# Maturing the Archium compiler

Graduation thesis

Robert Slagter

July 19, 2006

## Abstract

Software architectural design forms an important part of today's software development processes. By clearly defining the blueprint of a software system, it is a lot easier to maintain and modify.

The focus with current architectural design processes is too much on the results, rather than on the steps which led to these results. Archium is a programming language which allows for the specification of architectural design results, the rationale explaining the design steps, and the implementation of the architecture.

In order to make the language more mature, a number of extensions have been developed: a requirements model has been added, object referencing from rationale has been made possible, a managed threading model has been added and the visualization of running Archium systems has been enhanced and extended with the possibility of applying Design Decisions at run-time.

# Contents

# Chapter 1

# Introduction

Software architectures have become increasingly important over the last few years in software development [21]. Especially in larger projects, it is now generally agreed upon to first develop a software blueprint before starting work on its implementation. Too much time, money and effort has been spent on problems which could have been prevented by clearly defining the architecture of a software system.

With the architectural design phase now being recognized as an essential part of the successful development of a software project, new problems have been introduced as well. A problem which is of particular importance is the problem which arises from the fact that most architectural design is focused too much on the results of the design process, instead of also focusing on the design steps which have led to the results. These steps are often not clearly documented, or "stored" only in the minds of the software architect(s). This may lead to large problems when modifications to the software system have to take place, and the original software architect is not available (anymore). Modifying a system without (in detail) knowing why it was built as it is proves to be a very difficult task.

Related to this problem is the problem of design erosion where the design documents are not kept synchronized with the actual implemented system. Again, this may lead to confusion and issues such as:

- the actual system having been modified without updating the design documents;
- the design documents having been updated without implementing the changes in the actual system.

Summarizing, there are a lot of reasons for documenting the software architecture of a system, not only by explaining the results but also the rationale leading up to the results. In addition, updating and maintaining these design steps and keeping them synchronized with the actually implemented system should be as highly integrated with the design process as possible. This is where the Archium [2] language fits in.

Archium is a proof-of-concept programming language which provides facilities for including architectural design concepts as well as the rationale for the design steps in its syntax. It thus integrates the architectural design process with the design documentation process and by doing so reduces the risk of occurrence of the previously mentioned problems when the processes are separated. Archium is an experimental language, developed at the Rijksuniversiteit Groningen by Anton Jansen and continuously in development to make it fit the industrial design practices as well as possible. It is currently in the prototype stage.

## 1.1 Graduation project

As the Archium language is in its early phases of development, a number of extensions have had to be made to make it more mature, and use it for more complex (industrial) test cases. This has been the subject of my graduation project.

The graduation project started on September 1$^{st}$ 2005, and has been carried out in full at the SEARCH [25] group at the Rijksuniversiteit Groningen. The project was finished in August 2006. The project has been supervised by Anton Jansen, who is also a member of the SEARCH group.

## 1.2 Structure of this thesis

This thesis shows the research, design and implementation of the work done for the graduation project. Chapter 2 provides a more detailed introduction to Archium, followed by chapter 3 which discusses the problems that have been solved during the graduation project. Chapter 4 describes a case example which is used throughout this thesis to test and verify the extensions made to Archium. Chapters 5, 6, 7 and 8 provide the details of the analysis, design and implementation of the extensions which were selected to be developed during the graduation project. Chapter 9 revisits the case example and shows its development after completion of the implementation of the selected extensions. This is followed by a discussion of related work in chapter 10. Finally, chapter 11 concludes the thesis by summarizing the results and providing suggestions for future work.

A note for the reader: Archium concepts in the text of this thesis are recognizable by a capital first character of all words belonging to that concept, for example *Design Decision* and *Design Fragment*. Consequently, sometimes a concept used in this thesis may be written using the capital first characters, while at other times it may be written without them. An example is the requirement concept, which is referred to as "requirement" when it is used to denote software requirements in general, and as "Requirement" when it is used as the Archium Requirement entity.

# Chapter 2

# Archium

## 2.1 Introduction

Most programming languages only allow for implementation concepts in their source code. Software architecture design is assumed to be a different stage in the software development process resulting in separate artifacts. Both implementation and design are maintained separately. This can lead to mismatches between implementation and design, as a lot of effort has to be spent on synchronizing both kinds of artifacts, especially when evolving or modifying the system.

Archium is a programming language which takes a different approach: it incorporates the implementation as well as the design in its syntax. Archium allows and enforces specification of both of these concepts in source code. While this is not unique (there exist a few other languages that also take this approach, e.g. ArchJava [20]), Archium *is* unique in introducing the notion of *Design Decisions*, which go even further than allowing for both implementation and design concepts combined in source code.

Design Decisions include architectural changes but at the same the *rationale* describing the reasons, trade-offs, pros and cons of the change. Such knowledge is easily lost in traditional software development processes, as the focus is more on the results than on describing the ideas behind the results. Dedicated constructs are available in Archium that allow a designer to treat design decisions as first-class concepts.

Maintaining implementation, design *and rationale* at the same time allows for improved understanding, maintenance and evolution of a software system design. Also, design erosion is less likely to happen as a history of all taken decisions and knowledge is available at the time when (large) modifications/additions are about to take place.

Archium is built on top of Java and includes a compiler as well as a run-time platform. The additional keywords introduced allow for a clear and structured way of describing design and rationale, while at the same time making use of the powerful Java language for implementation of the design.

The remainder of this section consists of the following topics. Section 2.2 shows the three submodels which together form the Archium meta-model, and discusses the Archium programming language. Section 2.3 shows the available tools for Archium. Note that this is only a brief introduction to Archium, a more thorough discussion can be found in [2].

## 2.2 Meta-model

Archium was originally developed as a theoretical meta-model and consists of the following three sub-models: an architectural model, a design decision model and a composition model. In the next three sections, each model is explained separately. In section 4, the chat case is introduced, which contains a concrete example of how the three models operate together in practice. This may help the reader in understanding the, at first sight, rather complex abstract concepts.

### 2.2.1 Architectural model

The architectural sub-model of Archium allows modelling of software architectures using concepts of the Component & Connector View [23]. The main concepts used are the Component Entity and the Connector. Component Entities represent logical architectural building blocks (essentially a larger-grain version of the Class concept in object-oriented programming), and Connectors provide a means for communication between Component Entities. Connectors can only be connected to explicitly defined Ports on Component Entities, and each Port is marked *provided* or *required*, depending on the relationship with the connecting Component Entity.

### 2.2.2 Design Decision model

As said in section 2.1, Design Decisions are first-class concepts in Archium. They are modeled using the Design Decision and Design Fragment concepts. A Design Decision contains the rationale for making a certain architectural decision, the considered alternatives and, for each alternative, its pros and cons. Design Fragments form the link between the architectural model and Design Decisions. They contain the architectural elements involved in the Design Decision. Using Design Fragments, new architectural elements can be added to or deleted from the running system, or existing elements can be modified.

### 2.2.3 Composition model

Using the composition model, changes made by the Design Decision model are related to elements of the architectural model. Concretely, the composition model defines how changes should be mapped to entities defined in the architectural model. Therefore the composition model defines a number of composition techniques. It is worth noting that an explicit change entity exists in Archium, called a Delta. Every change (like introducing Component Entities, adding functionality to a Component Entity, connecting two Component Entities using a connector, etc.) 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.

### 2.2.4 Language

In addition to a theoretical model, a programming language has been developed which allows for specification of the modelling concepts discussed in sections 2.2.1, 2.2.2 and 2.2.3. Its syntax resembles that of Java and mixing Java with Archium expressions is possible in certain constructs. Rationale is an integral part of the language definition, and thus is a first-class member of the language.

Each Archium entity (like a Design Decision and a Delta) has its own construct consisting of a number of *blocks*, which allow for user-specified code. For example, figure 2.1 shows the declaration of a Delta named `exampleD`. The blocks `provides`, `requires`, `variables`, `run`, `initialization` and `implementation` are containers for user-specified code. For example, the `initialization` block allows may contain Java code to be executed when `exampleD` is

```
delta exampleD {
    provides {
        ...
    }

    requires {
        ...
    }

    variables {
        ...
    }

    run {
        ...
    }

    initialization {
        ...
    }

    implementation {
        ...
    }
}
```

Figure 2.1: A Delta and its blocks

initializing.

The general process of building an application in the Archium language is as follows. An application consists of a number of Design Decisions subsequently applied. Each Design Decision adds to, deletes from or modifies parts of the application defined by previous Design Decisions. An application is thus incrementally built up from scratch. The architect defines each Design Decision in the Archium language, including its rationale. Furthermore, the developer specifies the sequence in which the Design Decisions have to be applied. After compilation, when the developer executes the application, the specified sequence of Design Decisions will be applied, incrementally building up the final application.

Figure 2.2 illustrates the modeling of a system in Archium. The scope of a Design Decision consists of the running system (denoted by *RS*), a Design Fragment consisting of the entities that will be involved in the application of a Design Decision (denoted by *DF*) and a Design Fragment Composition (denoted by *DFC*) specifying *how* these entities in the Design Fragment will impact the running system. The result of the application of a Design Decision is again a running system.

## 2.3 Tooling

A number of tools accompany the Archium programming language. These are discussed in this section. Section 2.3.1 discusses the compiler, section 2.3.2 the run-time platform and section 2.3.3 describes the Archimon Visualizer.

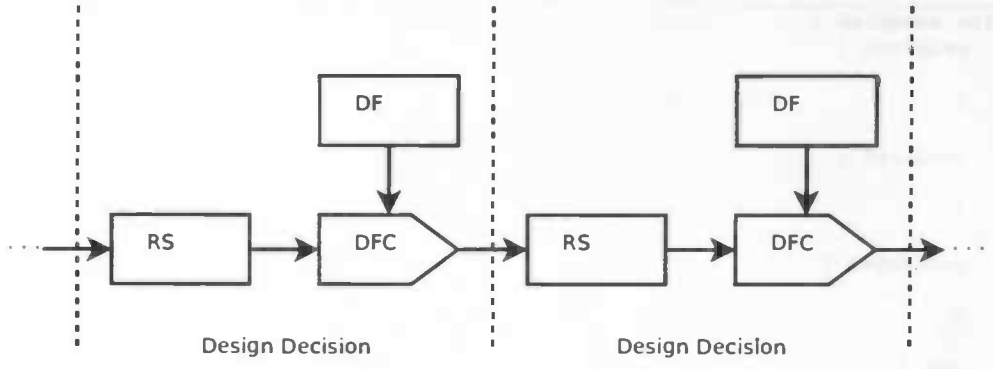


Figure 2.2: Modeling a system in Archium

### 2.3.1 Compiler

The Archium language is accompanied by a compiler. This compiler ultimately generates standard Java `.class` files but consists of intermediate compile steps. Archium source files (having extension `.archium`) are first compiled (using the Archium Compiler) into ArchJava files (having extension `.archj`).

ArchJava is an open-source library allowing for the specification of software architectures in Java. It thus consists of much functionality also provided by Archium (for example, adding Connectors between Component Entities). To prevent duplicating existing functionality, the decision was made to include ArchJava as part of Archium. ArchJava comes with its own compiler accepting `.archj` files which resemble the standard Java syntax but with some additions for the specification of architectural concepts. The output of the ArchJava compiler are Java `.java` files which contain calls to functionality provided by the ArchJava library.

The standard Java compiler compiles the ArchJava-generated `.java` files into `.class` files, ready for execution by the Java Virtual Machine. In addition of accepting Archium source files, the Archium compiler also accepts Java files which can be used within an Archium application. In that case, the intermediate ArchJava compile step is omitted and the Java files are immediately passed to the Java compiler. The executable resulting from the Java compiler is runnable on each platform that supports Java. For more information on the compiler internals, the reader is referred to [5].

### 2.3.2 Run-time platform

An important part of Archium is the run-time platform which serves two main functions. Firstly, it provides core functionality needed by user-compiled Archium applications at run-time. The application of Design Decisions is an example of such core functionality.

Secondly, the platform allows for the inspection of user-compiled applications at run-time. This means that for each user-defined Archium entity (like Component Entities and Connectors), a base class is provided by the run-time platform which contains methods for retrieving information about the entity. Not only is this information used by the run-time platform internally for providing its core functionality, it is also used for debugging purposes and the run-time inspection of entities, for example in the Archimon Visualizer (section 2.3.3). Examples of information to be retrieved by means of the run-time platform are:

- The Design Decision that introduced a certain Component Entity to the running system;

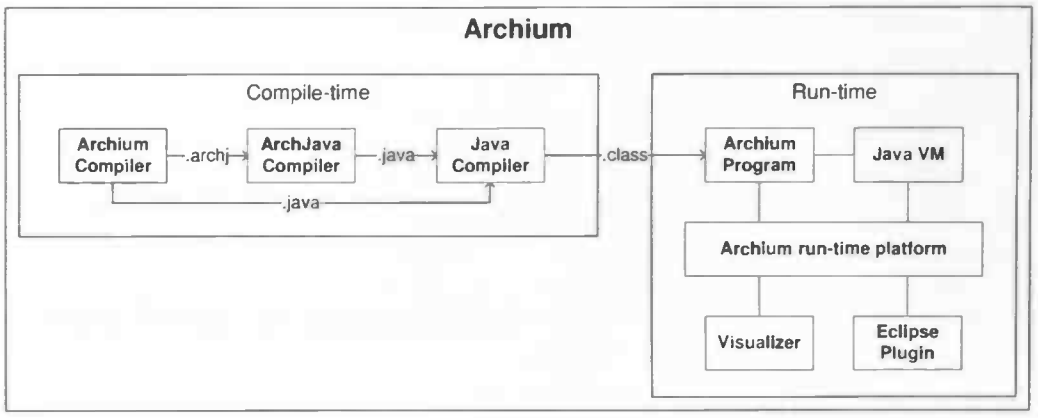


Figure 2.3: Overview of Archium

- The rationale for applying a certain Design Decision.

The run-time platform can thus be seen as the interface to any (external) application requiring the inspection of a user-compiled Archium application at run-time. The run-time platform is linked to each user-defined Archium application at compile-time.

### 2.3.3 Archimon Visualizer

A tool called the Archimon Visualizer helps the developer in validating his/her implementation. The Archimon Visualizer, which is part of the Archium source package, allows for a graphical display of Archium entities currently present in a running system built in Archium. It works closely with the run-time platform to obtain information about Archium entities.

Communication between the Archimon Visualizer and Archium takes place using a plug-in mechanism. The visualizer registers itself as a plug-in with the Archium run-time platform, and the run-time platform notifies the visualizer whenever entities are created, modified or destroyed. Using the same mechanism, an Eclipse-integrated [11] visualizer also exists which provides the same graphical information as the standalone visualizer, but is integrated with the development environment. Figure 2.3 shows an overview of Archium including all elements and relationships just discussed.

# Chapter 3

# Problem statement

## 3.1 Problems

The main problem for which a solution has been provided during the graduation project is, as this thesis' title implies: "How to make the Archium compiler more mature?". As the word "mature" is rather generic, four sub-problems have been chosen which currently prevent Archium in becoming mature. These problems are described in the following sections.

### 3.1.1 Requirements model

Software requirements are the cause of development of certain functionality in a system. Equivalently, software requirements are the cause of a Design Decision in Archium. Software requirements thus have a relationship with Design Decisions. The problem is that Archium currently does not support the expression of requirements in its language and thus does not have support for defining relationships between requirements and Design Decisions. To verify which requirements have been fulfilled and which ones are yet to be implemented, and to relate requirements to requirement categories and stakeholders, Archium should have a way of defining these concepts within its language and relate them to Design Decisions. The research question which will therefore be answered in order to provide a solution for this problem is: "How can a requirements model be added to Archium?". The answer and solution are discussed in chapter 5.

### 3.1.2 Referencing and resolving entities from rationale

Archium allows and encourages for natural text (rationale) at specific points in the source code of Archium-developed applications. This text is available at run-time for inspecting, for example to allow (automatic) documentation generation, and to verify the implementation against the requirements. In these natural texts, references can be made to architectural entities by specifying the name of the entity between square brackets. The problem is, however, that no algorithm for resolving such references to object references exists yet. Such an algorithm should be developed to allow the user to view at run-time the state of the running system and browse the available objects. The research question which will therefore be answered in relation to this problem is: "How can entity references be resolved from rationale in Archium?", and is answered in chapter 6.

### 3.1.3 Threading model

Archium currently does not support the threaded implementation of functionality very well. So-called Deltas (explained in more detail in section 4) in Archium can contain code that must be run as a thread. However the current threading implementation is basic and does not always

function as expected. Issues vary from threads that do not start executing to terminated threads that are being restarted, resulting in exceptions.

There are a two fundamental problems causing these issues. Firstly, the lack of managed threading functionality within Archium. Although application programmers can use Java code together with Archium code (and in this way use threading by means of the `java.lang.Thread` package), there is a need for centralized management of threads, as many parts of a running application may be affected by the application of a Design Decision, possibly requiring the suspension and resuming of threads. Which parts and how they are affected may not be known to the programmer so he cannot be held responsible for thread management. The responsibility for thread management has to be moved from the application programmer to the Archium run-time platform.

Secondly, threads currently start running without checking whether they are allowed to run or not, i.e. threads should only run if certain preconditions are met. This is due to the fact that the life-cycles of Archium entities are not clearly defined. The addition of life-cycle models to Archium entities needs to be addressed as well, as thread operations depend on the states which the entities are in.

In addition to the two fundamental problems underlying the issues described above, a well-implemented threading model is needed as otherwise the expressive power of the Archium language would be severely limited. Single-threaded systems are not useful for most (larger) applications, as certain functionality can only be achieved by means of a threaded implementation.

A new, managed threading model therefore has to be incorporated into Archium, resulting in the following research question: "How can a managed threading model be added to Archium?". The answer to this question, and an implementation, is provided in chapter 7.

### 3.1.4 No facility for the runtime application of Design Decisions

One of the goals of Archium is to allow dynamic recomposition of a software system (apply Design Decisions at run-time to a running system). Currently this can only be performed in a user-specified sequence at compile-time. There exists no facility for applying a user-specified Design Decision at run-time. Run-time Design Decision application functionality would demonstrate the flexibility of Archium and its benefits in comparison to traditional software development. This problem leads to the following research question: "How can run-time application of Design Decisions be achieved in Archium?". Chapter 8 shows the extensions made to the Archimon Visualizer, and the research question is answered in section 8.3.

# Chapter 4

# The chat case

The chat case is used throughout this thesis to validate the extensions made to Archium. This chapter describes the case in general and its goals in detail. Section 4.1 describes the general ideas behind the case and section 4.2 shows its architectural design.

## 4.1 Description

The chat case aims at developing a simple client-server messaging system in Archium, as a validation of the work done on the concepts, compiler, language and run-time system so far. The focus is not so much on providing much functionality, but on the fact that the system should be built using Archium concepts like Connectors, Component Entities and Design Fragments. Furthermore, it should feature some kind of run-time reconfigurability using Design Decisions, for example switching between a single- and multi-threaded server implementation for handling client connections. Concretely, the following goals must be met with the chat case:

1. Implementation must be done using Archium concepts;
2. Dynamic reconfigurability must be demonstrated;
3. Requirements must be specified within Archium source code;
4. Archium entities must be referable from within rationale;
5. Threaded code must be used in the implementation.

The chat case is divided into two sub-cases: a chat server and a chat client, which are also treated as separate projects in Archium, i.e. the resulting source code will be in separate projects. The chat server allows chat clients to connect to it, and send text messages to users also connected to the same server. The client provides a graphical user interface for sending text messages and allows for logging in and out on a chat server. Multiple instances of the client are be allowed to run concurrently and connect to the same server. The architectural design of both sub-cases is discussed in the following two sections.

## 4.2 Architectural design

This section provides the architectural design of the chat case. As well as serving as the blueprint for the final implementation, it also forms an introduction to Archium concepts which will help the reader to understand the remainder of this thesis better. Figure 4.1 functions as a legend for all concepts used in the graphical representation of the architectural design. In addition, each concept will be explained in the text as well.

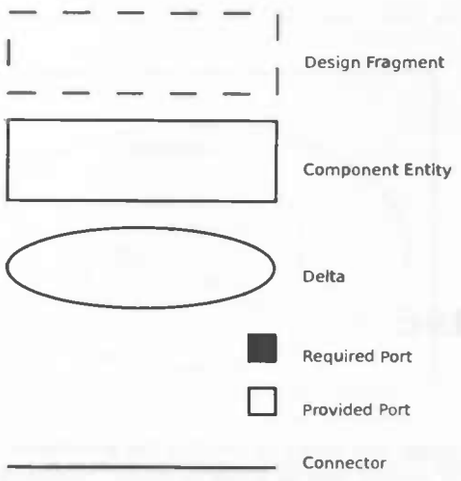


Figure 4.1: Legend of symbols used in the architectural design

### 4.2.1 Server

The architectural design of the server sub-case comprises four Design Decisions, and is described in the following sections.

