HOW TO MAKE RUNTIME ARCHITECTURAL DESIGN DECISIONS

ANNE-MATTIJS KERSTEN

University of Groningen

Master of Science Graduate Thesis

Software Engineering and Distributed Systems

October 2010

Anne-Mattijs Kersten: *How to Make Runtime Architectural Design Decisions,* Software Engineering and Distributed Systems, © October 2010

SUPERVISORS:
Dr. Anton Jansen
Prof. Dr. Ir. Paris Avgeriou
Prof. Dr. Wim Hesselink

ABSTRACT

The software architecture of a system can be seen as a set of decisions made by the architect that incrementally construct the system from its inception to its latest revision. These decisions necessarily start at a high level of abstraction (e.g. what architectural style to use) and progress to define the lower-level details of the architecture (e.g. the refinement of the internal structure of a specific architectural component). The dependency between these decisions is such that those taken earlier on will influence, enable or possibly constrain those taken subsequently.

While concise recording of all design decisions has been shown to decrease design erosion and improve understandability, a review of current design decision models shows that these only allow for limited dynamic architectural reconfiguration at runtime. This falls short of meeting the needs of software systems which must adapt their architecture to changes in external conditions while they execute.

This thesis focuses on the way in which architectural design decisions can be codified and modified during the runtime phase of a software system. Instead of a static chain of architectural decisions an alternative is designed and implemented which allows for the runtime traversal and modification of that chain. The impact of this additional flexibility on the underlying component and connector model of the architecture is described.

The goal is achieved through the extension of an existing architecture-aware programming language, compiler and runtime environment called Archium. The Archium language is a superset of Java that features first-class design decisions and other architectural constructs and applies decision semantics to an architectural model based on components and connectors.

The implemented extensions consist of adding functionality to load and unload design decisions at runtime, with an intermediate checkpoint layer that provides dependency tracking and higher order operations. The Archimon graphical interface is also extended to provide a graphical means of interacting with the new functionality. A number of challenges encountered are discussed along with solutions for overcoming them.

The results of the extensions to Archium are evaluated using an example implementation of a software system. The thesis concludes by discussing the achievement and possible future work in this field.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth

Computer science is no more about computers
than astronomy is about telescopes.

— Edsger Dijkstra

ACKNOWLEDGMENTS

I would like to thank first and foremost my thesis supervisor Anton Jansen for his exceptional support without which the completion of this thesis would not have been possible. Through his guidance in our many discussions I have learned much about software architecture and engineering and have overcome many challenges (including some very much of my own making). I would also like to thank Prof. Dr. Avgeriou and Prof. Dr. Hesselink for their supervision and flexibility in allowing me to complete this thesis on my own terms.

Above all I am indebted to my Ania for her unwavering support and patience throughout this endeavor.

CONTENTS

LIST OF FIGURES

LISTINGS

ACRONYMS

ADL Architecture Definition Language — analogous to a programming language, a language used to specify a software architecture

API Application Programming Interface — the standard set of functions exposed by a software library

AST Abstract Syntax Tree — a simplified representation of the parsed source code used internally by a compiler

ATAM Architecture Tradeoff Analysis Method — a risk mitigation process that may be used in the early stages of developing a software architecture

C2 Chiron-2 — an architectural design and development style

EJB Enterprise JavaBeans — a managed server-side component architecture for modular construction of enterprise applications

ERP Enterprise Resource Planning — a system used by businesses to track resources, finances, functions and other business information

IDE Integrated Development Environment — a feature-rich toolset used to develop software

Java EE Java Platform, Enterprise Edition — a widely used platform for server-side programming using the Java programming language

MVC Model-View-Controller — a design pattern at the component level of an architecture

OMS Order Management System — the part of an ERP system that tracks all activities related to order taking and fulfillment

RMI Remote Method Invocation — the Java object-oriented equivalent of remote procedure calls which enables processing over remote links transparently

UML Unified Modeling Language — an industry-standard object modeling and specification language

VM Virtual Machine — the virtual environment in which Java programs execute their bytecode

XML Extensible Markup Language

Part I

THESIS

1 INTRODUCTION

As the field of software engineering has matured over time the size and complexity of software systems has risen to meet the increasing demands placed on them. This has introduced several problems which today remain only partially solved.

Software architecture methodologies have attempted to bring structure to manage the added complexity by enforcing a mapping step between high-level requirements documentation and low-level programming. However, as the expected lifetime of software systems has lengthened to match the increased investment in them this has introduced an additional set of difficulties.

One of these is architectural *erosion*, in which an architecture becomes increasingly less receptive to change after many individual sets of changes have been applied to it to meet evolving requirements. No change is perfect, and therefore every change made to an architecture tends to push it further down an increasingly narrow path. After a certain amount of time a partial or complete rewrite is usually needed to accommodate the next new requirement that is to be added to the system's scope. Examples of this effect can be seen in many (if not most) large software packages that stay relevant over long periods of time, whether proprietary (Windows 95 being superseded by the Windows NT platform in the 1990's) or open source (Linux kernel version 2.2 retired while the newer and redesigned versions 2.4 and 2.6 are actively developed).

The second challenge, *vaporization* of accumulated architectural knowledge which tends to occur over time as those working on a software system leave an organization. This makes it yet more difficult to design and implement changes to a system in a predictable way without affecting functional (does it do what it should?) and non-functional (does it work reliably, securely, quickly, etc.) attributes of the system.

Erosion and vaporization may have similar effects on the quality (maintainability, understandability etc.) of a software architecture but should not be conflated as being the same thing.

To further complicate matters, there is more need than ever before for software systems that can grow robustly to meet changes in requirements, many of which were not at all foreseen when the system was originally designed and built. An example are mobile systems that now travel farther and are expected to act more autonomously while deployed such as the Mars Rover. Other systems simply cannot be taken out of operation for an occasional change to be made due to their mission-critical nature, such as safety control systems in power plants and traffic control. Lastly, with the increasing pace of business in a global economy software systems are expected to keep up with external change by offering more adaptable behavior themselves.

To address these difficulties a number of approaches have been proposed by the research community. One field of research has focused on how architectural methodology can be improved to address erosion and vaporization. A number of toolkits have been implemented that aim to support software architects by storing architecture knowledge and allowing better tracking of the changes to an architecture as these are designed and implemented. Alternatively, solutions have been sought for integrating such information directly into the architecture itself.

This can be done by enhancing the tools and abstract representations used to construct the architecture so that information can be embedded within it in a consistent way.

The other side of the coin has sparked its own field of research, that of dynamic runtime reconfiguration of software architectures. This approach allows a change to be made to the architecture of an application without having to take it offline. This implies continuous availability while applying the change as there is no need to stop the operation of the system. It also allows for a more flexible approach to the software lifecycle and the transition between its stages of analysis, evaluation, implementation and maintenance of the architecture. In this way a system can be more adaptable to changing external conditions and requirements.

The two paradigms of constraining architectural methodology to ensure change consistency while introducing the freedom for such architectures to reconfigure themselves at runtime may appear to be contradictory at first glance. There is however an approach that has shown promise in attempting to provide both at once. The architectural concept known as a design decision attempts to structurally capture the knowledge used in designing a software architecture through to its current state. At the same time it provides a framework for describing potential changes that may be made to the architecture.

This is done by representing every major decision made in the construction of an architecture as a first-class object. Such an entity can have information describing the reasoning behind the decision stored within it, and can expose a set of operations to the tooling environment to allow for easy manipulation by the architect.

The design decision approach has thus far been extensively applied only to the first of the two areas. This thesis intends to explore a solution for the second of the two, namely leveraging the design decision approach to allow to architectural change during runtime. An existing design decision-based toolset and runtime platform called Archium has been selected for extension. It is a Java-based platform that other applications can be implemented on top of to take advantage of its features.

It currently does not allow for extensive architectural changes to be applied at runtime. This functionality will be added to allow the decision entities to be manipulated during runtime to the extent that decisions can be undone (or applied in the reverse direction) while providing uninterrupted availability of the application running on Archium.

The current design decision model implemented in Archium works like a forward-only chain where decisions are made sequentially and cannot be reversed or changed after the fact. This model is extended to provide full-featured repeated *undo* operations and dependency resolution. As a result the chain of possibilities becomes a tree where various branches can be explored with the possibility of returning to the main trunk.

The changes to Archium bring with them a number of other challenges relating to the integrity and operational aspects of the system. They are evaluated to determine the impact to the existing concepts and implementation of the system. This is done using an example implementation of a software system that uses these constructs.

1.1 THESIS OUTLINE

The chapters of this thesis are structured as follows. In Chapter 2 the central problem statement and related research questions are postulated. Next, Chapter 3 surveys previous research and existing solutions in this problem space. Chapter 4 provides an analysis of the semantics of manipulating design decisions at runtime, the requirements the toolset must meet and the problems that need to be overcome. Based on this, a sample implementation of runtime design decision manipulation is given in Chapter 5. In Chapter 6 a working system is built using the implementation given in the previous chapter followed by an evaluation. Subsequently in Chapter 7 the solution is compared to existing research to identify its merits and shortcomings and contrast its approach with that which others have taken. Finally, Chapter 8 concludes this thesis and describes future work to be done in this domain.

2 PROBLEM STATEMENT

2.1 INTRODUCTION

This chapter examines the concerns related to this area of research in order to define the scope of this thesis, and ends with the stating of the research questions.

The motivation for this thesis finds its origins in the intersection of the design decision and dynamic software architecture domains. The design decision model of software architecture is able to capture architecture knowledge and make it available to stakeholders in the various phases of the architectural process. Using the design decision model as a stepping stone to the ability to manipulate an architecture during runtime provides a powerful solution in this growing research area. This model is described in more detail in Section 3.4.

In scenarios that call for systems with dynamic architectures, these need an efficient method to control the changes they must undergo to meet user needs. The central question is how to make use of the design decision model in order to enable such runtime architectural modification – hence the problem statement of

How to make runtime architectural design decisions?

2.2 MOTIVATION: INTERCONNECTING ORDER MANAGEMENT SYSTEMS

The following example of many possible ones from the business domain can be used to motivate and evaluate the application of the described methods to practice. It relates to an order management system as used by virtually any sizable business.

An Order Management System (OMS) is a major component of the Enterprise Resource Planning (ERP) infrastructure in most medium and large sized businesses. It is a good example of a loosely-coupled yet continuous availability system, because it's used to track (among other things) all orders placed by the customers of a business. This usually needs to happen from the moment they are created by the customer (whether logged via phone or entered directly through an online client portal) through to the packing and shipping of the order, and the processing of the invoice and the financial transaction behind the order. As such it is always a highly critical part (if not *the* most critical part) of the IT systems in the business: the cash flow and profits of the company depend directly on the correct and timely processing of the orders logged into its order management system.

As any other system, order management systems undergo architectural changes throughout their lifecycle to meet new business requirements. One such example that is increasingly common is the need to link multiple order management systems together. The systems are usually owned by companies that are in a buyer/supplier relationship, and doing so allows them to decrease the time it takes to deliver an

order to their customer. A scenario might look as follows: a customer places an order for a wooden desk with a furniture company, for which it requires metal components (such as hinges and handles) from a metal processing factory – the OMS allows this transaction to complete without any manual intervention. The factory is automatically notified of the need for the components, which it can then deliver to the furniture company just in time for their assembling into the finished product (the desk). Another is that the buying company (e.g. large supermarket chain) is simply reselling a manufacturer's product to consumers if the manufacturer does not have this capability itself. In this scenario it is also a large benefit to connect the two order management systems if the volume of orders is large enough. The automated data flow of order information in one direction and invoicing and inventory data in the other dramatically speeds up the interaction between the companies. It may for example allow orders to be delivered the day after placement which in turn makes it possible for the supermarket to keep a minimal inventory and thereby save money.

By the very nature of the many types of businesses in the world, there are a multitude of order management systems, along with even more ways to interconnect them. The technology platforms in use by companies ranges from old mainframe systems (still used in many large and mature businesses) to modern client-server systems and middleware platforms. The interconnection mechanisms in use therefore vary as well. For example, on the mainframe platform the *flatfile* is a common mechanism, whereby one application writes the full dataset it wishes to share to a file in one of a variety of non-standard formats. The recipient application can then read in the data, as long as it is aware of the format used by the sender. In more recent client-server based platforms data can be transmitted directly over a network connection, in a way that is on-demand and includes only the required data in a standard format.

2.2.1 *Benefits*

Today, businesses face significant challenges when adapting their order management systems to each other. The investment needed in terms of time and resources is generally high when implementing a new interconnection mechanism with a new business partner. Here the problems described in Chapter 1 come into play. Especially in the older systems that are hosted in a mainframe (which may be already running for 20 years or longer), the amount of effort required to design changes to the system is often increased due to knowledge vaporization as former architects have left their positions. This necessitates a longer testing phase to ensure that the changes have been implemented without any bugs remaining. Having to do this multiple times with multiple business partners spurs design erosion of the overall system that can quickly lead to performance issues – or worse.

At the same time, market pressure on businesses dictates the need for ways to adapt faster to changes in partnerships and strategic direction. It is no longer acceptable to require 6 months to integrate two order management systems or implement an upgrade. Any such delay may directly impact the time to market of a new product or service – which cannot be brought to market without the ability to manage the orders.

The business is therefore faced with the opposing constraints of needing to decrease the length of implementation and testing cycles (hence speeding up the architectural lifecycle) and having to deal with more vaporization and erosion issues as time goes on. This is where a system built on runtime architectural modification can provide a solution. The ability to reach back to structured and well-documented design decisions taken previously in the development of the OMS mitigates the vaporization of architectural knowledge. At the same time, it allows changes to be designed and implemented much faster and with more confidence that they will lead to the desired 'correct' behavior.

Today the usual way of dealing with such performance issues arising out of design erosion and the resulting patched quilt of systems is to to throw more hardware or duplicate systems at the infrastructure. This comes at tremendous cost to the business, which could be saved by mitigating the erosion and ensuring there is no stale and performance-reducing code left running in the system.

An additional benefit and important consideration for these high availability applications is that the effort required to roll-out the design change from the testing phase to the actual 'production' system in use by the business is decreased. This is because the production system does not have to be halted or shut down so the changes can be implemented while it remains running.

2.2.2 *Risks*

There are a number of risks that need to be kept in mind in the business scenario described above. For instance, it is important to ensure that the benefit of decreasing the length of the implementation phase is realized without increasing testing effort. An inherent risk in architectural modification at runtime is that the added complexity needed to realize it may bring an increased risk of system instability and failure.

An example of this is the notion that systems that undergo architectural change at runtime are less predictable than those which do not. Outside of academic research (which is described in Chapter 3) no experience has yet been built up to ensure that the methods for testing these systems are adequate, let alone in industry. For example, classic unit testing may not provide enough coverage to ensure predictability versus more intensive run-time testing. This means it will be seen as a risky technology to adopt and any business will look for a certain level of guarantee that the risk has been mitigated.

This can be done by adding safeguards such as a strongly 'typed' layer that ensures that the system's architecture remains internally consistent while undergoing change at runtime. However, adding such functionality will likely impact the performance of the system which must be taken into account when considering this solution. Only in this way can businesses be reassured that system stability is not impacted by the additional complexity, and that the testing effort is not significantly increased due to adopting this technology.

2.3 TRIGGERING RUNTIME DECISIONS

In order to further determine the scope of the research questions, a distinction can be made between automating the *process of deciding* which design decisions to make or change, and the infrastructure

necessary to execute such changes during runtime. In the former area there generally are two triggers for a runtime change in the architecture: the user (an explicit, external influence) or the system itself (an implicit, self-regulatory influence). For the latter, tools and techniques from the control theory and artificial intelligence fields can be used to shape the decision making process. An example is a closed-loop feedback-control mechanism[12] which works by continuously monitoring the execution of the system using a set of predefined criteria. If the measured value for a criterion (which in such a scenario could be the reaction time to a certain event or the processing speed in a pipe-and-filter style) falls too far below a threshold a decision is made as to how the architecture should be changed. This decision is then executed by the system.

For the purpose of this thesis such processes for making intelligent and automated choices are not considered. Instead the focus is on how a design decision-based architecture can be modified correctly, with the assumption that user demands control the decision process.

2.3.1 *Cost-effectiveness of adaptation*

Generally the benefits gained from a change made to the architecture must outweigh the costs associated with making the change. Costs include the performance and memory overhead of monitoring system behavior, determining if a change would improve the system, and paying the associated costs of updating the system configuration. As a secondary consequence of excluding automated decision-making, automated evaluation of the cost-effectiveness of system changes will also not be taken into account. These are considered as being in the scope of the architect planning and executing the changes.

2.4 RESEARCH QUESTIONS

The problem statement of this thesis can be broken down into a number of research questions. A clear concept is needed of what a design decision is in order to implement a system that can manipulate these at runtime. Some of its characteristics will be important when treating a design decision as a first-class entity while others are less relevant. Therefore the first research question posed is:

1. What are *architectural design decisions and how can they be represented?*

There are multiple ways of achieving this and it is instructive to review other methods that have been developed for this or similar goals before attempting an implementation. A survey and review of relevant literature is captured as the research question

2. What approaches exist that allow application of design decisions to a running system?

Once that has been established we come to the central question of what stands in the way of a successful fusion of the concepts of design decisions and software adaptability:

3. What are the challenges in applying design decisions at runtime?

Taking the output of these questions gives sufficient basis for describing technical requirements and proposing an implementation based on Archium. This is captured in the research question

4. How can a system be implemented that allows runtime application of design decisions?

In order to make the results of this thesis relevant to the real world, the behavior and performance of the proposed implementation must be examined. Their utility in a business scenario can indicate whether this research is set to advance beyond the boundaries of academia. In short, the final question is

5. How well does such a system perform?

The research questions stated above are discussed and answered in the remainder of this thesis.

3

RELATED WORKS

In this chapter previous research is surveyed and relevant solutions that have been proposed are described.

3.1 SOFTWARE ARCHITECTURE MODELS

The concept of software architecture has been the subject of research since the late 1960's. As software systems grow larger and more complex, the necessity of documenting their structure becomes apparent. Doing so makes the subsequent task of modifying the software easier as the needs of its users change with time and provides an effective way to transfer knowledge between software architects. As a result, a documented software architecture is now seen as a key artifact in the software development lifecycle that describes the high-level structure of a system[16].

Documenting a system's structure is generally accomplished by partitioning the system into discrete elements and describing the relationships between those elements. In this way, a software architecture document can be used as an input into the later development stages where the design and implementation of the individual parts is fleshed out. There are many ways to define a software architecture and, as of yet, no full consensus has been reached within the software architecture community as to which is generally most useful.

The most prevalent contemporary model used in research, and the one this thesis is based on, is by way of *components*, *connectors* and their arrangement into *architectural configurations*[31]. This model describes an architecture at a higher level than most other architectural models (such as the often-used "4+1" model described by Kruchten in [26]) and is therefore sometimes termed a *meta-model*. The elements which it partitions a system into to form a model are described in the following sections. Figure 1 shows a diagram of an example component and connector architecture.

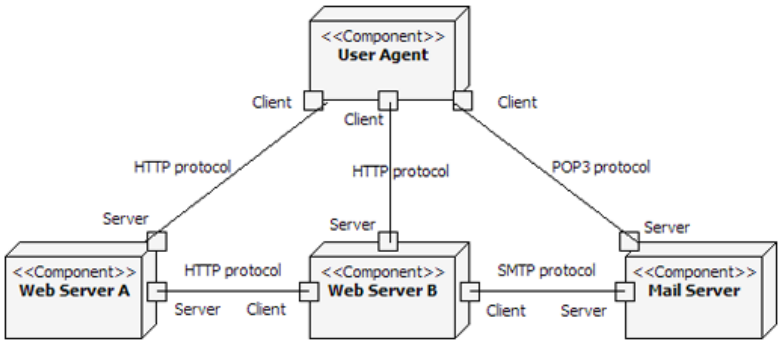

Figure 1: An example component and connector architecture for a web-based system in UML 2.0 notation, showing the protocols used for communication between components. (Source: http://www.restlet.org/documentation/1.0/images/tutorial04)

Components

The basic building blocks of an architecture using this model are the components, which are self-contained units of computation or data storage. A component may vary in scope and granularity from a procedure in a library to an entire application, but generally has a single goal it strives to achieve (or from an alternative viewpoint, a single service it provides to the rest of the architecture). Some examples of a component could be a simple client to interact with a remote web server or a large database subsystem to store all the data the software system needs to manipulate. As components are heterogeneous and independent of each other, they may be implemented in various programming languages and may use different frameworks and libraries to achieve their purpose.

As a consequence of the high level of independence between components, they must have a clear interface through which they communicate with the other components in the architecture and the outside world in general. Such interfaces are usually described in terms of the external functionality the component *requires* to operate correctly, and that which it *provides* to others. The component expects the former to be provided by another (set of) component(s). These two kinds of interfaces can further be broken down into *ports* which are the named points of entry and communication of a component. Other than providing an explicit access method ports also describe the accepted data types and format of communication the component expects.

The bundling of ports into interfaces allows the architect to work with components in the abstract and construct an architecture simply by linking them together. In essence, the components can be treated as black boxes, which is especially useful when the architect does not know anything about their internal composition or implementation. This is often the case when the component is supplied by a third party or when its implementation has not yet been completed. Despite this it is in theory easy to verify whether two components will be able to work together by examining their respective required and provided interfaces and the ports therein and determining whether they match. The black-box approach additionally allows for the freedom to replace one component with another in the architecture as long as the new component adheres to the same interface as the one it replaces. The resulting reusability is a powerful feature of the model.

Connectors

The communication between the components is what allows them to work together to achieve the overarching goal of the architecture. The glue between components is provided by another element called a connector that separates communication from the computation taking place within the components. Connectors are classified as a separate architectural element due to their complexity – they may do much more than simply linking a provides interface with a requires interface. A connector may for example:

- Connect two components separated by a large physical distance by transparently redirecting communications over a network connection;

- Translate between different communication protocols and data formats expected by the components; and even
- Perform functions such as buffering and synchronization of data streams which allow the bridging of two originally incompatible components' interface specifications.

