CHAPTER

Architectural Debt Management in Value-Oriented Architecting

9

Zengyang Li[1], Peng Liang[2], and Paris Avgeriou[1]

[1]*University of Groningen, Groningen, The Netherlands*

[2]*Wuhan University, Wuhan, China*

9.1 Introduction

In the field of software architecture (SA), there has been a paradigm shift from describing the outcome of the architecting process to documenting architectural knowledge (AK), such as architecture decisions and rationale, which are considered as first-class entities of a software architecture (ISO/IEC/IEEE, 2011). Architecture decisions often involve trade-offs made between a number of stakeholder concerns. In particular, technical concerns (e.g., system quality attributes) are often compromised to meet business concerns (e.g., development cost or time to market). For example, poorly designed legacy components may be reused instead of implementing their functionality from scratch, in order to achieve fast product delivery. Such trade-offs made in architecture design may lead to **architectural technical debt** (ATD, or shortly *architectural debt*).

In a broader scope, technical debt (TD) refers to immature software artifacts that fail to meet the required level of quality (Cunningham, 1992; Seaman and Guo, 2011). Accordingly, ATD refers to immature architecture design artifacts that compromise systemwide quality attributes (QAs), particularly maintainability and evolvability. On the one hand, TD needs to be repaid sooner or later, as it may have grave consequences for future software development cycles; on the other hand, TD (and ATD as a type of TD) is not necessarily a "bad thing," but rather something that can be leveraged for business advantage when incurred with full knowledge of the consequences, that is being explicitly managed (Kruchten et al., 2012).

Although many approaches have been proposed to document architecture decisions in the architecting process (e.g., decision views in architecture design; see van Heesch et al., 2012a, 2012b), the ATD caused by decisions is still not effectively managed. In most cases, ATD is not made explicit, and architecture decision making does not take into account the ATD that will be incurred by the different design options. This may cause problems particularly during system maintenance and evolution, when ATD is accumulated and difficult to repay. In this chapter, we present an initial attempt to tackle this problem through the following: (1) an ATD conceptual model; and (2) an architectural technical debt management (ATDM) process applying the proposed conceptual model, and aligned with a general architecting process. Our contribution to this end can facilitate optimal decision making in architecture design and achieve a controllable and predictable balance between the value and cost of architecture design in the long term.

The rest of this chapter is organized as follows: Section 9.2 discusses architectural technical debt, while section 9.3 proposes an ATD conceptual model. Section 9.4 presents an ATDM process integrating the proposed conceptual model and the application of the ATDM process in value-oriented architecting. Section 9.5 describes an industrial example to demonstrate how value-oriented architecting with ATDM works in real-life projects. Section 9.6 discusses work related to the topic of this chapter, and Section 9.7 concludes this chapter with future research directions.

9.2 Architectural technical debt

Technical debt in software development has attracted increasing interest from practitioners and researchers in the software engineering community. Technical debt is a metaphor, coined by Ward Cunningham in 1992, for the trade-off between writing "clean" code at higher cost and delayed delivery, and writing "dirty" code cheaper and faster by making shortcuts resulting in higher maintenance cost once it is shipped (Buschmann, 2011; Cunningham, 1992). This metaphor was initially proposed and concerned with software coding. Currently, the concept of technical debt is extended to other phases in the software development life cycle, such as software architecture design, detailed design, and even software documentation and testing (Brown et al., 2010; Ozkaya et al., 2011).

TD is essentially invisible to users because they cannot witness the existence of TD when they are using a software system that works well. Conceptually, technical debt concerns the technological gaps between the current solutions and the optimal solutions, which may have a negative impact on system quality, especially the maintainability and evolvability of a software system (Kruchten et al., 2012). Architectural technical debt (ATD) is a type of TD at the architectural level. It is caused mainly by architecture design decisions that compromise the maintainability and evolvability of a software system. In contrast, code-level technical debt is concerned with the quality of the code and is usually incurred by the poor structure of the code and disobedience of coding rules and best practices (i.e., bad code smells).

Maintainability and evolvability are the two main system quality attributes that are compromised when incurring ATD. According to the ISO/IEC FDIS 25010 standard (ISO/IEC, 2011), maintainability includes the following subcharacteristics (i.e., quality attributes): modularity, reusability, analyzability, modifiability, and testability. Evolvability is not defined in either ISO 9126 or ISO/IEC FDIS 25010. We define software evolvability as the ease of adding new or changing existing requirements (functional and nonfunctional). As an example of ATD, consider an architecture decision, which uses a legacy component implemented with an obsolete technology to speed up development. This may make it hard to add new functionalities with new technologies that are incompatible with the obsolete technology. In summary, ATD essentially results from the compromise of modularity, reusability, analyzability, modifiability, testability, or evolvability during architecting. In this chapter, we only consider the quality attributes (QAs) maintainability and evolvability, while other QAs are out of scope of ATD (Kruchten et al., 2012).

ATD, as a kind of TD, can be seen as an important type of risk for a software project in the long term (Seaman and Guo, 2011), but the architecture and management teams often ignore ATD. The main reason is that ATD concerns the cost of the long-term maintenance and evolution of a software system instead of the visible short-term business value. Furthermore, ATD is not easy to identify and measure since it is invisible until the following cases happen: Maintenance tasks are hard to conduct, new features are difficult to introduce, and system quality attributes are challenging to meet. This chapter helps solve this problem by making ATD explicit through a conceptual model and by offering a process to manage ATD through explicit cost-benefit trade-offs.

ATD is incurred by either explicit or implicit architecture decisions. ATD can be managed in two ways: When architecture decisions are being made and after decisions have been made. The former aims at dealing with ATD before it is incurred by an explicit architecture decision, whereas the latter focuses on handling ATD after it has been incurred by an existing explicit or implicit architecture decision. Both ATD management approaches are presented in section 9.4.

9.3 ATD conceptual model and template

This section proposes an ATD conceptual model for capturing and using ATD in the architecting process.

