university of groningen

University Medical Center Groningen

University of Groningen

Architectural design decisions

Jansen, Antonius Gradus Johannes

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2008

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Jansen, A. G. J. (2008). *Architectural design decisions*. s.n.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 06-12-2024

CHAPTER 4

SOFTWARE ARCHITECTURE AS A SET OF ARCHITECTURAL DESIGN DECISIONS

Published as: Anton Jansen, Jan Bosch. Software Architecture as a Set of Architectural Design Decisions, Proceedings of the 5th IEEE/IFIP Working Conference on Software Architecture (WICSA 2005), pp. 109-119, November 2005.

Abstract

Software architectures have high costs for change, are complex, and erode during evolution. We believe these problems are partially due to knowledge vaporization. Currently, almost all the knowledge and information about the design decisions the architecture is based on are implicitly embedded in the architecture, but lack a first-class representation. Consequently, knowledge about these design decisions disappears into the architecture, which leads to the aforementioned problems. In this paper, a new perspective on software architecture is presented, which views software architecture as a composition of a set of explicit design decisions. This perspective makes architectural design decisions an explicit part of a software architecture. Consequently, knowledge vaporization is reduced, thereby alleviating some of the fundamental problems of software architecture.

4.1 Introduction

Software architecture [119] has become a generally accepted concept in research and industry. The importance of stressing the components and their connectors of

a software system is generally recognized and has led to better control over the design, development, and evolution of large and increasingly dynamic software systems [11].

Although the achievements of software architecture are formidable, still some problems remain. The complexity, high costs of change, and design erosion are some of the fundamental problems of software architecture. We believe these problems are partially due to knowledge vaporization. Currently, almost all the knowledge and information regarding the design decisions on which the architecture is based (e.g. results of domain analysis, architectural styles used, trade-offs made etc.) are implicitly embedded in the architecture, but lack a first class representation.

The current perspective on software architecture lacks this notion of architectural design decisions, although architectural design decisions play a crucial role in software architecture, e.g. during design, development, evolution, reuse and integration of software architectures. In design, the main concern is which design decision to make. In development, it is important to know which and why certain design decisions have been taken. Architecture evolution is about making new design decisions or removing obsolete ones to satisfy changing requirements. The challenge is to do this in harmony with the existing design decisions. Reuse of software architecture is the use of earlier tried and tested combinations of design decisions (e.g. design patterns or components). In the integration of systems, the main concern is the unification of the design decisions and their assumptions.

To address this, we propose a new perspective on software architecture: we define software architecture as the composition of a set of architectural design decisions. This reduces the knowledge vaporization of design decision information, since design decisions have become an explicit part of the architecture.

The contribution of this paper is threefold. First, the problems with the current perspective on software architecture are presented. Second, it develops the notion of software architecture as the composition of a set of explicit architectural design decisions. Third, various views are presented for visualizing this new architecture perspective.

The remainder of this paper is organized as follows. The concept of architectural design decisions is presented in section 4.2. In section 4.3, the problems of software architecture with respect to architectural design decisions are explained in more depth. The next section introduces Archium: our approach for describing software architecture as a set of architectural design decisions. The approach is applied to a case and illustrated with various views on design decisions in section 4.6. After this, related work is discussed. The paper concludes with future work and conclusions in section 4.8.

4.2 Architectural design decisions

Although the term “architectural design decision” is often used [11, 29, 67], a precise definition is hard to find. Therefore, we define an architectural design decision as:

> A description of the set of architectural additions, subtractions and modifications to the software architecture, the rationale, and the design rules, design constraints and additional requirements that (partially) realize one or more requirements on a given architecture.

With the definition of architectural design decisions using the following elements:

- **Rationale** The reasons behind an architectural design decision are the rationale of an architectural design decision. It describes *why* a change is made to the software architecture.
- **Design rules** and **design constraints** are prescriptions for further design decisions. Rules are mandatory guidelines, whereas constraints limit the design to remain sound.
- **Design constraints** Design constraints describe the opposite side of design rules. They describe what is not allowed in the future of the design, i.e. they prohibit certain behaviors.
- **Additional requirements** A design decision may result in additional requirements to be satisfied by the architecture. These new requirements need to be addressed by additional design decisions.

An architectural design decision is therefore the outcome of a design process during the initial construction or the evolution of a software system. Architectural design decisions, among others, may be concerned with the application domain of the system, the architectural styles and patterns used in the system, COTS components and other infrastructure selections as well as other aspects needed to satisfy the system requirements.

We propose to view a software architecture as a set of explicit architectural design decisions. In this perspective, the software architecture is the *result* of the architectural design decisions made over time.