In practice, connectors may manifest themselves in ways ranging from a pointer to a shared memory buffer to a complex set of networking library calls. In general, connectors are usually seen as being less complex in terms of computation and state in comparison with components, which justifies their separate classification. However, some research disagrees with this viewpoint and instead treats a connector as a special case of a component (for example the "everything is a component" model by Cuesta et al. [13]).

Architectural configurations

The way in which components and connectors are combined together to perform a task at runtime is termed an architectural configuration which can be represented by a connected graph. The configuration shows which components can communicate with which others in the architecture. It therefore allows for reasoning about the behavior of the system as a whole as well as derived attributes of the architecture such as performance, reliability, security, and so on.

3.2 LIFECYCLE OF AN ARCHITECTURE

Throughout its useful life an architecture can be said to move through a lifecycle of phases. Such a model is described by Tang et al. in [40], where the phases correspond to the activities that are undertaken by the architects. The first three phases of *analysis*, *synthesis* (i.e. the construction of the architectural model) and *evaluation* are core architectural activities. The latter two phases of *implementation* and *maintenance* are linked to the lower-level design and implementation of the functionality within components and connectors. It may be necessary at any phase to return to the analysis activity when for example new architecturally significant requirements are identified that must be incorporated.

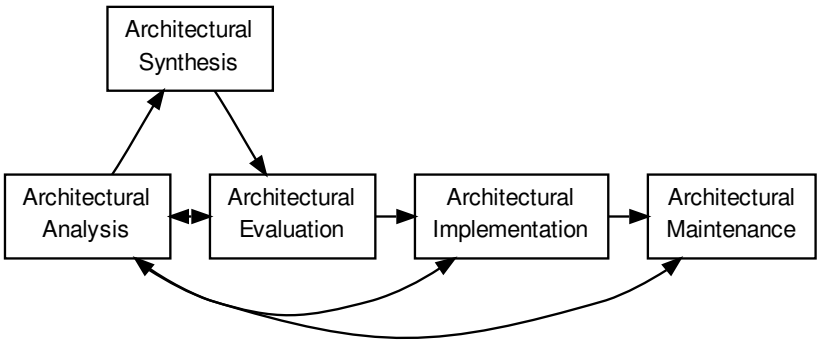

Figure 2: A model of the lifecycle of an architecture. Source: [40]

One of the characteristics of the architecture lifecycle is that the further along an architecture is in the process, the more costly (in terms of time, resources and therefore often money) it is to make a

change to it. Clearly, while in the analysis phase a change might be a simple addition to or removal from the list of requirements, a change in the maintenance phase will invariably involve many development and testing activities to actually implement the change and ensure it operates as designed.

Therefore, *if* the design decision model can make it easier for a business to introduce a change at any point in the lifecycle of an architecture, this alone would result in dramatically lowering the cost of the change.

3.3 ARCHITECTURE KNOWLEDGE

The body of architectural knowledge surrounding a system is more than just its architectural design

Once an architecture has been defined for a software system it should be codified. The resulting design document can be used for a number of purposes, including:

- Communicating the architectural design amongst architects, developers responsible for the implementation and other stakeholders;
- Retaining knowledge of the architectural design in the event the original architect(s) leave their roles;
- Verifying the correct operation of the software system, either manually or by an automated process supported by formal reasoning methods; and
- Assessing the quality of the system in advance of implementation by deriving quality attributes from the architecture.

Having a documented architecture also means no reverse engineering process is needed when a change needs to be made at a later time. Unfortunately, that process is still needed all too often when new architecturally significant requirements become known.

Additionally, past experience in research and industry shows that despite the presence of a documented architecture software construction projects are still prone to failure. While the pessimism of widely-cited reports such as the Standish Chaos Report and the Robbins-Gioia Survey – which report very low project success rates – is sometimes called into question[17] the fact remains that a significant proportion of software projects overrun their budget or time constraints, or underdeliver on scope or quality.

Software architecture knowledge is an area of research that aims to capture the body of knowledge that surrounds any architecture. This is desirable first and foremost to enable the understanding and reevaluation of the architectural process over time and by different groups of stakeholders. Its enables an existing architecture to evolve and deal with changes in scope, functionality and quality criteria and allows for a smoother handover between the various architects working with it.

The first three phases of the architecture lifecycle (as shown in Figure 2) are where software architecture knowledge (hereafter referred to as AK) is generated. Clearly, its capture and retention in those early phases becomes invaluable throughout the later phases of the lifecycle that are linked to detailed design and implementation of the components and connectors and subsequent maintenance activities. Less errors are made during the later phases if the architecture and the constraints it imposes are well understood[15].

Failure to capture this contextual information is firstly the cause of knowledge *vaporization*. This is simply the loss of knowledge that occurs over time as people shift their focus to other tasks or leave their roles in a team and as their organizations undergo other changes[22].

Secondly, architectural design *erosion* is also spurred in its absence. This term refers to the aging process inherent in the architecture of a system as it moves through the lifecycle phases. A result of the many small changes made to it over time is that the architecture becomes less flexible. Due to this aging the architecture leaves an increasingly narrow degree of freedom to meet the needs of new requirements as they become known[42]. Unfortunately, the older and more established an architecture is, the less likely it is that it will be overhauled (via a number of larger changes) and cleaned up by those that work with it – simply due to the significant investment of resources (such as time and funding) it represents at that point. Even though all architectures must live with erosion, the ability to capture more contextual knowledge makes it easier to identify the best possible change that can be made to an architecture to meet new requirements while minimizing erosion and associated future costs.

3.3.1 *Knowledge retention strategies*

In order to gain a clearer understanding of the bounds of AK, general knowledge management theory can be applied to the architectural scenario. As described by Hansen et al. in [20] the approach a group or organization takes towards managing knowledge can be classified at a high level as either a *codification* or a *personalization* strategy. The codification strategy is most familiar to software engineers, as it focuses on storing knowledge in a central repository and on making that repository as accessible as possible to all stakeholders. This does however require the body of knowledge to be pliable into a structure so that it can be stored and retrieved in an organized way. While it promises easy and quick reuse of knowledge, it in effect separates that knowledge from its source (in most cases the people it originated with) and it limits the amount of knowledge a knowledge-seeker may find to what's contained in the repository.

At the other end of the spectrum, personalization strategy aims for the subject experts to retain their knowledge and to match the knowledge-seeker directly with the originator by accurately recording and publicizing *who knows what*. The body of knowledge itself is not directly stored. An example is the formation of designated communities of practice within an organization. The effort required to encode knowledge into a repository (which can be a barrier to knowledge sharing) is removed. Furthermore discussion between providers and seekers of knowledge is stimulated which may raise the level of quality of the body of knowledge (this is however counteracted by the fact that there is no structured retention in place).

There is not much mention of personalization strategies in current AK literature in contrast to conventional knowledge management literature[29]. This may be due in part to the fact that personalization abstracts away the type and structure of knowledge, so that no AK-specific approach is needed.

As described in [5], many industry organizations do not have a rigorous knowledge management system in place and often unintentionally

personalize AK. While that approach can have benefits as explained above, the need for codification and therefore documentation of AK is becoming apparent. Avgeriou et al. however make mention of the need to make sure that a codification system supports whatever personalization is in practice in an organization, and explore best practices and anti-practices for the two approaches[2].

3.3.2 *Models of architecture knowledge*

Several models for classifying codified AK have been proposed, which put together give a good view of the scope of the body of knowledge that can exist outside of a component and connector-based architectural specification. These models are mostly orthogonal to each other and give complementary views.

3.3.2.1 *Explicit and/or documented*

Kruchten et al. categorize AK according to whether it is *implicit* or *explicit*, and *documented* or *undocumented*[27]. The first distinction is between knowledge that the architect is not conscious of or considers so basic it doesn't need to be recorded, and knowledge directly related to the situation at hand. For example, the decision to use Java instead of another implementation language could be considered implicit (it was dictated by company policy and should be obvious to those who work there) or explicit (Java provides automatic garbage collecting which is a critical feature for the problem domain in which the architecture operates).

Clearly, undocumented knowledge is very likely to vaporize over time until it is completely lost – for example due to architects changing to a different role or simply shifting focus to a new project. On the other hand there may be valid reasons to keep AK undocumented such as a company's policy or internal politics. Dutoit and Paech [15] describe a scale that measures the level to which AK is documented, starting from no capture through to after-the-fact reconstruction and unstructured capture, up to full integration of knowledge collection into the development lifecycle.

3.3.2.2 *System vs. process*

Dutoit and Paech argue in [15] that AK can be grouped into knowledge that is either directly related to the *system* which is under construction, or knowledge about the *process* required to develop that system. The two types of knowledge both cover the what *and* the why (rationale) of the system and process respectively. The audience for the former kind are clearly software architects and engineers, while the latter (which includes roles, tasks, resources) is also used by project managers to track progress of projects. Both of these groups can be further subdivided according to the scope of use, which can be either *product*-specific or *organization*-wide. The latter not only embodies broad knowledge of the software engineering discipline but also the models, patterns and best practices developed within a particular organization.

3.3.2.3 *General, context, rationale and design*

Tang et al. describe a different classification system that and divides AK into the following set of elements that can be linked to the phases of the architecture's lifecycle[40]:

- *General knowledge* consists of globally reusable design artifacts such as styles, patterns and tactics that are not tied to a particular architecture or system;
- *Context knowledge* embodies methods and requirements specific to the problem space of the current architecture;
- *Reasoning knowledge* are the decisions taken by the architect(s) during the construction of the architecture, and their rationale and discarded alternatives; and
- *Design knowledge* comprises the architectural views and models which are the outcomes of the architecture process.

While all types are used throughout the lifecycle phases, general and contextual knowledge are partially derived from pre-synthesis activities such as requirements engineering and are primarily used by the architect in the analysis and synthesis phases. Reasoning knowledge is used in the synthesis phase to develop the design knowledge that comes with a complete architecture. Design knowledge also lies on the boundary of AK as it will be used during synthesis activities such as component design and implementation. During the implementation phase design knowledge may be modified further while reasoning knowledge is referred back to to understand the architectural structure. In the maintenance phase design knowledge is used to evaluate proposed changes to the system.

3.4 DESIGN DECISIONS

The architectural analysis phase of the software development lifecycle is where all the high-level decisions regarding design are made, and understanding these decisions is critical to evolving a system throughout its lifetime. A novel approach to achieving this is to consider not only the architectural design document as a formal artifact, but also elevate the underlying decisions *that were taken during its creation* to a first-class formalism. As a result architectural knowledge can be redefined as the sum of these two components[27, 9].

While there has been debate over the scope of this formalism resulting in the formulation of several design decision models there is agreement that it is meant to cover not only the decision itself and its effect on the architecture, but also the *context* in which it was made and the *rationale* for the option chosen. The context could include all prior design decisions made before the current one, i.e. the current state of the architecture and the AK built up to that point. A rationale for a decision can stem from a wide variety of sources such as technical, business, process or organizational concerns. Tang et al. argue that this information should further be linked to elements such as stakeholders, processes and design artifacts, forming an interconnected network[40]. This for example allows a design decision to reference the stakeholder concerns connected with various options of which a single one is chosen after weighing their benefits. Exposing the relationships between these

entities naturally enhances the traceability between the design decision and all the factors influencing it[2].

Current research also makes mention of *design intent*[36, 3, 18], which describes the set of end goals an architecture strives to achieve. It can be argued that this is just another way of describing those things that motivate and drive all design decisions, in other words the rationale.

A recent approach to design decisions which this thesis is based on is to refine the model of software architectures themselves so that these are explicitly constructed out of design decisions. The decisions so become the glue that holds an architecture together. This model, postulated by Jansen and Bosch, takes the view that an architecture should be formally seen as the composition of a set of design decisions[23]. The design decisions build up the architecture from its beginnings to its current state. Along the way all architectural knowledge involved in the design is completely captured in the formalized design decisions and becomes an explicit part of the architecture itself[1].

Using this model of design decisions it ceases to be useful to view architecture knowledge as the sum of two parts – an architectural design and design decisions – as these are so intertwined in the model that they are inseparable.

A formal definition of the design decision concept will be given in Section 4.3.

3.5 SOFTWARE ADAPTABILITY

Since the dawn of the software age, those creating software systems have striven to make the behavior of their systems as adaptable as possible. Being adaptable means being able to serve a variety of different needs with a single piece of software. It means not having to create a new program for each and every specific need, but being able to rely on a library of software to solve not only existing but also new and previously unknown problems. The degree to which a software system exemplifies adaptability is shown in Figure 3. Software adaptability started out as asking the user for input at the start of program execution (e.g. via initial parameters), or while the program was already running. Depending on the input the program performed a different set of tasks. This most primitive form of adaptation relied on the development of hardware capable of executing conditional expressions, and is shown at the bottom of the figure. This feature is available in virtually every programming language via an *if–then–else* statement or its equivalent.

One step above conditional expressions (and at a higher level of abstraction) are online algorithms that are able to process their input in a serial fashion and make decisions at runtime based on the values of input received. This type of algorithm is designed in a way that removes the requirement that all future inputs are known at the time the program is started. It is therefore more adaptable to changing conditions than an offline algorithm (although in some scenarios the adaptability comes with the price of decreased performance).

Beyond online algorithms there are various degrees to which a software system can be made more adaptable, ranging from parameterized

1 This approach interoperates well with standard architectural views such as the 4+1 model defined in Kruchten [26], as such views then become windows on the set of design decisions, putting focus on their various features. Alternatively, design decisions can be seen an extra standard view in addition to the 4+1 model[11].

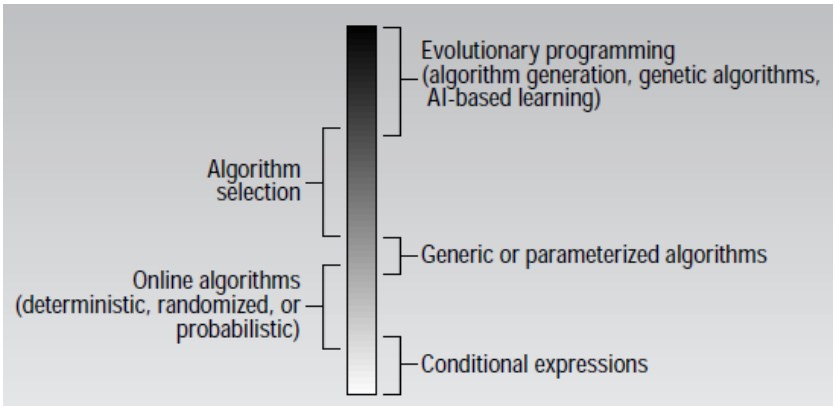

Figure 3: The spectrum of self-adaptability. Generally, approaches near the bottom select among predetermined alternatives, support localized change, and lack separation of concerns. Approaches near the top support unprecedented changes and provide a clearer separation of software-adaptation concerns. Source: [35]

algorithms which can behave differently on the same input data depending on the values of a set of parameters, up to using evolutionary programming to generate and select the most optimal algorithm from scratch.

Independent of which approach is taken to create an adaptable system, it is said to be *open-adaptive* if new application behaviors and adaptation plans can be introduced at runtime. A system is *closed-adaptive* if it is self-contained and not able to support the addition of new behaviors once it is in a running state[35].

The ability to make (and undo) design decisions at runtime should allow for open adaptability, as a design decision as defined above can describe an arbitrary complexity of behavior. The next section describes the utility of this approach as it forms the foundation of this thesis.

3.6 DYNAMIC SOFTWARE ARCHITECTURES

The evolution of a system's architecture during runtime is not a new concept and has been explored in scientific literature since the mid 1990's[28][34][30]. When a component and connector model of an architecture evolves during its lifespan this commonly means one or more of the following[30]:

A. *Adding new components* to the architecture, e.g. in order to meet new requirements or cope with a changing external environment;

B. *Replacing a component* with a different component for the purposes of upgrading the system, meeting new requirements or enabling an increase in performance;

C. *Removing components* which are not needed any more or in order to save system resources; or

D. *Changing the architectural configuration* by modifying the way in which components are connected to each other.

Although connectors have not been explicitly mentioned in the above, making the changes described above will also have an impact on the

connectors in the architecture and/or the ports and interfaces they are using.

Some of the reasons a system may need to change its architecture over time are the variable availability of resources such as bandwidth or external services it is dependent upon, faults within the system's own components or in its communication links to the external world which it must tolerate and changes in the wishes and requirements of the system's users. The flexibility that runtime architectural modifications bring are especially beneficial to certain classes of systems, which are briefly described below.

CONTINUOUS AVAILABILITY SYSTEMS These are the business-critical or mission-critical systems that must be in operation 24/7 and for which any downtime would have immediate hazardous or financial impact. Examples include safety control systems such as those found in power plants and air traffic control systems that monitor and direct flight patterns in real-time. Because offline reconfiguration is not an option, these systems will benefit from having a dynamic software architecture.

This has the added benefit of lowering the financial cost and risk involved with making a change to the system, as having to take it offline and subsequently bring it back online is avoided.

MOBILE AND DISTRIBUTED SYSTEMS These types of systems are by definition exposed to changing environments to which they need to adapt by way of dynamic reconfiguration[43]. For example, distributed sensor networks often have minimum downtime as a requirement so that their data collection tasks are not disrupted. A number of Architecture Definition Languages (ADLs) which have been explicitly designed to cope with these circumstances are described in [1].

SELF-HEALING SYSTEMS This is another category of systems that must operate continuously in harsh environments, which partially overlaps with those described above[37]. An example could be a robotic probe sent to autonomously explore a remote location.

LOOSELY COUPLED ENTERPRISE SYSTEMS These systems allow flexible connections between business partners in situations where their architectural structure is not known in advance[41]. Using dynamic reconfiguration businesses can adapt to changing market conditions with a shorter turnaround time, thereby decreasing time to market of their services and increasing profits.

What the aforementioned examples have in common is that from a quality attribute point of view, they require increased flexibility and availability over those of other software systems. There is inherent tension between these quality attributes and those of reliability and testability.

A number of frameworks and tools in this area of research have been devised in previous research. These range from tools for capturing AK through those linking it to a description of an architecture, to full-fledged implementations of runtime architectural reconfiguration. The design decision model is also used to a varying degree; in some, design decisions are first-class objects while the support for runtime application and/or change is limited, while in others no use is made of a design decision-based model at all.

Due to the cross-cutting nature of the problem statement of this thesis and varying state of existing solutions in terms of feature set on offer, examples from this broad spectrum of frameworks are selected and examined in this chapter. Their shortcomings are highlighted so that they may be taken into account during the development of a solution in this thesis.

3.7 APPROACHES FOR CAPTURING ARCHITECTURE KNOWLEDGE

This section examines implementations of knowledge capture that may support design decision retention, albeit not on a technically precise level.

3.7.1 *CBSP*

CBSP is an abbreviation of the four keywords component, bus, system and properties. It provides a so-called intermediate model that positions itself as bridging the gap between an initial rough set of requirements and an architecture by providing traceability links between the two.[19]

CBSP is as much a set of processes as a supporting tool. At its core is a strict and clearly defined process that guides a group of stakeholders through the architecting process. Referring to the lifecycle model in Figure 2 CBSP plays in the first three stages of the lifecycle and does not attempt to cover the entire cycle. The goal of the process is to identify architecturally significant requirements and to then derive from them an architectural description in the form of a set of component, bus or general system artifacts and/or properties.

The process can be summarized as follows. First the full list of requirements is filtered so that only architecturally relevant requirements remain, and of these a critical subset is chosen such that this set is easier to work with (CBSP lends itself to an incremental process and so other requirements can be processed in a later iteration of this process). Then, every requirement is closely examined and translated into one or more CBSP elements using a process of identification, mismatch resolution and refinement.

The outcome of this process is a CBSP model (or an intermediate version if it is one of a number of iterations) which is made up of a set of components, buses (an analog to connectors) etc. From this still fairly abstract model an actual system architecture can be derived by composing these CBSP elements together into a specific architectural style that has been chosen for the system in question. That architecture can be structured in any desired way, e.g. using the C2 layered architectural style. This last step illustrates the intermediate nature of the tool and its positioning between requirements and final system architecture.

While there is currently no tool that supervises this process from start to finish, several of the steps in the process have tool support. For example, in the second step described above individual contributions come together in a voting and discussion round intended to reach consensus before moving on. The aim is also to help the various stakeholders (including non-architects from business and management) to reach a consensus regarding the meaning of each significant requirement (e.g. is requirement X more of a general system-wide property, or is it actually a bus requirement?). This step of the process is tool-supported using a groupware platform with custom-developed extensions that

track input and voting and are specifically geared towards architectural discussion.

The CBSP method builds on the fact that it is not easy to go from a list of problems to be solved, associated constraints and desired features (in other words requirements) to a model that clearly describes the structure, properties and behavior of a system (the architecture). It is assumed that the requirements are in a form that is not easily processed and that is very far away from the architectural model.

The creation of a CBSP model according to the process outlined above involves making significant architecturally relevant decisions – some requirements may be discarded, others combined and modified. While these may not be full design decisions (they do not directly affect the architecture since the architecture itself is generated at a later stage), they nevertheless have a very strong influence on the shape of the architecture. One of the shortcomings of the method is that, although these decisions have adequate tool support (e.g. listing of voting scores along a set of predefined dimensions) their rationale is not explicitly captured.

Additionally, as can be understood from this summary there is no way of linking architectural elements with runtime execution, let alone dynamic reconfiguration. It is assumed that the architectural outcome of CBSP is processed with a separate set of tools.

