determine if another Simple Sensor can be deployed across the network without breaking any quality of service constraints. If so, then Sensor-Factory creates a new instance of Simple Sensor and initializes it to some default sensor setting. Sensor-Factory then notifies the Client that the Simple Sensor is ready for use. Client polls the Simple Sensor until it is done monitoring.

Consequences:

1. This design pattern reuses the provided functionality and interface of a distributed component to extract the desired attributes. However, if a component's interface is excessively polled, then it could interfere and alter the component's behavior.

2. Different types of sensors can be systematically deployed at run time while providing a flexible monitoring infrastructure that is amenable to adaptation.

3. This design pattern ensures system integrity by accessing a component's attributes through its interface.

4. The Registry and Resource Manager share existing sensors whenever possible. This avoids wasting resources in the form of duplicated sensors.

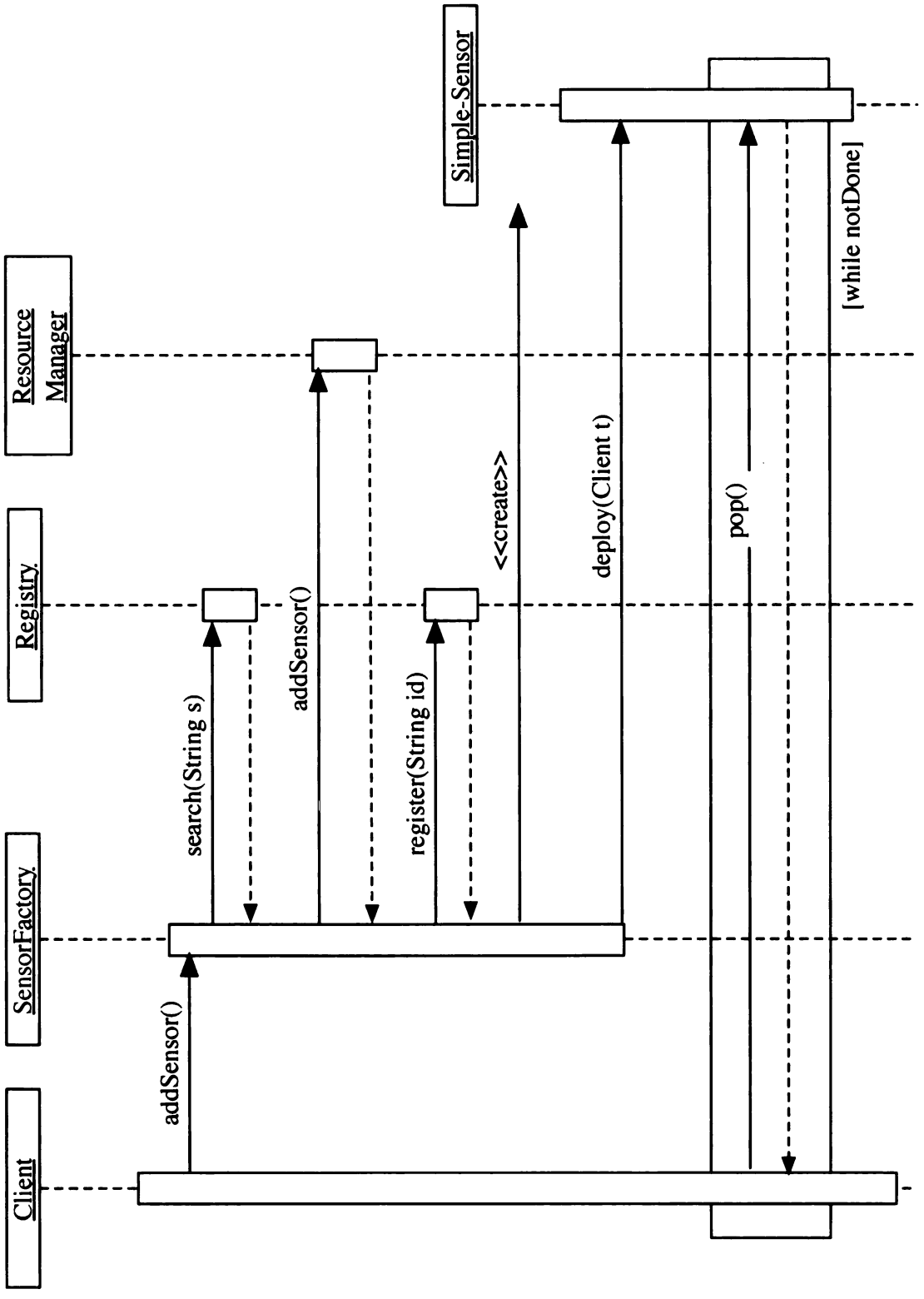

Figure 5.5: UML sequence diagram example of the *Sensor-Factory (41)* Pattern

5. This design pattern introduces a management layer between a Client and a sensor. This additional overhead may degrade performance.

6. Monitoring is only supported for those components with an interface to the required attributes.

Constraints:

- **Property 1:**

  Globally, it is always the case that if Resource Manager denies a Client request for a sensor, then Sensor-Factory does not create a sensor for the Client.

  $\Box$ ((ResourceManager.deny(Client) $\rightarrow$

  $\neg$ Sensor-Factory.createSensor(Client))

  This safety property ensures that Sensor Factory obeys the recommendations provided by the Resource Manager. Otherwise, if the system's resources are not properly maintained, then the entire application may suffer as a result.

- **Property 2:**

  Globally, it is always the case that if Client requests a sensor to Sensor-Factory, then Sensor-Factory will eventually grant access to a sensor.

  $\Box$ ((Client.request(sensor)) $\rightarrow$

  $\Diamond$ (Sensor-Factory.grant(Client))

  This liveness property guarantees that a Client will eventually get access to a sensor.

Related Design Patterns:

- ***Adapter* Design Pattern [26]:**

  This pattern can enable the interaction between a Client and a sensor whenever their interfaces are incompatible.

- **Reflective Monitoring (50) Design Patterns:**

  This pattern can be used whenever a component does not provide an interface to the required attributes. Such values may be accessible through Introspection.

- **Adaptation Detector (88) Design Pattern:**

  This pattern is responsible for interpreting the results provided by a sensor and determining when an adaptation is required.

Known Uses:

- REsource MOnitoring for network-aware applicationS [20].
- Rainbow Adaptation Framework [27, 30].
- A Distributed Monitoring Service Architecture (MonALISA) - via SNMP [66].
- SNMP4J-Agent [24].

5.5.2 *Reflective Monitoring (50)* Pattern

Classification:

Structural - Monitoring.

Intent:

Provide mechanisms to observe the internal state of a component and to change the monitoring scheme dynamically.

Context:

The *Reflective Monitoring (50)* Pattern may be used when:

- a component needs to be monitored and it does not provide an interface to the desired attributes.
- monitoring schemes need to be dynamically altered.

Motivation:

An external adaptation mechanism must be able to observe a component's internal state to properly evaluate its operational status [30]. Observing the internal state of a component may be difficult due to visibility constraints imposed by encapsulation techniques. Specifically, the only way to access private attributes in a component is through a predefined interface. The *Reflective Monitoring (50)* design pattern instruments components with introspection capabilities such that monitoring processes can probe a component's internal attributes. In addition, through the use of proxies and indirection, the *Reflective Monitoring (50)* design pattern facilitates the dynamic reconfiguration of monitoring schemes transparently.

Figure 5.6 shows a use-case diagram of the *Reflective Monitoring (50)* Pattern. Two goals of this pattern are to observe the internal state of a component and to dynamically reconfigure the monitoring scheme.

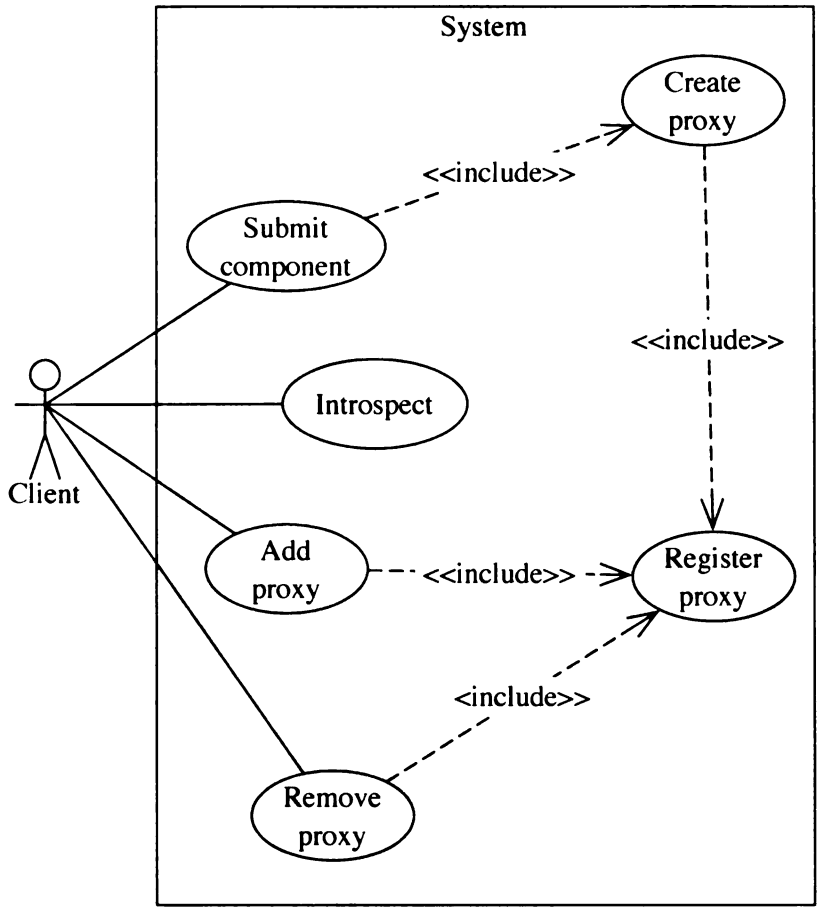

Figure 5.6: UML use-case diagram of the *Reflective Monitoring (50)* Pattern

Use-Case:	Submit Component
Actors:	Client
Description:	A client notifies the system it wants to monitor a specific component.
Includes:	Create proxy.

Use-Case:	Create proxy
Actors:	-
Description:	Creates a proxy object that supports the interface and functionality of the specified component and returns it to the Client transparently. This proxy also supports probing for internal attributes.
Includes:	Register proxy.

Use-Case:	Introspect
Actors:	Client
Description:	A client probes the proxy through its interface.
Includes:	-

Use-Case:	Register proxy
Actors:	-
Description:	Registers and tracks proxies across the system.
Includes:	-

Use-Case:	Add proxy
Actors:	Client
Description:	Adds a monitoring proxy to a specified proxy chain in the system. Specifically, it augments monitoring functionality.
Includes:	Register proxy.

Use-Case:	Remove proxy
Actors:	Client
Description:	Removes a proxy from the system.
Includes:	Register proxy.

Structure:

A UML class diagram for the *Reflective Monitoring (50)* Pattern can be found in Figure 5.7.

A reflective monitoring approach must overcome encapsulation techniques that hide private attributes from external entities. Two important constructs are required for reflective monitoring. First, the monitoring process must have access to a meta-data construct that provides structural and behavioral information about a particular object. Second, the monitoring process must have a transparent mechanism to intercede during an object's normal behavior and introduce additional behavior. The *Reflective Monitoring (50)* Design Pattern achieves reflective behavior through the

Metaobject and Proxy constructs. A Proxy can be created to supersede a Target object through the information contained in a Metaobject. The Proxy will provide the same functionality and interface to a Client as the Target did. In addition, the Proxy will provide trapping mechanisms such that additional behavior can be introduced at run time without affecting a Client.

Note, many programming languages now provide support for meta-objects and proxies [19]. Typically, these constructs will facilitate reflective programming. It is important, however, to check any constraints that might be imposed by such constructs.

Participants:

- Client: This class is used to represent any component that needs to perform either internal or external monitoring.
- InvocationHandler: This interface must be supported by the Proxy object. This enables a Proxy to perform computations before and after a specific method is invoked on the Target object without affecting the functional logic.
- Manager: Creates a Proxy object in response to a Client's request to monitor a Target object. It is responsible for adjusting the necessary permissions so that the Proxy is able to retrieve a Target's attributes.
- Meta-object: Provides information about a particular type of object such as its structure, attributes, modifiers, and interfaces. The Meta-object must be regularly updated to reflect the most recent information.
- Monitor: This class represents different monitoring schemes that can be invoked on a particular Target by a Proxy. For instance, a more complex monitoring scheme might include a chain of Proxy objects that first retrieve the attribute,

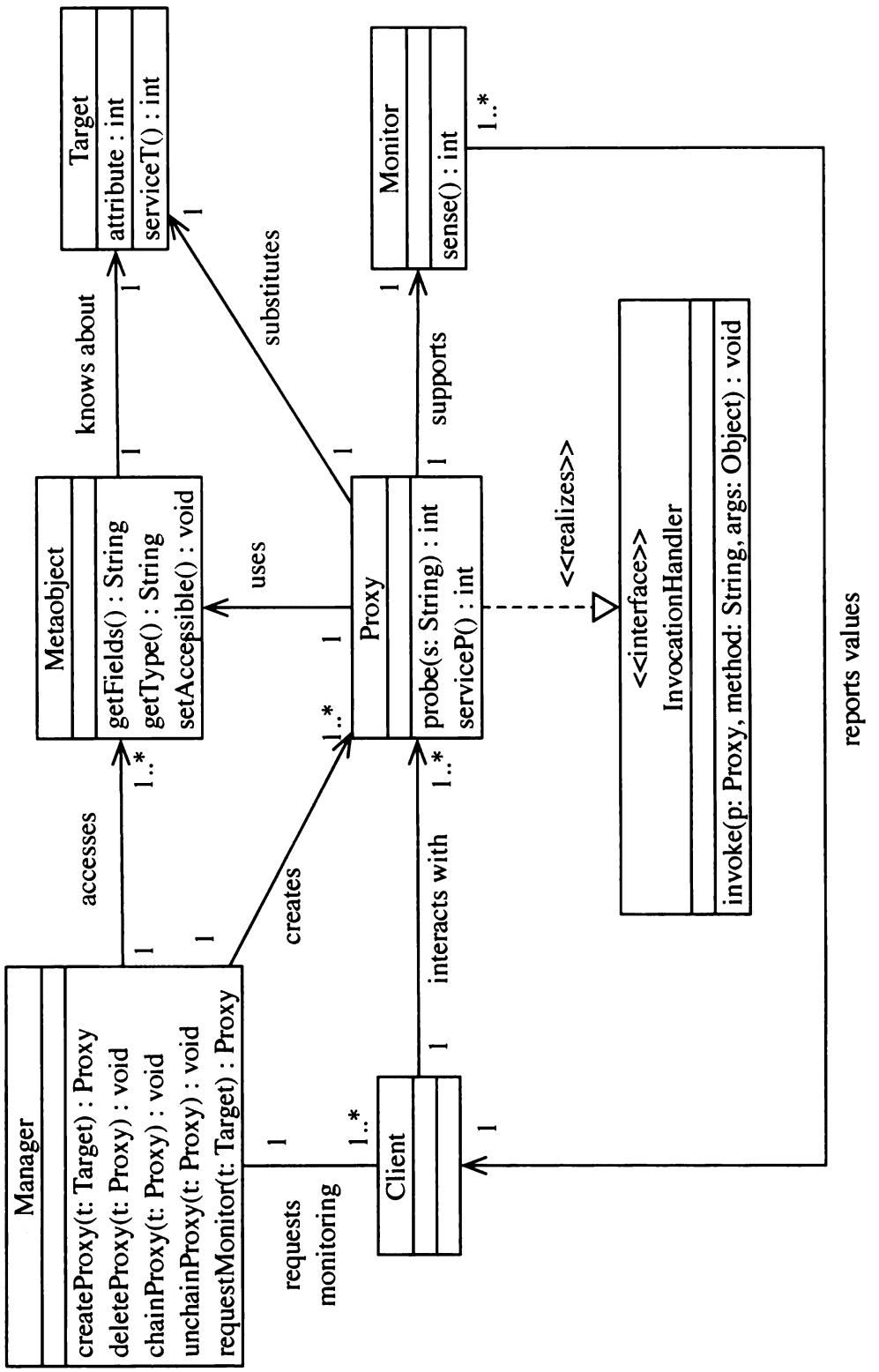

Figure 5.7: UML class diagram of the *Reflective Monitoring (50)* Pattern

encrypts the information, adds signal noise (covert monitoring), and then reports the value.

- Proxy: This class supports the Target object's functionality and interface. It is instrumented to introspect a Target's attributes but not to alter them. In addition, the Proxy object also supports probing information so the desired attributes can be retrieved transparently.

- Target: This is the object that will be monitored by a Client. It does not provide an interface to the desired attributes.

Behavior:

Figure 5.8 shows a UML sequence diagram for an example of the *Reflective Monitoring (50)* Pattern in a covert monitoring system. Specifically, Target does not provide any interface to the attribute that Client wishes to observe. A Monitoring Proxy object that supports the same interface as Target uses introspection to periodically check whether the attribute has changed or not. If the attribute has changed, then Monitoring Proxy invokes another proxy, Covert Proxy to add signal noise. This inserts random values into the channel to disguise possibly valuable information. After noise has been sent, the Monitoring Proxy returns the attribute value to the Client.

Consequences:

1. Dynamic proxies can be used to monitor components, even those that might not have been known during design and compile-time.

2. Proxies can be chained together at run time to compose new monitoring behavior without intruding upon the functional logic.

3. Many programming languages (C++, Java, Lisp, Prolog, Python, Smalltalk)

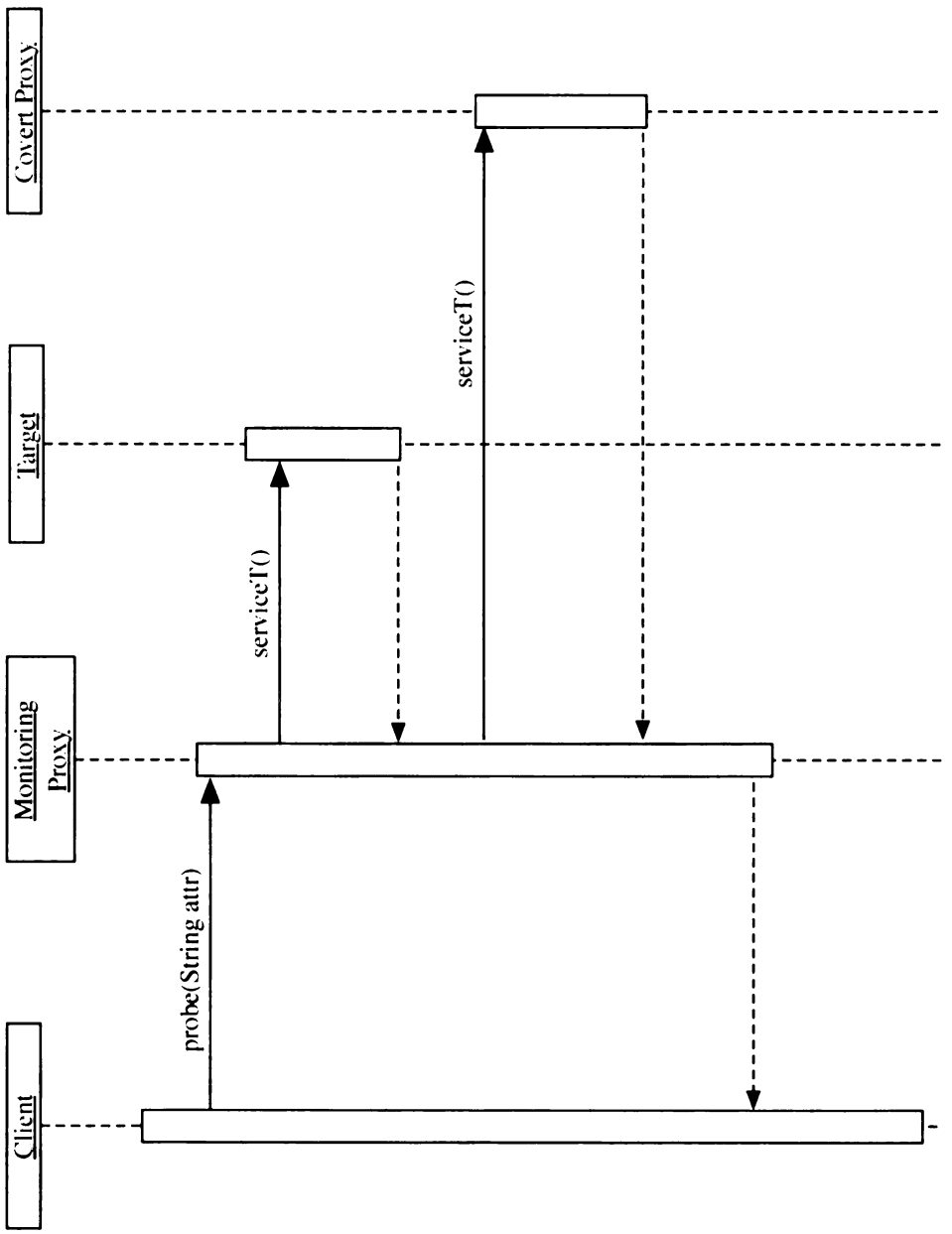

Figure 5.8: UML sequence diagram example of the *Reflective Monitoring (50)* Pattern

already support reflection mechanisms, thereby reducing the amount of effort required to implement reflective monitoring [19].

4. Dynamic proxies incur a performance penalty for the extra level of indirection.

Constraints:

- **Property 1:**

  Globally, it is always the case that if Client requests to monitor a Target, then eventually Manager returns a Proxy object.

  $\Box$ ((Client.requestMonitor(Target)) $\rightarrow$

  $\Diamond$ Manager.createProxy(Target))

  This liveness property ensures that if a Client requests a proxy, then eventually one will be returned by the Manager object.

- **Property 2:**

  Globally, it is never the case that Proxy modifies an attribute value.

  $\Box$ ((Proxy.serviceP(attr) $\rightarrow$ $\Diamond$ Target.serviceT(attr')) $\rightarrow$

  attr == attr')

  This safety property ensures that at no moment does a Proxy alter a value in Target. The *Reflective Monitoring (50)* pattern is not allowed to perform intercession upon the component it is monitoring.

Related Design Patterns:

- **Sensor-Factory (41) Design Pattern:**

  This design pattern can enable the monitoring of distributed components across a network.

- **Content-based Routing (59) Design pattern:**

  This design pattern can submit the monitoring information to a common repository where interested components can retrieve such information.

- ***Indicator* Design Pattern:**

  This pattern is responsible for interpreting the results provided by a sensor and determining when a reconfiguration is required.

Known Uses:

- Reflection Design Pattern [11].
- InsECTJ (A generic instrumentation framework for collecting dynamic information) [15].
- Adaptive Exception Monitor [19].
- Java Reflection In Action [25].

5.5.3 *Content-based Routing (59)* Pattern

Classification:

Structural - Monitoring.

Intent:

Route messages across a distributed monitoring infrastructure based on the content of the message.

Context:

The *Content-based Routing (59)* Pattern may be used when:

- Multiple clients need access to the same monitoring information.
- The predominant monitoring scheme is passive monitoring (notifications are sent when a change occurs.)

Motivation:

Adaptive systems may contain heavily monitored components. If multiple monitors are requesting the same information from a given component, then a significant overhead may be incurred at the component. To decrease this impact, a single monitor can be deployed to observe a component and then submit the gathered information to a repository that is accessible to multiple clients. This approach reduces the monitoring burden placed on any given component while ensuring that clients still gain access to the required information.

Figure 5.9 shows a use-case diagram of the *Content-based Routing (59)* Pattern. The goal of this pattern is to publish the monitoring information into a repository that can be accessed by different clients.

Use-Case:	Probe data
Actors:	Monitor
Description:	A monitor retrieves information from a given component.
Includes:	Publish data

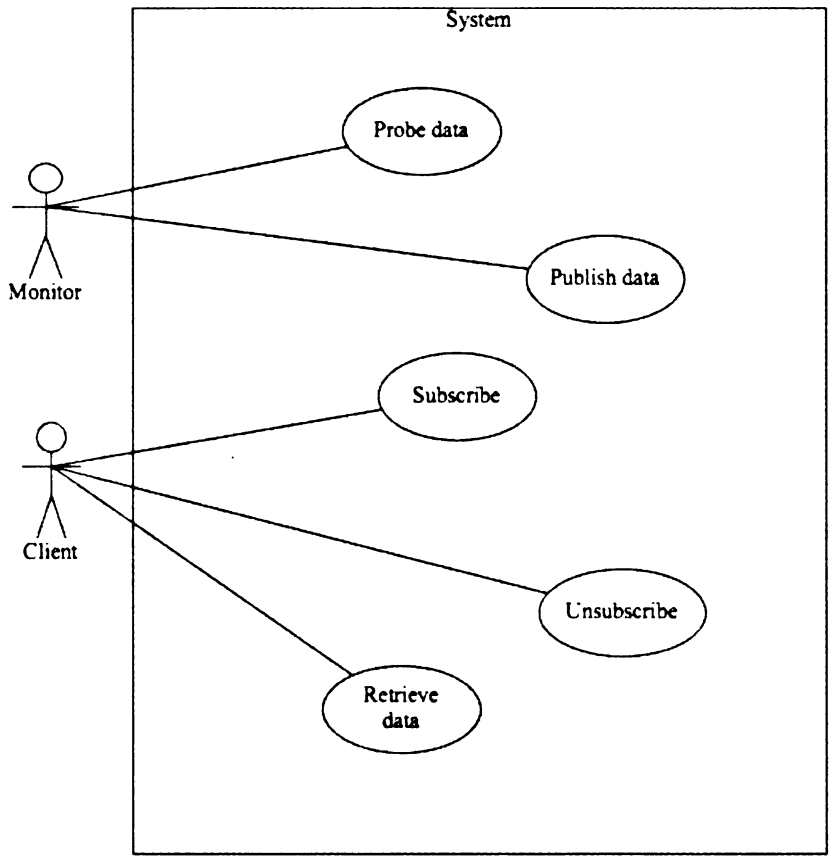

Figure 5.9: UML use-case diagram of the *Content-based Routing (59)* Pattern

Use-Case:	Publish data
Actors:	Monitor
Description:	Submits the monitoring data to a repository that is accessible to clients.
Includes:	-

Use-Case:	Subscribe
Actors:	Client
Description:	A client submits a request to be notified when new monitoring information is available about a particular component.
Includes:	-

Use-Case:	Unsubscribe
Actors:	Client
Description:	A client submits a request to disable further notifications when new monitoring information becomes available for a particular component.
Includes:	-

Use-Case:	Retrieve data
Actors:	Client
Description:	A client pulls the published data from the repository once it has been notified of its availability.
Includes:	-

Structure:

A UML class diagram for the *Content-based Routing (59)* Pattern can be found in Figure 5.10.

The *Content-based Routing (59)* design pattern adds a level of indirection between a Client and a Monitor. As a Monitor gathers information about a specific component, it sends it to a Server that is accessible to Clients. Each time new information is published by a Monitor, a Notification is generated to inform Clients that data is available. A Client retrieves the desired information from the EventService instead of continuously polling the monitored component.

Participants:

- Client: This class represents any component that needs to perform monitoring on some other component.
- Entry: This class holds a unique identifier and the address of the Monitor that published the data. EventService can use this information to authenticate the validity of an entry.

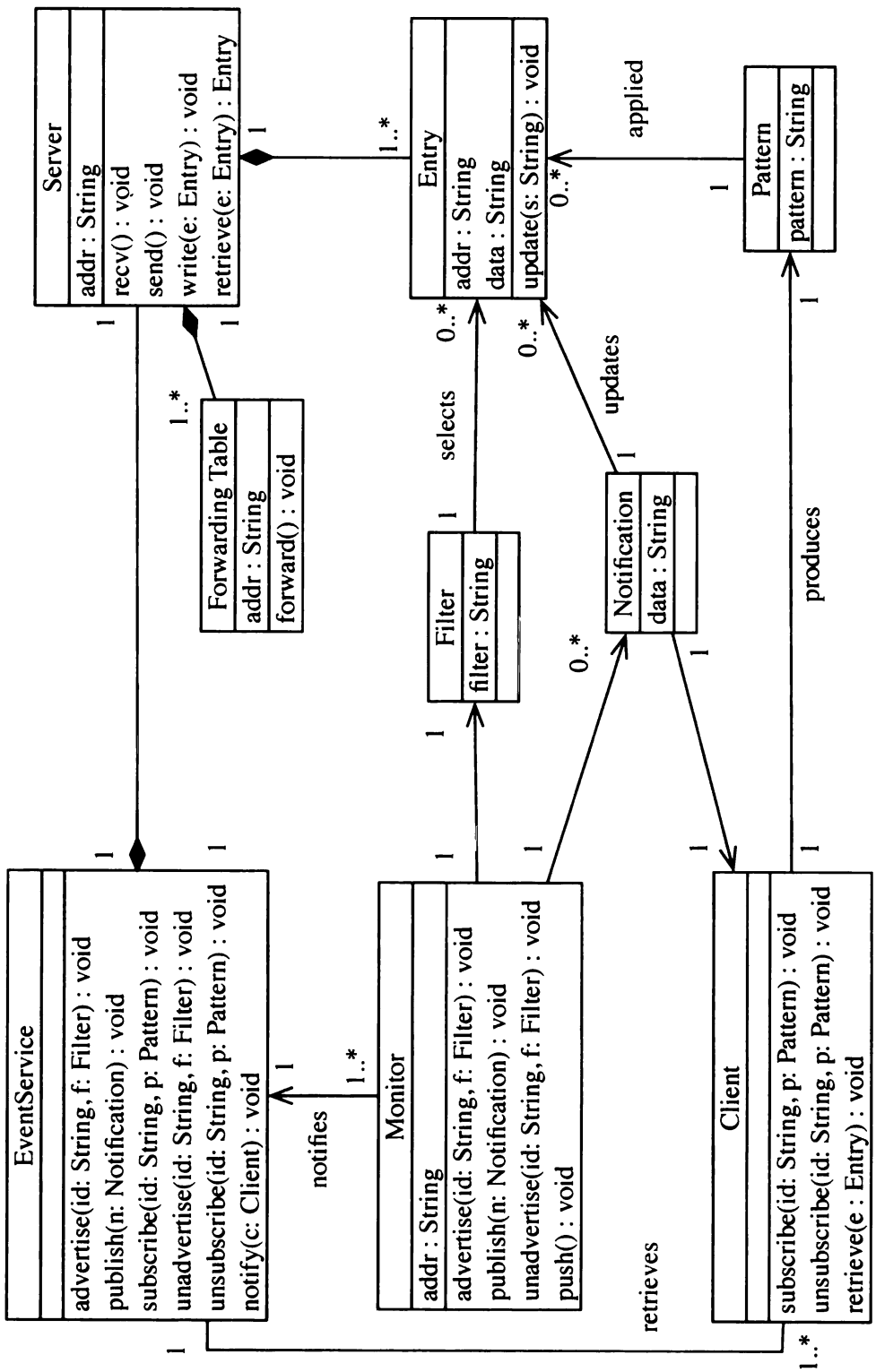

Figure 5.10: UML class diagram of the *Content-based Routing (59)* Pattern

- EventService: Mediates the communication between Monitors and Clients. Specifically, it manages the service by notifying a Client when a Monitor has published new information.

- Filter: Clients submit a Filter that indicates specific Monitors they want to observe. The Filter is applied to different patterns and those that match contain the monitoring information desired.

- Forwarding Table: Holds a listing of every Client in the system and the monitoring feeds to which they are subscribed. The EventService uses the Forwarding Table to send notifications when new monitoring information has been gathered.

- Monitor: Represents any simple or complex sensor that is currently probing a component.

- Notification: A Monitor sends a Notification to the EventService whenever it publishes any new information. Likewise, a Notification is forwarded to a Client when new information is published on the Server.

- Pattern: Represents a signature that can be used to associate data with a particular Sensor. The EventService applies Filters to Patterns to determine which data is of interest to a Client.

- Server: This class comprises a Forwarding Table and various Entry objects. Essentially, a Server holds the monitoring information for Clients to retrieve.

Behavior:

Figure 5.11 shows a UML sequence diagram for an example of the *Content-based Routing (59)* Pattern in a monitoring system. The Client object first sends a subscription notice to the Event Service. This subscription indicates which monitoring feed the Client is interested in receiving. Meanwhile, the Monitor gathers observations and

publishes them to the Event Service. To make the information available to Clients, the Event Service writes the monitoring information to an Entry on the Server. Once the information is stored, the Event Service creates a Notification and it alerts interested Clients that new information from Monitor is available. The Client then informs the Event Service to pull the relevant information from the Server. Finally, the Event Service returns the requested information to the Client.

Consequences:

1. Clients and the components being monitored are decoupled from each other. This separation facilitates the evolution of the monitoring infrastructure without affecting the functional logic.

2. The number of clients can change dynamically without affecting the component being monitored.

3. Network transparency enables various monitoring protocols to be incorporated.

4. Scalability issues may arise. Specifically, the repository where monitoring information is placed can become a bottleneck.

5. Security concerns may need to be addressed. For instance, authentication might need to be performed on the content on the message rather than on its origins.

Constraints:

- **Property 1:**

  If a Monitor publishes data of interest to a Client, then the Client should eventually be notified.

  $\Box$ ((Monitor.publish(Notification) $\wedge$ Client.subscribe(id,Monitor))

  $\rightarrow$

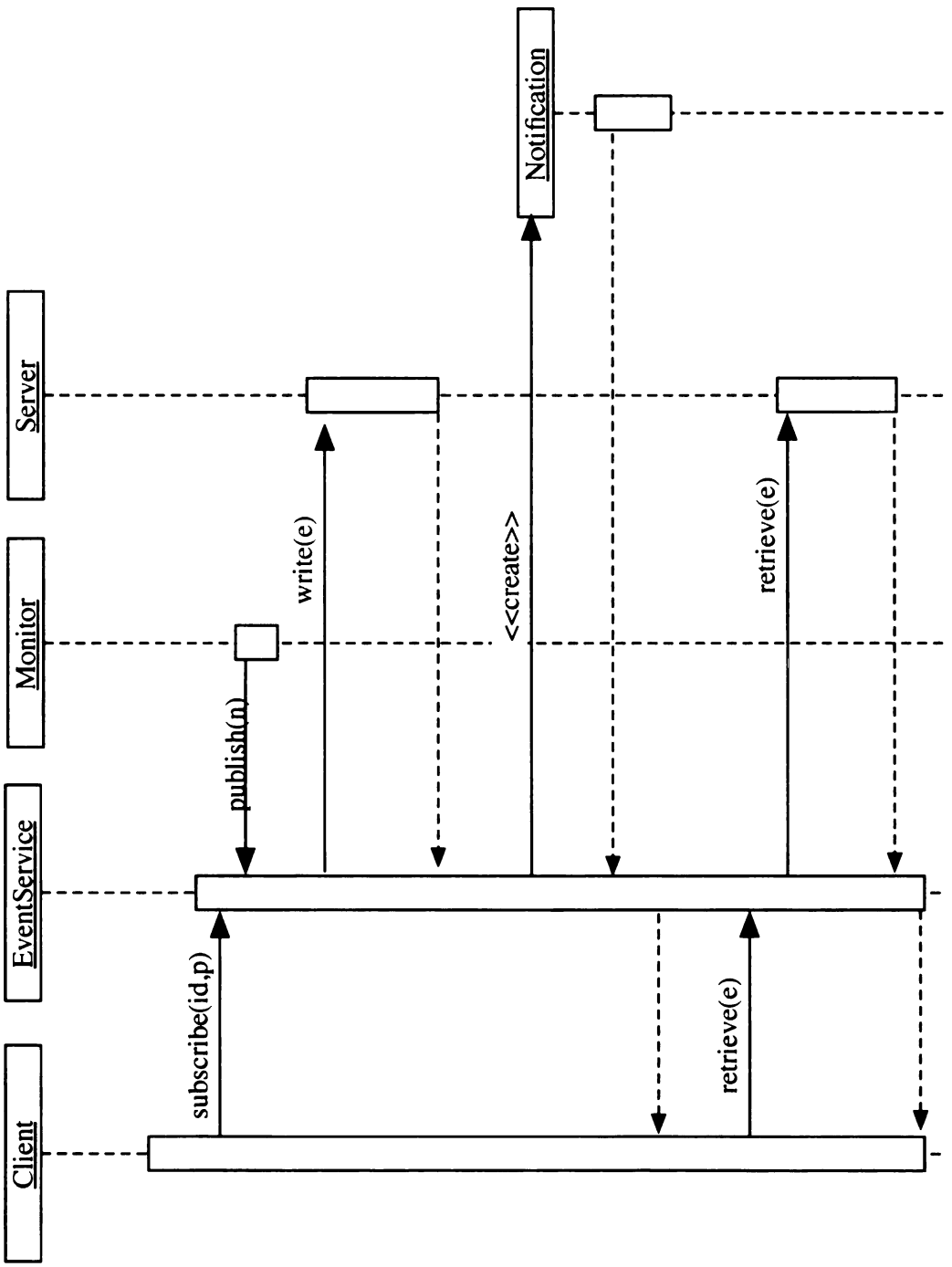

Figure 5.11: UML sequence diagram example of the *Content-based Routing (59)* Pattern

◊EventService.notify(Client))

This liveness property guarantees that if a monitor publishes data and at least one client is subscribed to that monitor, then eventually a notification will be sent. Although this is a desirable property, currently it does not specify any timing constraints.

- **Property 2:**

  Globally, it is always the case that if a Client receives a Notification, then the Client will eventually retrieve the information.

  □ ((EventService.notify(Client)) →

  ◊ (Client.retrieve(Entry))

  This liveness property guarantees that if a client receives a notification from the event service, then it will eventually retrieve the updated information. This property ensures that a client retrieves new information whenever it is updated by a sensor.

Related Design Patterns:

- ***Observer* Design Pattern:**

  The *Content-based Routing (59)* pattern can be considered an extension of the Observer design pattern [26]. Specifically, the Observer design pattern provides a one-to-many notification mechanism. The *Content-based Routing (59)* pattern, on the other hand, provides a many-to-many notification mechanism.

- **Sensor-Factory (41) Design Pattern:**

  This design pattern can enable the monitoring of distributed components across a network.

- **Adaptation Detector (88) Design Pattern:**

  This pattern is responsible for interpreting the results provided by a sensor and determining when a reconfiguration is required.

Known Uses:

- Rainbow Adaptation Framework [30].
- Siena Routing [43].
- JAMM Monitoring System [77].
- Rebeca [83].

5.5.4 *Case-based Reasoning (68)* Pattern

Classification:

Structural - Decision-Making.

Intent:

Apply rule-based decision-making to determine how to reconfigure the system.

Context:

The *Case-based Reasoning (68)* Pattern may be used when:

- The system must determine which adaptation to perform automatically.
- The criteria for reconfiguration is not complex and can be expressed through *if-then-else* statements.

Motivation:

Dealing with decision-making internally through constructs that trap errors at the implementation level is undesirable for two main reasons [27]. First, the overall context of what triggered the adaptation event is usually lost at such fine-grained levels. Second, evolving or correcting such a decision-making process is difficult to accomplish because the cause-effect relationships for triggers and events are tightly coupled to the functional logic. The *Case-based Reasoning (68)* design pattern separates the decision-making logic from the functional logic of the application. Specifically, it centralizes all the conditions and responses for reconfiguring a system such that they do not crosscut the functional logic. This separation of concerns results in an external and flexible decision-making architecture that facilitates change.

Figure 5.12 shows a use-case diagram of the *Case-based Reasoning (68)* Pattern. The main goal of the *Case-based Reasoning (68)* design pattern is to determine which reconfiguration to perform.

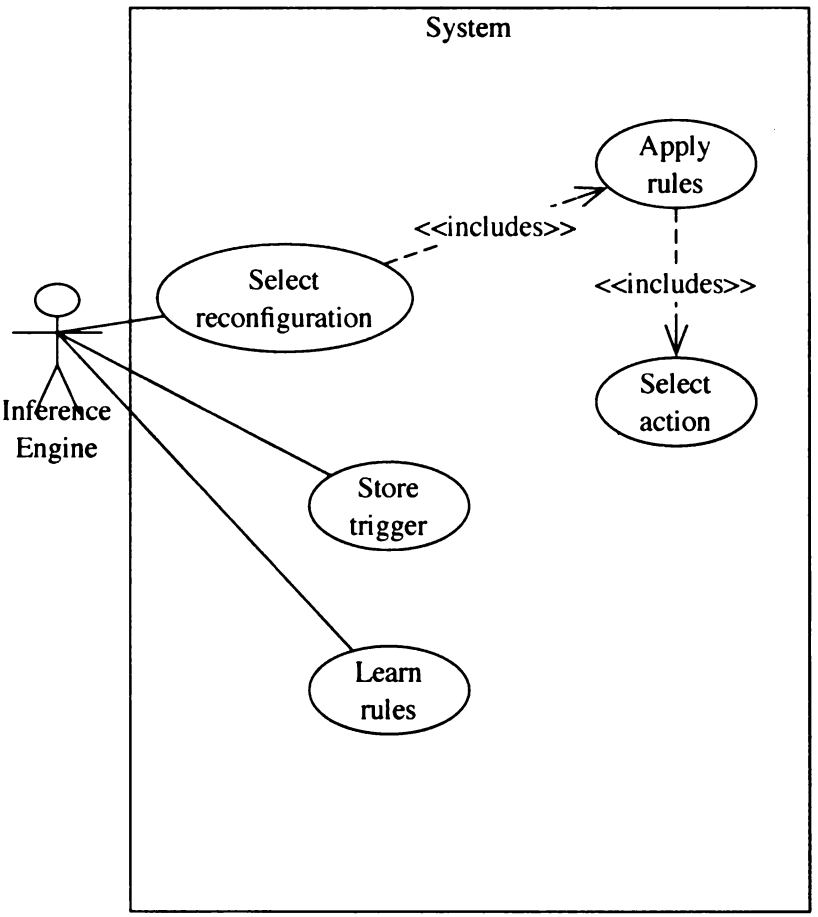

Figure 5.12: UML use-case diagram of the *Case-based Reasoning (68)* Pattern

Use-Case:	Select Reconfiguration
Actors:	Inference Engine
Description:	Determine which available reconfiguration will yield the desired behavior in the system.
Includes:	Apply rules.