**Design Decision 1: Introduce server component**

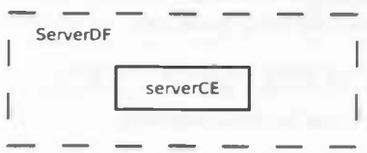


Figure 4.2: Design Fragment `ServerDF` containing Component Entity `ServerCE`

The first step in designing the server application involves introducing a basic server component on which detailed design will be carried out in later Design Decisions. Figure 4.2 shows the Design Fragment for Design Decision 1. A Design Fragment is displayed with a dashed border and is suffixed by the characters `DF`. It introduces entities to the running system which are involved in applying a Design Decision. In this case, a Component Entity is to be added to the running system by Design Decision 1. A Component Entity has a box-shaped figure and its name is suffixed by `CE`. A Component Entity is essentially a broader-grain version of the object-oriented programming concept *class*, and includes architectural aspects as well as implementation aspects. For example, it can be fleshed-out into subcomponents, and every subcomponent can be given an implementation. Additionally, every Design Decision can add to, change or delete a Component Entity's implementation supplied by previous Design Decisions.

In addition to a Design Fragment indicating *which* entities are involved in the application of a Design Decision, each Design Decision also has an associated Design Fragment Composition entity specifying *how* these entities influence the running system. As there is no running system yet (this is the first Design Decision to be executed), a Design Fragment Composition is not necessary. Instead, Design Fragment `ServerDF` is returned by Design Decision 1 to fulfill the role as running system.

**Design Decision 2: Flesh out the server component**

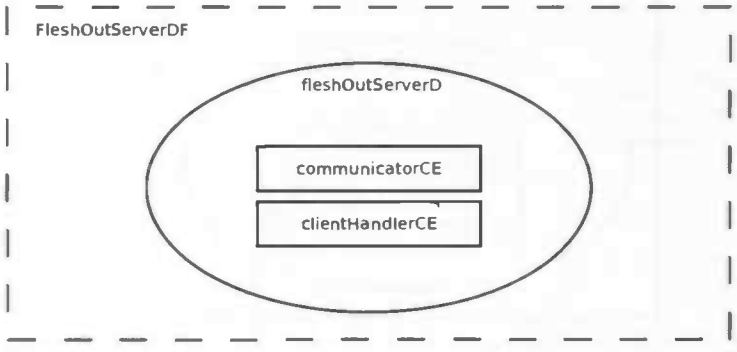


Figure 4.3: Introducing the `communicatorCE` and `clientHandlerCE` Component Entities

The chat server contains two logical functional parts: 1) a communication part for establishing connections with clients and maintaining them, and 2) a handling part which will process any request made to the server, like logging in or sending a text message to another user.

While it is possible to define all functionality required for the server in the `serverCE` Component Entity, for maintainability and understandability it is a better idea to divide `serverCE` into sub-components, each representing a logical building block. Therefore the Design Fragment associated with Design Decision 2 contains two new Component Entities which will take the roles of communicator and client handler, respectively. The Design Fragment is shown in figure 4.3.

Both Component Entities are contained within an ellipse-shaped figure. This entity is called a Delta and its name is suffixed with `D`. Changes to a Component Entity (in this case: `serverCE`) can only be made by composing a Delta with it. Each Delta composed with a Component Entity forms a new layer around it, and all composed Deltas together form the implementation of the Component Entity. This implies that a Component Entity initially has no implementation, which is exactly the case. As Deltas are stacked on top of each other, rules have to be defined how they communicate, but this issue will be addressed later. For now the situation is a Component Entity (`serverCE`) with which no previous Deltas have been composed yet.

As there is a running system (represented by `ServerDF`) now, the Design Fragment Composition has to be used to specify how the new elements introduced by `FleshOutServerDF` have to be integrated with the running system. In this case, the integration consists of composing `fleshOutServerD` with `serverCE`. The resulting running system is shown in figure 4.4.

**Design Decision 3: Implement the server**

After dividing the server component into logical sub-components, it is now time to give the sub-components a Java implementation. As implementation of Component Entities is provided by Deltas, two are needed: one for Component Entity `communicatorCE` (`communicatorD`) and one for Component Entity `clientHandlerCE` (`singleThreadedClientHandlerD`). In addition, they need to communicate with each other, as any client connections are accepted by the communicator and then passed to the client handler for processing any request made by the connected clients. Deltas have endpoints called Ports which allow them to communicate with other Deltas (using a Connector). Ports are bound to an Interface specification and must be marked *provided* or *required*, depending on the relationship with the connecting Delta. Figure 4.5 shows that `communicatorD` *requires* the functionality provided by `singleThreadedClientHandlerD`, so the Connector is connected to a *required* Port of `communicatorD` and to a *provided* port of

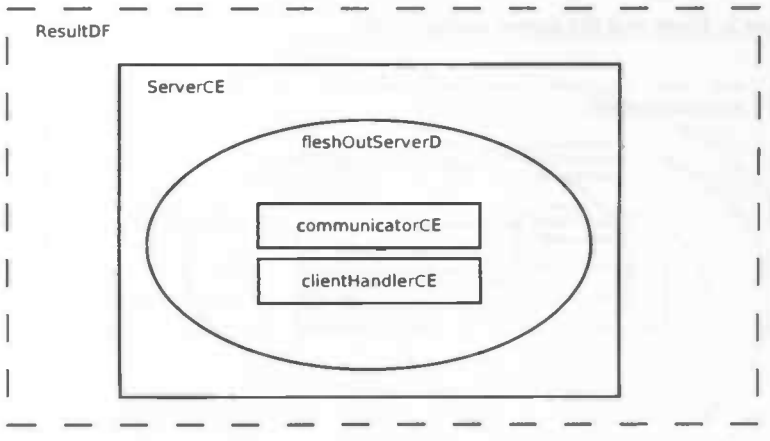


Figure 4.4: The running system after application of server Design Decision 2

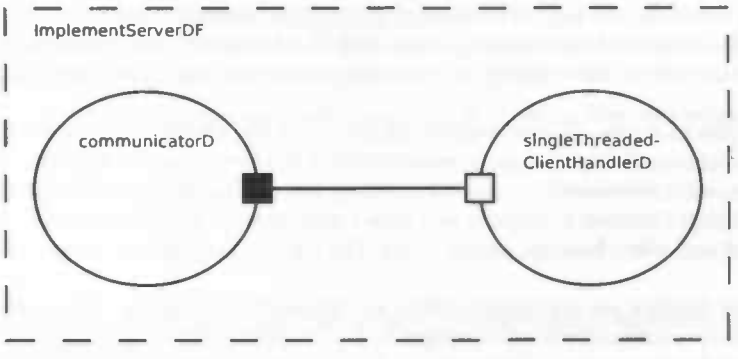


Figure 4.5: Design Fragment `ImplementServerDF`

`singleThreadedClientHandlerD`. They are both bound to the same Interface specification, so they each know exactly which methods to call (`communicatorD`) or provide/implement (`singleThreadedClientHandlerD`).

The Design Fragment Composition associated with Design Decision 3 specifies that the Delta named `communicatorD` should be composed with `communicatorCE` and the Delta named `singleThreadedClientHandlerD` with `clientHandlerCE`. The resulting running system is shown in figure 4.6.

**Design Decision 4: Changing the server implementation**

The final Design Decision for the chat server changes the implementation of the client handler from a single-threaded one to a multi-threaded one. This is accomplished by composing another Delta (`multiThreadedClientHandlerD`) with `clientHandlerCE`. The Design Fragment associated with this Design Decision is shown in figure 4.7. Just like the original, single-threaded Delta contained a provided Port for connecting with Delta `communicatorD`, Delta `multiThreadedClientHandlerD` also contains one (bound to the same interface specification as `singleThreadedClientHandlerD`), as the change of implementation of Component Entity `clientHandlerCE` should be transparent to `communicatorD`. Archium therefore supports a mechanism called *superimposition* which allows the user to specify how a Delta composed with a Component Entity changes the interface of that Component Entity (the set of

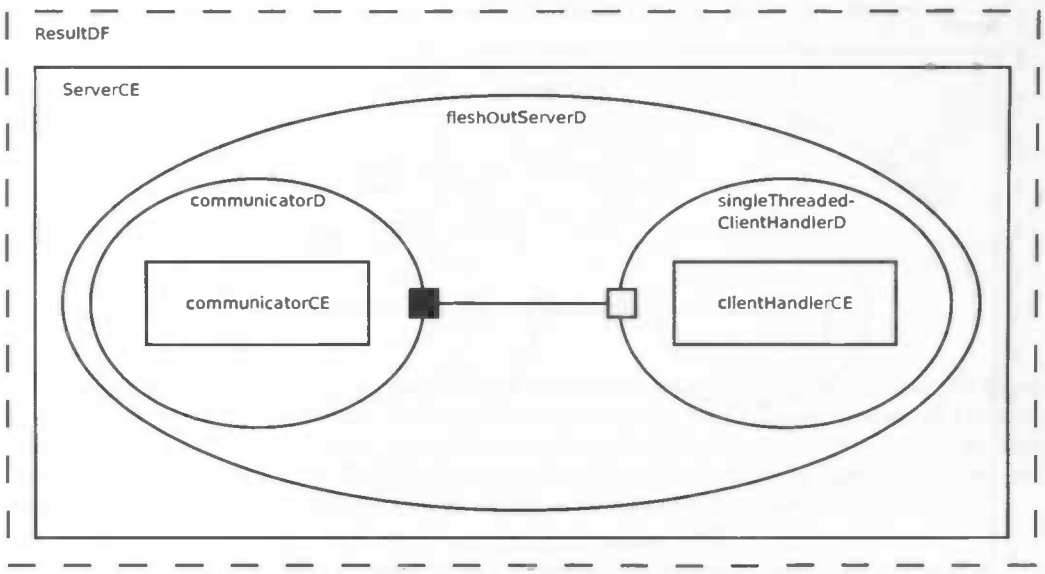


Figure 4.6: The running system after application of server Design Decision 3

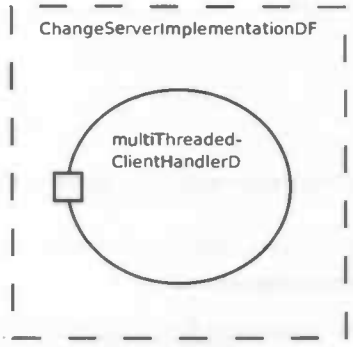


Figure 4.7: Design Fragment `ChangeServerImplementationDF`

provided and required Ports and their implementation). In this case, the change is to catch all method calls previously made to `singleTheadedClientHandlerD` and re-route them to calls to `multiThreadedClientHandlerD`. Remember that both Deltas only have one provided Port which is bound to the same interface, so for `communicatorD`, this re-routing is transparent. The superimposition technique to be used is specified in the Design Fragment Composition associated with this Design Decision, and in this case is called *Adapter*. Figure 4.8 shows the resulting running system after applying the Design Decision. Using superimposition, `multiThreadedClientHandlerD` is composed with `clientHandlerCE`, and the Connector, which previously was connected to `singleThreadedClientHandlerD` is now connected to `multiThreadedClientHandlerD`. Note that, although not shown in figure 4.8, the Design Fragments specifying which elements are involved in a certain Design Decision are not lost after the Design Decision has been applied, but keep existing in the running system for traceability purposes: each element can be traced back to the Design Fragment(s) in which it exists. So, for example, Design Fragment `ServerDF` (as shown in section 4.2.1) still exists in Design Decision 4 and still contains Component Entity `ServerCE`. Design Decisions other than the Design Decision representing the running system (`ResultDF`) have been left out of the various figures to prevent

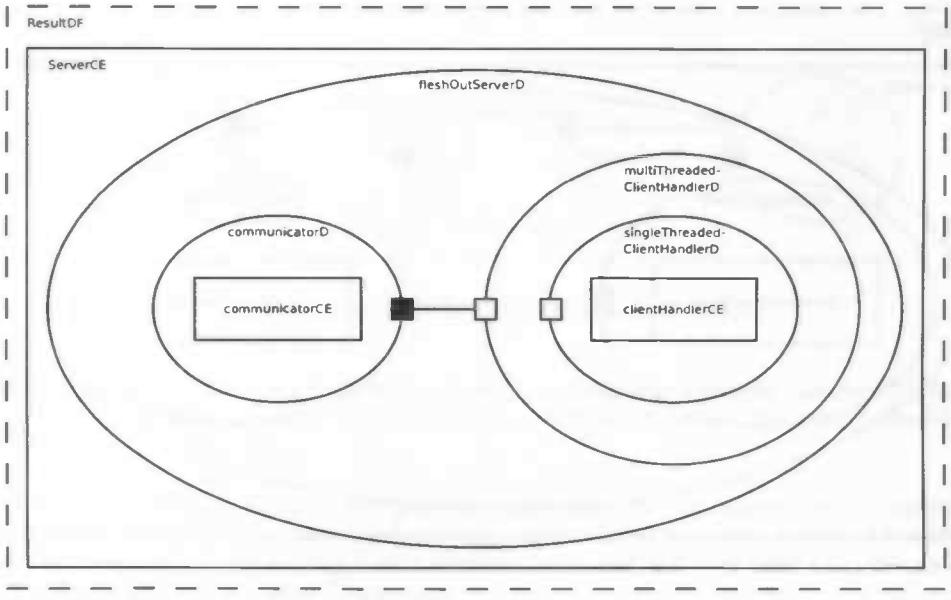


Figure 4.8: The running system after application of server Design Decision 4

cluttering.

### 4.2.2 Client

The architectural design of the client sub-case consists of three Design Decisions, and is discussed in the following sections.

#### Design Decision 1: Introduce client component

In this first Design Decision, a client Component Entity is introduced to the running system which will be used in subsequent Design Decisions for detailed design and implementation. Figure 4.9


Figure 4.9: Introducing the client component

shows the Design Fragment associated with this Design Decision. As there is no running system yet (this is the first Design Decision to be executed), a Design Fragment Composition is not necessary and `ClientDF` will represent the running system after the Design Decision execution completes.

#### Design Decision 2: Flesh out the client component

The second Design Decision consists of performing detailed design on `ClientDF`. Subcomponents are introduced for communication handling with the chat server, a graphical user interface and a controlling entity which will orchestrate the data-flow between the two other components. The Design Fragment associated with this Design Decision is shown in figure 4.10. As

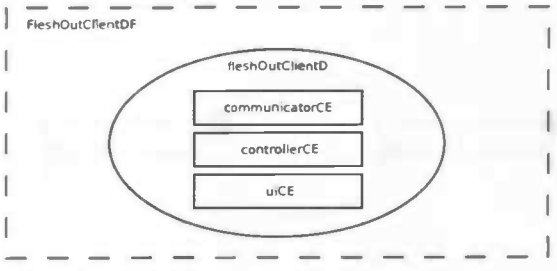


Figure 4.10: Design Fragment `FleshOutClientDF`

can be seen, the fleshing out is accomplished by means of a Delta (`fleshOutClientD`) introducing the three new Component Entities. By composing `fleshOutClientD` with `clientCE`, the three new Component Entities will become sub-components of `clientCE`. After composition (as specified in the Design Fragment Composition associated with this Design Decision), the resulting running system is represented by the Design Fragment shown in figure 4.11.

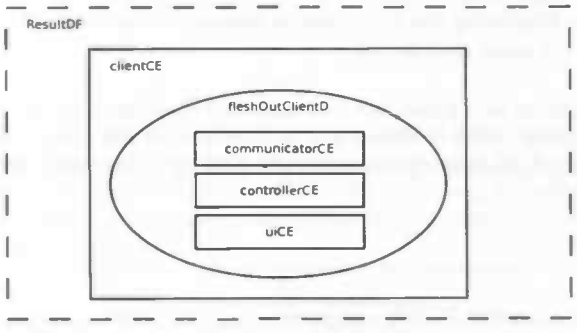


Figure 4.11: The running system after application of client Design Decision 2

**Design Decision 3: Implement the client component**

Design Decision 3 provides the implementation for the three introduced Component Entities. The associated Design Fragment (`ImplementClientDF`) therefore introduces a Delta for each Component Entity: `communicatorD` for `communicatorCE`, `controllerD` for `controllerCE` and `uiD` for `uiCE` (note that `communicatorD` is not the same Delta as the one discussed in section 4.2.1, only the names are equal). Figure 4.12 shows this situation. In addition, pairs of Connectors are provided between `communicatorD` and `controllerD` and between `controllerD` and `uiD`. The functionality each Delta provides is discussed separately in the next three paragraphs.

Delta `communicatorD` handles the connection with the chat server and sends and receives raw (encoded) messages. It *provides* functionality to `communicatorD` for sending a raw message to the chat server to which the client is connected. It *requires* `controllerD` for accepting raw messages.

Delta `controllerD` is responsible for decoding raw messages accepted from `communicatorD`, and encoding them from user actions performed in `uiD` (like logging in, out and sending a text message). Delta `controllerD` therefore *provides* functionality to `communicatorD` for accepting received raw messages, and *requires* `communicatorD` for sending raw messages. It *requires* `uiD` for performing actions after decoding messages (like showing the list of online users connected

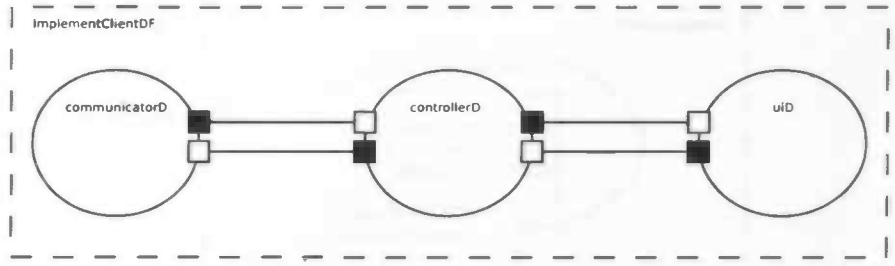


Figure 4.12: Design Fragment `ImplementClientDF`

to the chat server, or displaying the text of a received message), and *provides* functionality to `uiD` for sending a login message or sending a text message of which the text has been entered into a graphical control by the user.

Delta `uiD` displays and manages the graphical user interface. It *requires* `controllerD` for (among others) sending login/logout requests and of course text messages. It *provides* functionality to `controllerD` for displaying the list of online users and received text messages from other users also connected to the same chat server.

The need for each Connector and provided and required Port becomes immediately clear from the description of the Deltas. After composition, as specified in the associated Design Fragment Composition, the running system is represented by the Design Fragment shown in figure 4.13.

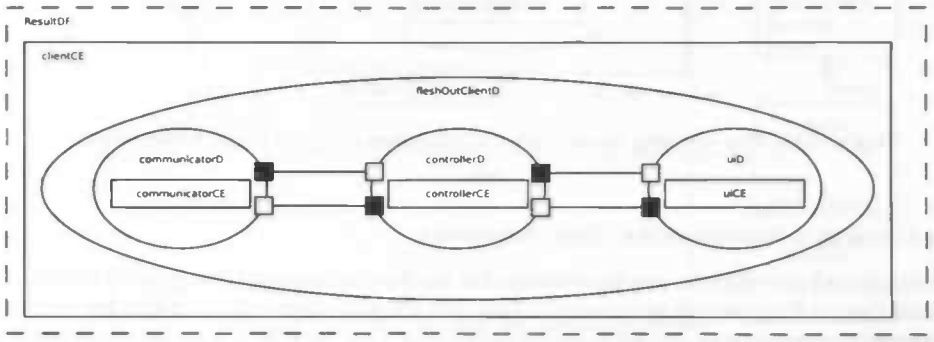


Figure 4.13: The running system after application of client Design Decision 3

The chat case is used throughout the remainder of this thesis to verify, illustrate and test the solutions found to the problems identified in chapter 3. In chapter 5, the chat case requirements will be modeled using the added requirements model. In chapter 6, actual chat case code fragments are shown illustrating the object referencing algorithm. Chapter 7 uses the chat server to demonstrate the added threading model and chapter 8 uses screenshots from the Archimon Visualizer in combination with the chat case.

# Chapter 5

# Adding a requirements model to Archium

## 5.1 Introduction

Software requirements are the cause of development of certain functionality in a system. Equivalently, software requirements are the cause of a Design Decision in Archium. They express a certain need for change and thus form a motivation for applying a Design Decision. Software requirements thus have a strong relationship with Design Decisions, as each Design Decision can be traced back to (a) certain requirement(s).

The problem is that Archium currently does not support the expression of requirements in its language and thus it has no support for defining relationships between requirements and Design Decisions. In order to allow traceability between Design Decisions and requirements, and to strengthen the relationship between architecture and rationale (in this case between architecture and requirements, which can be seen as a specialized form of rationale), a requirements model has been added to Archium.