9.3.1 Conceptual model

We constructed an ATD conceptual model that is depicted in Figure 9.1 using UML notation, based on our understanding of ATD and TD literature (Brown et al., 2010; Kruchten et al., 2012). The gray part of this model (*Architecture rationale*, *Architecture decision*, and *Concern*) represents the concepts

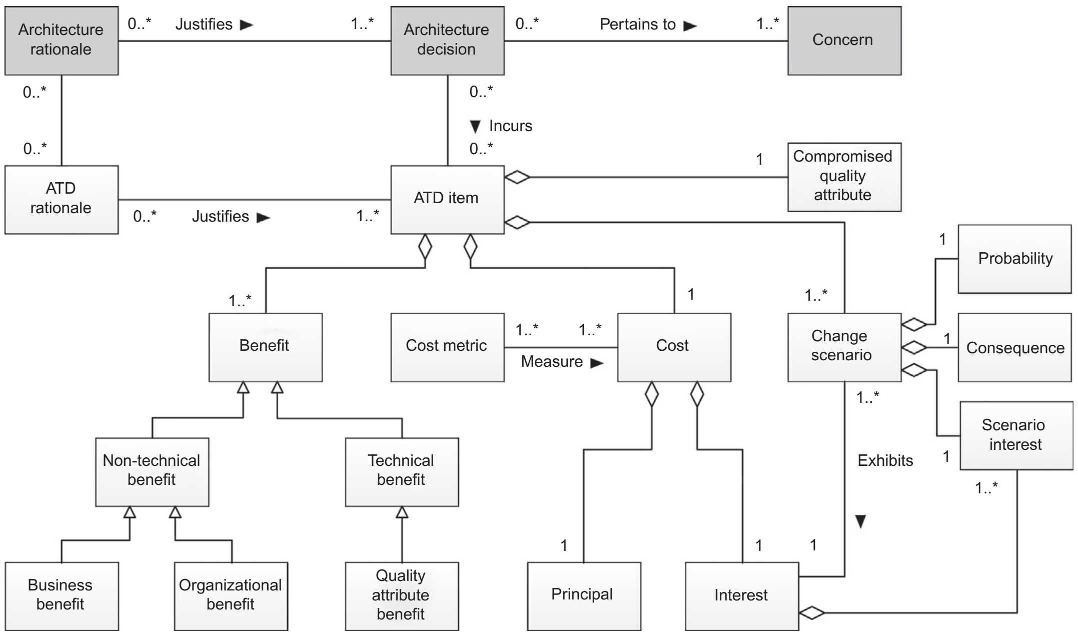


FIGURE 9.1

ATD Conceptual model.

adopted from the conceptual model of architecture decisions and rationale in ISO 42010:2011 (ISO/IEC/IEEE, 2011). In this conceptual model, the core concept is the ATD item, which acts as the basic unit to record ATD. An example of an ATD item is presented in section 9.3.2. Note that in the rest of this chapter, the phrases "resolve an ATD item" and "repay ATD" will be used interchangeably.

- An **ATD item** is a basic unit of ATD that is incurred by an architecture decision that compromises a system quality attribute: evolvability or maintainability. The detailed representation and description of this concept with a template are presented in section 9.3.2.
- **ATD rationale** justifies why an ATD item is incurred, and it records explanation, justification, or reasoning about an ATD item incurred. The ATD rationale for an ATD item may partially use the architecture rationale for the architecture decision that incurs the ATD item, when the architecture rationale explains trade-offs between maintainability or evolvability and other system quality attributes.
- A **compromised quality attribute** refers to the QA that is sacrificed to meet other concerns (e.g., business benefit). A compromised QA concerning an ATD item can only be either maintainability (which includes the following sub-QAs: modularity, reusability, analyzability, modifiability, and testability (ISO/IEC, 2011) or evolvability according to the clarification of why TD is incurred (Kruchten et al., 2012).
- **Cost** refers to the sum of the effort (e.g., person-day, time, or money) that is needed to resolve an ATD item and the added effort spent on maintenance and evolution tasks.
- A **cost metric** is used to measure the cost to resolve an ATD item in a quantitative way, and it can be person-days, calendar days, monetary units (e.g., U.S. dollar), or others.
- **Principal** refers to the cost if an ATD item is being resolved at the time the ATD item is identified, that is, according to the architecture design at that time.
- **Interest** refers to the extra cost due to maintenance or evolution work if an ATD item is not resolved. The interest of an ATD item may increase when related changes take place in the part of the software architecture that contains the ATD item. For instance, the principal of an ATD item is five person-days for the current release of the system, while the interest will be another three person-days when the ATD item is left unresolved in the next release of the system. The interest of an ATD item consists of the scenario interests caused by a set of relevant change scenarios, which are explained in the next bullet.
- A **change scenario** describes a possible change (an evolution or a maintenance task) that influences an ATD item and the consequence of this change. A change scenario can be used to calculate the interest of an ATD item. Typical change scenarios include: (1) the unimplemented features that are planned in the roadmap of the software system, but difficult to introduce without modifying the architecture; and (2) the maintenance tasks that improve certain QAs (except maintainability and evolvability which have been compromised in ATD) of the implemented software architecture. Each scenario consists of three elements: consequence, scenario interest, and probability.
- **Consequence** refers to extra work resulting from a change scenario when the related ATD item is unresolved.
- **Scenario interest** refers to the interest of a change scenario related to an ATD item.
- **Probability** refers to the likelihood a change scenario will actually happen in the next release.
- **Benefit** refers to the positive impact on the system when an ATD item is incurred, for example, shorter time to market or improved system quality.