Use-Case:	Apply rules
Actors:	-
Description:	Determine which cause and effect relationship holds true within the set of rules.
Includes:	Select action.

Use-Case:	Select Action
Actors:	-
Description:	Chooses a reconfiguration plan based on what triggered the adaptation and the rule that describes the cause and effect relationship.
Includes:	-

Use-Case:	Store trigger
Actors:	Inference Engine
Description:	Stores the cause of the adaptation request and the reconfiguration plan that was selected. The Inference Engine can use this information to learn new rules in the future.
Includes:	-

Use-Case:	Learn rules
Actors:	Inference Engine
Description:	Reviews previous adaptation causes and reconfiguration responses to discover new rules to apply in the future.
Includes:	-

Structure:

A UML class diagram for the *Case-based Reasoning (68)* Pattern can be found in Figure 5.13.

The *Case-based Reasoning (68)* design pattern separates the decision-making logic from the monitoring, reconfiguration, and functional logic. During execution, some monitoring event (not shown in this design pattern, refer to *Adaptation Detector (88)* for more information) will generate a Trigger and forward it to the Inference Engine. The Inference Engine comprises a set of Fixed Rules, a Trigger Repository, and a Learner algorithm. A Trigger is applied to a Fixed Rules set, a Decision is generated, and the result is stored in a Log so the Inference Engine can learn new rules. Since the *Case-based Reasoning (68)* design pattern centralizes all conditions and rules

regarding a reconfiguration plan, a Rule can be readily added, modified, or removed.

Participants:

- Decision: This class represents a reconfiguration plan that will yield the desired behavior in the system.
- Fixed Rules: This class contains a collection of Rules that guide the Inference Engine in producing a Decision. The individual Rules stored within the Fixed Rules can be changed at run time.
- Inference Engine: This class is responsible for applying a set of Rules to either a single Trigger or a history of Triggers and producing an action in the form of a Decision.
- Learner: Applies on-line and statistical-based algorithms to infer new Rules in the system. This is an optional feature of the *Case-based Reasoning (68)* design pattern.
- Log: This class is responsible for recording which reconfiguration plans have been selected during execution. Each entry is of the form Trigger-Rule-Decision.
- Rule: Represents a relationship between a Trigger and a Decision. A Rule evaluates to *true* if an incoming Trigger matches the Trigger contained in the Rule.
- Trigger: This class contains relevant information about what caused the adaptation request. A Trigger should at least provide information about the error source, the timestamp at which the error was observed, the type of error observed and whether it has occurred before or not. Additional information may be included as required. A Trigger is invoked by the *Adaptation Detector (88)* design pattern.

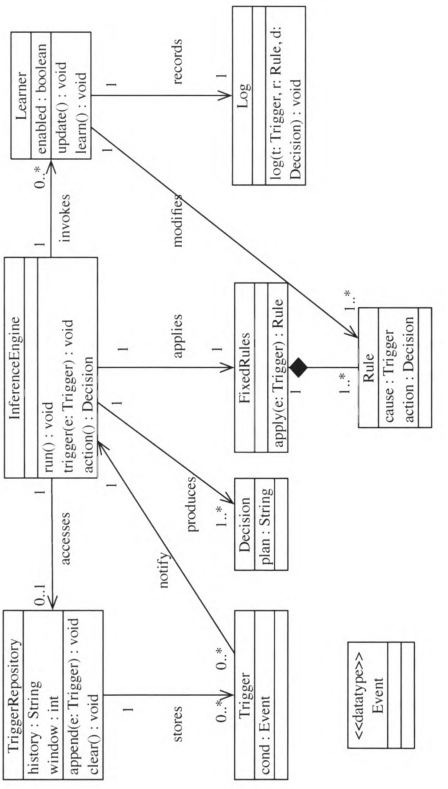

Figure 5.13: UML class diagram of the *Case-based Reasoning (68)* Pattern

- Trigger Repository: Contains a history of Triggers. This history can be used by the Learner class to identify trends that may warrant further reconfigurations.

Behavior:

Figure 5.14 shows a UML sequence diagram for an example of the *Case-based Reasoning (68)* Pattern in a simple adaptive system. A Trigger alerts the Inference Engine that an adaptation request has been submitted (see *Adaptation Detector (88)* Pattern for more information.) The Inference Engine uses the information provided by the Trigger to decide which reconfiguration plan will yield the desired results in the system. The Inference Engine applies a set of rules stored in the Fixed Rules object until a matching rule is found. Inference Engine then creates a Decision that includes the selected reconfiguration plan. The Inference Engine then logs the resulting action in the Log. In addition, the Inference Engine also stores the Trigger in the Trigger Repository for further analysis.

Consequences:

1. The decision-making logic is separate from the monitoring logic, the reconfiguration logic, and the functional logic, thereby facilitating the evolution of rules and actions at run time.

2. New rules can be learned dynamically to accommodate new reconfiguration scenarios.

3. If many reconfiguration scenarios are possible, then scalability issues such as overlapping rules may arise.

4. Rule-based decision-making can only select reconfigurations that are known prior to the current execution point.

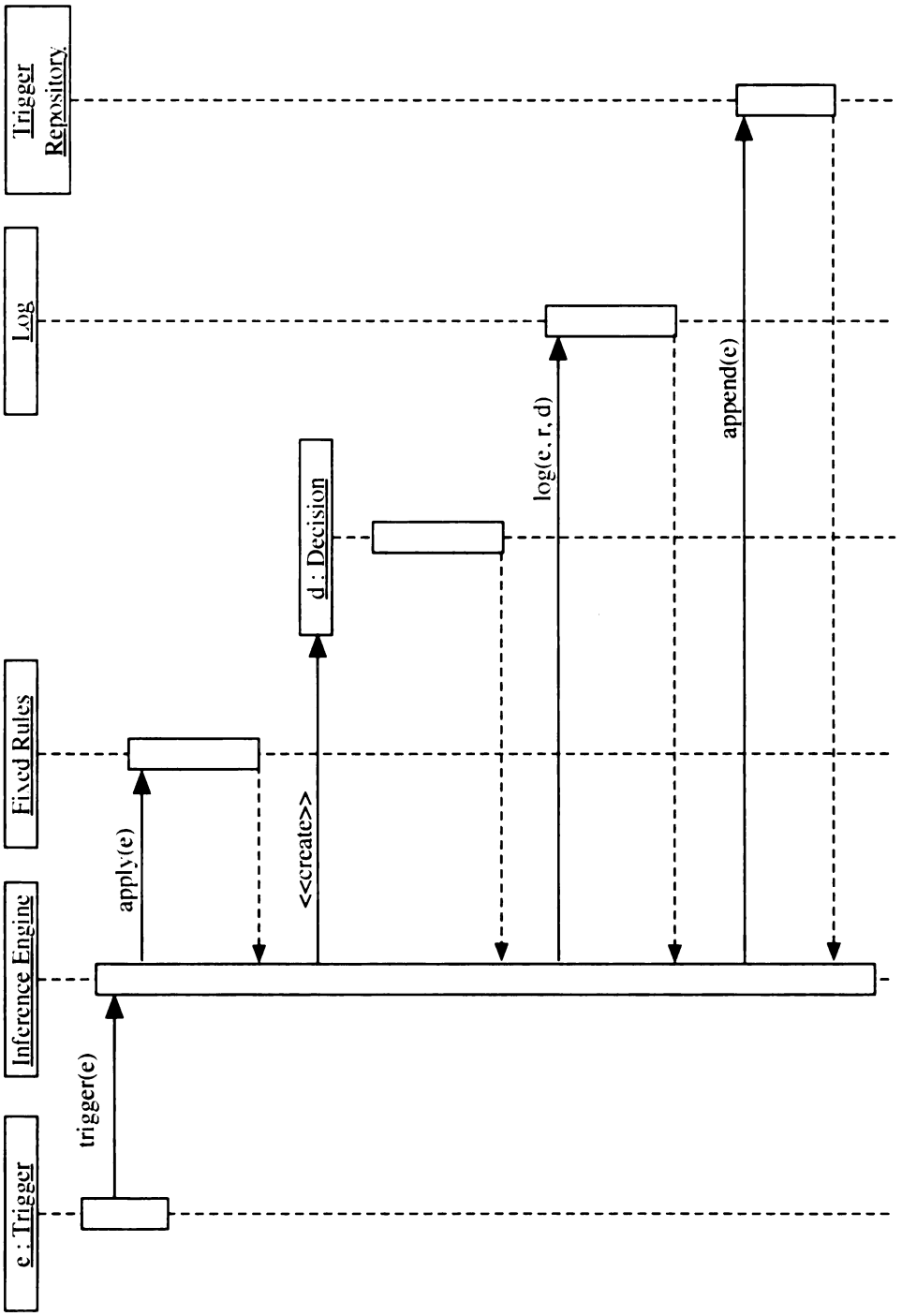

Figure 5.14: UML sequence diagram example of the *Case-based Reasoning (68)* Pattern

Constraints:

- **Property 1:**

  Globally, it is always the case that if a Trigger is received, then Inference Engine eventually produces a Decision.

  $\Box$ (InferenceEngine.trigger(Trigger) $\rightarrow$

  $\Diamond$ InferenceEngine.action())

  This liveness property ensures that a decision is eventually produced by the rule-based decision making process. One way to ensure this property holds is by creating a default conditional-action pair. At the very least, this default conditional could notify the system administrator that an event occurred and no matching reconfiguration plan was found.

- **Property 2:**

  Globally, it is always the case that if Inference Engine produces a Decision, then Log will eventually record the events.

  $\Box$ (InferenceEngine.action() $\rightarrow$

  $\Diamond$ Log.log(Trigger,Rule,Decision))

  This liveness property ensures that every reconfiguration selected by the rule-based decision-making process is eventually logged. This property is desirable because it would enable developers to keep track of how the system is executing at run time in response to changing requirements and environmental conditions.

Related Design Patterns:

- **Adaptation Detector (88) Design Pattern:**

This design pattern can be used to interpret monitoring information and determine when a reconfiguration is required. The notification can be used by the Inference Engine to select a reconfiguration plan.

- **Component Insertion (115) Design Pattern:**

  This design pattern can be used to safely insert a component at run time according to the reconfiguration plan selected by the rule-based decision-making process.

- **Component Removal (125) Design Pattern:**

  This design pattern can be used to safely remove a component at run time according to the reconfiguration plan selected by the rule-based decision-making process.

- **Server Reconfiguration (135) Design Pattern:**

  This design pattern can be used to safely reconfigure a server architecture at run time. The specific reconfiguration plan to enact those changes can be selected by the rule-based decision-making process.

- **Decentralized Reconfiguration (145) Design Pattern:**

  This design pattern can be used to safely reconfigure a decentralized architecture at run time. The specific reconfiguration plan to be enacted by a particular component can be selected by the rule-based decision-making process.

Known Uses:

- Decentralized self-adaptive component-based system [6].
- Rainbow Adaptation Framework [27].
- Architectural Approach to Autonomic Computing [41].

- Earth Management Application (Ontology-based mobile agents) [71].
- Kinesthetics eXtreme (KX) Framework [79].

5.5.5 *Divide and Conquer (78)* Pattern

Classification:

Structural - Decision-Making.

Intent:

Systematically decompose a complex reconfiguration plan into simpler reconfiguration plans.

Context:

The *Divide and Conquer (78)* Pattern may be used when:

- multiple reconfiguration plans need to be applied to achieve the desired behavior.
- a reconfiguration plan involves dependencies between distributed components.

Motivation:

Adaptive systems often comprise distributed components. Each component implements a part of the desired behavior of the system. Some of these components may include *fragments*, or parts of the component that are associated with different processes in the distributed system [8]. While adding or removing components, not all fragments of a component are added or removed simultaneously. This asynchronous behavior may lead to situations in which some processes have added/removed the component fragments while some have yet to do so [9]. To avoid problems, dependency relationships among the fragments should be handled correctly while adding and removing fragments. In other situations, multiple reconfigurations need to be performed in succession to achieve the overall desired behavior. To avoid these problems, dependency relationships between the different reconfiguration plans must be handled carefully. The *Divide and Conquer (78)* design pattern first determines dependency relations between different component fragments and then creates an ordering that will safely reconfigure the system by preserving the dependency relationships.

Figure 5.15 shows a use-case diagram of the *Divide and Conquer (78)* Pattern. The main goal of this design pattern is to decompose a complex reconfiguration plan into simpler reconfiguration plans that can be applied by the adaptive system.

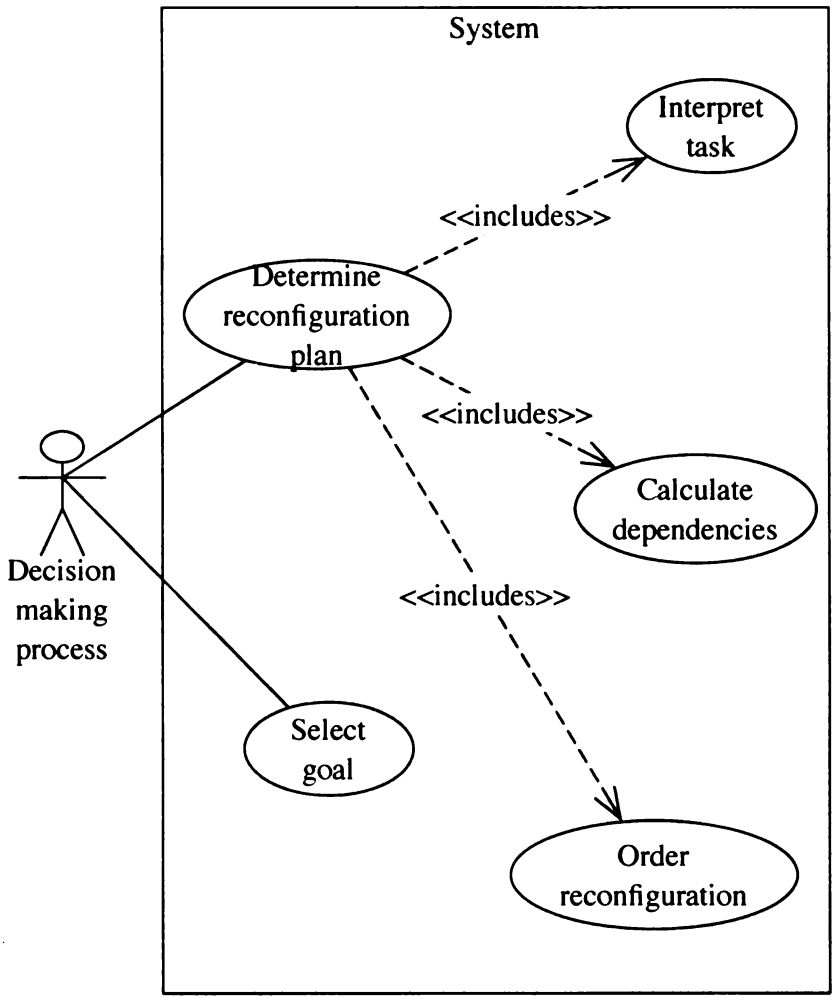

Figure 5.15: UML use-case diagram of the *Divide and Conquer (78)* Pattern

Use-Case:	Determine reconfiguration plan
Actors:	Decision-making process
Description:	Find a sequence of reconfigurations that will yield the desired behavior. The reconfiguration plan must specify the sequencing of each step.
Includes:	Interpret task, Calculate dependencies, Order reconfiguration

Use-Case:	Interpret task
Actors:	-
Description:	Analyze the reconfiguration plan goal, its constraints, and requirements.
Includes:	-

Use-Case:	Calculate dependencies
Actors:	-
Description:	Determine if there are any specific dependencies between reconfiguration plans.
Includes:	-

Use-Case:	Order reconfiguration
Actors:	-
Description:	Creates a sequence of steps that will safely reconfigure the system in order to provide the desired behavior.
Includes:	-

Use-Case:	Select goal
Actors:	Decision-making process
Description:	Choose a reconfiguration plan to analyze.
Includes:	-

Structure:

A UML class diagram for the *Divide and Conquer (78)* Pattern can be found in Figure 5.16. The *Divide and Conquer (78)* design pattern provides an approach for systematically decomposing tasks. This design pattern analyzes combinations of reconfigurations to determine the specific sequence of reconfiguration steps that will safely yield the desired behavior in the application. The overall approach is split into two main tasks. First, various sets of existing reconfiguration plans are searched and analyzed to determine if they can be combined to reconfigure the system as needed. Second, if a combination of existing reconfiguration plans satisfy the adaptation requirements, then any dependencies between the reconfiguration plans

are determined. The reconfiguration plans are then combined in a specific sequence of reconfiguration steps to yield the overall system adaptation.

Participants:

- Dependency Analysis: Determines which Tasks are dependent upon other Tasks. This information can be used by the Planner to generate a sequence of Tasks that will accomplish the main Goal.
- Goal: This represents the adaptation requirements that must be satisfied by applying a set of reconfigurations.
- Inference Engine: The Inference Engine is responsible for resolving how a given Goal can be decomposed into a set of Tasks that satisfy the adaptation requirements. This class makes use of the Knowledge Base (KnowledgeBase) and Rule Base (RuleBase) to perform either Informed or Uninformed forms of resolution and searching.
- Informed: A set of informed heuristics that can guide the resolution process carried out by the Inference Engine.
- KnowledgeBase: The Knowledge Base represents axioms known to the system. These axioms are used by the Rule Base to perform resolution tasks.
- Lexer: This optional class is responsible for converting data into a sequence of tokens. Specifically, both adaptation requirements and reconfiguration plans are stored in the system in various formats such as text, models, and so forth. The Lexer converts otherwise meaningless data into tokens recognizable by the system.
- Parser: This optional class analyzes the sequence of tokens produced by the Lexer. It is responsible for transforming Goals into data that can be directly

Figure 5.16: UML class diagram of the *Divide and Conquer (78)* Pattern

used by Solver and Inference Engine.

- Planner: Takes a set of dependencies between Tasks and applies an ordering that will safely solve the Goal without violating dependencies.
- RuleBase: Represents conditionals and actions known by the system that can be used to evaluate and select Tasks to solve a Goal. Note, this class is not related to the set of rules in *Case-based Reasoning (68)*.
- Solver: This class is responsible for organizing the overall task decomposition process. It invokes the Lexer and Parser classes to interpret Goals. It also invokes the Inference Engine and fine-tunes the search procedure until a sequence of Tasks that solve the Goal is found.
- Task: This class is used to represent a reconfiguration plan. Ideally, by applying a series of Tasks the Goal will be satisfied.
- Uninformed: A set of uninformed approaches that exhaustively search the solution space. These approaches do not exploit any particular knowledge that may optimize the search.

Behavior:

Figure 5.17 shows a UML sequence diagram for an example of the *Divide and Conquer (78)* Pattern. The Solver coordinates the process of task decomposition by first invoking the Inference Engine. The Inference Engine proceeds to retrieve known facts about Goals and Tasks from the Knowledge Base (KB). In addition, the Inference Engine also retrieves a set of rules from the Rule Base (RB). The Inference Engine then proceeds to resolve the known facts with the available rules. Once a set of Tasks have been identified, the Inference Engine invokes the Dependency Calculator to determine an ordering. The Dependency Calculator uses the Planner object to create a sequence of steps in which the Tasks must be solved.

Consequences:

1. More complex adaptation requirements can be satisfied by reusing and composing multiple reconfiguration plans.

2. Reconfiguration plans that are not dependent upon each other can be parallelized to enhance performance.

3. Dependencies among the reconfiguration plans will not be violated by the generated sequence.

4. It may not be possible to satisfy *all* adaptation requirements by decomposing them into sequences of reconfiguration plans.

5. There is an increased overhead in determining how to reconfigure the system.

Constraints:

- **Property 1:**

  Globally, it is always the case that a proposed set of Tasks satisfies the Goal.

  $\Box$ ((DependencyCalculator.constraintChecker(Tasks)='True') $\rightarrow$ $\Diamond$(Goal.completed='True'))

  A set of reconfiguration plans must correctly reconfigure the system without violating any dependencies. If such a plan is found, then eventually the goal should be satisfied.

Related Design Patterns:

- ***Architecture-Based (97)* Design Pattern:**

  This design pattern can be used to represent a system and its reconfiguration plans as architectural models. Models that satisfy the adaptation requirements indicate how the system should be reconfigured.

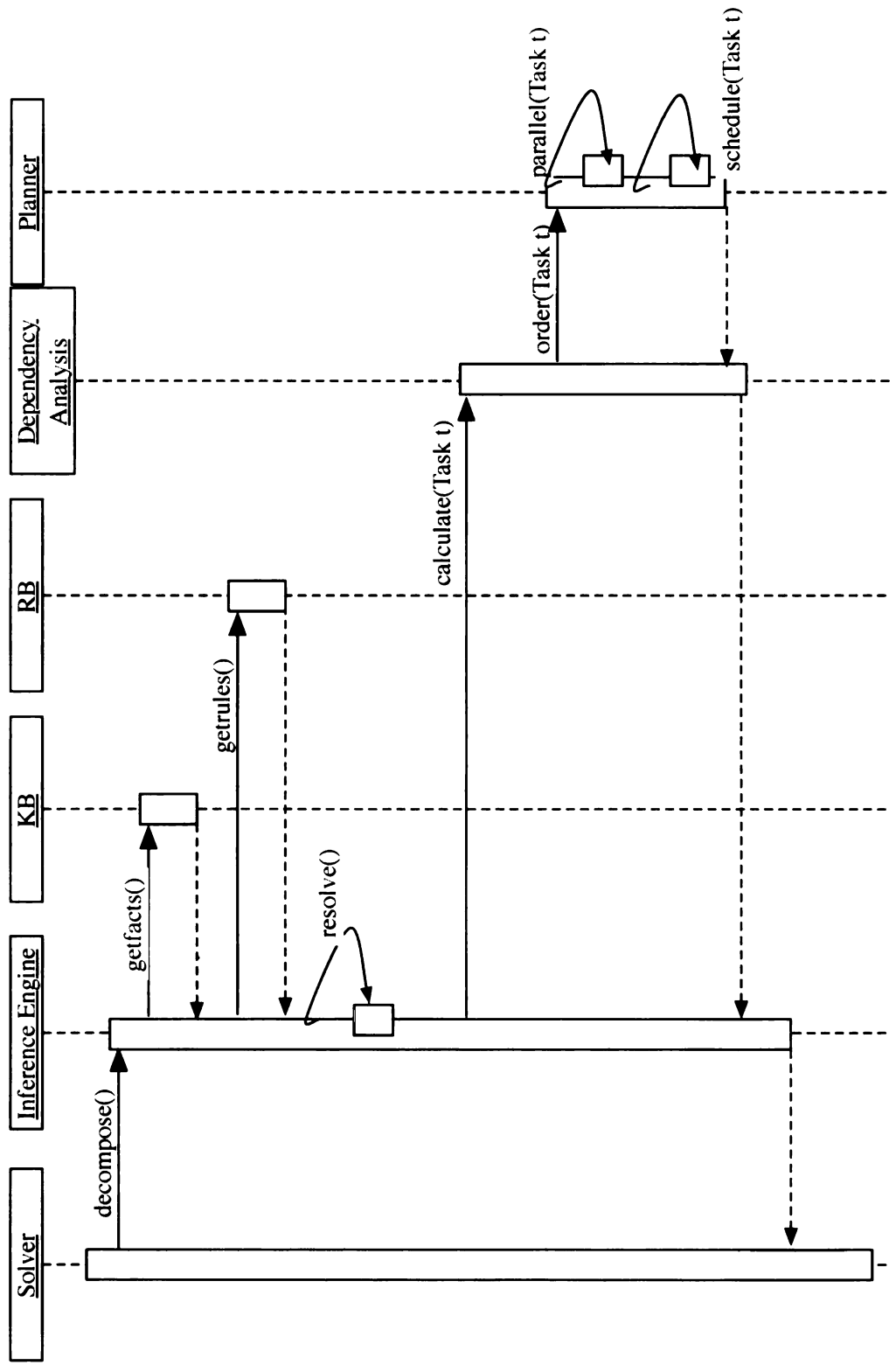

Figure 5.17: UML sequence diagram example of the *Divide and Conquer (78)* Pattern

- ***TradeOff-Based (106)* Design Pattern:**

  This design pattern can be used to select a reconfiguration plan that best balances multiple objectives. Complex reconfiguration plans can be decomposed by the *Divide and Conquer (78)* pattern.

- ***Component Insertion (115)* Design Pattern:**

  This design pattern can be used to safely insert a component at run time.

- ***Component Removal (125)* Design Pattern:**

  This design pattern can be used to safely remove a component at run time.

- ***Server Reconfiguration (135)* Design Pattern:**

  This design pattern can be used to safely reconfigure a server architecture at run time. If the reconfiguration plan is complex, it can be decomposed by the *Divide and Conquer (78)* pattern.

- ***Decentralized Reconfiguration (145)* Design Pattern:**

  This design pattern can be used to safely reconfigure a decentralized architecture at run time. Any component can decompose a complex reconfiguration plan through the use of the *Divide and Conquer (78)* pattern.

Known Uses:

- Task Decomposition [1].
- Rainbow Adaptation Framework [27, 29].
- Care-O-Bot II (uses metric-FF) [40, 45].
- An Architectural Approach to Autonomic Computing [41].
- Simple Hierarchical Ordered Planner (SHOP2) [65].

- Proactive Control, Monitoring and Maintenance (PCMM) Modules for Autonomous Systems [78].
- Task Control Architecture (TAC) for Mobile Robots [76].

5.5.6 *Adaptation Detector (88)* Pattern

Classification:

Structural - Decision-Making.

Intent:

Interpret monitoring data and determine when an adaptation is required.

Context:

The *Adaptation Detector (88)* Pattern may be used when:

- An adaptive system must automatically determine when an adaptation is required.

Motivation:

Adaptive systems need to determine when an adaptation is required. Usually, adaptive systems employ monitors to observe both the system's internal behavior as well as its environment. Simple sensors typically provide raw data feeds that must be interpreted by the system to be of any use. Once interpreted, this information can be used to identify situations where observed behavior deviates from expected behavior. If the components are distributed across a network, accomplishing this task becomes increasingly difficult because the system must track where the problem originated [30]. The *Adaptation Detector (88)* design pattern retrieves and analyzes relevant data from one or more sensors and generates a health indicator value. The health indicator value is determined by comparing the observed behavior versus the expected behavior. As soon as a deviation is identified, the decision-making process is notified with all the relevant information available so it may determine which reconfiguration plan to apply.

Figure 5.18 shows a use-case diagram of the *Adaptation Detector (88)* Pattern. The main goal of this design pattern is to determine when an adaptation is required.

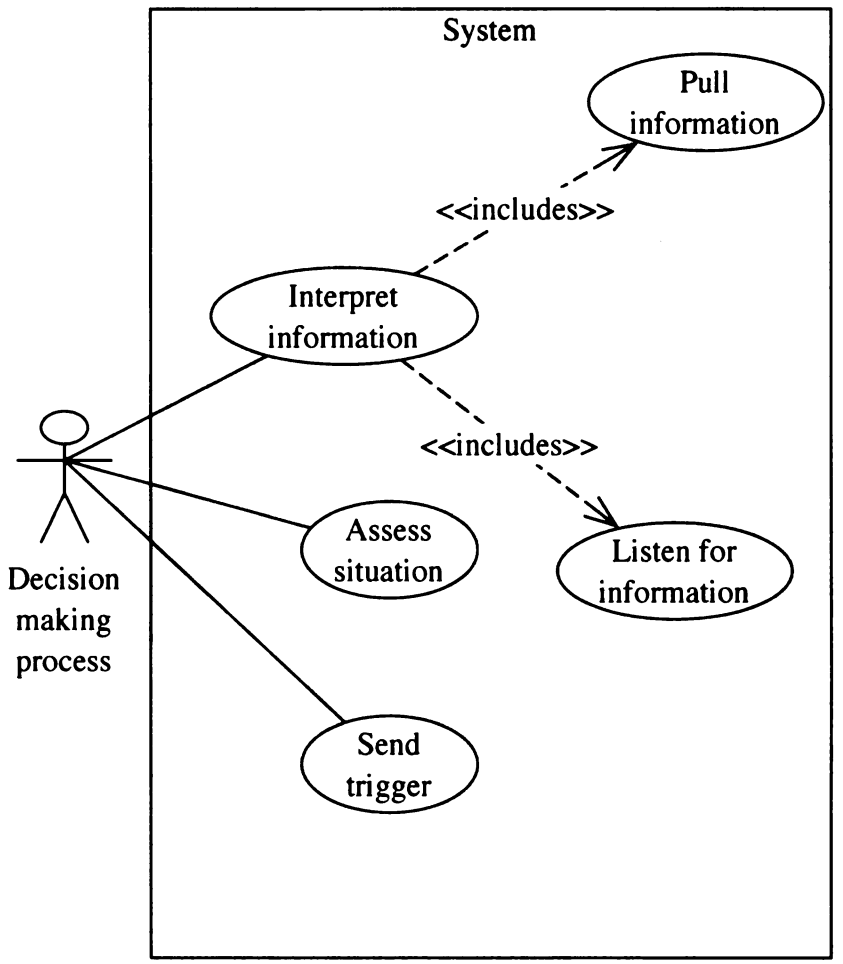

Figure 5.18: UML use-case diagram of the *Adaptation Detector (88)* Pattern

Use-Case:	Interpret information
Actors:	Decision-making process
Description:	Assign meaning to a monitoring feed.
Includes:	Pull information, Listen for information

Use-Case:	Pull information
Actors:	-
Description:	Actively retrieve the monitoring information from a passive sensor.
Includes:	-

Use-Case:	Listen for information
Actors:	-
Description:	Passively wait for a notification that an event has occurred.
Includes:	-

Use-Case:	Assess situation
Actors:	Decision-making process
Description:	Evaluates the monitored values against certain thresholds. If the values exceed those thresholds, then an adaptation is required.
Includes:	-

Use-Case:	Send trigger
Actors:	Decision-making process
Description:	Notify the decision-making process that an adaptation is required. The trigger should include information about what triggered the adaptation request.
Includes:	-

Structure:

A UML class diagram for the *Adaptation Detector (88)* Pattern can be found in Figure 5.19.

Adaptive systems must determine when observed behavior deviates from expected behavior. These situations typically require some form of adaptation. The *Adaptation Detector (88)* Design Pattern acts as a gateway between the monitoring and decision-making processes. Specifically, the *Adaptation Detector (88)* Design Pattern is responsible for interpreting monitoring values and deciding when a reconfiguration is needed. To accomplish this objective, a Health Indicator is associated with a specific Sensor. The Health Indicator can be used with either passive or active Sensors. Once the Analyzer determines that a Threshold has been exceeded, the Health Indicator generates a Trigger to issue an adaptation request.

Participants:

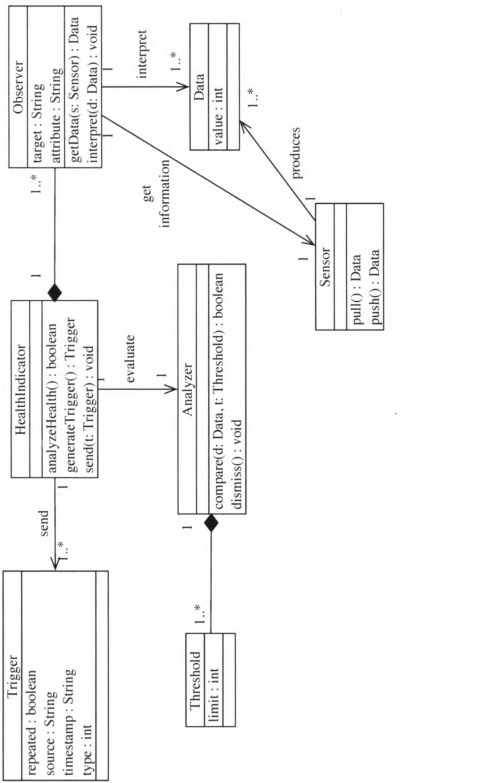

Figure 5.19: UML class diagram of the *Adaptation Detector (88)* Pattern

- **Analyzer**: This class is responsible for deciding when observed behavior deviates from expected behavior. Specifically, the **Analyzer** compares the interpreted values from a **Sensor** against discrete values stored in a **Threshold**. An adaptation is warranted when observed values exceed a given threshold.

- **Data**: Represents the values reported by a **Sensor**. The basic data types supported include integers, doubles, and booleans. Other data types may be used as well in conjunction with a complex **Sensor**.

- **Health Indicator**: Coordinates the process of interpreting a **Sensor**'s values and determining whether an adaptation is required.

- **Observer**: Interacts with a specific **Sensor**. The **Observer** is responsible for interpreting **Data** produced by a **Sensor**. This process is required before the **Analyzer** can evaluate the monitoring information against certain **Thresholds**.

- **Sensor**: This class represents any type of simple or complex **Sensor** that is capable of reporting integers, booleans, or floats. Complex **Sensors** that produce other types of values can be used with the *Adaptation Detector (88)* Design Pattern, but may need further customization according to the computational resources provided.

- **Threshold**: Stores discrete values that represent the boundaries between normal and abnormal behavior.

- **Trigger**: This class contains relevant information about what caused the adaptation request. A **Trigger** should at least provide information about the error source, the timestamp at which the error was observed, the type of error observed and whether it has occurred before or not. Additional information may be included as required.

Behavior:

Figure 5.20 shows a UML sequence diagram for an example of the *Adaptation Detector (88)* Pattern in an adaptive system. The Health Indicator first invokes the Observer to gather and interpret data from the Sensor. A Sensor periodically produces Data, which may contain information about the system's internal behavior or the environment surrounding the application. The Observer *pulls* Data from the Sensor and interprets it. Once the information has been interpreted, the Health Indicator invokes the Analyzer to compare those values against some Thresholds. In this particular scenario, the Threshold is exceeded and the Health Indicator generates and sends a Trigger to notify the decision-making process that an adaptation is required.

Consequences:

1. This design pattern separates the monitoring and decision-making processes from each other. Changes in one process should not affect the other.

2. The specific threshold values used to represent the boundaries between normal and abnormal behavior are kept within the *Adaptation Detector (88)* design pattern and not dispersed throughout the source code. This facilitates evolving the system at run time.

3. If many monitoring probes are deployed, then scalability issues may arise. Specifically, this design pattern adds a level of indirection between monitoring and decision-making processes which could add a considerable time delay.

Constraints:

- **Property 1:**

  Globally, it is always the case that if Observer obtains monitoring information, then Analyzer will eventually compare the data against a specific Threshold.

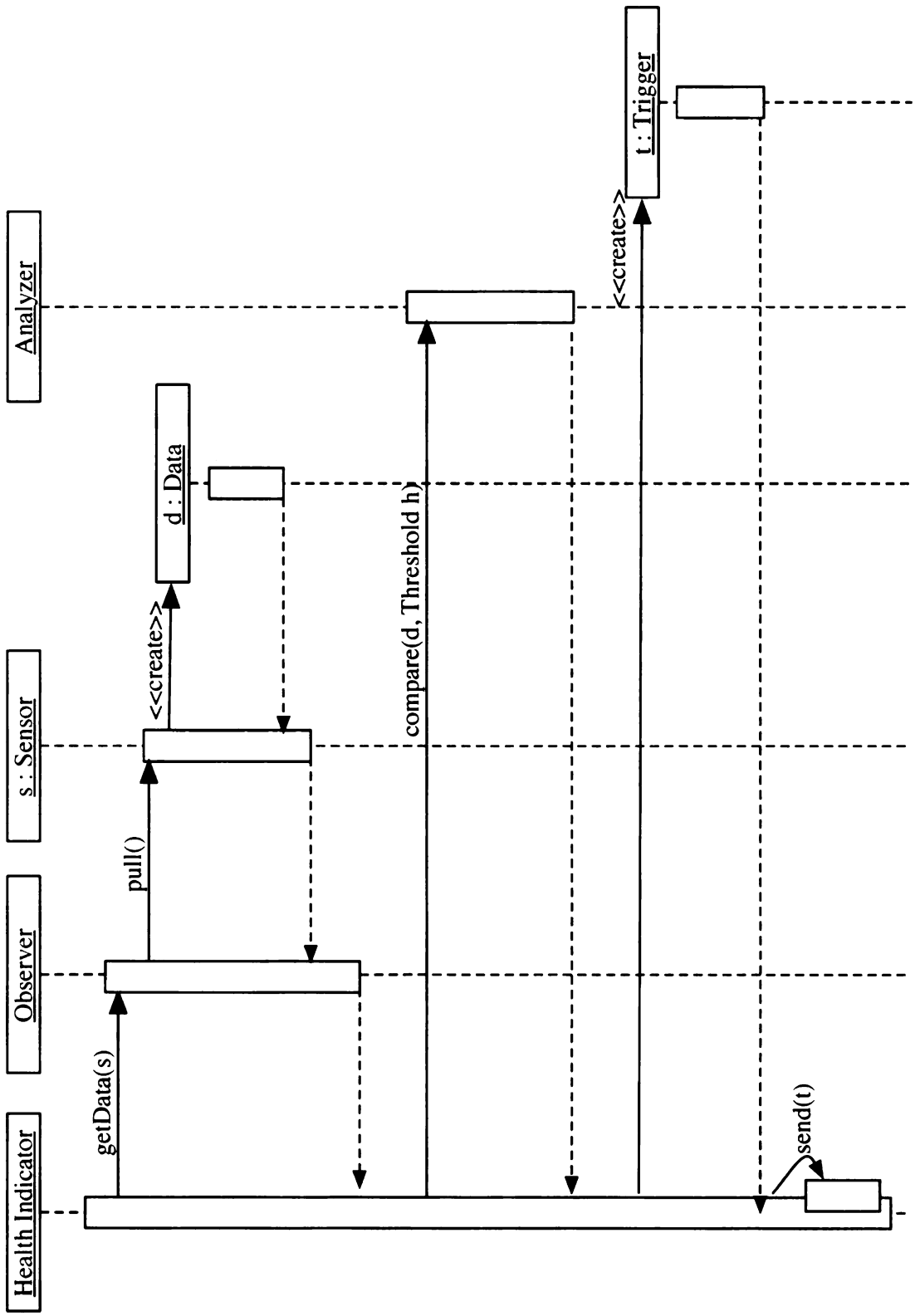

Figure 5.20: UML sequence diagram example of the *Adaptation Detector (88)* Pattern

□ (Observer.getData(Sensor) →

◊ Analyzer.compare(Data,Threshold))

This property is desirable because it guarantees that values reported by the monitoring process will be compared against thresholds that define what constitutes normal behavior from abnormal behavior. Some application domains may want to further strengthen this property by imposing a timing constraint.

- **Property 2:**

  Globally, it is always the case that if a Threshold is exceeded, then eventually a Trigger will be sent to the decision-making process.

  □ (Analyzer.compare(Data).'True' →

  ◊ HealthIndicator.send(Trigger))

  This property is desirable because it ensures that whenever a constraint violation is detected, the decision-making process will be notified so it may select an appropriate reconfiguration plan.

Related Design Patterns:

- ***Sensor-Factory (41)* Design Pattern:**

  This pattern can be used to deploy sensors across a distributed environment and probe components.

- ***Reflective Monitoring (50)* Design Pattern:**

  This design pattern can be used to determine when a monitored attribute exceeds a specific constraint.

- ***Content-based Routing (59)* Design Pattern:**

The *Adaptation Detector (88)* pattern could subscribe to the *Content-based Routing (59)* pattern notification service. Whenever a monitoring value is published, the *Adaptation Detector (88)* pattern is notified of the update.

- ***Case-based Reasoning (68)* Design Pattern:**

  This pattern can be applied to satisfy simple adaptation requirements that typically involve only one reconfiguration plan.

- ***Architecture-Based (97)* Design Pattern:**

  This design pattern can be used to represent a system and its reconfiguration plans as architectural models. Models that satisfy the adaptation requirements indicate how the system should be reconfigured.

- ***TradeOff-Based (106)* Design Pattern:**

  This design pattern can be used to select a reconfiguration plan that best balances multiple objectives at run time. The *Adaptation Detector (88)* pattern could be used to detect changes that warrant an adaptation.

Known Uses:

- SmartEvents (part of XUES) [37].
- PBX - Design Patterns for Software Health Monitoring [59].
- Java Agents for Monitoring and Management (JAMM) - event gateway [77].
- Kinesthetics eXtreme (KX) [79].

5.5.7 *Architecture-Based (97)* Pattern

Classification:

Structural - Decision-Making.

Intent:

Provide an architecture-based approach for selecting reconfiguration plans.

Context:

The *Architecture-Based (97)* Pattern may be used when:

- Reconfiguration plans are expected to change regularly.
- Reconfiguration plans should be reused.

Motivation:

Low-level adaptation mechanisms that are tightly coupled with application code present two major challenges to developers [29]. First, it is difficult to correctly determine the true source of the problem and the remedial action at such detailed levels of abstraction. Second, adaptation policies are not localized, thereby hindering their evolution. Architectural perspectives, on the other hand, shift focus away from source code to coarse grained components and their interconnections [67]. Architectural approaches externalize adaptation and facilitate the reuse of reconfiguration plans. As an application adapts and evolves, however, preserving an accurate and consistent model of the system and its constituent parts becomes increasingly difficult. The *Architecture-Based (97)* Design Pattern manages the evolution of an architectural model while preserving a correspondence between the model and the implementation. Specifically, architectural models are used to represent both the current state of the system as well as the possible target systems after a reconfiguration has been applied.

Figure 5.21 shows a use-case diagram of the *Architecture-Based (97)* Pattern.

The main goal of this design pattern is to manage the evolution of a system's architectural model.