3.7.2 *Web-based: ADDSS and PAKME*

The Architecture Design Decision Support System focuses on providing tool support with less emphasis on an underlying process. It is a web-based tool for storing design decisions, requirements and other related information[11]. It does not cover the architectural design itself beyond the uploading of images representing the state of the architecture at various points in time. It is possible to relate design decisions together and so form a tree or graph structure. Within that structure it is then possible to trace back from any design decision to the set of requirements it (in)directly originates from. A decision can make use of pre-defined design patterns or architectural styles which are defined in the system. It is also possible to use a free-form text field to enter rationale and other relevant information.

ADDSS does not reuse any components from others tools or platforms. While it lacks more sophisticated collaboration functionality, it is the only tool that allows for the setting of access controls on AK stored in the system, for example so that a project manager has access to multiple categories of knowledge while an architect only sees information relevant to him. Another strong feature is the tracking of historical changes made to data so that it is possible to view a timeline of the architecture and decisions as the system evolves.

The Process-centric Architecture Knowledge Management Environment (PAKME) uses web-based forms together with an underlying data model to capture architecture knowledge. This knowledge includes such items as design decisions, scenarios, risks and more[4]. While the knowledge model is quite extensive it does not model the architecture itself but only the AK connected to an architecture. Much like in ADDSS, the only way in which the two can be linked is via the uploading of images. The various elements of the data model are connected to each

other and provide traceability through hyperlinking. Every edit made to the knowledge model is tracked to provide history information.

PAKME can serve as a central repository of AK for an organization, and supports the recording of both project-specific knowledge and organization-wide reusable patterns. It is based on an existing groupware platform (Hipergate) which comes with teamwork functionality built in. This allows multiple and geographically dispersed stakeholders to collaborate on the knowledge base. The platform also brings functionality that supports personalization by tracking who entered which data into the system.

It is possible to specify a generic architectural pattern in PAKME which can then be reused as an answer to a design decision (these elements are referred to as design options). The degree of abstraction of design decisions can vary and they can make use of architectural patterns, design patterns and design tactics. The rationale of the decisions can be specified to great detail using the fields available.

PAKME also directly supports architectural evaluation methods such as the Architecture Tradeoff Analysis Method (ATAM), by recording evaluation finding and building result trees to visualize risks. In addition a collaborative decision-making component is being developed that will follow the principles of a wiki (another form of knowledge exchange).

3.7.3 *AREL*

The Architecture Rationale and Elements Linkage (AREL) system consist of a UML based model together with a supporting tool and aims to integrate architectures with design decisions[39]. It allows for describing design decisions in both qualitative and quantitative ways and its traceability allows for analyzing the impact of changes to the architecture as well as tracing a part of the design back to its original root cause. It intends to cover the entire architectural lifecycle from analysis through to implementation and maintenance.

The entities in the AREL model can be one of two different types: Architecture Rationale (AR) and Architecture Element (AE). The AEs can be anything from business requirements, technical/organizational constraints, assumptions, up to the actual elements of the final architectural design. ARs represent the decision points and fully encapsulate qualitative and quantitative rationale in addition to possible alternative outcomes of the decision (which can again consist of AEs). While the qualitative portion captures the usual elements of a design decision (trade-offs, strengths, the decision itself etc) it is also possible to record numeric data regarding cost, benefit and risk associated with the AR.

Every design decision is represented as a mapping from one set of AEs (the *motivational reason* that either motivates or constrains the design decision) to a second set (the *design outcome*), the two sets being linked together by one AR. By stringing multiple design decisions together in order to build a causal chain of relationships, an acyclic graph is created of which traceability is an inherent property. Both the ARs and the AEs are represented by UML objects to create a visual representation. There is also an extension to this model called eAREL which covers the evolution of an architecture by also retaining past information of the model as it changes and develops.

The tool support for AREL consists of a modification to the Enterprise Architect UML tool. In addition there is a separate custom tool

that allows forward, backward and historical tracing within a given model. Unfortunately, AREL does not explicitly support teamwork or cooperation between multiple stakeholders.

3.7.4 *Matrix*

The Matrix/CORE model [7] is an overarching model of AK that aims to provide a generic abstraction for various other models to interoperate. This is accomplished by defining a subset of elements that are present in most or all of the other models. A set of translation layers placed on top of the core model allows for the translation of architectures based on other models to be translated to the CORE domain. In this way multiple architectures originally defined using different models can be viewed using one model and the associated AK can be freely shared amongst them. The ultimate aim is to ease AK management across different architectures and collaborating organizations.

The CORE model is heavily based on design decisions and on the notion that the outcome of their summation is an architecture. Proposing and ranking alternatives to come to a decision is the primary activity undertaken by the stakeholders. Representations of the architecture and associated AK takes place in artifacts.

A central CORE concept is the decision loop which is iterated through as part of the architectural synthesis. A decision which has been taken may lead to a new architectural concern becoming apparent. Stakeholders must investigate these concerns, some of which may turn out to be decision topics. The alternatives for these must first be weighed before one is selected to form a new decision. The loop illustrates the relationship between the various design decisions, and the iteration through this loop will progress from coarse-grained decisions to more detailed decisions as these build on each other.

Rationale is not called out specifically in the CORE model but is an interwoven part of the cycle that links all elements together. Traceability is a subset of rationale, allowing upstream/downstream links between concepts in the model such as design decisions.

A newer project based on CORE, called GRID, takes the cross-model data integration qualities of CORE and builds a service structure on top using a grid infrastructure. Various organizations can use the common vocabulary of CORE to communicate with each other, together with central storage of design decisions and services (e.g. a word processor plugin) that allow easy access to the information contained within the model.

3.8 APPROACHES FOR RUNTIME ARCHITECTURAL RECONFIGURATION

In this section existing implementations of dynamic changes to a system's architecture are discussed. The solutions take various approaches to implementing a platform that the runtime reconfiguration is built on top of and with different feature sets.

3.8.1 *C2 SADL & DRADEL*

Some of the earliest attempts to construct a framework for dynamic architectural reconfiguration used the Chiron-2 (C2) architectural style as a foundation. C2 is an architectural style that is based on a component and connector model but which adds additional constraints on how these may be connected together in an architectural configuration. Connectors take the forms of buses, so that each component can be attached to at most 2 connectors: one 'above' it and one 'below' it. Furthermore, the architecture is message-based with connectors acting as message buses and components listening for specific messages and responding with their own, in a unicast or multicast fashion up or down the connector hierarchy.

Basing a reconfiguration framework on such a constrained architecture simplifies the implementation of a solution. In Medvidovic [30] such a solution is proposed in the form of C2 SADL. This ADL firstly provides the notation needed to describe a C2 architecture of components, connectors and the topology in which they are arranged together.

The ADL is then extended with a new set of keywords termed *ACN* for architecture construction notation. The ACN provides an Application Programming Interface (API) to the architect for accessing the architecture both during development and later during runtime. In the latter case, requested changes to the architecture are dynamically interpreted and executed. The operations possible include adding or removing a new component to the architecture, linking or unlinking it to connectors and disabling the ports on a component without removing it altogether. The functionality and internal composition of the components themselves cannot be changed.

In Medvidovic et al. [32] a further development is described called DRADEL which extends the above with a method for evolving individual components. This is made possible through an architectural type system reflexively implemented in C2 SADL. An architectural type not only specifies the interface of a component but also a set of invariants that describe its behavior. In this way an existing component can be safely replaced by another sub-classed component with guarantees that it will offer the same functionality - although it's implementation can be quite different.

3.8.2 *ArchStudio*

ArchStudio is an Eclipse-based multi-platform software and systems architecture development environment[14]. It is based on a design decision model. In addition it uses xADL 2.0 as modeling notation, which is an ADL based on Extensible Markup Language (XML). This is a modular and extensible ADL that can be used to describe not only components and connectors but also many other elements including architectural states and product-line concerns. The xADL 2.0 modules are defined in XML schema specifications which can easily be extended so that new ADL entities and features can be developed to meet the needs of a specific problem domain.

The ArchStudio toolset can be used for modeling an architecture, and also for visualizing and analyzing it. An architecture analysis tool allows for the running of tests on the architecture using a set of included

testing algorithms and also allows user-specified algorithms to be developed. These can test configuration semantics such as making sure all provides and requires interfaces are fully compatible and connected.

A number of ADL modules allow for the representation of dynamic reconfiguration. First of all the versioning XML schema tracks different versions of components and connector (and other object) types. The options schema allows for the expression of a point of variation in the architecture where objects can be either included or excluded, while the variant schema allows for the choice amongst a number of alternatives that can be substituted for objects such as components and connectors. The last two are clearly related to the more abstract concept of software product lines.

3.8.3 *Plastik*

In the Plastik framework[6], the authors have applied the principle of reuse by fusing two existing tools into a dynamically reconfigurable environment. The first of the two tools is an extension of the high-level ADL implementation named ACME, which comes with a set of code generation tools. It is not domain specific and can describe any architectural style that is based on the component and connector model. Via a compiler an input architecture is transformed into a set of architectural objects that can be instantiated at runtime. These objects are the well-known components, ports and connectors; as well as *roles* that describe a connector's functionality when coupled with a port and *attachments* that define a set of port/role associations. The attachments are what links the architecture into a unified whole. Components are considered to be composite objects, and their internal specification can be described in any number of alternate so-called *representations* that decompose the components into other components.

The ACME ADL additionally allows for the specification of constraints that describe limitations on how an architecture can be put together; for example that only a specific number of component types can be connected together in a specific way (e.g. when implementing a network stack). These constraints together with a set of architectural objects form an *architectural style*.

The second part of the equation is a reflective component runtime that enables the architecture to come alive. This is an extended version of the OpenCOM runtime. It describes very similar architectural objects as ACME does, and the authors have integrated them by creating a mapping from one to the other. For example in the runtime, an architectural style defined in the ADL is mapped to a *component framework* - which provides a level of abstraction in between an individual component and the whole system. A component framework is meant to represent a cluster of components that provide a focused scope of functionality to the system, for example a network stack implementation in a larger middleware system. An architecture is therefore composed using Plastik by composing and configuring an appropriate set of component frameworks. During runtime the component frameworks apply the constraints described in the architecture ensuring that the architecture is internally consistent and that architectural reflection can be used only withing those safe constraints.

The resulting meta-framework operates on three levels: the *style level* that defines constraints and abstract groupings of components, connec-

tors, etc.; the *instance level* where components are instantiated and given functionality; and the *runtime level* where the architecture operates. The architecture can be reconfigured based on rules implemented programmatically during its design and also at runtime. Architectural change is managed by an architectural configurator at the style and instance level and a runtime configurator at the runtime level. This implies there are two levels of runtime ad-hoc configuration. An issue is that these cannot be mixed together as causality is not enforced between the two configurators. This is likely due to the added complexity of allowing architectural change at two levels of abstraction using different supporting tools.

The runtime configurator can apply reverse operations to connectors by detaching them from roles and deactivating them altogether. Invariants are checked at the time of executing the change to ensure that they are not violated.

3.8.4 *PKUAS*

The Peking University Application Server (PKUAS) is a reflective and adaptive middleware platform that is meant to host applications based on a component model [33]. Like any middleware platform it provides services to software components and enables transparent communication and interaction between these whether they are located on the same machine or across a network. It is implemented in Java and conforms to the Java Platform, Enterprise Edition (Java EE) and Enterprise JavaBeans (EJB) standards. It is the first adaptive middleware platform to do so, which means a large number of existing Java-based applications should be able to take advantage of its features.

PKUAS aims to provide reflective power with a focus on usability towards the components it hosts. The reflection capability applies to the components themselves and also to the middleware platform – for both, the internal runtime states and behaviors are made observable and adaptable. This is possible because the middleware platform itself is also represented as a set of components and connectors. Instead of providing full visibility into the middleware platform's internal state however, a simplified view is made accessible to the application layer for manipulation.

In order to allow this manipulation to occur, PKUAS wraps all application components in a container. This allows for safe execution by ensuring a component behaves within the limits imposed by the system. It also means less adaptation is required to adapt existing applications to this particular middleware platform. The component can then be deactivated and replaced by performing the corresponding operations on its container, again ensuring overall system integrity.

The middleware services run alongside the containers. Here reflection is implemented by creating a 'meta object' for every service component that exposes its internal state. This allows for lifecycle management (stopping and starting the service, for example) and also more invasive operations such as adding, replacing or removing the service altogether. There are some limitations however – core components such as the communication service do not allow themselves to be altered or removed.

Technically, the adaptation during runtime is implemented using the Java classloading mechanism. By extending a default classloader

for either a service or an application component, modified code can be introduced to alter the behavior of the middleware platform. The platform as a whole runs on top of an immutable microkernel layer which provides the base class loaders along with timer and monitoring infrastructure.

3.8.5 *OpenRec*

The OpenRec framework as described in Hillman and Warren [21] is an open framework for managing dynamic reconfiguration. As with other implementations it allows the usual set of operations to be made on architectures using it, e.g the addition and substitution of components. In addition it provides additional infrastructure to allow for algorithmic reconfiguration of architectures at runtime.

This is done by providing a sample set of algorithms that can be extended as needed by the architect. The algorithms work by measuring the 'cost' of different adaptation choices in trying to match a requested change, and selecting the option with least cost. The cost itself can be composed of the time it takes to for example implement a change to a component; and the degree of disruption experienced by other components in the topology. The way in which the cost is measured is extendable by way of plug-in points in various areas within the reflective layer (termed the reconfiguration manager in OpenRec). The ultimate goal if this feature is to be able to determine ahead of making a change what its impact will be on the architecture's non-functional attributes such as availability, response-time, and throughput.

In practice the framework consists of three layers. At the top, a change driver accepts requests for changes and passes these on for processing. The actual change requests are formulated using an XML-based notation. These are sent to the reconfiguration manager which is the central reflective management layer that in turn manages the lowest layer of the stack – the actual architecture. The structure of this meta-framework allows the architect to not only make changes to the architecture itself during runtime, but also changes to the reconfiguration manager that defines *how* such changes should be implemented.

The reconfiguration manager allows the architecture to use structural reflection to inspect and change its own topology. Additionally using the extensible plug-in points mentioned above, so-called behavioral reflection is also possible. This is the monitoring of and the implementation of changes based on the architecture's behavior and the communication between components flowing through the connectors.

4

RUNTIME DESIGN DECISION REQUIREMENTS

This chapter explores the central challenges in applying the design decision concept as introduced in the previous chapter, at runtime. The Archium platform which will be used as a basis for implementing this functionality is introduced. Finally a set of requirements and constraints are derived that should be met in order to come to a usable implementation.

4.1 CHALLENGES FOR ARCHITECTURAL ADAPTATION AT RUNTIME

"what needs to be changed"

When considering an architectural model that consists of a set of design decisions, the application and reversal of these decisions is what shapes the architecture. The design decisions become the vehicle for architectural adaptation. The architectural adaptation is therefore not managed by an out-of-band entity that is external to the system, but entirely within the system itself.

When a change to an architecture is being considered, both structural and behavioral properties of the system need to be considered to understand the ramifications of the change. For example, the dependency between components and the services provided by each (both of which are described in the provides and requires interfaces) need to be understood if one or more of the components is to be replaced.

4.1.1 *Architectural reflection*

The architectural model will therefore need to allow inspection of the architectural structural information and its configuration – which components are connected to other via which connectors. Introspection of the design decisions themselves is also needed. Both these can be realized by way of *reflection*, the ability of the system to reason about and act on its own structure.

Reflective systems are generally composed of two levels, a base level and a meta level. These levels are linked and causally connected so that a change made to one triggers a change in the other. In the case of a design-decision based architecture, accessing a representation of a piece of the architecture and making a change to it should result in the actual structural modifications to the architecture. This can work in the following way:

A. The *base level* which in this case corresponds to the actual components and connectors in the architecture

B. The *meta level* offers an interface that can be called during runtime, and offers two complementary types of reflection: structural and behavioral.

   - *Structural reflection* exposes the system's architectural configuration to the system itself. The relationships between them can be traced by starting at a given component and retrieving its neighboring components and connectors linking

them. In this way the entire architectural configuration can be navigated. In addition provides and requires interfaces can also be inspected and modified at the meta level.

- *Behavioral reflection* allows for the inspection of the activity taking place within the components and connectors, and their interaction. This can be realized by providing appropriate monitoring hooks throughout the base structure's implementation. These hooks contain the needed accounting and monitoring functionality and are called any time a significant change is system behavior is taking place. It is then possible to gain an understanding of which critical pieces of system state are contained in which component in the architecture.

One of the most powerful features that the design decision model of an architecture therefore enables is the use of an implementation of *reflection* which makes the decisions taken accessible to the system itself while it is executing. This is done by adding support for this introspection at the appropriate places in the support libraries and runtime model.

4.1.2 *Maintaining system integrity*

In addition to simply adding that support however, additional safeguards must be implemented to guarantee internal consistency. This is due to two issues the use of reflection brings with it.

The first is the synchronization between various parts of the architectural model. This becomes important when an action taken in one area of the model has an impact on other parts of the model. It is a safe assumption that at any point in time there is a flow of communication between various components via their intermediate connectors, for example calls on their interfaces to receive and act on requests. This flow should not be interrupted and left in an inconsistent state. This might happen if components participating in the communication are shut down without first closing the communication itself gracefully. Other components awaiting the reply on a request might crash or simply wait forever for a reply that will never come.

Secondly, the internal state of a component (if not constant but variable) should always be preserved when operations are performed on it. The addition of a provides interface or replacing the entire component with a set of finer-grained components and connectors should be transparent to the rest of the architecture. For instance, a component that functions as the interface to a hardware part should never allow operations to affect its internal state as it could become out of sync with the state of the hardware. This might have severe consequences. Another example where reliance on internal state dictates additional reconfiguration safeguards is a component that generates unique identifiers. If it is to be replaced, the replacement component(s) must preserve the knowledge of which identifiers were already generated in the past so that their duplication can be prevented. Failing to do so would compromise system integrity.

The best way to resolve these issues is to bring the system into a known and safe state prior to allowing any reconfiguration to happen. This should be done inside the meta layer, acting on the base layer of

the architecture. It may be necessary to identify ahead of the reconfiguration exactly which components are involved in it and in which way so that the right steps can be taken. An example in the case of communication flow between components would be to allow pending requests to complete while not allowing new requests to be queued up for processing by the component. After a finite time the component's pending requests would complete and thus a known safe state is reached. The meta layer can in parallel trap any new incoming requests and buffer these, so they can be forwarded once the reconfiguration is complete.

Several reconfiguration algorithms have been developed, which can be divided into static and dynamic algorithms. The former type uses knowledge of the architectural configuration to identify which components need to be brought into the safe state. This will typically include all components that may possible be affected by the upcoming change. Dynamic algorithms instead use more detailed run-time information of the communication flows in the system to limit the set of components that must be made safe to the minimal set. The disruption to the system is decreased as a result, and hence performance during application of a change is better.

A second factor is the *unit of disturbance* that determines the approach taken to gracefully suspending the communication flow in the system. This can either be done by suspending the operation of entire components or by selectively suspending connectors between components and allowing components limited ability to continue their operation. The latter again ensures that the impact on the executing system is lowered.

4.2 WHERE TO ADAPT

Architecture-level adaptation can be seen as occurring "above the application layer"[12] due to the fact that the application itself is the object of the transformation. A second approach has been the subject of research that can be described as "below the application layer". It instead enhances the middleware or operating system layer which the system is built on top of to support adaptive behavior – which applies mostly to distributed scenarios where many systems are working together to achieve a common goal. As the focus of change is elsewhere, a benefit is that existing systems and architectures can make use of this enhanced behavior without needing to be changed or rewritten in a design decision framework. For example, the middleware layer can use interposition and interception techniques to profile the system running on top of it. When needed changes can be made to the way dynamic libraries are loaded and called.

4.2.1 *The two concerns*

There must be constraints imposed on the runtime evolution of an architecture so that it does not evolve beyond all recognition and without structure or limit. Generally, adding complexity to the system to achieve increased flexibility will bring costs in terms of reliability and speed (in addition to non-runtime quality attributes such as modifiability and testability). The functionality which the system is delivering must not be impacted by the need for change. An overarching principle of

runtime modification needs to be present to separate the two primary concerns present. These are:

A. the purpose of the application

B. it's ability to dynamically adapt to demands for architectural change

For this reason, almost all implementations of dynamic architecture modification implement some form of "monitor code" to play the former role. This is a part of the architecture which is a constant and does not undergo dynamic change at runtime. Depending on the way in which the dynamic architecture has been implemented this may be a module that sits "on top" of the rest of the architecture, a component within the model, or a part of the support library or reflection model that the architecture uses.

4.2.2 *Describing runtime changes*

What parts of a system can undergo change in a component and connector model? First of all it must be possible to create components and connectors by supplying an appropriate description or implementation and instantiating it. Secondly, it must be possible to make subsequent changes to their original definition, whether it is in terms of their internal implementation, the interface (ports) they exposes externally and the actual connections between them (the topology in which they are arranged). A more complex operation such as replacing a component with a completely different implementation that has nothing in common with the first could then be defined as a composite of a set of more simple operations.

A second class of change operations use the recursive nature of the component and connector model to decompose components into a set of smaller components. Conversely an existing component can be wrapped inside a larger one. The components could be viewed as forming a hierarchy.

4.3 INTRODUCING ARCHIUM

In Jansen and Bosch [23] an implementation of a design decision model which integrates components and connectors along with executable code is proposed called *Archium*. It is further refined in the papers Kremer [24], Kremer and van der Burgh [25], Burgler and Kok [10] and Slagter [38].

Archium consists of a compiler and runtime platform built on top of the Java language. It is the currently most promising starting point for exploring the problem statement of this thesis and will therefore be used throughout as the basis for implementing runtime design decision functionality.

Archium's defining feature is its integration with the featureful and mature Java development platform. Archium is built on top of Java by adding special keywords that are interpreted by the Archium compiler – making it in effect a superset of the Java language. A special set of architectural knowledge and architectural construction concepts are implemented as keywords in the Archium language that can be used on top of regular Java. This allows the software engineer to embed the

full power of Java inside Archium's software architecture model and reap the benefits of both. The overall architecture is shown in Figure 4.