- **Technical benefit** refers to the benefit gained in terms of design-time or runtime quality attributes of the software system, when an ATD item is incurred.
- **Nontechnical benefit** refers to the benefit gained in terms of business and organizational aspects when an ATD item is incurred.
- **Business benefit** refers to the benefit gained in business aspects when an ATD item is incurred, such as shorter time to market or decreased development cost.
- **Organizational benefit** refers to the benefit gained to the organization that develops the software system (i.e., the benefit to the organization instead of the software system itself) when an ATD item is incurred. As an example, consider an organization that chooses to reuse in a new system existing components developed in other projects, though these components do not fit the software architecture perfectly. As a result, the organization does not need to maintain various components with similar functionalities.
- **Quality attribute benefit** refers to the benefit gained in terms of improvement of a specific quality attribute of the software system when an ATD item is incurred. The improved quality attribute can be any type except maintainability and evolvability. Here is an example: A software system adopts the relaxed layered pattern instead of the strict layered pattern to achieve higher performance at the cost of lower maintainability.

Note that the interest of an ATD item will increase as changes occur in the part of the architecture that influences the ATD item; we propose to measure the interest of an ATD item using software release as the time unit. The time length within which one can predict possible change scenarios, influences the amount and accuracy of the estimated interest of ATD items. The longer time length one adopts, the more change scenarios one will get but with less accuracy. We argue that architects can predict change scenarios in the next release of a software system more reasonably since they are more certain about what changes will occur in the next release than in the next two or more releases.

9.3.2 ATD item

As shown in Table 9.1, the ATD item template provides detailed information needed to document an ATD item. This template is based on the ATD conceptual model, so most of the elements in this template are adopted from the model. The explanation of each element is briefly described in the template. We provide more detailed description about some key elements in the template. *ID* is the unique identification number of the ATD item, so that an ATD item can be referred to within the architecture description of a software system by using this ID. The *Name* of an ATD item reflects the essence of the ATD item. The *Status* of an ATD item can be unresolved or resolved. Resolved ATD items of a software architecture are a type of architectural knowledge (AK) of the software system. This type of AK shows how the ATD of a software system was managed; therefore, it can benefit future decision making of this system, or it can be reused in similar systems. The unresolved ATD items of a software system should be monitored. An ATD item is *Incurred by* an architecture decision. In order to keep the scope of an ATD item manageable, if ATD is incurred by a group of architecture decisions, we decompose it into several ATD items so that each item is caused by an individual architecture decision. *Change scenarios* are used to measure the interest of an ATD item. Each scenario with a *scenario number (#)* consists of *scenario description*, *consequence*, *scenario interest*, and *probability*. The *interest* of an ATD item is the

Table 9.1 Template for Documenting an ATD Item

ID	An Unique Identification Number of the ATD Item
Name	The name of this ATD item
Date	The date when this ATD item was generated or changed
Status	Resolved or unresolved
Incurred by	The decision incurs this ATD item
Responsible	The name of the person or team responsible for managing this ATD item
Compromised QA	The QA(s) that are compromised, from modularity, reusability, analyzability, modifiability, testability, or evolvability
Rationale	The reason the ATD item is incurred
Benefit	The value gained when this ATD item is incurred
Cost	The cost suffered by incurring this ATD item, which is the sum of principal and interest below.
Principal	The cost if this ATD item is resolved at the time when the ATD item is identified
Interest	The interest that this ATD item accumulates (the interest is calculated based on the predicted change scenarios described below)

Change Scenarios	**#**	**Scenario description**	**Consequence**	**Scenario interest**	**Prob.**
	1	Scenario 1	consequence of scenario 1	I_1	P_1
	2	Scenario 2	consequence of scenario 2	I_2	P_2
	. . .	. . .	. . .	. . .	. . .
	n	Scenario n	consequence of scenario n	I_n	P_n
	The interest of this ATD item (total interest) $= \sum_{k=1}^{n} I_k \times P_k$				
Architecture Diagram	A diagram or model that illustrates the concerned part in the architecture design				
History	Change history of this ATD item				

sum of the product of the *scenario interest* of each scenario and its *probability* as calculated by the formula in Table 9.1.

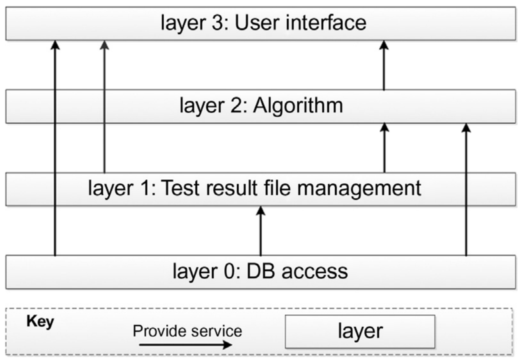
To explain the use of this template to document ATD items, a concrete example of an ATD item is presented in Table 9.2. This ATD item is from an architecture design of an industrial information system with which users can query product test results and customize reports on product test results. The test results of various products, which are generated by external automatic test systems, are stored in a remote database. There are also some pictures and text files as the test results of some types of tests and these files are stored in file servers. The current architecture is made up of four layers as shown in Table 9.2. The User Interface layer handles users' events. The Algorithm layer is responsible for generating product quality reports. The Test Result File Management layer is responsible for getting test result files from right server and parsing test results files into specific forms. The Data Access layer provides services to store and query the data in the database.

Table 9.2 An Example ATD Item

ID	ATD-6
Name	Compromised modifiability due to relaxed layered pattern
Date	26-10-2012
Status	Unresolved
Incurred by	Decision-6: Using relaxed layered architectural pattern to speed up development
Responsible	Zengyang
Compromised QA	Modifiability
Rationale	From a technical perspective, the strict layered pattern is a more appropriate solution. We decided to employ the relaxed layered architectural pattern to allow invocations to go across layers. Compared with the strict layered pattern, it is not necessary to encapsulate everything within the upper layers from the lower layers. Thus, we can save development time by skipping the effort of encapsulating every service of a layer from the layer below. Meanwhile, the performance of this software system is improved since invocations can go across layers in the relaxed layered pattern. But this solution will have a negative impact on modifiability of the system when the predicted change scenarios happen since a change in a lower layer may cause modifications of all upper layers that directly depend on the lower layer.
Benefit	i. 11 person-days of development time are saved by implementing the relaxed layered pattern compared to implementing the layered pattern in the system. ii. The performance of the software system is improved.
Cost	Principal + Interest = 10 + 15.2 = 25.2 person-days
Principal	10 person-days (if we use the layered pattern to resolve this ATD item at the time when the ATD item is identified)
Interest	$12 \times 0.6 + 10 \times 0.8 = 15.2$ person-days

Change Scenarios	**#**	**Scenario description**	**Consequence**	**Scenario interest**	**Prob.**
	1	Change file transfer method in layer 1	Both layer 2 and layer 3 are influenced and must be modified accordingly.	12 person-days	0.6
	2	Replace the data access method of the lowest layer (DB Access) because a new technology is adopted	All the upper layers need to be modified accordingly.	10 person-days	0.8

Architecture Diagram



History
Created: 18-07-2012 by Zengyang
Updated: 26-10-2012 by Zengyang, revised the probability of change scenario 1 from 0.5 to 0.6.