4.3 Problems of software architecture

The current perspective on software architecture lacks a clear view on why the architecture looks as it does [21, 93]. In the current notion of a software architecture, the results of the design decisions underlying the architecture are implicitly embedded within the architecture. Consequently, knowledge about the design decisions underlying the architecture is lost [158]. This vaporization of design decision information leads to a number of problems associated with software architecture:

- **Design decisions are cross cutting and intertwined:** Design decisions are often intertwined with each other, as they work in close relationship together. Furthermore, they typically affect multiple parts of the design simultaneously. This leads to the situation that the design decision information is fragmented across various parts of the design, making it hard to find and change the decisions. Both effects increase the overall complexity of the software architecture, as numerous seemingly unrelated relationships (e.g. dependencies) between architectural entities are introduced.
- **Design rules and constraints are violated:** During the evolution of the system, designers can easily violate design rules and constraints arising from previously taken design decisions. Violations of these rules and constraints lead to architectural drift [119] and its associated problems (e.g. increased maintenance costs). As design rules and constraints influence future design decisions, they have a steering influence on the future direction of the architecture.
- **Obsolete design decisions are not removed:** When obsolete design decisions are not removed, the system has the tendency to erode more rapidly. In the current design practice, removing design decisions is avoided, because of the effort needed, and the unexpected effects this removing can have on the system.

As a result of these problems, the developed systems have a high cost of change, and they tend to erode quickly. Also, the reusability of the architectural artifacts is limited if design decision knowledge vaporizes into the design. These problems are caused by the focus in the software architecture design process being on the resulting artifacts, instead of the decisions that lead to them. Although the effects of the decisions made are present in the design, the decisions themselves are not visible. Clearly, design decisions currently lack a representation in software architecture designs.

Defining software architecture as a set of architectural design decisions is a step forward in solving the aforementioned problems. This would also help the architect with:

- **Guarding the conceptual integrity** of the software architecture. The design decisions describe the rules and constraints that should be obeyed. In current practice, software engineers and architects are often unaware of the conceptual integrity that they break of the architecture. Explicit design decisions help in creating the necessary awareness and reference points for these constraints and rules.
- **Explicit design space exploration** helps the architect in preventing obvious mistakes. It forces the architect to reflect on the software architecture. Furthermore, it enables communication of the explored design space with others.
- **Analysis** of both the software architecture and the design process. For example, in evolution the architect wants to play "what if" scenarios of considered design decisions in the context of existing ones.
- **Improved traceability** of the design decisions and their relationship to features, design aspects, concerns, and among themselves. This helps the architect with obtaining a better understanding of the software architecture.

However, the following requirements need to be satisfied to realize this:

- **First class architectural design decisions** are required to describe a software architecture as a set of design decisions. Furthermore, first class design decisions can be communicated, related and reasoned about. This provides information about the architecture, which is currently often missed.
- **Explicit architectural changes** form the bridge between the first class architectural entities and the architectural design decisions. This is needed to have a well-defined relationship between the proposed solutions of an architectural decision and the involved architectural entities.
- **Support for modification, subtraction, and addition** changes are required to have sufficient expressiveness. The characteristic types of change often distinguished are the corrective, perfective, and adaptive types. However, the focus of this classification is on the reasons behind the change, not on the effect of the changes.
- **Clear, bilateral relationship between architecture and realization** Viewing a software architecture as a set of design decisions, makes evolution an inherent part of the description of an architecture. Changes in the architecture will have an effect on the realization of the system and vice versa. It is therefore important to have a bilateral relationship between the software architecture and the realization.
- **First class architectural concepts** As software architecture deals with abstractions, it is important to define these abstractions in a first class way. Abstraction choices are very subjective and greatly influence the resulting architecture.

Expressing these abstractions in a consistent and uniform way is therefore essential for software architectures.

4.4 Archium

The aforementioned problems of section 4.3 clearly show that the notion of an architectural design decision is an important one. Currently, no models for representing architectural design decisions exist [76]. Some general design decision models [128] do facilitate the description of abstract elements of an architectural design decision model, but these approaches fail to satisfy most of our requirements [76].

This is mainly due to the ill-defined relationship between these design decision models and software architectures. Therefore, we have developed an approach called Archium, which tries to define this relationship. Archium maintains this relationship during the complete life-cycle of the system. In this paper, the focus is on the software architecture aspects of Archium. The approach is based on a conceptual architectural design decision model, which describes the elements of architectural design decisions and their relationships in greater detail than the abstract design decision models. In the remainder of this section, this conceptual model is presented.

4.4.1 Architectural design decision model

Figure 4.1 presents our conceptual model for architectural design decisions. At the heart of the model is the *Problem* element, which together with the *Motivation* and *Cause* elements describes the problem, a *Motivation* as to why the problem is a problem, and the *Cause*s of this problem. The *Problem* is the goal the architectural design decision wants to solve. The solutions element contains the *Solution*s that have been proposed to solve the problem at hand. A *Decision* is made, which solution should be used, resulting in an *Architectural modification* that changes the *Context*.