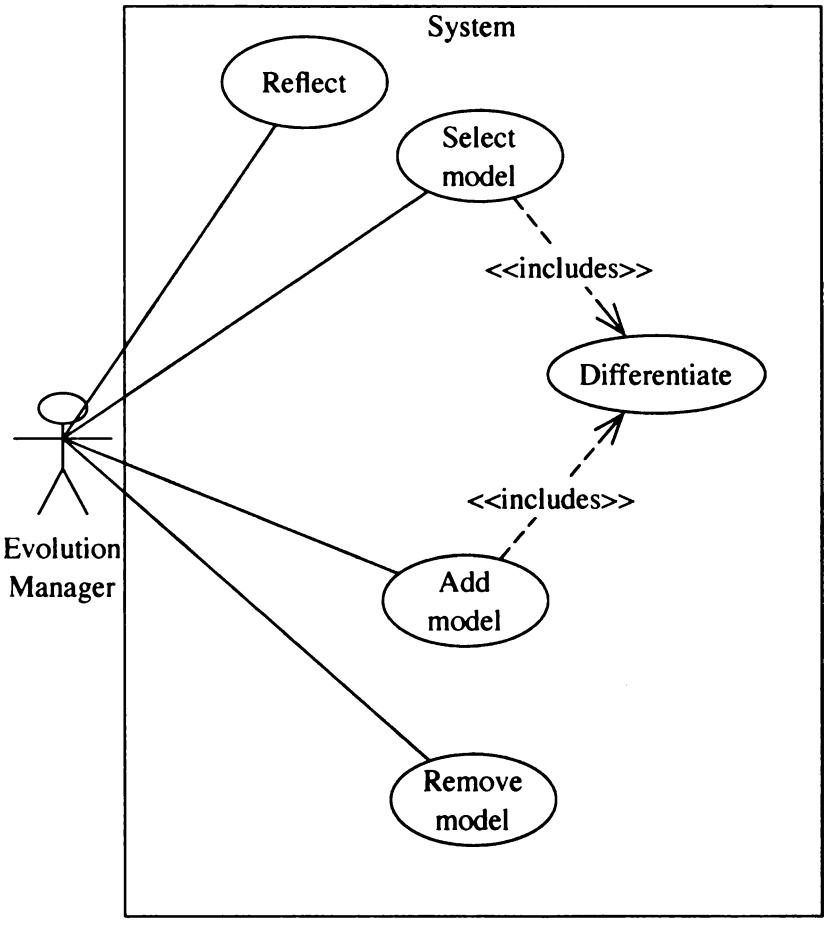

Figure 5.21: UML use-case diagram of the *Architecture-Based (97)* Pattern

Use-Case:	Reflect
Actors:	Evolution manager
Description:	Update the architectural model representation of the system such that it is consistent with the system implementation.
Includes:	-

Use-Case:	Select model
Actors:	Evolution manager
Description:	Select an architectural model of the system that satisfies the adaptation requirements.
Includes:	Differentiate

Use-Case:	Add model
Actors:	Evolution manager
Description:	Insert a new architectural model of the system into the repository.
Includes:	Differentiate

Use-Case:	Differentiate
Actors:	-
Description:	Determine the difference between two architectural models. This can be used to either build a reconfiguration plan or avoid duplicate architectural models in the repository.
Includes:	-

Use-Case:	Remove model
Actors:	Evolution manager
Description:	Delete an architectural model from the repository.
Includes:	-

Structure:

A UML class diagram for the *Architecture-Based (97)* Pattern can be found in Figure 5.22.

Architecture-based approaches for self-adaptive systems typically perform two key tasks when selecting a reconfiguration strategy [29, 67]. First, the architectural models must be kept consistent with respect to the system's current implementation. Second, candidate reconfiguration plans must be evaluated against specific constraints to determine if they satisfy the adaptation requirements. The *Architecture-Based (97)* Design Pattern accomplishes the first task by introducing an Evolution Manager that

updates the current architectural model in response to the application's evolution. To accomplish the second task, a Repair Engine searches through a collection of architectural models and checks whether any of them satisfy the adaptation requirements. If a model that satisfies the specified requirements is found, then the key differences between the source and target architectural models are identified. Identifying these differences will facilitate the development of a reconfiguration plan that will adapt the application as needed.

Behavior:

Figure 5.23 shows a UML sequence diagram for an example of the *Architecture-Based (97)* Pattern in an adaptive system. In this specific scenario, the Evolution Manager updates its architectural model of the implemented system. The Evolution Manager begins the reconfiguration selection process by first invoking the Repair Engine to search for an architectural model that satisfies the adaptation requirements. The Repair Engine retrieves architectural models from the ArchitecturalRepository and then uses the Constraint Checker to determine whether that architectural model satisfies the required properties. Once an architectural model is found, the Repair Engine determines what are the key differences between the source and target architectural models. With this information, the Evolution Manager can suggest a reconfiguration plan that will correctly adapt the application.

Participants:

- Architectural Model: Each model represents a specific configuration of the entire application. The overall observable behavior results from the different configurations of both Components and Connectors.
- ArchitecturalRepository: Provides access to a collection of Architectural Models.
- Component: These are responsible for implementing an application's behavior.

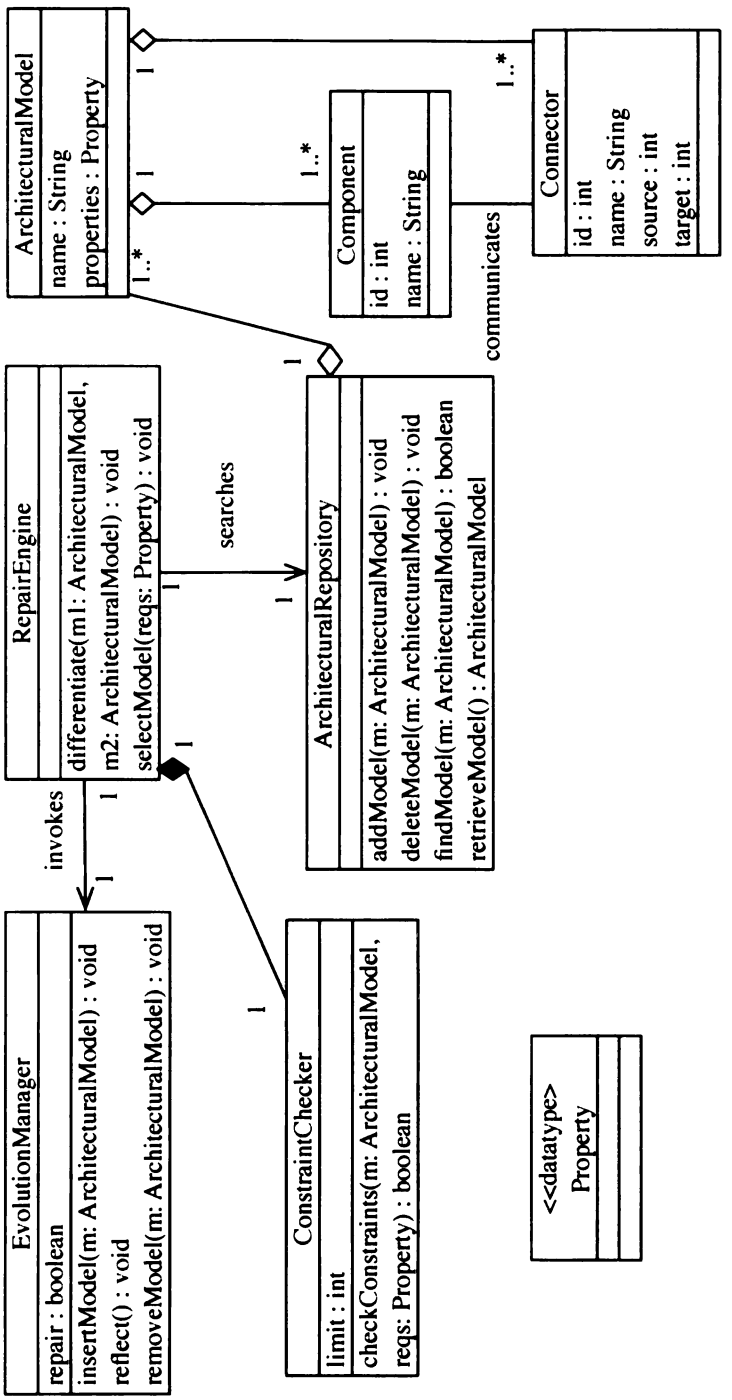

Figure 5.22: UML class diagram of the *Architecture-Based (97)* Pattern

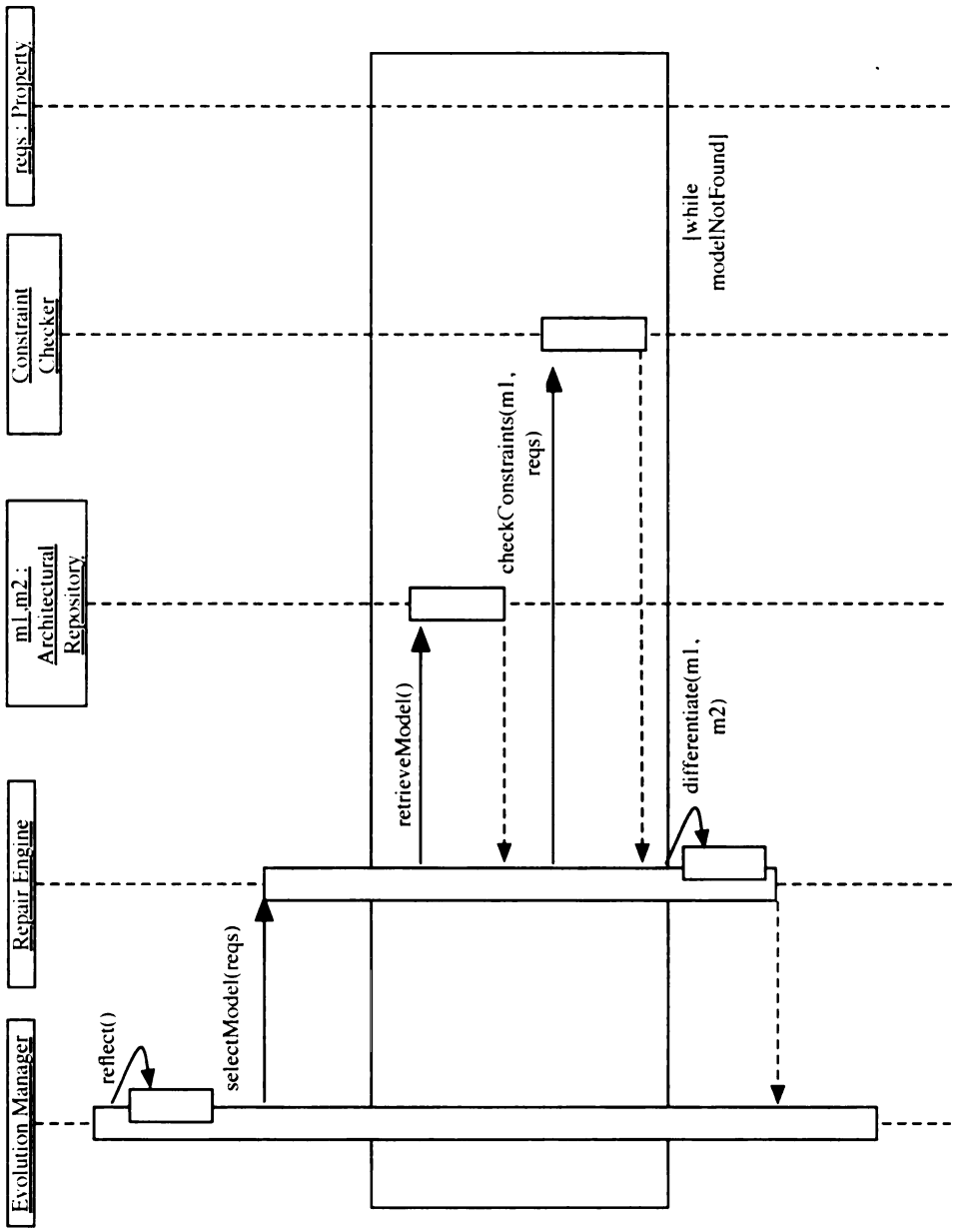

Figure 5.23: UML sequence diagram example of the *Architecture-Based (97)* Pattern

- Connector: These represent the various interconnections between Components in the system.

- Constraint Checker: The main function of this class is to evaluate whether a particular Architectural Model satisfies a set of specific adaptation requirements.

- Evolution Manager: This class oversees the two primary goals of the *Architecture-Based (97)* Design Pattern. First, monitoring information provided by the system (see *Adaptation Detector (88)* Design Pattern for more information) is used to update the current architectural model representation of the system. Second, the Evolution Manager guides a constraint-based approach at selecting the appropriate reconfiguration plan.

- Repair Engine: This class is responsible for searching and extracting Architectural Models from the Model Repository. It also supports the addition and removal of Architectural Models, thus facilitating the evolution of reconfiguration plans.

  Searches and extracts Architectural Models from the ArchitecturalRepository. It also supports the addition and removal of Architectural Models at run time. To differentiate between different Architectural Models, the Repair Engine employs graph-based algorithms to identify structural differences.

Consequences:

1. Reconfiguration plans are localized, thus facilitating the evolution of reconfiguration strategies at run time.

2. In general, architectural models are not application specific, thereby facilitating the reuse of reconfiguration plans between different systems [29].

3. A computational overhead is introduced by searching and evaluating potential reconfiguration models when adapting the system.

Constraints:

- **Property 1:**

  Globally, it is always the case that if a particular model does not satisfy some requirement, then that model is never selected for execution.

  $\Box$ (`ConstraintChecker.checkConstraints(m,reqs).'False'` $\rightarrow$

  $\neg$ `RepairEngine.selectModel(m)`)

  This safety property guarantees that the system will not select a reconfiguration plan that violates a specific property.

Related Design Patterns:

- ***Adaptation Detector (88)* Design Pattern:**

  This pattern can be applied to determine when a reconfiguration is required. It can also localize the source of the problem and guide the search for a reconfiguration plan.

- ***Divide and Conquer (78)* Design Pattern:**

  This pattern can be applied to decompose a complex reconfiguration plan into simpler tasks.

- ***Component Insertion (115)* Design Pattern:**

  This pattern can be used to safely insert a component at run time.

- ***Component Removal (125)* Design Pattern:**

  This pattern can be used to safely remove a component at run time.

- ***Server Reconfiguration (135)* Design Pattern:**

This pattern can be used to safely reconfigure a server architecture at run time. The *Architecture-Based (97)* pattern can be used to represent the server architecture.

- ***Decentralized Reconfiguration (145)* Design Pattern:**

  This pattern can be used to safely reconfigure a decentralized architecture at run time. The *Architecture-Based (97)* pattern can be used to represent the overall system configuration.

Known Uses:

- Rainbow Adaptation Framework [27, 29].
- MADAM [31].
- Distributed Configuration Routing (DCR) [42].
- C2 [67].
- Kinesthetics eXtreme (KX) [79].

5.5.8 *TradeOff-Based (106)* Pattern

Classification:

Structural - Decision-Making.

Intent:

Systematically select a reconfiguration plan that best balances multiple objectives.

Context:

The *TradeOff-Based (106)* Pattern may be used when:

- more than one dimension must be considered for adaptation.
- multiple reconfiguration plans satisfy the adaptation requirements.

Motivation:

Rule-based approaches are inadequate for expressing the adaptation expertise involved in trade-off decisions in the presence of multiple objectives [16]. Maintaining consistency between the trade-off preferences becomes unmanageable as the number of cases grow. In addition, these approaches require policy makers to be intimately familiar with low-level details of system function, a requirement antagonistic to adaptive and autonomic principles [80]. Utility functions map each possible state of an entity into a real scalar value, thus providing an objective function for selecting reconfigurations. The *TradeOff-Based (106)* design pattern supports multiple heterogeneous services by encapsulating differences at a local level and providing a uniform mean of communicating requirements to an arbiter.

Figure 5.24 shows a use-case diagram of the *TradeOff-Based (106)* Pattern. The main goal of this design pattern is to evaluate reconfiguration plans during an adaptation and select the best plan that balances multiple objectives.

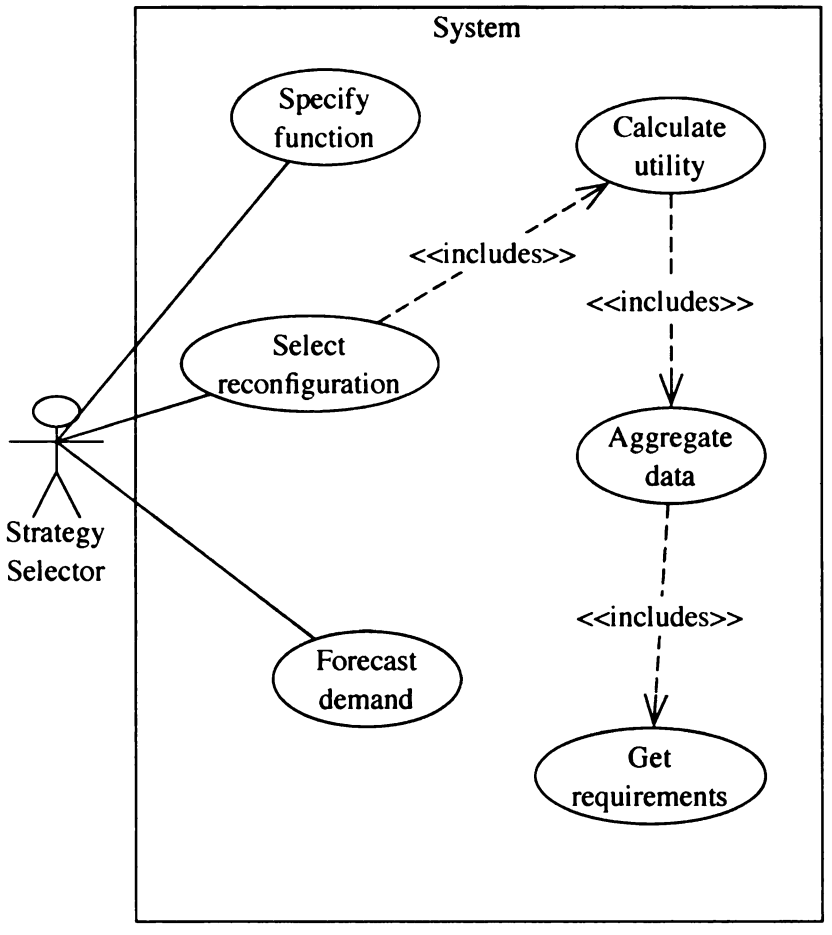

Figure 5.24: UML use-case diagram of the *TradeOff-Based (106)* Pattern

Use-Case:	Specify function
Actors:	Strategy selector
Description:	Select the utility function to be used.
Includes:	-

Use-Case:	Select reconfiguration
Actors:	Strategy selector
Description:	Select the reconfiguration plan with the highest utility value.
Includes:	Calculate utility.

Use-Case:	Calculate utility
Actors:	-
Description:	Apply the utility function to various reconfiguration plans.
Includes:	Aggregate data.

Use-Case:	Aggregate data
Actors:	-
Description:	Aggregate the different reconfiguration objectives so they may be uniformly input to the utility function.
Includes:	Get requirements.

Use-Case:	Get requirements
Actors:	-
Description:	Elicit different reconfiguration objectives.
Includes:	-

Use-Case:	Forecast demand
Actors:	Strategy selector
Description:	Delete an architectural model from the repository.
Includes:	-

Structure:

A UML class diagram for the *TradeOff-Based (106)* Pattern can be found in Figure 5.25.

It is important to evaluate all objectives when selecting a reconfiguration strategy, even if the objectives are heterogeneous in nature. The *TradeOff-Based (106)* Design Pattern provides an approach to integrate the various objectives and select the best reconfiguration plan that balances those objectives. This design pattern provides mechanisms to collect reconfiguration objectives and aggregate them into a format suitable for a utility function. The Utility Function provides the means to normalize all demands and the utility they provide to the system so they may be compared

objectively. This particular approach incorporates not only what a reconfiguration plan proposes to be its utility value to the system, but also provides a means to incorporate a predictive factor based on previous reconfigurations. The Inference Engine selects the Objective that provides the maximum numerical utility value.

Participants:

- Arbiter: Mediates the interaction between multiple Objectives. This class pre-processes multiple Objectives so they can be input to the Utility Function.
- Client: Represents a stakeholder in the system. Each Client has a set of objectives it wants to satisfy.
- Data Aggregator: Combines and pre-processes Objectives so they can be input to a Utility Function.
- Decision: This class represents the reconfiguration plan with the maximum utility value.
- Demand Forecaster: Predicts the utility gained by a specific Objective. This prediction is determined based on previous knowledge about Objectives.
- Inference Engine: This class is responsible for coordinating two major steps of the reconfiguration plan selection process. First, it calculates the utility value for each Objective. Second, it selects the Objective that produces the maximal utility value.
- Objective: Represents any reconfiguration goal in the system. For instance, two Objectives could be "increase bandwidth" and another could be "increase latency".
- Utility Function: Represents any function that objectively compares multiple Objectives in terms of utility gain. These functions should return a numerical

Figure 5.25: UML class diagram of the *TradeOff-Based (106)* Pattern

value in the range of [0,1] for each Objective.

Behavior:

Figure 5.26 shows a UML sequence diagram for an example of the *TradeOff-Based (106)* Pattern in an adaptive system.

In this specific scenario, the Inference Engine first retrieves the various Objectives that will be considered when selecting a reconfiguration plan. The Arbiter retrieves each Objective from the various Clients requesting an adaptation. The Arbiter also processes this information so it can be input to a utility function. The Inference Engine also invokes a Demand Forecaster to determine, based on prior reconfiguration experiences, what the estimated utility will be from a specific reconfiguration plan. Meanwhile, the Inference Engine applies the Utility Function to each Objective. Once the maximum utility value is determined, the Inference Engine creates a Decision that contains the selected reconfiguration plan.

Consequences:

1. Multiple dimensions of reconfiguration and its outcomes can be evaluated uniformly at run time.

2. The selected reconfiguration plan provides the maximum utility (desired outcome) for the entire system.

3. Developers must express a utility function to normalize each objective and its outcome. Since each utility function is application-specific, the reuse of utility functions is limited.

Constraints:

- **Property 1:**

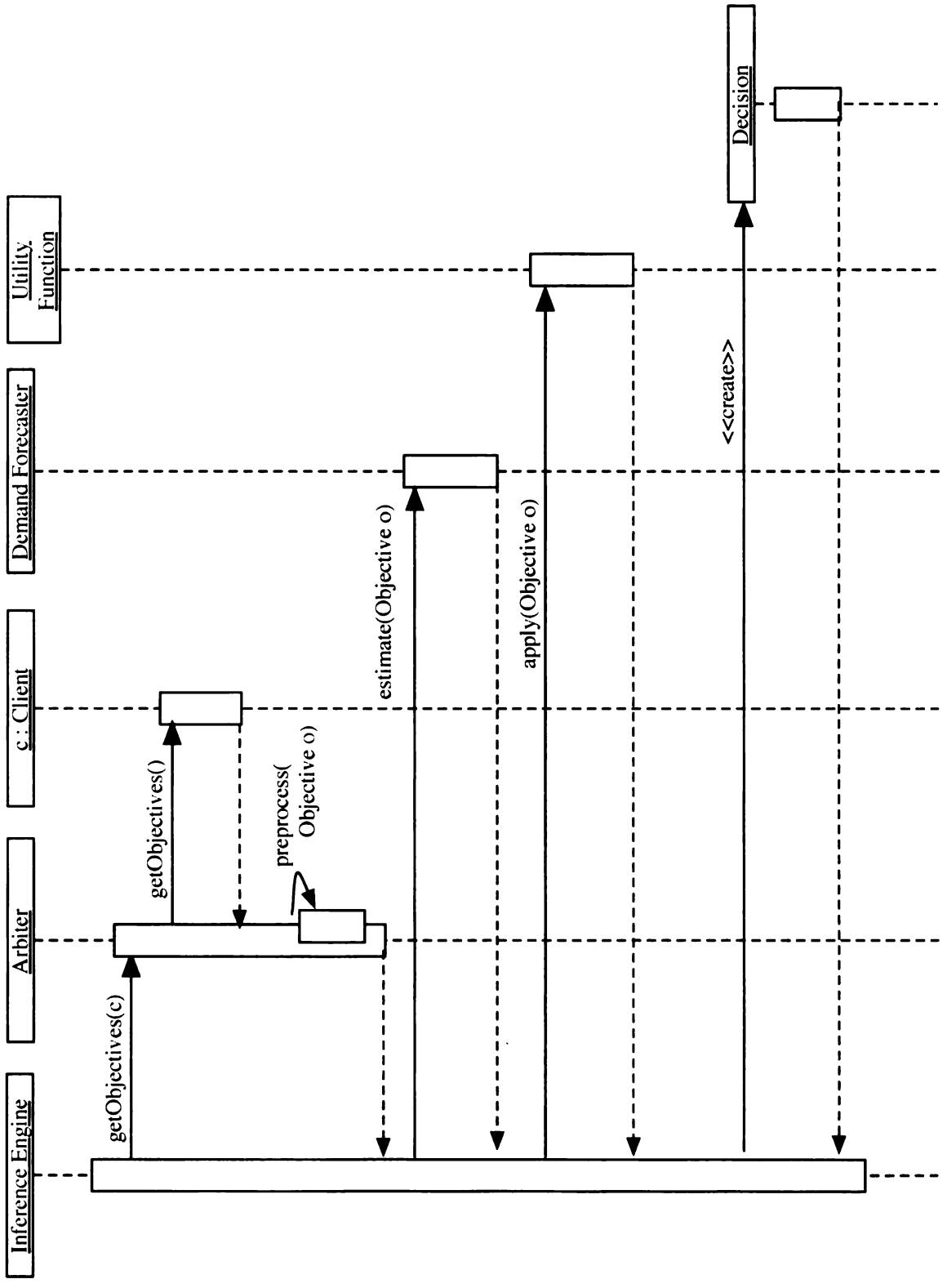

Figure 5.26: UML sequence diagram example of the *TradeOff-Based (106)* Pattern

Globally, it is always the case that if Arbiter receives Objectives, then the InferenceEngine will eventually select the optimal reconfiguration plan.

$\Box$ ((Arbiter.getObjective(Objective) $\rightarrow$

$\Diamond$InferenceEngine.selectOptimal())

This liveness property ensures that the reconfiguration plan with maximum utility is eventually selected whenever an Objective is obtained. This specific property can be strengthened by specifying an additional timing constraint.

- **Property 2:**

Globally, it is always the case that if InferenceEngine calculates the utility of a particular Objective, then InferenceEngine applies the selected utility function as well as estimates the probable utility based on past observations.

$\Box$((InferenceEngine.calculateUtilities(Objective)) $\rightarrow$

(UtilityFunction.apply(Objective) $\wedge$ DemandForecaster.estimate(Objective)))

This safety property ensures that whenever the InferenceEngine calculates the utility of a given Objective, it does so by applying the selected utility function and by estimating the probable utility value based on previous observations.

Related Design Patterns:

- ***Sensor-Factory (41)* Design Pattern:**

This pattern can be used to deploy sensors across a distributed environment and probe components. The monitoring information gathered from this approach can be used to maintain consistency between the architectural model and the implementation.

- ***Adaptation Detector (88)* Design Pattern:**

  This pattern can be applied to determine when a reconfiguration is required. It can also localize the source of the problem and guide the search for a reconfiguration plan.

Known Uses:

- Rainbow Adaptation Framework [16].
- Unity (Autonomic Prototype by IBM) [17].
- MADAM [31].
- Utility Based Allocation [80].

5.5.9 *Component Insertion (115)* Pattern

Classification:

Behavioral - Reconfiguration.

Intent:

Safely insert and initialize a component at run time.

Context:

The *Component Insertion (115)* Pattern may be used when:

- components need to be added at run time.
- the application disallows downtimes.
- the functional logic has been instrumented with an interface to guide components into *active*, *passive*, and *quiescent* states.

Motivation:

Inserting components into an executing system presents some subtle challenges. In many circumstances it is not enough to load a new component into memory and immediately begin its execution process as this may leave the system in an inconsistent state [46]. A new component must be given the chance to initialize itself, either to a default state or to a previously preserved state. In addition, the new component must be properly linked with the rest of the system so it may communicate and process information as intended. The *Component Insertion (115)* design pattern controls the operational status of the system such that components can be safely inserted at run time.

Figure 5.27 shows a use-case diagram of the *Component Insertion (115)* Pattern. The goal of this design pattern is to guide the insertion of a new component at run time.

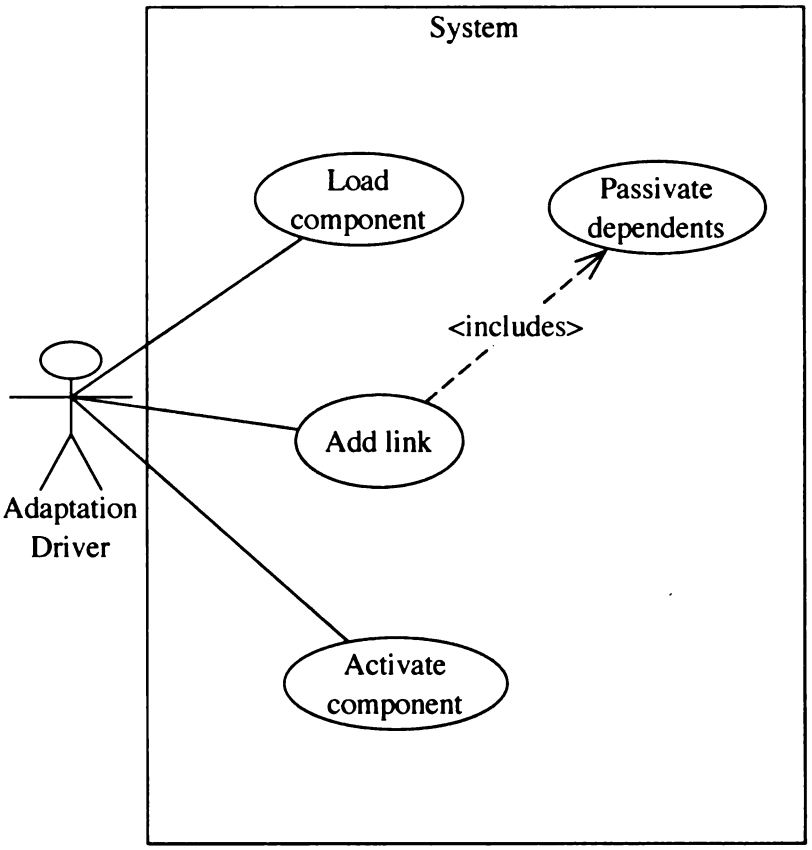

Figure 5.27: UML use-case diagram of the *Component Insertion (115)* Pattern.

Use-Case:	Load component
Actors:	Adaptation Driver
Description:	Load a new component into the executing system.
Includes:	-

Use-Case:	Add link
Actors:	Adaptation Driver
Description:	Create a connection between two components in the system.
Includes:	Passivate dependents.

Use-Case:	Passivate dependents
Actors:	-
Description:	Guide every component that will share a connection with the new component to a state in which it cannot initiate new transactions and it is not currently engaged in a transaction that it initiated.
Includes:	-

Use-Case:	Activate component
Actors:	Adaptation Driver
Description:	Instruct a component to resume its normal behavior.
Includes:	-

Structure:

A UML component diagram for the *Component Insertion (115)* Pattern can be found in Figure 5.28.

The *Component Insertion (115)* design pattern coordinates the sequence of steps required to safely insert a component at run time. Components participating in the reconfiguration process must *realize* the States interface. The States interface can be used by the Adaptation Driver to issue commands that will drive Components into active, passive, or quiescent states. A Change Manager interacts with the executing environment and provides support for basic reconfiguration primitives and rules for loading and unloading components. The Adaptation Driver uses the Change Manager component to effect the necessary changes throughout the system as specified by the Reconfiguration Rules.

Participants:

- Adaptation Driver: Oversees the reconfiguration process of inserting a new Component into the system.
- Change Manager: Provides support for loading and unloading Components and

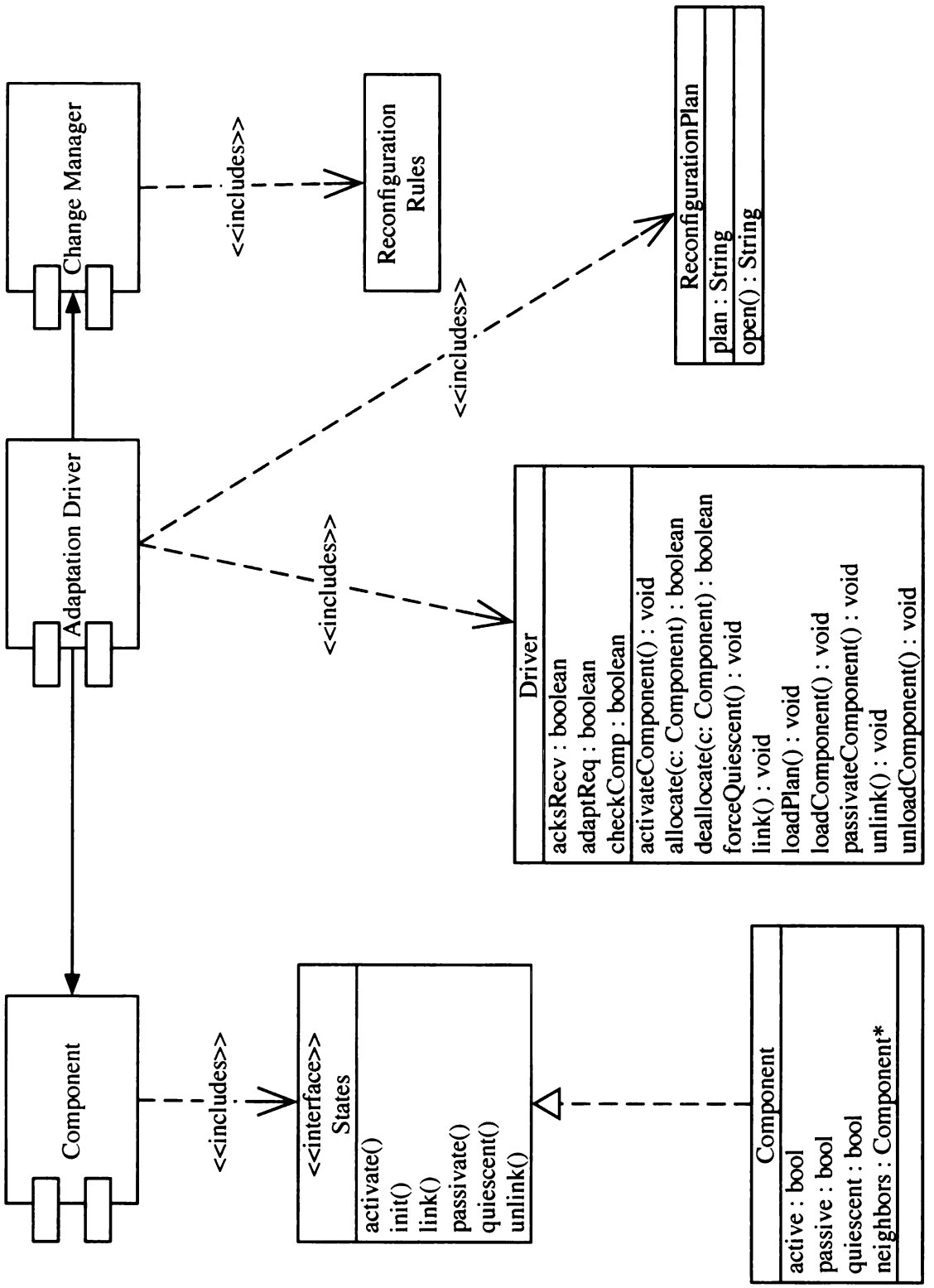

Figure 5.28: UML component diagram of the *Component Insertion (115)* Pattern

their interconnections.

- Component: Represents any executable component that can be deployed throughout the system. Each Component that may be involved in a reconfiguration must realize the States interface. Note that each Component maintains a pointer to its neighboring Components (this holds for all remaining reconfiguration patterns).

- Driver: Manages the operational states of components involved in a reconfiguration through the States interface. Specifically, it can guide a Component to *active*, *passive*, and *quiescent* states.

- Reconfiguration Plan: Stores the specific sequence of instructions for reconfiguring the system at run time.

- Reconfiguration Rules: Contains rules and instructions for specifying how basic reconfiguration operations are carried out in the system. Some basic reconfiguration operations include component insertion, removal, and swapping.

- States: This interface forces a Component to define which functional states correspond to *active*, *passive*, and *quiescent* states. An Adaptation Driver can control a Component's behavior through this interface.

Behavior:

A UML statechart diagram for the *Component Insertion (115)* pattern is shown in Figure 5.29. Specifically, this statechart models the possible behavior of the Driver class found in the Adaptation Driver component. First the object loads the reconfiguration plan that must be applied. At this step, we assume that a component must be inserted into the application at run time, otherwise this reconfiguration pattern would not be applicable. Thus, the new Component is dynamically loaded into the

system. Before the Component becomes operational, however, it must first be properly initialized to either a default state or some previously preserved state. After the Component has been initialized, it is set to a *passive* state. The reconfiguration plan is analyzed once more to determine which Components will share a connection with the new Component. These *neighboring* Components are sent *passivate* commands. After the Driver receives all pending acknowledgments that neighboring Components are *passive*, then the new Component can be *linked* by the Driver. Lastly, *activate* commands are sent to the affected components so they may resume their normal behavior.

Consequences:

1. Components can be inserted at run time without leaving the system in an inconsistent state.

2. Components are properly initialized to a consistent state before becoming active.

3. Several components may become *passive* during the reconfiguration process. These components may not initiate any new transactions while they are passive, thereby inducing a processing delay across the system until the reconfiguration process is complete.

4. Components must provide an interface to reach active, passive, and quiescent states. This interface can be either built-in during development or inserted through techniques such as AOP.

Constraints:

- **Property 1:**

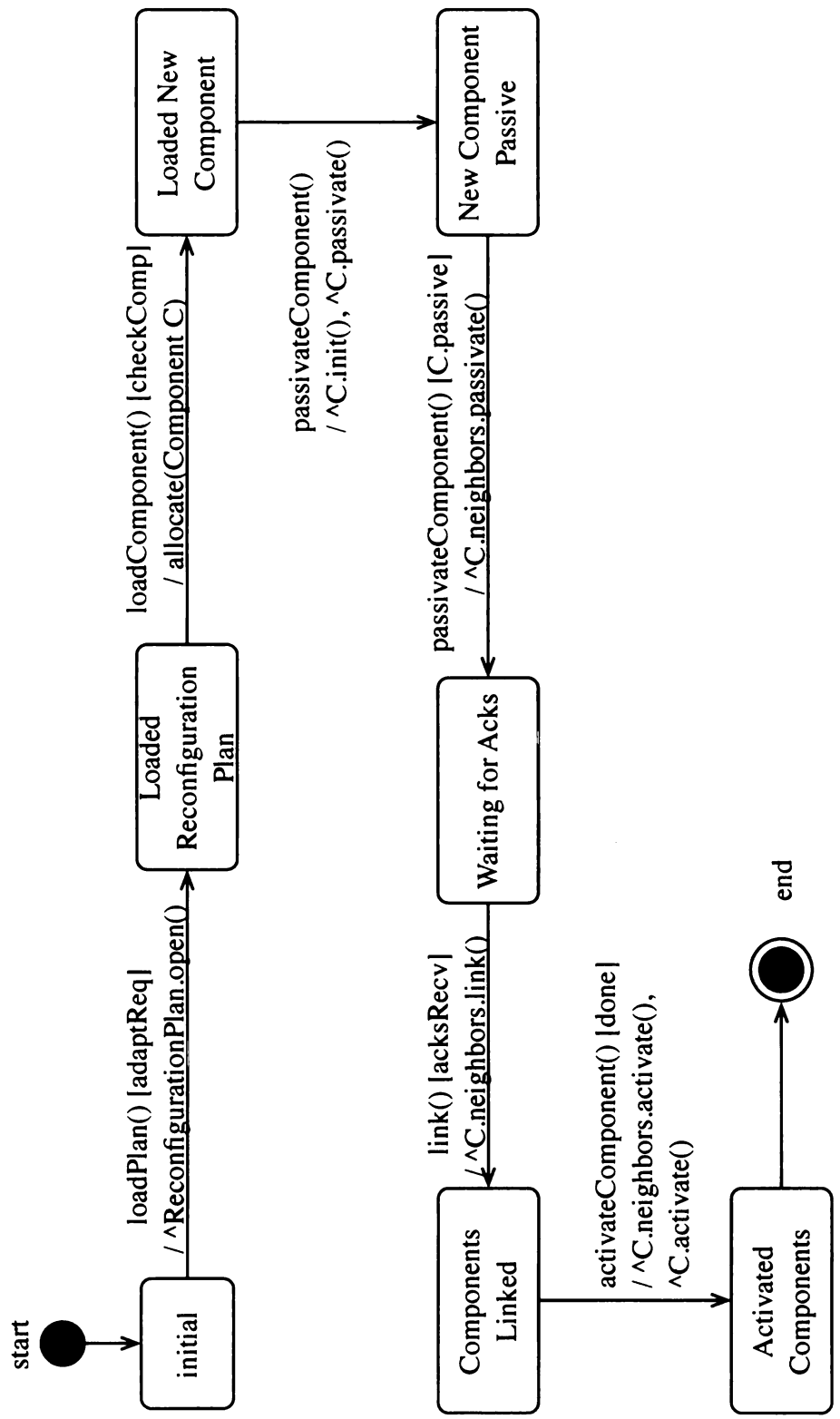

Figure 5.29: UML state diagram example of the *Component Insertion (115)* Pattern

During adaptation, neighboring components that will be linked with x must first be guided to a passive state. To achieve this, a restricted condition, R_{Cond} must be established such that the system may safely reconfigure in bounded time.

$$R_{Cond} = \Box(\neg Component.neighbor.active() \wedge \neg Component.link()).$$

This restricted state, R_{Cond} prevents neighboring components from being activated by the AdaptationDriver until the new component has been linked to them. Likewise, R_{Cond} also prevents x from being linked to active components. This restricted space can be enforced by preventing neighboring components from accepting new transaction requests until the reconfiguration is complete.

The complete guided-adaptation property in A-LTL is given by:

$$((Component.neighbors == null) \wedge (\diamond(adaptReq) \overset{true}{\rightharpoonup} R_{Cond})) \overset{true}{\rightharpoonup}$$

$$(Component.neighbors! = null).$$

This A-LTL property ensures that when a component is inserted into the system at run time, it does not get linked to any other component that is currently active. Thus, when A_{Req} is received the corresponding neighbors must first be guided to a passive state before links can be established between these components. Links can be created once the new component's neighbors are in a passive state.