9.4 Method

This section proposes a decision-based ATDM (DATDM) approach. To derive the process, we collected activities for technical debt management from a number of publications surveyed in (Li et al., 2013), and we combined these activities to form an ATDM process (i.e., the DATDM approach) that is based on architecture decisions and integrates the proposed ATD conceptual model. The DATDM approach is introduced and employed in the architecting process to facilitate various architecting activities.

9.4.1 ATDM process

We surveyed publications on technical debt management and identified a number of activities for managing TD (Li et al., 2013). These activities have been adapted from TD to ATD. ATD identification, measurement, and monitoring are adopted from Brown et al. (2010). ATD prioritization is selected from Zazworka et al. (2011). ATD repayment is introduced in Brown et al. (2010). The details of each ATDM activity as well as their input and output in the architecting process are as follows.

1. **ATD identification** detects ATD items during or after the architecting process. An ATD item is incurred by an architecture decision; thus, one can investigate an architecture decision and its rationale to identify an ATD item by considering whether the maintainability or evolvability of the software architecture is compromised.
2. **ATD measurement** analyzes the cost and benefit associated with an ATD item and estimates them, including the prediction of change scenarios influencing this ATD item for interest measurement. For interest measurement, three types of change scenarios are considered: (1) the planned new features according to the release plan of the software project; (2) the already-known maintenance tasks that enhance specific QAs (except maintainability and evolvability) of the implemented software architecture; and (3) the emerging requirements. The first two types of change scenarios can be predicted, while the third is unforeseeable. For some complex software systems (e.g., operating systems), the time interval between two releases can be very long. For instance, Microsoft Windows 7 Service Package 1 was released 16 months after the first release of Microsoft Windows 7. For such kind of software system, it is inevitable that new requirements emerge during the development of a new release. Some of these new requirements need to be implemented in the release. Thus, in such cases, to ensure a reasonable accuracy of interest measurement, the interest of related ATD items should be re-measured at different times during the development of the release.
3. **ATD prioritization** sorts all the identified ATD items in a software system using a number of criteria. The aim of this activity is to identify which ATD items should be resolved first and which can be resolved later depending on the system's business goals and preferences. There are a number of ATD items in a software system, and not all the ATD items will be resolved at one time owing to their costs or technical issues. The ATD items have different financial and technical impacts on the system. Consequently, it is wise to choose the items with higher priorities to be resolved first. Software projects have different contexts, and there are no standard criteria to decide the priority of an ATD item in a project. However, the following factors need to be taken into account in ATD prioritization: (1) the total cost of resolving an

ATD item; (2) the cost/benefit ratio of the ATD item; (3) the interest rate of the ATD item; (4) how long the ATD item has been incurred; and (5) the complexity (e.g., the number of involved components of an ATD item) of resolving an ATD item. Since not all types of benefits can be measured in a unified metric, it is hard to automatically prioritize the ATD items by tooling. However, an appropriate tool, which reasonably deals with the factors described above, can facilitate ATD prioritization.

4. **ATD repayment** concerns making new or changing existing architecture decisions in order to eliminate or mitigate the negative influences of an ATD item. An ATD item is not necessarily resolved at once. In certain situations, only part of an ATD item is resolved, because it could be too expensive to resolve the entire ATD item, and resolving part of the ATD item can make the ATD item under control with an acceptable cost. When an ATD item is partially resolved, the ATD item will be revised and split into two parts: the part that is resolved and the part that is not.
5. **ATD monitoring** watches the changes of the costs and benefits of unresolved ATD items over time. When an architectural change occurs in the part of architecture design containing an unresolved ATD item or when one ATD item is partially resolved, the affected ATD item will be recognized as a changed ATD item. All the changed ATD items will be measured in the next ATDM iteration. This ATDM activity makes ATD changes explicitly and consequently keeps all the ATD items of the system under control.

Figure 9.2 shows the process of ATDM with the inputs and outputs of each ATDM activity. In a software project, the ATDM process is generally performed multiple times during the life cycle of the project. An ATDM iteration is defined as a period in which the ATDM process goes through all possible ATDM activities. For instance, an ATDM iteration can be a release, an increment development period, or a Sprint in Scrum. The time when an ATDM iteration will happen depends on the actual necessity of performing ATDM activities. The activities in the ATDM process can be revisited when necessary. In addition, it is not mandatory that all ATDM activities should be performed in every iteration of the ATDM process. In the rest of this section, we discuss how this ATDM process deals with ATD items. The detailed ATDM process is presented as follows:

1. When a new architecture decision made results in compromising maintainability or evolvability, the architect identifies a new ATD item. If an architecture decision is reconsidered, there are two options: If it already compromised maintainability or evolvability, then the existing ATD item is revised; if it did not previously compromise maintainability or evolvability, then a new ATD item is created. First, the architect identifies which QA is compromised by this decision by analyzing the documented architecture decision and its rationale. Then, taking the compromised QA as the basis of the *ATD identification*, the architect further identifies the ATD item and rationale.
2. The ATDM process moves to the *ATD measurement*. After change scenarios related to the architecture decision are predicted, the architect estimates the interest of this ATD item based on these change scenarios. With the ATD rationale, the architect also estimates the principal and the benefit of the ATD item.
3. The ATDM process continues with *ATD prioritization*. ATD prioritization sorts all the identified ATD items according to specific criteria (e.g., the ATD item with higher cost gets a higher priority to be resolved) when a new ATD item is introduced or any existing ATD item is changed. The criteria are defined by architects according to the concrete architecting context