The added requirements model allows architects to verify which requirements have been (partially) fulfilled and which ones are yet to be implemented. Furthermore, it allows for classification of requirements into user-defined requirement categories and relate stakeholders to requirements.

An existing requirements model [15] has been taken as a basis for the development of the added requirements model. The existing model had been developed as an initial requirements model but was never actually implemented in Archium. It has been part of the Archium source package as a guideline for implementing a "real" requirements model.

The existing requirements model has been slightly modified and has afterwards been implemented in the Archium parser, code generator and run-time platform. This chapter provides the details of the existing model, modifications made to it and its integration with Archium. The research question which is answered throughout the chapter is "How can a requirements model be added to Archium?" (as raised in section 3.1.1). The remainder of this chapter is subdivided into sections as follows. Section 5.2 provides the analysis of the necessary modifications to the existing requirements model and section 5.3 contains its implementation details.

## 5.2 Analysis

### 5.2.1 Existing model

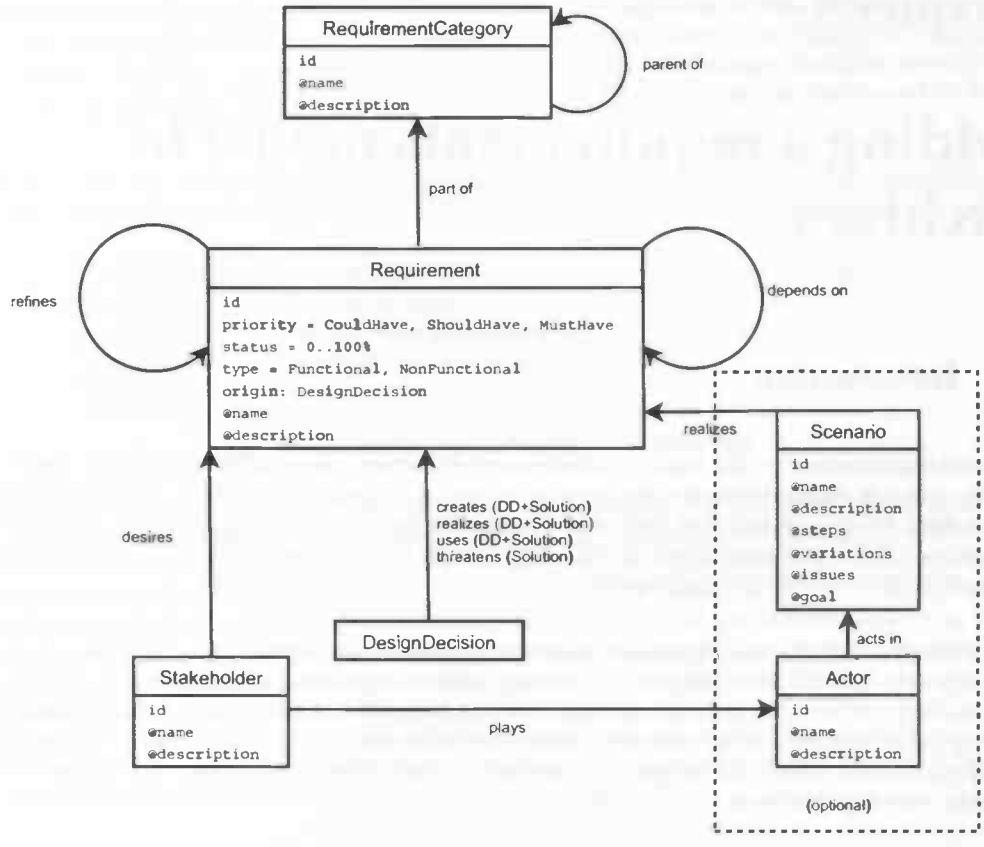


Figure 5.1: The existing requirements model

The model which had already been developed when work started on this problem is depicted in figure 5.1. The model contains six entities (Requirement Category, Requirement, Stakeholder, Design Decision, Scenario and Actor) and their attributes. Some of the attributes are modifiable by the user programming in the Archium language (static, compile-time attributes), while others are maintained internally by the Archium run-time platform (dynamic attributes).

In addition to the entities, the relations between them are also shown in figure 5.1, as arrows together with a description of the nature of the relationship. The cardinality of the relationships had not been specified in the original model, so deciding and specification of the cardinality is one of the modifications made to the model.

It was immediately decided to leave out the optional part (the Scenario and Actor entities and their relationships), as consensus had not yet been reached about the attributes which should be included in these entities. The following subsections describe the model modifications in detail.

### 5.2.2 Requirements

The Requirement entity is the main entity of the model, as most other entities in the requirements model can have a relationship with it. The static attributes of this entity are listed in table 5.1. The "Obligatory" column of this table indicates whether the programmer is forced to specify a value for this attribute in his Archium code. The set of obligatory attributes was chosen to be the minimum level of information necessary for a software requirement to be useful. However, as this is an initial requirements model, the (obligatory) attributes of the Requirement entity may change in the future.

| Attribute | Type | Obligatory |
|---|---|---|
| `id` | A unique identifier not containing spaces | Yes |
| `@name` | A name in natural text | No |
| `@description` | A description in natural text | No |
| `priority` | One of `CouldHave`, `ShouldHave`, `MustHave` | Yes |
| `status` | A percentage in the range 0-100% | Yes |
| `type` | One of `Functional`, `NonFunctional` | Yes |

Table 5.1: Static attributes of the Requirement entity

```
requirement Chat_Client_2REQ {
    @name {#
        Client supports simple protocol.
    #}

    @description {#
        The chat client must support a simple protocol for allowing
        clients to:
        - login;
        - logout;
        - send text messages to other logged-in users;
        - list all other logged-in users.
    #}

    priority MustHave;
    status 0%;
    type Functional;
}
```

Figure 5.2: A Requirement expressed in the Archium language

Figure 5.2 shows a requirement taken from the chat client which is expressed in the Archium language using the attributes described above. The `id` attribute follows immediately after the `requirement` statement and has a value of `Chat_Client_2REQ`. It is used for managing (a large number of) Requirements by a user- or company-defined numbering/identification scheme. The `name` attribute allows for a more detailed name in natural text. The `priority` attribute describes the level of importance of implementation of the functionality described in this Requirement. Similarly, the `status` attribute specifies the initial percentage of completion of the described functionality. Finally, the `type` attribute distinguishes between functional and non-functional requirements.

The dynamic attributes of the Requirement entity are shown in table 5.2. As dynamic attributes are not specified at compile-time (in Archium code), an "Obligatory" column is not included in this table.

| Attribute | Type |
|---|---|
| `status` | A percentage in the range 0-100% |
| `origin` | The design decision(s) that created this requirement |

Table 5.2: Dynamic attributes of the Requirement entity

Note that the `status` attribute appears in both the static and dynamic attribute tables. While the Archium programmer has to specify an initial value for the percentage of fulfillment of a certain Requirement, this value may change at run-time depending on which design decisions have been applied. The same goes for the origin of the Requirement. Both attributes are maintained by the Archium run-time platform.

A requirement can have relations with other Requirements, Requirement Categories and Stakeholders. This is illustrated in table 5.3. The "Cardinality" column shows the cardinality between

| Relationship | This Requirement ... | Cardinality |
|---|---|---|
| `category` | ... belongs to some Requirement Categories | $1-\star$ |
| `refines` | ... refines some Requirements | $1-\star$ |
| `depends on` | ... depends on some Requirements | $1-\star$ |
| `desired by` | ... is desired by some Stakeholders | $1-\star$ |

Table 5.3: Relationships of the Requirement entity

the related entities. A "$\star$" means "zero or more". For example: a Requirement can be desired by zero or more Stakeholders. The names of some of the relations have been slightly modified compared to the original model: where the relation between Requirements and Requirement Categories was called `part of` in the original model, it is now called `category`, as this gives a clearer description of the nature of the relation. The name `part of` was discarded as being too general. Furthermore, the relation between Requirements and Stakeholders has been renamed to `desired by` (was: `desires`) and the direction of the arrow has also been changed, in order to prevent the Stakeholder entity from becoming unconnected to any other entity. This would have required additional functionality during implementation, as some mechanism would have been needed to find the unconnected Stakeholders. The Requirement example shown in figure 5.2 can be extended to also include relationships, as is shown in figure 5.3.

### 5.2.3 Requirement categories

Requirements can be part of zero or more Requirement Categories. This allows for grouping of Requirements. As all attribute values of Requirement Categories are set at compile-time, this entity contains no dynamic attributes. The static attributes are given in table 5.4.

| Attribute | Type | Obligatory |
|---|---|---|
| `id` | A unique identifier not containing spaces | Yes |
| `@name` | A name in natural text | No |
| `@description` | A description in natural text | No |

Table 5.4: Static attributes of the Requirement Category entity

It is possible to create a hierarchy of Requirement Categories. Therefore the relationship `parent of` exists, of which the details are listed in table 5.5.
As can be seen, a Requirement Category can be a parent of multiple other Requirement Categories. An example of a Requirement Category, taken from the chat server, expressed in the Archium language is shown in figure 5.4.

```
requirement Chat_Client_2REQ {
    @name {#
        Client supports simple protocol.
    #}

    @description {#
        The chat client must support a simple protocol for allowing
        clients to:
        - login;
        - logout;
        - send text messages to other logged-in users;
        - list all other logged-in users.
    #}

    priority MustHave;
    status 0%;
    type Functional;
    category ClientRequirementsREQC;
    desired by EndUserSH;
}
```

Figure 5.3: A Requirement including relationships, expressed in the Archium language

| Relationship | This Requirement Category ... | Cardinality |
|---|---|---|
| parent of | ... is parent of some other Requirement Categories | $1 - \star$ |

Table 5.5: Relationships of the Requirement Category entity

```
requirement category ServerRequirementsREQC {
    @name {#
        Server requirements.
    #}

    @description {#
        All requirements having to do with
        (non)functional parts of the chat
        server.
    #}
}
```

Figure 5.4: A Requirement Category, expressed in the Archium language

### 5.2.4 Stakeholders

The Stakeholder entity is an even less complex entity compared to the Requirement Category. It contains only static attributes and has no relationships with other entities. The static attributes are shown in table 5.6. An example of a Stakeholder is shown in figure 5.5.

### 5.2.5 Design Decisions

In order to be able to refer to Requirements from Design Decisions, a few relations have to be added to the Design Decision and Design Solution entities. The additions to the Design Decision entity are explained in this section, while the modifications to the Design Solution entity are discussed in section 5.2.6.

A design decision can create (raise the need for), realize (implement the specified functionality

| Attribute | Type | Obligatory |
|---|---|---|
| `id` | A unique identifier not containing spaces | Yes |
| `@name` | A name in natural text | No |
| `@description` | A description in natural text | No |

Table 5.6: Static attributes of the Stakeholder entity

```
stakeholder EndUserSH {
    @name {#
        The end user of the system.
    #}

    @description {#
        (Possibly) non-technical people who just want to chat
        without worrying about implementation details.
    #}
}
```

Figure 5.5: A Stakeholder, expressed in the Archium language

of), use (make use of) or obsolete (removes the need for) Requirements. Therefore these relations have to be added to the Design Decision entity. The details of these relationships are given in table 5.7.

| Relationship | This Design Decision ... | Cardinality |
|---|---|---|
| `creates` | ... creates some Requirements | $1 - \star$ |
| `realizes` | ... realizes some Requirements | $1 - \star$ |
| `uses` | ... uses some Requirements | $1 - \star$ |
| `obsoletes` | ... obsoletes some Requirements | $1 - \star$ |

Table 5.7: Additional relationships of the Design Decision entity

Note that the `realizes` relationship takes an optional argument: a percentage specifying the percentage of completion of the functionality described in the referred Requirement. An example of a Design Decision, taken from the chat server, using the added relationships is shown in figure 5.6.

### 5.2.6 Design Solutions

The additional relationships for the Design Solution entity are equivalent to those of the Design Decision entity, except for one additional relationship: the `threatens` relationship. Which solution to choose is always a trade-off: it will realize some Requirements, but may well threaten other ones. The `threatens` relationship does not exist within the Design Decision entity, as making a decision which deliberately threatens some Requirements would not be wise and should not happen in practice. The details of the additional relationships for Design Solutions are given in table 5.8.

Like the Design Decision entity, the `realizes` relationship takes an optional argument: a percentage specifying the percentage of completion of the functionality described in the referred Requirement. An example of a Design Solution, taken from the chat server, using the added relationships is shown in figure 5.7.

```
design decision DD1CreateServerDD() {
    ...

    realizes Chat_Server_1REQ by 25%;

    ...

    potential solutions {
        solution ScratchServer {
            ...
        }

        solution ThirdPartyServer {
            ...
        }
    }

    decision {
        decision ScratchServer;
    }
}
```

Figure 5.6: A Design Decision realizing a Requirement by 25%

| Relationship | This Design Solution... | Cardinality |
|---|---|---|
| creates | ... creates some Requirements | $1 - \star$ |
| realizes | ... realizes some Requirements | $1 - \star$ |
| uses | ... uses some Requirements | $1 - \star$ |
| threatens | ... threatens some Requirements | $1 - \star$ |
| obsoletes | ... obsoletes some Requirements | $1 - \star$ |

Table 5.8: Additional relationships of the Design Solution entity

```
potential solutions {
    solution ImplementServer {
        architectural entities {
            ...
        }

        realizes Chat_Server_2REQ by 100%,
                 Chat_Server_3REQ by 100%,
                 Chat_Server_3_1REQ by 100%;


        realization {
            ...
        }
}
```

Figure 5.7: A Design Solution fully realizing three Requirements

### 5.2.7 Resulting model

The resulting model (after applying the modifications mentioned in the previous sections) is shown in figure 5.8. The optional part of the original model has been left out, the modified

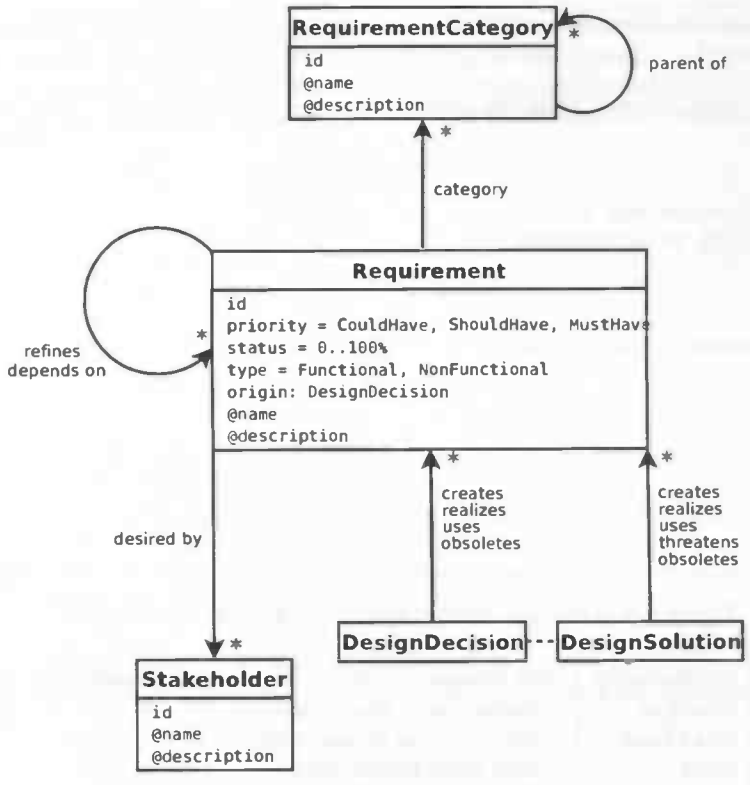


Figure 5.8: The resulting requirements model

relationship names have been added, and two or more relationships between entities have been grouped. Finally the relationships between the Design Solution and the Requirement entity have been made explicit. The relationship between the Design Decision and the Design Solution is not important for the requirements model, so it has been rendered dotted. This relationship has only been depicted to show that each Design Decision consists of a number of Design Solutions, but this is part of the Design Decision model (see section 2.2.2) and not of the requirements model. The implementation of the resulting requirements model is discussed in section 5.3.

## 5.3 Implementation

### 5.3.1 Parser

In order to recognize requirements model entity instances in Archium source code, the parser had to be adapted to recognize certain tokens and productions. As the Requirement entity is the most complex one of the requirements model (not including the Design Decision and Design Solution entities), I will use that entity throughout this section as a running example of the implementation process.

The parser generator JavaCC [17] is used to generate the Archium parser. JavaCC accepts an input-file which is largely made up of a notation similar to BNF[1]. Therefore the code in this section is expressed in BNF. It should be relatively easy to recognize the BNF statements in this section in the actual parser source code. The first step in modifying the Archium parser is there-

fore adding some tokens to the JavaCC input-file, as shown in appendix A.1. As can be seen, these tokens resemble the attribute and relationship names as found in figure 5.8. In addition to these tokens, a production rule had to be added to the JavaCC input-file, which is shown in more detail in appendix A.2. The <IDENTIFIER>, <LBRACE>, <COMMENTBLOCK>, <SEMICOLON> and <RBRACE> (non-)terminals already existed in the Archium parser input-file and will not be discussed here. The <PERCENTAGE> non-terminal reads an integer number and a percentage character.

JavaCC's mechanism of building abstract syntax trees is used to store the parsed information in nodes. For example, the requirement identifier (<IDENTIFIER>) is stored, as well as the priority type, status and a list of requirement identifiers which this requirement refines.

### 5.3.2 Reflection

Archium's run-time platform largely consists of a reflection API, allowing for run-time inspection of entity instances created by a programmer in his Archium source. As most existing entities in Archium (like Deltas, Component Entities and Design Fragments) have their own dedicated reflection classes, the reflection API had to be extended to incorporate Requirements, Requirement Categories and Stakeholders as well. The class diagram for Requirement entities is discussed in appendix A.3. The class diagrams for Requirement Categories and Stakeholders are defined similarly, and are not discussed in this thesis.

All reflection classes in Archium must implement a remote interface for inspection using Remote Method Invocation (RMI). The inspection is used by external tools requiring access to run-time information about entities existing in a running Archium applications. Therefore the Requirement reflection class implements the `RIRequirement` interface. Although this interface contains no method signatures yet, as there are no external tools yet to use run-time information about the Requirement class, this could be needed in the future.

### 5.3.3 Code generator

Archium's code generator is the link between the parser and the reflection API. Using the parsed information, the code generator generates reflection objects in Java code. The attribute values of those objects are determined (once again) by the parsed information. Finally the generated Java code is compiled by the Java compiler, after which (if no errors have been found) an executable is created.

Each abstract syntax tree node type has its own code generation method within the Archium code generator, so in order to generate code for Requirement abstract syntax tree nodes, a corresponding method has been added. The code generation method for Requirement entities generates Java source which defines a class that extends the aforementioned Requirement reflection class. Furthermore, a `getInstance()` method is generated, as the singleton pattern for Requirement entities is used. The reason for choosing this pattern to use is that at most one instance of a certain Requirement reflection class should exist at any one time in a running Archium application. All references to a certain Requirement reflection class should point to the same instance. Furthermore, the first time of use of a Requirement cannot be determined in advance, so the singleton pattern allows for lazy instantiation: only instantiate the single instance of a Requirement reflection class when it is needed for the first time. Requirements which are never being referred to therefore never get initialized, saving resources and thus being more efficient. Note that the single instance of each Requirement entity is stored in the private variable named `instance`.

[1] Backus-Naur Form, a formal way to describe formal languages.

Finally the parsed information must be transferred from the abstract syntax tree nodes to generated lines of code. For example, if the Archium programmer specified a Requirement named `R1.2`, the code generator would generate the line `instance.setName("R1.2");`. This part of the code generation is straightforward as we have all parsed information available in the abstract syntax tree node.

# Chapter 6

# Object referencing from rationale

## 6.1 Introduction

Archium allows and encourages for natural text (rationale) at specific points in the source code of Archium-developed applications. This rationale is available at run-time for inspecting, for example to allow (automatic) documentation generation, and to verify the implementation against the requirements. An example of a rationale element is shown in figure 6.1. In this case, the rationale

```
@problem{#
    An initial architecture for the chat server
    is needed [Chat_Server_1REQ].
#}
```

Figure 6.1: An example rationale element

element is called `problem` and contains natural text together with a reference to Requirement `Chat_Server_1REQ`. A reference within rationale to a certain Archium entity is made by specifying the name of the entity between square brackets. In addition to Requirements, other entities like Component Entities, Deltas and Connectors can be referred to as well. References thus add traceability between rationale and architecture, i.e. allow the user to inspect why a certain entity was added to the system by viewing its rationale. As Archium focuses very much on integrating architecture, implementation and rationale, this is a very important notion.