To solve the described *Problem*, one or more potential *Solutions* can be thought up and proposed. For each of the proposed solutions, we define the following elements (which are not shown in the figure 4.1) :

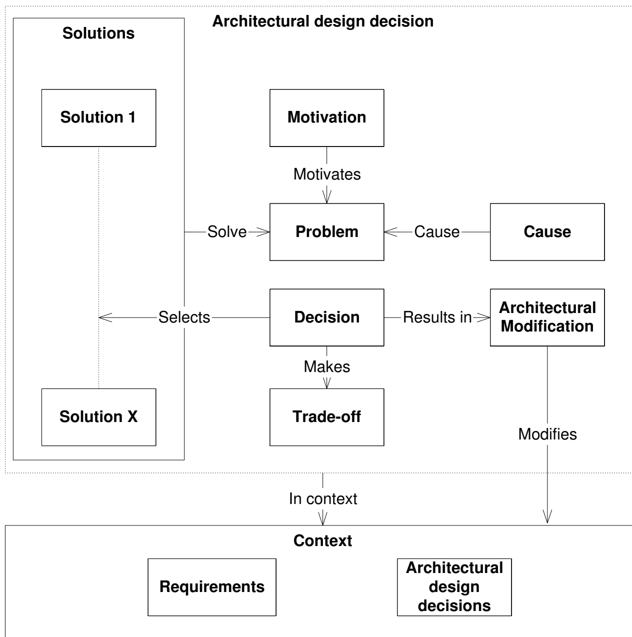

Figure 4.1: Model for architectural design decisions

- **Description** The description element describes the solution being proposed. The needed modifications are explained and rationale for these modifications is provided.
- **Design rules** A potential solution can have one or more design rules. Design rules define partial specifications to which the realization of one or more architectural entities has to conform. This allows a solution to define parts of how it should be realized in order to have a solution that solves the problem.
- **Design constraints** Besides design rules, a solution can have design constraints. Design constraints define limitations on the further design of one or more architectural entities. These constraints have to be obeyed for the potential solution to solve the problem at hand.
- **Consequences** The consequences element is a description of the expected consequences of the solution on the architecture. The element should provide additional rationale behind the pros and cons of the solution.
- **Pros** This model element describes the expected benefit(s) from this solution to the overall design and the impact on the requirements.
- **Cons** Solutions can also have a downside. The expected negative impact on the overall design is as important as the positive side.

Translating the conceptual model into concrete model(s) is a big challenge. Our earlier investigation [76] revealed a gap between design decisions and software architecture models. Therefore, the rest of this paper focuses on the interaction between architectural design decisions and software architecture. Specifically, we discuss how the *decision*, *solution*, *architectural modification*, *software architecture*, and *architectural design decisions* conceptual model elements are modeled and formalized to describe a software architecture as a set of design decisions.

4.5 Archium meta-model

In Archium, the functionality of the architectural modification is expressed as a *change* in functionality. New functionality is regarded as the change of nothing to something. In this perspective, Archium is fundamentally different to most other design methods, as it does not promote design for or with change, but rather designing *using* change.

A software architecture in Archium is described as a set of changes, which together form the software architecture. To be more precise, in Archium a $software\ archictecture = dd_1 + dd_2 + ... + dd_n$, where dd_x is a design decision. The exact elements required to achieve this are described in the rest of this section.

The Archium approach is based on a meta-model, which describes the central concepts of our approach and their relationships. Figure 4.2 presents this meta-model. The model consists of three sub-models: an architectural model, a design decision model, and a composition model. The architectural model defines software architecture concepts, which are similar to the concepts used in existing architecture models [107]. The design decision model contains design decisions as a first class concept. The composition model introduces the model elements needed to unite the two previous sub-models. Each sub-model is explained in more detail in the remaining part of this section.

A (trivial) running example of a subsystem of a measurement system exemplifies the different concepts. The measurement system acquires certain properties of a physical item that enters the system for measurement. The architecture of this system is visualized in the top of figure 4.4.

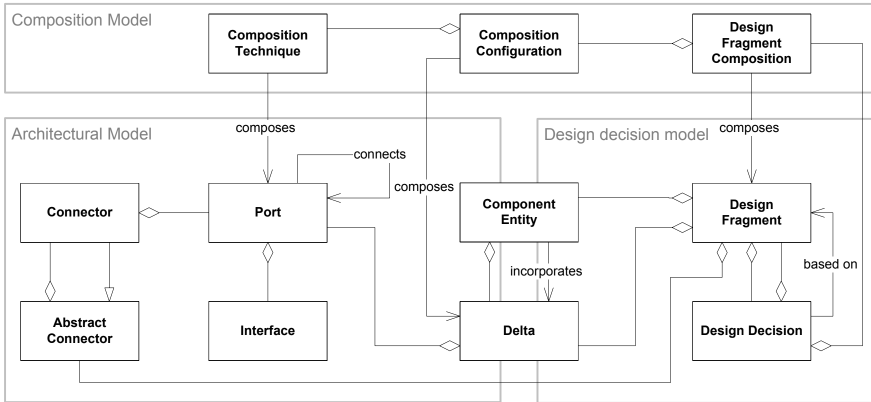

Figure 4.2: Meta-model of a software architecture with first class design decisions

4.5.1 Architectural Model

The architectural model part of the Archium meta-model uses concepts of the Component & Connector view [29]. The relationships of these concepts is visualized in figure 4.2. Following is a more in-depth description of these concepts and their relationships:

Component Entity is an abstraction of a component. A component entity describes the decomposition aspect of a component. The functionality of a component entity is not defined in the component entity itself, but in the deltas related to the component entity. A component in Archium is a specific *instance* of a component entity with known functionality, i.e. the deltas incorporated in the component entity instance are known.