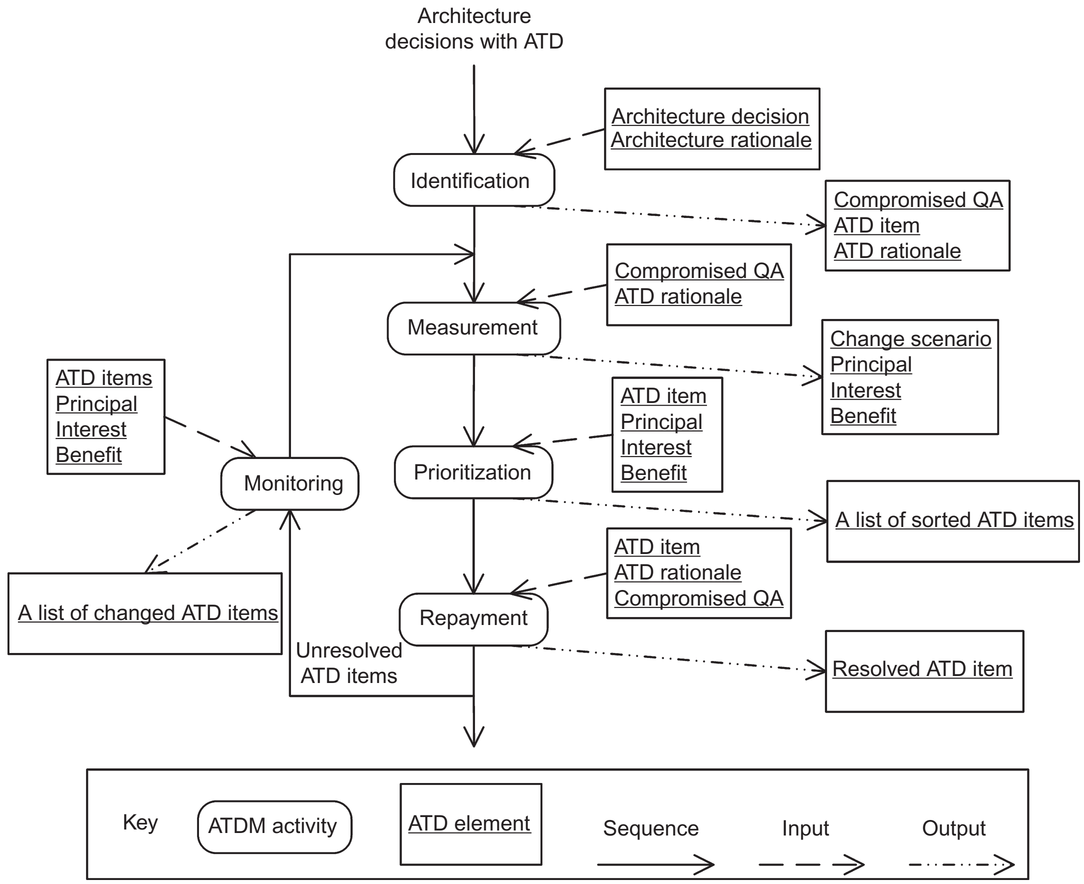


FIGURE 9.2

ATDM process with inputs and outputs of ATDM activities.

since there is no uniform unit to compare all the costs and benefits of ATD items. The architect then browses the prioritized ATD items and decides the ATD items that should be resolved.

4. If there are ATD items that should be resolved, the ATDM process moves to *ATD repayment*. Otherwise, the ATDM process moves to *ATD monitoring* of existing ATD items. *When any* change occurs with the monitored ATD *items, the ATDM* process will repeat the following activities sequentially: *ATD measurement*, *ATD prioritization*, and *ATD repayment*.

9.4.2 ATDM within the architecting process

Hofmeister et al. proposed an architectural design model that consists of three core architecting activities: architectural analysis, synthesis, and evaluation (Hofmeister et al., 2007). Architectural analysis "examines, filters, and/or reformulates architectural concerns and context in order to come

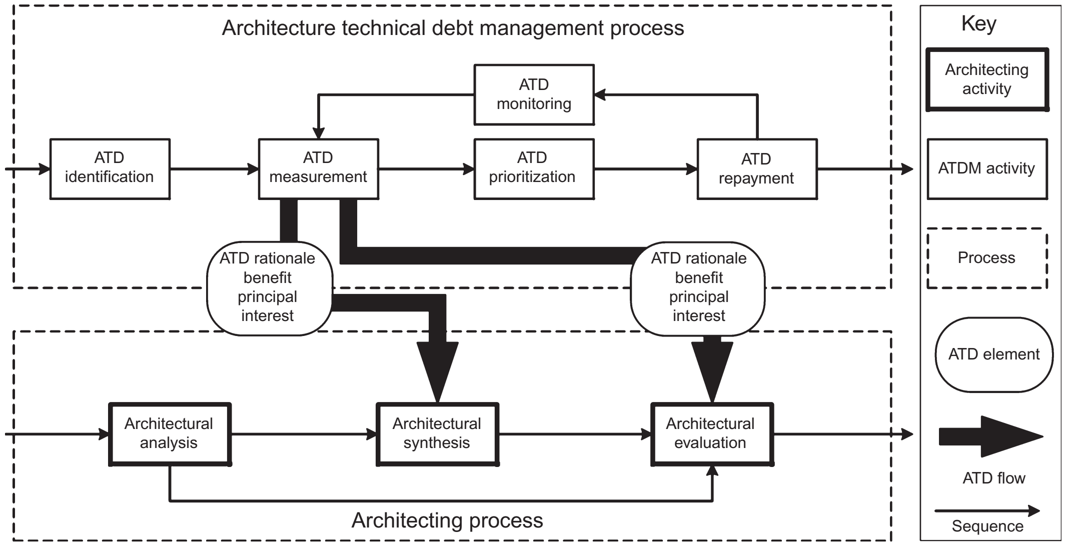


FIGURE 9.3

ATDM in architecting process with ATD flow.

up with a set of architecturally significant requirements (ASRs)" (Hofmeister et al., 2007, p. 113). Architectural synthesis proposes a collection of architecture solutions to address the ASRs identified during architectural analysis (Hofmeister et al., 2007). Architectural evaluation evaluates the candidate architectural solutions that are proposed during architectural synthesis against the ASRs (Hofmeister et al., 2007). These architecting activities are not performed sequentially but are iterated during architecture design. Furthermore, these activities are also performed during software maintenance and evolution in order to maintain the consistency and correctness of the architecture.