- **Property 2:**

  If a Component has not been initialized, then it is never the case that the Component is linked to its neighboring components.

$\Box$ ($\neg$ Component.initialize() $\rightarrow$ $\neg$ Component.link())

This safety property guarantees that if a component has not been initialized, then it will not be connected to any other components.

Related Design Patterns:

- **Case-based Reasoning (68) Design Pattern:**

  This pattern can be used to select a reconfiguration plan based on the available monitoring information. If the reconfiguration plan involves adding components at run time, then the *Component Insertion (115)* design pattern can perform this task.

- **Divide and Conquer (78) Design Pattern:**

  This pattern can be used to determine the specific sequence of steps required to safely perform a reconfiguration. Whenever a step requires that a component be inserted, it can be carried out by the *Component Insertion (115)* Pattern.

- **Architecture-Based (97) Design Pattern:**

  This pattern can be used to select a reconfiguration plan. Any reconfiguration that involves adding components to the system can use the *Component Insertion (115)* Pattern.

- **TradeOff-Based (106) Design Pattern:**

  This pattern can be used to select a reconfiguration plan that best balances the overall set of objectives that various stakeholders may have. If the reconfiguration plan includes inserting components, then the *Component Insertion (115)* Pattern can be used.

- **Server Reconfiguration (135) Design Pattern:**

This pattern can be used to reconfigure an application structured as a server - client architecture. Components can be inserted into the server architecture through the *Component Insertion (115)* Pattern.

Known Uses:

- Znews.com - Rainbow framework [16].
- Software Reconfiguration Patterns [34].
- Monitor - Dynamic Reconfiguration in Distributed Systems [46].
- Evolving Philosophers - Dynamic Change Management [57].

5.5.10 *Component Removal (125)* Pattern

Classification:

Behavioral - Reconfiguration.

Intent:

Safely remove a component at run time.

Context:

The *Component Removal (125)* Pattern may be used when:

- components need to be removed at run time.
- the application disallows downtimes.
- the functional logic has been instrumented with an interface to guide components into *active*, *passive*, and *quiescent* states.

Motivation:

Safely removing a component from a system at run time is a difficult task. If a component is removed without first preparing the application for such a change, then the entire application may be left in an inconsistent state [57]. For instance, if a component is removed arbitrarily, then some previously initiated transactions may not terminate properly. The *Component Removal (125)* design pattern safely removes a component at run time by explicitly controlling the operational status of components that are directly connected to the component being removed. As a result, the *Component Removal (125)* design pattern ensures all transactions have completed before a component is removed.

Figure 5.30 shows a use-case diagram of the *Component Removal (125)* Pattern. The goal of this design pattern is to guide the removal of a component at run time.

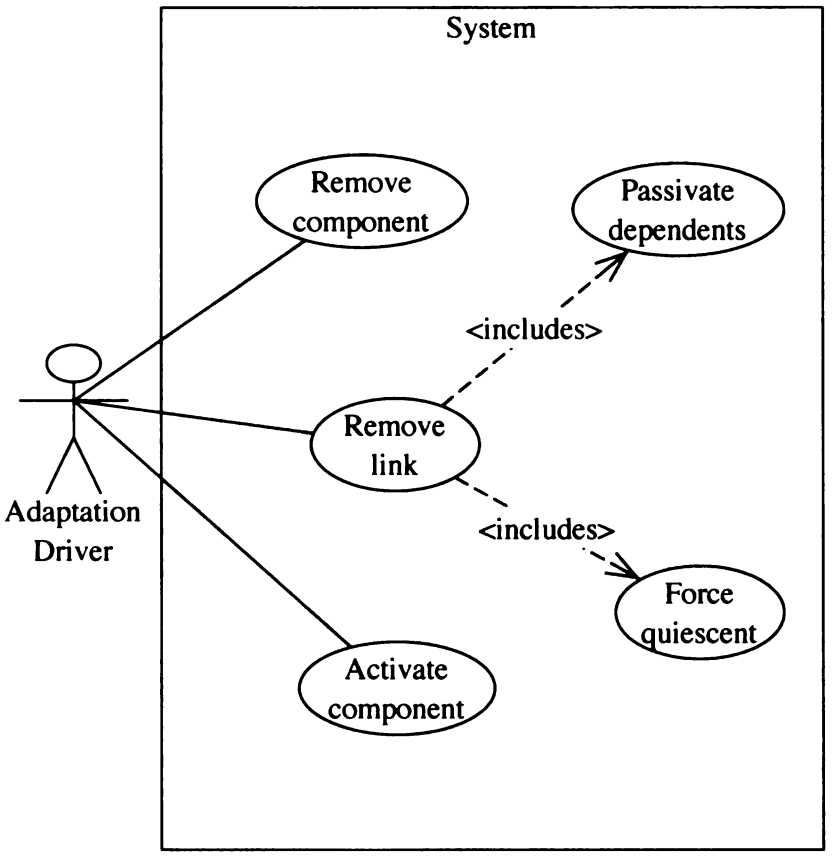

Figure 5.30: UML use-case diagram of the *Component Removal (125)* Pattern

Use-Case:	Remove component
Actors:	Adaptation Driver
Description:	Unload a component from the executing system.
Includes:	-

Use-Case:	Remove link
Actors:	Adaptation Driver
Description:	Remove a connection between two components in the system.
Includes:	Passivate dependents, Force quiescent.

Use-Case:	Passivate dependents
Actors:	-
Description:	Guide every component that shares a connection with the component to be removed to a state in which it cannot initiate new transactions and it is not currently engaged in a transaction that it initiated.
Includes:	-

Use-Case:	Force quiescent
Actors:	-
Description:	Instruct a component to reach a quiescent state.
Includes:	-

Use-Case:	Activate component
Actors:	Adaptation Driver
Description:	Instruct a component to resume its normal behavior.
Includes:	-

Structure:

A UML component diagram for the *Component Removal (125)* Pattern can be found in Figure 5.31.

The *Component Removal (125)* design pattern coordinates the sequence of steps required to safely remove a component at run time. Components participating in the reconfiguration process must *realize* the States interface. The States interface can be used by the Adaptation Driver to issue commands that will drive Components into active, passive, or quiescent states. No Component will be removed until every affected component has reached their required state. A Change Manager interacts with the executing environment and provides support for basic reconfiguration primitives and rules for unloading and unlinking a component. The Adaptation Driver uses the Change Manager component to effect the necessary changes throughout the system as specified by the Reconfiguration Rules.

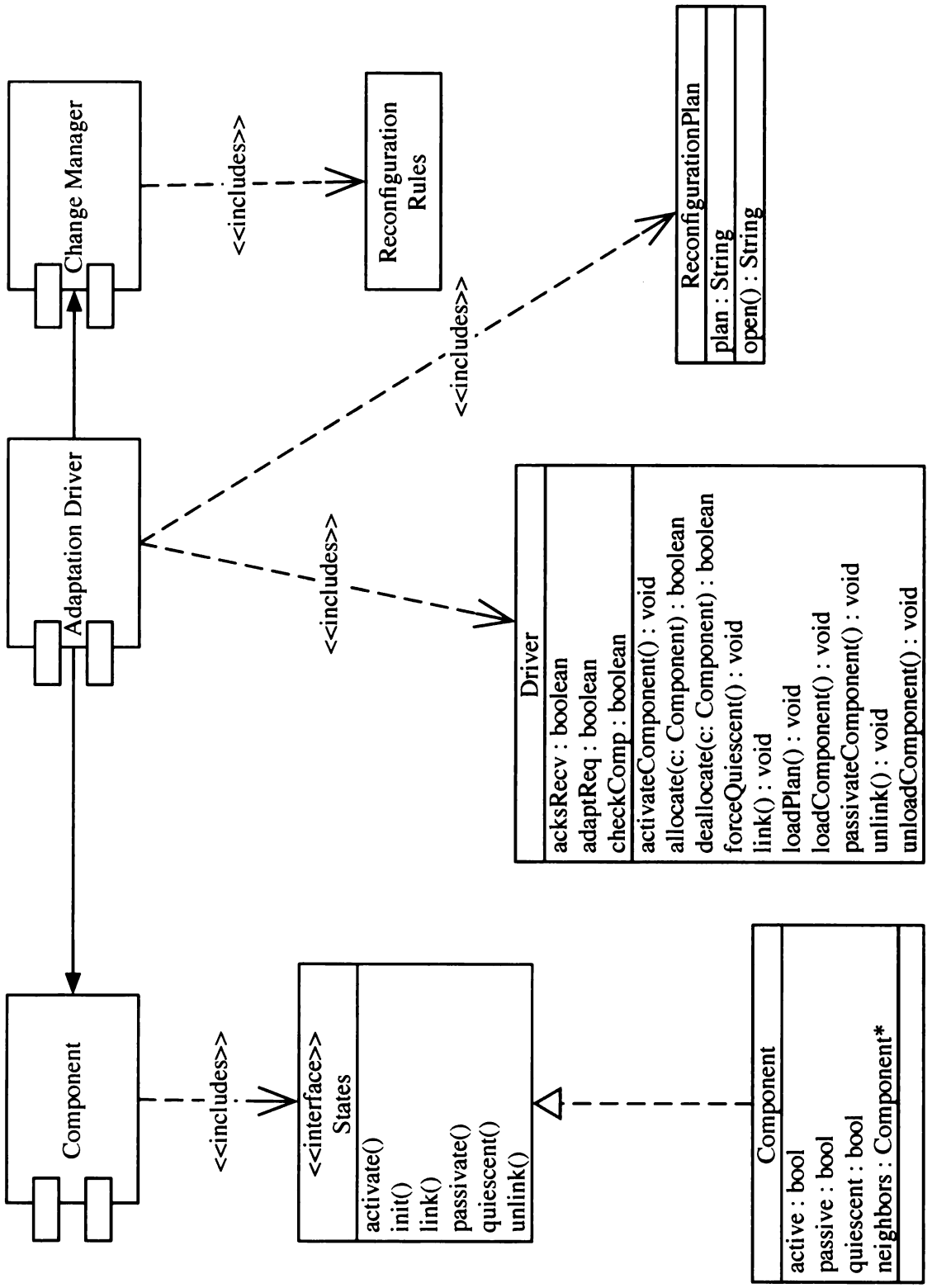

Figure 5.31: UML component diagram of the *Component Removal (125)* Pattern

Participants:

- Adaptation Driver: Oversees the reconfiguration process of removing a Component from the system.
- Change Manager: Provides support for loading and removing Components and their interconnections.
- Component: Represents any executable component that can be deployed throughout the system. Each Component that may be involved in a reconfiguration must realize the States interface.
- Driver: Manages the operational states of components involved in a reconfiguration through the States interface. Specifically, it can guide a Component to *active*, *passive*, and *quiescent* states.
- Reconfiguration Plan: Stores the specific sequence of instructions for reconfiguring the system at run time.
- Reconfiguration Rules: Contains rules and instructions for specifying how basic reconfiguration operations are carried out in the system. Some basic reconfiguration operations include component insertion, removal, and swapping.
- States: This interface forces a Component to define which functional states correspond to *active*, *passive*, and *quiescent states*. An Adaptation Driver can control a Component's behavior through this interface.

Behavior:

A UML statechart diagram for the *Component Removal (125)* pattern is shown in figure 5.32. Specifically, this statechart models the possible behavior of the Driver class found in the Adaptation Driver component. First the object loads the reconfiguration plan that must be applied. At this step we assume that a component must

be removed from the application at run time, otherwise this reconfiguration pattern would not be applicable. Thus, the target Component is sent a *quiescent* command. Before the Component becomes *quiescent*, however, it must first complete any pending transactions and, if necessary, preserve its state. After the Component becomes *quiescent*, the Driver sends *passivate* commands to neighboring Components. After the Driver receives all pending acknowledgments that neighboring Components are *passive*, then the target Component can be *unlinked* from its neighbor Components by the Driver. The target Component can then be unloaded from the system. Lastly, *activate* commands are sent to the affected components so they may resume their normal behavior.

Consequences:

1. Components can be removed at run time without leaving the system in an inconsistent state.

2. Components can preserve state before they are removed from the system. This facilitates replacing components at run-time.

3. Removing a central component from the system may cause many other components to become passive until the reconfiguration process is complete. Since passive components cannot initiate new transactions, a significant processing delay may be incurred by the system until the reconfiguration process terminates.

4. Components must provide an interface to reach active, passive, and quiescent states. This interface can be either built-in during development or inserted through techniques such as AOP.

Constraints:

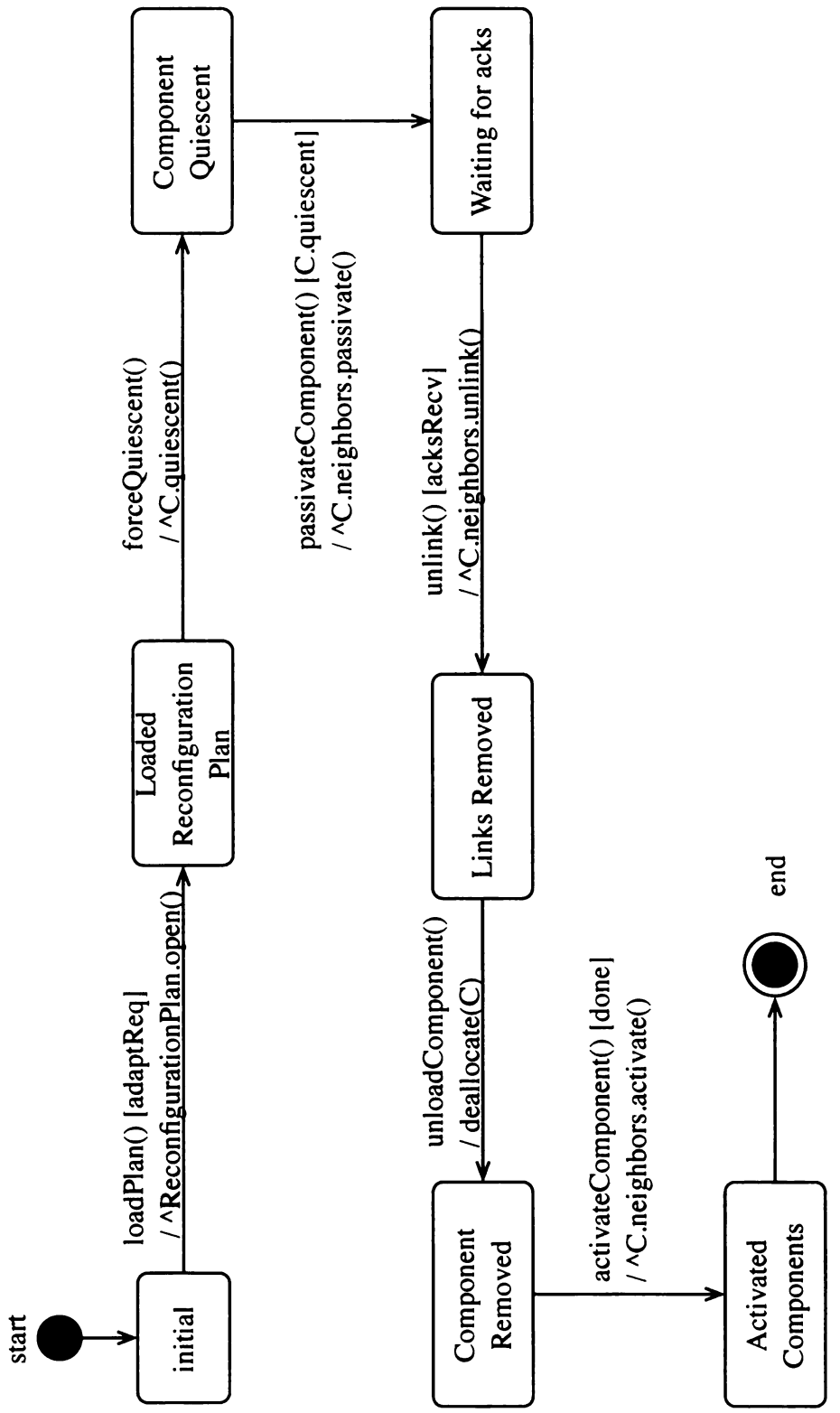

Figure 5.32: UML state diagram example of the *Component Removal (125)* Pattern

- **Property 1:**

  During adaptation, neighboring components that will be unlinked from x must first be guided to a passive state. To achieve this, a restricted condition, R_{Cond} must be established such that the system may safely reconfigure in bounded time.

$$R_{Cond} = \Box(\neg Component.neighbors.active \wedge \neg Component.unlink()).$$

  This restricted state, R_{Cond} prevents neighboring components from being activated by the AdaptationDriver until the component has been unlinked from them. Likewise, R_{Cond} also prevents x from being unlinked from active components. This restricted space can be enforced by preventing neighboring components from accepting new transaction requests until the reconfiguration is complete.

  The complete guided-adaptation property in A-LTL is given by:

$$((Component.neighbors! = null) \wedge (\diamond(adaptReq) \overset{true}{\rightharpoonup} R_{Cond}))$$

$$\overset{true}{\rightharpoonup} (Component.neighbors == null).$$

  This A-LTL property ensures that once a source program receives an adaptation request ($adaptReq$), it enters a restricted condition R_{Cond}. In this restricted source program, no component may be unlinked as long as any of its neighbors are active. Once neighboring components enter a passive state, links can be safely removed.

- **Property 2:**

  If a Component is not in a quiescent state, then it is never the case that the Component is unloaded from the system.

$\Box$ ($\neg$ `Component.isPassive()` $\rightarrow$ $\neg$ `unloadComponent(Component)`

This safety property guarantees that a component will not be removed from the system unless it is in a quiescent state.

Related Design Patterns:

- **Case-based Reasoning (68) Design Pattern:**

  If the reconfiguration plan selected by the *Case-based Reasoning (68)* pattern involves removing a component, this step can be performed by the *Component Removal (125)* Pattern.

- **Divide and Conquer (78) Design Pattern:**

  This pattern can be used to determine the specific sequence of steps required to safely perform a reconfiguration. Whenever a step requires that a component be removed, it can be carried out by the *Component Removal (125)* Pattern.

- **Architecture-Based (97) Design Pattern:**

  This pattern can be used to select a reconfiguration plan. Any reconfiguration that involves removing components from the system can use the *Component Removal (125)* Pattern.

- **TradeOff-Based (106) Design Pattern:**

  This pattern can be used to select a reconfiguration plan that best balances the overall set of objectives that various stakeholders may have. If the reconfiguration plan includes removing components, then the *Component Removal (125)* Pattern can be used.

- **Server Reconfiguration (135) Design Pattern:**

This pattern can be used to reconfigure an application structured as a server - client architecture. Components can be removed from the server architecture through the *Component Removal (125)* Pattern.

Known Uses:

- Znews.com - Rainbow framework [16].
- Software Reconfiguration Patterns [34].
- Evolving Philosophers - Dynamic Change Management [57].

5.5.11 *Server Reconfiguration (135)* Pattern

Classification:

Behavioral - Reconfiguration.

Intent:

Safely reconfigure a server - client component architecture at run time.

Context:

The *Server Reconfiguration (135)* Pattern may be used when:

- a server architecture needs to be reconfigured at run time.
- the application disallows downtimes.
- multiple client require services provided by the server component.

Motivation:

Client and server architectures are scalable in terms of the number and functions of clients that interact with the system [34]. Reconfiguring a server at run time is a difficult task for two specific reasons. First, while the presence or absence of a particular client does not affect the overall availability or behavior of the system, the presence of a server is crucial in the continued availability of the system. As a result, it is undesirable to reconfigure a server offline. Second, incoming client requests may continue to arrive while the server is being reconfigured. As a result, to ensure the reconfiguration process is transparent to clients, no requests may be lost during this period. The *Server Reconfiguration (135)* design pattern provides a behavioral template that describes how a server may be reconfigured at run time without losing client requests in the process.

Figure 5.33 shows a use-case diagram of the *Server Reconfiguration (135)* Pattern. The goal of this pattern is to safely reconfigure a server architecture at run time.

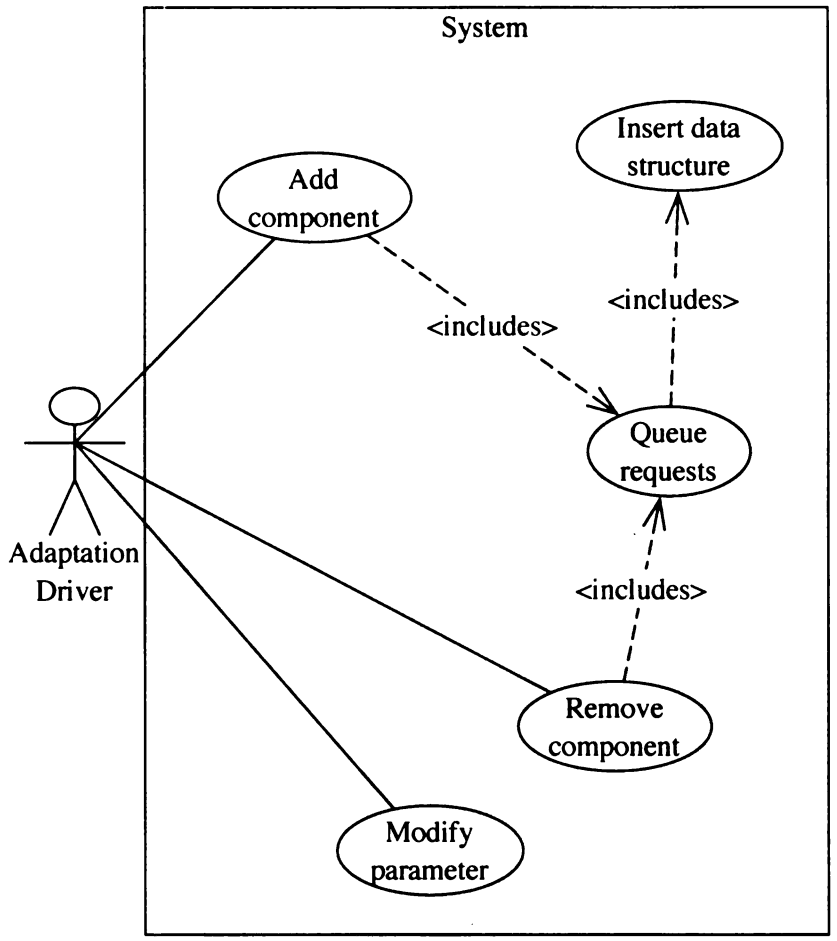

Figure 5.33: UML use-case diagram of the *Server Reconfiguration (135)* Pattern

Use-Case:	Add a component
Actors:	Adaptation Driver
Description:	Insert a component into the server architecture at run time.
Includes:	Queue requests.

Use-Case:	Queue requests
Actors:	-
Description:	Store incoming client requests so they may be serviced once the reconfiguration is complete.
Includes:	Insert data structure.

Use-Case:	Insert data structure
Actors:	-
Description:	Insert a data structure that will store incoming client requests so they may be processed after the reconfiguration process terminates.
Includes:	-

Use-Case:	Remove component
Actors:	Adaptation Driver
Description:	Unload a specific component from the server.
Includes:	Queue requests.

Use-Case:	Modify parameter
Actors:	Adaptation Driver
Description:	Adjusts a specific parameter that will fine-tune the server's behavior.
Includes:	-

Structure:

A UML component diagram for the *Server Reconfiguration (135)* Pattern can be found in Figure 5.34.

The *Server Reconfiguration (135)* design pattern coordinates the reconfiguration of a server architecture at run time. Every Component that is involved in the reconfiguration process must *realize* the States interface. The Adaptation Driver uses the States interface to guide a Component to its *active*, *passive*, and *quiescent* states. During the reconfiguration process, Clients may continue to submit requests to the Server component. For safety and operational reasons, however, incoming requests will be stored in a Request Buffer until the reconfiguration process terminates. The Adaptation Driver uses the Change Manager component to interact with the executing environment and effect the reconfiguration steps.

Participants:

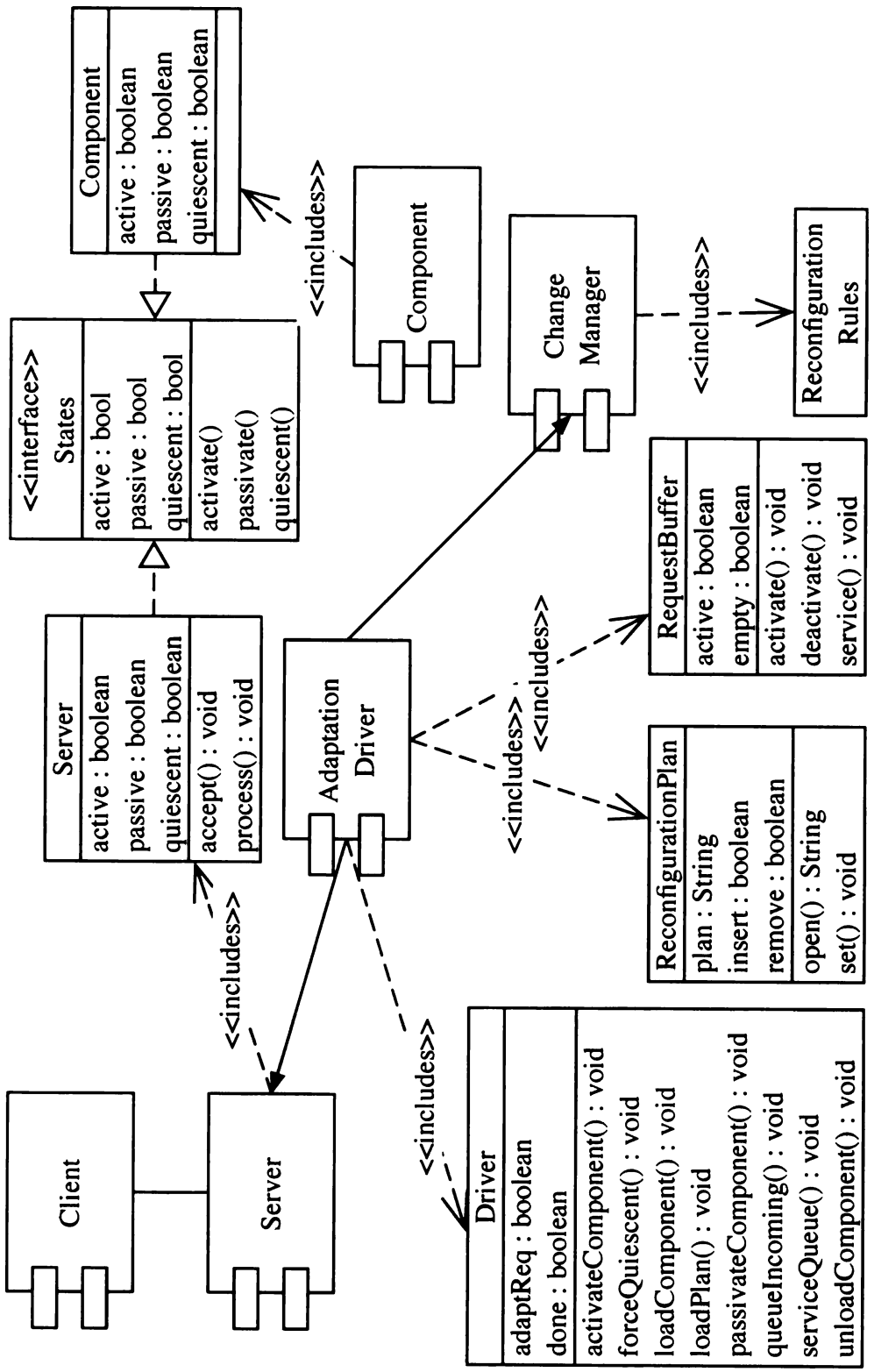

Figure 5.34: UML class diagram of the *Server Reconfiguration (135)* Pattern

- Adaptation Driver: Oversees the reconfiguration process for the Server. This component is responsible for ensuring that incoming Client requests are queued for further processing and that a Server completes unfinished transactions before the reconfiguration begins.

- Change Manager: Provides support for loading and unloading Components and their interconnections.

- Client: Represents any component that requires services provided by the Server.

- Component: Represents any executable component that can be deployed throughout the system. Each Component that may be involved in a reconfiguration must realize the States interface.

- Driver: Manages the operational states of components involved in a reconfiguration through the States interface. Specifically, it can guide a Component to *active*, *passive*, and *quiescent* states.

- Reconfiguration Plan: Stores the specific sequence of instructions for reconfiguring the system at run time.

- Reconfiguration Rules: Contains rules and instructions for specifying how basic reconfiguration operations are carried out in the system. Some basic reconfiguration operations include component insertion, removal, and swapping.

- Request Buffer: This is a data structure for storing Client requests while a reconfiguration takes place.

- Server: Represents a set of components that provides services to multiple Clients.

- States: This interface forces a Component to define which functional states correspond to active, passive, and quiescent states. An Adaptation Driver can control a Component's behavior through this interface.

Behavior:

A UML statechart diagram for the *Server Reconfiguration (135)* pattern is shown in figure 5.35. Specifically, this statechart models the possible behavior of the **Driver** class found in the **Adaptation Driver** component. First the **Driver** loads the reconfiguration plan that must be applied. Two possible reconfigurations are possible at this stage, either a component needs to be inserted or removed. Notice that to perform a swap operation we would first perform a removal followed by an insertion. Depending on whether a component needs to be inserted or removed, the **Driver** can perform either an insertion through the *Component Insertion (115)* pattern or a removal through the *Component Removal (125)* pattern. Two options are possible after this operation is complete. Either more structural changes need to be performed or the queued requests need to be serviced. If no more structural changes need to be performed, then the queued requests can be serviced until the **Request Buffer** is empty. At this point, the server reconfiguration is complete.

Consequences:

1. Pending transactions are completed before the reconfiguration process begins. This ensures the server is in a consistent state before it is reconfigured.

2. No incoming request is lost during the reconfiguration process.

3. The system can be transparently reconfigured at run time with respect to a client.

4. Latency can increase significantly during the reconfiguration process depending on the complexity of the reconfiguration. Specifically, numerous components may be driven to a *passive* state during the reconfiguration procedure. Until the reconfiguration is complete, these components will not be able to initiate new transactions with other components.

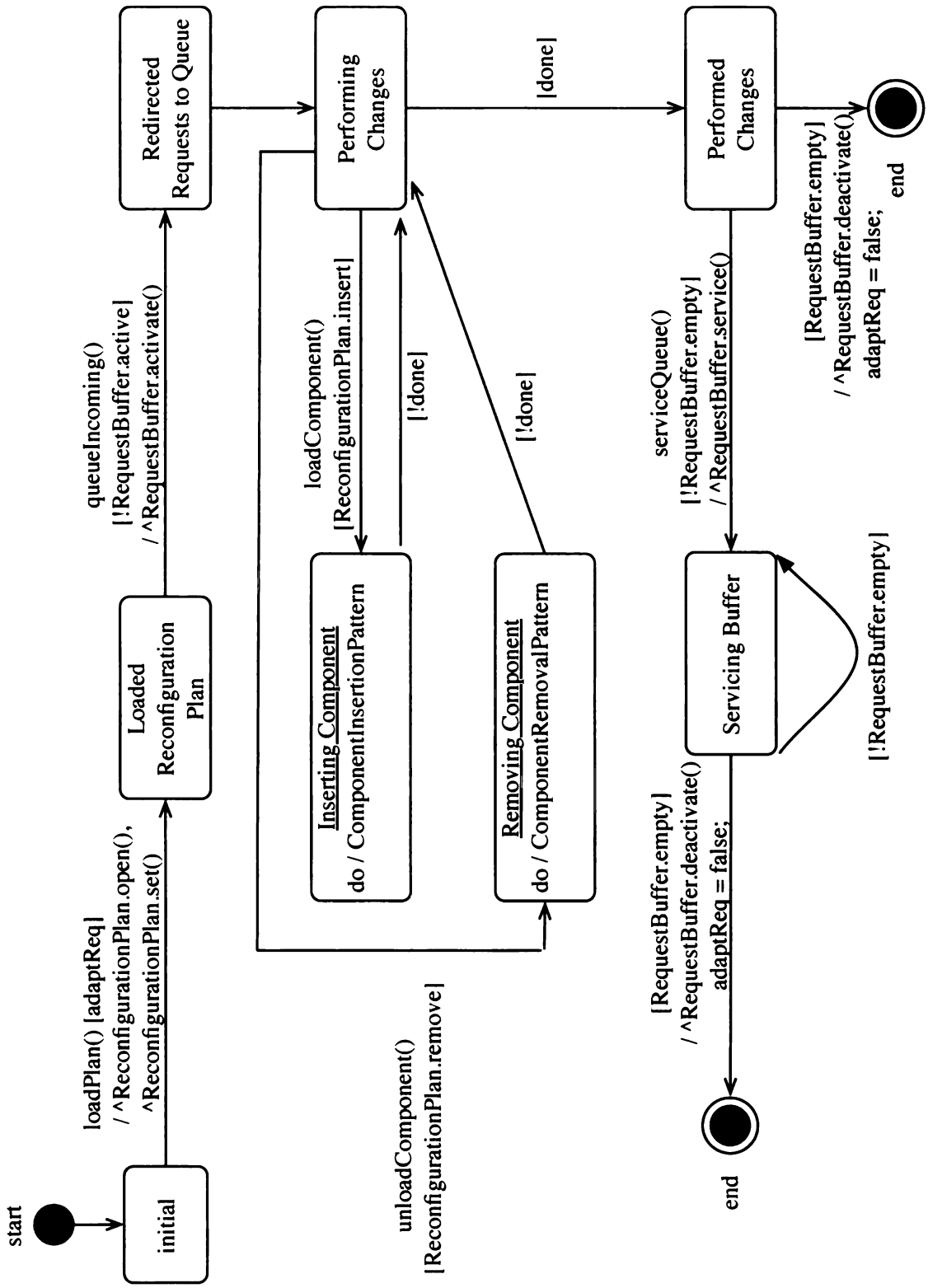

Figure 5.35: UML state diagram example of the *Server Reconfiguration (135)* Pattern

Constraints:

- **Property 1:**

  During adaptation, Clients will continue to submit transaction requests. If the Server keeps servicing incoming requests, then it may not reach a quiescent state in bounded time. To guide the Server to a quiescent state, the AdaptationDriver forwards all incoming Client requests to a RequestBuffer during the reconfiguration phase. Once the reconfiguration is complete, the Server may proceed to service all the Client requests until the message queue is empty. To achieve this, a restricted condition, R_{Cond} must be established such that the system may safely reconfigure in bounded time.

$$R_{Cond} = \Box(\neg \mathsf{Server}.accept() \wedge \mathsf{RequestBuffer}.active).$$

  This restricted state, R_{Cond} prevents a Server from accepting any new incoming connections during the reconfiguration phase. Likewise, R_{Cond} also ensures that the AdaptationDriver queues incoming requests for further processing. This restricted state can be enforced by redirecting all incoming connections to a message buffer. As a result, the Server will be able to complete any pending transactions that were already started before A_{Req} was received.

  The complete guided-adaptation property in A-LTL is given by:

$$((\mathsf{AdaptationDriver}.queueIncoming() \wedge (\diamond(adaptReq) \overset{true}{\rightharpoonup}$$

$$R_{Cond})) \overset{true}{\rightharpoonup} (\mathsf{RequestBuffer}.empty).$$

  This A-LTL property ensures that when an active server needs to be dynamically reconfigured, it will first be allowed to reach a passive state in bounded

time. Specifically, once an adaptation request, *adaptReq* is received, the AdaptationDriver blocks any incoming requests from being delivered to the server. Instead, incoming requests are stored in a RequestBuffer for further processing. A server will reach a quiescent state once it completes all pending transactions. At this point, the server can be safely reconfigured. Once the server becomes active again, it must first service all pending requests stored in the RequestBuffer. The reconfiguration process is complete once the RequestBuffer is empty.

- **Property 2:**

  Globally, it is always the case that if a message is stored in RequestBuffer, then it will be eventually serviced.

  $\Box$ ((RequestBuffer.queue(request)) $\rightarrow$

  $\Diamond$ (RequestBuffer.retrieve(request))

  This liveness property guarantees that if a message is queued, then it will eventually be retrieved and processed. This property ensures that messages are always processed at some point.

Related Design Patterns:

- ***Case-based Reasoning* Design Pattern:**

  This pattern can be used to select a reconfiguration plan to be performed by the Adaptation Driver in *Server Reconfiguration (135)*.

- ***Task Decomposition* Design Pattern:**

  This pattern can be used to determine the specific sequence of steps required to safely perform a reconfiguration. These steps can be carried out in *Server Reconfiguration (135)* if the application is structured as a server - client architecture.

- **Architecture-Based (97) Design Pattern:**

  The *Architecture-Based (97)* pattern can be used to determine the reconfigurations that need to be performed on a server architecture. The *Server Reconfiguration (135)* pattern can then perform these changes at run time.

- **TradeOff-Based (106) Design Pattern:**

  This pattern can be used to select a reconfiguration plan that best balances the overall set of objectives that various stakeholders may have. If the application is structured as a server, then the *Server Reconfiguration (135)* design pattern can perform the necessary reconfigurations.

- **Component Insertion (115) Design Pattern:**

  This pattern can be used to safely insert a new component at run time.

- **Component Removal (125) Design Pattern:**

  This pattern can be used to safely remove a component at run time.

Known Uses:

- Z.com - Rainbow Adaptation Framework [16].

- Server/Client Reconfiguration Pattern [34].

5.5.12 *Decentralized Reconfiguration (145)* Pattern

Classification:

Behavioral - Reconfiguration.

Intent:

Safely insert and remove components from a decentralized component architecture at run time.

Context:

The *Component Insertion (115)* Pattern may be used when:

- components need to be added and removed at run time.
- the application disallows downtimes.
- no single component coordinates the activities of the distributed components.

Motivation:

Decentralized applications function without any single component coordinating the activities of the distributed components [34]. The lack of a centralized coordinator implies that every component is responsible for the collective reconfiguration of the entire application. If components do not adhere to a reconfiguration protocol when they are inserted or removed from the system, then the entire distributed application may be left in an inconsistent state. The *Decentralized Reconfiguration (145)* design pattern provides a behavioral template that every component in the distributed application should follow to properly engage and disengage from other components during a reconfiguration.

Figure 5.36 shows a use-case diagram of the *Decentralized Reconfiguration (145)* Pattern. The goal of this design pattern is to guide the insertion and removal of components in a distributed application at run time.

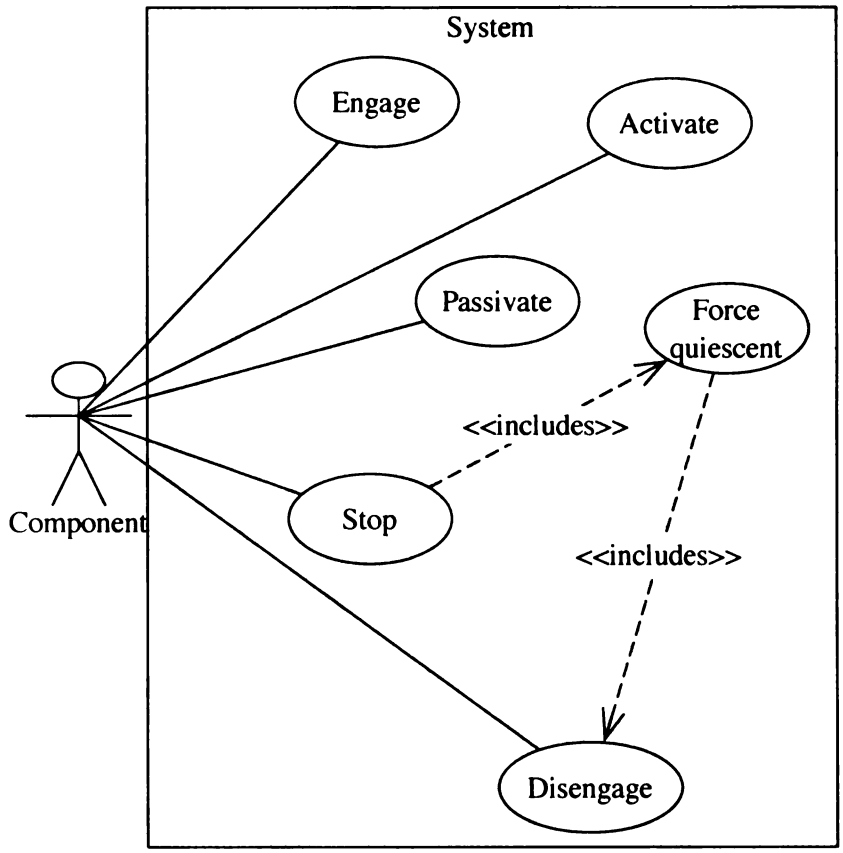

Figure 5.36: UML use-case diagram of the *Decentralized Reconfiguration (145)* Pattern.

Use-Case:	Engage
Actors:	Component
Description:	Initiate a transaction with another component.
Includes:	-

Use-Case:	Activate
Actors:	Component
Description:	Start requesting and servicing transactions with other components.
Includes:	-

Use-Case:	Passivate
Actors:	Component
Description:	Terminate pending transactions while not initiating new transactions.
Includes:	-

Use-Case:	Stop
Actors:	Component
Description:	Terminate the component.
Includes:	Force quiescent.

Use-Case:	Force quiescent
Actors:	-
Description:	Terminate all pending transactions and await for every neighboring component to acknowledge the component's departure from the system.
Includes:	Disengage.

Use-Case:	Disengage
Actors:	Component
Description:	Notify neighboring components of intention to go quiescent.
Includes:	-

Structure:

A UML component diagram for the *Decentralized Reconfiguration (145)* Pattern can be found in Figure 5.37.

The *Decentralized Reconfiguration (145)* design pattern provides a behavioral template for reconfiguring decentralized components at run time. Components are responsible for safely reconfiguring the overall application. Each Component must *realize* the States interface so they may autonomously cooperate with other Components during a reconfiguration. Components can notify each other of their intentions through the States interface. This enables a Component to properly engage and disengage other Components without leaving the system in an inconsistent state.