For example, in the measurement system (see top of figure 4.4) a decomposition has been made in two parts, which are made up of the *Measurement Item* and the *Sensor* component entities. The *Measurement Item* component entity represents the object to be measured and the *Sensor* measures some properties of the measured object.

Delta is the first-class representation of a change to the behavior of a component entity, which is provided by the deltas already incorporated in the component entity. A component entity incorporating a delta includes the modification of the delta to its behavior. The merging of the behavior of different deltas is performed using the elements of the composition model (see section 4.5.3).

In the example, the functionality of the *Measurement Item* and *Sensor* components are defined in the *SensorDelta* and *MIDelta* deltas. The *SensorDelta* contains the

functionality to measure an item and the *MIDelta* has the functionality to store these measurements. These changes are not visible in figure 4.4, as they are being incorporated into the components.

Interface A definition of a collection of method signatures, representing a specific named semantic.

Port An external visible interface required or provided by a delta or connector. A port represents the provided or required "service" of a delta or connector. Deltas and connectors are only allowed to communicate with others through their defined ports. Ports of a delta and connector can be connected together, to form a connection, thereby creating a specific configuration of deltas and connectors.

In the measurement system (see figure 4.4), two ports are defined: a provided port for the *Sensor* and a required port for the *Measurement Item*. Neither port is defined in the component entities, but instead is part of the deltas incorporated in the components.

Connector defines the "glue" between one or more deltas, i.e. a connector is a first class representation of the interaction or communication between these deltas. The ports of a connector can be bound to the provided and required ports of deltas, thereby forming the "glue" between them. A connector therefore defines the specific functionality used for the communication between connected ports.

In the example, the communication between the *Sensor* and the *Measurement Item* is defined in the connector *CMISensor*.

Abstract Connector is an abstraction from a Connector, as it does not have an interface associated with it. It defines the communication type (i.e. synchronous, asynchronous) between two or more deltas. In addition, it defines a set of connectors that actually communicate between these deltas.

The abstract connector used in our example defines that the *SensorDelta* and *MIDelta* communicate using method invocation with each other. In addition, the abstract connector contains the connector *CMISensor* connecting the two deltas.

4.5.2 Design Decision Model

In this subsection, the design decision model part of Archium is presented. The relationship between architectural design decisions and the architectural concepts is defined using the concept of a design fragment. Following is a more in-depth description of both concepts:

Design Fragment is an architectural fragment defining a collection of architectural entities. An architectural entity can be part of multiple design fragments. A

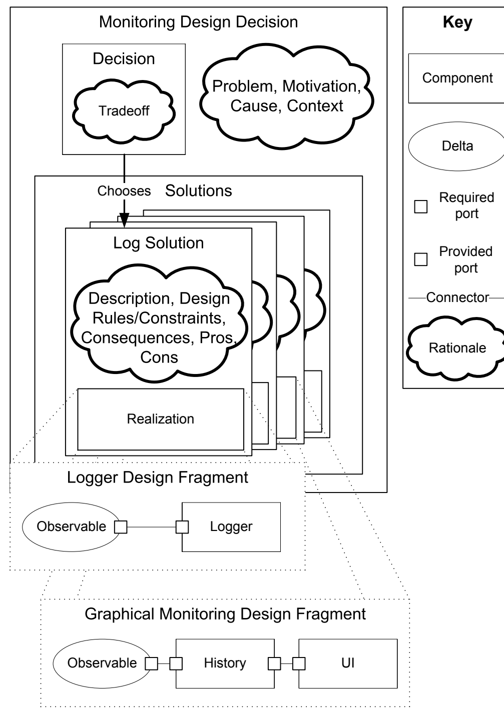

Figure 4.3: Example of a design decision

design fragment is a boundary-less container for maintaining traceability. The primary use is to define the scope of a solution of a design decision. For example, a design fragment can contain deltas and their configuration of (abstract) connectors, component entities incorporating certain deltas, and other design fragments, which describe a particular solution. A design fragment is therefore a (partial) description of the system, which can include explicit change (modeled as deltas) and structure, i.e. component entities and (abstract) connectors.