In this work we apply the proposed ATDM process (i.e., DATDM approach) to managing ATD within the general architecting process in Hofmeister et al. (2007). A software architecture can be considered as a set of architecture decisions (Jansen and Bosch, 2005). Therefore, the architecting process can be regarded as a decision-making process. The objective of this approach is to facilitate the architecture decision-making process by managing ATD and, consequently to assist architects in making appropriate and well-founded decisions and to ensure that the ATD of a system remains controllable.

Figure 9.3 shows ATDM within the architecting process and focuses on how the ATD flow can facilitate the architecting activities. The ATD flow refers to a kind of data (artifacts) flow from the ATDM process to the architecting process. The ATD flow, consisting of instances of ATD concepts, bridges the gap between the ATDM process and the value-oriented architecting process. ATDM can facilitate both architectural synthesis and evaluation. In particular, measured ATD items can be used as input for architectural synthesis and evaluation activities: (1) In architectural synthesis, an architect can reflect on the design options for a decision topic in terms of ATD, and particularly in terms of the costs and benefits of the identified ATD items. (2) In architectural evaluation, the evaluator assesses the architecture decisions made (either implemented into code or not) against the related architecturally significant requirements (ASRs), for example, scenarios pertaining to a certain quality attribute. The identified and measured ATD items can be used as inputs and outputs of architectural evaluation. The existing architectural evaluation methods tend to assess to what extent the architecture design meets the existing requirements. ATDM is concerned with the balance of cost and benefit of a software system from an ATD perspective. Thus, ATDM provides a complementary perspective on the costs and benefits of architecture decisions caused by ATD to existing evaluation methods.

ATDM can be used to facilitate decision making in architectural synthesis (i.e., make architecture decisions for decision topics) and to evaluate architecture decisions that have been made. ATDM activities can be triggered in the following situations: (1) maintenance tasks are hard to complete, or new functionalities are difficult to add and implement in an existing architecture; (2) there is a need to update the information of existing ATD items of an architecture due to changes of this architecture; and (3) there is a need to evaluate the existing decisions at any point of time.

In architectural synthesis, for each decision topic, the main steps in using the DATDM approach are:

1. Proposing design options for the decision topic.
2. Identifying ATD items. The architect identifies the ATD items incurred by each design option for the decision topic by analyzing each design option.
3. Measuring ATD items. For each ATD item identified in Step 2, the cost and benefit are estimated.
4. Making the architecture decision with consideration of ATD items. The ATD items are also recorded as part of the rationale of the architecture decision.

In architectural evaluation, the main steps in using the DATDM approach are:

1. Collecting architecture decisions of those types: (i) that have not incurred ATD; (ii) that are related to architectural maintenance tasks hard to conduct; (iii) that are related to new requirements difficult to introduce; and (iv) that are related to changes of the existing architecture design. Architecture decisions with their rationales are the inputs for the next step.
2. Identifying ATD items. For each architecture decision collected in Step 1, identify the ATD items incurred by the architecture decision through analyzing the decision and its rationale.
3. Measuring ATD items. For each architecture decision, estimate the costs and benefits of the ATD items (including the newly identified ATD items in Step 2 and the ATD items already identified in architectural synthesis and previous architectural evaluations).
4. Evaluating architecture decisions with consideration of ATD items, more specifically the costs and benefits of ATD items.

After architectural evaluation, all the identified ATD items will be prioritized and the ATD items with the higher priorities will be the candidate ATD items to be resolved first. Then, the ATDM process moves to ATD repayment. To resolve an ATD item, the architecting process revisits architectural synthesis activity with this ATD item as an input.

9.5 Case study

This section presents an industrial example to demonstrate how the proposed ATD conceptual model and process for ATDM can support reaching a balanced architecture design in terms of value and cost in the long term. The software system used in this example is a system for automatic testing of hardware products with embedded systems running on them—Automatic Test System (ATS)—which is a real-life project developed by a leading telecommunications equipment manufacturer in China. The rest of this section introduces the background of the ATS, the architecture design of the ATS, and the use of decision-based ATDM in the architecting process.

9.5.1 Background

ATS aims to automatically test hardware products. Typical tested hardware products by ATS are mainboards of high-performance routers and base stations of wireless telecommunications. Generally speaking, two kinds of tests need to be performed on the products before delivery: functional test and performance test. The functional test is used to ensure that the functionalities of the tested products work in various conditions (e.g., high-temperature and low-temperature environments), while the performance test is to make sure the tested products satisfy the minimal performance requirements (e.g., a specific radio frequency sensitivity of telecommunication base stations) in different conditions. The ATS in this example is dedicated to performing functional tests of base stations.

Typically, an ATS is a combination of hardware and software subsystems. The hardware subsystem is used as a test environment. The software subsystem is responsible for communicating with the Unit Under Test (UUT), perform the test, and generate the test results in various forms. In this example, the ATS refers to the software subsystem in a typical ATS. In this case study, the ATS provides features, such as test automation, test results visualization, and test results persistence (i.e., storing test results to a database).