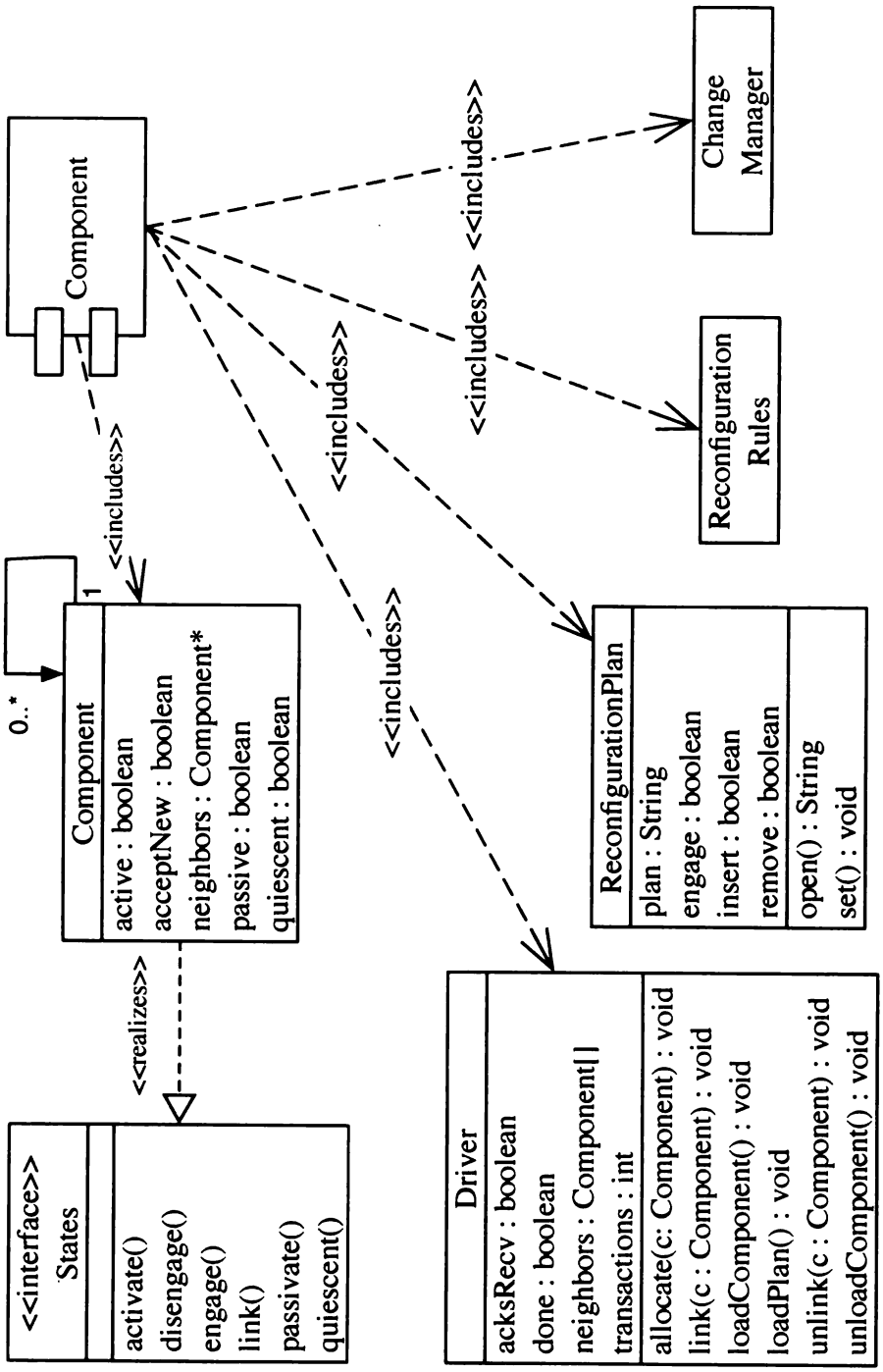

Figure 5.37: UML component diagram of the *Decentralized Reconfiguration (145)* Pattern

Participants:

- Change Manager: Provides support for loading and unloading Components and their interconnections.
- Component: Represents any executable component that can be deployed throughout the system. Each Component that may be involved in a reconfiguration must realize the States interface.
- Driver: Manages the operational states of components involved in a reconfiguration through the States interface. Specifically, it can guide a Component to *active*, *passive*, and *quiescent* states. Additionally, a Driver may issue engage and disengage commands to other Components to notify its intentions.
- Reconfiguration Plan: Stores the specific sequence of instructions for reconfiguring the system at run time.
- Reconfiguration Rules: Contains rules and instructions for specifying how basic reconfiguration operations are carried out in the system. Some basic reconfiguration operations include component insertion, removal, and swapping.
- States: This interface forces a Component to define which functional states correspond to active, passive, and quiescent states. In addition, it provides a negotiation protocol that specifies whether a Component wants to engage or disengage from other Components.

Behavior:

Explain 3 possible sequences. First, we can load another component and activate it. After that, each component is responsible for its own connections. Second, we can create a connection (link) between two components. Third, we can remove a connection (link) between two components. If there are no more links remaining and

the component needs to be removed (die), then the component unloads itself from the system. Otherwise it just remains in the system until a component requests a connection.

A UML statechart diagram for the *Decentralized Reconfiguration (145)* pattern is shown in Figure 5.38. Since there is no Component to organize the reconfiguration process, each Component is responsible for reconfiguring the entire application. Specifically, this statechart models the possible behavior of the Driver class found in a Component. First the Driver loads the reconfiguration plan that must be applied. Two possible reconfigurations are possible at this stage, either a component needs to be inserted or the Component determines it should unload itself from the system. If a Component must be inserted, then it is loaded into the system and initialized. However, every Component is responsible for engaging and disengaging other Components in the system on its own. If the Component determines it must remove itself from the system, then it initiates *disengage* transactions with its neighboring Components. After the Component receives acknowledgments from its neighboring Components, then it can unlink itself from them. Finally, with no pending transactions, a Component can unload itself from the system.

Consequences:

1. Components are responsible for the overall reconfiguration of the entire system. This facilitates the evolution of the system at run time.

2. Components must receive engagement acknowledgments from components it will interact with before it commences any transactions with them. This ensures that the insertion of a component does not leave the system in an inconsistent state.

3. Components must receive disengagement acknowledgments from neighboring

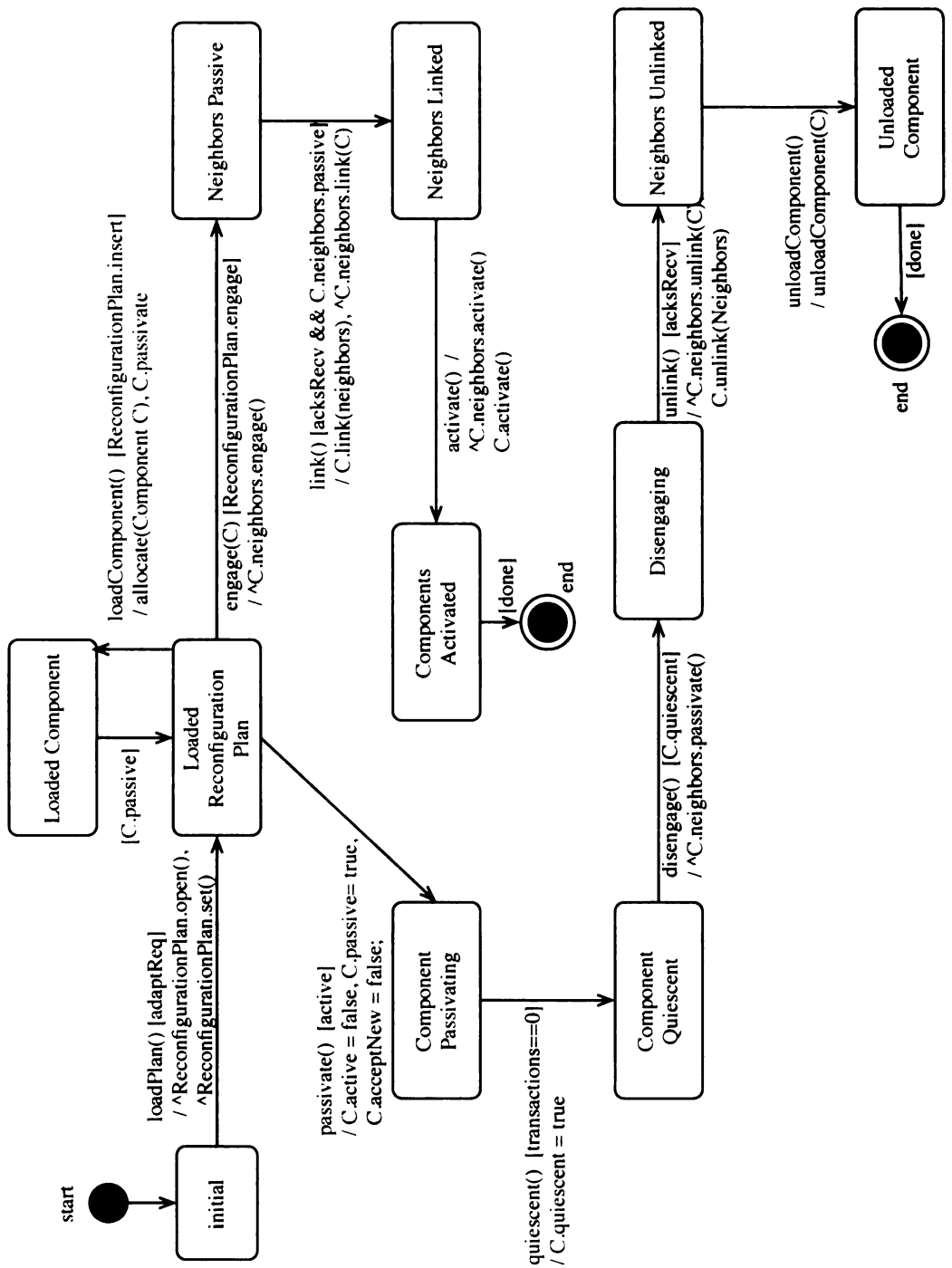

Figure 5.38: UML sequence diagram example of the *Decentralized Reconfiguration (145)* Pattern

components before it can be removed. This ensures the removal of a component does not leave the system in an inconsistent state.

4. Each component is responsible for reconfiguring the entire application. This complicates verifying and analyzing that a reconfiguration plan is correct.

5. Components must provide an interface to reach active, passive, and quiescent states. This interface must also provide a protocol to notify other components of engaging and disengaging intentions. This interface can be either built-in during development or inserted through techniques such as AOP.

Constraints:

- **Property 1:**

  During adaptation, neighboring components that will be linked with x must first be guided to a passive state. To achieve this, a restricted condition, R_{Cond} must be established such that the system may safely reconfigure in bounded time.

$$R_{Cond} = \Box(\neg Component.neighbors.active() \wedge \neg Component.link())$$

  This restricted state, R_{Cond} prevents neighboring components from becoming active until the new component has been linked to them. Likewise, R_{Cond} also prevents x from being linked to active components. This restricted space can be enforced by preventing neighboring components from accepting new transaction requests until the reconfiguration is complete.

  The complete guided-adaptation property in A-LTL is given by:

$$((Component.neighbors == null) \wedge (\diamond(adaptReq) \overset{true}{\rightharpoonup} R_{Cond})) \overset{true}{\rightharpoonup}$$

$$(Component.neighbors! = null).$$

This A-LTL property ensures that when a component enters the system, it does not establish any communication links with active components. Specifically, the system must first enter a restricted condition, R_{Cond} in which neighboring components will eventually reach a passive state. Links between the new component and its neighbors can be created once the component and its neighbors are in a passive state.

- **Property 2:**

  During adaptation, neighboring components that will be unlinked from x must first be driven to a passive state. To achieve this, a restricted condition, R_{Cond} must be established such that the system may safely reconfigure in bounded time.

$$R_{Cond} = \Box(\neg Component.neighbors.active \wedge \neg Component.unlink())$$

This restricted state, R_{Cond} prevents neighboring components from becoming active until the component has been unlinked from them. Likewise, R_{Cond} also prevents x from being unlinked from active components. This restricted space can be enforced by preventing neighboring components from accepting new transaction requests until the reconfiguration is complete.

The complete guided-adaptation property in A-LTL is given by:

$$((Component.neighbors! = null) \wedge (\diamond(adaptReq) \overset{true}{\rightsquigarrow} R_{Cond}))$$

$$\overset{true}{\rightsquigarrow} (Component.neighbors == null).$$

Where S_{Spec} are the specifications that must be satisfied in the source program before the adaptation request, A_{Req} is received and T_{Spec} is the specification that must be satisfied after the component has been removed.

- **Property 3:**

  If a Component is not in a quiescent state, then it is never the case that the Component is unloaded from the system.

  $\Box$ ($\neg$ `Component.quiescent` $\rightarrow$ $\neg$ `Driver.unloadComponent()` .

  This safety property guarantees that a component will not be removed from the system unless it is already in a quiescent state.

Related Design Patterns:

- **Case-based Reasoning (68) Design Pattern:**

  A component participating in the *Decentralized Reconfiguration (145)* pattern can use case-based reasoning mechanisms to determine which reconfigurations are necessary.

- **Divide and Conquer (78) Design Pattern:**

  A component participating in the *Decentralized Reconfiguration (145)* pattern can use the *Divide and Conquer (78)* pattern to determine the specific sequence of steps required to safely perform a reconfiguration.

- **Architecture-Based (97) Design Pattern:**

  A component participating in the *Decentralized Reconfiguration (145)* pattern can use architectural models to determine which reconfigurations are necessary.

- **TradeOff-Based (106) Design Pattern:**

A component participating in the *Decentralized Reconfiguration (145)* pattern can use the *TradeOff-Based (106)* pattern to select a reconfiguration plan that best balances the overall set of objectives that various stakeholders may have.

- **Component Insertion (115) Design Pattern:**

  The *Component Insertion (115)* design pattern can be used to safely insert a component into the system at run time.

- **Component Removal (125) Design Pattern:**

  The *Component Removal (125)* design pattern can be used to safely remove a component from the system at run time.

Known Uses:

- Unity - IBM Autonomic System [17].
- Software Reconfiguration Patterns [34].
- Evolving Philosophers - Dynamic Change Management [57].

Chapter 6

Process

This chapter presents two ideas to facilitate the modeling and formal analysis of adaptive systems. First, we provide more details on how to accomplish several of the modeling steps briefly mentioned in the model-based development process [85]. In particular, we explain how the design patterns in this thesis can be leveraged by the model-based development process when building adaptive systems. Second, we illustrate how UML state diagrams can be used to model the behavioral aspects of the adaptive logic. Specifically, we introduce an iterative process that leverages automated formal analysis tools and techniques for analyzing the models against certain safety properties.

6.1 Model-based Development Process

It is increasingly important to be able to adapt an application's behavior at run time in response to changing requirements and environmental conditions. As a result of their high complexity, adaptive programs are generally difficult to specify, design, verify, and validate [85]. In order to leverage the benefits of model-driven engineering, including code generation, it is advantageous to address adaptation early in the development process, starting with requirements, progressing to design and then

eventually to code in a systematic fashion. Previously, Zhang and Cheng introduced a model-based development process to model and analyze adaptive systems [85]. In particular, the process focused on the assurance of the adaptations at each step of the development process. This thesis extends that work to focus on the creation of the design models that can be used as part of the design phase of the overall development process.

We assume steps (1) through (4) of the model-based development process have been completed (see Figure 6.1). Requirements R_i have been specified for a given domain D_i that satisfy the overall system goal G. These requirements are operationalized in the form of design models M_i. Furthermore, non-adaptive models M_i and M_j have been created and verified against their respective local (Φ_i and Φ_j) and global global properties (INV), specified in terms of formal specification languages such as linear temporal logic. Nonetheless, at this stage, the resulting models lack the infrastructure required for self-adaptation. Each of these models must be instrumented with monitoring and decision-making capabilities in order to automate the tasks of introspection and intercession. To provide the required monitoring and decision-making functionality, step (4) of the model-based development process has been extended to incorporate the design patterns presented in this thesis.

Developers must analyze the specific domains, requirements, and constraints of the non-adaptive models M_i and M_j to determine which monitoring and decision-making design patterns are suitable. Once the domain and its requirements are understood, developers can browse the available design patterns for reusable solutions that address their needs. A design pattern's Context field is particularly helpful at this stage as it identifies under what circumstances should the pattern be applied. Once an initial set of design patterns have been identified, developers should consult the Related Patterns field to review other design patterns commonly used together. This helps identify alternative approaches that were previously unforeseen during the

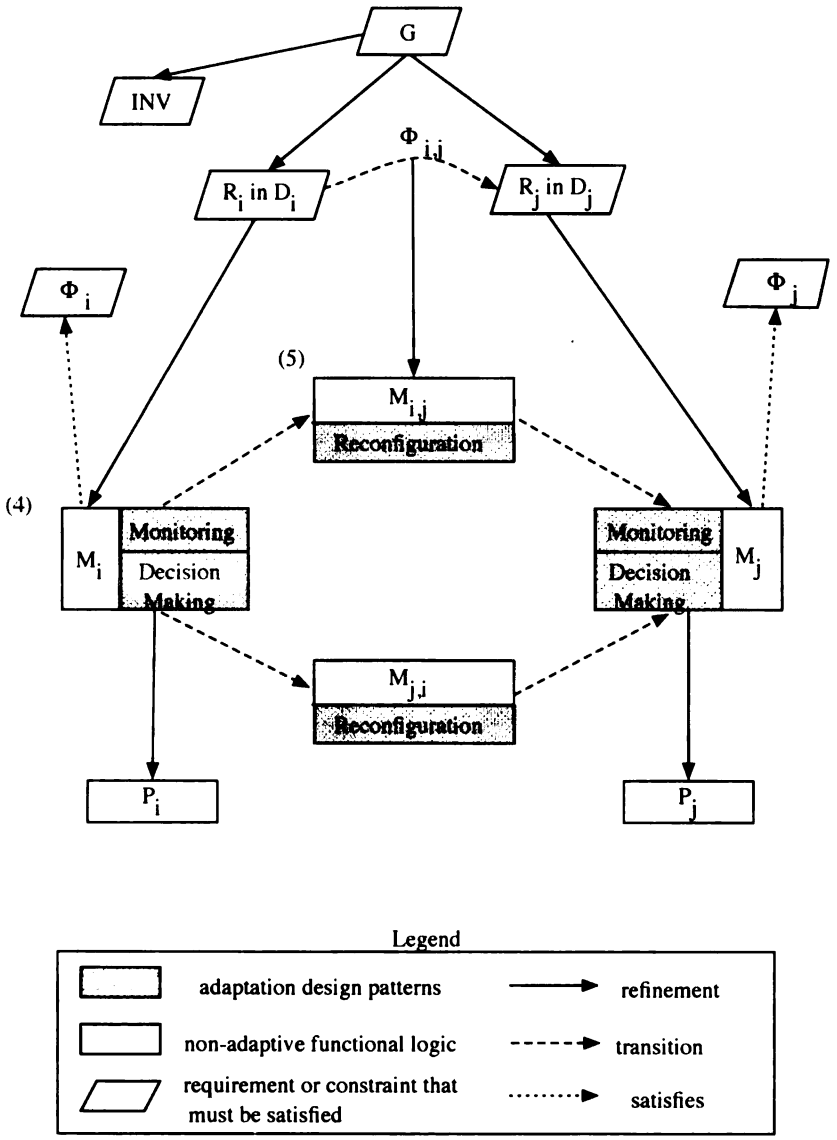

Figure 6.1: Adaptation Design Patterns Within Model-Based Development Process.

design phase. If various design patterns are applicable, then developers should analyze the Consequences field to determine the tradeoffs incurred by applying the particular pattern. After a set of design patterns is selected, the patterns can be instantiated, integrated with the non-adaptive models M_i and M_j and formally analyzed against safety critical properties.

Although the non-adaptive models M_i and M_j are now instrumented with monitoring and decision-making capabilities, these models are still incapable of reconfiguration. Step (5) of the model-based development process focuses on the creation of models to represent the adaptive logic (e.g., $M_{i,j}$ and $M_{j,i}$) and verifies them for correctness. The reconfiguration design patterns presented in this thesis can be incorporated at step (5) to guide the development of the adaptive models. These reconfiguration design patterns should be selected according to the reconfiguration scenarios that are possible as well as the overall architecture of the self-adaptive system. Analyzing a system's architecture helps determine the different interactions between components as well as any dependencies that might exist between them. This is important as it guides developers in identifying the quiescent states in M_i, the starting states in M_j, and the integration of these states with the adaptive model $M_{i,j}$.

6.2 Modeling Adaptive Logic

To fully leverage model-driven development technology for adaptive systems, such as systematic refinement of abstract models to more concrete models, it is necessary to model both the structural and behavioral portions of an adaptive system. While several approaches focus on the modeling and analysis of the architectural (structural) dimension of an adaptive system [16, 29], it is also important to model and analyze the behavioral portion of an adaptive application. Model checking has

been previously used to verify properties of adaptive software [3, 13, 56] and correct errors before the system is implemented and deployed. By analyzing the design models against functional and adaptation properties, we minimize the potential to propagate errors from the design to the implementation and maintenance phases. Despite the formal analysis capabilities, the combination of the functional logic intertwined with the adaptive logic makes the maintenance and analysis of the overall adaptive system models challenging.

Previously, we presented an iterative approach to constructing and analyzing UML design models for adaptive systems, where we separate the modeling of the functional logic from the adaptive logic [68]. For the purposes of this thesis, we consider an adaptive system P to comprise n *steady-state* programs, where a *steady-state* program is a state machine that is non-adaptive. We consider adaptation to be the transition from one *steady-state* source program to another target program. Thus, while it is possible for P to transfer execution between any of the n programs, only one *steady-state* program may be executing at any given time. In general, our approach decomposes large adaptive programs by first separating the *steady-state* program verification from the adaptive logic verification. Our approach verifies three key types of properties. Local properties are those that must hold within a particular domain. Global properties or invariants are those that must hold at all times. Once each *steady-state* program is verified against their local and global properties, they do not need to be verified again if involved in a different adaptation scenario. Thus, by separating concerns at the model level, the complexity of verifying the model correctness of adaptive programs is decreased.

Our iterative approach is as follows (see Figure 6.2):

1. Create UML statecharts for objects that are involved between target model M_j and source model M_i.

2. Use Hydra [63] to automatically translate the resulting collection of statecharts

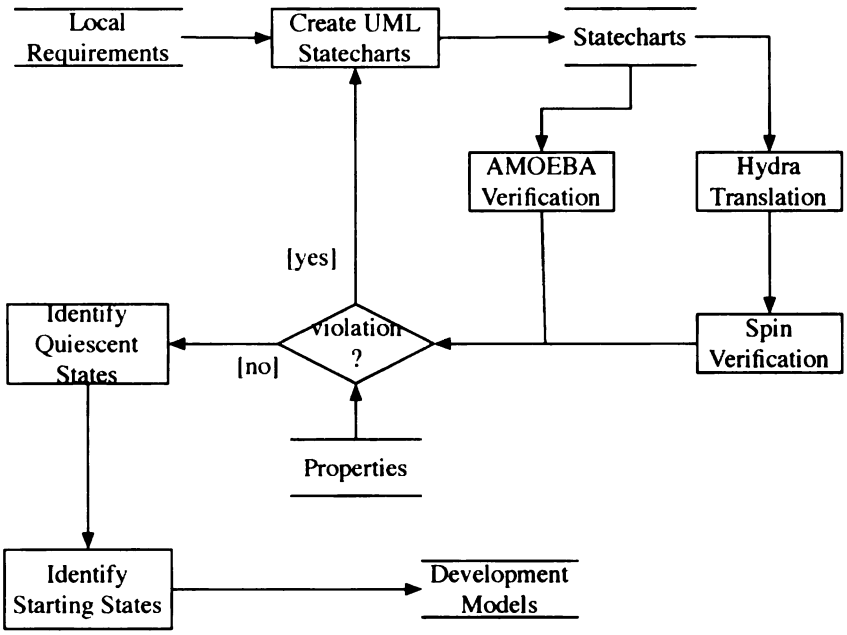

Figure 6.2: Iterative Process for Modeling Adaptive Logic.

that constitute the adaptive logic into Promela code.

3. Analyze the adaptive model for adherence to the global and transitional properties using SPIN [47], and the adaptive propreties through the AMOEBA model checker [86]. If any global proprety is violated, return to (1).

4. Identify quiescent states in source model M_i and add transitions from these states to adaptive model $M_{i,j}$.

5. Identify starting states in target model M_j and add incoming transitions to these states from adaptive model $M_{i,j}$.

In order to complete step (1), developers should refer to the source and target models. Examining the structural differences between these models will help identify which objects were affected by the adaptive logic. Unfortunately, just examining the structural differences will not identify all the objects involved in the adaptive logic. Some objects exist only within the scope of the adaptive logic and will not show up in

either set of diagrams. For instance, a temporary buffer in the adaptive infrastructure that holds data while the system initializes the target steady-state program will not show up in either source or target models since it is not present in either of them. These objects, however, will become apparent as the adaptive logic is modeled since they bridge the gap between the source and target models.

Steps (2) and (3) of this iterative modeling process rely on automated tools for formally verifying programs. Specifically, step (2) uses Hydra, a tool based on a UML formalization framework developed by McUmber and Cheng that automatically translates UML state diagrams into Promela code [63]. Step (3) uses the SPIN model checker [47]. SPIN takes Promela code as input and can be run in either simulation or verification mode. Simulation mode is helpful for visualizing sample executions of the system. Nonetheless, verification mode is required to state any claims about satisfying or violating specific properties. If a property is not satisfied, SPIN produces a trace output showing where the error occurred. One of the main advantages of using Hydra and SPIN together is that enables automated formal analysis of development models. That is, the same models that are used to guide development are used in the verification and analysis process.

Step (5) involves identifying several application-specific states. Several researchers [34, 57, 46] advocate that a component is quiescent when it has completed all pending transactions and is not engaged in communications with any other component. Currently, developers manually identify which states conform to those requirements in an application. As a result, the identification of quiescent states is application-specific. Likewise, when a component is loaded at run time it might need to be initialized to a particular configuration.

Chapter 7

Case Study

This chapter presents a "proof of concept" case study that applies several adaptation design patterns in the construction of a self-adaptive application. First, we describe the application domain. We then list the specific functional and adaptation requirements that must be satisfied by the resulting application. Next, we present our design and implementation. Finally, we present a comparison between a pattern-oriented approach and a framework-oriented approach at constructing an adaptive system.

7.1 Application Description

The Z.com case study was originally described in [16]. Z.com is a fictional news site that is planning to use adaptation to address the "slashdotting effect" where news sites when listed on slashdot (or brought to the wider public's attention through some other means) are unable to deal with the larger number of requests for content and either suffer from high latency or else are unable to serve content altogether. For instance, on "Black Friday 2006", the Wal-Mart website was inaccessible into late afternoon costing the store millions and the day before Amazon.com traffic was problematic due to the demand for the Xbox 360 [16].

Garlan *et al.* modeled the Z.com system as a set of clients and servers with the overall constraint that latency must fall within a given threshold. Nonetheless, adaptation concerns for this application are multi-faceted. Some of the utility concerns that must be balanced at run time include cost, latency, and fidelity. Cost is incurred by the company whenever it runs a server. Latency measures the amount of time it takes for a server to provide the requested content to a client. Fidelity represents the content format, either graphical or textual. The purpose of the adaptive Z.com system is to optimally balance the costs incurred by running servers while providing quality content at a low latency.

This case study uses the adaptation design patterns presented in this thesis to re-engineer the Z.com adaptive web server previously presented in [16]. Although our adaptive system is implemented using a notably different approach, it exhibits similar behavior to the one Cheng *et al.* created using the Rainbow framework. Specifically, our version of the Z.com application provides the same reconfiguration capabilities as Z.com provided. Having two implementations of the same adaptive system enables us to perform a more comprehensive comparison of the key differences between the approaches.

7.2 Requirements

Since this case study is aimed at replicating Rainbow's Z.com adaptive news server [16], their same functional and adaptive requirements apply to our case study. Specifically, there are conflicting requirements between operational costs and quality of service constraints. The following requirements were identified by Garlan *et al.* for Z.com:

- The news server will provide basic HTML functionality to requesting clients.

- The operational cost may not be exceeded at any time. Specifically, Z.com has a maximum monetary budget for providing its services. Being a company, it will attempt to maximize its profits by not exceeding its allocated budget.

- The quality of the content should be the best one possible. Specifically, whenever possible, service client's requests in graphical content mode.

- The system will avoid losing customers due to a high response time if it can somehow provide faster content. Specifically, if the server's average response time is too high, the content may be switched to textual mode in order to reduce transmitting large file sizes.

Even though Rainbow uses a utility-based approach for selecting which reconfiguration to apply depending on different stakeholder's needs, it must first quantify conditions that warrant adaptations. The Z.com application requirements and its implementation define several macros to represent these operational boundaries at a high level of abstraction. For instance, cost is expressed as being within budget or exceeding budget. Likewise, the three possible values for latency include low, middle, and high. Each macro is replaced during compilation with the specific numerical values that developers consider to represent different operational levels. This facilitates the task of evolving and fine-tuning the adaptation requirements as needed.

Given the three objectives of minimizing operational costs and latency and providing graphical news content whenever possible, Garlan *et al.* reasoned about the possible adaptation scenarios that might arise for Z.com. For instance, Z.com will increment its server pool by one integral amount if the response time is high and the budget will not be exceeded. Otherwise, Z.com will switch to textual content mode if it is not already in that mode. Additionally, when the response time is low, Z.com will decrement its server pool size by one integral amount if it is near budget limit. If the response time is low, then the servers will be switched to graphical mode if they

are not already in that mode. Lastly, when the response time is in the medium range, Z.com will switch to graphical mode if the mode is textual, while the server pool size may either be incremented to decrease response time or decremented to reduce cost.

As an example, consider the following reconfiguration scenario for Z.com. Under normal conditions, Z.com is hosted in one active server. Breaking-news are issued at Z.com's homepage and many clients start connecting to the server and requesting the HTML web page. Gradually, the latency increases to a high level. Two possible reconfigurations are available. First, Z.com could add another server and split the load from incoming users while maintaining graphical content delivery. Second, Z.com could switch to textual delivery mode without adding any servers. Since the current budget at Z.com allows at least two servers to be executing without exceeding the allocated budget, the first reconfiguration plan is selected. This provides the best user experience without exceeding cost limitations. The reconfiguration plan is applied, a new server is initialized and the latency drops back to a low level.

7.3 Application Design

We re-engineered the Z.com application in three major stages. First, we modeled and implemented the functional logic according to the functional requirements identified in [16]. Next, we identified a set of monitoring and decision-making design patterns that were applicable to our version of Z.com and we proceeded to instantiate them. To ensure our design satisfied certain properties, we analyzed the resulting models against local properties and invariants before we implemented them. Lastly, we modeled and implemented the adaptive logic responsible for reconfiguring the server architecture.

7.3.1 Non-Adaptive Design

Our Z.com application is modeled after a multi-threaded server-client architecture (see Figure 7.1). Specifically, our design comprises a Gateway, a Gateway Thread, a Load Balancer, a Server, a Server Thread, and a Stats class. The Gateway, Load Balancer, and Gateway Thread classes are responsible for taking incoming HTML requests from different web browsers, determining which available Server currently has the lowest average response time, and redirecting the web browser to that Server, respectively. The Server and Server Thread classes are responsible for servicing common HTML requests such as retrieving a file across a network. The Stats class follows the *Singleton* design pattern [26] and enables the Load Balancer to keep track of usage rates for each Server.

We implemented the design models in the JAVA programming language. Any web browser capable of displaying HTML content can readily connect to the application and retrieve web content. This interface also facilitated the use of a simple script to test parts of the application. The script performs a basic load test by repeatedly requesting a web page at various intervals. For instance, running the load test on a single server with no adaptation capabilities quickly showed an average latency of over 500 miliseconds. This simple test facilitated the analysis of our Z.com application under different configurations and usage rates.

We selected a set of monitoring and decision-making design patterns based on the context of the Z.com application. We chose to apply the *Sensor-Factory (41)* design pattern to periodically monitor the average latency for two reasons. First, a distributed monitoring scheme is required for Z.com's networked architecture. Second, our functional logic already provides an interface to the attributes that need to be monitored at run time. Although this might seem impractical, the same approach was followed in [16]. Garlan *et al.* noted that system administrators would have

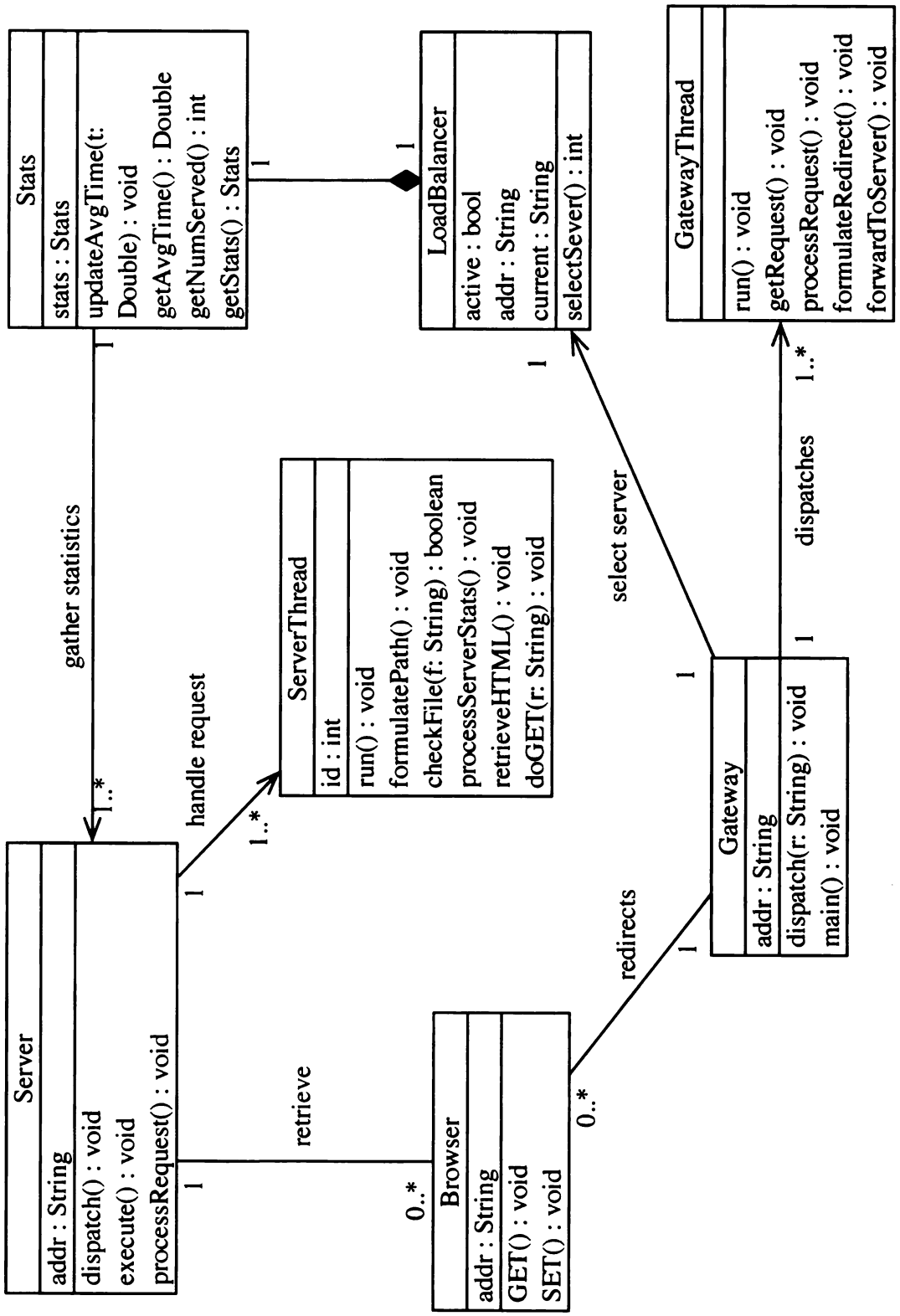

Figure 7.1: UML class diagram example of the Z.com functional logic

access to monitoring information from the application's interface. Since the objective of Z.com is to automate tasks performed by a system administrator, it makes sense to provide an interface to these attributes that can be remotely probed by a sensor.

The instantiated version of the *Sensor-Factory (41)* pattern is shown in Figure 7.2. Our Z.com application uses Simple-Sensors to probe active Servers for their average latency. Given the nature of our application, the instantiated *Sensor-Factory (41)* pattern does not include Complex-Sensors. In addition, since our Simple-Sensors *realize* the Abstract Sensor interface, there is no need for an Adapter pattern [26] to facilitate the communication between a Client and a Simple-Sensor with incompatible interfaces.

We used Hydra [63] to automatically convert several state-based models of Resource Manager, Sensor-Factory, and Simple-Sensor into Promela code. A constraint violation was found when we attempted to verify the Promela models in the SPIN model checker [47]. The violated property stated that if a Resource Manager denies a sensor request, then Sensor-Factory would not deploy that sensor. However, an earlier version of the *Sensor-Factory (41)* allowed the existence of multiple Resource Managers across the system. Although this did not seem problematic at first sight, it enabled the following scenario: "the same sensor request is granted by instance x of Resource Manager and denied by instance y of Resource Manager." Since our focus is not on distributed consistency, we resolved the problem by permitting only one Resource Manager in the system. Notice that if *distributed* resource management techniques are applied, then this constraint can be lifted.

Two decision-making design patterns were applied to Z.com, *Adaptation Detector (88)* and *Case-based Reasoning (68)*. The *Adaptation Detector (88)* pattern was selected to interpret the monitoring data supplied by the *Sensor-Factory (41)* pattern and detect when a reconfiguration was required. Figure 7.3 shows the instantiated

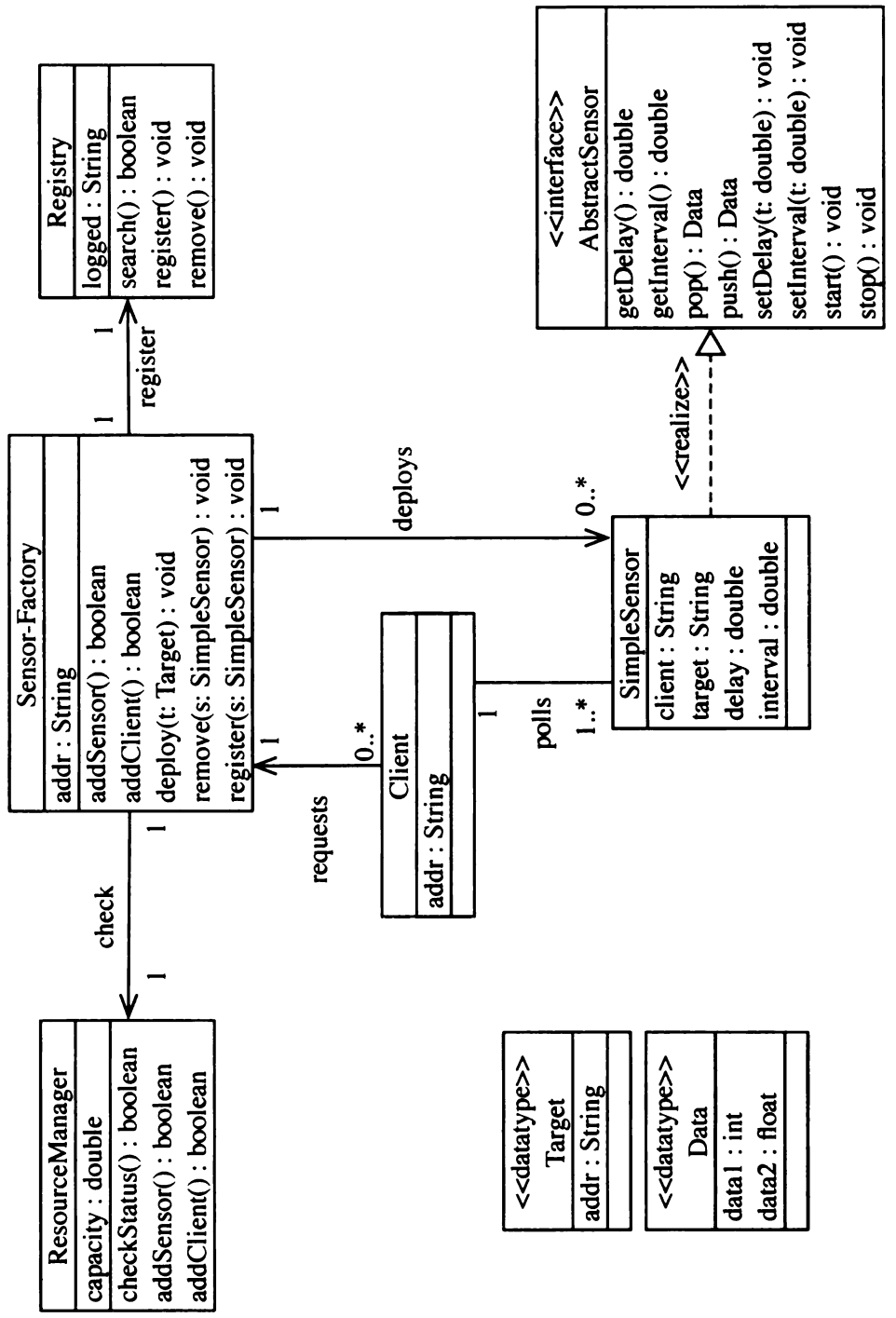

Figure 7.2: UML class diagram of the *Sensor-Factory (41)* Pattern applied to Z.com

version of the *Adaptation Detector (88)* design pattern. The key participants in this design pattern are **Observer** and **Analyzer**. A single **Observer** interacts with a **Simple-Sensor** (not shown) to obtain the monitored attributes. The **Observer** interprets the data by casting it to whichever type it expects to receive from the **Simple-Sensor** data feed. This data is then compared against three specific value boundaries by the **Analyzer**. If the boundaries are exceeded, then a **Trigger** is created and sent to the **Inference Engine** of the *Case-based Reasoning (68)* pattern.