In Archium, the concept of a software architecture and a design pattern are specializations of the design fragment concept. The first class concept of a design fragment makes these two concepts elements in the Archium model. A software architecture is a design fragment describing the system as a whole. This description consists of the component entities and (abstract) connectors and their configuration, which is a specialized subset of the architectural entities a design fragment can contain.

For example, the architecture of the measurement system itself is a design fragment. Figure 4.4 visualizes this, with the architectural entities (*Measurement Item*, *CMISensor*, *Sensor*) being enclosed within the *SensorFragment*.

Design patterns [49] are often seen as predefined design decisions, which is not completely true. They define predefined *parts* of design solutions, which can be reused. However, design patterns still need to be instantiated and configured in a design decision to be of use in a specific architecture.

Design Decision is a first class concept in Archium. It defines the solutions considered and the one decided upon (i.e. the decision) to solve a described problem (see figure 4.3). A software architecture (i.e. a design fragment) describes the context in which this design decision is made. The considered potential solutions consist of one or more design fragments, which act upon this context design fragment by changing it according to the selected solution.

Figure 4.3 presents an example of a design decision. It consists of the rationale described in section 4.4.1, a decision element, and one or more solution elements. Each solution contains its own rationale and a realization part. The realization is a design fragment, which is mapped onto a design fragment representing the architecture. The mapping is explained in more detail in the next subsection.

A design decision is regarded as a change function within Archium. It has optional parameters, which are the design fragments describing the context that the design decision changes. Applying a design decision on these context design fragments results in a new design fragment, which includes the design decision it originated from. This explains the mutual relationship between a design fragment and design decision in the Archium meta-model (figure 4.2).

An example of the application of a design decision is provided in figure 4.4. In this case, the measurement system needs to be changed to allow monitoring of the activities within the system. The design decision is made to change the *SensorFragment* to include a logger. This design decision is presented in figure 4.3. The logger logs the actions of the *Measurement Item* on the *Sensor*. This modification is defined in the *LoggerFragment*, which is another design fragment. The composition of the design fragments, as a means to change the measurement system, is described in the next subsection.

4.5.3 Composition Model

The composition model is responsible for relating the changes of the design decision model to the elements of the architectural model. It defines the way in which a delta interacts with other composed deltas. In Archium, the following first class citizens are concerned with describing this:

- **Composition Technique** describes the way in which a delta changes a port of a component entity. For example, it can define that a delta introduces a new

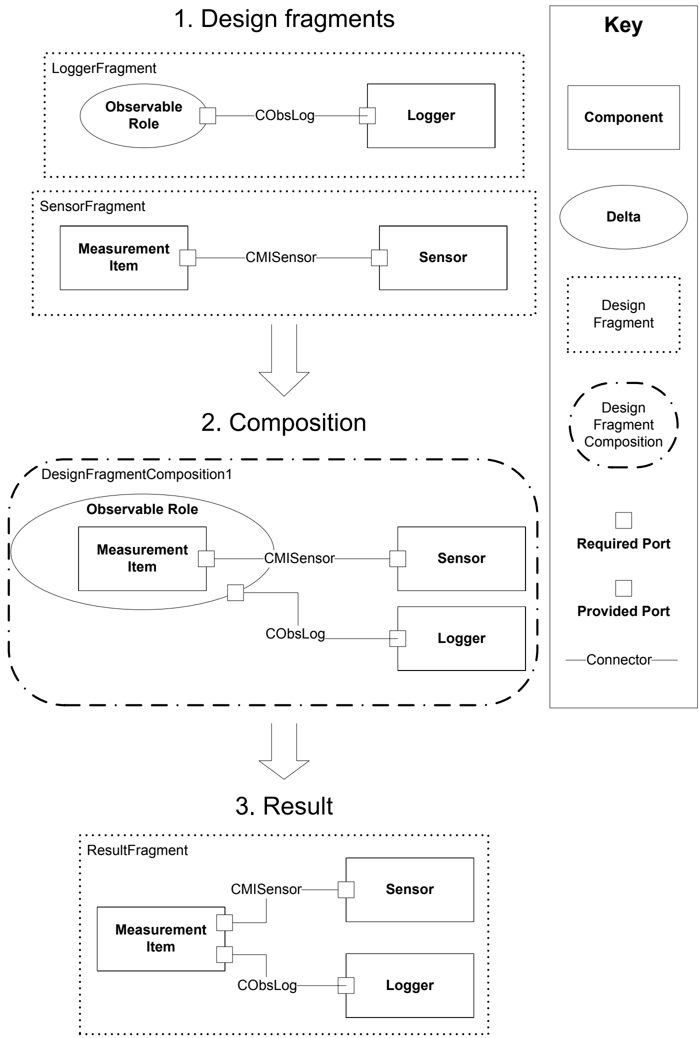

Figure 4.4: Example of the composition of two design fragments

port to the component entity, subtracts, or modifies an existing one. Examples of composition techniques include: Inheritance, Delegation, Replacement, and Adapter (adapt a port interface).