Although the idea is clear, the implementation of a entity-resolving algorithm was lacking. References were simply ignored. This algorithm has now been developed and its analysis and development are discussed in this chapter. The research question which therefore is answered in this chapter is: "How can entities be resolved from rationale in Archium?" (as raised in section 3.1.2). The remaining subsections of this chapter contain the analysis (section 6.2) and implementation (section 6.3) details of making object referencing from within code comments possible.

## 6.2 Analysis

An investigation of available rationale elements in Archium resulted in table 6.1. Also included in that table are the elements which can be referenced from each rationale element.
As can be seen from the table, Requirements, Requirement Categories and Stakeholders are referable from all rationale elements and have global scope. In addition, Design Solutions also include a number of elements which have local or foreign (within a Design Fragment) scope.

| Entity | Rationale elements | Referable objects |
|---|---|---|
| Design Decision | `@problem`<br>`@motivation`<br>`@cause`<br>`@context`<br>`@rationale` | The current design[1], Requirements, Requirement Categories and Stakeholders. |
| Design Solution | `@description`<br>`@design rules`<br>`@design constraints`<br>`@consequences`<br>`@pro`<br>`@con` | The current design[1], architectural entities within this Design Solution[2], architectural entities within Design Fragments[3], Requirements, Requirement Categories and Stakeholders. |
| Requirement | `@name`<br>`@description` | Requirements, Requirement Categories and Stakeholders. |
| Requirement Category | `@name`<br>`@description` | Requirements, Requirement Categories and Stakeholders. |
| Stakeholder | `@name`<br>`@description` | Requirements, Requirement Categories and Stakeholders. |

Table 6.1: Rationale elements in Archium

What is needed is an algorithm that can resolve references at run-time. Compile-time resolving is not possible, as the set and sequence of applied design decisions may not be known in advance. The next section contains the implementation details of this algorithm.

## 6.3 Implementation

The implementation of object referencing from rationale elements required modification of the code generator and the reflection API of Archium. The modifications to each of them are discussed in this section.

The natural text of rationale elements is parsed as so-called *comment-blocks*, i.e. blocks of text within {# and #} tokens. Currently the parsed text is stored within the abstract syntax tree nodes without inspecting the contents. In other words, references in the text are ignored altogether. In order for object referencing to work, each reference must be resolved and stored in a list of object identifiers. For that purpose, the following regular expression is used:

```
[(.*?)]
```

This regular expression matches the shortest occurrence of a string starting with a left bracket and ending with a right bracket. We must always match the *shortest* occurrence as otherwise multiple references within the text would be matched as a single one. The question mark character accomplishes this. The parentheses are used for processing the actual reference, i.e. without the brackets.

Each reference is made up of one or more identifiers, separated by double colons (for example `sensorDF::sensorD` which consists of the identifiers `sensorDF` and `sensorD`, and indicates a reference to `sensorD` within `sensorDF`). A double colon between two identifiers indicates

[1] As supplied by the single parameter defined in the Design Decision header.
[2] That is, architectural entities declared in the `architectural entities` block.
[3] That is, architectural entities declared in a Design Fragment which is declared in the `architectural entities` block.

that the left-hand-side identifier contains a referable entity which can be referred to by the right-hand-side identifier. So in order to resolve a reference, it is split at its double colons and each identifier is resolved using an iterative algorithm. For each split identifier, the algorithm determines whether it is a reference to a global object (concretely: a Requirement, Requirement Category or Stakeholder), or an object which is referable in the current scope (initially the object in which the comment-block occurs). The algorithm ends when the entire identifier has been resolved (to either an object instance or null), or an error has occurred due to the fact that a non-existing element has been referenced.

As each class may provide a different method for publishing its referable objects, the resolving procedure differentiates between object types. For example, the `DesignFragment` entity provides a `getEntity()` method for accessing its referable objects. Special care has to be taken when resolving a reference within a Design Solution, as also the constructor parameter(s) of its parent design decision are referable.

Figure 6.2 shows an example of the resolving algorithm using reference `a::b::c`. As can be seen, the reference is iteratively resolved. Assume that all objects publish their referable entities by means of a `getEntity()` method. Then, object `a` is asked to return the object referred to by `b::c`, which in turn generates a call to `getEntity("c")` on object `b`.

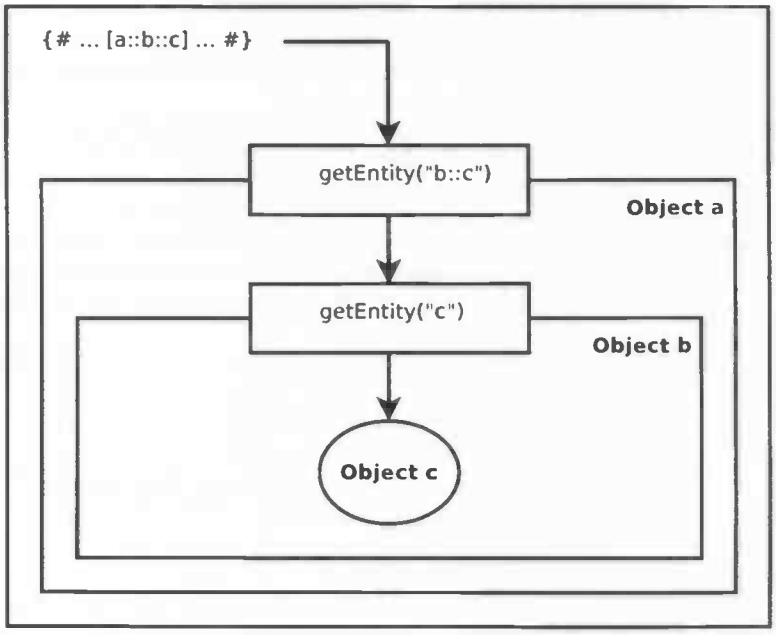


Figure 6.2: Resolving `a::b::c`

The algorithm depends largely on the Java reflection API for inspecting classes and objects. This allows for run-time resolving of references instead of compile-time resolving, which is not possible, as references are introduced through the application of Design Decisions. The sequence of introduction of design decisions may not be known at compile-time.

To store the object references found in comment-blocks, a Java vector is used. This is implemented in a dedicated class `Comment` which also stores the parsed comment-block text. The `Comment` class is part of the run-time reflection API. The `Comment` class diagram is shown in

appendix B.1. All reflection API classes which previously ignored references and just stored the comment-block text in a `String` variable, now use `Comment` class instances instead.

The list of referred objects for each entity containing comment-blocks is maintained at run-time. For Design Decisions, object references are resolved in its constructor and for design solutions during activation (the `activate()` method). Object references in Requirements, Requirement Categories and Stakeholders are resolved on demand (i.e., at first use).

# Chapter 7

# Adding a managed threading model to Archium

## 7.1 Introduction

As explained in section 2.2.3, all changes made to a Component Entity are established by means of composing a Delta with that Component Entity. A Delta may, for example, add Ports to a Component Entity, or remove an existing one. Another example, which has been shown in section 4.2.1, is the fleshing-out of a Component Entity into subcomponents.

Deltas also have the ability to add a threaded implementation to a Component Entity. Users can specify the Java code to be executed as a thread within the `run{}` block of a Delta. For each `run{}` block in each Delta, the Archium compiler generates a `java.lang.Thread`-extending class containing the user-specified Java code in its `run()` method. This thread class will be instantiated at run-time and started after composition has taken place with a Component Entity. An example thread declaration within a Delta is shown in figure 7.1. After composition with a Component Entity, this thread would repeatedly[1] print the numbers 0 to 99.

```
Delta CountD {
    ...

    run {
        for (int i = 0; i < 100; i++)
            System.out.println(i);
    }

    ...
}
```

Figure 7.1: Code inside the `run`-block of a Delta is executed as a thread

It is possible to compose multiple Deltas with a Component Entity, in this way forming a so-called Delta stack, a layered structure around a Component Entity consisting of all Deltas composed with the Component Entity. Consequently, it is possible to add multiple threaded implementations to a Component Entity, i.e. have multiple threads running concurrently in a Component Entity at run-time. An example Delta stack is shown in figure 7.2. Threads in a Delta stack of a certain Component Entity may make use of each other. For example, in figure 7.2, threads in Deltas `aD` and `cD` may cooperate in performing a task.

[1] By default, the user-code inside the `run{}` block is iterated, however this can be disabled by calling the `stopRepeat()` method inside the `run{}` block.

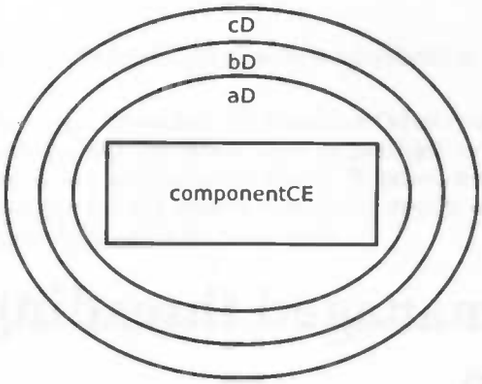


Figure 7.2: A Delta stack around Component Entity `componentCE`

Two large problems arise with the existing Delta-threading implementation in Archium which call for a new threading model:

- Threads containing user-specified Java code are started immediately after their containing Deltas have been composed with a Component Entity, even if they should not;
- Threads containing user-specified Java code are unmanaged.

Threads containing user-specified Java code should not start running immediately after their containing Deltas have been composed with a Component Entity for a number of reasons:

- Not all of the containing Delta's required Ports may have been connected (using a Connector), resulting in errors if methods are executed on them;
- Other Deltas in the Delta stack of the Component Entity with which the Delta has been composed may have unconnected required Ports resulting in similar errors (remember that threads in different Deltas in a Delta stack may make use of each other).

In order to call a method on a required Port (of which the implementation thus has been provided by another Delta using a *provided* Port), that Port must be connected to the providing Delta using a Connector. As Deltas in a Delta stack (and their threads) may call each other's methods, all Deltas in a Delta stack must have all their required Ports connected.

The fact that threads containing user-specified Java code are currently unmanaged is problematic as Archium should have a way of controlling them in case composition takes place on a Component Entity, possibly requiring suspension of (a group of) threads containing user-specified Java code in the Component Entity's Delta stack. For example, in a Delta stack of a certain Component Entity, assume that a thread $t$ in one of the Deltas is depending on the availability of a certain Connector. If another Delta would be composed with that Component Entity, removing the Connector without first suspending the threads in the Deltas in the Delta stack, errors would occur, as $t$ is suddenly not able anymore to communicate. Therefore, if a Delta is to be composed with a Component Entity, all threads in all Deltas in its Delta stack should be suspended first, after which composition takes place. Finally, each Delta should re-evaluate if it safe for its threads to resume running (i.e. verify if all Connectors are still available). To be able to suspend threads in a Delta stack, Archium should have a way of knowing which threads exist within a Delta, which is not possible using standard `java.lang.Thread`-extending classes, as they do not allow for registering themselves with any entity. This implies the need for some sort of thread-managing entity with which threads can register themselves.

The remainder of this chapter is organized as follows, and answers the following research question: "How can a managed threading model be added to Archium?" (as raised in section 3.1.3). Section 7.2 discusses the principles and definitions which form the basis for the new threading model. In section 7.3 state models are defined for each Archium entity involved in the new threading model. Section 7.4 shows the pseudo code for the algorithms needed as specified in the principles and definitions. Section 7.5 provides a step-by-step run-through of the chat server using the concepts of the new threading model. Finally, section 7.6 discusses the implementation details.

## 7.2 Principles and definitions

Analysis of the problems occurring using the existing threading model (as shown in in section 7.1 have led to the following principles and definitions. Each principle is accompanied by an explanatory text or example situation. Each definition is expressed in pseudo-code in section 7.4.

**Threading Model Principle 1**
*A Component Entity should be suspended before a Delta can be composed with it.*

As Deltas in a Component Entity Delta stack may depend on each other's functionality, all running threads inside the Component Entity (i.e. all threads running in Deltas composed with that Component Entity) should be suspended before a Delta is composed with the Component Entity.

**Threading Model Definition 1** ***(Suspending a Component Entity)***
*Suspending a Component Entity involves suspending all Deltas composed with it.*

**Threading Model Definition 2** ***(Suspending a Delta)***
*Suspending a Delta involves suspending all entities related to the Delta, in the order as defined in Threading Model Definition 3.*

**Threading Model Definition 3** ***(Entities related to a Delta)***
*The following entities are related to a Delta:*

1. *Connectors connected to a provided Port of the Delta;*
2. *Child entities as defined by the user in the* `variables{}` *block (for example, nested Deltas);*
3. *The Component Entity with which the Delta has been composed;*
4. *The thread representing the* `run{}` *block as well as all threads defined in user-code inside the* `run{}` *block.*

**Threading Model Definition 4** ***(Suspending a Connector)***
*Suspending a Connector involves suspending all Deltas connected to a provided Port of the Connector.*

**Threading Model Definition 5** ***(Suspending child entities of a Delta)***
*Child entities of a Delta are entities defined by the user in the* `variables{}` *block of the Delta. This may be Archium entities (like Deltas or Component Entities), but Java objects as well (for example,* `java.util.Vector`*). Suspending child entities therefore means suspending all suspendable entities, i.e. entities which support suspending/unsuspending. As shown in section 7.4, suspendable entities must implement the* `ISuspendable` *interface.*

**Threading Model Definition 6** ***(Suspending the thread representing the*** `run{}` ***block as well as all threads defined in user-code inside the*** `run{}` ***block)***
*This is where the managing functionality of the threading model is necessary. All threads running inside a Delta (i.e. the thread representing the* `run{}` *block as well as all user-defined threads defined within the* `run{}` *block) are grouped into a pool of threads. Each Delta has its own pool and its own pool-managing entity. The pool-managing entity holds references to all threads registered with it. Suspending all these threads therefore simply means giving a suspend command to the encapsulating entity. More details can be found in section 7.4.*

**Threading Model Principle 2**
*After a Component Entity has been suspended, a Delta can be composed with it.*

As the Component Entity has been suspended, all threads of Deltas composed with it are suspended, meaning no thread is making use of functionality provided by others. Therefore it is now safe to compose a Delta with the Component Entity. To ensure that the Delta to be composed with the Component Entity has not already started any threads defined within it before composition, the following principles and definitions apply.

**Threading Model Principle 3**
*Threads defined within a Delta can only run if all requirements for running threads within a Delta have been satisfied.*

A number of prerequisites for running threads have been defined which prevent unwanted behavior, like threads starting before their containing Deltas have been composed with a Component Entity. These prerequisites are described below.

**Threading Model Definition 7** ***(Requirements for running threads within a Delta)***
*The following prerequisites must have been satisfied in order for threads defined within a Delta to run:*

- *The Delta must have completed its initialization phase, i.e. have completed construction, composition and configuration on entities introduced to the running system by it;*
- *The Delta must have been composed with a Component Entity;*
- *All other Deltas composed with that Component Entity must also satisfy all requirements for running threads, as Deltas in a Delta stack of a Component Entity may depend on each other;*
- *If the Delta has been introduced to the running system by another Delta, that parent Delta must also fulfill all requirements for running threads.*

**Threading Model Definition 8** ***(Fully-connectedness of a Delta)***
*A Delta is fully connected if all required Ports are connected to a Connector.*

The following section describes the state diagrams of the different entities which may be involved in composition.

## 7.3 Defining state models for Archium entities

The life-cycles of Archium entities have not been clearly defined. However, for a threading model to work reliably, knowledge of the state of each entity is essential. For example, threads may not be safely started if certain initialization code has not been executed yet. It must be *known* that the initialization has been completed before starting threaded code. Therefore, state models have been developed for Archium entities involved in the threading model (Delta, Component Entity and Connector). These models are discussed in the following subsections. Note that each state name in the state diagrams in the following subsections has been suffixed with an abbreviation of the state name. These abbreviation are used in section 7.5.

### 7.3.1 Delta

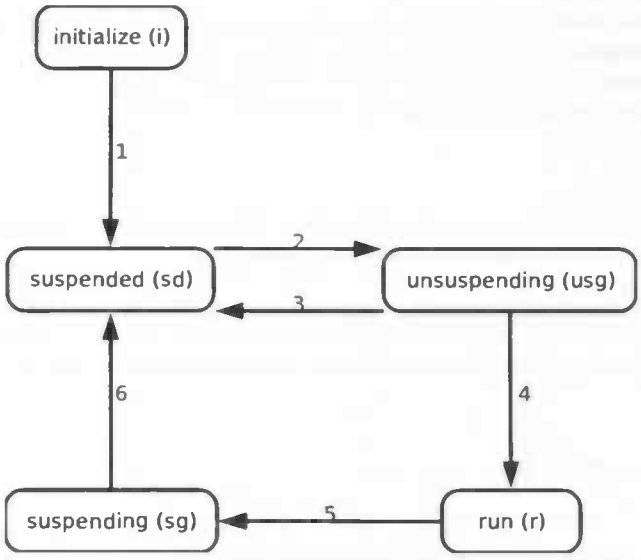


Figure 7.3: The Delta state diagram

## States and transitions

All states and transitions in figure 7.3 are discussed in this section. Also described is the relationship between the states, transitions and Threading Model Principles and Definitions. Initially, a Delta is in the *undefined* state which serves no other function than to indicate that the Delta has not been given a meaningful state yet.

- **initialize**
  After a Delta has been constructed, it will enter the *initialize* state and user code in the `initialization{}`, `configuration{}` and `composition{}` blocks will be executed to (recursively) instantiate entities defined in the `variables{}` block and perform configuration and composition on those entities. After the user code in these three blocks has been executed, the Delta will transition to state *suspended* (transition 1), as it has not been composed with a Component Entity yet. This is to ensure that threaded code in a newly instantiated Delta does not start running immediately, which is one of the fundamental rules of the developed threading model (Threading Model Principle 3).

- **suspending**
  On entering this state, entities related to the Delta will be suspended as described in Threading Model Definition 2. After all related entities have been suspended, the state of the Delta is changed to *suspended* (transition 6).

- **suspended**
  All entities possibly involved in composition (as described in the *suspending* state) have been suspended, and composition can now take place. After composition has been completed, the Delta will transition to the *unsuspending* state (transition 2).

- **unsuspending**
  The unsuspending Delta will unsuspend all entities suspended in the *suspending* state, if their respective thread running criteria have been met. The same goes for the unsuspending Delta itself. If it satisfies all requirements in Threading Model Principle 3, the entity managing the thread pool will be signalled to unsuspend its threads. That entity will decide for if the threads it manages should be *unsuspended* (if the threads had been suspended

before) or to *start* them (if they had not yet been started due to the fact that the Delta had just been introduced to the running system and had transitioned from the *initialization* to the *suspended* state). In case all requirements for thread running are satisfied the Delta will transition to the *run* state (transition 4). Otherwise, it will return to the *suspended* state (transition 3).

- **run**
  All threads in the Delta are running and iterating. The Delta will transition to the *suspending* state (transition 5) if a composition is about to take place.

As can be seen in figure 7.3, there exists a loop containing the *run*, *suspending*, *suspended* and *unsuspending* states. Each time when a Delta needs to suspend due to composition this loop is traversed, always checking if all requirements for thread running remain satisfied (Threading Model Principle 3).

### 7.3.2 Component Entity

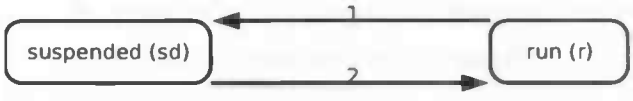


Figure 7.4: The Component Entity state diagram

A Component Entity can be viewed as the result of the application of a number of Deltas. References to all Deltas applied to a Component Entity are stored within the Component Entity. As Deltas may rely on functionality provided by other Deltas composed with the same Component Entity, an all-or-nothing principle applies: in order for one Delta to be able to run, all other Deltas must also be able to run.

Each time after a Delta has been composed with a Component Entity, the Component Entity's state is recalculated. If all composed Deltas fulfill all requirements for thread running (Threading Model Definition 3) , the state of the Component Entity will be set to *run*. Otherwise, it will be *suspended*.

#### States and transitions

All states and transitions in figure 7.4 are discussed in this section. Also described is the relationship between the states, transitions and Threading Model Principles and Definitions. Initially, a Component Entity is in the *undefined* state which serves no other function than to indicate that the Component Entity has not been given a meaningful state yet.

- **run**
  If a Component Entity has state *run*, it means that all Deltas composed with it fulfill all requirements for thread running (Threading Model Definition 3). If, after recalculation due to composition, at least one Delta does not satisfy all requirements for thread running, the Component Entity will transition to the *suspended* state.