9.5.2 Architecture design

Figure 9.4 shows the main components in the software architecture of the ATS. ATS Controller handles the commands from the GUI component. The typical test process is described as follows: (1) A user logs in to the ATS through the *GUI* component, and *ATS Controller* deals with the user login in *User Management*. (2) The user configures the current test through *GUI* component, and the *ATS Controller* deals with the test configuration in *Test Settings*. (3) The user starts an automatic test in the *GUI* component, and the *ATS Controller* starts the *Test Engine* that is the most important component and responsible for executing the *Test Item Flow*. (4) The *Test Engine* loads the test configuration data for the current test, requests the *Test Item List Parser* to parse all the test items of this test, and loads the specific *Test Item Flow* for test items. *Test Item Flow* implements the concrete test execution flow for all test items of a specific test. (5) The *Test Engine* executes each test item and generates test results. The *Test Engine* updates these test results to the *GUI* component and *Test Result Saver*, so that the user can see the real-time test results in the *GUI* component and the test results can be collected and stored in right time. (6) *Test Result Saver* stores test results to the database (*MS SQL Server*) through *DB Handler*, which provides services to store data to and query data from the *MS SQL Server*. (7) If any fault or exception happens during the test, the *Logger* records the fault or exception information in a log file to assist problem tracing and fixing.

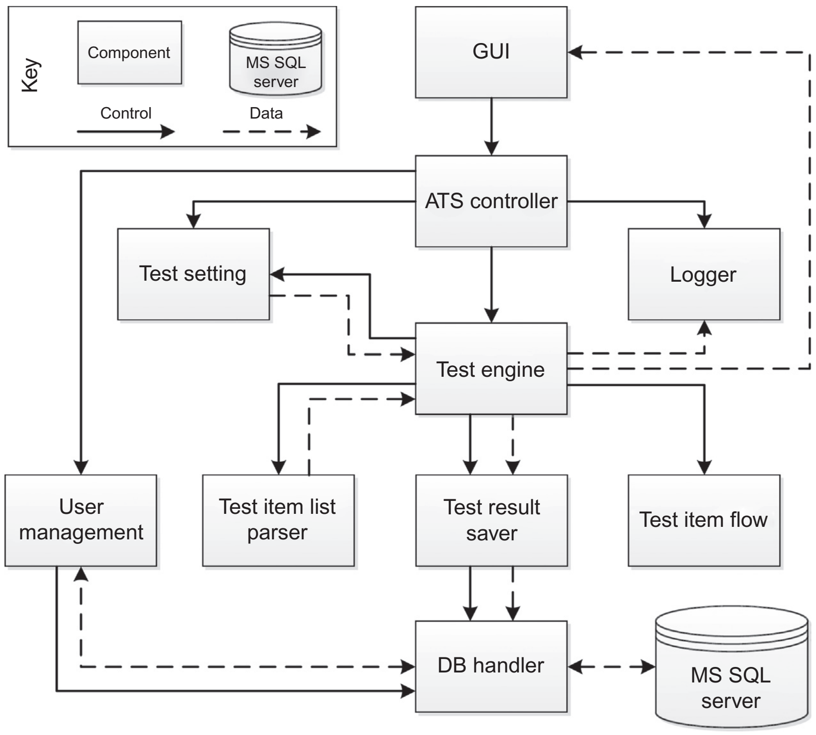


FIGURE 9.4

Component diagram of ATS software architecture.

9.5.3 Using DATDM in architecting activities

This subsection illustrates two concrete examples using DATDM in architectural synthesis and evaluation, respectively, in the context of the ATS architecture design.

9.5.3.1 ATDM in architectural synthesis

In architectural synthesis, the architect proposes design options for each architecture decision topic and then chooses the more appropriate one that addresses the decision topic according to the pros and cons of the design options.

To improve the reliability of the test results persistence and to keep the ATS running uninterruptedly when the remote database server (MS SQL Server) is inaccessible, a decision topic arises (of AD-12): how to store test results locally in a "cache" when the remote database server is inaccessible. The architect proposes two design options for this decision topic: storing test results in XML files (design option 1) and storing test results in MS Access (design option 2). Since the test results are stored in a remote database and a local database (or file), any change to the table design in the remote database needs to be updated to the local one. Otherwise the test results cannot be stored correctly. In addition, the test results in the local database (or file) need to be synchronized to the remote database. Therefore, the two design options will incur ATD. The ATD items ATD-10 and ATD-11 that are incurred by the two design options are presented in Table 9.3 and Table 9.4, respectively.

After comparing the ATD items with their costs and benefits, the architect chooses design option 2 as the solution (design decision) to address the decision topic, since the cost of ATD-11 (incurred by option 2) is less than ATD-10 (incurred by option 1) and the benefit of ATD-11 is more valuable for the project currently in the current release.

9.5.3.2 ATDM in architectural evaluation

Suppose that the first release of the ATS product has been delivered; the architecture of ATS needs an architectural evaluation since the first delivery was developed in a tight schedule and its architecture design did not receive a serious external evaluation by an independent party. As an example, architecture decision AD-8 was evaluated using the DATDM approach. The architecture decision is described below:

AD-8: The Test Engine updates test results to the GUI component and Test Result Saver. This architecture decision is adopted from a previous automatic test application, so that some legacy components can be reused in this ATS.

According to the DATDM approach to architectural evaluation described in section 9.4.2, the following steps are performed:

1. Architecture decision AD-8 with its rationale is used as the input of the architectural evaluation.
2. An ATD item (i.e., ATD-3) is identified using the ATD item template as shown in Table 9.5.
3. Two change scenarios are predicted. These scenarios are negatively influenced by AD-8, and they are used to measure the cost of ATD-3. The benefit of ATD-3 is estimated according to the decision and its rationale. The benefit and cost of ATD-3 are analyzed and described in Table 9.5.
4. ATD-3 is considered as part of output of this architectural evaluation.