For example, in figure 4.4 a composition technique is used to describe how the provided port of the *ObservableDelta* reacts on the activities on the required port of *MeasurementItem*.

- **Composition Configuration** specifies how a component entity incorporates a certain delta. The composition configuration uses composition techniques to specify the change on a per-port basis. In this sense, the composition configuration is nothing more than a set of composition techniques to describe the way in which a delta changes a component entity.
- **Design Fragment Composition** is used to define how a design fragment can change another design fragment. It uses composition configurations and design fragments to relate architectural entities of one design fragment to another, thereby creating a new, changed design fragment.

  For the example of figure 4.4, the design fragment composition composes the *LoggerFragment* and *SensorFragment*. It uses a composition configuration to relate the *ObservableDelta* with the *MeasurementItem*.

4.6 Athena case

The previous section introduced the Archium meta-model and illustrated parts of it in a trivial example. In this section, we validate our approach by applying it to a case. First, the case is introduced, after which two design decisions are presented in more detail.

4.6.1 Introduction

Athena is a submission system for (automatic) judgement, review, manipulation, and archival of computer program sources. The primary use is supporting students in learning programming. To develop the programming skills of a student, he or she has to practice a lot. Small programming exercises are often used for this end. However, providing feedback on these exercises is laborious and time-consuming. Athena helps students (and teachers) by testing their solutions to functional correctness and provides feedback (e.g. test results, test inputs, compilation information etc.) on this.

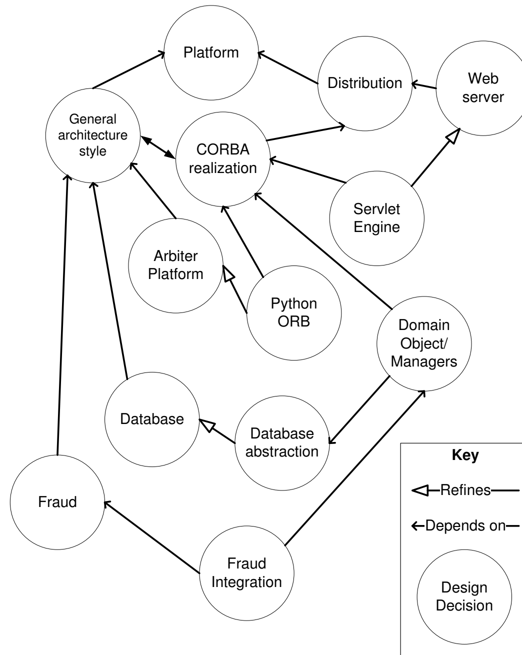

Figure 4.5: Design decisions of Athena

The architecture of Athena (called "Original design") is illustrated on the top of figure 4.6. Athena uses a three tier architectural style consisting of *Database*, *Middleware*, and *Client* components. The *Middleware* component consists of a *Connection Broker*, which provides database abstraction and connection handling. A *Domain Object* represents the different specific domain objects used in Athena (e.g. Student, Submission, Assignment etc.). A *Manager* provides query and instantiation services for the *Client* and *Domain Object* components. In the *Client* component, students submit their work with the help of the *Submission Client*. A *Arbiter* tests this work and students can view the results with the *Student Web Interface*. Teachers and their assistants configure Athena with the help of a *Management Tool*.

4.6.2 Design decisions

The software architecture of Athena is the result of multiple design decisions. Figure 4.5 presents a part of these design decisions in a design decision dependency view. An architect would like to navigate between this view and other views on the

architecture. The view allows for the management of the dependencies among design decisions. For example, "what if" scenarios can be played, where the impact of potential design decisions is examined.

The focus in the remainder of this section is on one dependency between two design decisions, as space constraints hinder a more elaborate description. Based on the different elements of Archium's design decision model (see section 4.4.1) the *Fraud* and *Fraud Integration* design decision are described. Both design decisions are made after the initial deployment of the system. They are described as follows:

Fraud design decision

Problem Some of the students don't create solutions for the exercises themselves, but rather copy the work of their fellow students. **Motivation** A result of this is that the students don't obtain an adequate programming experience, which is required for more advanced courses. **Cause** The large number of students (100+) in courses where Athena is used leads to anonymity and a small chance to get caught. On top of this, the high pressure resulting from the strict deadlines imposed by the system increases the temptation to commit fraud. **Context** The original design of the Athena system, as depicted on the top of figure 4.6.

Potential solutions

- ***Moss***

  Description Moss [134] is an anti-fraud system, which employs various code finger printing techniques to detect plagiarism. The Moss system uses a client-server architecture. The client consists of perl script, which communicates with the Moss Internet server over TCP/IP. The client provides the user with an URL pointing to the results of the analysis.
  Design rules For each assignment it should be clear whether it should be scanned for fraud or not.
  Design constraints Moss works in a batch oriented way; all the data to be tested for fraud should be delivered at the same time and increments are not possible.
  Consequences The Athena middleware becomes dependent on the Moss server.
  Pros +Good, confident fraud detection. +Can ignore base frameworks provided to students +Support multiple programming languages +Free to use
  Cons -Integration and archival of the analysis results can be difficult.