- **suspended**
  The *suspended* state is the inverse of the *run* state. Not all requirements for thread running have been satisfied by all Deltas. If, after recalculation due to composition, all composed Deltas satisfy all requirements for thread running, the Component Entity will transition to the *run* state.

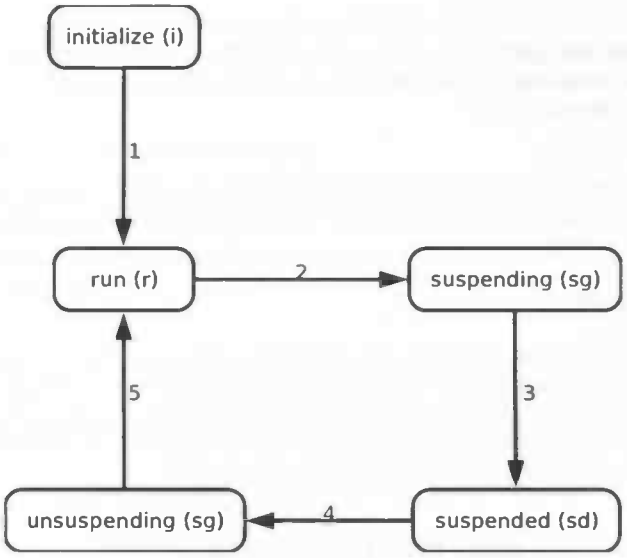


Figure 7.5: The Connector state diagram

### 7.3.3 Connector

A connector currently only passes messages between Deltas connected to its Ports. However, in the future a more sophisticated Connector may be desired with an ability to run threaded code.A more "intelligent" Connector could then be specified, for example including a load-balancing algorithm in a distributed system. The state diagram (and threading model) therefore has been prepared for such an extension and looks much like the Delta state diagram (figure 7.3), to allow for suspension/unsuspension of threads.

#### States and transitions

All states and transitions in figure 7.5 are discussed in this section. Also described is the relationship between the states, transitions and Threading Model Principles and Definitions. Initially, a Connector is in the *undefined* state which serves no other function than to indicate that the Connector has not been given a meaningful state yet.

- **initialize**
  After a Connector has been constructed, it will enter the *initialize* state and user code in the `initialization{}` block will be executed. After the user code in this block has been executed, the Connector will transition to state *run* (transition 1), ready for passing messages between Deltas.

- **run**
  The Connector will pass any messages received by its Ports to the correct receiving Delta (as defined by the user). On receiving a call to suspend by a Delta which will be involved in composition, state *suspending* is entered (transition 2).

- **suspending**
  All Deltas connected to a provided Port of this Connector will be suspended in this state (Threading Model Definition 4). After this process completes, the Connector will transition to state *suspended* (transition 3).

- **suspended**
  Composition can now take place. After composition has been completed, the Delta which suspended this Connector will unsuspend it, and the Connector will transition to the *unsuspending* state (transition 4).

- **unsuspending**
  All Deltas connected to a provided Port of this Connector will be unsuspended in this state. After this process completes, the Connector will transition to state *run* (transition 5).

## 7.4 Pseudo code

Using the state diagrams defined in section 7.3, this section describes the behavior of the involved Archium entities during composition, suspension and unsuspension. Section 7.4.1 focuses on composition of a Delta with a Component Entity and the remaining sections discuss suspension and unsuspension for Component Entities, Deltas and Connectors, respectively. Each section first provides the pseudo code, after which it is explained in text.

### 7.4.1 Composition of a Delta with a Component Entity

```
public void ComponentEntity.compose(Delta dAdd) {
    suspend();

    perform composition logic on dAdd;

    if all Deltas composed with this Component Entity fulfill the
        requirements for thread running {
        set the state of this Component Entity to "Run";
    }
    else {
        set the state of this Component Entity to "Suspended";
    }
}
```

After suspension of all Deltas composed with the Component Entity (the `suspend()` method, see section 7.4.2), composition logic is performed on the Delta to be added (`dAdd`). This involves modifying the Component Entity's Ports, interfaces etc. as specified by the user. Furthermore, the Delta is added to the list of composed Deltas and will be reused immediately to recalculate the Component Entity's state.

Note that unsuspension of suspended Deltas does not take place immediately after composition for the following reason. The method `compose()` is called whenever an `xCE plays yD` statement (Component Entity `xCE` is to be composed with Delta `yD`) is encountered in a user's `composition` block inside a Design Fragment, Delta or Design Fragment Composition. Consider the following `composition` block inside (for example) a Delta:

```
composition {
    aCE plays aD;
    bCE plays bD;
}
```

In addition, assume that `aD` and `bD` will be communicating using a Connector. If unsuspending would take place immediately after `aCE plays aD` had been executed, `bD` would not have been

composed with bCE yet, resulting in an error (Deltas must be composed with a Component Entity before they may perform any functionality). Therefore, unsuspending will only be done after *all* composition user-code has been executed for a particular Design Decision.

### 7.4.2 Suspending a Component Entity

```
public void ComponentEntity.suspend() {
    for each Delta dComposed composed with this Component Entity {
        dComposed.suspend();
    }
}
```

Suspending a Component Entity merely involves suspending all Deltas composed with it.

### 7.4.3 Unsuspending a Component Entity

```
public void ComponentEntity.unsuspend() {
    for each Delta dComposed composed with this Component Entity {
        dComposed.unsuspend();
    }
}
```

Unsuspending a Component Entity merely involves unsuspending all Deltas composed with it. Note that a Component Entity re-calculates its state only after composition (see section 7.4.1), so not when suspending/unsuspending itself.

### 7.4.4 Suspending a Delta

```
public void Delta.suspend() {
    set the state of this Delta to "Suspending";

    for each provided Port pp {
        if a Connector conn is connected to pp {
            conn.suspend();
            suspendedEntities.add(conn);
        }
    }

    for each child element child defined in the variables{} block {
        if child is an instance of ISuspendable {
            child.suspend();
            suspendedEntities.add(child);
        }
    }

    suspend the Component Entity with which this Delta has been composed;
    add that Component Entity to suspendedEntities;

    suspend any threads running inside this delta by signaling
```

```
    the thread-pool managing entity to suspend;

    set the state of this Delta to "Suspended";
}
```

As can be seen, entities related to a Delta are suspended in the order defined in Threading Model Definition 2. In addition, references to all suspended entities are stored in an ordered list named `suspendedEntities`. Unlike suspending a Delta, which is a recursive process (recursively suspending Connectors and child entities), unsuspending is *not*, as suspended entities may not be unsuspended due to the fact that some entities were removed from the running system by applying a Design Decision. Imagine two Deltas that were connected before the application of a Design Decision using a Connector. If a certain Design Decision removes the Connector, using a recursive unsuspension algorithm would mean at least one of the Deltas would not be unsuspended due to the fact that there is no reference anymore to the Delta as the Connector has been removed. Therefore, each Delta contains a list in which all suspended entities are stored. All elements of this list are unsuspended when unsuspending the Delta.

### 7.4.5 Unsuspending a Delta

```
public void Delta.unsuspend() {
    set the state of this Delta to "Unsuspending";

    for each entity ent in suspendedEntities {
        ent.unsuspend();
    }

    if this Delta fulfills all requirements for running threads {
        unsuspend any threads running inside this delta by signaling
        the thread-pool managing entity to unsuspend;
        set the state of this entity to "Run";
    }
    else {
        set the state of this Delta to "Suspended";
    }
}
```

As all suspended entities are stored in the list `suspendedEntities`, the first phase of unsuspending a Delta consists of unsuspending each entity stored in `suspendedEntities`. The second phase involves checking if the Delta fulfills all requirements for thread running (Threading Model Principle 3) and, in case it does, signaling the thread-pool managing entity to unsuspend its threads.

### 7.4.6 Suspending a Connector

```
public void Connector.suspend() {
    set the state of this Connector to "Suspending";

    for each provided Port pp {
        if a Connector conn is connected to pp {
            conn.suspend();
```

```
            suspendedEntities.add(conn);
        }
    }

    set the state of this Connector to "Suspended";
}
```

Suspending a connector resembles the first part of the suspension algorithm of a Delta: all entities (in this case: Deltas) connected to a provided Port of the Connector are signalled to suspend. All suspended entities are stored in a list `suspendedEntities`.

### 7.4.7 Unsuspending a Connector

```
public void Connector.unsuspend() {
    set the state of this Connector to "Unsuspending";

    for each entity ent in suspendedEntities {
        ent.unsuspend();
    }

    set the state of this Connector to "Run";
}
```

As all suspended entities are stored in the list `suspendedEntities`, unsuspending a Connector consists of unsuspending each Delta stored in `suspendedEntities`.

## 7.5 Example: chat server

As a validation of the designed threading model, this section provides a graphical view of the evolution of the chat server application (of which the architectural design is shown in section 4.2.1) after each application of a Design Decision. Also, each entity involved in the threading model is annotated with its state. Figure 7.6 shows the legend of symbols used throughout this

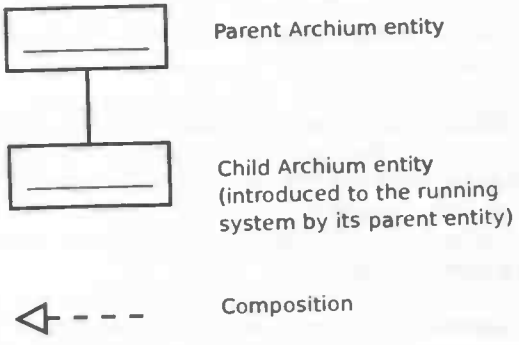


Figure 7.6: Legend of symbols used in the threading model example

section. A parent-child relationship between entities is depicted top-to-bottom and composition is indicated by a dashed arrow of which the source is a Delta and the target a Component Entity.

### 7.5.1 Design Decision 1: Introduce server component

In Design Decision 1, the `serverCE` Component Entity is introduced to the running system by means of Design Fragment `ServerDF`. As can be seen in figure 7.4, newly instantiated Component Entities immediately enter state *run*, so in figure 7.7, `serverCE` has been suffixed with *(r)*. The suffixes defined in section 7.3 are used to indicate the state of each entity in this section. Design Fragment `ServerDF` has no role in the threading model, so it has not been given a state (nor suffix).

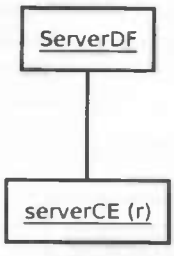


Figure 7.7: Introducing the `serverCE` Component Entity

### 7.5.2 Design Decision 2: Flesh out the server component

Design Decision 2 fleshes the `serverCE` Component Entity out into two Component Entities: `communicatorCE` and `clientHandlerCE`. The Design Fragment responsible for introducing these two Component Entities (`FleshOutServerDF`) is shown in figure 7.8. It contains only Delta `fleshOutServerD`, which in turn will introduce the two Component Entities to the running system. Every instantiated Delta will first enter the *initialization* state (as specified in figure 7.3).

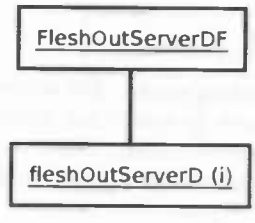


Figure 7.8: Delta `fleshOutServerD`, introduced by Design Fragment `FleshOutServerDF`

In its *initialization* state, Delta `fleshOutServerD` will construct, compose and configure any user-defined entities within it. In this case, that means constructing `communicatorCE` and `clientHandlerCE`. As shown before, newly instantiated Component Entities immediately enter the *run* state, so the resulting situation is shown in figure 7.9. After finishing its *initialization* phase, Delta `fleshOutServerD` will enter the *suspended* state, awaiting composition with a Component Entity. This is shown in figure 7.10.

Associated with Design Decision 2 is a Design Fragment Composition which specifies how Design Fragment `FleshOutServerDF` is to be integrated with the entities in the running system. In this case, the Design Fragment Composition specifies that Delta `fleshOutServerD` in `FleshOutServerDF` has to be composed with `serverCE` (which is already part of the running system, as it has been introduced to it by Design Decision 1). To illustrate the composition process, both the running system and `FleshOutServerDF` have been depicted in figure 7.11.

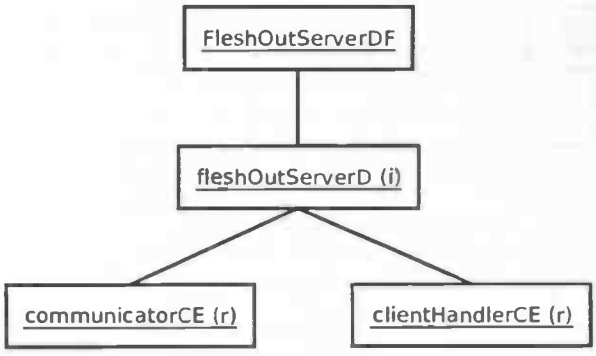


Figure 7.9: Delta `fleshOutServerD` constructs `communicatorCE` and `clientHandlerCE`

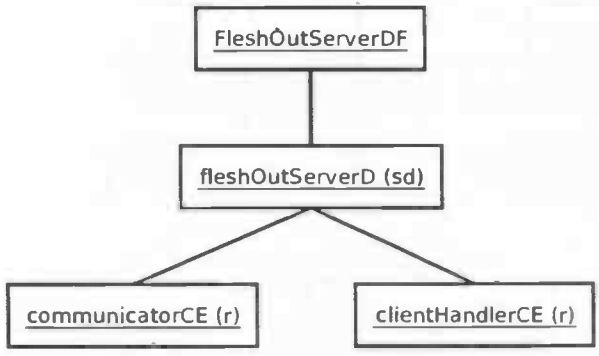


Figure 7.10: Delta `fleshOutServerD` is awaiting composition with a Component Entity

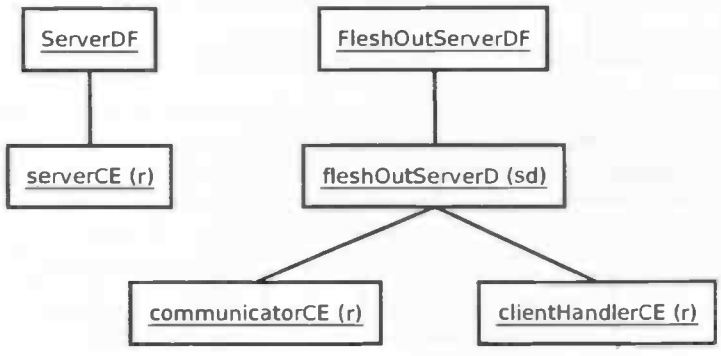


Figure 7.11: The running system and `FleshOutServerDF`

Upon composition, the Component Entity on which the composition is about to take place (Component Entity `serverCE`) first suspends all Deltas applied to it (as defined in section 7.4). In this case, no Deltas had been composed with `serverCE` yet, so no Deltas are suspended. Afterwards, the actual composition takes place which means that `serverCE` will be split up into two sub-Component Entities `communicatorCE` and `clientHandlerCE` (shown in figure 7.12). Delta `fleshOutServerD` now enters the *unsuspending* state (figure 7.13) and determines if it fulfills all requirements for running threads. In fact it does fulfill all requirements, however it contains

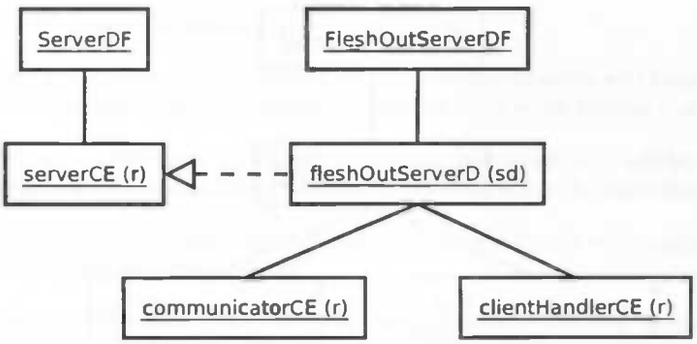


Figure 7.12: Composition of `fleshOutServerD` with `serverCE`

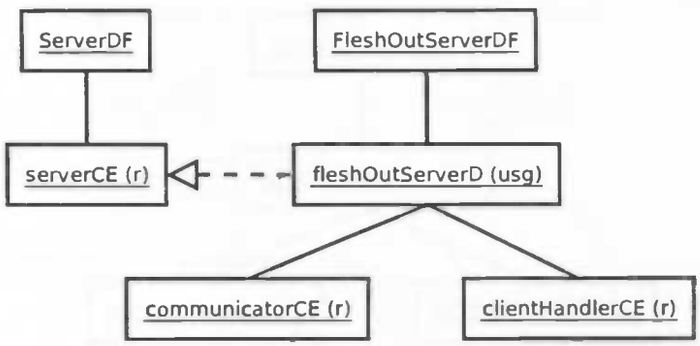


Figure 7.13: Delta `fleshOutServerD` is unsuspending

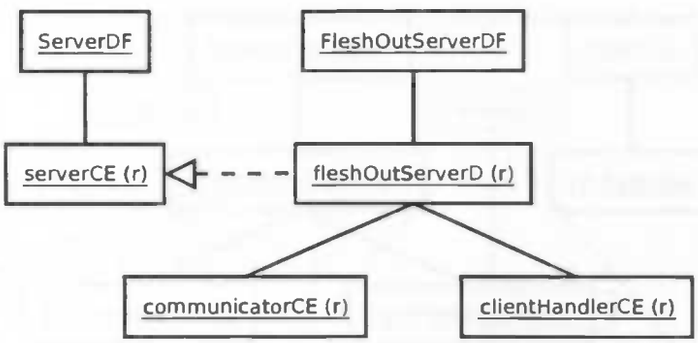


Figure 7.14: Delta `fleshOutServerD` is running

no threads (it only has a fleshing-out role, adding no functionality to any Component Entity). Finally, as dictated by the Delta state diagram (figure 7.3), `fleshOutServerD` enters the *run* state (figure 7.14).

### 7.5.3 Design Decision 3: Implement the server

The role of Design Decision 3 is to provide the implementation of both the `communicatorCE` and `clientHandlerCE` Component Entities. Therefore, two Deltas for providing that implementa-

tion (communicatorD and singleThreadedClientHandlerD) are introduced in the corresponding Design Fragment (ImplementServerDF). In addition, both Deltas are connected via a Connector which is also introduced in ImplementServerDF. Like Delta fleshOutClientD in Design Fragment FleshOutClientDF in Design Decision 2, both Delta communicatorD and Delta singleThreadedClientHandlerD first enter the initialization state after which they transition to the *suspended* state, awaiting composition with a Component Entity.

The focus is now on Connector communicatorClientHandlerC. After construction by Design Fragment ImplementServerDF, it first enters the initialization state (figure 7.15). As shown in figure 7.5, it will then enter state *run* (figure 7.16).

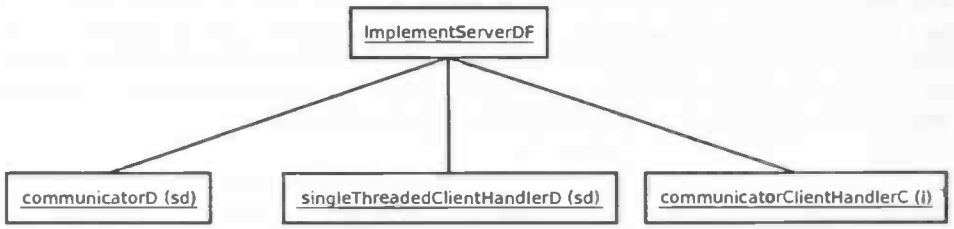


Figure 7.15: Connector communicatorClientHandlerC enters the *initialization* state

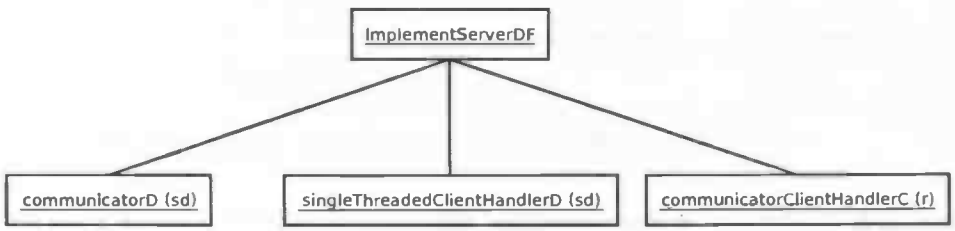


Figure 7.16: Connector communicatorClientHandlerC enters the *run* state

Design Fragment ImplementServerDF will now connect Delta communicatorD and Delta singleThreadedClientHandlerD using communicatorClientHandlerC. The Design Fragment Composition associated with Design Decision 3 specifies that the Component Entities named communicatorCE and clientHandlerCE (both introduced to the running system by Design Decision 2) should be composed with communicatorD and singleThreadedClientHandlerD, respectively. Both the running system and ImplementServerDF are shown in figure 7.17.