As with the *Sensor-Factory (41)* pattern, we used Hydra to automatically convert UML state-based models of **Health Indicator**, **Analyzer**, and **Observer** into Promela code. We then used the SPIN model checker to analyze the two properties previously specified for the *Adaptation Detector (88)* design pattern. Specifically, we wanted to ensure that if **Observer** received any monitoring value, then it would eventually be compared against the **Threshold**. Likewise, we wanted to ensure that if a **Threshold** was exceeded, then a **Trigger** would be created. The SPIN model checker did not find any violations for these two properties.

The *Case-based Reasoning (68)* pattern was selected to determine which reconfiguration plan should be applied based on the monitoring information available. Garlan *et al.* used a utility-based approach for selecting reconfiguration plans in their version of Z.com [16]. Nonetheless, the reconfiguration scenarios for this application are simple enough that they can be efficiently captured in a set of "if-then" rules. Figure 7.4 shows the instantiated version of the *Case-based Reasoning (68)* design pattern. Though the instantiated version is similar to the *Case-based Reasoning (68)* pattern, there are slight differences between the two. Specifically, the case-based reasoning decision-making process for Z.com does not support any learning capabilities. As a result, Z.com will only support reconfiguration scenarios that are known ahead of time.

Figure 7.3: UML class diagram of the *Adaptation Detector (88)* Pattern applied to Z.com

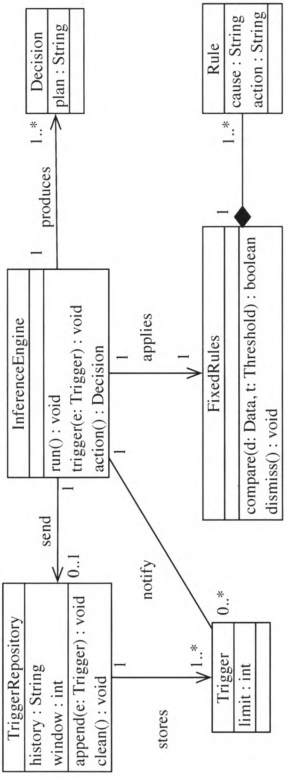

Figure 7.4: UML class diagram of the *Case-based Reasoning (68)* Pattern applied to Z.com

We used Hydra to automatically convert UML state-based models of **Inference Engine** and **Fixed Rules** into Promela code. These Promela models were then analyzed with the SPIN model checker against two properties. First, we want to ensure that if a Trigger is received, then it is always the case that a Decision is produced. Our application design includes a default rule and decision pair to enforce this constraint in the event that no other pair matches at run time. Second, we want to ensure that if a decision is produced, then it is always the case that it is logged. Logging the reconfiguration decision and why it was selected will help developers understand how their system is behaving at run time. The SPIN model checker did not find any instances of property violations for either of these two properties.

7.3.2 Adaptive Design

At this stage we have augmented the Z.com application with monitoring and decision-making capabilities. Specifically, we can periodically observe the average latency, detect when a reconfiguration is required, and select a reconfiguration plan that will yield the desired behavior. Figure 7.5 shows the resulting class diagram of integrating the monitoring and decision-making processes with the functional logic of the Z.com application. This class diagram also includes an **Adaptation Driver** that oversees the specific reconfiguration steps of the entire system. The **Inference Engine** from the *Case-based Reasoning (68)* pattern notifies the **Adaptation Driver** that a specific reconfiguration plan needs to be applied. The **Adaptation Driver** then performs reconfigurations drawn from the behavioral templates found in the *Component Insertion (115)*, *Component Removal (125)*, and *Server Reconfiguration (135)* patterns.

Previously, we identified four possible reconfigurations for our version of the Z.com adaptive news server. Two of those reconfigurations involve tuning parameters to alternate between content delivery mode. The two other reconfigurations

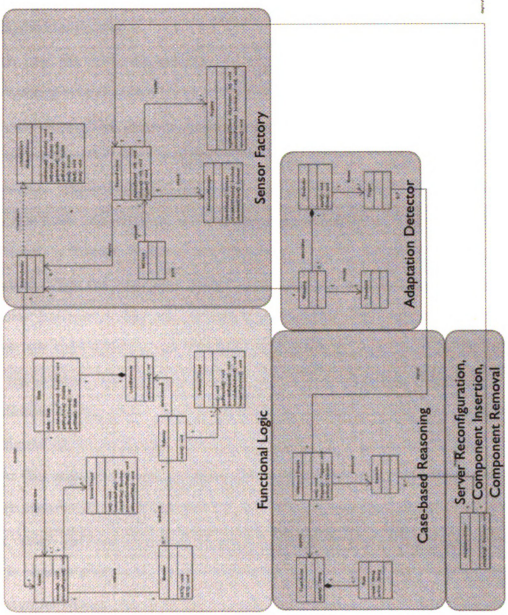

Figure 7.5: UML class diagram of the Z.com application

involve either adding or removing a Server at run time. In contrast to the parameter reconfiguration, we must first prepare the system before a server is added or removed. Specifically, incoming client requests need to be queued so they may be processed after the reconfiguration is complete. Otherwise, client requests may be lost during the reconfiguration process.

We used the *Server Reconfiguration (135)* pattern to safely reconfigure the Z.com application in scenarios that involved the addition or removal of a server. The *Server Reconfiguration (135)* pattern, in turn, reuses the *Component Insertion (115)* and *Component Removal (125)* reconfiguration design patterns to safely add and remove components, respectively. For instance, to add a server at run time, the Adaptation Driver first loads and initializes a new Server and Load Balancer. The Adaptation Driver then inserts a Request Buffer to store incoming requests during the reconfiguration procedure. Then the Adaptation Driver sends passivate commands to both the Servers and Load Balancer so they can be safely reconfigured. Once these components are *passive*, the Load Balancer can be driven to a *quiescent* state so it can be removed from the system. Notifications are then sent by the Adaptation Driver to *activate* the affected components. Finally, once all the queued requests are serviced, the reconfiguration is complete and the system continues to operate as normal.

At this stage, the Z.com application comprised instantiations of one monitoring pattern, two decision-making patterns, and three reconfiguration patterns. This set of six design patterns rendered our Z.com application with self-adaptive behavior. That is, sensors periodically probed the servers for the average latency, and whenever a substantial change was detected, an adaptation request was issued. The decision-making determined which reconfiguration plan to apply based on the monitoring information. The reconfiguration plan was then applied by the Adaptation Driver and either switched between content delivery modes or added or removed servers at run time. When compared to the Rainbow version of Z.com, our application provides

similar functionality and reconfiguration capabilities.

7.4 Results

Re-engineering the Z.com application originally presented by Garlan *et al.* in [16] enables us to compare the advantages and disadvantages of both approaches. Our design pattern approach has several advantages over a framework-oriented approach at developing dynamically adaptive systems. For instance, design patterns impose few initial constraints on the system being developed. As a result, patterns provide a flexible approach that can be readily customized to specific systems. Frameworks, on the other hand, incorporate many design decisions already made by the framework developers. Likewise, design patterns do not entail a steep learning curve in order to apply them successfully. To properly use a framework, however, a developer must understand the underlying framework mechanisms and how they relate to the application being built. Additionally, instantiated versions of the design patterns can be analyzed through formal verification tools and techniques to ensure a design satisfies certain key properties before it is implemented. Attempting to verify the correctness of a framework is, at best, impractical. Lastly, with our design pattern approach, developers select only those adaptation mechanisms their application will require. As a result, adaptive applications built with our adaptation patterns contain only those features it needs, rather than including many features that may not be necessary.

Framework-oriented approaches, on the other hand, have several major advantages over our design patterns for building dynamically adaptive systems. For instance, adaptation-enabling frameworks provide large amounts of code that can be directly reused when building adaptive applications. If the application and the framework share the same context and domain, then a large section of development overhead can be avoided by reusing the framework's code. Design patterns, however, offer

no code at all. After the system is designed and the patterns are instantiated, they must be implemented by developers. Likewise, adaptation-enabling frameworks, such as Rainbow, tend to support a wide range of adaptation approaches and techniques. Each desired functionality must be carefully integrated and implemented into the application with a pattern-oriented approach. Lastly, adaptation-enabling frameworks hide the internals of dealing with specific reconfiguration scenarios from developers. In contrast, design patterns must be instantiated and customized for the particular application being developed. As a result, developers have to deal with the details and complexities of reconfiguring applications at run time.

Chapter 8

Conclusions

Our work with adaptation design patterns has yielded three main contributions: an adaptation design pattern template that assists developers in understanding and designing adaptive systems; a set of twelve adaptation-focused design patterns to promote the reuse of successful design decisions; and extensions to the model-based development process introduced by Zhang and Cheng [85] that incorporate the use of adaptation design patterns and state-based modeling techniques. We describe these contributions in more detail below.

First, we developed an adaptation-focused design pattern template for the development of adaptive systems. We extended the pattern template used by Gamma *et al.* for describing design patterns [26] with the *Behavioral* and *Constraints* fields. Developers can use the *Behavioral* field to analyze the interactions between different objects in the design pattern. Likewise, developers can use the *Constraints* field to ensure their instantiated design pattern satisfies the specified properties. In addition, we modified the *Related Pattern* field such that it indicates which other adaptation design patterns are commonly used together when building an adaptive system. The information provided in the template enables developers to understand the consequences and trade-offs incurred by applying a pattern. Furthermore, the use of a

design pattern template enforces the uniform organization of every adaptation design pattern, thus facilitating their use.

Second, we introduced twelve design patterns to support monitoring, decision-making, and reconfiguration of adaptive systems where the patterns facilitate the separate development of the functional logic and the adaptive logic. Each design pattern was harvested from at least two successful design solutions and generalized so that they may be applied across different adaptive domains. To assess their maturity, we validated each design pattern against instances in adaptive systems that were not used in the harvesting process. In addition, we successfully applied a subset of the patterns in the development of an adaptive news web server. This example helped illustrate how various design patterns could be combined to construct self-adaptive and autonomic computing systems.

Finally, we extended the model-based development process previously introduced by Zhang and Cheng [85] in two key ways. First, we developed concrete guidelines for using set of adaptation design patterns to realize the design modeling steps of their process. Specifically, we incorporated the monitoring and decision-making patterns into the creation of the non-adaptive models and the reconfiguration patterns into the creation of the adaptive models. Second, we incorporated the use of UML state-based models to represent the adaptive logic within the model-based development process. Using UML state-based models to represent the adaptive logic facilitates the visual inspection and formal verification of the instantiated design pattern models against specified properties through the use of automated tools such as Hydra [63] and the SPIN model checker [47]. This verification step enables developers to ensure a design satisfies specific constraints before the implementation phase.

Several directions for future work are possible. First, additional design patterns for adaptation could be identified and integrated with the set of design patterns presented in this thesis. Second, we could examine how these design patterns can be

inserted into a non-adaptive application through the use of aspect-oriented techniques [38, 82]. Third, we could explore the use of digital evolution techniques to automatically identify the points of an application to insert the monitoring patterns. Lastly, we could explore the use of digital evolution techniques to determine how adaptation design patterns can be evolved and instantiated to satisfy new sets of properties [33].

APPENDICES

Appendix A

Sample Instantiations

This chapter presents sample original implementations that were used to harvest the twelve adaptation-oriented design patterns presented in this thesis. In particular, for each adaptation design pattern, at least two sample original implementation sources are presented and compared against the resulting design pattern. It is important to note, however, that while there may be some similarities between each design pattern and its corresponding original implementations, there will not be a one-to-one correlation between them. Specifically, each design pattern provides a generalized solution based on these original implementations (as well as others). Each original implementation is presented through UML class diagrams, component diagrams, object diagrams, or state diagrams. Furthermore, for clarity, each model presented in this section has been elided from its original version to include only those elements relevant to the design pattern being considered.

A.1 *Sensor-Factory (41)* Pattern

Rainbow Adaptation Framework [16, 27, 30]. The Rainbow adaptation framework provides a reusable infrastructure for probing components, determining when an adaptation is warranted, and then effecting the necessary changes. Figure A.1 shows an elided UML class diagram of the probing mechanisms employed in Rainbow. Various similarities can be observed between Rainbow's probing infrastructure and the *Sensor-Factory (41)* design pattern. Specifically, both define explicit interfaces to which RegularPatternGauges (probes, sensors) must adhere. This interface facilitates the creation of various types of interchangeable probes across the system. The *Sensor-Factory (41)* pattern follows a similar approach by defining an interface to which every Sensor must adhere. Moreover, in the Rainbow probing framework, each RegularpatternGauge is associated with some GaugeInstanceDescription, which defines the type and configuration of the gauge. Likewise, both Rainbow and the *Sensor-Factory (41)* pattern employ objects to create, deploy, manage, and remove probes. The GaugeCoordinator is responsible for creating, configuring and deleting instances of RegularPatternGauges. Notice that this functionality is provided by both SensorFactory and Registry in *Sensor-Factory (41)*. Nonetheless, there is no explicit use of a ResourceManager to oversee the allocation and deallocation of probes in the Rainbow framework.

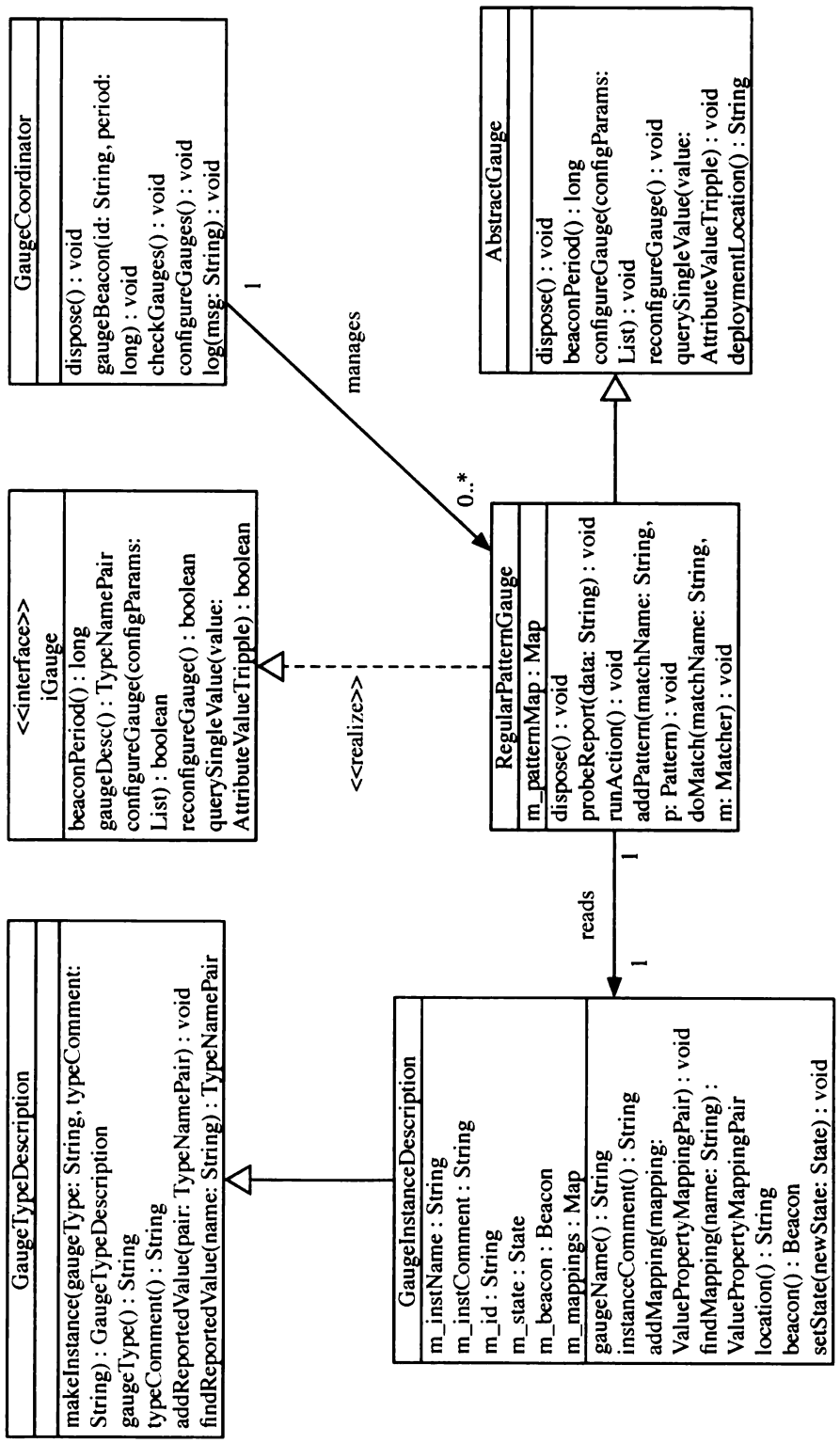

Figure A.1: UML class diagram of the Rainbow Adaptation Framework [16].

SNMP4J-Agent [24]. SNMP4J [24] is an enterprise class (open-source, commercial) SNMP implementation for the JAVA language. In particular, the SNMP4J-Agent is capable of periodically polling various SNMP-conforming entities across the network. Figure A.2 shows a UML object diagram of the SNMP4J-Agent [24], which is similar in structure and behavior to the *Sensor-Factory (41)* pattern. In SNMP4J, a user enters commands through a CommandProcessor, which then submits each request to the RequestFactory. Likewise, various Clients in the *Sensor-Factory (41)* pattern submit their monitoring requests to the Sensor-Factory. Once the Request-Factory receives a request in SNMP4J, both a SnmpRequest and a RequestStatus are created. These two constructs facilitate the tracking of requests as they get processed through the system. Once a SnmpRequest is granted, then the RequestHandler creates a Snmpv2MIB, which is an interface through which the user can obtain the monitoring information he desires. Likewise, in the *Sensor-Factory (41)* pattern, when a request for a Sensor is granted by the ResourceManager, then the Sensor-Factory returns the requested object to the Client.

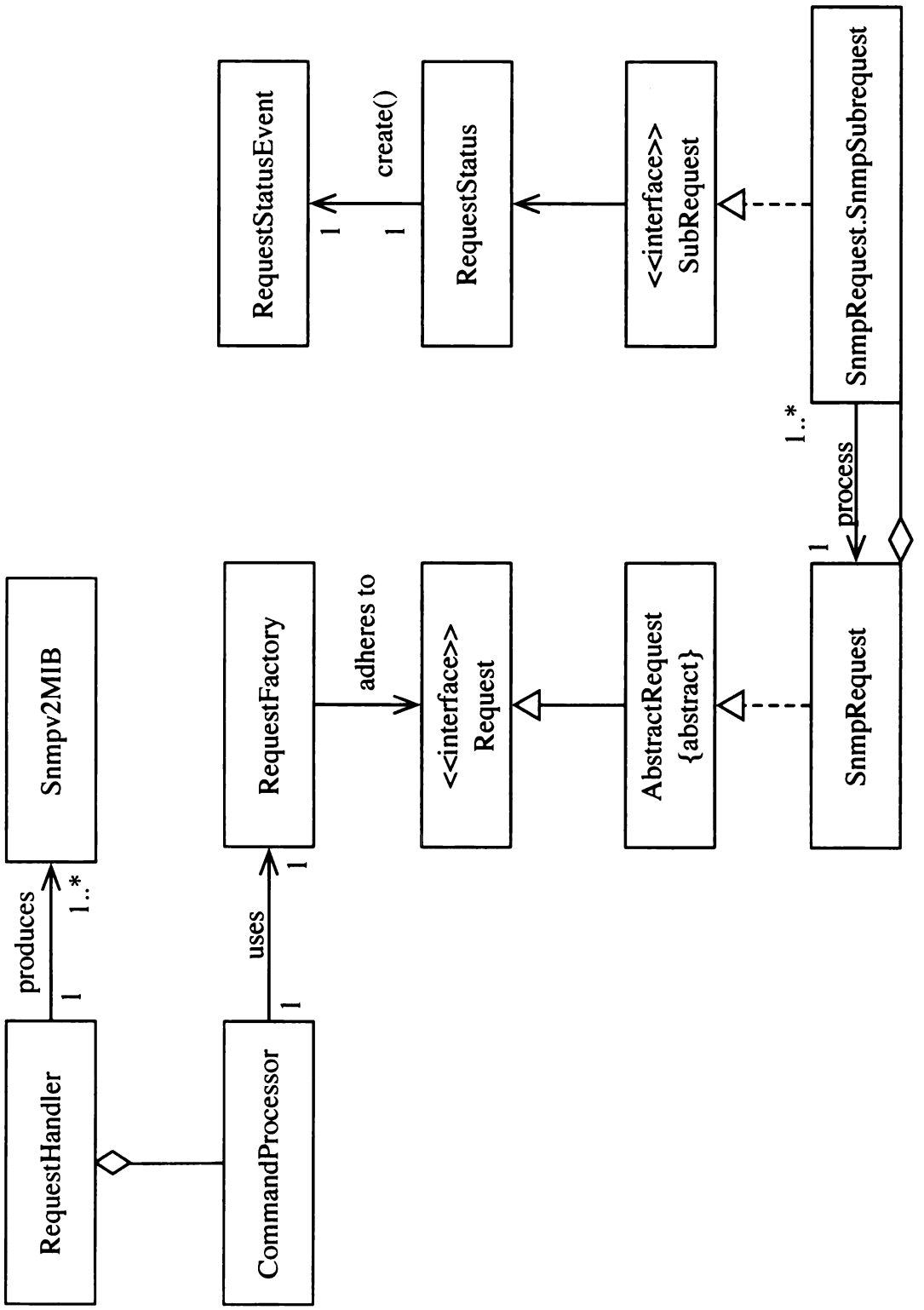

Figure A.2: UML object diagram of the SNMP4J-Agent [24].

A.2 *Reflective Monitoring (50)* Pattern

Adaptive Exception Monitor [19]. The Adaptive Exception Monitor implemented in [19] is based on the JAVA reflection package [25]. This exception monitor exploits the intercessional processing capabilities of the JAVA language and runtime system to transparently inspect each exception generated by pre-specified objects. Each exception is logged into a knowledge-based of the autonomic element for further processing and analysis. Figure A.3 shows an elided UML class diagram of the Adaptive Exception Monitor [19]. Strong similarities can be observed between the Adaptive Exception Monitor and the *Reflective Monitoring (50)* design pattern. In particular, both make explicit use of MetaObjects, Proxies, and InvocationHandlers. MetaObjects provide information about a particular type of object such as its structure, attributes, modifiers, and interfaces. Proxies make use of the information provided by the MetaObject to provide a functional and transparent wrapper around some specific object. The InvocationHandler is then able to intercede whenever a targeted method of a Proxy is invoked. Specifically, application-specific data is logged before and after these methods are invoked by the Proxy. Furthermore, InvocationHandlers can link various monitoring Proxies together to construct more complex forms of monitoring. One notable difference between this sample instantiation and the *Reflective Monitoring (50)* pattern is that in the Adaptive Exception Monitor, the equivalent of Manager is implemented by Controller. Although similar in responsibilities, the Controller is decentralized in the sense that each InvocationHandler has its own Controller. Thus, instead of there being one centralized Manager that tracks the various monitoring Proxies, there are several of them interspersed throughout the implementation.

Figure A.3: UML class diagram of the Adaptive Exception Monitor [19].

Reflective Monitoring of Real-Time Systems [5]. Monitoring mechanisms in real-time systems must not introduce a significant overhead since the process of monitoring is already computationally expensive. One approach to monitor the execution of a real-time system is to exploit reflective programming through the use of meta-level objects. Figure A.4 shows a reflective monitoring approach for monitoring real-time systems [5], which is similar in structure and behavior to the *Reflective Monitoring (50)* pattern. Specifically, both approaches separate base-level objects from meta-level objects, which contain information about base-level objects. In the reflective monitoring approach presented in [5], every object that will be monitored, such as **Task**, must realize a **MetaProgram** interface. It is through this interface that other objects are able to probe for the desired information. This approach is similar to the one found in the *Reflective Monitoring (50)* pattern, in which **Metaobjects** provide an interface to base-level objects, such as **Target**, so they may be probed by other objects. Note that there is no explicit use of a **Proxy** in this approach since every monitored object directly implements the **MetaProgram** class. Nonetheless, the monitoring procedure works similarly. Whenever a predetermined event occurs, such as a method call, the **Monitor** collects the necessary information from the **MetaProgram**. This process is paralleled by the **InvocationHandler** in the *Reflective Monitoring (50)* pattern.

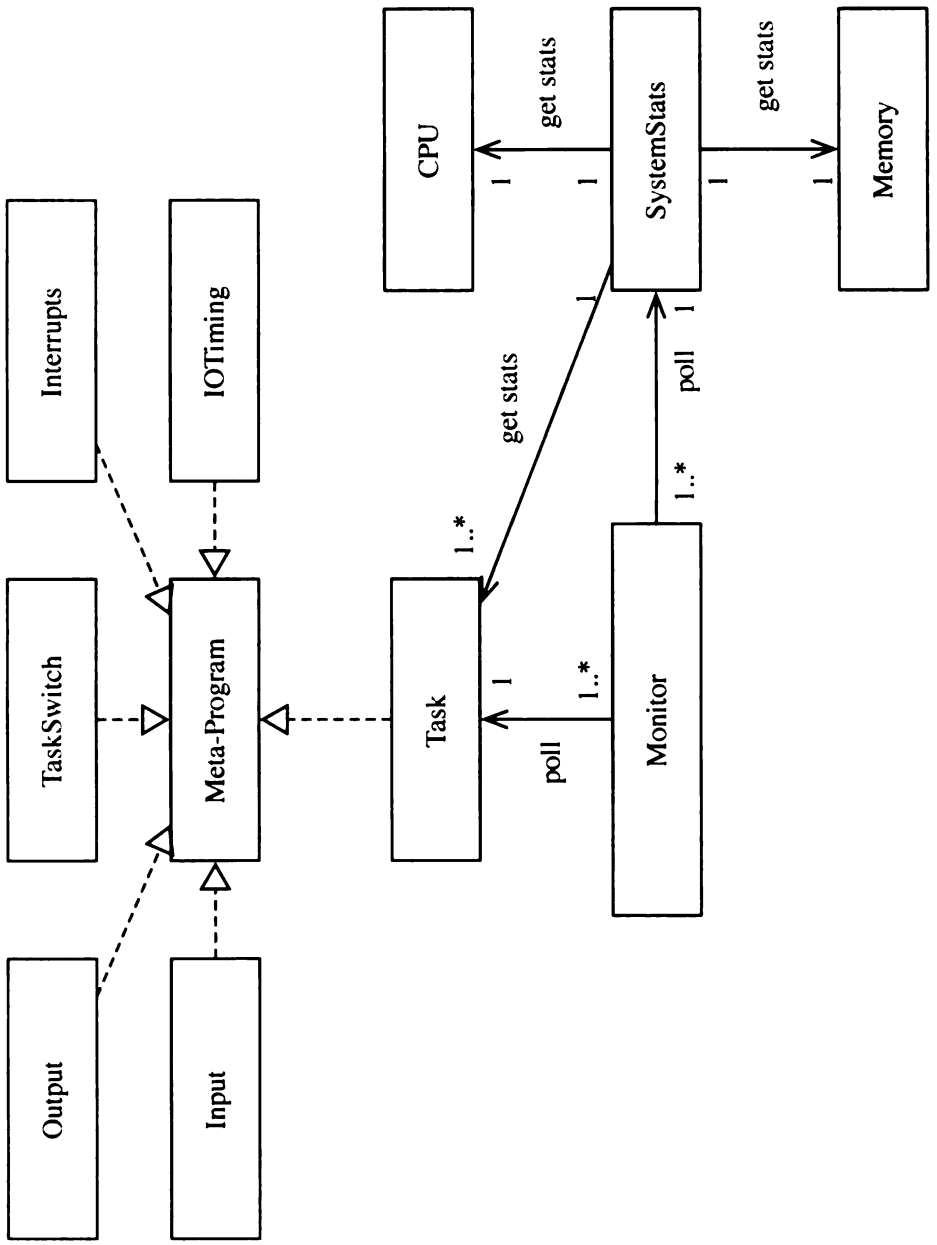

Figure A.4: UML object diagram for Reflection-based Monitoring of Real-Time Systems [5].

A.3 *Content-based Routing (59)* Pattern

Rebeca Notification Infrastructure [83]. Rebeca is a content-based publisher/subscriber service that has been extended to support mobile computing [83]. Their middleware-based approach enables existing applications to be adapted from a static to a mobile scenario without having to adapt client applications. Figure A.5 shows a UML class diagram of Rebeca as it relates to the *Content-based Routing (59)* pattern. Figure A.5 shows a simplified UML class diagram of Rebeca as it relates to the *Content-based Routing (59)* pattern. The specific UML class diagram was obtained from Rebeca's documentation. Although Rebeca is a sophisticated content-based publisher/subscriber implementation, strong similarities can be seen with respect to the *Content-based Routing (59)* pattern. In Rebeca, an EventBroker acts as an access point to the publish/subscribe system. It is defined as an *interface* so that different EventBrokers can be swapped in or out as necessary. In terms of the *Content-based Routing (59)* pattern, the EventBroker maps to the Manager class. Although Manager is not defined as an interface, it provides the same functionality as EventBroker. Rebeca also defines a Filter and two subclasses Subscription and Advertisement. These correspond to Filter and Pattern in *Content-based Routing (59)*, respectively. Specifically, an Advertisement is used to identify the origin of the data being published to the EventBroker. A Filter, on the other hand, is used to select a specific event in the EventBroker. Finally, notice that *Case-based Reasoning (68)*'s Server and Forwarding Table provide similar functionality as EventTransport and RoutingTable, respectively, in Rebeca.

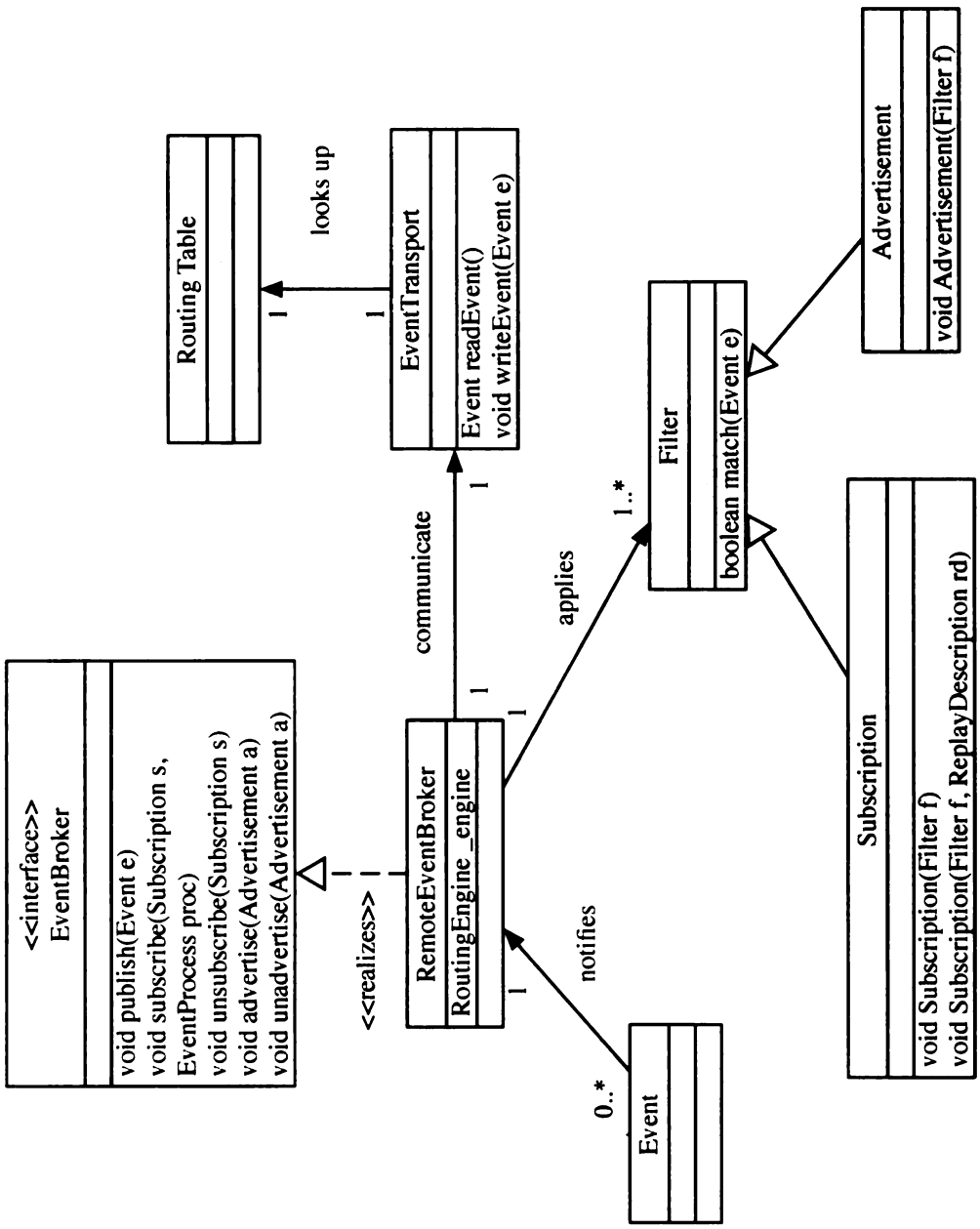

Figure A.5: UML class diagram of the Rebeca Notification Infrastructure [83].

Siena Routing Infrastructure. The second *Sample Instantiation* is taken from the Siena Routing Infrastructure [14, 43]. Siena is a wide-area event notification service based on a publisher / subscriber architecture. Siena routes messages across a network based on their contents. Specifically, clients send a Subscription to the Siena service. A Packet Receiver stores the incoming Subscription. Although the *Content-based Routing (59)* does not have a Packet Receiver, Clients still submit their Subscriptions to the EventService. After some event occurs, a Notification is sent to the Siena service. Since the *Content-based Routing (59)* pattern is oriented towards monitoring processes, Notifications are explicitly sent from Sensors. This restriction, however, is not evident in Siena. Both *Content-based Routing (59)* and Siena use a series of Patterns and Filters to map notifications to Clients. Specifically, each update is characterized by a Pattern that identifies the update source. The Event Service then applies a Filter to the update to determine which Clients should be notified. The PacketSender then proceeds to send a Notification to those Clients.

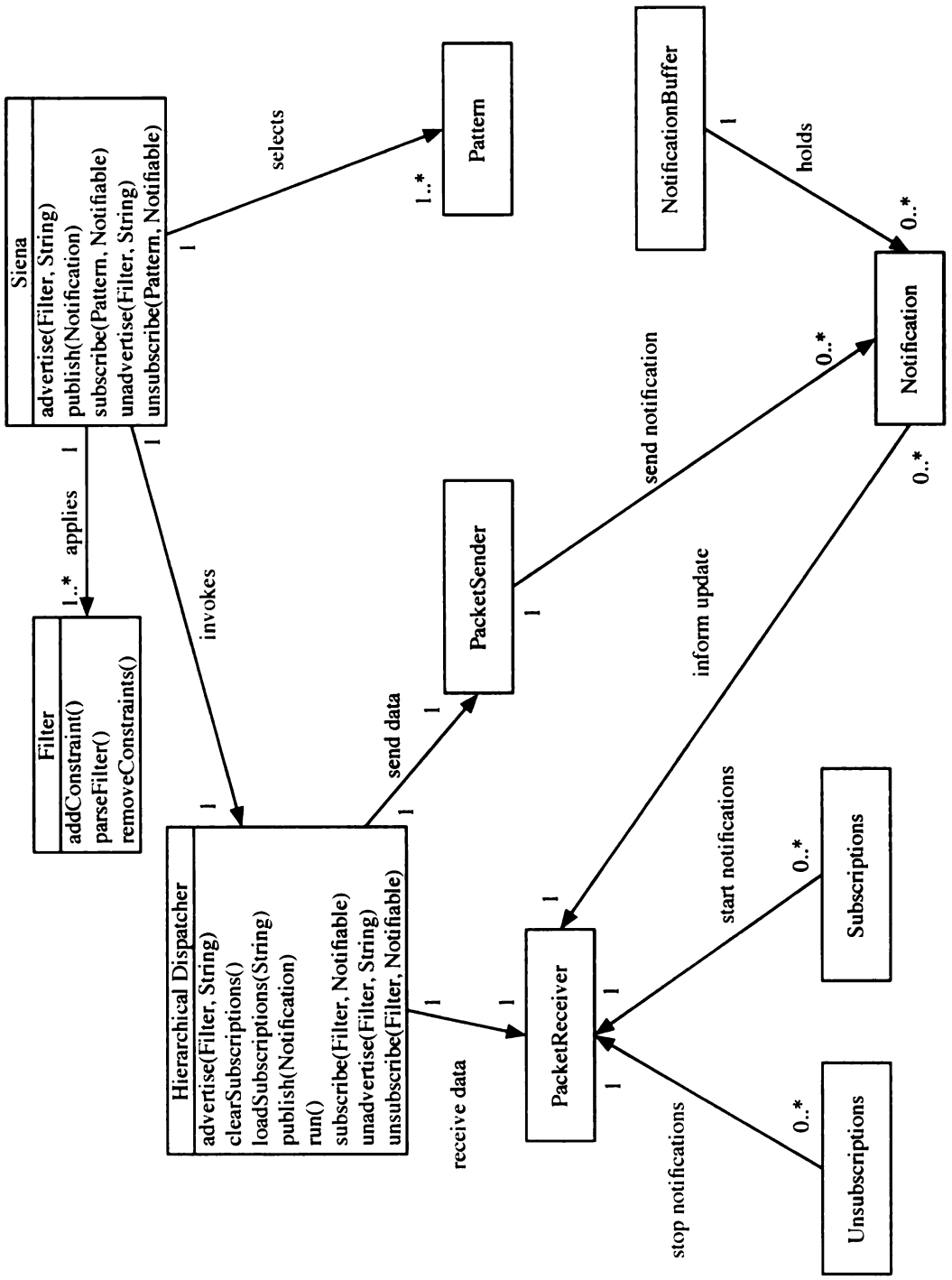

Figure A.6: UML object diagram of the Siena Notification Infrastructure [14].

A.4 *Case-based Reasoning (68)* Pattern

ForkLift Agent [4]. The ForkLift Agent application is a simulated multi-agent system that uses case-based reasoning as the main inference engine for task planning. Specifically, autonomous forklift agents unload, transport, and stack crates from a truck to a determined location in a warehouse. Agents are equipped with sensors to interact with their environment and communication channels to relay important information to other agents. Figure A.7 shows the UML class diagram for the ForkLift agent and its case-based reasoning constructs provided by the FraMaS framework [4]. The case-based reasoning used by the ForkLift agent is similar to the one presented in the *Case-based Reasoning (68)* design pattern. Specifically, both accept some form of event input from the environment and match it against predetermined conditionals. In the ForkLift agent, this task is performed by the Matching class, which searches through the CaseLibrary for a conditional that satisfies the triggered event. Additionally, the ForkLift agent attempts to assimilate similar, yet previously not encountered events, to those found in the CaseLibrary through the Ranking class. The Ranking class takes some event and produces an ordered list of the most applicable cases stored in the CaseLibrary. Once a matching pair of events-case is found, then the appropriate plan is loaded by the CaseAgent. Note, however, that the ForkLift agent is not able to learn new pairs of cases and actions as *Case-based Reasoning (68)* does. As a result, the Learner and Log found in *Case-based Reasoning (68)* is not applicable to the ForkLift agent.

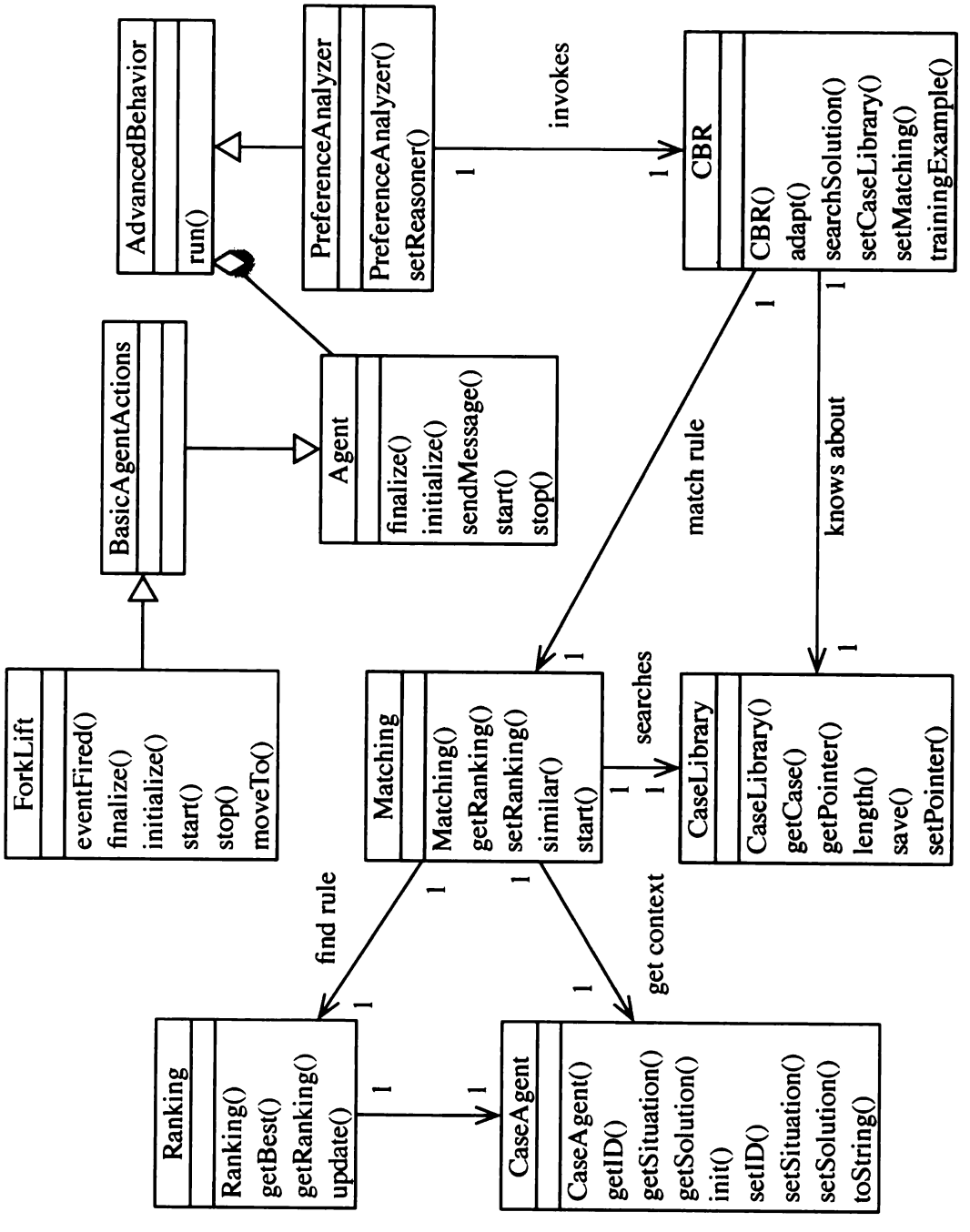

Figure A.7: UML class diagram of the ForkLift Agent [4].