- ***JPlag***

  Description JPlag [103] is a plagiarism detection system similar to Moss. JPlag parses the submitted files and searches for similarities in their parse trees. The JPlag architecture uses a client-server architecture. The Java client sends the

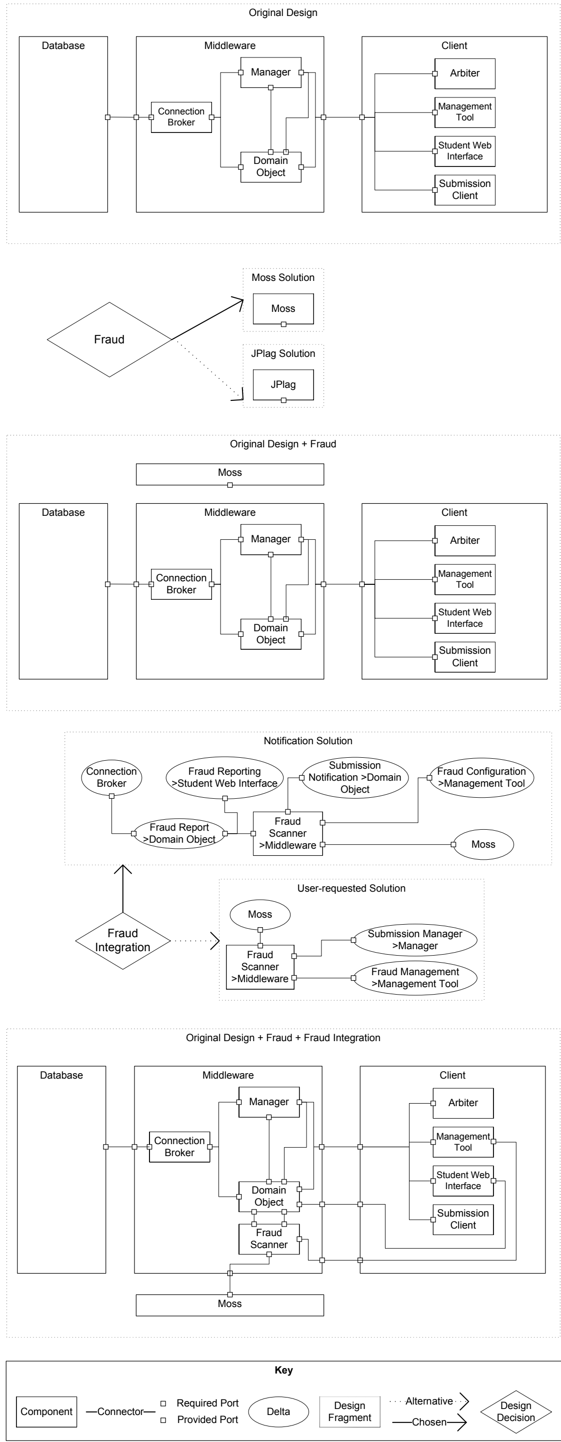

Figure 4.6: Two design decisions of Athena

files for scanning to the server. The results of the analysis can be viewed through a web interface.
Design rules For each assignment it should be clear whether it should be scanned for fraud or not.
Design constraints JPlag works in a batch oriented way, all the data to be tested for fraud should be delivered at the same time and increments are not possible.
Consequences The Athena middleware becomes dependent on the JPlag server.
Pros +Free to use
Cons -Supports a relatively small number of programming languages -No demo available

Decision The Moss solution is chosen, as it supports more programming languages and can ignore base frameworks provided to the students. **Architectural modification** The architectural modification of this design decision is depicted in figure 4.6.

Fraud integration design decision
Problem The Moss Internet server should be integrated with the Athena system.
Motivation The Athena users should use the anti-fraud functionality in a transparent way. **Cause** The need for a fraud system, as described in *Fraud* design decision.
Context The design of the Athena system, as depicted in figure 4.6 under the title "Original Design + Fraud".
Potential solutions

- ***Notification***

  Description The *Fraud Scanner* with the help of the *Fraud Configuration* and the *Moss* server keeps the *Fraud Report* for an assignment up-to-date. The *Fraud Reporting* uses the *Fraud Report* to inform the users.
  Design rules The Domain Object responsible for the processing of the student submissions should notify the *Fraud Scanner*, when a new submission for an assignment is made.
  Design constraints The availability of the Moss server may not interfere with the submission process.
  Consequences Every submission by a student leads to a new *Fraud Report*.
  Pros +The data for *Fraud Reporting* is instantly available +Allows for immediate feedback on the detection of fraud
  Cons -Heavy load induced on the Moss server

- ***User-requested***

  Description The user initiates a fraud analysis. The *Fraud Scanner* delivers the analysis using the *Moss* server.