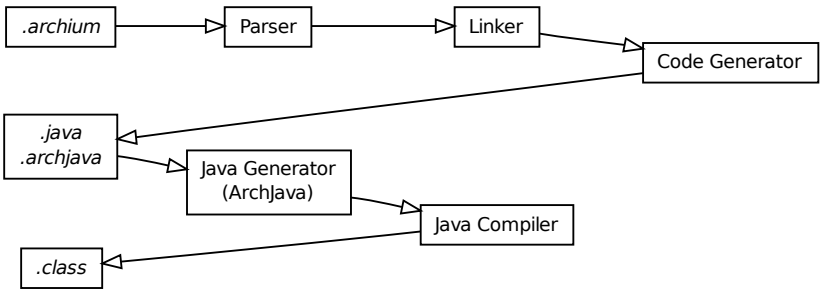

Figure 4: Overview of the Archium architecture

The Archium toolset unsurprisingly includes a compiler that compiles Archium source to Java. This happens in a two-phase process that involves the use of an intermediate language called ArchJava. ArchJava is itself a superset of Java that specializes in implementing a component and connector model and ensuring communication integrity between these architectural objects. Archium uses ArchJava to ensure that components can only communicate with others they have been explicitly connected to via connectors.

Archium can be viewed as a meta-model that is built on top of three interwoven sub-models that taken together capture architectural knowledge and enable runtime architectural change. These models are the following:

A. The architecture model,

B. The design decision model, and

C. The composition model.

In the following subsections these models are described in more detail, before going into how Archium should be adapted to meet the central research question.

4.3.1 *The architecture model*

The architecture model uses standard concepts like the components, ports and connectors as described in Section 3.1 to codify the component view of the architecture. The Component Entity entity is the Archium-specific equivalent of an architectural component. It represents a logical architectural building block. The modifier *Entities* is used to distinguish this phrase from the general usage of the term *components* throughout the software architecture research field. The use of the word *entity* throughout Archium is also meant to avoid confusion with e.g. Java-level objects.

The purpose of Component Entities within a software architecture defined in Archium is analogous to the principle of a Class as commonly used in object-oriented programming – in the sense that it is a self-contained entity that exposes a defined set of interfaces to the external world, while hiding most of its internal implementation from view.

Component Entities can have a set of defined incoming and outgoing communication channels defined within them called *Ports*. Ports are

marked explicitly as being either *required* or *provided*, depending on whether the Port expects to consume data or functionality provided by another Component Entity or if it will provide this to another Component Entity. The binding agent between Ports are the Connector entities which are simple data conduits between the Components Entities.

In summary the architectural sub-model of Archium allows modeling of software architectures using concepts of the Component and Connector View. The main concepts used are the Component Entity and the Connector, and Connectors provide a means for communication between Component Entities. Connectors can only be connected to explicitly defined Ports on Component Entities. Archium does not incorporate an existing ADL but has designed its own from scratch.

4.3.2 *The design decision model*

The design decision model in Archium consists of two primary concepts: the Design Decision entity itself and the Design Fragment. Both are treated are first-class concepts in this model, on par with e.g. Component Entity in the architecture model.

The overall function of a Design Decision entity is to capture the outcome of the software architect's deliberation and decision-making process, and link it with other Archium entities. Some of these are second-class entities of Archium in the sense that while they are not very tightly integrated in the model and do not expose much functionality, they are essential to the AK aspect of the model. They can be thought of existing more for reference purposes. Examples of such entities in Archium's AK model include Requirements, Stakeholders, and Rationale. A Design Decision can refer to any number of these to capture the origins of the design process. These will not be discussed further.

A Design Decision is also the place where separate entities from the architectural model are constructed into a whole architecture. Firstly, a Design Decision is structured in terms of decision point from the view of the software architect. Given that a part of the existing architecture of the system presents an opportunity for change or improvement; this is the impetus for the Design Decision, which is followed by a number of alternative solutions to the opportunity. In every valid Design Decision, one of these alternative choices must be chosen to be applied to the software architecture. All solutions can reference additional information regarding the pros and cons of making that choice, along with a rationale for the decision made by the architect. This of course enables the architect and his/her successors to base future design decisions on this AK.

The question arises how exactly Design Decisions implement changes to the software architecture *as it runs*. This takes place in the Design Fragments that thereby form the link between the architectural model and the design decision model. Design Fragments acts as containers for the architectural elements involved in the Design Decision – the Component Entities, Connectors and so forth. Design Fragments are therefore a collection of architectural entities that represent part of the complete software architecture.

Design Decisions use Design Fragments to add new architectural entities to the overall architecture, or to remove existing ones from the running system. Also, existing entities can be modified in some

way. The mechanics of how this works are captured in the composition model described below.

Design decisions are the highest-level elements in Archium from a source code point of view; the entire architecture is created by defining entities and instances of this type and linking them together. Design decisions also contain both levels of reflection, base reflection and meta reflection.

4.3.3 *The composition model*

The composition operation used within the chosen Design Solution of the Design Decision applies a change to an existing Design Fragment. The actual change to be applied is described in another Archium entity called a Delta. The change to be applied could for example be the addition or removal of a port to a Component Entity or the specification of a more detailed internal design by decomposing a Component Entity into other Component Entities and Connectors.

Every change to the architecture is established by means of a Delta. By making change explicit, the evolution of a system can be seen as a series of changes, and each part of a running system can be traced back to a certain change, improving traceability.

The composition model can be viewed as the glue between the design decision and architectural models of Archium. Concretely, the composition model defines how desired changes should be mapped to architectural entities defined in the architectural model. To that end the composition model defines a number of *composition techniques*. One of these is the superimposition technique which is described in more detail in Bosch [8]. It allows Component Entities to merge with Deltas in a number of clearly-defined and controlled ways. The detailed mapping between these entities is further defined in Design Fragment Composition entities.

4.4 MODELING DESIGN DECISIONS IN ARCHIUM

As stated Design Decisions are first class entities in the Archium language (which will hereafter be referred to simply as *DDs*). In fact, each Archium source file describes exactly one Design Decision, along with any other supporting architectural entities needed to implement the solution selected within the Decision.

The model used to represent design decisions in Archium is depicted in Figure 5. Central to a Design Decision entity is the *problem* the decision is attempting to address. A single problem can of course have several different workable *solutions*. These can be independently specified and implemented in detail if desired. The DD then chooses from among the available solutions to come to the actual *decision* being made to address the problem.

```
design decision InitialSystem_DD0() {
  potential solutions {

    solution alpha {
      architectural entities {
        DD1ClinetServerDD dd1DD;
      }
```

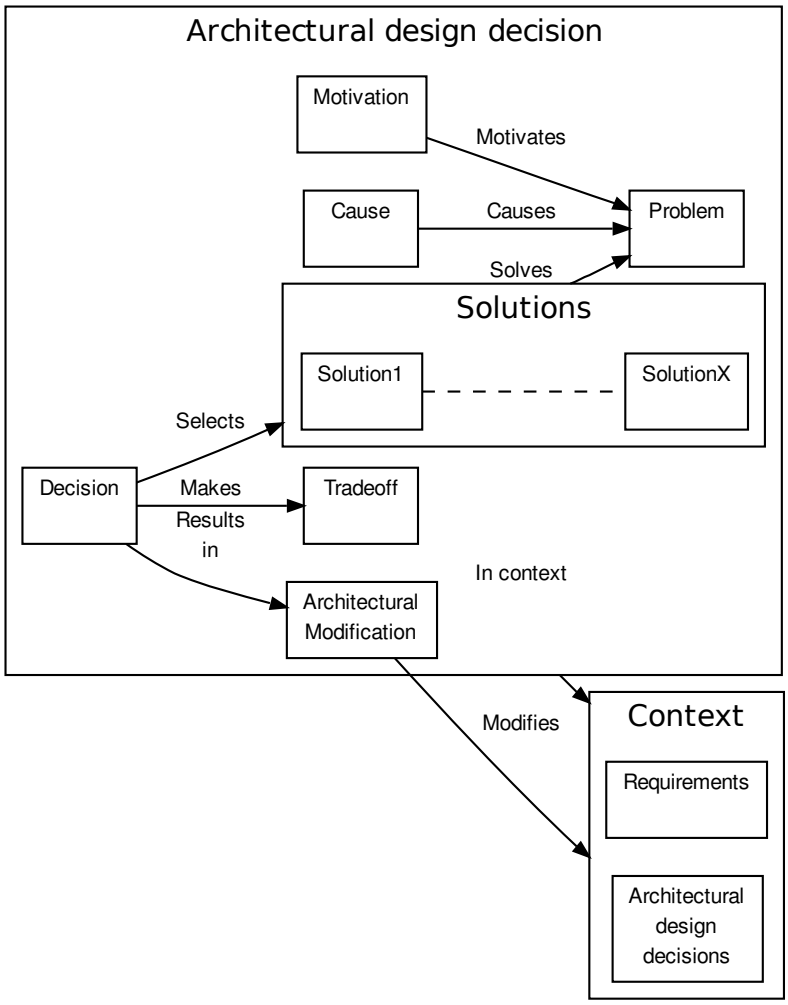

Figure 5: The model of design decisions used by Archium. Source: [23]

```
    realization {
      dd1DD = new DD1ClientServerDD();
      return dd1DD;
    }
  }

  solution beta {
    architectural entities {
      DD1ProducerConsumerDD dd1DD;
    }

    realization {
      dd1DD = new DD1ProducerConsumerDD();
      return dd1DD;
    }
  }
}
```

```
  decision {
    decision alpha;
  }
}
```

Listing 1: A bare-bones Design Decision using the Archium language, showing the solution selection and realization. The simple effect of this Design Decision is to load a second one. This technique is commonly used to bootstrap a set of Design Decisions to shape the high-level architecture.

There are some optional elements of a DD not linked to any other first-class architectural entities, such as a textual description of a motivation for solving the problem, what caused the problem to arise in the first place and the trade-off the decision makes among a number of factors when selecting a solution to the problem.

4.5 ARCHIUM'S ARCHITECTURE

On a technical level the Archium toolset can be divided into the following modules:

A. parser generator,

B. compiler/linker and

C. runtime platform.

The first of these is able to take a language syntax specification (for example, for Java 1.5) and generate an Archium parser that can parse both the Archium-specific keywords and regular Java syntax. Its workings will be left out of the scope of discussion as it is only useful when retargeting Archium to a different base syntax that should be extended.

The compiler's first task is to use the parser to parse the the input files and construct the Abstract Syntax Tree (AST). It is constructed out of a set of nodes of different types containing the Archium and Java statements along with additional annotations.

The Archium runtime platform is actually a set of Java classes implementing reflection capabilities which are then subclassed by the Java output files generated by the Java/ArchJava compilation and linking process.

4.6 STABILIZING ARCHIUM

Before starting to add new functionality to Archium, a few additional fixes and improvements were required to stabilize the software and allow for the execution of the included examples and test cases. In many cases this involved the Component Composition functionality which appears to have been a recent addition to the Archium platform.

The first area in which several improvements were made is the parser module and its handling of code generation for references to Archium variables. Archium support both static and dynamic references to its entities. A dynamic reference in this context means a reference that consists of multiple parts which correspond to layers in the architecture. This allows for the referencing of a specific Component Entity that is

located within another. The internal data structures of the AST (called *NodeData*) were not being tracked correctly. Static references were also not correctly generated in all cases, particularly when dealing with Composition Configuration entities where incorrect AST nodes were adding extraneous strings into the compiled output.

Once smaller pieces of Java source code have been generated, e.g. for references, they are then integrated with an existing template file to produce the finished `.java` file. The template engine used by Archium internally is based on the Apache Velocity engine with a number of extensions and adaptations, and allow for complex logical expressions to be used while the template's output is being produced. These expressions can even use the internal state of the Archium parser to decide which sections of the template should be included in the output. The base templates on which generated Composition Configuration `.class` files are built were then fixed to ensure that the generated Java files compile correctly after the code generation phase.

A number of problems with the templates used by Archium were fixed in this process as well. One such example was the Design Fragment Composition template. This template was not generating Java output correctly due to layering issues when adding Ports to Component Entities through the use of Deltas. The superimposition logic was corrected to avoid null pointer exceptions at runtime and to allow messages to flow between Component Entities.

Lastly a number of new features were added, most notably naming convention adherence and an AST visualizer tool. The former assures that all parts of Archium's naming system for its variables conform to a set of standard rules. This forces Archium users to clearly identify their variables by appending a unique terminator string to all names. In turn this enables a number of internal efficiency improvements in the Archium parsing and linking stages and provides an extra level of error detection preventing accidental misuse of variables.

The AST visualizer tool takes as input a parse tree generated by the compiler as it runs and allows for navigation through that tree. This helps in understanding the parser and AST implementation which is useful for debugging the code generation stage. The tool is shown in Figure 6.

4.7 REQUIREMENTS

Currently in Archium there are no explicit relationships between the Design Decisions. As described above it is possble to define several Solutions within each DD, one of which must be chosen to be executed when the architecure is executed at runtime. The alternative Solutions remain accessible to the application but there is no way to change the choice of solution – in other words to change the design decision that was taken – at runtime. The decision as taken during compile-time is final.

The ovearching requirement this thesis will attempt to satisfy is therefore the following:

REQUIREMENT 0

> The software architect must be able to undo and/or change a design decision while the Archium system is running and online

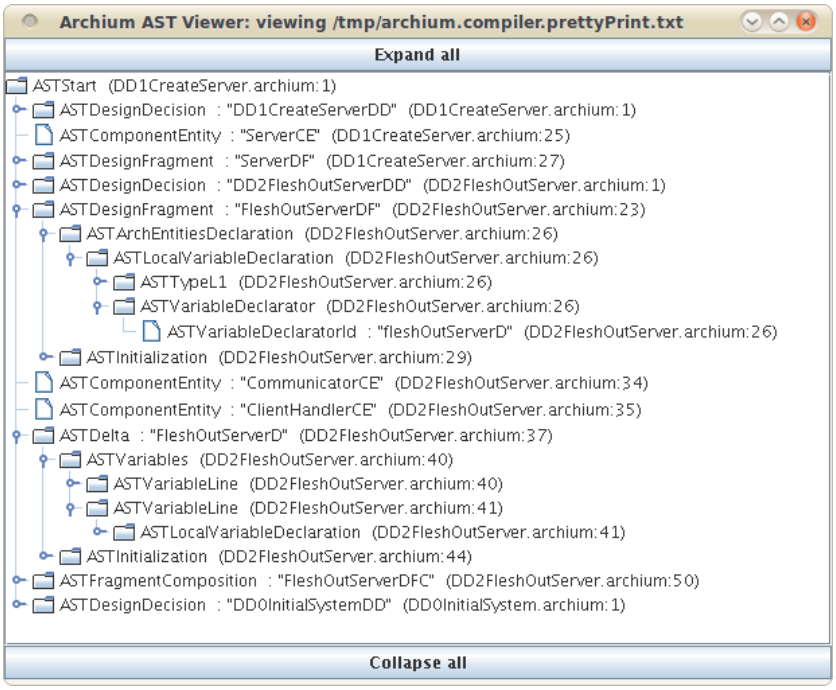

Figure 6: The AST View tool

This requirement can be broken down into the following subrequirements, which also translate it into specific Archium terminology:

REQUIREMENT 1
: A Design Decision entity must be able to switch between Solutions α and β contained within it while it is running

REQUIREMENT 2
: It must be possible to insert a new Design Decision entity in the architecture while it is running

The ability to change the selected Solution within a Design Decision itself has a few implications. First, the effects of a particular 'active' Solution α that was selected must be undone. This will involve reversing the steps of the Design Fragment Composition in composing Component Entities and Connectors. Once the effects of one Solution are undone, it should be possible to load another Solution β in its place. Secondly, there must be a mechanism to signal the running Archium system at runtime to carry out these steps. Requirement 1 is therefore further detailed as follows.

REQUIREMENT 1.1
: It must be possible to halt a particular Solution within a Design Decision, undoing its effects on the architecture.

REQUIREMENT 1.2
: It must be possible to select a particular Solution within a Design Decision after a previous Solution has been halted.

REQUIREMENT 1.3
: There must be a mechanism to request the halting and starting of Solutions within a Design Decision.

Requirement 2 builds on the functionality described in Requirement 1. Once the active Solution α is halted, the architecture is in a state where it has not been modified in any way by the parent Design Decision Δ. In order to truly achieve runtime decisions about the architecture it should be possible for the architect to insert a new Design Decision into the architecture. There are several ways of making this possible, some easier than others. Two have been selected below and marked as must-have and may-have, in accordance with the level of complexity expected in the implementation.

REQUIREMENT 2.1 (MUST-HAVE)
It must be possible to halt a particular Solution within a Design Decision, undoing its effects on the architecture.

REQUIREMENT 2.2 (MAY-HAVE)
It must be possible to select a particular Solution within a Design Decision after a previous Solution has been halted.

The next chapter will describe how these requirements have been implemented by improving Archium.

5

IMPLEMENTING RUNTIME ARCHITECTURAL DESIGN DECISIONS

5.1 INTRODUCTION

This chapter describes the design and implementation of new features in the Archium platform that meet the requirements and constraints posited in the previous chapter. The work needed to implement these features can be roughly broken down into three different areas:

1. Loading and unloading the solution selected by a design decision while the system is running;
2. Creation of an intermediate support layer that monitors the loading and unloading of arbitrary design decisions; and
3. Creating a user interface to allow the end user to interact with the new functionality.

The new functionality in all three of these areas must work seamlessly together to achieve the final goal. From the point of view of Archium's architecture, the language specification (and therefore the parser and compiler) will not be changed.

5.2 LOADING AND UNLOADING DESIGN DECISIONS

In this section the meaning and mechanics of loading and unloading of Design Decisions from a running Archium system are discussed.

As illustrated in the previous chapter, the top-level entities in Archium are Design Decisions. Analogous to Java, by convention each Design Decision is usually described in it's own `.archium` source file (although it's possible to include multiple Design Decisions into a single source file, it does not help legibility).

As a result of the compilation process, each source file produces a number of Java `.class` files – for all the architectural entities contained within the Design Decision. The act of loading of a Design Decision comes down to loading the `.class` file into the Java virtual machine where Archium is running.

For the purposes of this chapter a differentiation is made between loading a Design Decision *statically* and *dynamically*. Static loading implies that the loading is the result of explicit instructions to do so during the run-time initialization of the architecture. For example, given the following Archium source defining two Design Decisions;

```
design decision DD1CreateServerDD() {
...
  component entity ServerCE;
...
}

design decision DD2FleshOutServerDD(DD1CreateServerDD
    createServerDD) {
```

```
...
  createServerDD::serverCE plays newlyintroducedDF::
      fleshOutServerD;
...
}
```

Listing 2: Example set of two Design Decisions with compile-time dependency

it is clear that the first Design Decision `DDFleshOutServerDD` will be loaded without any user intervention if the second is loaded, because the latter makes an explicit reference to the former.

In fact, it is customary that the very first Design Decision created in an Archium architecture (the *initial* Design Decision) is of the following format:

```
design decision DD0InitialSystemDD() {
  potential solutions {

    solution InitialSystem {

      architectural entities {
        DD1CreateServerDD dd1DD;
        DD2FleshOutServerDD dd2DD;
        DD3FleshOutServerDD dd3DD;
      }

      realization {
        dd1DD = new DD1CreateServerDD();
        dd2DD = new DD2FleshOutServerDD(dd1DD);
        dd3DD = new DD3FleshOutServerDD(dd2DD);
        return dd3DD;
      }
    }
  }
  decision {
    decision InitialSystem;
  }
}
```

Listing 3: Example of an initial Design Decision that loads all other required decisions

Such a Design Decision has no dependencies itself but prescribes exactly which other decisions must be (statically) loaded and in which order. This provides the necessary framework required to combine the set of Design Decisions into a coherent architecture.

The static loading instructions are generated by the Archium parser before compiling the architecture into Java `.class` files. These rely on a generic implementation of loading available in the reflection layer. No user intervention is needed and the architecture first completes the loading of all dependent Design Decisions before handing control over to the user.

In contrast, dynamic loading of Design Decisions is not parser-generated but supported through a generic implementation.

5.2.1 *Loading a Design Decision dynamically*

Before the mechanics of loading are described, a short note about linking. Archium does not provide for run-time linking, i.e. resolving names of references to find the objects they refer to. Instead all linking is done at compile-time. [1] A consequence is that any Design Decision that is loaded at runtime (whether during startup or dynamically) must have been compiled along with the source code of Design Decision it is loaded on top of. This means new Design Decisions cannot be designed and compiled separately from those they have a dependency on (and most Design Decisions have a dependency on at least one other).

As was described in Section 4.3, the output of a Design Decision is a Design Fragment that represents the running architecture after that Design Decision has been applied. When Design Decisions are chained together and one is applied to another, the net result is that the Design Fragments contained within their selected Design Solutions are combined to create a new merged Design Fragment.

Applying a Design Decision therefore has an effect on the Design Fragment that represents the running system by modifying it. The output of that process is a new Design Fragment. Figure 7 illustrates this.