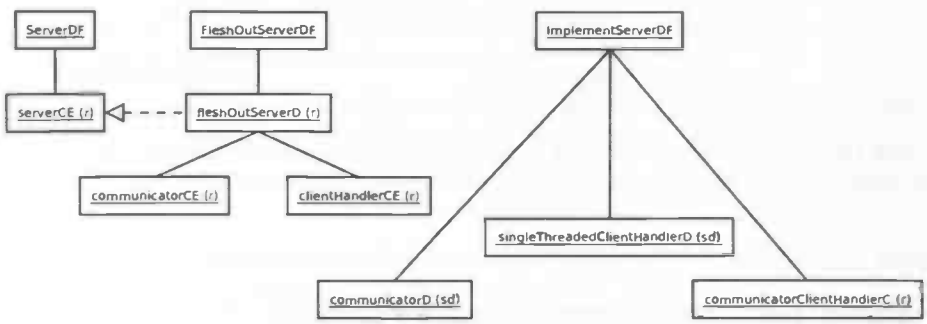


Figure 7.17: The running system and ImplementServerDF

The sequence of events now is as follows:

- Component Entities `communicatorCE` and `clientHandlerCE` will both try to suspend any Deltas composed with them, however there are none so no entities will be suspended;
- Deltas `communicatorD` and `singleThreadedClientHandlerD` will be composed with Component Entities `communicatorCE` and `clientHandlerCE`, respectively (figure 7.18);
- Component Entities `communicatorCE` and `clientHandlerCE` will recalculate their state: both find that their state remains *run*;
- Both Deltas will determine if they fulfill all requirements for running threads (figure 7.19); they both do, so their threads start running (as shown in figure 7.20): Component Entity `communicatorCE` now has a threaded implementation for accepting chat client connections Delta `singleThreadedClientHandlerD` now has a (single-) threaded implementation for handling any request made by the connected chat clients.

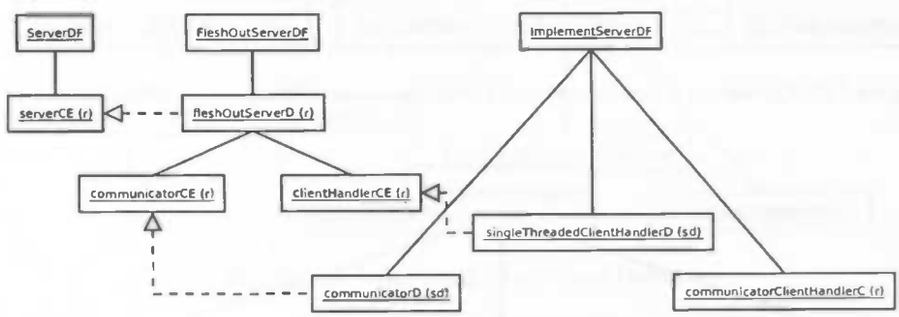


Figure 7.18: Composition of `communicatorD` and `singleThreadedClientHandlerD`

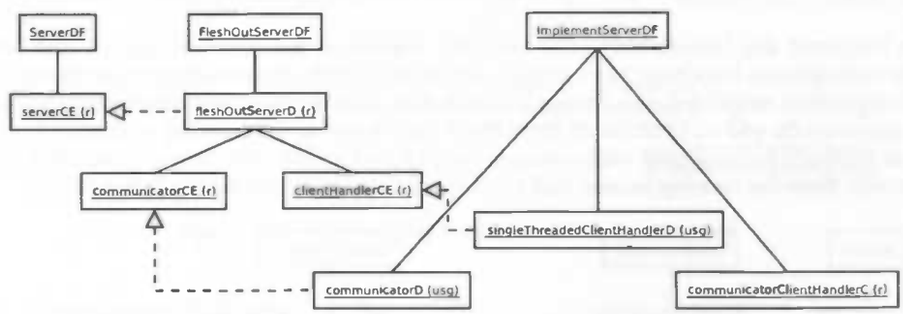


Figure 7.19: Deltas `communicatorD` and `singleThreadedClientHandlerD` are unsuspending

### 7.5.4 Design Decision 4: Changing the server implementation

In Design Decision 4, the implementation of `clientHandlerCE` is changed from a single-threaded client handler to a multi-threaded client handler. This is accomplished by introducing a Delta

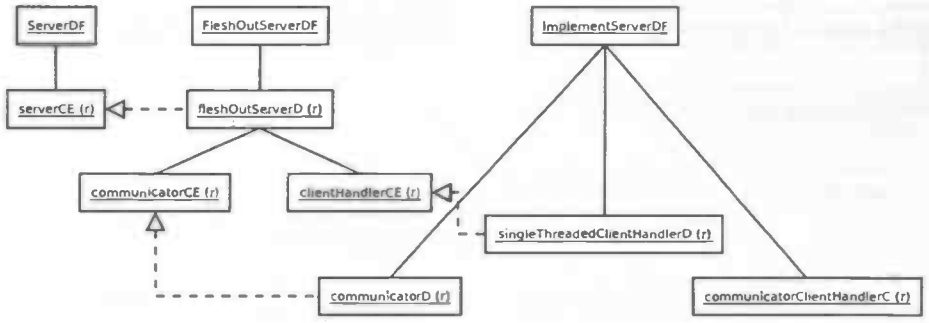


Figure 7.20: Threads within `communicatorD` and `singleThreadedClientHandlerD` are running

`multiThreadedClientHandlerD` to the running system which will be composed with Component Entity `clientHandlerCE` and catch all requests previously made to functionality provided by `singleThreadedClientHandlerD` using a composition technique called *Adapter* [4].

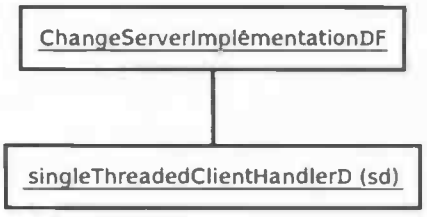


Figure 7.21: Delta `multiThreadedClientHandlerD` ready for composition

Figure 7.21 shows Delta `singleThreadedClientHandlerD` after initialization and waiting upon composition with a Component Entity. Figure 7.22 shows the running system and Design Fragment `ChangeServerImplementationDF`, the Design Fragment which introduced `singleThreadedClientHandlerD`.

The Design Fragment Composition associated with Design Decision 4 specifies that Component Entity `clientHandlerCE` should be composed with `multiThreadedClientHandlerD`. The threading model therefore takes a number of actions:

- `clientHandlerCE` suspends all Deltas composed with it (its Delta stack), in this case only Delta `singleThreadedClientHandlerD`;
- `singleThreadedClientHandlerD` suspends `communicatorD`, as it is connected to one of its provided Ports using `communicatorClientHandlerC` (figure 7.23);
- After both involved Deltas have suspended (figure 7.24), composition takes place (figure 7.25);
- All suspended Deltas transition to the *unsuspending* state (figure 7.26), including the newly introduced `multiThreadedClientHandlerD`;
- And finally, each unsuspending Delta checks if it fulfills all requirements for thread running (in this case: each one does) and transition to the *run* state (figure 7.27).

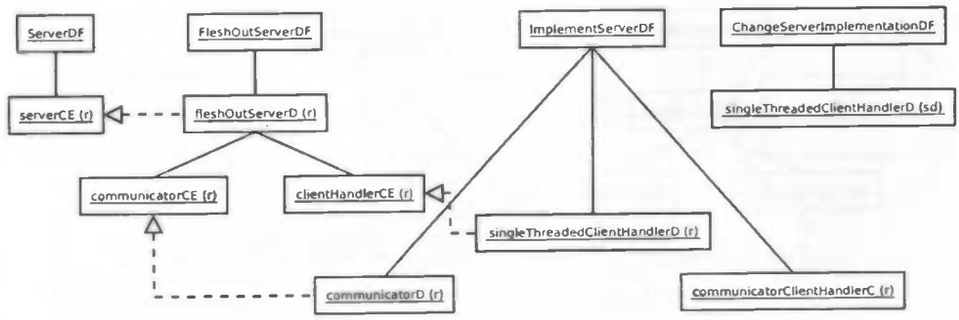


Figure 7.22: The running system together with Design Fragment `ChangeServerImplementationDF`

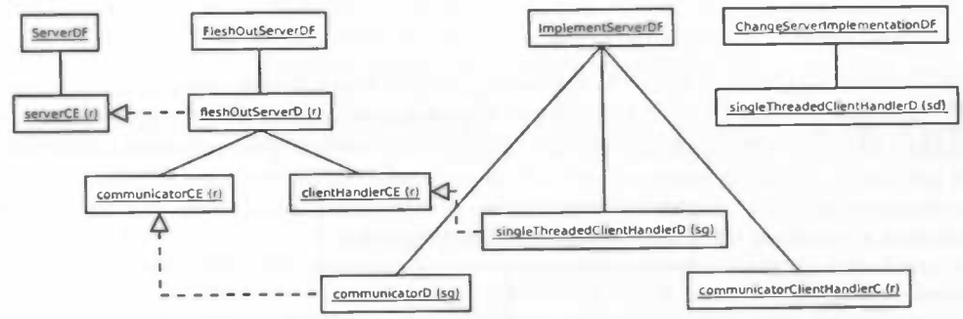


Figure 7.23: Suspending all involved Deltas

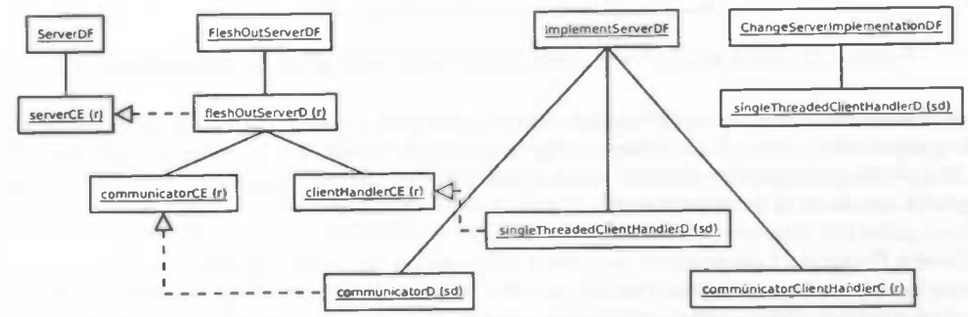


Figure 7.24: All involved Deltas have been suspended

## 7.6 Implementation

This section discusses the implementation of the added threading model. Section 7.6.1 discusses the addition of a managed threading framework and section 7.6.2 shows the modification made to the Reflection API.

### 7.6.1 Managed threading framework

Previous work [14] included a managed threading framework which has now been incorporated into the Archium source code. The framework consists of two classes, which are discussed briefly in this section.

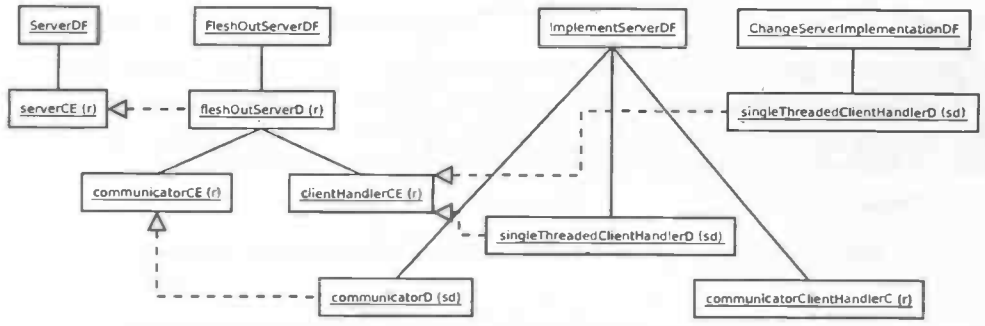


Figure 7.25: Composition takes place

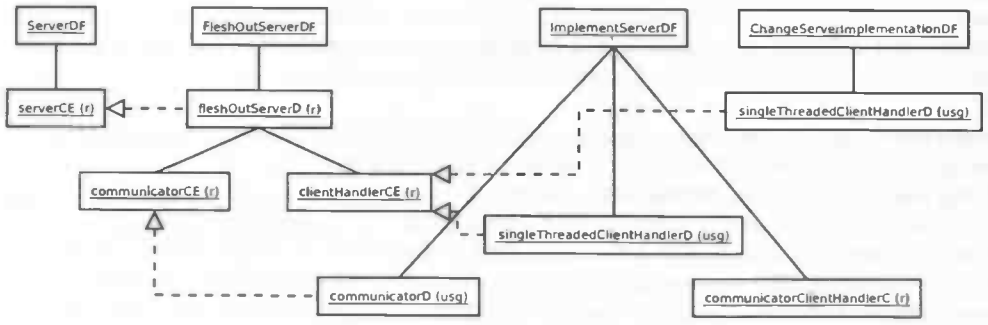


Figure 7.26: Unsuspending all involved Deltas, including the newly introduced `multiThreadedClientHandlerD`

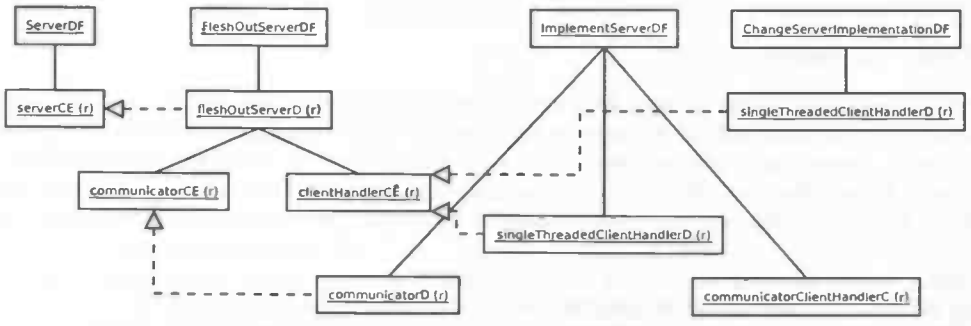


Figure 7.27: All involved Deltas are running again

## The class `ManagedThreadPool`

The class `ManagedThreadPool` is a class for grouping a number of threads and performing operations like starting, stopping, suspending and unsuspending on them. In the managed threading model, each Delta has been extended with one instance of a `ManagedThreadPool`, with which all threads defined within the Delta (in the `run{}` block) are registered. This instance provides an easy way for suspension/unsuspension all threads within a Delta using a single function call in case of composition.

A small modification had to be made to the existing `ManagedThreadPool` class after the following problem occurred when developing the traditional Producer-Consumer application in

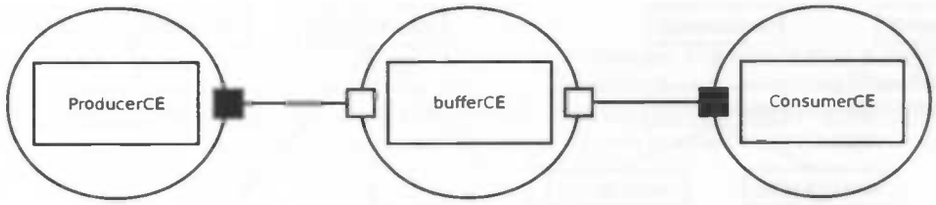


Figure 7.28: The producer-consumer application modeled in Archium

Archium. Assume both the producer and consumer are represented by a Component Entity, each composed with a Delta for providing the producer and consumer functionality, respectively. In addition, there is a shared buffer (with limited capacity) Component Entity in which the producer tries to write values and the consumer tries to read values. Both the functionality of the consumer and producer are provided by a threaded implementation and both are connected to the buffer by means of a connector. The situation is depicted in figure 7.28.

Both the producer and consumer apply the Java `wait-notify` mechanism for signaling each other that an item has been produced or consumed. Assume that another Delta is to be composed with the buffer Component Entity. Also assume that the consumer is waiting (using the Java `wait()` method) as the buffer is empty. The buffer Delta will suspend both the consumer and producer as they are connected to a provided Port of the buffer. If the producer is suspended first, the consumer will never receive a `notify()` call, and will not suspend (a thread may only suspend after an iteration of `iterate` has been completed, which is not the case as the consumer is waiting). This results in deadlock. As there is no way of knowing what the right order of suspending is for an arbitrary application, a generic solution has been found: allow threads that do not suspend immediately to run for three more seconds before forcefully suspending them.

### The class `ManagedThread`

Conventional `java.lang.Thread` instances cannot register themselves with an instance of `ManagedThreadPool`, as they do not provide enough information for them to be manageable by the pool. Instead, an extended version of `java.lang.Thread`, called `ManagedThread`, is to be used. It provides methods similar to those found in the conventional `java.lang.Thread` class, but also introduces new functionality specifically needed for the managed threading model.

As well as being the class into which user-code from the `run{}` block is generated, users can define their own thread classes by extending the `ManagedThread` class, similar to extending the `java.lang.Thread` class in conventional Java programming. Users have to implement their threaded code in the `iterate()` method, which will be executed iteratively, unless a call to `stopRepeat()` is made inside the `iterate()` method. This is similar to implementing the `run()` method in conventional `java.lang.Thread`-extending classes.

The expressive power of Managed Threads is exactly the same as the standard Java thread class `java.lang.Thread`, as the class `ManagedThread` itself extends `java.lang.Thread`. This means that users do not have to make compromises when switching to Managed Threads, and can still use functionality like `wait`, `notify`, locking, scheduling and so forth.

## 7.6.2 Reflection API

As well as adding a managed threading framework, the Reflection API needed modification as well, which is discussed is this section.

### The class `ArchitecturalEntity`

All Archium entities extend from `ArchitecturalEntity.java`, so additional functionality required by multiple/all entities has to be placed in this file. In case of the managed threading model, the only functionality that had to be added is an enumeration of entity states and associated getter- and setter methods. These methods are used as dictated by the the state diagrams defined in section 7.3.

### The class `Delta`

The `Delta` class had to be provided with `suspend()` and `unsuspend()` methods, as shown in section 7.4. Furthermore, helper methods had to be introduced for determining if a Delta fulfills all requirements for thread running (section 7.2). These methods are:

- `isFullyConnected()` for determining if all provided Ports have a Connector attached to them;
- `isComposed()` for determining if a Delta has been composed with a Component Entity.

In addition, a `public` method `isAllowedToRun()` has been created which, in addition to calls to the above methods, also checks:

- if the Component Entity with which the Delta has been composed has state *run*;
- if the Delta has been introduced to the running system by a parent Delta, if that parent Delta also `isAllowedToRun()`.

This method `isAllowedToRun()` is used by Component Entities for re-calculating their state after composition has taken place.

The final change which had to be made to the `Delta` class was to keep references to all child elements defined in the `variables{}` block, as they are used in the suspension/unsuspension processes.

### The class `ComponentEntity`

Like the `Delta` class, the `ComponentEntity` class also needed implementation of the `suspend()` and `unsuspend()` methods. The implementation resembles the pseudo-code shown in section 7.4. Furthermore, the method responsible for composition of a Delta with a Component Entity (`addDelta`) needed modification to incorporate the pseudo-code given in section 7.4.

### The class `Connector`

The `Connector` class required a relatively small modification: the class has been extended with `suspend()` and `unsuspend()` methods as specified in section 7.4.

## 7.6.3 Code generation

The only large modifications which had to be made in order for the managed threading model to work had to be made in the Delta code generation template. These changes are discussed below.

## Delta

The code generation process for a Delta required two major changes. The first change is the generation of user-code in the `run{}` block into a `private`, `ManagedThread`-extending class called `Runner`. Previously the code had been generated into a `ThreadDelta`-extending class, however this class has now become obsolete. Each Delta has its own `Runner` instance representing the user-code in the `run{}` block.

The second major change is the scanning of the child entities defined in the `variables{}` block for uses of conventional `java.lang.Thread` objects. As these are not manageable by an instance of `ManagedThreadPool`, and thus not by Archium, use of them is discouraged, and the user is informed of this by means of a warning message on compilation of his/her Archium application source-code.

# Chapter 8

# Visualizer extensions

In order to validate at run-time the added requirements model and threading model, the Archimon Visualizer has been extended with additional functionality. Section 8.1 shows how the Design Decision view has been modified to display relations between Design Decisions. Section 8.2 shows the addition of a process view, and section 8.3 discusses the development of a facility for applying Design Decisions has been incorporated, thereby answering the research question "How can run-time application of Design Decisions be achieved in Archium?" (as raised in section 3.1.4).