Design rules The *Submission Manager* should provide the student submissions for a fraud analysis for the *Fraud Scanner*.
Design constraints Fraud analysis should be only performed when a user requests for this information in the *Management Tool*.
Consequences The result of the fraud analysis is not stored in the Athena system, but by *Moss*.
Pros +Relatively easy to develop +Light load induced on the Moss server
Cons -Automatic fraud feedback to students is not available.

Decision The decision is made to use the Notification solution, which provides a more active feedback from the system to the users. **Architectural modification** The architectural modification is presented in figure 4.6.

Figure 4.6 presents a view on these two design decisions, which visualizes part of the history of the Athena architecture with the help of design decisions. The top displays the architecture on which the *Fraud* design decision is based. The realization and choice part of the *Fraud* design decision is shown together with the "resulting" architecture below them. The same is done for the *Fraud Integration* design decision.

Note that in the view of design decision (figure 4.6) the mapping of the change elements onto the architecture is visualized in the change elements themselves. For example, in the *Notification* of the *Fraud Integration* design decision the *Submission Notification* delta is mapped onto the *Domain Object*. This mapping is defined with the help of the composition model (see section 4.5.3). However, the visualization of this mapping is not visible in this view. Instead, when the mapping is not clear from the delta name, the $>$ symbol followed by the target of the delta is used to clarify the mapping.

From both design decisions emerge a number of additional requirements. For example, in the *Fraud* design decision an Internet connection to the Moss server is now required for the Athena system to function completely. The same happens with the *Fraud integration* design decision, where requirements are needed about the expected feedback of the fraud system.

Note that the description of the design decisions itself was sufficient to describe the software architecture evolution and its reasons. For example, the Archium model contains all the information needed to explain why the *Fraud Scanner* component is part of the system. Usually with the term "architecture", the latest incarnation of a design is intended. In this case this would be the architecture illustrated on the bottom of figure 4.6. However, as shown, this architecture is the result of a number of design decisions.

4.7 Related work

Archium employs concepts from the field of software architecture [119]. Important concepts in this field are components and connectors, which are believed to lead to better control over the design, development and evolution of large and increasingly dynamic software architectures [11]. Software architecture documentation approaches [29, 67] try to provide guidelines for the documentation of software architectures. However, guidelines for design decisions are absent in these approaches, whereas Archium does provide them.

Architectural Description Languages (ADLs) [107] describe software architectures in a formal language that supports first class architectural concepts. Whereas ADLs try to describe an architecture, Mae [161] and Archium try to describe the evolution leading to an architecture. Mae [161] is an architectural change management tool that tracks changes to an architecture definition by a revision management system. However, it lacks the notion of a design decision and delta. Therefore, it can only track arbitrary changes and not the dependencies between design decisions that the architect is interested in.

Component languages like ArchJava[2] and Koala [170] are programming languages supporting some architectural concepts as first-class entities to various degrees. Archium shares some the concepts of these component languages, but differs as design decisions and architectural change are first-class citizens.

AOP[88] with its implementations like AspectJ[87] and genVoca [13] are approaches using different techniques to achieve multiple separation of concerns. Traditional use of AOP focuses on the code level, on the other hand Archium focusses on the cross-cutting concerns of design decisions at the architectural level.

A design pattern is a special type of design decision. Design patterns [49] are sets of predefined design decisions with known functionality and behavior. The rationale of these decisions, as documented in the description of a design pattern, can be related to the realization [9]. Archium differs from [9] as it keeps design patterns first class in the realization.

Knowledge systems [128], like IBIS [31], model decision processes and try to capture the rationale or knowledge used in these processes. Design decision models [128] are a special type of knowledge system, as they try to capture rationale of design decisions. Archium expands these decision models, as it integrates the decision model with an architectural model.

4.8 Conclusion

Architectural design decisions play an important role in the design, development, integration, evolution, and reuse of software architectures. However, the notion of architectural design decisions is not part of the current perspective on software architectures. We have identified several problems due to this, including high costs of change, design erosion, and limited reuse, which are primarily caused by the vaporization of these design decisions into the architecture.

To address these problems, we propose a new perspective on software architecture, where software architectures are described as set of design decisions. The presented Archium approach is centered around this idea. Archium models the relationship between software architecture and design decisions *in detail* for the first time. It uses a conceptual model consisting of the notions of deltas, design fragments, and design decisions to describe a software architecture. The different concepts were exemplified with the Athena case.

Ongoing and future work on Archium includes the development of tool support (see [5]) facilitating experimentation of the various concepts. In addition, we intend to add support for multiple views on the architecture as different views show different concerns about the architecture [29, 67].

This paper presented a first step in modeling the perspective of software architectures as a set of design decisions. Many research challenges remain in this perspective. For example, how can we distinguish important design decisions? What are interesting relationships between design decisions? What are the crucial factors influencing a design decision?