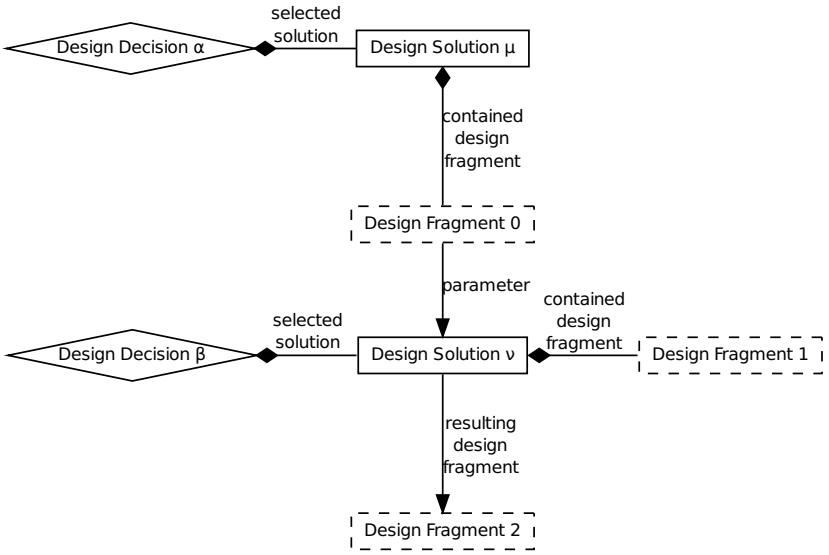

Figure 7: Constructing an architecture by chaining Design Decisions to alter the running Design Fragment. Design Fragment 0 is composed with Design Fragment 1 by Design Decision β to create Design Fragment 2.

Loading a Design Decision at runtime involves a number of extra steps which are required to initialize the state of the Design Decision. During static loading these steps are executed automatically as a result of being compiler-generated.

Listing 4 shows in more detail the (pseudo-code) steps which are needed to execute this process. The Java class-loading mechanism is used to load the `.class` file representing the Design Decision into the

1 The Archium source code and documentation does make references to both 'static' and 'dynamic' linking; those are however both forms of compile-time linking. The latter refers to translating source code text strings describing the encapsulation of an object into a correct pointer to that object.

running architecture. A set of verification steps then ensure that the class file is valid and has a compatible constructor.

```
1. Load the class file
2. Inspect constructor to determine if it takes 2 parameters
    (ArchitecturalEntity, Object) or 3 parameters ((
    ArchitecturalEntity, DesignFragment, Object)
  a. In the first case the Design Decision is not dependent
      on other Design Decisions
  b. In the second case the Design Decision is dependent on
      another Design Decision
3. Create a new instance of the class with the correct
    parameters
4. Activate the instance
```

Listing 4: Bootstrapping a Design Decision class file

Once the class has been loaded it is activated as any other Design Decision, using functionality implemented in the reflection layer. Listing 5 lists (in pseudo-code) the steps required to activate the decision.

The first step is to instantiate the involved Design Fragments and Design Fragment Compositions; this happens within the code classes generated by Archium at compile-time. If the Design Decision depends on a prior one, the resultant Design Fragment of its predecessor is passed to it during initialization.

Next the two Design Fragments (again, in the usual case where a Design Decision builds on a predecessor) are merged via the *composition* operation. This happens with the help of the Design Fragment Composition entity, which describes in detail how the various entities *contained within* the Design Fragments should be merged into a consistent whole. In many cases a Design Fragment Composition is defined which contains further instructions on how to merge the two Design Fragments. An example is given in Listing 6 where the composition of a Component Entity with a Delta is given.

```
1. From the list of Design Solutions, activate the one
    selected in the Design Decision:
  a. Instantiate all Design Fragments contained in the Design
      Solution
  b. Instantiate and initialize all Design Fragment
      Compositions contained in the Design Solution
  c. Start composition of the Design Fragments (e.g. add
      Delta's to Component Entities)
  d. Unsuspend the Delta's in the newly combined DF
2. Add the Design Decision being activated to the entity list
     of the target Design Fragment
3. Add a reference to this design decision to the origin
    lists of all requirements that this design decision
    creates
4. Set the percentages of requirements that are being
    realized
```

Listing 5: The activities implemented in the reflection layer to load and activate a Design Decision

```
// realization of the selected Design Solution of this Design
     Decision
realization {
  fleshOutServerDF = new FleshOutServerDF();
  fleshOutServerDFC = new FleshOutServerDFC(createServerDD,
      fleshOutServerDF);
  return createServerDD composed with fleshOutServerDF using
      fleshOutServerDFC;
}

...

// definition of the Design Fragment Composition referenced
    above
design fragment composition FleshOutServerDFC(
    DD1CreateServerDD targetDD, FleshOutServerDF sourceDF) {
  composition {
    targetDD::serverCE plays sourceDF::fleshOutServerD;
  }
}
```

Listing 6: The Design Fragment Composition instructs two Design Fragment to merge using the specified composition techniques

5.2.2 *Unloading a Design Decision dynamically*

Unloading a Design Decision at runtime is in many ways similar to loading one, except that the required steps are performed in the reverse order. There are however a few additional considerations that come into play.

The first one is the dependency relationship between the Design Decision that is being unloaded and the rest of the architecture. As this involves more than one Design Decision it will become important in the next section. At this point the working assumption is that there are no dependencies left to resolve before going ahead and unloading the selected Design Decision, i.e. the DD we are dealing with is a 'leaf' in the dependency tree.

The second consideration is that unloading a Design Decision requires the halting of threads that are executing within the architecture. This must be done as early as possible in the process. Luckily there is existing functionality present in the Delta reflection class that makes this job easier.

Listing Listing 7 shows the (pseudo-code) steps required to unload the Design Decision.

```
1. Find the Design Decision to unload in the running primary
    Design Fragment
2. Shutdown its introduced Design Fragment
  a. Remove the Deltas contained within the Design Decision
      from the architecture by unstacking them from their
      respective Component Entities
    - Find the solution selected to run within the Design
        Decision
```

```
    - Find the Design Fragment introduced by that solution
    - Call shutdown() on this Design Fragment to suspend all
        running threads within its Deltas. Their Component
        Entities as a whole must be suspended also.
    - Remove the Deltas from the Design Fragment
    - Unload enclosed Component Entities from other Component
         Entities on which they were stacked
    - Cleanup Composition Configuration entities
    - In the Design Solution, set the Design Fragment equal
        to NULL and remove all references to it.
3. Deactivate the Design Decision
  a. Remove all references to the Design Decision from the
      other entities in the architecture
4. Unsuspend the affected Component Entities and their
    remaining Deltas
  a. The architecture continues on with its execution
```

Listing 7: The activities implemented in the reflection layer to deactivate and unload a Design Decision

The first few steps of the unloading process focus on locating the affected Deltas by searching through the appropriate runtime data structures in the architecture. Each architectural element keeps an updated list of all its child elements, its parent element and in many cases the other elements which refer to it (e.g. the foreignEntities HashMap property) as well.

Before anything else can be done a run-time reference to the Design Decision that is to be removed must be created. The easiest way to do this is to start with the Design Fragment that represents the entire running architecture and search its element list until the correct entity is found.

The process must then in a sense be repeated to reach the Design Solution that was chosen by the Design Decision to run. This is important because that entity leads us to the Design Fragment which is the entity where we need to start the unloading process. The reason is that the Design Fragment contains within it all other architectural entities that the Design Decision has introduced into the architecture.[2]

At this point the Design Fragment should initiate shutdown. The previous version of that method has been extended to provide all the required functionality. The first step is to suspend all of the Deltas contained within the Design Fragment. This is needed to ensure the integrity of the architecture, as only once the threads of execution within the Deltas have suspended it is safe to modify and rearrange them. All effected Component Entities will automatically suspend as well (i.e. they will suspend the entire stack of Deltas running on top of them).

It is now safe to remove the Deltas from the Design Fragment. This is done by removing all references to the Delta objects from a number of HashMaps and other internal data structures. While the Deltas are removed, special care must be taken to also remove in a similar manner the Component Entities enclosed inside the Deltas (this is the case when

2 The Design Solution is in fact a mere intermediate step, there to provide the functionality of choosing among multiple defined Design Fragments if the user chooses to use that functionality. Unfortunately, in practice this feature is hardly used as is the case in most of the Archium example architectures.

a Delta decomposes one Component Entity into multiple finer-grain ones).

The final remaining step of the Design Fragment cleanup are the other objects used to merge its payload with the running architecture – entities such as Composition Configurations and Composition Techniques.

Returning to the Design Solution level, the Design Fragment has now been fully deactivated and all references to it removed. It is now unloaded entirely from the runtime environment by ensuring all references to it are set to `NULL`.

Now at the final Design Decision level, references to the outgoing entity are removed from its neighbor Design Decisions after which it is removed entirely as well. The Component Entities (and their Deltas) which had been suspended can now be resumed. At this point a smaller overall Design Fragment remains of the running architecture, and it will resume execution at the level of functionality provided before the unloaded Design Decision was loaded in the first place.

Contrary to the work required for loading a Design Decision, the reverse operation does not require a composition operation. This means this aspect of the process is simpler than its counterpart due to the nature of the Archium implementation.

5.2.3 *Exposing functionality*

Now that the procedures for safely loading and unloading Design Decisions have been defined, this new functionality needs to be exposed to the external world so that Archium plugins such as Archimon (see Section 5.4) can make use of it.

The solution is to implement these procedures inside the reflection layer of Archium, where they can be called from anywhere in the runtime. The added methods were quite evenly distributed over the Design Decision, Design Solution, Design Fragment, Delta and Component Entity reflection classes. In many cases the loading functionality was rewritten and the unloading functionality added as a new (set of) method(s).

The main entry point to the new functionality was constructed in the Design Fragment reflection class in the form of the four methods listed in Listing 8. The reason for this is that both loading and unloading require the 'primary' Design Fragment representing the entirety of the running system as a starting point; it therefore seemed a natural place to add the following four methods.

```
public static void applyDesignDecision(DesignFragment df,
    Class ddClass, boolean createCP)
performDesignDecisionApplication(Class ddClass)

unloadDesignDecision(DesignFragment df, DesignDecision dd,
    boolean createCP)
performDesignDecisionUnloading(Object name)
```

Listing 8: The newly added methods to facilitate Design Decision loading and unloading

5.2.4 *Dealing with RMI constraints*

The reason for adding two separate methods for each operation has to do with the nature of using Java's Remote Method Invocation (RMI) library to communicate with other Java runtimes. As will become clear in Section 5.4 of this chapter, it is necessary that the load and unload operations are able to be invoked from outside the Java virtual machine in which the Archium architecture is running.

RMI allows for transparent communication between virtual machine instances, in the sense that the running Java classes do not have to be aware that the manipulation of objects and parameters they execute is actually taking place remotely. The only requirement to use RMI is the creation of an interface class that defines the methods that are remotely callable via RMI and their parameters. Also, a small program called `rmiregistry` needs to be running on either of the machines. It functions as a broker in order to route requests from a local virtual machine to a remote virtual machine.

The way RMI works in practice is that passing objects from one Java Virtual Machine (VM) to another becomes transparent. A number of prerequisites need to be adhered to such as compiling skeleton files followed by other associated setup work at runtime. Archium already includes these in its compilation and runtime processes, they simply need to be extended.

RMI allows for the passing of Java objects by copy or by reference. In the case of the latter, the destination VM receives an instance of a skeleton class which represents the remote object living in the origin VM. All the class' methods (or those which have been included in the skeleton's definition) can be called by the receiver and will behave as if the receiver was manipulating a local object.

The challenge that arises with RMI is when one VM passes a reference to one of its local objects to the remote VM, after which the remote VM wishes to pass that same reference back again. This is clearly not possible since the remote VM never 'owned' the object in the first place (by virtue of not being its creator) – it only possesses a skeleton representation and not a true reference to the original object.

However there are instances where references to Design Decisions and Design Fragments need to be passed back and forth in order to allow plugins to make use of the loading and unloading functionality. This brings us to the reason for the 4 method signatures instead of simply 2; the workaround is that the origin VM passes a reference to the remote VM as normal, and that the remote VM passes back a reference to a String object containing the name of the architectural entity in question. Figure 8 illustrates this situation. Hence two of the four methods are devoted to unpacking objects referenced 'by name' and translating these references back into local Java objects, to be passed on to the 'real' entry point methods for loading and unloading.

5.3 CONSTRUCTION OF THE CHECKPOINT FRAMEWORK

The next phase of the implementation is the construction of a framework for the Checkpoint layer. This new layer will be inserted in between the Archimon plugin and the reflection classes implementing Design Decision loading/unloading as described above. It forms an explicit *change layer* on top of the Design Decision and Design Fragment entities

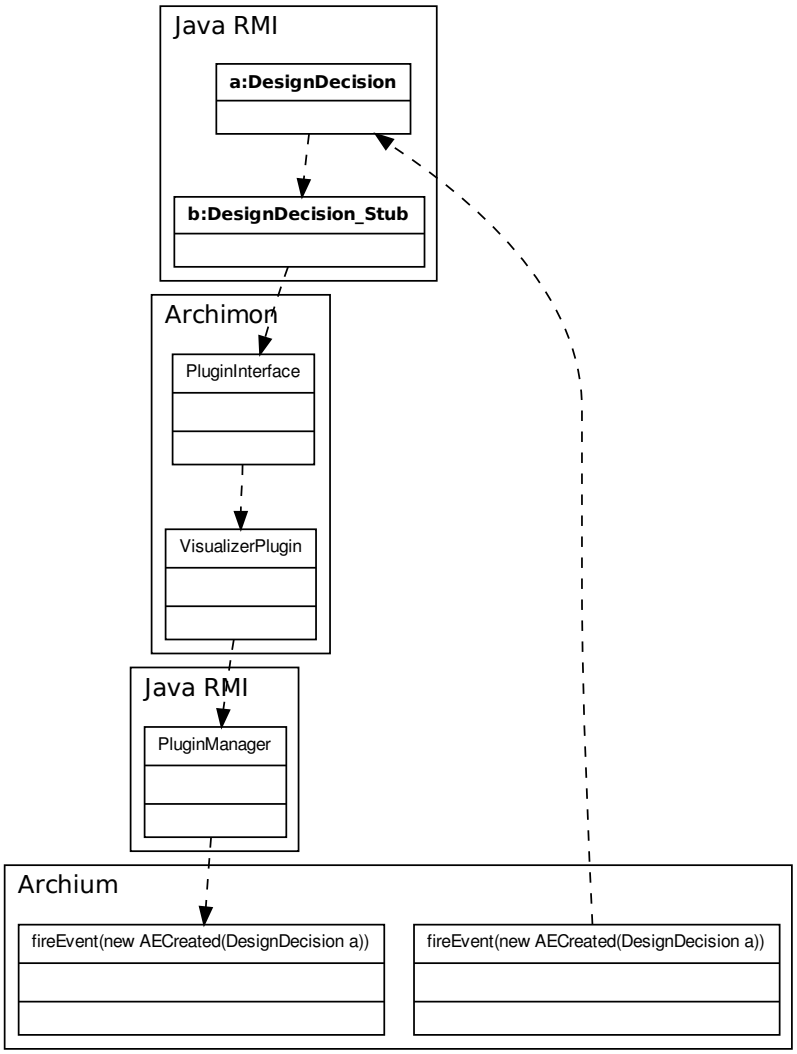

Figure 8: The circuitous nature of using RMI to communicate between the running architecture and the Archimon plugin

in the reflection package. It can be described as a "command and control layer" and its functionality breaks down into the following areas.

DEPENDENCY TRACKING Tracking the relationship between Design Decision entities and their Design Fragments (which of the former has introduced which of the latter) along with the Design Decision's possible tree of dependency relationships. It may also track the entities they contain (such as Deltas, Component Entities, Connectors, etc.) while they are loaded.

ARCHITECTURE STATE TRANSITION TRACKING Checkpoints represent the change of architectural state through removal or addition of Design Decisions. They must provide "change points" that capture the current configuration of Design Decisions running in the active architecture fragment.

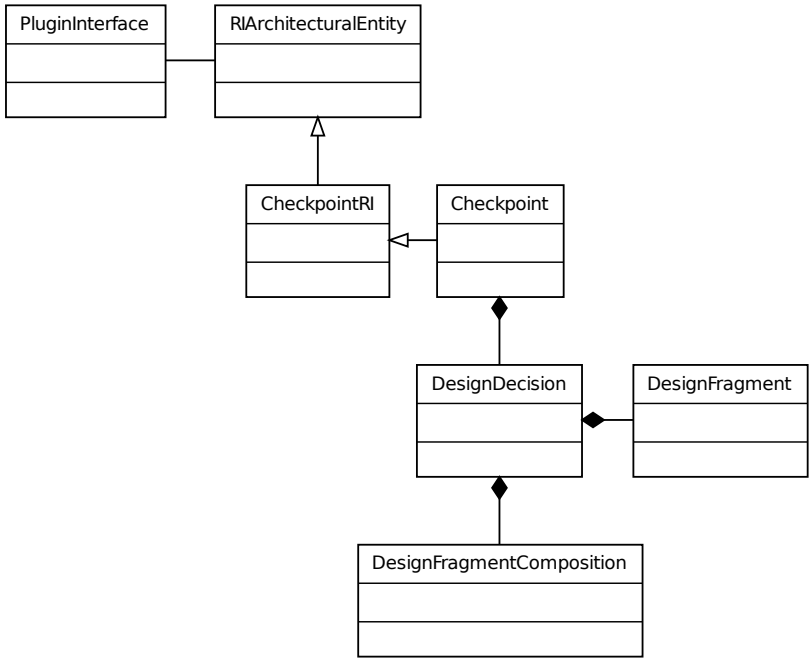

Figure 9: Architecture of the Checkpoint class and its interactions with other Archium classes

ALLOW FOR HIGHER ORDER TRANSITIONS BETWEEN STATES The Checkpoint layer must allow the creation of new states given heretofore unknown Design Decisions that build on the set of currently running ones. This may involve some manual intervention from the user. They may also allow a transition from the current state to any past state the architecture has encountered which will hopefully require less manual intervention. Such state transitions can be described using the well-known concepts of *undo* and *redo* as we know them from many popular applications. Instead of creating a new architectural state, these operations may allow us to transition back to an earlier state or forward to a subsequent state (both of which existed as some point in the past).

INTERACT WITH ARCHIMON Archimon is the graphical interaction tool provided by Archium. The Checkpoint layer is an ideal candidate to interact with this plugin so that it can provide a user interface to the checkpoint functionality. This means that the Checkpoint layer will need to interact safely with the reflection functionality introduced in the previous section.

The definition of a Checkpoint used for the purposes of this thesis is an entity that describes the state change of the architecture in the following way:

$$\text{A tuple of } \{\textit{Operation}, \textit{Design Decision}, \textit{Design Fragment}\} \\ \text{where } \textit{Operation} \in \{\textit{LOAD}, \textit{UNLOAD}\}$$

5.3.1 *Design considerations*

The goal when implementing this new layer is to minimize the impact to other areas of Archium as much as possible. This reduces the risk

of bugs and instability. The existing usage pattern of Archium will therefore remain unchanged:

1. Implement a set of `.archium` files that reference each other in a stacked fashion;
2. Compile using Archium;
3. Call Java interpreter on the primary Design Decision class;
4. All compiled Design Decisions are loaded sequentially until none are left that are dependent on an already running Design Decision.

This is also why building the Checkpoint classes as part of the reflection package is the best option, and allows the very complex compiler generation to be left unaffected. It will only be during the runtime of the platform that these new features will be available, which is exactly where they are needed.

A few simplifying assumptions must be made at this point in order to keep the Checkpoint implementation manageable. As was described previously the Design Solution entity is one of the 'thinner' reflection classes and exists mostly to offer the option of building multiple Design Fragments into one Design Decision if the user so desires. While it would be a useful feature to be able to stop one of the Design Fragments incorporated into a Design Decision and then launch a 2nd fragment belonging to that same decision entity, the added complexity suggests to omit this feature for now.

Instead, the same effect can be achieved in a sightly more complex fashion by unloading the Design Decision in question, editing its source code to change the Design Solution selection and then recompiling and reloading the Design Decision. The fact that none of the Archium source files currently in existence actually uses multiple fully fleshed-out Design Solutions suggests this is an acceptable trade-off.[3]

Another consequence of these design choices is that if a new Design Decision were to be created and inserted into the running architecture, it should have to be dependent on one or more of the currently running Design Decision entities (either a specific one or else the 'primary' decision entity that launches and links the rest together). In order to be able to compile such a Design Decision source file it must be placed in a location where the Archium compiler can find the source files of the entities it depends on. Archium currently provides no runtime dynamic linking capability which would allow this restriction to be overcome.

5.3.2 *Implementation*

The actual implementation of the Checkpoint layer can be found in the file `Checkpoint.java` in the package `Reflection`. It extends the remote interface so that it can be called from a remote Java VM just as the other reflection classes. Almost all of its methods are exported via RMI so that they may be called by Archium.

The existing reflection classes such as those for Design Decisions, Design Fragments, etc. have been modified to work seamlessly with the Checkpoint class. This was done by editing their templates that

3 Again as mentioned previously it might instead be an illustration of a challenge Archium has yet to overcome.

supply the Java code skeleton on which the runtime instantiations of those classes are built (i.e. which they inherit from).

For example, Design Decision entities now create a new Checkpoint entity as part of their loading procedure. There is one exception to this rule; the very first Design Decision to load (referred to as the 'primary' Design Decision) does not cause a Checkpoint to be created. This is because it is solely responsible for loading all other Design Decisions needed for the architecture and connecting them together appropriately. The creation of a new Checkpoint is therefore conditional and there will always be 1 less Checkpoint entity than the number of Design Decision entities in the architecture.

```
Checkpoint(ArchitecturalEntity ie, String action,
    DesignFragment df)
public String getName()
public String getOperation()
public String getOperand()
public void undo()
public void redo()
```

Listing 9: The methods of the Checkpoint class that are externally visible and can be invoked via RMI.

The methods provided in the Checkpoint class are listed in Listing 9. As would be expected the constructor for the Checkpoint class takes three parameters which represent the architectural state change that should occur after the Checkpoint is created. The remaining methods are helper methods of which the first three are mostly used internally in the `undo` and `redo` methods. The latter act much like static methods do, even though the Checkpoint class itself is not a static class.

The primary reason is convenience, as it means the caller of these methods will not have to concern herself with getting the reference correct with respect to the Checkpoint representing the current runtime state with all it's ancestors in the chain taken into account. All Checkpoint objects maintain an internal (static) counter which points to the Checkpoint object representing the current architectural state. The `undo` and `redo` methods can then be called on that instance and will use the first three non-static methods (`getName`, `getOperation` and `getOperand`) to recover the details of this instance and act appropriately.