UM-PRS [60]. The Procedural Reasoning System (PRS) is a generic case-based reasoning system that can be applied in domains where procedures are available for handling predetermined situations [60]. Figure A.8 shows an object diagram of the UM-PRS system being used to control a real outdoor vehicle that changes its behavior based on what it senses from the environment. UM-PRS is similar in both structure and function to the *Case-based Reasoning (68)* pattern. Specifically, as new information is obtained by a Monitor, the Database searches for any particular matches. If an event is matched in the Database, then the Interpreter obtains the corresponding plan from the KALibrary. Likewise, the *Case-based Reasoning (68)* pattern receives notifications, in the form of Triggers. Each Trigger is processed by the InferenceEngine in which it attempts to match the Trigger against a particular Rule in the FixedRules object. If a Rule matches the Trigger, then the InferenceEngine produces a Decision which contains the appropriate response plan. Note, however, that the UM-PRS does not have any mechanisms present to facilitate the discovery of new rules once the system is deployed.

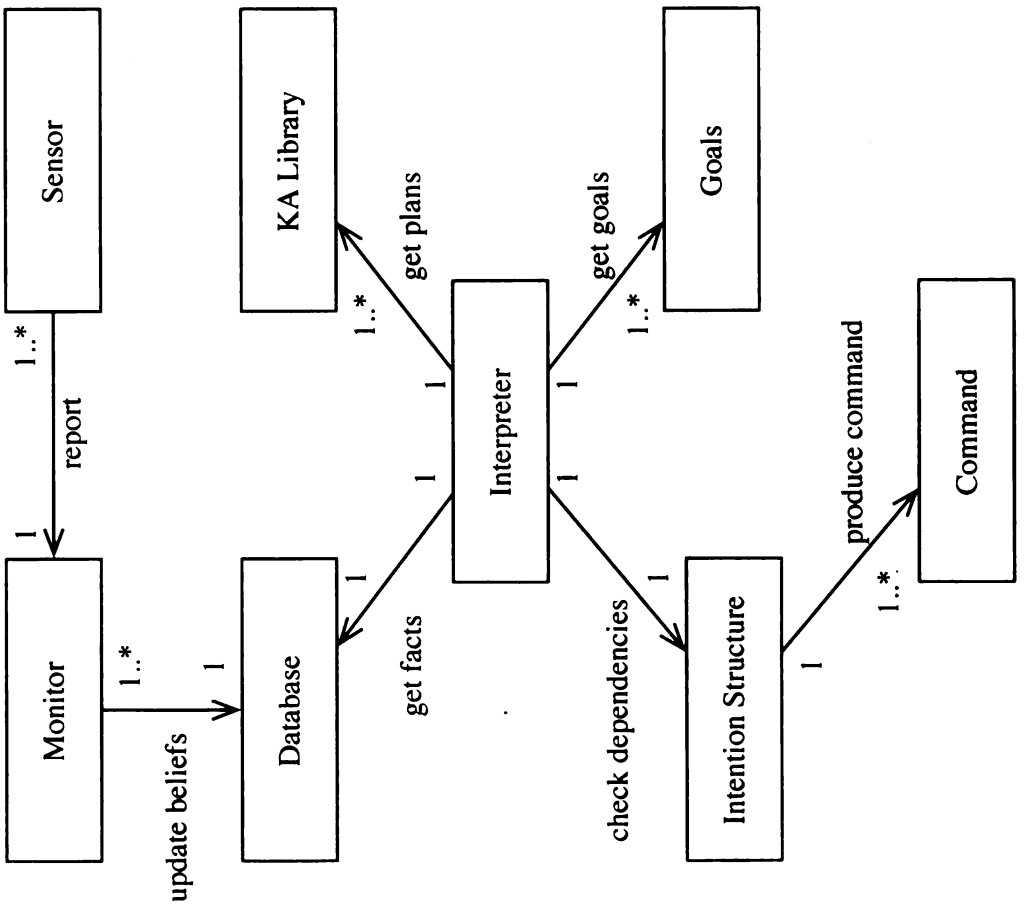

Figure A.8: UML object diagram of the UM-PRS [60].

A.5 *Adaptation Detector (88)* Pattern

XUES Event Distiller Package [37]. XUES (XML Universal Event Service) is an event manipulation system that forms part of the Kinesthetics eXtreme (KX) project and interfaces with the SmartEvents service [79]. XUES comprises three key components to interpret monitoring data, an event packager, an event distiller, and an Event Notifier. The Event Distiller (ED) is a flexible event pattern-recognition and gauge architecture. The ED obtains values reported by probelets and supports multiple-event pattern recognition, time-based validation, and wildcard event matching. Specifically, the ED interprets the monitoring values against specific patterns, or constraints, and submits event notifications. Figure A.9 shows the UML class diagram of the XUES Event Distiller package. Although XUES utilizes state machines to represent the various patterns that can be matched, functionally it is very similar to the *Adaptation Detector (88)* design pattern. For instance, the EDStateMachine is functionally analogous to the Analyzer found in the *Adaptation Detector (88)* pattern. Each EDStateMachine comprises a set of EDStates and EDConsts. The EDStateMachine compares a particular EDState and EDConst against the values reported by the probelets. Likewise, in the *Adaptation Detector (88)* pattern, the Analyzer compares Data produced by sensors against Thresholds. Whenever an event matches a particular pattern or threshold, then a Notification (or Trigger in the *Adaptation Detector (88)* pattern) is generated and sent to the EDBus, which is serviced by the Siena publisher/subscriber infrastructure [14].

Figure A.9: UML class diagram of the XUES Event Distiller [37].

Software Health Monitor [59]. Figure A.10 shows a UML object diagram of the software health monitor presented in [59]. This software health monitor is responsible for processing monitoring information. Specifically, as a Sensor provides information about a particular entity in the system, the Indicator determines whether those values are within acceptable bounds or not. Information from the Sensor is obtained either through the Observer, when the Sensor pushes new information out, or by polling the Sensor directly whenever the Timer determines that a timeout has occurred. If the values reported by the Sensor are deemed unacceptable by the Indicator, then a notification is sent to the HealthMonitor. A similar structure and functionality is provided by the *Adaptation Detector (88)* pattern. In the *Adaptation Detector (88)* pattern, an Observer is responsible for both receiving and polling the Sensor for Data. Once Data has been obtained, the HealthIndicator invokes an Analyzer to compare the monitored values against specific Thresholds. If any value exceeds its corresponding Threshold, then a Trigger is created by the HealthIndicator and fowarded to an entity responsible for determining how the system must be reconfigured.

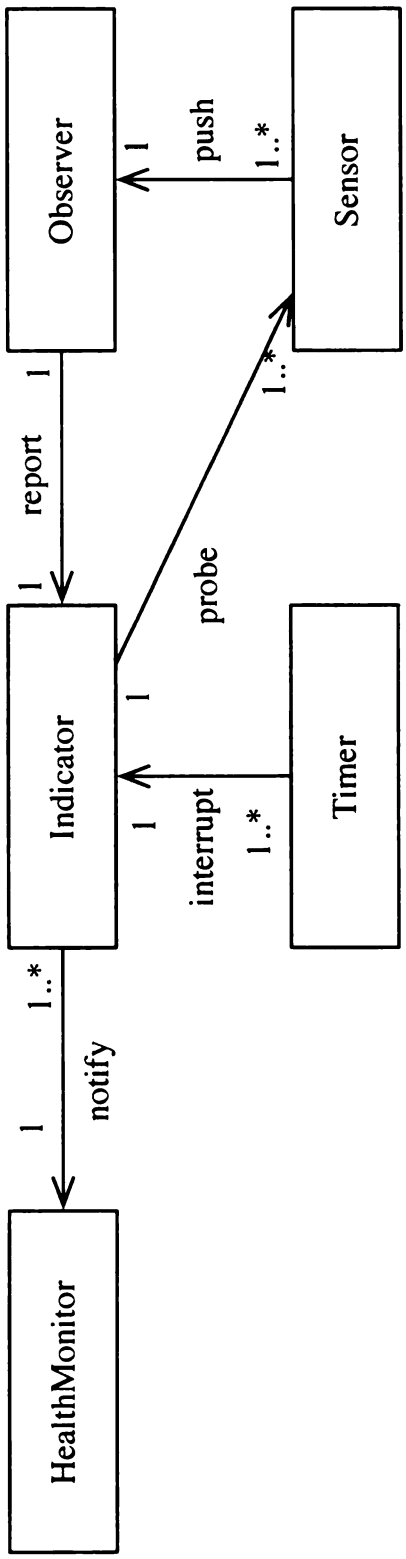

Figure A.10: UML object diagram of the Software Health Monitor [59].

A.6 *Divide and Conquer (78)* Pattern

Care-O-Bot II [39, 40]. Care-O-Bot II is an experimental robot that provides human assistance such as fetching and carrying items around a room. Robots such as Care-O-Bot II need an appropriate control system that achieves high-level complex goals while interacting with a complex and often dynamic environment. For instance, a seemingly simple goal such as "fetch cup" might entail loading an updated map of the environment, calculating the robot's current position and the cup's position, generating a navigation plan, moving towards the cup while probing the environment (to avoid collisions), managing battery power consumption, and so forth. Task decomposition is a commonly used approach to decompose complex goals into simpler goals that can be readily solved. Care-O-Bot II incorporates the metric-FF task decomposition module based on the Planning Domain Description Language(PDDL) [45].

Figure A.11 shows the metric-FF task decomposition module [45] that is used by Care-O-Bot II. In metric-FF, Main sequences the process of task decomposition. Essentially, Main combines the functionality of Solver and Inference Engine from the *Divide and Conquer (78)* pattern. lex-fct-pddl uses the scan-fct-pddl.y, scan-ops-pddl.y, lx-ops-pddl.l and lx-fct-pddl.l files to perform lexing tasks. Main begins by lexing and parsing various files through the lex-fct-pddl and Parse classes. These two classes correspond to the Lexer and Parser classes in *Divide and Conquer (78)*, respectively. To represent goals, metric-FF uses the State class. Metric-ff then uses the Search class to perform an enforced hill-climbing with deletion heuristic search. Likewise, the *Divide and Conquer (78)* pattern uses the Informed class to employ heuristic-based searches. Both *Divide and Conquer (78)* and metric-FF rely on known facts (KB and Fct, respectively) to guide the search process. After the search process yields a solution, metric-FF invokes the Orderings class to produce a sequence of tasks that

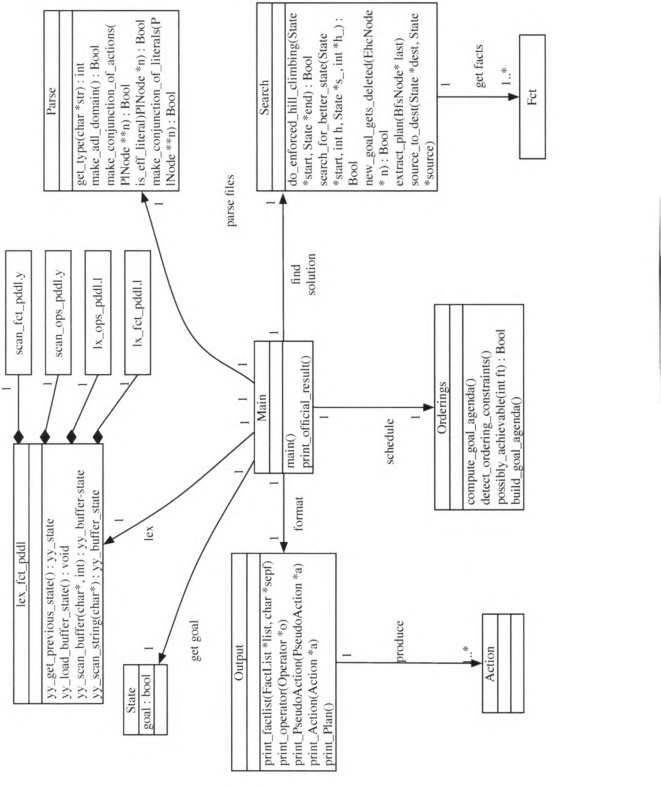

Figure A.11: UML class diagram of the metric-FF (Care-O-Bot II) [39, 40].

will satisfy the overall goal. Likewise, *Divide and Conquer (78)* employs the use of a Dependency Calculator and a Planner to organize the tasks. Finally, metric-FF an Output class to format the solution in terms of Actions.

Rainbow Adaptation Framework [16, 27]. The Rainbow framework incorporates a task decomposition approach when selecting a reconfiguration plan to apply at run time. Figure A.12 shows an object diagram of the various entities that make up Rainbow's task decomposition approach and how they each map with respect to the *Divide and Conquer (78)* pattern. First, in Rainbow, a Goal is represented as an architectural constraint that must remain true throughout execution. The Stitch Language Interpreter [16] is used to translate the utility formula, tactics, and strategies provided by the developer into an internal representation used by Rainbow at run time. The Stitch Language Interpreter provides a Lexer and Parser to provide these functions. Rainbow's Inference Engine is utility-based (described in *TradeOff-Based (106)*). The Dependency Calculator and Planner from *Divide and Conquer (78)* are implemented in Rainbow as Strategies. A Strategy is used to group different tactics (Tasks in *Divide and Conquer (78)*) together into sequences and alternatives that work together to repair the system after a constraint has been violated. Lastly, the Solver class found in *Divide and Conquer (78)* does not exist in Rainbow *per se*. Rather, this capability is handled by the Rainbow framework as a whole.

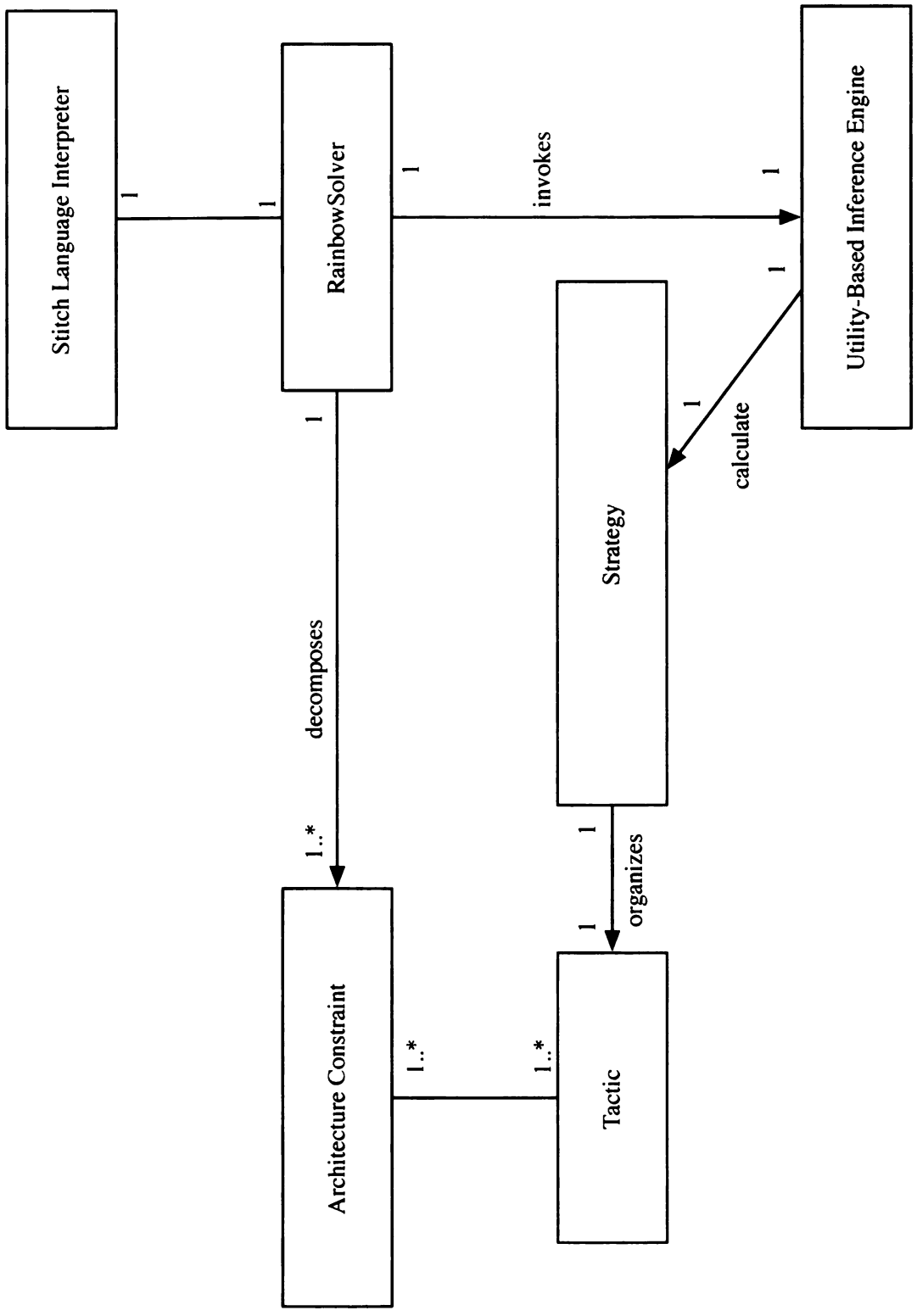

Figure A.12: Object Model for the Task-Decomposition Pattern in Rainbow [16, 27].

A.7 *Architecture-Based (97)* Pattern

Rainbow Adaptation Framework [16, 27]. The Rainbow adaptation framework uses architectural models [28] as part of its decision-making process in two key ways. First, Rainbow analyzes probing data against architectural model constraints to determine if any properties have been violated during execution. If a property has been violated, then Rainbow triggers a reconfiguration request in order to restore the system to safety. Second, Rainbow uses architectural models to determine which reconfiguration plan to apply. Specifically, various architectural models are evaluated and if they satisfy the adaptation requirements, then they are executed. Figure A.13 shows a UML class diagram of the various entities in Rainbow that provide architecture-based reconfiguration, which are similar in both structure and function to the *Architecture-Based (97)* design pattern. For instance, in Rainbow, the AdaptationManager is akin to the EvolutionManager in the *Architecture-Based (97)* pattern. Both are responsible for overseeing the safe reconfiguration of the application whenever a property is no longer satisfied. To determine when a property has been violated, both Rainbow and the *Architecture-Based (97)* pattern rely on architectural models, model repositories and utility functions to evaluate the effects of a model before it is applied. Rainbow wraps Acme constructs (components and connectors) with RainbowModels, which implement the interfaces of Model and ModelRepository. Although the *Architecture-Based (97)* pattern does not explicitly include Acme models, similar component-connector models are encapsulated within ArchitecturalModels. Given the amount of architectural models possible, both Rainbow and the *Architecture-Based (97)* pattern include some entity to manage the set of models. This management functionality is provided in Rainbow by the ModelManager and by ArchitecturalRepository in the *Architecture-Based (97)* pattern. Lastly, the ArchEvaluator in Rainbow evaluates whether a specific architectural model satisfies a set of properties or not. The same functionality is provided by the ConstraintChecker in the *Architecture-Based (97)*

pattern. Thus, when a property is violated, the decision-making process searches for some architectural model that corrects the problem and propagates the structural changes.

Figure A.13: UML class diagram of Rainbow's adaptation manager [16, 27].

MADAM [64]. The Mobility and Adaptation Enabling Middleware (MADAM) uses component-connector models to represent all different configurations, or contexts, that the system can adopt during execution. Figure A.14 shows an elided UML class diagram of the various entities responsible for managing the different forms of components and their connections in MADAM. Although MADAM is a complex middleware that comprises several key components for automatically managing dynamic reconfigurations, it shares several key similarities with the *Architecture-Based (97)* pattern. In particular, in both MADAM and the *Architecture-Based (97)* pattern, every entity in the system is represented as both a set of components and a set of connections through which they can communicate. In MADAM, an InstanceManagement tracks all the component-connector models that have been instantiated in the system. Likewise, in MADAM, a ComponentManagement is responsible for storing, searching, and evaluating all the different models. In the *Architecture-Based (97)* pattern, these functionalities are provided by both the ArchitecturalRepository and the RepairEngine. Although not shown in Figure A.14, in MADAM, every component-connector model is associated with a specific set of properties, which can be verified by the AdaptationManagement. Likewise, in the *Architecture-Based (97)* pattern, a ConstraintChecker evaluates each ArchitecturalModel to determine whether it satisfies any given property.

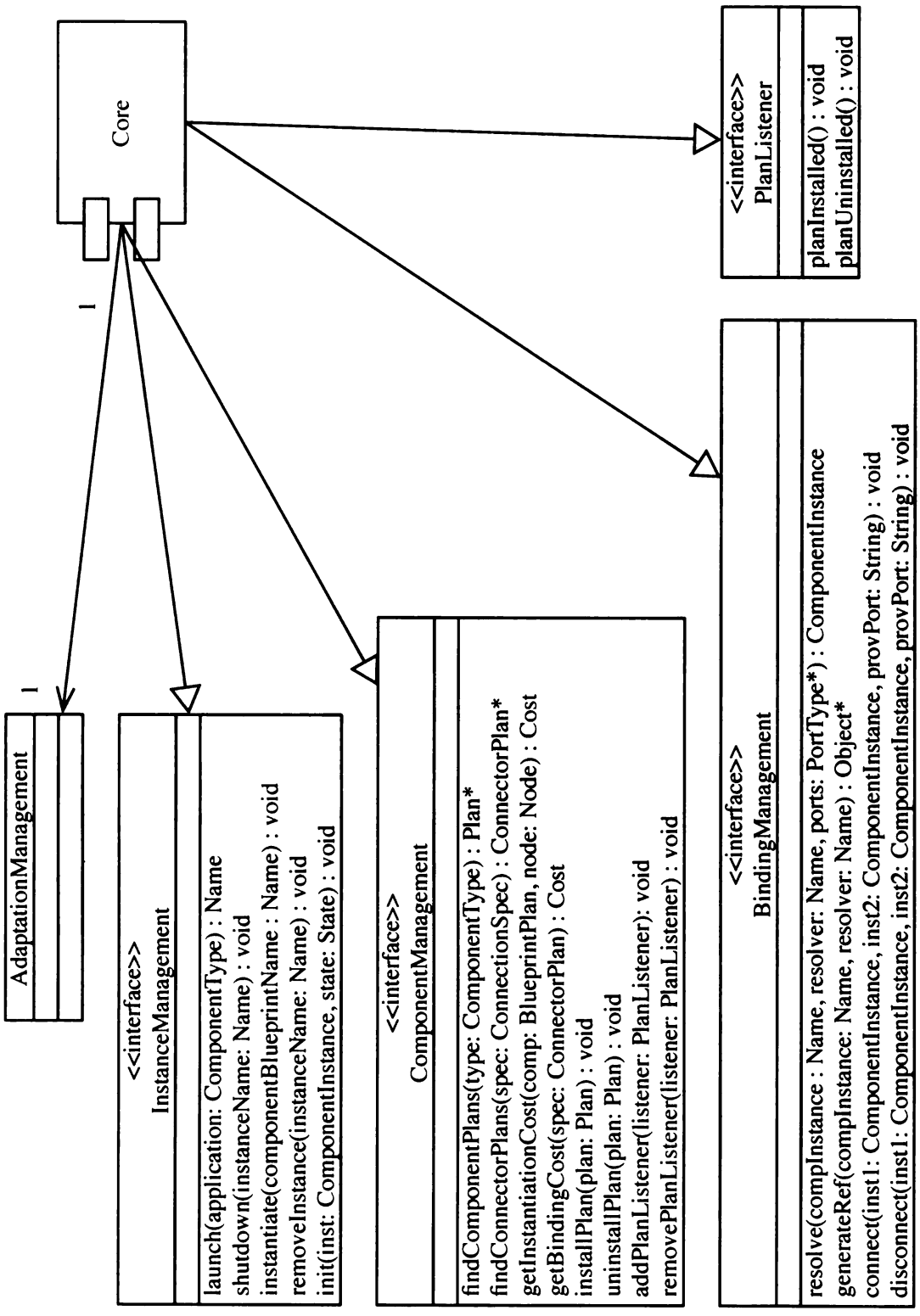

Figure A.14: UML class diagram of MADAM's Core [64].

A.8 *TradeOff-Based (106)* Pattern

Mobility and Adaptation Enabling Middleware (MADAM) [64]. The Mobility and Adaptation Enabling Middleware (MADAM) system uses utility theory in its decision-making process to select specific reconfiguration plans [64]. Figure A.15 shows an elided UML class diagram of the classes responsible for utility-based decision-making in MADAM. Although MADAM is a complex middleware that comprises several key components for automatically managing dynamic reconfigurations, it shares several key similarities with the *TradeOff-Based (106)* design pattern. Specifically, an AdaptationManager is responsible for reasoning on the impact of context reconfigurations across the system. To select a particular reconfiguration and execute it, the Adaptation Manager invokes the AdaptationCoordinator. The AdaptationCoordinator uses various EvaluatorAdapters and ConstProperties to quantify the effects of different reconfiguration plans. The reconfiguration plan that yields the best utility gain is selected by the AdaptationCoordinator and then effected throughout the system. Although not shown in Figure A.15, MADAM also comprises a component that arbitrates different context reconfiguration requests. Likewise, in the *TradeOff-Based (106)* pattern, a Arbiter oversees the different requests for context reconfigurations. The InferenceEngine then determines whether any of the reconfiguration plans would yield better system performance than the current configuration. To maximize utility, UtilityFunctions are employed by the InferenceEngine to quantify the effects of a reconfiguration plan. Notice that MADAM makes no use of DemandForecaster to predict the utility outcome of different reconfiguration plans based on previous knowledge.

Figure A.15: UML component diagram of MADAM [64].

Rainbow Adaptation Framework [16]. The Rainbow framework incorporates an approach that maximizes utility gain when selecting a reconfiguration plan to apply at run time. Figure A.12 shows an object diagram of the various entities that make up Rainbow's utility-based decision-making approach and how they each map with respect to the *TradeOff-Based (106)* pattern. First, in Rainbow, a Goal is represented as an architectural constraint that must remain true throughout execution. The Stitch Language Interpreter [16] is used to translate the utility formula, tactics, and strategies provided by the developer into an internal representation used by Rainbow at run time. Rainbow's Inference Engine is utility-based in the sense that it evaluates a set of different reconfiguration plans (Strategies) and selects the one that yields the maximum utility gain. Likewise, in the *TradeOff-Based (106)* pattern, an InferenceEngine applies a set of UtilityFunctions to quantify the effects of applying a reconfiguration plan. Whichever reconfiguration plan yields the maximum value in terms of utility gain is selected by the InferenceEngine. Note that in Rainbow there are no DemandForecaster and Arbiter objects. Instead, Rainbow's InferenceEngine is a centralized process that selects a reconfiguration plan only to repair a violated constraint. In contrast, the *TradeOff-Based (106)* pattern provides functionality support for users to submit different Objectives to the InferenceEngine.

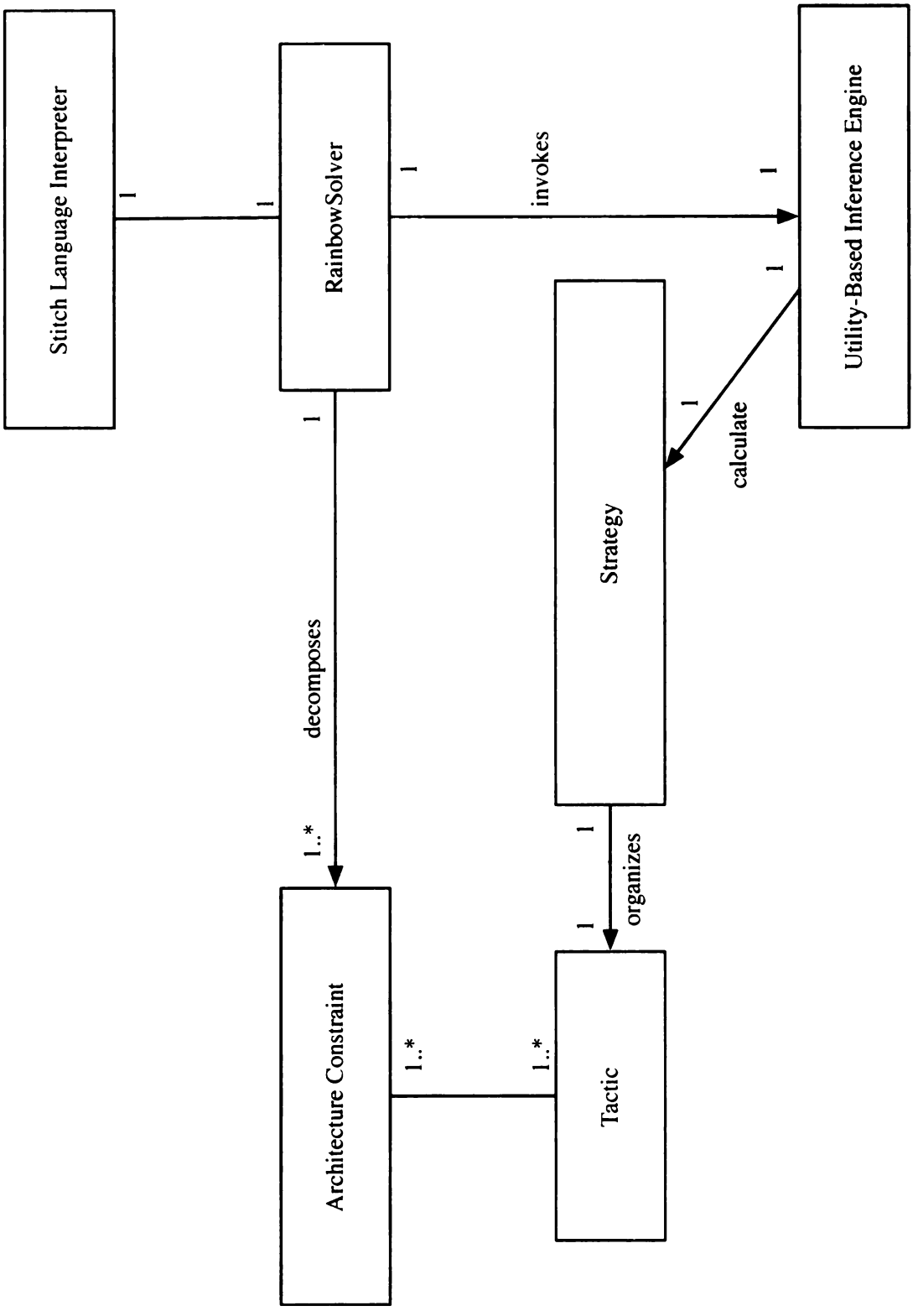

Figure A.16: Object Model for Rainbow's Utility Decision-Making [16].

A.9 *Component Insertion (115)* Pattern

Monitor Reconfiguration [7]. Figure A.17 shows the component diagram for the reconfiguration driver. This simplified component diagram is similar to the one presented in *Component Insertion (115)*. Driver is the key component that oversees the reconfiguration process. To enact some of the changes across the system, however, Driver invokes the functionality of Change Manager.

The reconfiguration consists of inserting a logging component between the Network (Network-comp) and Link (Link-comp) layers at run time (see Figure A.18). Their component insertion approach is similar in behavior to the one presented in the *Component Insertion (115)* pattern. First, both approaches load the respective component into the executing environment. In this specific application, the load command dynamically allocates memory to hold the logging component. Once the component has been allocated into memory, the component is then instantiated to either a default state or some previously stored configuration. Next, the Network-comp and Link-comp are unlinked from each other so the logging component can be inserted between them. The process of unlinking two components from each other is handled by the ArchMM. Specifically, to unlink the components, the system must first passivate Network-comp and Link-comp. This passivating step is accomplished by sending a message to each component, in this case Network-comp and Link-comp, such that they stop producing output and accepting input from each other. Links are then prepared between the newly loaded component and the system. In particular, the logging component will be connected to both the Network-comp and Link-comp components. Again, this step is performed by the ArchMM by redirecting the Network-comp's output to logging and setting logging's output to Link-comp's input. Once the links are in place, the loggingI component is activated.

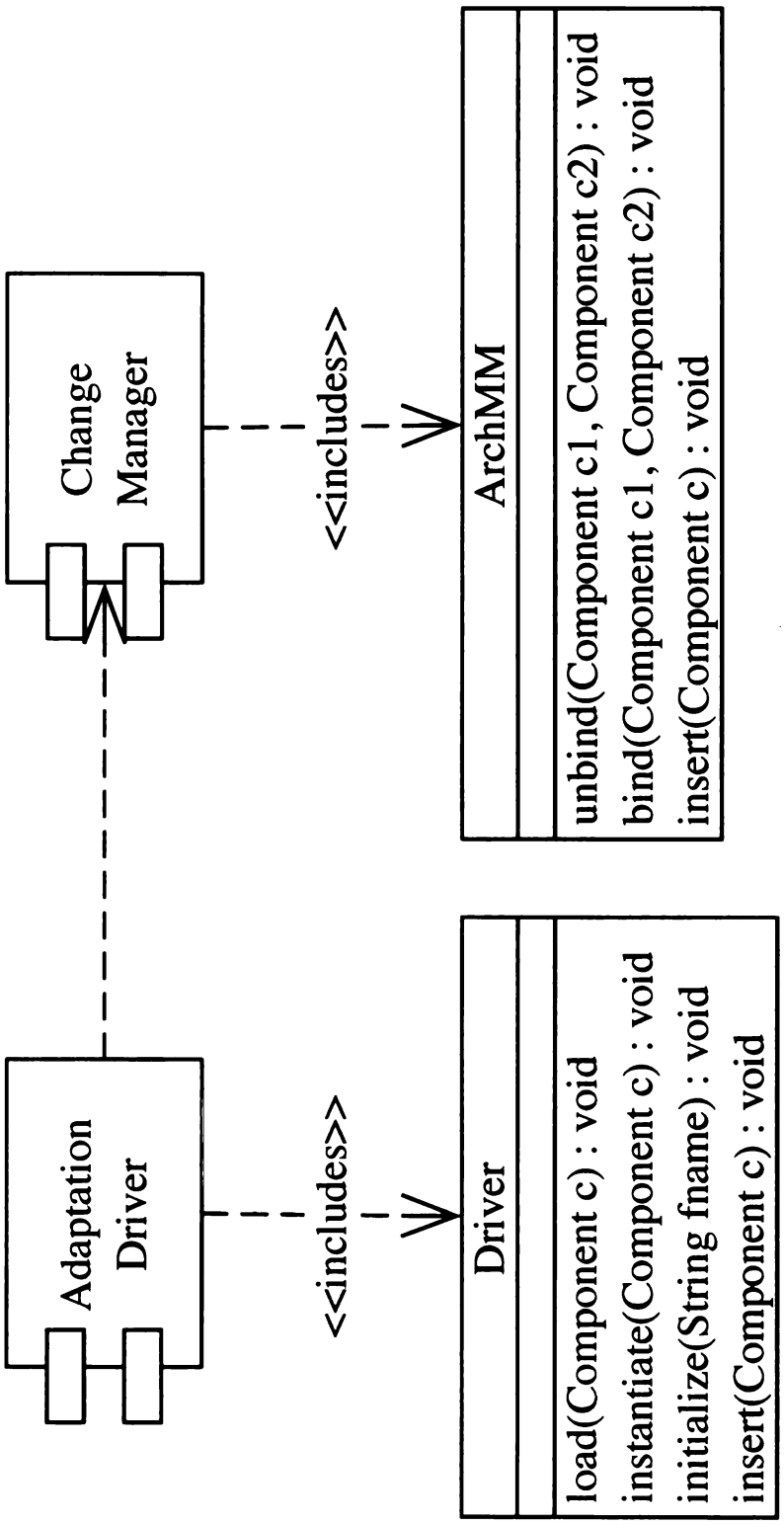

Figure A.17: UML component diagram of the Monitor Reconfiguration [7].

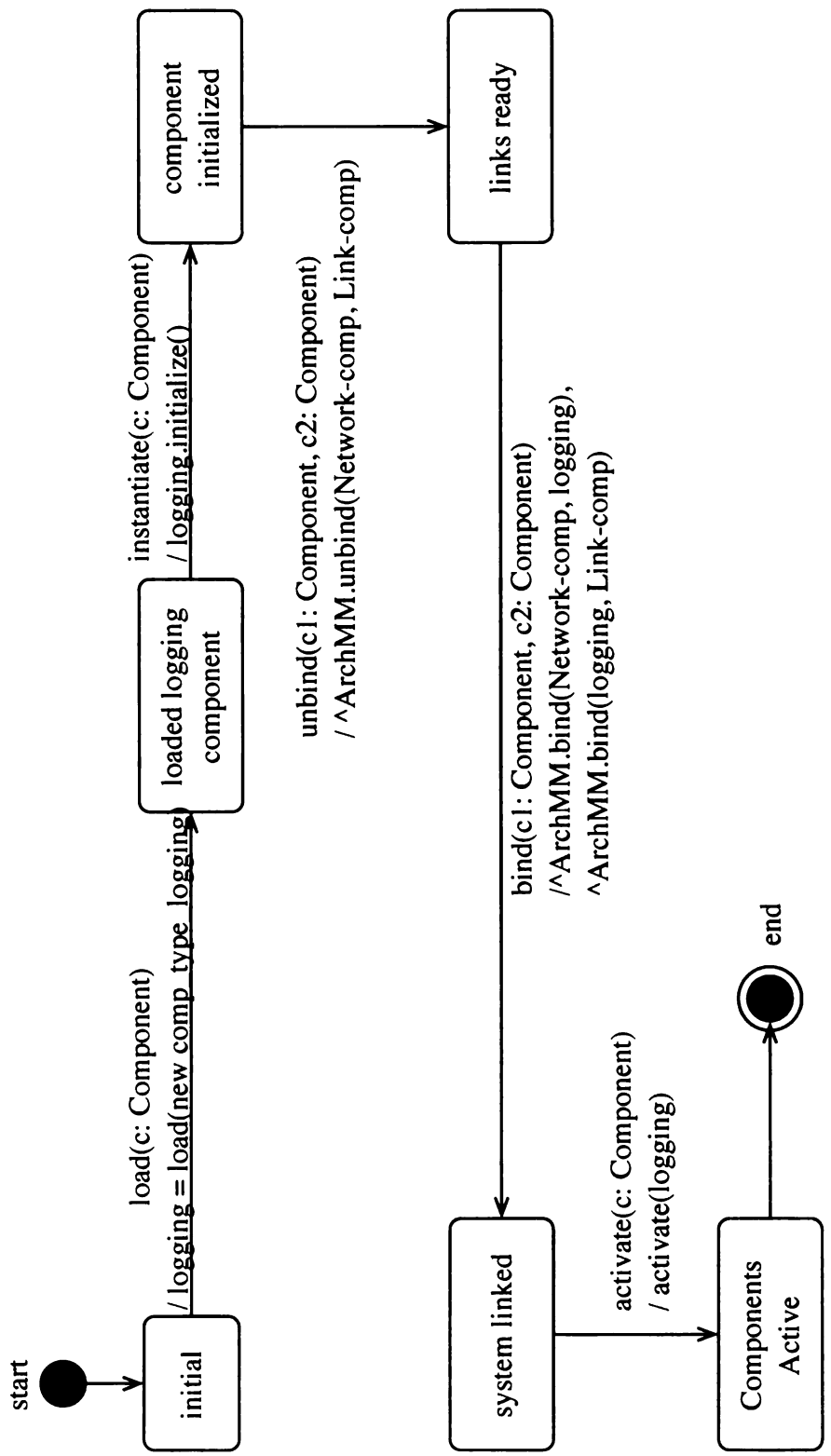

Figure A.18: UML state diagram of inserting a logging component at run time [7].

Conic [61]. Figure A.19 illustrates how components can be inserted at run time with the use of Conic [61]. Specifically, a new display screen is added to the system at run time. The system must reconfigure its input and output displays to properly display data on both screens. Behaviorally, the component insertion reconfiguration is similar to the one presented in *Component Insertion (115)*. However, in this particular example, there is no need to either initialize a screen nor to passivate other screens in the system. As a result, the first step of the reconfiguration process is to create an instance of the display screen, **screen1**, to represent the new component loaded at **sun1**. Next, links are created to connect the two screens together. In particular, the output of **screen1** is connected to the input of **screen2** and the output of **screen2** is connected to the input of **screen1**. For instance, to set up **screen1**'s output to **screen2**, it's output address is set to that of **screen1** and vice versa. Since every display screen remained active during the reconfiguration process, no activation commands are needed to terminate the reconfiguration process.