## 8.1 Design Decision view

The Design Decision view of the Archimon Visualizer shows all Design Decisions applied in a running system. However, originally, it did not show the relationships between Design Decisions. The view therefore has been extended to show an arrow originating at Design Decision $DD_1$ and pointing at Design Decision $DD_2$ if $DD_1$ introduces a Delta uses elements introduced by $DD_2$. Figure 8.1 shows the chat server Design Decisions and their relationships. Another extension made to the Design Decision view is to display the rationale of a Design Decision as a tool-tip when the mouse cursor is hovering over a Design Decision element. This is illustrated in figure 8.2, which shows the rationale for Design Decision 2 of the chat client. In addition, elements referenced from within the rationale text (as described in chapter 6), will show up as a hyperlink. Although currently nothing happens when clicking on such a hyperlink, in the future the visualizer may switch to a view of the referred element, or provide some other use for the hyperlinks.

## 8.2 Process view

In conjunction with the managed threading model, the visualizer has been extended with a Process View to also display the relations between Managed Threads and Deltas and the state of each Managed Thread and Delta. The aforementioned producer-consumer example is shown as an example in figure 8.3. An ellipse-shaped figure represents a Delta, and a triangle-shaped figure represents a Managed Thread. Each Delta and Managed Thread have their state suffixed in their name and each Managed Thread is connected to its containing Delta using a line segment. The figure shows two producer-threads and two consumer-threads at work (their state is *run*).

Extending the Archimon Visualizer to incorporate a Process View was a relatively easy task. Each Archium entity, when created, fires an event indicating that it has been created. The Visualizer receives the event and adds a graphical representation of the created entity to every view that is

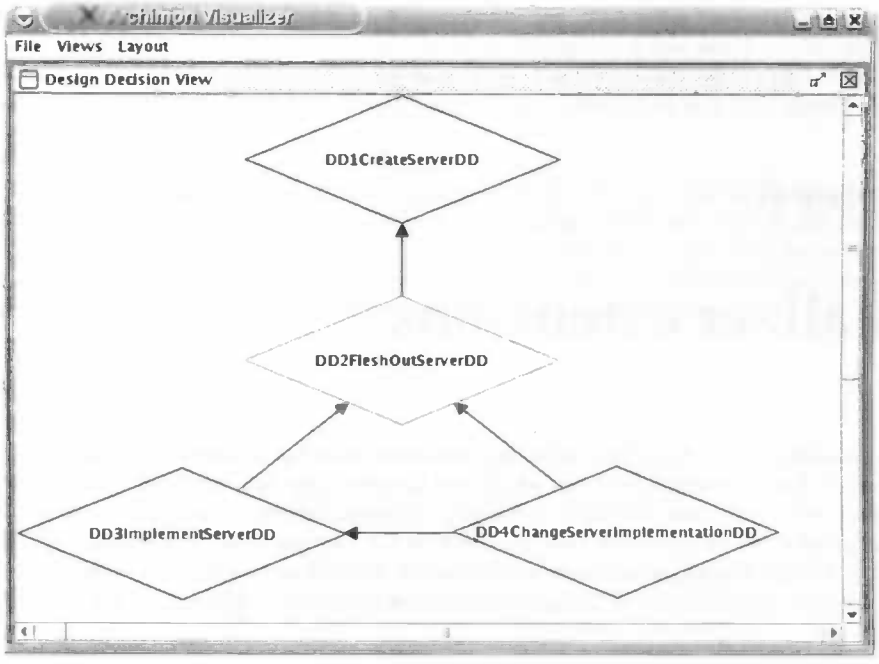


Figure 8.1: The Design Decision view displays relationships between Design Decisions

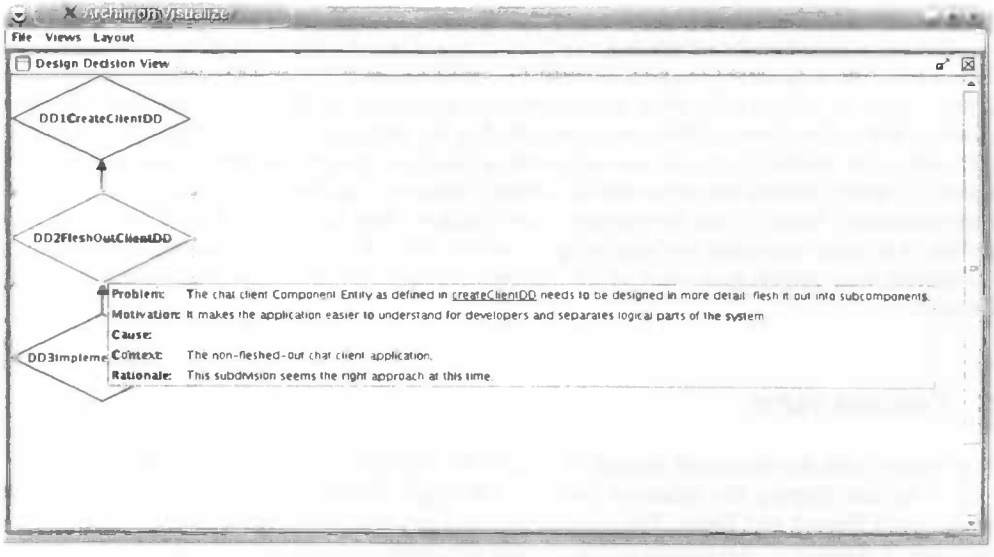


Figure 8.2: The Design Decision view shows the rationale for a Design Decision

interested in the entity. Like the "Component and Connector View" is interested in Component Entities and Connectors, the Process View is interested in Deltas and Managed Threads.

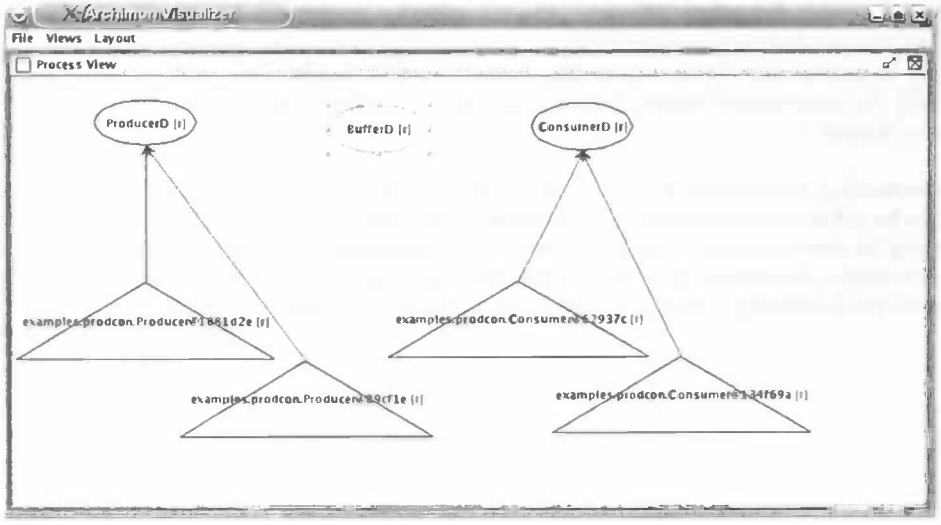


Figure 8.3: The Process View of the Archimon Visualizer

## 8.3 Applying Design Decisions at run-time

One of the goals of Archium is to allow recomposition of (i.e., apply Design Decisions to) an Archium-built application at compile-time as well as at run-time. Currently this can only be performed in at compile-time. There exists no facility for applying a user-specified Design Decision at run-time.

To demonstrate run-time reconfigurability, a facility for applying Design Decisions at run-time has been added to the Archimon Visualizer. As a Design Decision is always applied to a Design Fragment (usually the Design Fragment representing the running system), the first step has been to add a new view to the visualizer, called the Design Fragment view. This view is shown in figure 8.4. The figure shows all Design Fragments defined in the chat client. By selecting a Design

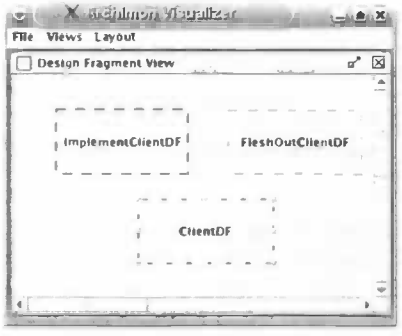


Figure 8.4: The Design Fragment view of the Archimon Visualizer

Fragment, and subsequently clicking *Apply Design Decision* in the *File* menu, a file-chooser dialog appears allowing the user to select a Java `.class` file representing a Design Decision. This Design Decision will then be applied to the selected Design Fragment.

Applying a Design Decision involves applying all changes specified in the Design Decisions to

the selected Design Fragment. The core functionality for applying a Design Decision already existed so this needed no additional implementation. However, a few small modifications were necessary to incorporate the new threading model into the Design Decision application process (especially the suspension/unsuspension of threads in Deltas), but this has already been described in chapter 7.

By implementing this facility, the research question "How can run-time application of Design Decisions be achieved in Archium?" is answered, but some additional work is still need: a bug concerning the compile-time resolving of entities was identified which prevents the application of Design Decisions at run-time from always functioning correctly due to the fact that some entities are not resolved correctly. This bug affects other Archium functionality as well.

# Chapter 9

# Validation

## 9.1 Introduction

This section provides an overview of the implementation of the chat case, incorporating the requirements model, threading model and comment-block referencing, and serves as validation of the work done. The full source code is available in the Archium CVS tree, and excerpts will be presented here to illustrate the implementation.

## 9.2 Protocol

Both the chat client and server implement a simple protocol consisting of string-encoded commands. The `LOGIN` command is sent by the client to the server to indicate that a user wants to log

| Command | Arguments |
|---|---|
| `LOGIN` | Username |
| `LOGOUT` | - |
| `MESSAGE` | Recipient/sender username, message |
| `LIST` | (List of online users) |

Table 9.1: The simple chat protocol commands

in. As communication takes place using sockets, the server records the socket to which the client has been connected on login. Therefore, when a `LOGOUT` command is issued by a client, sending a username again is not necessary, as the server can look up the client which has been connected to the socket on which the `LOGOUT` command has been issued. The `MESSAGE` command is used by both the server and client to send text messages, and the `LIST` command is used by the client to request the list of online users and by the server to push the list of online users to the client. All arguments are appended to the command string by means of a separating character (#). Optional arguments (only with the `LIST` command) are shown in parentheses. Figure 9.1 shows an example conversation using the simple chat protocol. After Bill has logged in and has been greeted by the server, he requests the list of online users, which in this case happen to be Alice and Bob. He then sends a greeting to Alice, who responds by returning a greeting. Finally, Bill logs out and is wished goodbye by the server.

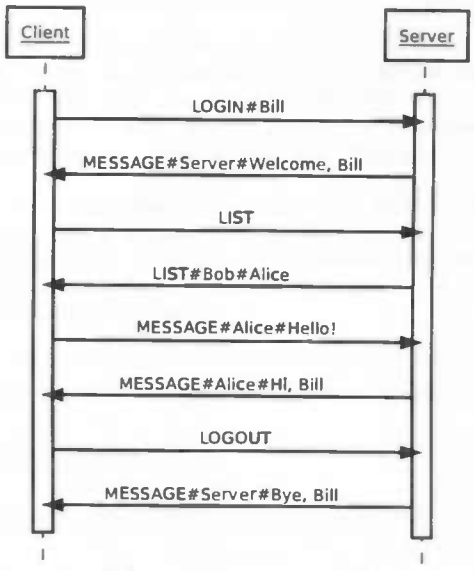


Figure 9.1: Example conversation using the simple chat protocol

## 9.3 Server

### 9.3.1 Requirements

All server requirements have been captured in a separate file, allowing them to be referenced from comment-blocks throughout the server's source code. For example, the requirement specifying that the server should support single-threaded operation is shown in figure 9.2. As can be seen, this requirement has been classified as a *must have*, functional requirement, and its initial implementation status is 0%, meaning no (partial) implementation exists yet for this requirement. It refines requirement `Chat_Server_3REQ`, which says that *the chat server must be able to run on high-end systems (with a large amount of resources) as well as on lower-end systems (with less*

```
requirement Chat_Server_3_1REQ {
    @name {#
        Server single-threaded operation.
    #}

    @description {#
        The chat server must be able to operate in single-threaded mode
        for systems with a small amount of resources.
    #}

    priority MustHave;
    status 0%;
    type Functional;
    refines Chat_Server_3REQ;
    category ServerRequirementsREQC;
    desired by SysAdminSH;
}
```

Figure 9.2: Requirement `Chat_Server_3REQ`

```
stakeholder SysAdminSH {
    @name {#
        System administrators.
    #}

    @description {#
        People who maintain the systems on
        which the chat server/clients will
        run.
    #}
}
```

Figure 9.3: Stakeholder SysAdminSH

```
requirement category ServerRequirementsREQC {
    @name {#
        Server requirements.
    #}

    @description {#
        All requirements having to do with
        (non)functional parts of the chat
        server.
    #}
}
```

Figure 9.4: Requirement Category ServerRequirementsREQC

*resources)*. The stakeholders who desire this requirement to be implemented are system administrators, as they will be installing and maintaining the chat server software. They are referred to by SysAdminSH and are defined as illustrated in figure 9.3. Requirement Chat_Server_3_1REQ is part of Requirement Category ServerRequirementsREQC, which is declared as shown in figure 9.4. By specifying the requirements in Archium source code, goal 3 of the chat case goals (as described in section 4.1) is met.

### 9.3.2 Comment-block referencing

As all server requirements are now defined, they can be referenced using the comment-block referencing mechanism developed in section 6. For example, when taking a look at server Design Decision 3, the text inside the @description comment-block refers to many Requirements, as shown in figure 9.5. As well as Requirements, Requirement Categories and Stakeholders, which are system-wide referable, locally available objects are referable as well. For example, Design Fragment instance implementServerDF, which is constructed within Design Decision 3, is referenced in the text of the @description comment-block, as also shown in figure 9.5. By allowing objects to be referenced from within comment-blocks, goal 4 of the chat case goals (as described in section 4.1) is met.

### 9.3.3 Single- and multithreaded operation

As described in section 4.2.1, the chat server supports single- and multithreaded client-handling. The Delta responsible for introducing the single-threaded client-handling implementation is shown in figure 9.6 (slightly simplified). The code inside the run{} block is executed as a thread and repeatedly checks, for each connected client, if a protocol command (as defined in table 9.1) has been received in its respective socket buffer. If a command has been received, it is handled by the handleMessage() method, otherwise nothing happens for that client. All connected

```
@description {#
    Provide both the communicator and the client handler
    Component Entities with an implementation by composing a
    Delta with each one. As well as the two Deltas, a
    Connector is contained in the Design Fragment [implementServerDF],
    which will handle messages between the two Deltas.

    In this solution, the implementation for the client handler is
    explicitly chosen to be a single-threaded one, as resources on
    the target platform may be limited [Chat_Server_3_1REQ]. The
    communication protocol is chosen to be a simple one, as defined
    in requirement [Chat_Server_2REQ].
#}
```

Figure 9.5: The `@description` comment-block of server Design Decision 3

```
delta SingleThreadedClientHandlerD {
    provides {
        port communicatorP implements CommunicatorClientHandlerI;
    }

    variables {
        java.util.Vector<java.net.Socket> sockets;
        examples.chatjava.UserSocketTable ust;
    }

    run {
        synchronized(this) {
            try {
                for (int i = 0; i < sockets.size(); i++) {
                    java.net.Socket s = (java.net.Socket) sockets.get(i);
                    java.io.BufferedReader br =
                        new java.io.BufferedReader(
                            new java.io.InputStreamReader(
                                s.getInputStream()));

                    if (br.ready()) {
                        handleMessage(s, br.readLine());
                    }
                }

            }
            catch (Exception e) {
            }
        }
    }

    initialization {
        sockets = new java.util.Vector<java.net.Socket>();
        ust = new examples.chatjava.UserSocketTable();
    }

}
```

Figure 9.6: Delta `singleTheadedClientHandlerD`

```
delta MultiThreadedClientHandlerD {
    provides {
        port communicatorP implements CommunicatorClientHandlerI;
    }

    variables {
        examples.chatjava.UserSocketTable ust;
    }

    initialization {
        ust = new examples.chatjava.UserSocketTable();
    }

    implementation {
        public void acceptClient(java.net.Socket socket) {
            getManagedThreadPool().addThread(
                new examples.chatjava.ClientHandlerThread(
                    getManagedThreadPool(), socket, ust));
        }
    }
}
```

Figure 9.7: Delta `multiTheadedClientHandlerD`

clients are mapped to the server socket to which they are connected by means of an instance of `UserSocketTable`. Server Design Decision 4 specifies that the client handling implementation should be changed from a single-threaded to a multi-threaded implementation. The Archium Delta introducing the new functionality is shown in figure 9.7. Instead of handling each client command within the same thread, each time a client connects, a new client-handling thread is created by using the `addThread()` method on the Delta's `ManagedThreadPool` instance. Again, an instance of `UserSocketTable` instance is used for mapping clients to server sockets. Figure 9.8 shows the new threaded client-handler class. It extends `ManagedThread`, and the code to be iterated is placed inside the `iterate()` method. By using threaded code and dynamically switching from a single-threaded to a multi-threaded client-handling implementation, goals 2 and 5 of the chat case goals (as described in section 4.1) are met.

```
package examples.chatjava;

import java.io.BufferedReader;

public class ClientHandlerThread extends ManagedThread {
    private BufferedReader in;
    private UserSocketTable ust;
    private Socket socket;

    public ClientHandlerThread(
        ManagedThreadPool mtp, Socket socket, UserSocketTable ust) {

        super(mtp);
        try {
            this.in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));
            this.socket = socket;
            this.ust = ust;
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }

    public void iterate() {
        try {
            if (in.ready()) {
                handleMessage(in.readLine());
            }
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }

    public void handleMessage(String message) {
        ...
    }
}
```

Figure 9.8: Class `ClientHandlerThread`

## 9.4 Client

### 9.4.1 Requirements

Like the chat server, the chat client also has its requirements defined in a separate file. An example has been provided in figure 9.9, showing the need for a graphical display of the list of online users. This requirement is a *must have*, functional requirement for which no implementation exists yet (its status is 0%). It is desired by Stakeholders of type `EndUserSH` (defined in figure 9.10) and is part of Requirement Category `ClientRequirementsREQC` (defined in figure 9.11).

```
requirement Chat_Client_3REQ {
    @name {#
        Chat client shows other users connected to the chat sever.
    #}

    @description {#
        The chat client must graphically show all users currently
        connected to the same server as the chat client.
    #}

    priority MustHave;
    status 0%;
    type Functional;
    category ClientRequirementsREQC;
    desired by EndUserSH;
}
```

Figure 9.9: Requirement `Chat_Client_3REQ`

```
stakeholder EndUserSH {
    @name {#
        The end user of the system.
    #}

    @description {#
        (Possibly) non-technical people who
        just want to chat without worrying about
        implementation details.
    #}
}
```

Figure 9.10: Stakeholder `EndUserSH`

```
requirement category ClientRequirementsREQC {
    @name {#
        Client requirements.
    #}

    @description {#
        All requirements having to do with
        (non)functional parts of the chat
        client.
    #}
}
```

Figure 9.11: Requirement Category `ClientRequirementsREQC`

```
@description {#
    We provide implementations for all
    Component Entities fleshed-out in [fleshOutClientDD]:

    1) [fleshOutClientDD::fleshOutClientD::communicatorCE]:
    an implementation is provided by [implementClientDF::communicatorD]
    which handles communication with the chat server using
    a simple protocol [Chat_Client_2REQ].

    2) [fleshOutClientDD::fleshOutClientD::uiCE]:
    a graphical user interface is provided by
    [implementClientDF::uiD] which lets
    users send and receive text messages. The history
    of sent and received messages is shown [Chat_Client_4REQ]
    as well as all users connected to the same chat server as
    the chat client [Chat_Client_3REQ]. The user to send
    a message to can be selected using a mouse click in
    the list of online users [Chat_Client_3_1REQ].

    3) [fleshOutClientDD::fleshOutClientD::controllerCE]:
    an implementation is provided by [implementClientDF::controllerD] which
    handles the dataflow between [implementClientDF::communicatorD] and
    [implementClientDF::uiD].
#}
```

Figure 9.12: The `@description` comment-block of client Design Decision 3

### 9.4.2 Comment-block referencing

Comment-block referencing has been extensively used in the chat client, as can be seen in the `@description` block of client Design Decision 3, shown in figure 9.12. As well as Requirements, Requirement Categories and Stakeholders, which are system-wide referable, locally available objects are referable as well. For example, object `fleshOutClientDD`, which is passed to Design Decision 3 as a parameter (and represents the currently running system), is referred to many times. Also entities *within* `fleshOutClientDD` are referred to by using the `::` notation.

### 9.4.3 Graphical User Interface

A Graphical User Interface has been developed for the chat client to allow easy and convenient communication by users. It is shown in figure 9.13. The menu items allow for logging in and out on a chat server and for exiting the client application. The small right frame displays the users also online on the same server, and the large left frame displays received messages. The bottom frame allows for entering a message and subsequently sending it to the user selected in the list of online users.