There are no other methods in the Checkpoint class because of its inbuilt interaction with the existing reflection classes. This means that any plugins are expected to use the Design Fragment class methods for direct loading and unloading Design Decisions, and those of the Checkpoint class to facilitate undo and redo operations.

5.4 CREATING A USER INTERFACE

Several options for realizing a user interface were investigated, including a graphical front-end and a command-line interface. While the command line interface has as advantage that it would likely be easier to implement, a visual interface is more easy to understand and therefore a more powerful way for the architect to interact with the system.

In addition, there is an existing visual modeler available for Archium called Archimon. Archimon leverages the plugin system that is part

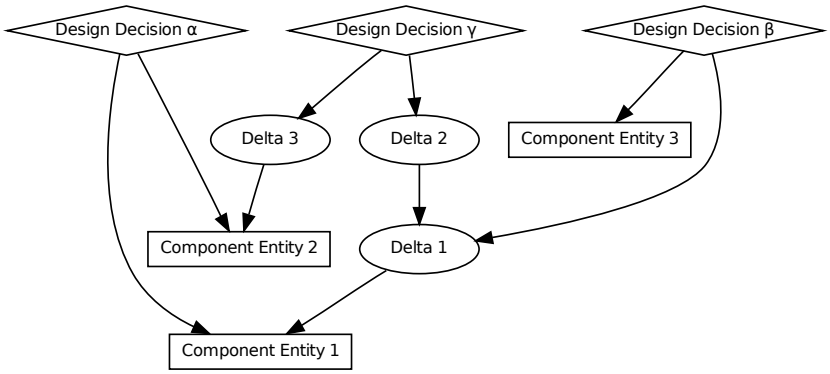

Figure 10: An example of dependencies between Design Decisions. Design Decision α instantiates two Component Entities. Design Decision β then defines a third Component Entity and stacks a Delta on top of the second. Design Decision γ stacks Deltas on top of the first Component Entity (in effect stacking on top of Delta 1) and the second.

of Archium to communicate at runtime with the running system and this provides some limited ability for interaction. Due to the benefits of a visual interface and an existing base to build on, Archimon was selected as host for the Checkpoint functionality.

In addition to a standalone version of the Archimon visualizer, a second version is available as an Eclipse plugin. This provides some additional ease of use in the case that the architect is working with Archium using the Eclipse Integrated Development Environment (IDE). However this version does bring with it a large number of dependencies and interactions with the Eclipse platform, which complicates its implementation. Therefore only the standalone version was selected for extension as part of this thesis.[4]

Archimon makes use of the RMI library to provide a communication link from the visualizer to the running Archium instance. This is necessary because Archimon and Archium proper both run in their own Java virtual machine instance, which means they could equally likely be running side by side on a single machine or on two different machines separated by the Internet.

5.4.1 *Archimon architecture*

Archimon is built using the JGraph Java library which specializes in drawing graphs (in the mathematical sense – that is edges, vertices, etc.). As a result of building on top of a structured foundation of JGraph primitives, Archimon is constructed in an extensible way.

Figure 11 shows the high-level architecture of the Archimon visualizer. The Archium plugin manager class (*PluginManager*) exposes a number of methods which external plugins can take advantage of. This class mainly serves the purpose of enabling the Archium runtime to communicate changes in its state with listening plugins. This happens using an event-based model by firing events in the form of event objects. These objects are received by all plugins that have registered them-

4 Porting the new features to the Eclipse-based version of the plugin should not be hard to do and is left as an exercise for the reader.

selves via the *registerPlugin* method. This is a standard method of the *PluginInterface* interface which all Archium plugins must implement.

In summary, at a high level the Archimon plugin is responsible for creating visualizations of the Archium state by turning incoming events into changes in visual representation.

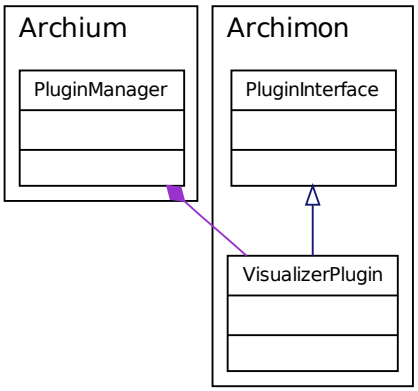

Figure 11: Archimon plugin architecture

5.4.2 *Communicating with Archium*

The Archimon *VisualizerPlugin* class implements the *PluginInterface* interface. It implements the *fireEvent* method, another mandatory method which is responsible for receiving the *AEEvent* objects from the running Archium system. Figure 12 and Figure 13 show the types of events generated by a running instance of Archium and the collaboration between them.

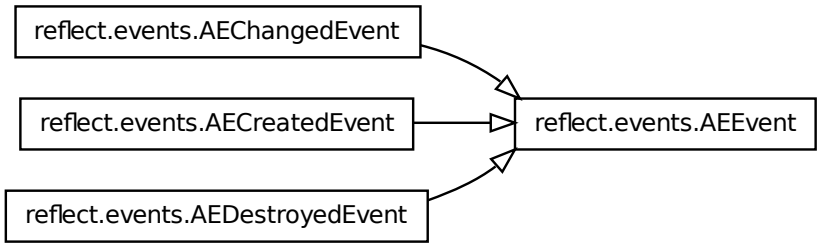

Figure 12: The types of event generated by a running Archium instance and passed to all listening plugins

These event objects are created in Archium's reflection layer. All reflection classes that represent a runtime entity (i.e. which inherit from *ArchitecturalEntity*) fire events at the appropriate times when they are created, destroyed or undergo any other significant change. In fact, the superclass takes care of firing the creation event from its constructor.

The new *Checkpoint* class also had to be modified to fire these events as a prerequisite to building the link with Archimon.

Each *AEEvent* contains a reference to the reflection object responsible for firing it. This object is unpacked by the *VisualizerPlugin* and used to construct a *GraphElement* object. This is a JGraph construct that can be used later on to present the object visually to the user. This activity is illustrated in Figure 14.

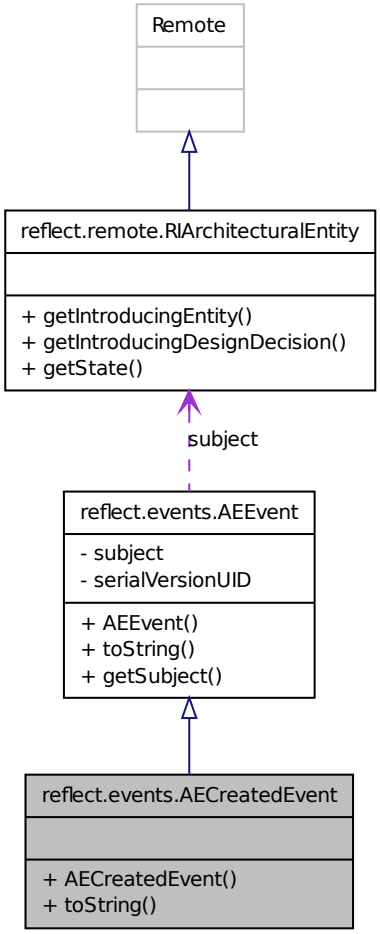

Figure 13: The AECreatedEvent class and its interaction with other classes

It should be mentioned at this point that due to the use of RMI, the Archimon instance that receives a *Checkpoint* object wrapped in an *AEEvent* is actually working with a stub class. This stub class (which is generated by the RMI compiler in the initial compilation phase) transparently redirects each call of one of its methods through the RMI layer to the instance of the true *Checkpoint* object (living inside the Java VM running Archium) waiting to receive it.

5.4.3 *JGraph element*

A new type of *GraphElement* (i.e. a subclass) is introduced to Archimon called *CheckpointElement*. Objects of this class represent a Checkpoint entity from the perspective of the JGraph framework. This works by taking the original Checkpoint instance as a parameter to its constructor and initializing accordingly. The class is illustrated in Figure 15.

The CheckpointElement class also contains useful methods to be used by the other packages of Archimon. These will not have direct access to the embedded Checkpoint instance due to the Model-View-Controller (MVC) nature of JGraph. This class plays the 'Controller' part of the design pattern.

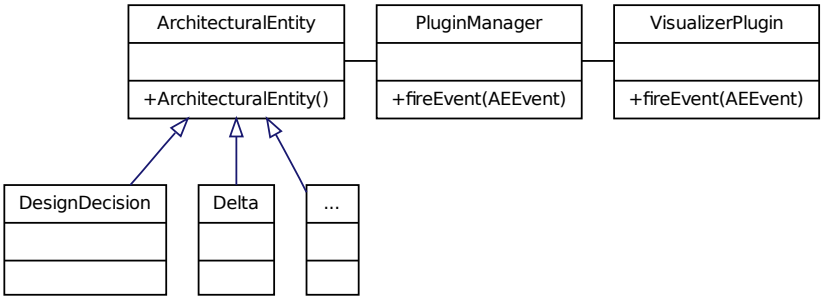

Figure 14

5.4.4 *A new view*

In order to be able to display Checkpoints to the Archimon user in the first place, a new view was created. Currently the following views are available as shown also in Figure 16:

- AllElementView which shows all architectural elements in one single window;
- CheckpointView which shows only Checkpoints;
- ComponentConnectorView which shows only Components and Connectors;
- DesignDecisionView which shows only Design Decisions along with their dependency relationships;
- DesignFragmentView which shows only Design Fragments; and
- ProcessView which shows only Deltas and the threads they are running.

A new class CheckpointViewElement (the 'Model' piece in the MVC pattern) was created to bridge the CheckpointElements into the CheckpointView correctly. Each time a CheckpointElement is created within Archimon, a corresponding CheckpointViewElement is also created and added to the Archimon renderer.

5.4.5 *Graphical rendering*

The third extension of the JGraph framework is the introduction of the CheckpointRenderer class which plays the 'View' piece. It is responsible for drawing a representation of the Checkpoint. The class is illustrated in Figure 17.

The choice was made to represent Checkpoints with triangles, because most other geometric shapes already have an association in the Archimon visualizer (see e.g. the illustrations in Chapter 6).

The border coloring of the Checkpoints was also made conditional depending on which Design Decision is currently highlighted. In this way it is easy to see which Design Decision load was responsible for creating which Checkpoint.

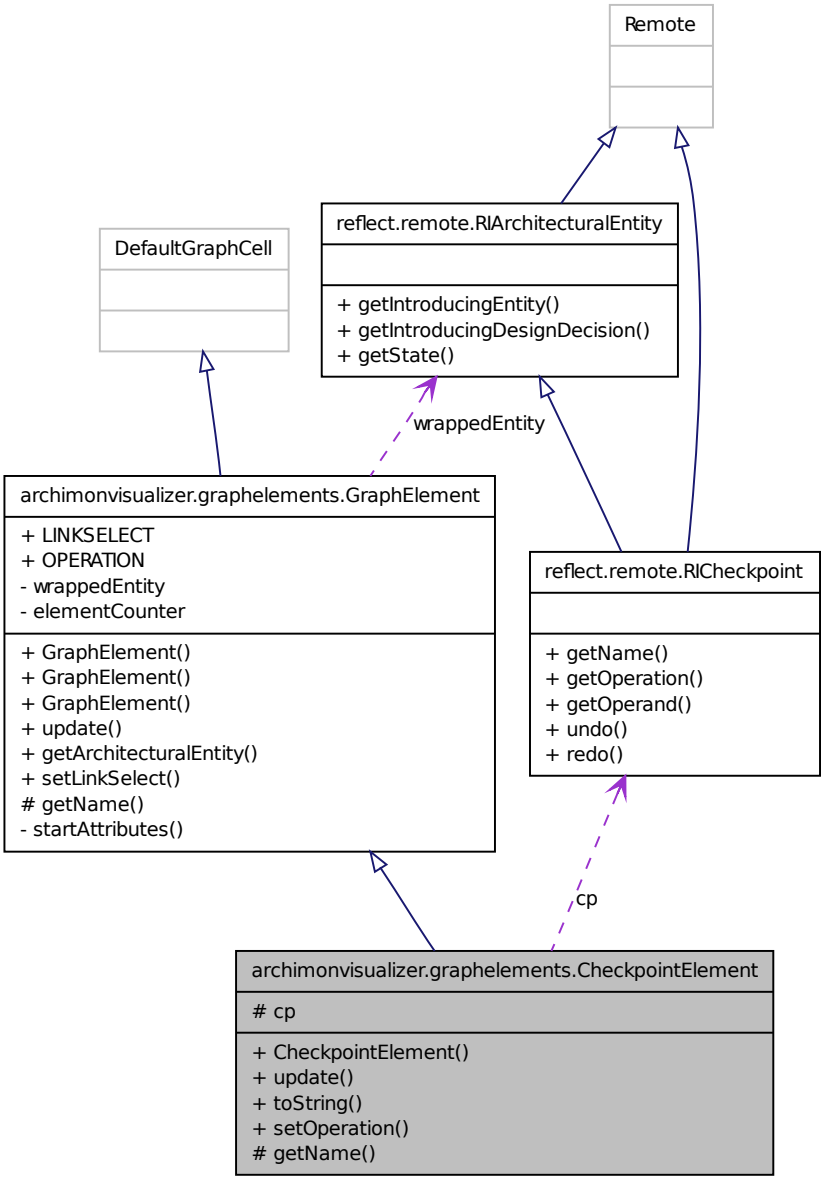

Figure 15: The CheckpointElement forms the basis of the representation of Checkpoints in the Archimon graphical interface.

5.4.6 *Menu entries*

The last remaining task is to make the new functionalities available to the end user of Archimon. This was accomplished by creating a new menu entry titled *Interact*, to give a sense that this is a set of operations that can be performed on a running architecture. The menu contains four entries mapping to the four new functionalities.

The new CheckpointView was added as an option to the *Views* menu.

In order to implement the menu a number of classes had to be created that implemented the right interfaces and callback methods alongside the other menus and menu options.

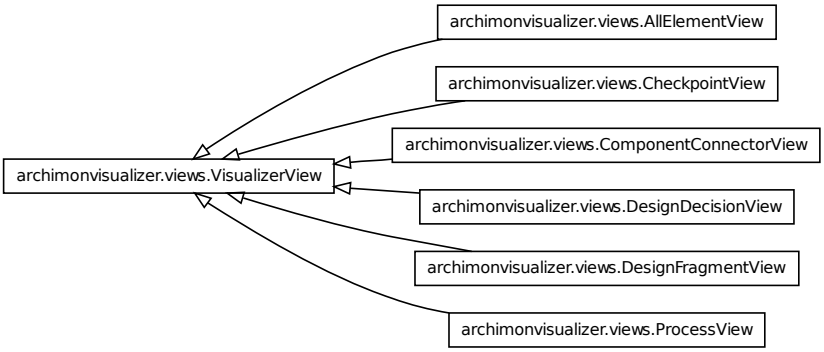

Figure 16: The views available in Archimon.

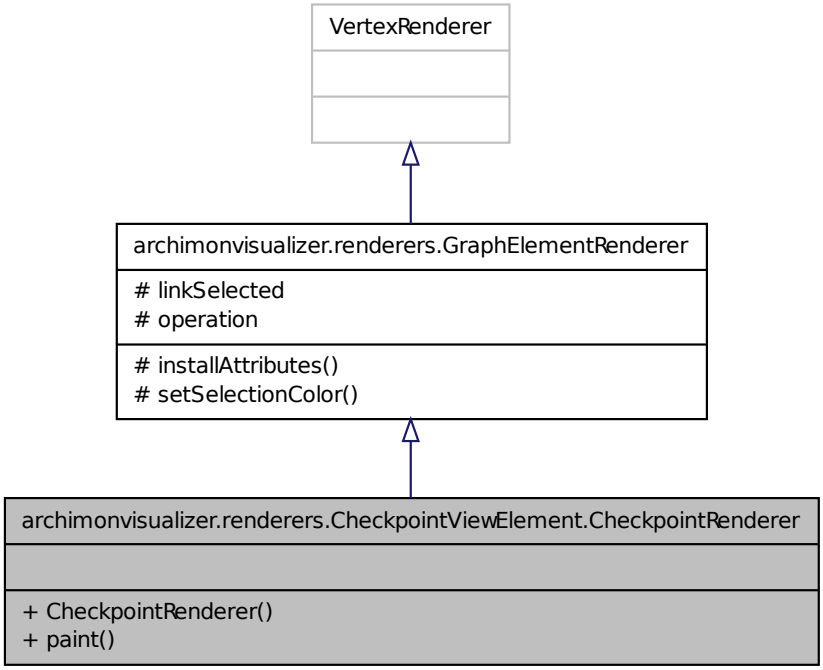

Figure 17: The Checkpoint renderer and its dependencies on related classes.

6 EVALUATING THE IMPLEMENTATION

In this chapter the implementation described in Chapter 5 is evaluated. The new Archium features are tested by applying them to an example architecture. As the Archium platform as a whole is not yet sufficiently stable for the implementation of a full case study-style architecture, a simpler example will be used to demonstrate the way in which an end user can interact with the running system.

Appendix D describes in detail the steps followed in order to reach the results described in this chapter.

6.1 INITIALIZING THE ARCHITECTURE

Assuming that the Archium compiler itself has been compiled from Java source, it is ready to be used to in turn compile Archium source into Java classes.

The first step is therefore to compile the example files using the compiler. The files themselves are listed in Appendix C. Then before the compiled primary Design Decision can be run the Java RMI registry tool must be started using the command `rmiregistry`.

The next step is to start Archimon. Figure 18 shows the initial view after startup. The new *Interaction* menu is visible between *File* and *Views* menus.

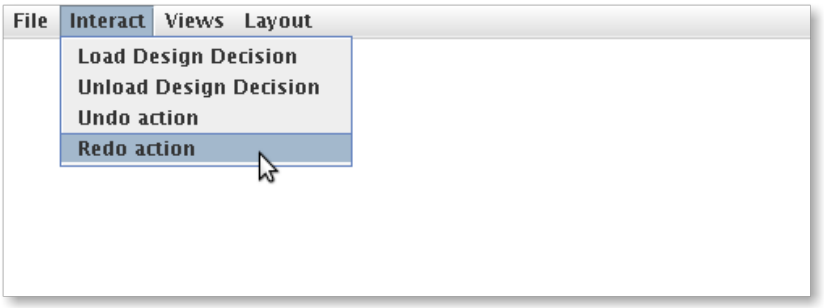

Figure 18: Archimon and the new Interact menu showing the four operations that are now supported

At this point the primary Design Decision can be started up in a second VM. It automatically connects to the RMI registry, searches for the plugin manager and registers itself with Archimon. As the architecture starts to run all the architectural entities are registered with Archimon while they initialize. The Design Decision entities cause a new Checkpoint to be created which can later be used to return to the state the architecture was in before they were loaded.

There is no user interaction in the initial startup phase as control is only handed over to the end user after the entire predefined chain of Design Decisions has completed loading. The loading phase stops once there are no more Design Decisions left that reference other as yet unloaded ones. In this respect much effort has been spent on ensuring the new features do not impact the user experience.

Once the architecture has completed loading, the Checkpoint view can be opened to inspect the checkpoint state. As Figure 19 illustrates, there are two Checkpoints each representing the `LOAD` operation on a Design Decision, in this case `DD1CreateServer` and `DD2FleshOutServer`. The Checkpoints are simply named sequentially starting at zero.

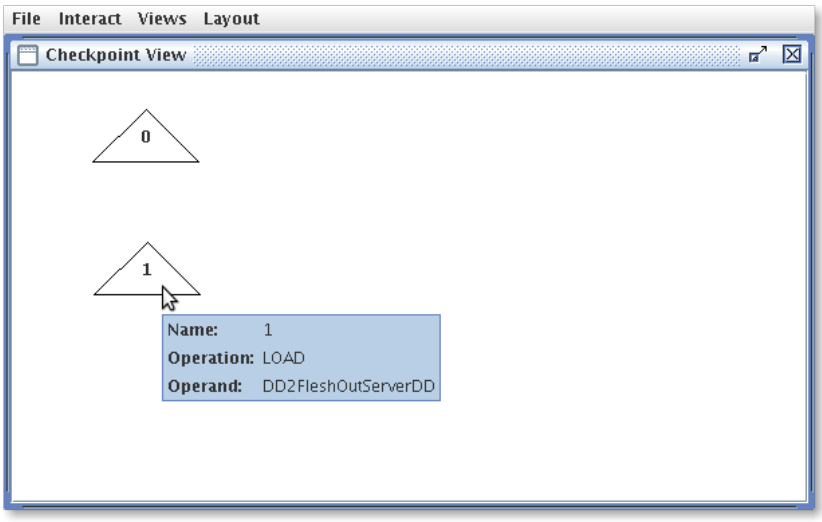

Figure 19: The new Checkpoint view in Archimon showing that the running architecture state contains two Checkpoints

Note that there is no Checkpoint representing the primary Design Decision of the architecture – `DD0InitialSystem`. This is the correct behavior since that first Design Decision should never be unloaded. If it were, there would be nothing left of the architecture itself.

Figure 20 shows the correspondence between the Design Decisions and the Checkpoint entities. Firstly the relationship between the Design Decisions is highlighted in the Design Decision view. While the primary Design Decision is the basis for the architecture, it does not have a direct dependency relationship on any other Design Decision – unlike the second Design Decision which depends on the first.

In addition, the red outline of the entities appears after they have been clicked and shows the relationship between Checkpoints and Design Decisions. In this case the correspondence is `Checkpoint 0` ↔ `DD1CreateServer` and `Checkpoint 1` ↔ `DD2FleshOutServer`.

6.2 UNLOADING A DESIGN DECISION

To illustrate the new functionality a Design Decision is now unloaded from the architecture. As discussed in the previous chapter, the operands to this operation are a Design Fragment representing the running architecture and the Design Decision to remove. While it would be possible to auto-detect the correct Design Fragment this has not been implemented yet, therefore the user has to select the correct entity in the Design Fragment view first (see Figure 21 for an illustration).