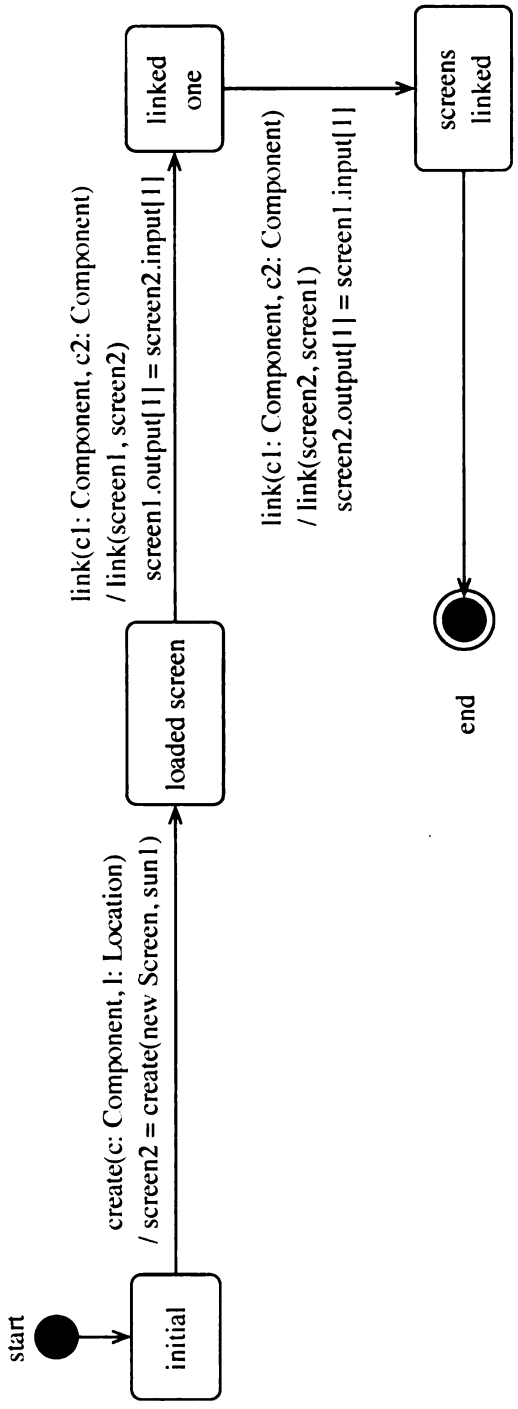

Figure A.19: UML state diagram of inserting a logging component at run time with Conic [61].

A.10 *Component Removal (125)* Pattern

Mobility and Adaptation Enabling Middleware (MADAM) [64]. The Mobility and Adaptation Enabling Middleware (MADAM) [64] is capable of inserting, removing, and modifying components at runtime whenever a context change occurs. Figure A.20 shows an elided UML component diagram of MADAM. Figure A.21 shows a UML state diagram of how MADAM removes a component from the system at runtime. When a reconfiguration is required, the AdaptationManagement is notified of a context change. The AdaptationManagent then invokes the AdaptationCoordinator to evaluate and select the appropriate reconfiguration plan, which in this case is a component removal. One of the key steps in removing a component is first placing it in a quiescent state. AdaptationManagement accomplishes this by sending a suspend notification to the corresponding component, which must implement the Configurator interface. The suspend function is implemented independently by every component that is managed by MADAM and it ensures that the component will not initiate any new transactions and that it will terminate any transactions that may be pending. Once the component is in a quiescent state, the AdaptationManagement proceeds to unbind the component from other components in the system. The unbind operation essentially disconnects the component from the system. Lastly, once all connections have been severed, the AdaptationManagement issues a command for the core system to unload the component.

Figure A.20: UML component diagram of MADAM [64].

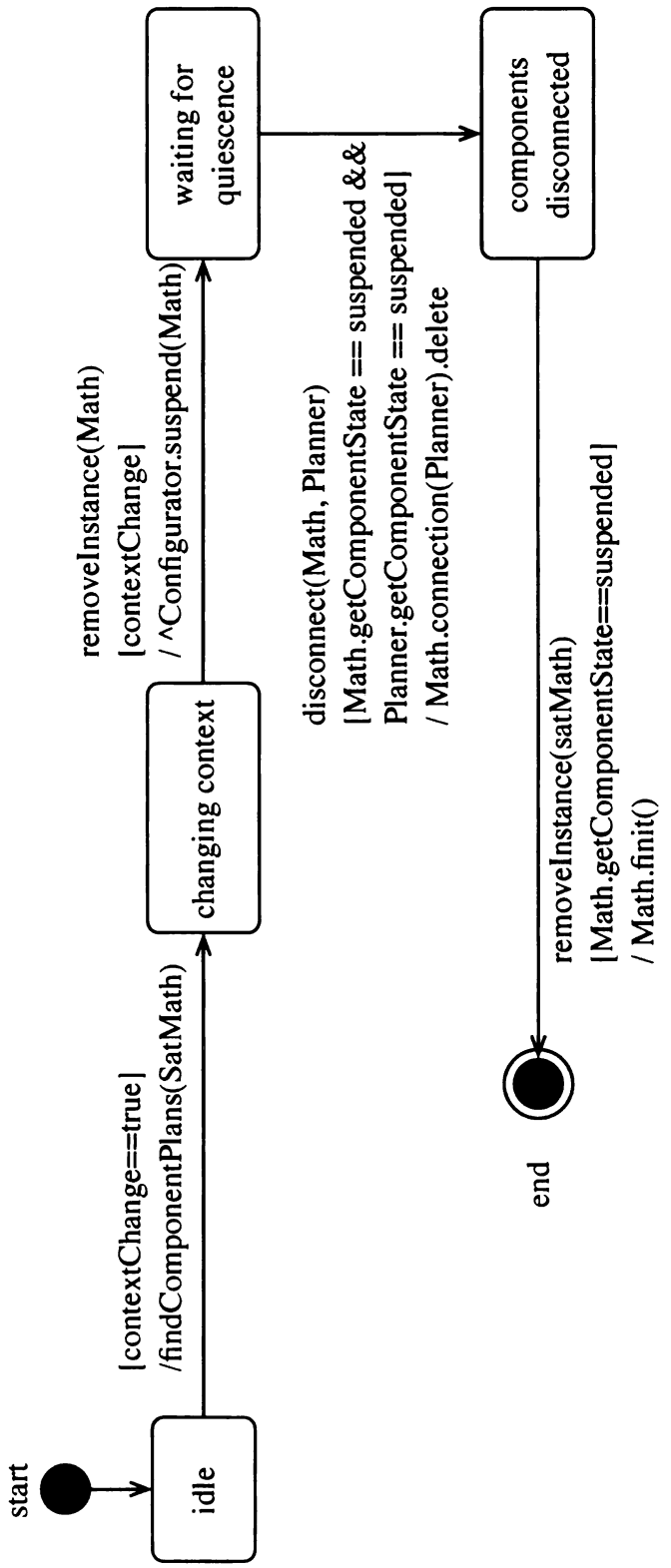

Figure A.21: UML state diagram of removing a component at run time with MADAM [64].

Mobility and Adaptation Enabling Middleware (MADAM) [64]. Figure A.20 shows an elided UML component diagram of MADAM [64]. The Configurator component defines the interface that must be implemented by all components in the MADAM system. This interface explicitly defines all the possible execution states for a given component. Figure A.22 shows a UML state diagram of all the possible states that a component may undergo within MADAM. More importantly, this UML state diagram also shows the specific sequence of execution states that a component must undergo when being inserted and removed from the system. In particular, notice that when a component is in the active status, it must enter the suspended status before MADAM may remove it from the system. In MADAM, when a component enters the suspended status, it may no longer communicate and collaborate with other components.

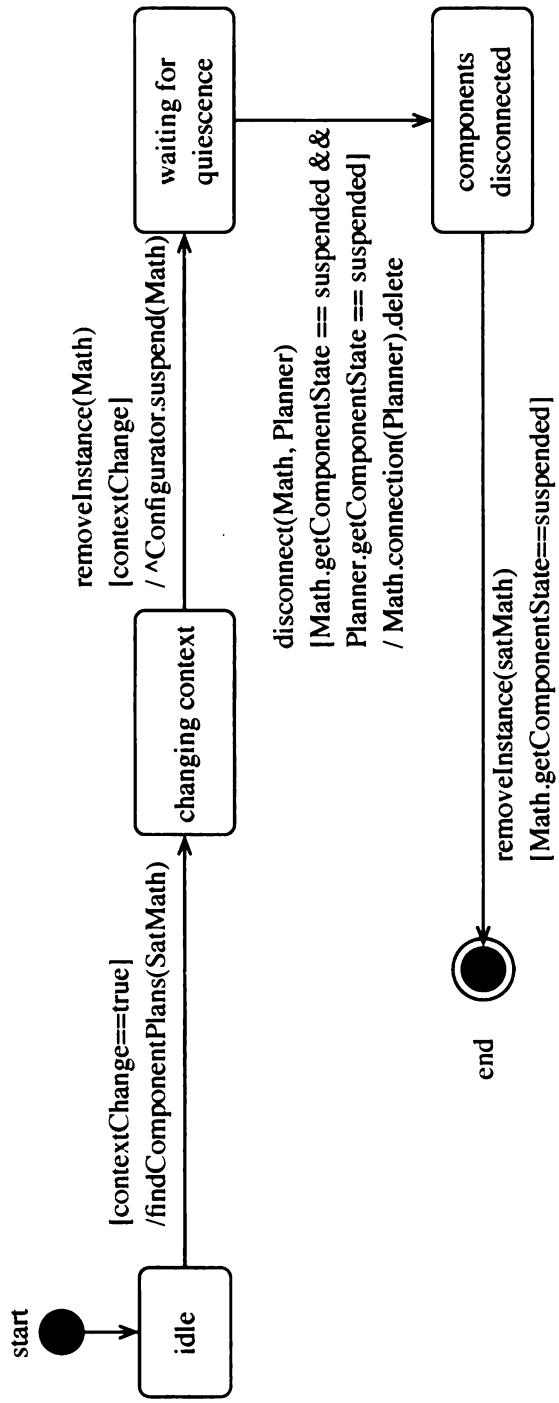

Figure A.22: UML state diagram of a component's states in MADAM [64].

A.11 *Server Reconfiguration (135)* Pattern

Reconfiguring Servers in Equus [52]. Kindberg described how to safely reconfigure a server architecture at runtime in [52]. Figure A.23 shows a UML component diagram with the key entities responsible for reconfiguring a server architecture at runtime. Figure A.24 shows a UML state diagram illustrating the process of reconfiguring a server architecture through the use of proxies. Specifically, the solution is centered around a generic server Proxy such that it interacts with both Clients and Servers during the reconfiguration process. That is, during normal operation, the Proxy forwards all incoming Client messages directly to the active Server. When a reconfiguration is required, however, the Proxy queues incoming Client messages into a Buffer. Meanwhile, the system waits for the active Server to finish servicing all pending transactions and then enter a quiescent state. Once the Server is quiescent, it is deactivated and unloaded from the Equus distributed system. A new Server can then be loaded by the environment and initialized. After the new Server is activated, it will proceed to service all pending Client messages until the Buffer is empty. Lastly, the Proxy will reorganize connections such that new incoming messages are forwarded directly to the new Server. Thus, in this manner, the client/server architecture is reconfigured transparently by means of a proxy. The same behavior can be found in the *Server Reconfiguration (135)* pattern. Although the design pattern does not involve a Proxy, the steps required to safely reconfigure an active server are similar. Specifically, incoming requests must be stored somewhere so they are not lost during the reconfiguration process. While incoming messages are queued, the Server will eventually reach a quiescent state. Likewise, after the reconfiguration is complete, the queued requests are serviced by the new Server.

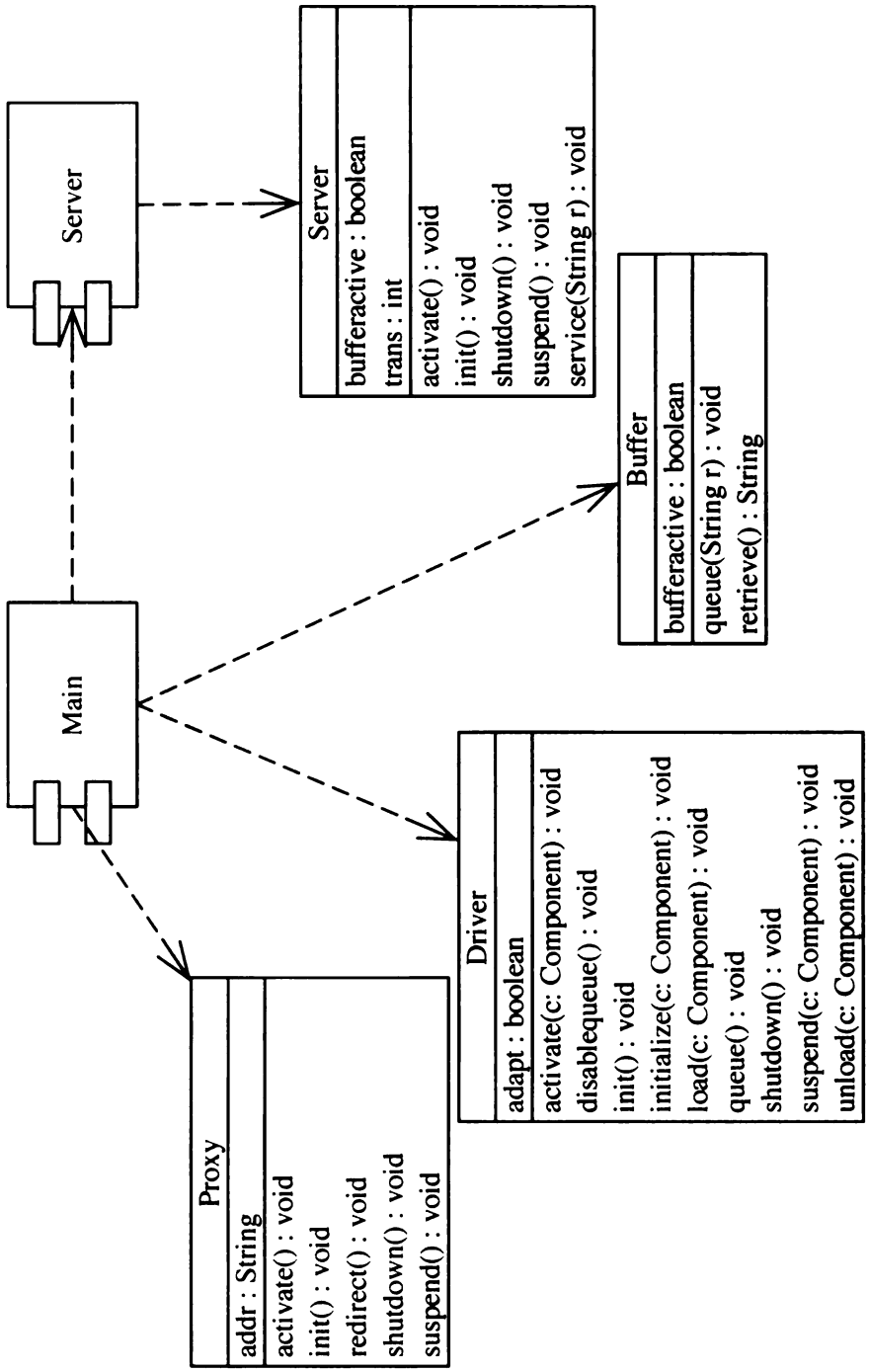

Figure A.23: UML component diagram of the Equus distributed environment [52].

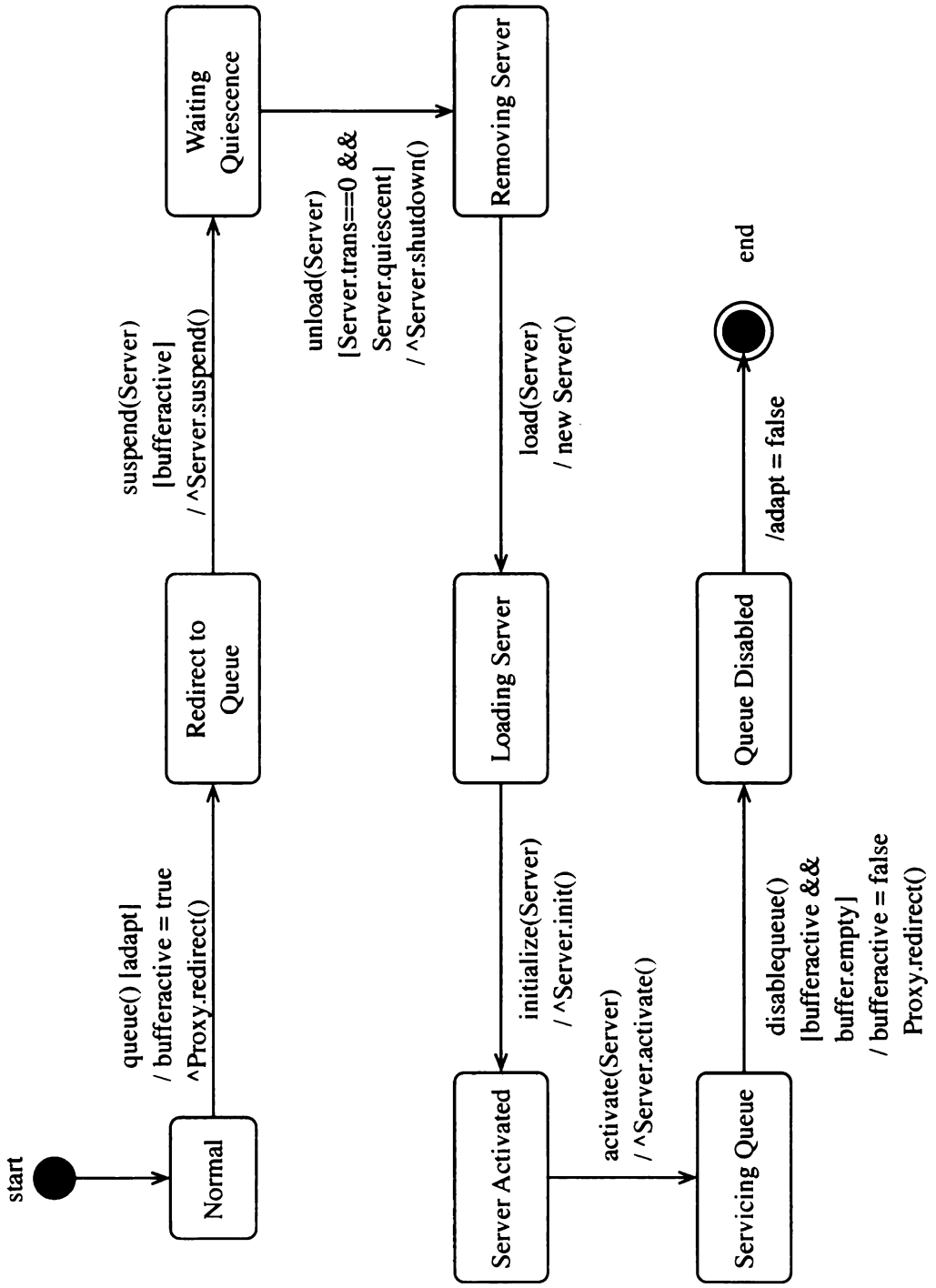

Figure A.24: UML state diagram of removing a server component at run time in the Equus distributed environment [52].

A.12 *Decentralized Reconfiguration (145)* Pattern

OpenRec Reconfiguration [44]. Figure A.25 shows a UML component diagram for OpenRec, a framework for managing dynamic reconfiguration [44]. The UML component diagram highlights two key components in the framework, the OpenRec component and the Algorithm component. The OpenRec component manages the various interactions between components and connectors in the OpenRec system at any time. The Algorithm component is responsible for dynamically reconfiguring the components and connectors managed by the OpenRec component. Figure A.26 shows a UML state diagram for a typical component removal in a decentralized environment. Specifically, the component that wants to disengage and terminate will first commence the reconfiguration process by blocking incoming transaction requests. In addition, the component is given enough time to terminate any pending transactions it may be currently servicing. These two actions enable a component to reach a quiescent state in bounded time. Once the component is in a quiescent state, it disconnects itself from other components. Lastly, once all connections have been removed, the component proceeds to call its terminate function, which cleans up any allocated memory and finishes execution.

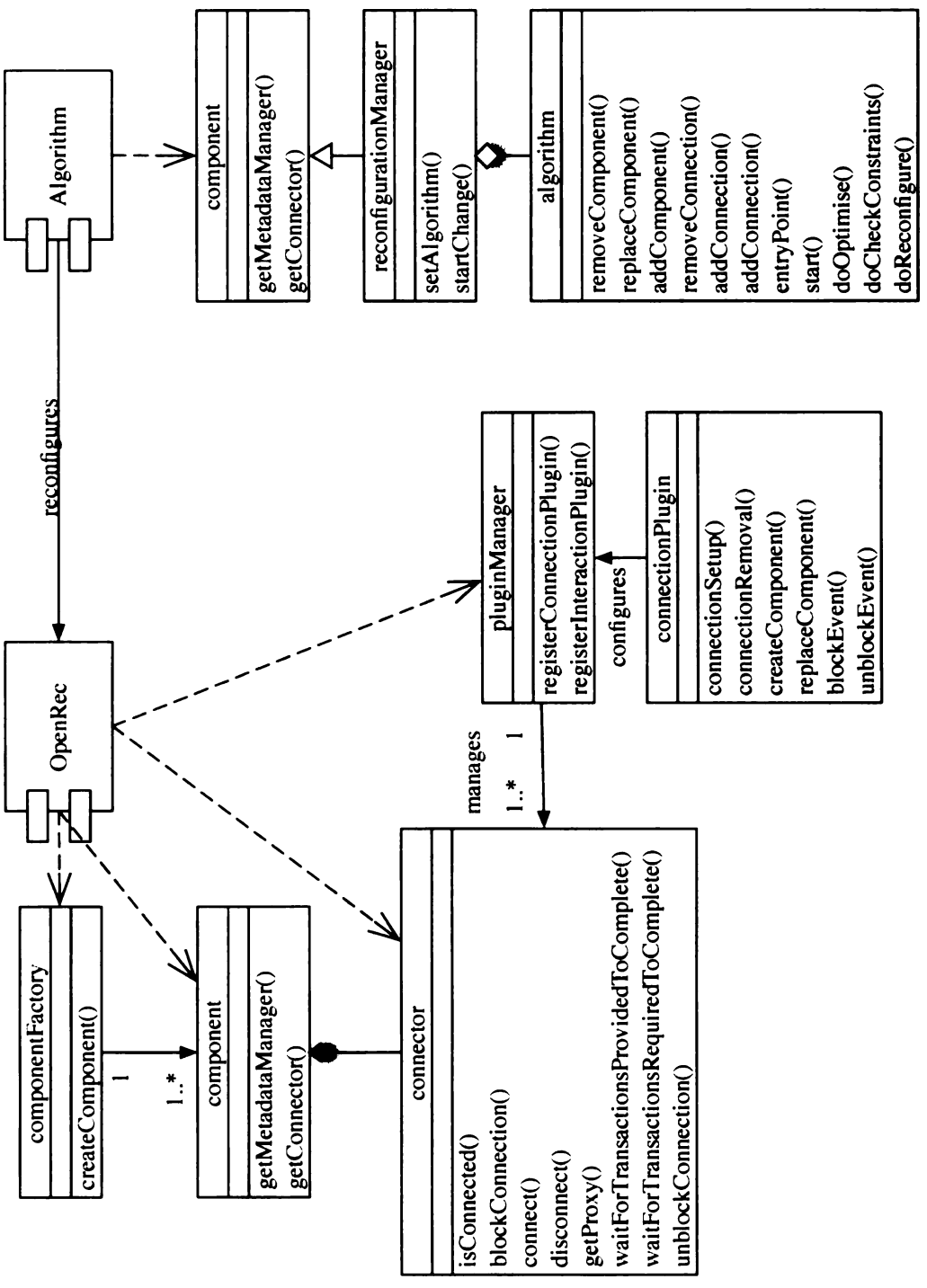

Figure A.25: UML component diagram of OpenRec [44].

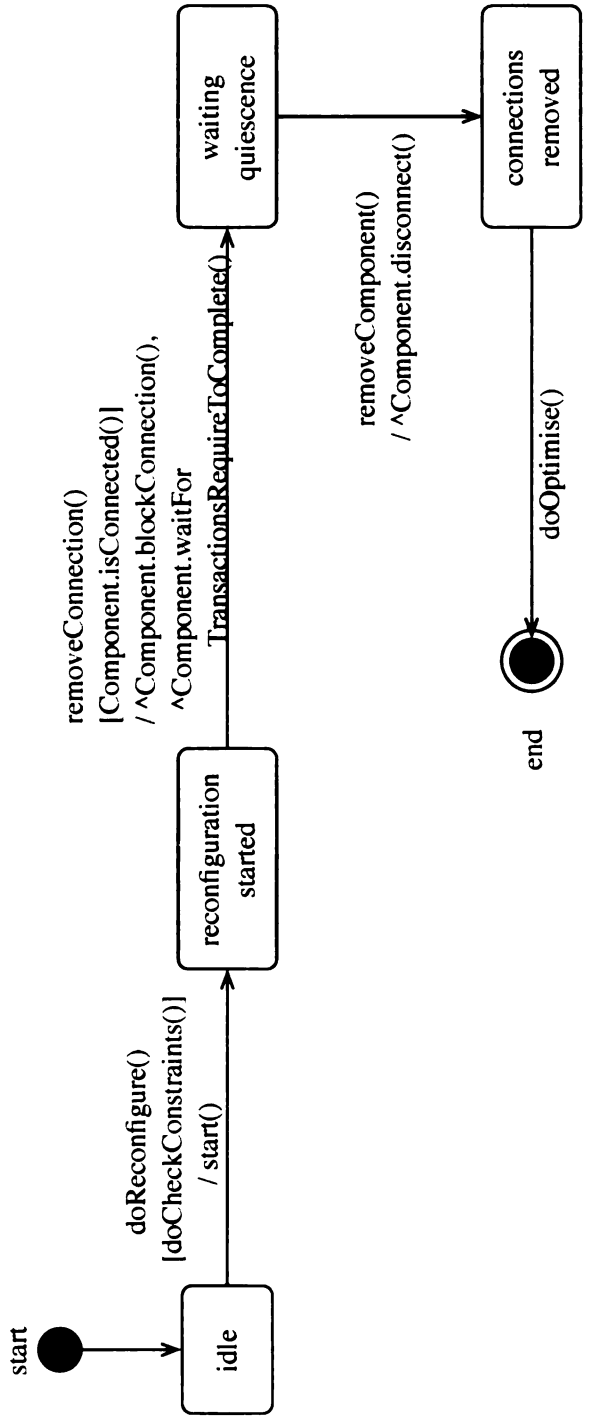

Figure A.26: UML state diagram of a decentralized component being removed in OpenRec [44].

OpenRec Reconfiguration [44]. Figure A.25 shows a UML component diagram for OpenRec, a framework for managing dynamic reconfiguration [44]. The UML component diagram highlights two key components in the framework, the OpenRec component and the Algorithm component. The OpenRec component manages the various interactions between components and connectors in the OpenRec system at any time. The Algorithm component is responsible for dynamically reconfiguring the components and connectors managed by the OpenRec component. Figure A.27 shows a UML state diagram for a typical component insertion in a decentralized environment. The first task is to create and allocate the necessary resources for the new component. Then, the component that wants to engage will first commence the reconfiguration process by adding connections to other components it wants to communicate with. Lastly, once all connections have been established, the component proceeds to operate normally.

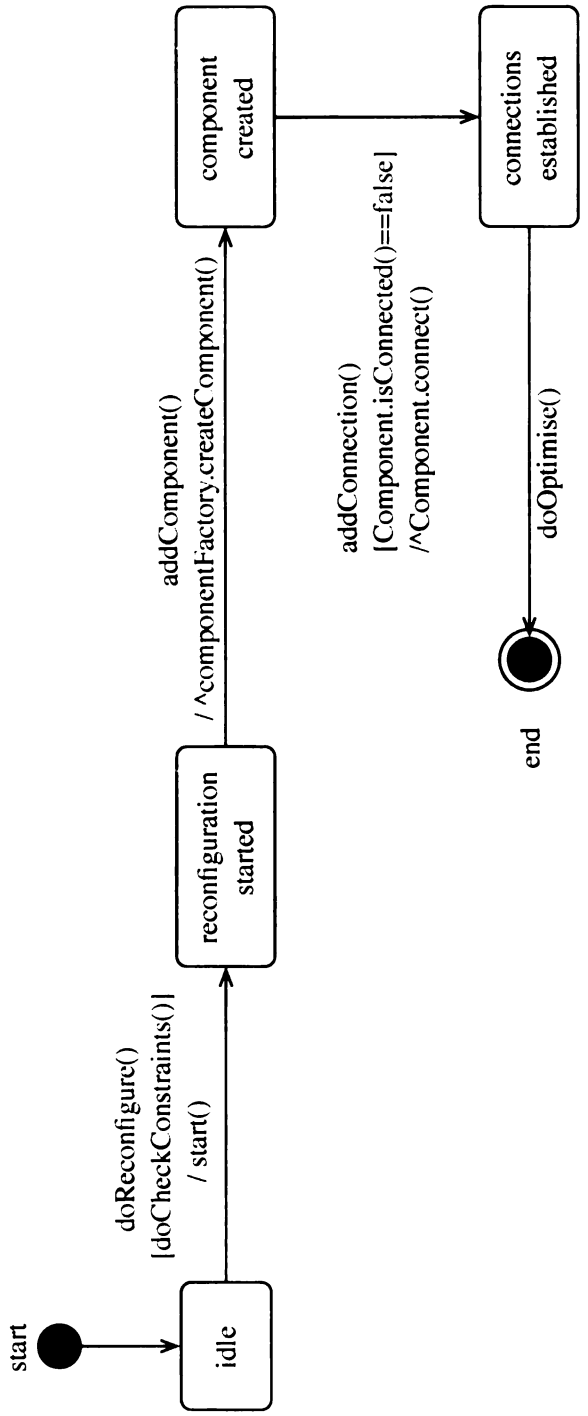

Figure A.27: UML state diagram of a decentralized component being removed in OpenRec [44].

BIBLIOGRAPHY

Bibliography

[1] James S. Albus. Task decomposition. In *Proceedings of the 8th IEEE International Symposium on Intelligent Control*, August 1993.

[2] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Schlomo Angel. *A Pattern Language.* Oxford University Press, 1977.

[3] Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382, 1998.

[4] Henri Avancini. Framas a java framework for multiagent systems. Master's thesis, Buenos Aires National Central University, 2000.

[5] Ricardo Barbosa and Luis M. Pinho. Mechanisms for reflection-based monitoring of real-time systems. Technical report, Polytechnic Institute of Porto, Porto, Portugal, 2004.

[6] Luciano Baresi, Sam Guinea, and Giordano Tamburrelli. Towards decentralized self-adaptive component-based systems. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 57–64, New York, NY, USA, 2008. ACM.

[7] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In *EWSA*, pages 1–17, 2005.

[8] Karun Biyani. Dynamic composition of distributed components. Master's thesis, Michigan State University, 2003.

[9] Karun N. Biyani and Sandeep S. Kulkarni. Building component families to support adaptation. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[10] Frank Buschmann, Kevlin Henney, and Douglas Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing (Wiley Software Patterns Series).* John Wiley & Sons, 2007.

[11] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented software architecture: a system of patterns.* John Wiley & Sons, Inc., New York, NY, USA, 1996.

[12] Javier Cámara, Carlos Canal, Javier Cubo, and Juan Manuel Murillo. An aspect-oriented adaptation framework for dynamic component evolution. *Electron. Notes Theor. Comput. Sci.*, 189:21–34, 2007.

[13] Carlos Canal, Ernesto Pimentel, and José M. Troya. Specification and refinement of dynamic software architectures. In *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 107–126, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[14] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Content-based addressing and routing: A general model and its application. Technical Report CU-CS-902-00, Department of Computer Science, University of Colorado, jan 2000.

[15] Anil Chawla and Alessandro Orso. A generic instrumentation framework for collecting dynamic information. *SIGSOFT Softw. Eng. Notes*, 29(5):1–4, 2004.

[16] Shang-Wen Cheng, David Garlan, and Bradley Schmerl. Architecture-based self-adaptation in the presence of multiple objectives. In *SEAMS '06: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems*, pages 2–8, New York, NY, USA, 2006. ACM.

[17] David M. Chess, Alla Segal, and Ian Whalley. Unity: Experiences with a prototype autonomic computing system. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 140–147, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Geoff Coulson, Paul Grace, Gordon Blair, Wei Cai, Chris Cooper, David Duce, Laurent Mathy, Wai Kit Yeung, Barry Porter, Musbah Sagar, and Wei Li. A component-based middleware framework for configurable and reconfigurable grid computing: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(8):865–874, 2006.

[19] Dylan Dawson, Ron Desmarais, Holger M. Kienle, and Hausi A. Müller. Monitoring in adaptive systems using reflection. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 81–88, New York, NY, USA, 2008. ACM.

[20] T. Dewitt, T. Gross, B. Lowekamp, N. Miller, P. Steenkiste, J. Subhlok, and D. Sutherland. Remos: A resource monitoring system for network aware applications, 1997.

[21] Yunsi Fei, Lin Zhong, and Niraj K. Jha. An energy-aware framework for dynamic software management in mobile computing systems. *Trans. on Embedded Computing Sys.*, 7(3):1–31, 2008.

[22] S. Fickas and M.S. Feather. Requirements monitoring in dynamic environments. *re*, 00:140, 1995.

[23] Scott D. Fleming, Betty H. C. Cheng, R. E. Kurt Stirewalt, and Philip K. McKinley. An approach to implementing dynamic adaptation in c++. In *DEAS '05: Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, pages 1–7, New York, NY, USA, 2005. ACM.

[24] Frank Fock. The snmp api for java. http://www.snmp4j.org/index.html.

[25] Ira R. Forman and Nate Forman. *Java Reflection in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2004.

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[27] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[28] David Garlan, Robert T. Monroe, and David Wile. Acme: An architecture description interchange language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[29] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 27–32, New York, NY, USA, 2002. ACM.

[30] David Garlan, Bradley Schmerl, and Jichuan Chang. Using gauges for architecture-based monitoring and adaptation. In *Proceedings of the Working Conference on Complex and Dynamic Systems Architecture*, 12-14 December 2001.

[31] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A comprehensive solution for application-level adaptation.

[32] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems - survey and synthesis. *Decis. Support Syst.*, 42(4):2164–2185, 2007.

[33] Heather Goldsby and Betty H.C. Cheng. Automatically generating behavioral models of adaptive systems to address uncertainty. In *To appear in the MODELS (2008) conference*, 2008.

[34] Hassan Gomaa and Mohamed Hussein. Software reconfiguration patterns for dynamic evolution of software architectures. In *WICSA '04: Proceedings of the Fourth Working IEEE/IFIP Conference on Software Architecture*, page 79, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Michael M. Gorlick and Rami R. Razouk. Using weaves for software construction and analysis. In *ICSE '91: Proceedings of the 13th international conference on Software engineering*, pages 23–34, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.

[36] Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter, and Danny Hughes. Dynamic reconfiguration in sensor middleware. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks*, pages 1–6, New York, NY, USA, 2006. ACM.

[37] Philip N. Gross, Suhit Gupta, Gail E. Kaiser, Gaurav S Kc, and Janak K. Parekh. Model for systems monitoring. In *Working Conference on Complex and Dynamic Systems Architecture*, December 2001.

[38] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. *SIGPLAN Not.*, 37(11):161–173, 2002.

[39] M. Hans and W. Baum. Concept of a hybrid architecture for care-o-bot. In *In proceedings of ROMAN-2001*, pages 407–411, 2001.

[40] Matthias Hans. The control architecture of care-o-bot ii. In *E. Prassler et al (Eds.): Advances in Human-Robot Interaction, STAR 14*, pages 321–330. Springer-Verlag Berlin Heidelberg 2005, 2004.

[41] James E. Hanson, Ian Whalley, David M. Chess, and Jeffrey O. Kephart. An architectural approach to autonomic computing. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 2–9, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Matthew J. Hawthorne and Dewayne E. Perry. Exploiting architectural prescriptions for self-managing, self-adaptive systems: a position paper. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 75–79, New York, NY, USA, 2004. ACM.

[43] Dennis Heimbigner and Alexander Wolf. Definition, deployment and use of gauges to manage reconfigurable component-based system. Technical Report A082924, University of Colorado, 2004.

[44] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[45] J. Hoffmann and B. Nebel. The ff planning system: Fast plan generation through heuristic search. In *Journal of Artificial Intelligence Research*, volume 14, pages 253–302, 2001.

[46] Christine Hofmeister and James M. Purtilo. Dynamic reconfiguration in distributed systems: Adapting software modules for replacement. In *ICDCS*, pages 101–110, 1993.

[47] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[48] E. Kasten and P. McKinley. Adaptive java: Refractive and transmutative support for adaptive software, 2001.

[49] John Keeney and Vinny Cahill. Chisel: A policy-driven, context-aware, dynamic adaptation framework. *policy*, 00:3, 2003.

[50] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[52] Tim Kindberg. Reconfiguring client-server systems. Technical report, Proc. International Workshop on Configurable Distributed Systems (IWCDS94, 1993.

[53] Michael Kircher and Prashant Jain. *Pattern-Oriented Software Architecture: Patterns for Resource Management.* John Wiley & Sons, 2004.

[54] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, a Claudio Magalh and Roy H. Campbell. Monitoring, security, and dynamic configuration with the dynamictao reflective orb. In *Middleware '00: IFIP/ACM International Conference on Distributed systems platforms*, pages 121–143, Secaucus, NJ, USA, 2000. Springer-Verlag New York, Inc.

[55] Sascha Konrad, Betty H. C. Cheng, and Laura A. Campbell. Object analysis patterns for embedded systems. *IEEE Transactions on Software Engineering*, 30(12):970–992, 2004.

[56] J. Kramer and J. Magee. Analysing dynamic change in software architectures: A case study. In *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*, page 91, Washington, DC, USA, 1998. IEEE Computer Society.

[57] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[58] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Washington, DC, USA, 2007. IEEE Computer Society.

[59] Alexander Lau. Design patterns for software health monitoring. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, pages 467–476, Washington, DC, USA, 2005. IEEE Computer Society.

[60] Jacho Lee, Marcus J. Huber, Edmund H.Durfee, and Patrick G. Kenny. Um-prs: An implementation of the procedural reasoning system for multirobot applications. In *AIAA/NASA Conference on Intelligent Robots in Field, Factory, Service, and Space (CIRFFSS '94)*, pages 1–8. American Institute of Aeronautics and Astronautics, 1994.

[61] Jeff Magee, Jeff Kramer, and Morris Sloman. Constructing distributed systems in conic. *IEEE Transactions on Software Engineering*, 15(6), 1989.

[62] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[63] William E. McUmber and Betty H. C. Cheng. A general framework for formalizing uml with formal languages. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 433–442, Washington, DC, USA, 2001. IEEE Computer Society.

[64] Marius Mikalsen, Nearchos Paspallis, Jacqueline Floch, Erlend Stav, George A. Papadopoulos, and Akis Chimaris. Distributed context management in a mobility and adaptation enabling middleware (madam). In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 733–734, New York, NY, USA, 2006. ACM.

[65] D. Nau, H. Munoz-Avila, Y. Cao, A. Lotem, and S. Mitchell. Total-order planning with partially ordered subtasks. In *Proceedings of IJCAI01*, pages 425–430. Morgan Kaufmann, San Francisco, 2001.

[66] H. B. Newman, I. C. Legrand, P. Galvez, R. Voicu, and C. Cirstoiu. Monalisa : A distributed monitoring service architecture, 2003.

[67] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[68] Andres J. Ramirez and Betty H. C. Cheng. Verifying and analyzing adaptive logic through uml state models. In *ICST*, pages 529–532, 2008.

[69] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorenson. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[70] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[71] Sarmad Sadik, Arshad Ali, Hafiz Farooq Ahmad, and Hiroki Suguri. Policy based ontology framework for mobile agents. In *ACIS-ICIS*, pages 483–488. IEEE Computer Society, 2007.

[72] S. M. Sadjadi and P. K. McKinley. Act: An adaptive corba template to support unanticipated adaptation. *icdcs*, 00:74–83, 2004.

[73] S. Masoud Sadjadi, Philip K. McKinley, and Betty H. C. Cheng. Transparent shaping of existing software to support pervasive and autonomic computing. *SIGSOFT Softw. Eng. Notes*, 30(4):1–7, 2005.

[74] Douglas C. Schmidt. Middleware for real-time and embedded systems. *Commun. ACM*, 45(6):43–48, 2002.

[75] Mary Shaw. Beyond objects: a software design paradigm based on process control. *SIGSOFT Softw. Eng. Notes*, 20(1):27–38, 1995.

[76] Reid Simmons, Long-Ji Lin, and Christopher Fedor. Autonomous task control for mobile robots. In *Proceedings of the 5th IEEE International Symposium on Intelligent Control*, September 1990.

[77] Brian Tierney, Brian Crowley, Dan Gunter, Mason Holding, Jason Lee, and Mary Thompson. A monitoring sensor management system for grid environments. In *HPDC*, pages 97–104, 2000.

[78] Antoine Trad and Damir Kalpic. Proactive Monitoring and Maintenance of Intelligent Control Systems: A Design Pattern for Autonomous Systems. In *26th International Conference on Information Technology Interfaces ITI2004*, pages 637–642, 2004.

[79] Giuseppe Valetto and Gail Kaiser. A case study in software adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 73–78, New York, NY, USA, 2002. ACM.

[80] William E. Walsh, Gerald Tesauro, Jeffrey O. Kephart, and Rajarshi Das. Utility functions in autonomic systems. In *ICAC '04: Proceedings of the First International Conference on Autonomic Computing*, pages 70–77, Washington, DC, USA, 2004. IEEE Computer Society.

[81] Danny Weyns, Robrecht Haesevoets, Bart Van Eylen, Alexander Helleboogh, Tom Holvoet, and Wouter Joosen. Endogenous versus exogenous self-management. In *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, pages 41–48, New York, NY, USA, 2008. ACM.

[82] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An aspect-oriented approach to dynamic adaptation. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 85–92, New York, NY, USA, 2002. ACM.

[83] Andreas Zeidler and Ludger Fiege. Mobility support with rebeca. *icdcsw*, 00:354, 2003.

[84] Ji Zhang and Betty H. C. Cheng. Specifying adaptation semantics. In *WADS '05: Proceedings of the 2005 workshop on Architecting dependable systems*, pages 1 7, New York, NY, USA, 2005. ACM.

[85] Ji Zhang and Betty H. C. Cheng. Model-based development of dynamically adaptive software. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 371–380, New York, NY, USA, 2006. ACM.

[86] Ji Zhang, Betty H.C. Cheng, and Heather J. Goldsby. Amoeba-rt: Run-time verification of adaptive software. In *Proceedings of the International Workshop on Models@run.time as part of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MODELS 2007)*, Nashville, TN, USA, October 2007.