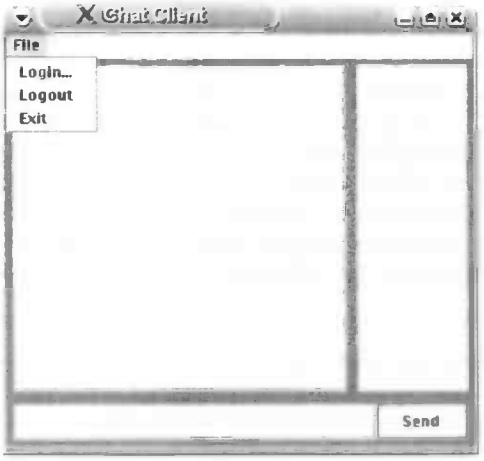


Figure 9.13: Graphical User Interface of the chat client

# Chapter 10

# Related work

The concepts used in Archium are based on well-established research in the field of software architectures [9] [21]. Its programming language resembles some characteristics from Architectural Description Languages (ADLs) [22], however a major difference is the fact that Archium supports the expression of Design Decisions and rationale in its language, which is often separated from or loosely coupled to "traditional" ADLs.

The Archium ADL is a component-based [7] [1], aspect-oriented [13] language. It allows for cross-cutting concerns, i.e. concerns that influence multiple parts of a software architecture, to be specified in a single Design Decision. Aspects are integrated within the component-oriented programming concepts requiring no separate aspect entity (Prisma [19]) nor a separate language (AspectJ [12]) for implementing cross-cutting concerns. In this sense Archium is more similar to FuseJ [8]. What sets Archium apart from other component-based, aspect-oriented ADLs is the fact that rationale is an integral part of each aspect and that each aspect is traceable to a Design Decision.

The requirements model incorporated in Archium uses many concepts also found in other popular requirements models, processes and templates, e.g. Volere [16] and UPEDU [24], however Archium is unique in relating Requirements to Design Decisions, and have them available at run-time for inspection and traceability purposes. Other models are also not integrated with a programming language, something which is a key feature of Archium.

Referring to objects from within comments/rationale has not been a topic of much research, however related projects include Javadoc [18] and Doxygen [10], which are used to generate documents based on comments encountered in source code. In contrary to this compile-time-only process, Archium is able to display such comments at run-time. Furthermore, Archium explicitly enables referring to architectural entities declared by the user in source code. These references are resolved at run-time as an architecture may well change an architecture is a dynamic system and thus references usually cannot be resolved at compile-time.

Dynamic reconfigurable component-based languages and systems (of which Archium is also a member) have been extensively researched, e.g. [26] [6], however they often do not pay much attention to the notion of threading in relation with dynamic reconfiguration. Most often applications considered are single-threaded and thread (un)suspension is not really an issue in such applications. Archium supports the development of multi-threaded applications and features a sophisticated threading model.

Much work has already been done on visualization of software systems, however most approaches focus on static, compile-time visualization. An overview of existing application visualization approaches is given in [3], together with a fresh, open-source approach called SoftViz

which allows static as well as dynamic (i.e. at run-time) visualization of software systems. Some of the aspects of SoftViz may well be suitable for inclusion in the Archium visualizer.

# Chapter 11

# Conclusion

## 11.1 Summary

In the previous chapters the extensions made to Archium have been discussed in detail. This section relates each extension to a research sub-question (as defined in section 3) and answers the question whether the problems forming the basis for each sub-question have been solved. By answering each research subquestion, the main research question for this thesis ("How to make the Archium compiler more mature?") is answered.

The first problem was the lack of a requirements model. Its associated research question was: "How can a requirements model be added to Archium?". Chapter 5 shows the development of a requirements model, integrated with the Archium language, which allows for specification of Requirements, Requirement Categories and Stakeholders. Furthermore, it allows developers to relate Requirements to Design Decisions, and have them available at run-time. This allows for the run-time inspection of the impact (in terms of fulfillment of Requirements) each Design Decision has, for example in the Visualizer.

The second problem involved the absence of an algorithm for referencing and resolving entities from rationale. The research question belonging to this problem was: "How can entities be resolved from rationale in Archium?". The algorithm has been developed, as shown in chapter 6. Each referred entity is now resolved at run-time, allowing future use of these resolved entities to be displayed in, for example, a tool for inspecting objects at run-time.

The third problem was the lack of a threading model, and its research question was "How can a managed threading model be added to Archium?". This question has been answered in chapter 7, which shows the analysis and development of a threading model of which the control lies in the hands of Archium, instead of in the hands of the developer, as was previously the case. The new threading model uses the state of Archium entities to determine the next course of action and the developer should have little problems migrating from standard Java threading to managed Archium threading, as implementation of these is very similar to the classical thread implementation process.

The fourth and last problem arised from the fact that there existed no facility for the runtime application of Design Decisions in Archium. The associated research question was: "How can run-time application of Design Decisions be achieved in Archium?". This facility now is present (as described in chapter 8.3) and integrated with the Archimon Visualizer. Unfortunately, a problem arised with the compile-time resolving algorithm, which effectively prohibits the use of the facility for the runtime application of Design Decisions in Archium in most cases.

In addition to providing solutions for the problems, the Archimon Visualizer has been extended with a number of views allowing for run-time inspection of a system, in particular its threads and Deltas. This is described in chapter 8.

## 11.2 Reflection

By providing solutions for all problems described in chapter 3, a solution for the main problem "How to make the Archium compiler more mature?" has been provided. The compiler certainly has more potential as essential parts have been added. Whereas previously only relatively simple applications could be developed using Archium, it is now possible to express larger and more complex projects in Archium, of which the chat case is an example.

Of course, the compiler remains a prototype and is still experimental, but the fundamental concepts are here and ready for use. There are however, additional extensions necessary to further mature the compiler. These are described in section 11.3.

Working on a prototypical, experimental compiler which is very much a work in progress is a challenging task, as the requirements change often due to the incorporation of new ideas which originate from testing and validation sessions.

Developing the threading model has been the most challenging problem solved during the graduation project. Particularly the analysis of the Archium entity state models, testing the developed models on theoretical examples and then refining them has taken more time than expected. However, threading is a complicated issue in all contexts and especially in combination with another complex matter: Archium.

Archium requires one to think of the development of a software architecture in a very different way than the one being taught from textbooks or in class. Especially the explicit change element, the Delta, requires some time to getting used to. But once the idea is clear, it becomes a very elegant way of describing the evolution of a software architecture. It took me about two months to understand the major concepts of Archium.

As for the future of Archium, I think it may take some time to convince architects to explicitly include rationale in the Design Decisions they make. Writing down explanations for each Design Decision and maintaining them requires some discipline. However, in the long run, architects will benefit from this extra work, as no longer guesses have to be made about why an architecture is as it is. The rationale explaining the architecture is available at the time when it is needed.

## 11.3 Future work

This section provides some pointers to additional work needed to further mature the Archium compiler, which was identified when implementing the solutions of the problems described in chapter 3.

### 11.3.1 Compile-time entity resolver

During implementation of the facility for the run-time application of Design Decisions (as shown in section 8.3), a problem was encountered whereby entities referred to at compile-time in Archium code would generate incorrect code.

In some blocks (e.g. `initialization{}` inside a Design Fragment Composition), the resolving mechanism for entities (e.g. `targetDD::clientCE::clientUICE`) does not generate the

necessary casts for each referenced entity (`targetDD`, `clientCE` and `clientUICE` in the example). Instead each element is cast to `Object`, which results in the method `getEntityObject` not being found for such an element, which is required for resolving at compile-time.

### 11.3.2 Requirements model

At the time of the graduation project the exact use of the requirements model was not yet determined. Therefore Archium includes an initial implementation containing the functionality described in the previous sections. It is possible that this functionality has to be extended, however the basic framework is there and all attribute value setting and linking of entities is implemented and working. In this section some possible extensions and uses for the requirements model are described.

#### Adding exceptions

It is thinkable that the use of the `creates`, `realizes`, `uses` and `threatens` keywords can raise exceptions. For example, should a design decision be able to realize a certain requirement if it has not been created first, or should this raise an exception? The same goes for a design decision that uses some requirement before it has been (fully) realized. A clear sequence of usage of these keywords can be developed, raising exceptions if the sequence is violated.

#### Requirements visualization

There already exists a visualization plug-in which can display architectural elements in a running Archium application. This visualizer could be extended to also display dependencies between design decisions/design solutions and requirements. Also the dependencies between requirements, requirement categories and stakeholders could be visualized.

The reflection functionality for requirements, stakeholders and requirement categories already contains (empty) interfaces for RMI so only the appropriate methods have to be defined and implemented on the side of the Archium run-time platform. Modifying the visualizer might take more time to complete.

#### Adding scenarios and actors

Although present in the existing model (figure 5.1), the Scenario and Actor entities didn't make it into the resulting model (figure 5.8). If agreement can be reached about the attributes and relationships to include, scenarios and actors could become a welcome extension to the requirements model.

### 11.3.3 Replace ArchJava with a custom implementation

Connectors form an important part of any software architecture built in Archium. In addition to adding them between Component Entities, it should be possible to remove them using a Delta. This is currently not possible, due to limitations of the intermediate ArchJava compile step, as explained in section 2.3.1. This is a limitation that needs to be resolved as larger projects (not necessarily the chat case) may well need it. ArchJava is an intermediate compile process step which has been part of Archium since its initial stages of development. As it provides functionality needed by Archium, unnecessary code duplication was avoided by incorporating ArchJava in Archium. However, its limitations are now beginning to show and it would be best to replace it by an implementation specifically tailored to Archium.

### 11.3.4 Assigning Deltas to Component Entities

During initial implementation of the chat client example in Archium the following issue arised. Assume for the chat case that two client Component Entities are needed as well as a server Component Entity. The two clients need exactly the same functionality (provided by a Delta). To define the first client, we would write:

```
...
component entity Client;
...
```

Followed by:

```
...
client = new Client();
...
clientDelta = new ClientDelta();
...
client plays clientDelta;
...
```

For the second client, we would need to write equivalent statements:

```
...
client2 = new Client();
...
clientDelta2 = new ClientDelta();
...
client2 plays clientDelta2;
...
```

As the two clients require exactly the same Delta, at present we would need to duplicate all `plays`-statements for the two client instances (composition is expressed by means of the `plays`-statement, e.g. `serverCE plays serverD`). Therefore the idea is to allow assignment of Deltas to Component Entities as well as to Component Entity *instances*. Both elements will then keep a list of Deltas they are playing. Component Entities will keep a list of *general* Deltas that are common to all Component Entity instances, and *specific* Deltas can be applied to individual Component Entity instances. This mechanism is comparable to the `extends`-mechanism in Java: a Component Entity is equivalent to a base class with certain functionality (provided by Deltas) while a Component Entity *instance* extends the base class by providing some additional functionality not necessarily also provided by other instances.

In the above example, this translates to, for example:

```
...
Client plays ClientDelta;
...
client = new Client();  // Inheriting all Deltas composed with Client
client2 = new Client(); // Inheriting all Deltas composed with Client
...
```

Both `clients` now inherit the functionality provided by `ClientDelta`. Specialized Deltas can now be applied to `client` and `client2`, for example:

```
...
client plays firstClientSpecializedDelta;
client2 plays secondClientSpecializedDelta;
...
```

If a facility for composing Deltas with Component Entities as well as with Component Entity *instances* is to be constructed in Archium, no doubt a lot of additional issues will arise. To mention a few:

- How to merge the general and specific Deltas into one list of Deltas?
- When should thread Deltas be run? Probably not at the time of assignment to a Component Entity. This issue is related to that described in section 3.1.3.

### 11.3.5 Extending the application of Deltas

Related to the issue described in section 11.3.3 is the fact that Deltas can currently only be applied to Component Entity instances. It is necessary to extend the application of Deltas to other architectural entities like Connectors in order to further develop the case.

### 11.3.6 Java files cannot be used in recursive-compile mode

The Archium compiler supports a compile flag to recursively compile all files in a directory. As mixing Archium code with Java code is possible in certain constructs, Java files should be compiled together with Archium files. However, the Archium parser parses each file as if it is an Archium source file, with Archium-specific keywords and constructs. This necessitates placing the Java code outside the directory to be compiled, which is not very convenient.

### 11.3.7 The Archium entity namespace is too global

All names of entities existing in a running Archium system are currently stored in a single namespace, which becomes problematic when an entity name is reused. Introducing the concept of modules (like `packages` in Java) could solve this.

# Bibliography


[1] Andy Ju An Wang and Kai Qian. *Component-Oriented Programming*. Wiley, 2005.

[2] Anton G.J. Jansen and Jan Bosch. Software Architecture as a Set of Architectural Design Decisions. In *Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005)*, November 2005.

[3] Benjamin Kurtz. Softviz: A Runtime Software Visualization Environment. *Graduation thesis*, 2003.

[4] Jan Bosch. Superimposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5):257–273, 25 March 1999.

[5] Wouter-Tim Burgler and Marnix Kok. Improving Archium. Taking Archium's code generation to the next level. *Graduation thesis*, January 2006.

[6] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos Zarras Irisa. A Dynamic Reconfiguration Service for CORBA. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems (CDS 1998)*, May 1998.

[7] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997.

[8] Davy Suvée, Bruno De Fraine, and Wim Vanderperren. A Symmetric and Unified Approach Towards Combining Aspect-Oriented and Component-Based Software Development. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006)*, June 2006.

[9] Dewayne Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[10] Doxygen website. http://www.stack.nl/~dimitri/doxygen.

[11] Eclipse website. http://www.eclipse.org.

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, June 2001.

[13] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, June 1997.

[14] Zef Hemel. *Life-cycle Model of Archium*. Part of the Archium source package.

[15] Zef Hemel and Anton G.J. Jansen. *Requirements Model for Archium*. Part of the Archium source package.

[16] James Robertson and Suzanne Robertson. *Mastering the Requirements Process*. Addison-Wesley, 2nd edition, 2006.

[17] JavaCC website. https://javacc.dev.java.net.

[18] Javadoc website. http://java.sun.com/j2se/javadoc.

[19] Jennifer Pérez, Nour Ali, Jose A. Carsí, and Isidro Ramos. Designing Software Architectures with an Aspect-Oriented Architecture Description Language. In *Proceedings of the 9th International Symposium on Component-Based Software Engineering (CBSE 2006)*, June 2006.

[20] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, May 2002.

[21] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*. Prentice-Hall, 1996.

[22] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[23] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures, Views and Beyond*. Addison-Wesley, 2002.

[24] Pierre N. Robillard and Philippe Kruchten. *Software Engineering Processes: With the UPEDU*. Addison-Wesley, 2002.

[25] Software Engineering & Architecture website. http://search.cs.rug.nl.

[26] Thais Batista, Ackbar Joolia, and Geoff Coulson. Managing Dynamic Reconfiguration in Component-based Systems. In *Proceedings of the European Workshop on Software Architectures*, June 2005.

# Definition of terms

| | |
|---|---|
| **Component Entity** | Logical architectural building blocks which represent arbitrarily sized functional software components. Functionality is added to Component Entities by means of composing Deltas with them. Therefore, Component Entities initially contain no functionality. |
| **Connector** | Representation of interaction between Component Entities. Component Entities are only allowed to communicate with each other if they have been connected using a Connector. Dedicated endpoints can be defined on Component Entities (called Ports) which allow the attachment of Connectors to it. |
| **Delta** | Representation of a change to a Component Entity. Deltas, when composed with a Component Entity, may establish various changes, for example performing detailed design, adding or removing a Connector, or adding a (threaded) implementation. |
| **Delta stack** | The structure consisting of all Deltas composed with a certain Component Entity. As conceptually each Delta forms a new layer around the Component Entity, the term "stack" is used. Deltas within a Delta stack may communicate with each other, i.e. may make use of the changes each Delta has made to the Component Entity. |
| **Design Decision** | A set of changes to be applied to an existing architecture, as well as the rationale for the changes. The rationale includes a description of the problem the Design Decision tries to solve, the problem context and a motivation for solving the problem. Furthermore, a number of alternative solutions (called Design Solutions) are evaluated. When applied, a Design Decision, taking as input a Design Fragment representing the architecture to be modified, results in the original Design Fragment modified accordingly to the changes specified in the Design Decision. |
| **Design Fragment** | A collection of architectural entities (for example, Connectors and Component Entities), representing (part of) a software architecture. |
| **Design Fragment Composition** | Defines the mapping of changes specified in a Design Decision to the entities in a Design Fragment. |
| **Design Solution** | A solution to the problems described in a Design Decision. As often many alternative solutions are considered in a Design |

| | |
|---|---|
| | Decision, each Design Solution includes rationale explaining its associated pros, cons, design constraints and design rules, allowing for a well-argumented choice of solution. |
| **Interface** | A specification of method signatures, identifying the methods allowed to call on a certain Port. |
| **Managed Thread** | A thread which allows itself to be managed (i.e. to be started, stopped, suspended and unsuspended) by an external entity called a Managed Thread Pool. It has the same expressive power as standard Java threads have. |
| **Managed Thread Pool** | An entity for managing a group of Managed Threads which register themselves with the Managed Thread Pool. Each Delta has an associated Managed Thread Pool, allowing for the management of all threads defined inside the Delta. |
| **Port** | A dedicated endpoint which can be defined on Component Entities to allow for the attachment of Connectors. Ports can be subdivided into two types: required and provided Ports. Required Ports specify that functionality is required from another Component Entity to allow the Component Entity on which the required Port has been defined to operate correctly, and vice versa. Ports also explicitly define the way in which communication is allowed through any Connectors connected to them by implementing an Interface. |

# Appendix A

# Requirements model

## A.1 Added tokens

This section shows the tokens which were added to the Archium parser in order to parse Requirement entities, as discussed in section 5.3.1.

```
<ATDESCRIPTION> ::= "@description"
<ATNAME>        ::= "@name"
<CATEGORY>      ::= "category"
<COULDHAVE>     ::= "CouldHave"
<DEPENDS_ON>    ::= "depends on"
<DESIRED_BY>    ::= "desired by"
<FUNCTIONAL>    ::= "Functional"
<MUSTHAVE>      ::= "MustHave"
<NONFUNCTIONAL> ::= "NonFunctional"
<PRIORITY>      ::= "priority"
<REFINES>       ::= "refines"
<REQUIREMENT>   ::= "requirement"
<SHOULDHAVE>    ::= "ShouldHave"
<STATUS>        ::= "status"
<TYPE>          ::= "type"
```

## A.2 Added production rule for Requirement entities

This section shows the production rule which was added to the Archium parser in order to parse Requirement entities, as discussed in section 5.3.1.

```
<REQ_PROD> ::=
  <REQUIREMENT> <IDENTIFIER> <LBRACE>
  [ <ATNAME <COMMENTBLOCK> ]
  [ <ATDESCRIPTION <COMMENTBLOCK> ]
  <PRIORITY> (<COULDHAVE> | <SHOULDHAVE> | <MUSTHAVE>) <SEMICOLON>
  <STATUS> <PERCENTAGE> <SEMICOLON>
  <TYPE> (<FUNCTIONAL> | <NONFUNCTIONAL>) <SEMICOLON>
  [ <DEPENDS_ON> <IDENTIFIER> {<COMMA> <IDENTIFIER>} <SEMICOLON> ]
  [ <REFINES> <IDENTIFIER> {<COMMA> <IDENTIFIER>} <SEMICOLON> ]
  [ <CATEGORY> <IDENTIFIER> {<COMMA> <IDENTIFIER>} <SEMICOLON> ]
  [ <DESIRED_BY> <IDENTIFIER> {<COMMA> <IDENTIFIER>} <SEMICOLON> ]
  <RBRACE>
```

## A.3 Requirements reflect class diagram

Figure A.1 shows the reflection API class diagram for the Requirement entity (as discussed in section 5.3.1). Each attribute and relationship in figure 5.8 has its own pair of getter/setter methods. Exceptions are the private variables of type `Comment`. They only have a getter method, as the `Comment` class provides the necessary getter and setter methods. The `Comment` class has been explained in chapter 6. The one method whose functionality might not be immediately clear is the `isActive()` method. It indicates whether the Requirement has any origins (Design Decisions which create this requirement).


Figure A.1: The Requirement reflect class

# Appendix B

# Comment-referencing from rationale

## B.1 `Comment` class diagram

Figure B.1 shows a diagram of the `Comment` class, as discussed in section 6.3.

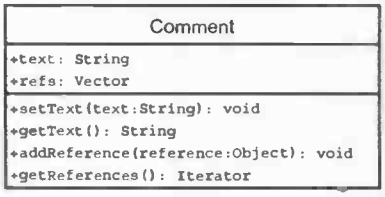


Figure B.1: The Comment class