The next step is to select the Design Decision to be unloaded. While any Design Decision entity can be safely selected, the operation will only succeed on an entity that has no dependencies on any other running Design Decision. In case a dependent entity is selected, a warning message appears informing the user of which other Design

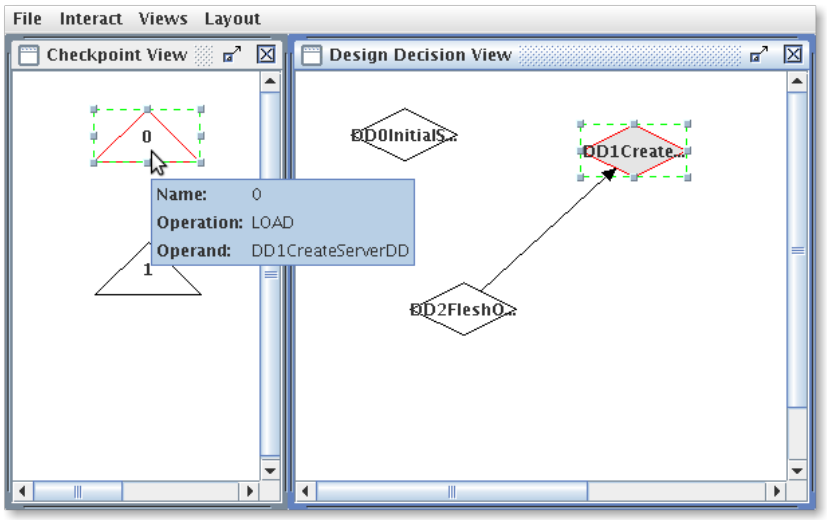

Figure 20: Archimon showing the Checkpoint and Design Decision views side by side. Color coding indicates which Design Decision is associated with which Checkpoint.

Decisions must be unloaded first before the removal of the selected one can succeed.

After selecting the Design Fragment and the Design Decision, the *Unload* action in the *Interact* menu executes the operation. Because this is not an undo or redo operation, this immediately causes a new Checkpoint to be created that describes the unload action and the involved entities. The total number of Checkpoints has now increased to three.

As shown in Figure 21 the operation results in the Design Decision object disappearing from the architecture (and the Design Decision view). The Design Fragment (`FleshOutServer`) introduced by that Design Decision has also been removed leaving only the initial `ServerDF` entity.

6.3 LOADING A DESIGN DECISION

We can now load a Design Decision into the running architecture and observe its effects. In this instance the `FleshOutServerDD` will be loaded back into the architecture. The correct Design Fragment must once again be selected (there is only one in this case) after which the operation *Load Design Decision* can be selected. A file browser window appears (see Figure 22) in which the `.class` file of the Design Decision to be loaded must be selected.

Note that the filename of the entity does not correspond exactly to the name given in the Archium source (`DD2FleshOutServerDD$C`). This is due to the ArchJava representation into which the Design Decision has been compiled in the intermediate stage before being reduced to Java. In some cases two class files can be seen for one entity – in this case the file ending in `$C.class` should always be selected.

The result is shown in Figure 23. A fourth Checkpoint is now created referencing the `LOAD` operation. At this point Checkpoints `1`, `2` and `3` are all referencing the same Design Decision. In this way the state transitions of the architecture are recorded. The behavior implemented

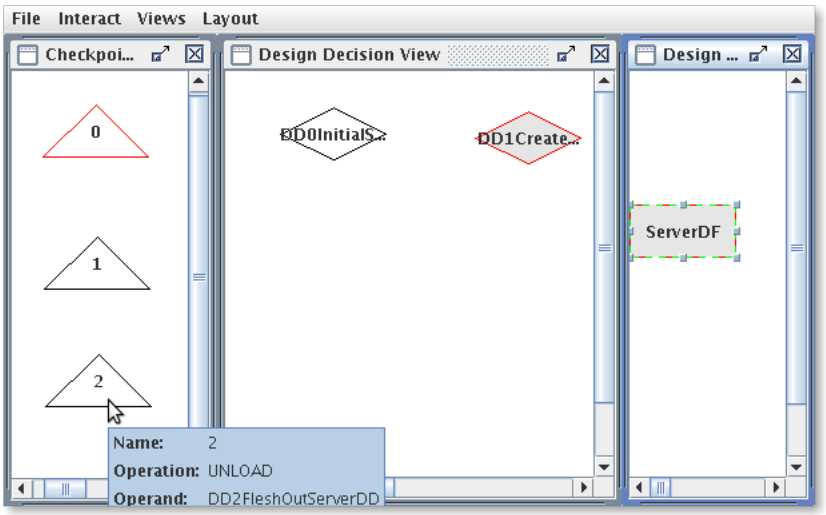

Figure 21: After unloading a Design Decision, there are now three Checkpoint entities. The first two read as `LOAD` operations and the third as `UNLOAD`. Note the absence of the `FleshOutServer` Design Fragment.

in the third Design Decision is now once again started, and the Design Decision and Design Fragment entities are visible in their respective views.

6.4 UNDO & REDO

The undo and redo operations are higher-order operations that leverage the Checkpoint chain to navigate between architecture states. In addition to the chain itself a pointer is used to track where in the chain the architecture state is currently positioned. Before any undo or redo operations have been executed it points to the newest Checkpoint in the chain.

At this point in the example the undo operation could be executed a maximum of three times sequentially. Each time it is invoked the state transition recorded in the Checkpoint to which the pointer points is undone, after which the pointer is decremented. After the third invocation it does not point to a Checkpoint but to the state which existed before `DD1CreateServer` was loaded.

The redo operation functions in reverse. The pointer is incremented and the transition described in the corresponding Checkpoint is executed. In this way, the architecture can transition anywhere along the chain of Checkpoints in a non-destructive way. It is only once a new load or unload operation is carried out that a new Checkpoint entity is created.

When that happens, the Checkpoints 'above' the current entity are discarded, and the newly created Checkpoint is inserted at the top of the chain. This reinforces the view of arrangement into a chain versus that of a tree – there is currently no method of tracking multiple branches of state transitions.

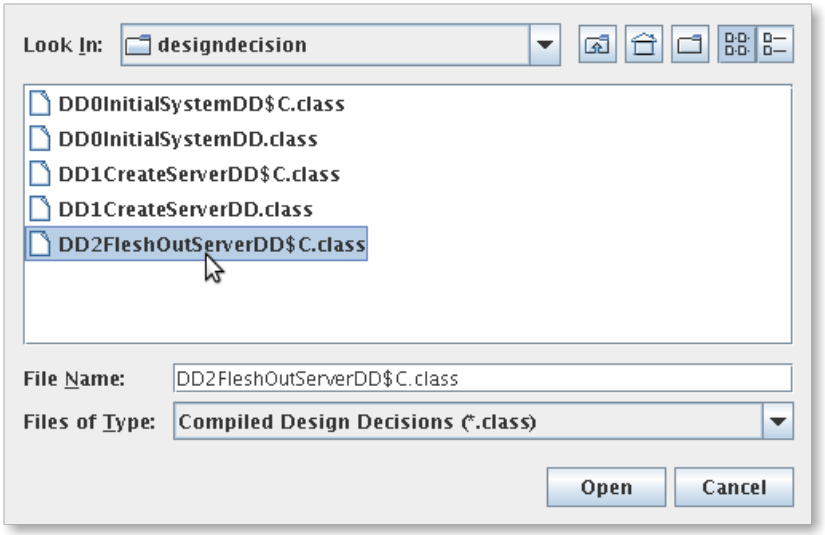

Figure 22: Selecting a Design Decision to load. Here the compiled `.class` file representing the third Design Decision in the architecture is selected using the file manager.

6.5 FULFILLED REQUIREMENTS

At this point we return to the requirements defined in Section 4.7 to ensure they have been met. The first requirement reads "*It must be possible to halt a particular Solution within a Design Decision, undoing its effects on the architecture.*" This has been accomplished through the unload operation. While the operation works at the level of the entire Design Decision, the effect is the same in that it causes the Design Solution chosen within the Decision to be undone.

The second requirement reads "*It must be possible to select a particular Solution within a Design Decision after a previous Solution has been halted.*" The load operation makes this possible. While again it operates at the Decision level instead of the Solution level, the effect is the same. A newly created and compiled Design Decision can be loaded into a running architecture and affect its architectural structure, implementation and behavior. Alternatively, an existing Design Decision can be unloaded, modified, recompiled and then reinserted to gain the effect of switching to a new Design Solution.

The third requirement reads "*There must be a mechanism to request the halting and starting of Solutions within a Design Decision.*". This refers to the necessity of a user interface instead of simply static functionality. The new *Interact* menu is currently the only way to initiate these actions, however future plugins will be able to make use of it due to the exposed interface.

The implementation has thereby satisfied all three requirements. This includes the may-have requirement, although it has an admittedly less direct solution versus the other two (due to the need to recompile an existing Design Decision even if one wants only to switch between its Design Solutions).

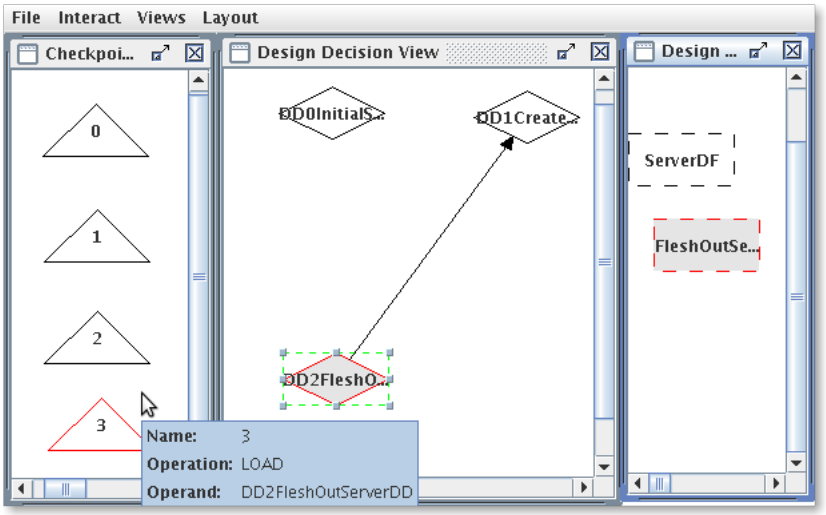

Figure 23: After loading a Design Decision, there are now four Checkpoint entities. The first two read as `LOAD` operations, the third as `UNLOAD` and the fourth as `LOAD`. Note the presence of the `FleshOutServer` Design Fragment.

7 SOLUTION COMPARISON

This chapter relates the improvements to Archium described in the previous chapters to the research identified in Chapter 3 that is closest to Archium's functionalities. The comparison is instructive in that it shows that Archium is not a silver bullet and there are features and functionality that is does not provide and possibly may never be able to provide in its current form.

7.1 THE ARCHIUM APPROACH

In contrast to other existing tools, Archium integrates its AK model directly into Java program code which is then compiled into a functional executable architecture. Because of this code integration Archium continues to be useful in the later stages of the development lifecycle, i.e. design, implementation and maintenance of the software solution.

The approach does however force Archium to operate at a lower level of abstraction that provides sufficient detail to be intimately connected with executable code. While it is possible to create a tree of Design Decisions in Archium that introduce only minimal constraints on the architecture, they bring with them the overhead of entities such as Design Solutions and Design Fragments. It is these entities which are the entry point into the architecture's executable behavior. Archium *does* provide one set of entities somewhat unrelated to runtime behavior which are the Requirements, Requirement Categories and Stakeholders. These are useful to document the requirements that the architecture should meet (along with those that may be considered optional and nice to have), group them into logical groups and identify their external stakeholders.

Other architectural tools which are not tightly integrated with implementation can conceivably capture design decisions at a higher level of abstraction. They are not tied to linking these decisions explicitly to pieces of the architecture's design. Whether this is a disadvantage to the Archium user remains to be seen. The fact that other platforms are not aware of lower level details may at the very least make it difficult to integrate Archium with these, due to a kind of 'impedance mismatch' as can also be seen when attempting to interact with e.g. a functional language using an object-oriented one.

Another consequence is that the end users of Archium must be acquainted with a programming mindset and therefore have a technical background. While there are visual tools such as the Visualizer available these do not support the initial creation of an architecture from scratch. This precludes direct Archium use by non-technical stakeholders such as business managers or other business partners.

Finally, the platform provides no explicit support for teams of users working together to construct an architecture. All Archium artifacts are contained within source code files so a limited level of team collaboration can be achieved by leveraging an existing source code management tool (or revision control system) that also provides source code collabo-

ration features. Beyond that Archium lacks some of the visual features that other systems do include.

7.2 COMPARISON WITH OTHER APPROACHES

In comparison with Archium, the ArchStudio platform provides a much more visual approach to creating an architecture. Its *Archipelago* visual editor allows the end user to visually create and define changes to an architecture. As described in Section 3.8.2 ArchStudio is similar in approach to Archium in two ways.

Firstly, its options schema allows for the indication of variation points in the architecture where objects can be either included or excluded. Secondly, its variant schema allows for the choice amongst a number of alternatives that can be substituted for entities such as components and connectors. While these schemas clearly originate from the more abstract concept of software product lines, they are in some ways similar to the Archium Design Fragment entities which also are combined together to construct the whole architecture. ArchStudio also provides a method of connecting the architecture to a Java implementation although it is not as closely connected as Archium's.

An intriguing property of ArchStudio is moreover that it is self-hosting and has been built incrementally using its own tools and techniques. This is something which unfortunately cannot be said for Archium.

In contrast, the Plastik platform uses an ADL (ACME) reminiscent of the structure and keywords in Archium source files (see Section 3.8.3). However its 3 complementing meta-frameworks work quite differently from those that Archium possesses. An area where it is clearly more advanced than Archium is the formal specification of architectural changes. This is done by specifying action verbs such as 'detach' and 'remove' in ACME and running these as input to a running architecture. These may even be triggered on predefined conditions. An example is the unloading of a video decoding component and the loading of a lower-bandwidth consuming replacement under conditions of network lag. In Archium such a transition can now be fully executed at runtime, but this must be manually executed by the user.

The final body of research that bears resemblance to Archium's scope and functionality is the OpenRec platform (see Section 3.8.5). This is because of its plugin architecture allowing multiple tools to monitor the platform as it is running. It also uses the concept of meta-layers to represent the application component and reconfiguration layers as separate from the running architecture. It appears to offer a high degree of flexibility in terms of how reconfiguration operations should be executed by leaving much up to the end user to define. It therefore offers a different balance of complexity that must be managed by the end user versus flexibility provided. In terms of tools OpenRec provides a visualizer similar to the one included with Archium. Reconfiguration operations however must be triggered through handcrafted reconfiguration scripts. There is also a reconfiguration measurement tool provided that appears to be unique from those in all other platforms. The tool measures the effective downtime of all components involved in a reconfiguration as a result of being paused or replaced, which allows the end user to determine the impact of the reconfiguration. Future reconfigurations can then be adjusted to reduce their impact based on this knowledge.

8 CONCLUSION

This thesis has taken a broad look at what it means to defer decisions affecting the architectural structure of a software system until such time that the system is running. After a review of the software engineering field in question in Chapter 3, the proposed changes in the design of Archium to achieve this goal were illustrated in Chapter 4. This was followed by an implementation of the changes described in Chapter 5 and a review of the results in Chapter 6.

8.1 SUMMARY

This section relates the discussions in the previous chapters to the research questions posed in Chapter 2 to show these have been adequately resolved. By answering the individual research questions the overarching problem statement of *How to make runtime architectural design decisions?* in turn answered.

The first research question was "*What are architectural design decisions and how can they be represented?*". This was addressed in Chapter 3 by reviewing the common definitions and models used to derive the utility of software adaptability and the design decision concept. The key features of a design decision as an entity that captures architectural knowledge and interconnects this knowledge to prevent vaporization of that knowledge and subsequent architectural erosion were identified.

The second research question was "*What approaches exist that allow application of design decisions to a running system?*". Following from the definition of the concept, existing research software tools and how these are using the concept of a design decision were reviewed (also in Chapter 3). The approaches ranged from those focused solely on capturing architecture knowledge (thereby separating this concern from the implementation of the system) to those allowing for reconfiguration of the architecture at runtime. No existing research combining both approaches was found, showing the viability of this thesis to contribute to this field.

The third research question was "*What are the challenges in applying design decisions at runtime?*". In Chapter 4 an analysis was given to answer this question. Various factors influencing the safety and stability of runtime reconfiguration were identified and functionality necessary for success (such as reflection capability) was also discussed. These concerns were then applied to the existing Archium platform, leading to the identification of a set of requirements to be met by implementing changes.

The fourth research question was "*How can a system be implemented that allows runtime application of design decisions?*". The implementation of the proposed changes was discussed in Chapter 5. Three areas of the Archium platform needing modification were the reflection layer, the introduction of a new checkpoint layer and the creation of a user interface for the functionality in the visualizer. A number of challenges were encountered and overcome during the implementation phase.

The fifth and last research question was "*How well does such a system perform?*". Chapter 6 provides an answer to this question by looking at a scenario in which the new functionality is used, showing it works as expected. Additionally Chapter 7 returns to the related works found to be most similar to Archium to provide a comparison of their functionality with that of the newly improved Archium platform.

In summary – before the work described in this thesis was started, Archium was only capable of very limited runtime composition of Design Decisions, taking place once at the initial loading of the architecture. After the changes implemented in this thesis it is possible to safely perform compositions at any later time during execution and also to undo the effects of a previous composition step or set of steps.

8.2 REFLECTION

Why is there a need for another programming language, and one which adds complexity and overhead? In essence Archium is meant to tackle large scale software systems by easing the design, maintenance and runtime adaptability all at once. This has the potential to reduce the end-to-end cost of design through maintenance and thereby the total cost of ownership. It is also expected that with the future advances in compiler optimizations and execution speed the overhead introduced by the language will be minimized.

Archium brings a number of features which are very different, one could almost say orthogonal, to classic (object-oriented) programming. This is no surprise and is a consequence of the architectural functionality integrated into the platform. This does mean that the learning curve for developing using Archium is somewhat steep. Above and beyond the well-known component and connector model, one must learn how to use the additional complexity and entities such as Composition Configurations and Design Fragment Compositions. Also the Delta as an entity that represents change to a set of other entities is a new concept to get used to.

The complexity of extending and improving Archium is higher still. The platform is clearly in the research stage as it unfortunately suffers from a few of the things it tries to solve. It takes a significant amount of time to reach a full understanding of the internal structure of the compiler and surrounding toolset. It then takes more time to be able to start making changes to the implementation.

The most challenging part of the work completed in this thesis was the design of the runtime loading and unloading of Design Decisions. A fair amount of reverse engineering and runtime debugging was required to fully understand the inner workings of the reflection layer.

The implementation itself has been worked on by various students and researchers. While the usefulness has steadily increased with time, signs of architectural erosion are evident in places. Many parts of the code use complex conditionals to determine function calls, accompanied with notes of suboptimal implementation due to time constraints. While specific parts of the implementation are well or even excellently documented, other areas suffer from knowledge vaporization.

It certainly can be said that Archium itself is a good example of the type of system which could benefit enormously from its own features. If Archium were to become self-hosting at some point in the future this might lead to a powerful development platform. The only other

obstacle to adoption by architects and developers might be the overhead of the Archium-specific parts of the implementation, such as needing to break Design Decisions into Solutions, with accompanying rationale for the choice that was made. It is hoped that this thesis gives more than sufficient arguments for why this overhead (whether incurred by using Archium or otherwise) is necessary for the long term success of the software architecture field.

8.3 FUTURE WORK

There are many further improvements that could be made to the Archium platform beyond those described in this thesis. These will be needed to make it a viable platform that software architects are willing to use in practice. They can be broken down into the following areas.

8.3.1 *Platform stability*

The stability of Archium needs be improved further. There are exist many corner cases in which correct source code is not compiled correctly or at all. Additionally there are several weak areas in the runtime platform remaining, such as the Composition Configuration layering mechanism that facilitates communication between Deltas and Connectors. While certain features may work in separation, combining these can lead to unpredictable results. In some cases the instability was an impediment to the work carried out in this thesis, and necessitated additional time spent on fixing bugs and improving stability.

The continuing dependence of Archium on ArchJava for providing a component and connection integrity layer should also be mentioned here. Many Archium source files are compiled first to a ArchJava source in an intermediate step before they are recompiled once again as Java class files. This causes code bloat and unfortunately is becoming a source of architectural erosion for the platform as a whole. It also stands in the way of freeing up the platform to enable more complex manipulations, for example the ability to use Deltas to manipulate the linkage between Component Entities and Connectors and disconnect or reconnect these dynamically.

8.3.2 *Platform features*

Once a higher level of stability has been achieved, those aspiring to extend the platform may want to consider adding features that resemble those offered by more traditional programming languages. Examples include dynamic linking, packages, an exception mechanism and IDE integration.

The method of loading and unloading Design Decisions implemented in this thesis shows a clear need for the ability to dynamically link Archium entities at runtime. It's currently not possible to compile a Design Decision without the source code for all the entities it depends on (other Design Decisions and their dependencies) being available at compile-time. This places a burden on the developer to ensure that the source code is available. Also the Archium environment must be setup so that the source files can be found by the compiler. In some scenarios it might be desirable to distribute an architecture without

its source code, e.g. for commercial purposes. Dynamic linking would remove these limitations and thereby make it much simpler to extend an architecture at runtime.

One of the potential risks of dynamic linking is symbol mismatches if two different architectural entities have the same name. A solution would be to implement full support for 'packages', at the same time adding a useful feature. Currently there is no scoping of architectural entities, which means that all Archium entities share a single namespace. While this may be inconvenient for a single developer designing a large architecture who needs to spend time ensuring entity names are not reused, further difficulties arise when multiple developers want to develop parts of an architecture which are later intended to be merged. Being able to create namespaces to keep entities separated would increase interoperability. An approach similar to that of Java could be taken where packages are declared explicitly and used by the linker in the linking stage.

Another feature that is common to object oriented languages is the concept of exceptions that get triggered if an error condition is encountered. They work to ensure that error conditions cannot leave the program in an undetermined or inconsistent state by forcing the explicit handling of the situation. This concept could be applied to Archium to increase its safety and user-friendliness. Examples where this could be used include attempting illegal operations such as adding a Delta to a non-existent Component Entity or connecting one port to another that does not share the same interface.

The final feature that could make it easier to work with Archium would be better integration with its favored IDE, Eclipse. There is a certain level of integration already present as Archium includes an Eclipse plugin that provides a limited visualizer plugin for the runtime. The implementation is also separate from the visualizer that was extended in this thesis which means it does not benefit from improvements made to the external visualizer. Ideally the integration with Eclipse could make it possible to build a debugger that could trace the execution of a running Archium architecture in more detail and allow for direct interaction with all its entities.

8.3.3 *Architectural concepts*

It can be argued that the functional features described above do little more than provide a higher level of comfort to the developer. Other than those there are also a number of improvements that could be made to the architectural model offered by Archium to increase the expressive power it offers to the user.

For one, while Archium treats both Component Entities and Connectors as distinct classes of entities, it seems unbalanced that Deltas can provide runtime behavior to the former but not the latter. Connectors currently provide only a conduit for communication and are not capable of transforming the information they pass along.

Once basic transformation capability is available using Connectors, it's not much of a stretch to imagine that this could be leveraged for network-transparent communications, compression and other such features. Being able to run parts of an architecture on various different computers open up intriguing possibilities. The need for the developer

to bother with implementation details involving protocol design etc. would disappear.

Thirdly, the requirements model that tracks how the implementation of Design Decisions satisfies Requirements could be extended further. The Requirements are not represented in the current version of the Visualizer and cannot be monitored while the architecture is running.

The final proposed area where the architecture model could be improved by implementing a concept from the broader programming field is inheritance between entities. Currently no reuse is possible below the component level and inheritance would be one way of resolving that situation. For example, instead of having to assign the same Delta to each and every instance of a particular Component Entity, it would be more efficient to create a parent entity which has the Delta assigned to it, which can then be 'subclassed' as it were as many times as required. This would also allow for the reuse of architectural patterns on a larger scale by subclassing a Component Entity that has already had its internal structure refined with other Component Entities, Connectors and other entities.

Part II

APPENDICES

A ARCHIUM TERMINOLOGY

Checkpoint	A representation of the state of the running system that includes functionality for modifying that state. The state representation includes the available and currently loaded design decisions and the dependencies between them. The state may be altered by loading and/or unloading Design Decisions from the system in a safe way. Any alteration results in a new Checkpoint instance. Checkpoints thereby allow the state of the architecture to be saved so that future changes can be undone or redone as necessary. Checkpoints are the primary subject of this thesis.
Composition Configuration	Defines in detail how aspects of a Design Fragment Composition should be implemented, with the use of various composition techniques. Is instrumental in gluing a Delta together with a Component Entity.
Composition Technique	A particular method to compose a Delta with a Component Entity (which may result in stacking the Delta on top of preexisting Deltas which the Component Entity is running). A technique can define e.g. whether the Delta's ports are added to the Component Entity, or whether preexisting ports are removed.
Component Entity	Logical architectural building block that provides a base on which system functionality (encapsulated in Deltas) can run. Component entities are an abstraction of components (similar to the class/object relationship) and themselves provide no inherent functionality. They provide the structure of the architecture.
Connector	Links two Component Entities together in order for information to be exchanged between them. Unlike Component Entities, Connectors cannot be extended to provide additional functionality. Every port on a Connector must share an interface with the corresponding port of a Component Entity.

Delta — Provides runtime behavior in the form of runnable threads. Deltas must be instantiated within a Component Entity and can extend it by adding, removing or changing its Ports. A number of Deltas can be stacked on top of each other within a single Component Entity to further modify it's behavior and to provide a logical breakdown into other Component Entities and Connectors.

Design Decision — The top-level entity in Archium within which all other entities reside, and which supplies the content and rationale for, and chooses from, one or more Design Solutions. Design Decisions represent a set of changes to an architecture and take as input a Design Fragment that represents that architecture. The output is again a Design Fragment with the changes incorporated.

Design Fragment — A container in which architectural entities such as Component Entities, Deltas, Connectors, Composition Configurations etc. can be placed. A Design Fragment represents a (part of a) software architecture by grouping together a logical set of entities (which e.g. fulfill a requirement or implement a design pattern).

Design Fragment Composition — Maps the changes specified in a Design Decision to an existing Design Fragment, e.g. which Deltas should be stacked on top of which Component Entities.

Design Solution — A solution to a problem as described in a Design Decision, which may supply rationale such as pros, cons and constraints. Design Decisions often contain multiple Design Solutions out of which one is picked along with a rationale for that choice. The selected Design Solution determines which Design Fragment is applied to the running system by its host Design Decision.

Interface — A specification of method signatures that identify the methods which may be called through a particular Port.

Port	An endpoint that exists on both Connectors and Component Entities to attach these to each other. Ports are either of the *required* or *provided* type, depending on whether functionality from another Component Entity is required for correct operation of the Port's host Component Entity; or vice versa. Ports implement an Interface to define how communication must occur.
Requirement	Represents a system requirement and tracks its fulfillment status during the execution of the architecture. It is introduced by a Design Decision and distinguishes between functional and non-functional requirements. It classifies these according to priority (could have, should have, must have). Requirements may depend on other requirements and may refine them. A Requirement is usually linked to a Stakeholder.
Requirement Category	Requirements may be grouped into categories in any way desired.
Stakeholder	A party which desires a Requirement to be met by the architecture.

The diagram below shows the relationships between the entities in the Archium architectural model.

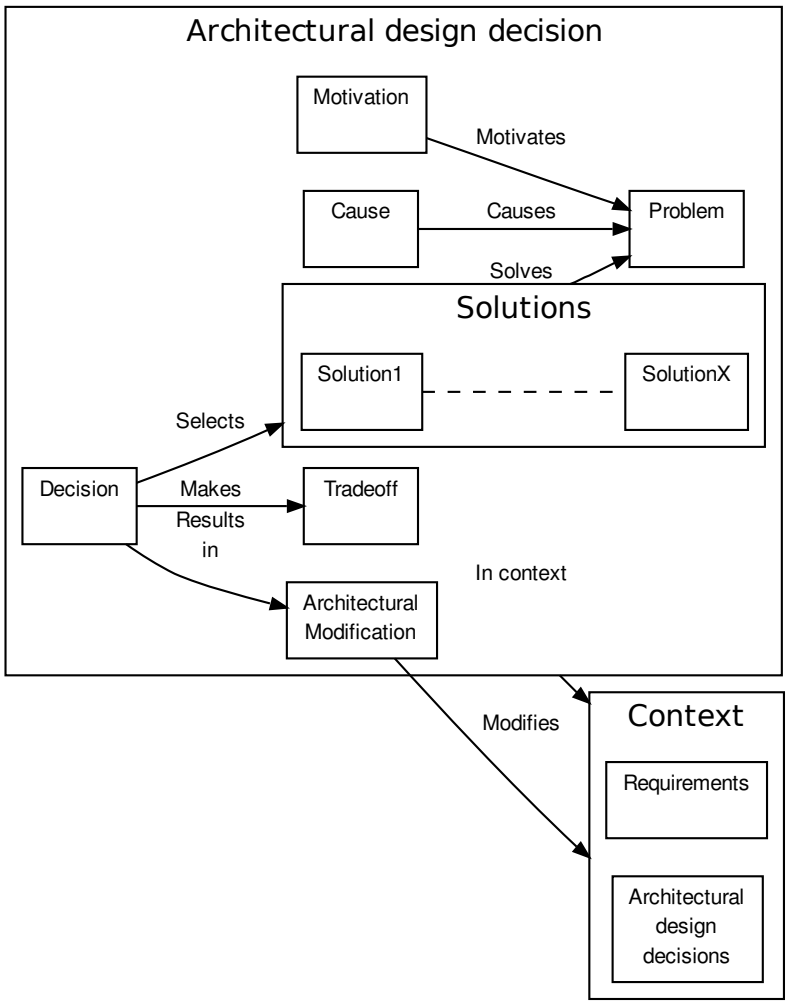

Figure 24: Design Decision model

A Design Decision contains multiple optional fields describing the architect's rationale. Within the mandatory Design Solution block, one or more solutions can be defined – one of which is chosen to be selected and executed.

C

EXAMPLE ARCHIUM SOURCE CODE

The listings below are those used in Chapter 6 to evaluate the improvements made to Archium in this thesis.

```
design decision DD0InitialSystemDD() {
  potential solutions {

    solution InitialSystem {

      architectural entities {
        DD1CreateServerDD dd1DD;
        DD2FleshOutServerDD dd2DD;
      }

      realization {
        dd1DD = new DD1CreateServerDD();
        dd2DD = new DD2FleshOutServerDD(dd1DD);
        return dd2DD;
      }
    }
  }

  decision {
    decision InitialSystem;
  }

}
```

Listing 10: DDoInitialSystem.archium

```
design decision DD1CreateServerDD() {

  potential solutions {
      solution ScratchServer {

      architectural entities {
        ServerDF serverDF;
      }

      realization {
        serverDF = new ServerDF();
        return serverDF;
      }

    }

    }

    decision {
```

```
      decision ScratchServer;
    }

}

component entity ServerCE;

design fragment ServerDF {
  architectural entities {
    ServerCE serverCE;
  }

  initialization {
    serverCE = new ServerCE();
  }
}
```

Listing 11: DD1CreateServer.archium

```
design decision DD2FleshOutServerDD(DD1CreateServerDD
    createServerDD) {

  potential solutions {
    solution FleshOutServer {
      architectural entities {
        FleshOutServerDF  fleshOutServerDF;
        FleshOutServerDFC fleshOutServerDFC;
      }

      realization {
        fleshOutServerDF = new FleshOutServerDF();
        fleshOutServerDFC = new FleshOutServerDFC(
            createServerDD, fleshOutServerDF);
        return createServerDD composed with fleshOutServerDF
            using fleshOutServerDFC;
      }
      }
  }

  decision {
    decision FleshOutServer;
  }
}

design fragment FleshOutServerDF {
  // the DF FleshOutServerDF contains one Delta
  architectural entities {
    FleshOutServerD fleshOutServerD;
  }

  initialization {
    fleshOutServerD = new FleshOutServerD();
  }
}
```

```
component entity CommunicatorCE;
component entity ClientHandlerCE;

delta FleshOutServerD {
  // the Delta FleshOutServerD contains two CE's
  variables {
    CommunicatorCE communicatorCE;
    ClientHandlerCE clientHandlerCE;
  }

  initialization {
    communicatorCE = new CommunicatorCE();
    clientHandlerCE = new ClientHandlerCE();
  }
}

design fragment composition FleshOutServerDFC(
    DD1CreateServerDD targetDD, FleshOutServerDF sourceDF) {
  // Flesh out the server Component Entity (introduced in
      Design Decision DD1CreateServerDD)
  // using Delta fleshOutServerD contained in Design Fragment
       sourceDF.

  composition {
    // compose the fleshOutServerD Delta with the serverCE
        Component Entity
    // stack the delta fleshOutServerD on top of serverCE
    // no Composition Configuration needed as there are no
        ports?
    targetDD::serverCE plays sourceDF::fleshOutServerD;
  }
}
```

Listing 12: DD2FleshOutServer.archium

D

NEW FUNCTIONALITY TEST CASE

1. Start `rmiregistry`
2. Start Archimon
3. Start the example architecture
4. Open *All Elements* view showing 3 Design Decisions and 2 Checkpoints (among other entities)
5. Close *All Elements* view and open *Design Decision*, *Design Fragment* and *Checkpoint* views. Ensure color coding works when selecting Checkpoints and/or Design Decisions.
6. Select Design Fragment `ServerDF` and *unload* Design Decision `DD2FleshOutServer`.
7. Refresh the appropriate views.[1]
8. Verify the creation of a third Checkpoint.
9. Select Design Fragment `ServerDF` and *load* Design Decision `DD2FleshOutServer`.
10. Verify the creation of a fourth Checkpoint.
11. Select the *undo* operation.
12. Verify that no Checkpoints have been created or destroyed while there are only one Design Fragment and two Design Decisions executing.
13. Select the *redo* operation.
14. Verify that no Checkpoints have been created or destroyed while there are two Design Fragments and three Design Decisions executing.

1 This step may be needed after several other steps due to a bug in the view refresh functionality.

BIBLIOGRAPHY

[1] Nour Ali, Carlos Solís, and Isidro Ramos. Comparing architecture description languages for mobile software systems. In *SAM '08: Proceedings of the 1st international workshop on Software architectures and mobility*, pages 33–38, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-022-7. doi: http://doi.acm.org/10.1145/1370888.1370897. (Cited on page 19.)

[2] Paris Avgeriou, Philippe Kruchten, Patricia Lago, Paul Grisham, and Dewayne Perry. Architectural knowledge and rationale: issues, trends, challenges. *SIGSOFT Softw. Eng. Notes*, 32(4):41–46, 2007. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1281421.1281443. (Cited on pages 15 and 17.)

[3] Paris Avgeriou, Philippe Kruchten, Patricia Lago, Paul Grisham, and Dewayne Perry. Sharing and reusing architectural knowledge – architecture, rationale, and design intent. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, pages 109–110, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2951-8. doi: http://dx.doi.org/10.1109/SHARK-ADI.2007.4. (Cited on page 17.)

[4] Muhammad Ali Babar and Ian Gorton. A tool for managing software architecture knowledge. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, page 11, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2951-8. doi: http://dx.doi.org/10.1109/SHARK-ADI.2007.1. (Cited on page 21.)

[5] Muhammad Ali Babar, Remco de Boer, Torgeir Dingsoyr, and Rik Farenhorst. Architectural knowlege management strategies: Approaches in research and industry. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, page 2, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2951-8. doi: http://dx.doi.org/10.1109/SHARK-ADI.2007.3. (Cited on page 14.)

[6] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. *EWSA*, 2005: 1–17, 2005. (Cited on page 25.)

[7] Remco de Boer, Rik Farenhorst, Patricia Lago, Hans van Vliet, Viktor Clerc, and Anton Jansen. Architectural knowledge: Getting to the core. In *International Conference on Quality of Software-Architectures (QoSA)*, page 18, 2007. (Cited on page 23.)

[8] Jan Bosch. Superimposition: A component adaptation technique. *Information and Software Technology*, 41(5):257–273, 25 March 1999. URL `citeseer.ist.psu.edu/281749.html`. (Cited on page 34.)

[9] Jan Bosch. Software architecture: The next step. In *First European Workshop on Software Architecture (EWSA 2004)*, pages 194–199. Springer-Verlag, 2004. (Cited on page 16.)

[10] Wouter-Tim Burgler and Marnix Kok. Improving Archium – taking Archium's code generation to the next level. Technical report, Hanzehogeschool Groningen, 2006. (Cited on page 31.)

[11] Rafael Capilla, Francisco Nava, Sandra Pérez, and Juan Dueñas. A web-based tool for managing architectural design decisions. *SIGSOFT Softw. Eng. Notes*, 31(5):4, 2006. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1163514.1178644. (Cited on pages 17 and 21.)

[12] Shang-Wen Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008. (Cited on pages 8 and 30.)

[13] Carlos Cuesta, Pablo de la Fuente, and Manuel Barrio-Solárzano. Dynamic coordination architecture through the use of reflection. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 134–140, New York, NY, USA, 2001. ACM. ISBN 1-58113-287-5. doi: http://doi.acm.org/10.1145/372202.372298. (Cited on page 12.)

[14] Eric Dashofy, Hazel Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. Archstudio 4: An architecture-based meta-modeling environment. *29th International Conference on Software Engineering*, 2007. (Cited on page 24.)

[15] Allen Dutoit and Barbara Paech. Rationale management in software engineering. *Handbook on Software Engineering and Knowledge Engineering*, 2001. URL `citeseer.ist.psu.edu/dutoit00rationale.html`. (Cited on pages 13 and 15.)

[16] David Garlan and Mary Shaw. An introduction to software architecture. *Advances in Software Engineering and Knowledge Engineering*, 1, 1993. (Cited on page 10.)

[17] Robert Glass. The Standish report: does it really describe a software crisis? *Commun. ACM*, 49(8):15–16, 2006. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1145287.1145301. (Cited on page 13.)

[18] Paul Grisham, Matthew Hawthorne, and Dewayne Perry. Architecture and design intent: An experience report. In *SHARK-ADI '07: Proceedings of the Second Workshop on SHAring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, page 12, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2951-8. doi: http://dx.doi.org/10.1109/SHARK-ADI.2007.4. (Cited on page 17.)

[19] Paul Grünbacher, Alexander Egyed, and Nenad Medvidovic. Reconciling software requirements and architectures: The CBSP approach. In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, pages 202–211, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1125-2. (Cited on page 20.)

[20] Morten Hansen, Nitin Nohria, and Thomas Tierney. What's your strategy for managing knowledge. *Harvard Business Review*, 77(2): 106–117, 1999. (Cited on page 14.)

[21] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *Proceedings of the 26th International Conference on Software Engineering*, page 603, 2004. (Cited on page 27.)

[22] Anton Jansen. *Architectural Design Decisions*. PhD thesis, University of Groningen, 2008. (Cited on page 14.)

[23] Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. *wicsa*, 0:109–120, 2005. doi: http://doi.ieeecomputersociety.org/10.1109/WICSA.2005.61. (Cited on pages 17, 31, and 35.)

[24] Frans Kremer. Developing the Archium platform. Traineeship report search group, University of Groningen, 2004. (Cited on page 31.)

[25] Frans Kremer and Joris van der Burgh. Merging Archium with Java. Master's thesis, University of Groningen, January 2005. (Cited on page 31.)

[26] Philippe Kruchten. Architectural blueprints – the "4+1" view model of software architecture. *IEEE Software*, 12(6):42–50, nov 1995. URL `citeseer.ist.psu.edu/kruchten95architectural.html`. (Cited on pages 10 and 17.)

[27] Philippe Kruchten, Patricia Lago, and Hans van Vliet. Building up and reasoning about architectural knowledge. In *QoSA*, pages 43–58, 2006. (Cited on pages 15 and 16.)

[28] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. *ACM SIGSOFT Software Engineering Notes*, 21(6):3–14, 1996. (Cited on page 18.)

[29] Chris McMahon, Alistair Lowe, and Steve Culley. Knowledge management in engineering design: personalization and codification. *Journal of Engineering Design*, 15(4):307–325, 2004. doi: 10.1080/09544820410001697154. (Cited on page 14.)

[30] Nenad Medvidovic. ADLs and dynamic architecture changes. In *Foundations of Software Engineering*, pages 24–27. ACM New York, NY, USA, 1996. (Cited on pages 18 and 24.)

[31] Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, pages 70–93, 2000. (Cited on page 10.)

[32] Nenad Medvidovic, David Rosenblum, and Richard Taylor. A language and environment for architecture-based software development and evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 44–53, 1999. (Cited on page 24.)

[33] Hong Mei and Gang Huang. PKUAS: An architecture-based reflective component operating platform. In *10th IEEE International Workshop on Future Trends of Distributed Computing Systems, 2004. FTDCS 2004. Proceedings*, pages 163–169, 2004. (Cited on page 26.)

[34] Peyman Oreizy. Issues in the runtime modification of software architectures. Technical report, Department of Information and Computer Science, University of California, Irvine, CA 92697, August 1996. (Cited on page 18.)

[35] Peyman Oreizy, Michael Gorlick, Richard Taylor, Dennis Heimbigner, Gregory Johnsona, Nenad Medvidovic, Alex Quilici, David Rosenblum, and Alexander Wolf. An architecture-based approach to self-adaptive software. *IEEE INTELLIGENT SYSTEMS*, 1094-7167:54–62, 1999. (Cited on page 18.)

[36] Dewayne Perry and Paul Grisham. Architecture and design intent in component & COTS based systems. In *ICCBSS '06: Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (ICCBSS'05)*, page 155, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2515-6. doi: http://dx.doi.org/10.1109/ICCBSS.2006.6. (Cited on page 17.)

[37] Yang Qun, Yang Xian-Chun, and Xu Man-Wu. A framework for dynamic software architecture-based self-healing. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–4, 2005. (Cited on page 19.)

[38] Robert Slagter. Maturing the Archium compiler. Master's thesis, University of Groningen, July 2006. (Cited on page 31.)

[39] Antony Tang, Yan Jin, and Jun Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, 80(6):918–934, 2007. doi: 10.1016/j.jss.2006.08.040. (Cited on page 22.)

[40] Antony Tang, Paris Avgeriou, Anton Jansen, and Rafael Capilla. A comparative analysis of architecture knowledge management tools. In *Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008) – to be published*, 2008. (Cited on pages 12 and 16.)

[41] Sebastian Thöne. *Dynamic Software Architectures: A Style-Based Modeling and Refinement Technique with Graph Transformations*. PhD thesis, University of Paderborn, October 2005. (Cited on page 19.)

[42] Jilles van Gurp. *On The Design & Preservation of Software Systems*. PhD thesis, University of Groningen, 2003. (Cited on page 14.)

[43] Qun Yang, Xianchun Yang, and Manwu Xu. A mobile agent approach to dynamic architecture-based software adaptation. *SIGSOFT Softw. Eng. Notes*, 31(3):1–7, 2006. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1127878.1127889. (Cited on page 19.)

COLOPHON

This thesis was typeset with LaTeX using the `classicthesis` package with *Palatino* and *Euler* type faces and *Bera Mono* for the listings. Diagrams were created using `Graphviz`.