In addition, suppose that a number of ATD items are identified and measured in the ATS architecture design in the architectural evaluation using DATDM. After analyzing and comparing the benefits and costs of these ATD items, ATD-3 is prioritized as the most critical ATD item, and the architect decides that this ATD item should be resolved urgently since it negatively influences the evolution of the ATS. Therefore, the ATD-3 enters the ATD repayment activity in the DATDM process. The root reason resulting in ATD-3 is that the generation of test results is not transparent to use of the test results. If the architect adds a *Test Result Repository* as an intermediate component between *Test Engine* and *GUI component* with *Test Result Saver*, the *Test Engine* is then free from direct uses of test results. Consequently, adding a new functionality that uses the test results will not result in the modification and testing cost to the *Test Engine*. Therefore, a solution to resolving architectural technical debt ATD-3 is to add a *Test Result Repository* that stores the test results temporarily, as shown in Table 9.6. The *Test Engine* updates the latest test results to the *Test Result Repository*, and all the components that use test results will request the test results from the *Test Result Repository* instead of the *Test Engine*. The resolved ATD item ATD-3 is shown in Table 9.6, which only depicts the updated elements comparing with the ATD-3 in Table 9.5.

Table 9.3 An ATD Item Identified from Design Option 1 of Architecture Decision AD-12

ID	ATD-10
Name	Compromised modifiability due to using XML files to store test results
Date	10-11-2012
Status	Unresolved
Incurred by	AD-12: storing test results in XML files (design option 1)
Responsible	Tom
Compromised QA	Modifiability
Rationale	If XML is used to store test results temporarily when the remote database server is inaccessible, extra functions to store and read the test results to and from XML files are needed. In addition, any new table added in the remote database requires a new XML schema.
Benefit	i. XML files are platform-independent, and they can be reused in the ATS when running in other operating systems. ii. The developers are experienced in developing and testing with XML, so that they do not need additional training, which saves time and cost.
Cost	$1 + 8.1 = 9.1$ person-days
Principal	1 person-day (this is the architecture redesign cost to resolving this ATD item, and there is no implementation cost of a design change during architecture design phase)
Interest	$5 \times 0.9 + 4 \times 0.9 = 8.1$ person-days

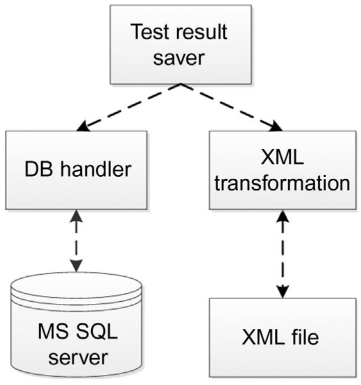
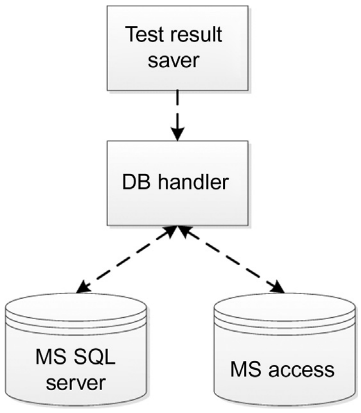
Change Scenarios	**#**	**Scenario description**	**Consequence**	**Scenario interest**	**Prob.**
	1	Synchronize the data in local storage files to the remote database server	Add functions to read the test results from XML files and store the test results to MS SQL Server	5 person-days	0.9
	2	Add new tables (around 10) to record the new test results in MS SQL Server	Design XML schemas and add functions to store the test results to XML files	4 person-days	0.9
Architecture Diagram	 Test result saver DB handler XML transformation MS SQL server XML file				
History	Created: 10-11-2012 by Tom				

Table 9.4 An ATD Item Identified from Design Option 2 of Architecture Decision AD-12

ID	ATD-11
Name	Compromised modifiability due to using MS Access to store test results
Date	10-11-2012
Status	Unresolved
Incurred by	AD-12: storing test results temporarily in local MS Access database (design option 2)
Responsible	Tom
Compromised QA	Modifiability
Rationale	If MS Access as a local database is used to store test results temporarily when the remote database server is inaccessible, any change of the table design in the remote database requires the according modification to the tables in MS Access. In addition, data in the local MS Access database need to be uploaded and synchronized to the remote database.
Benefit	i. DB Handler can be reused to store data to the MS SQL Server since both MS SQL Server and MS Access support the SQL standard. ii. The performance of MS Access is better than file-based storage methods (e.g., XML). iii. This design option can enrich the development team with the experience of using MS Access as a database, which may be helpful to other ATS projects.
Cost	$1 + 2.25 = 3.25$ person-days
Principal	1 person-day (same reason as described in Table 9.3)
Interest	$2 \times 0.9 + 0.5 \times 0.9 = 2.25$ person-days

Change Scenarios	**#**	**Scenario description**	**Consequence**	**Scenario interest**	**Prob.**
	1	Synchronize the data in local storage files to the remote database server	Develop functions to query test results from MS Access and test the functions	2 person-days	0.9
	2	Add new tables (around 10) to record the new test results in MS SQL Server	Add the new tables in MS Access and test them	0.5 person-days	0.9

Architecture Diagram



History Created: 10-11-2012 by Tom

Table 9.5 An ATD Item Identified from Architecture Decision AD-8

ID	ATD-3
Name	Compromised evolvability due to dealing with test results in the *Test Engine*
Date	26-10-2012
Status	Unresolved
Caused by	AD-8: The Test Engine updates test results to *GUI component* and *Test Result Saver*
Responsible	Zengyang
Compromised QA	Evolvability
Rationale	The use of test results is heavily related to the most important component—*Test Engine*. Adding any new functionality that needs to use test results will result in modification and testing of the *Test Engine* and any functionality depending on the *Test Engine*.
Benefit	i. The reliability of test result persistence of the ATS is good since the test results are updated to the *GUI component* and the *Test Result Saver* immediately when the results are generated. ii. The performance of updating test results to the *GUI component* and *Test Result Saver* is good since the test results are updated directly to these two components without crossing intermediate components. iii. The ATS can reuse a component of a legacy system that employs a similar strategy to the *Test Engine* in the ATS; therefore, 6 person-days are saved.
Cost	$8 + 14.4 = 22.4$ person-days
Principal	8 person-days
Interest	$10 \times 0.9 + 6 \times 0.9 = 14.4$ person-days

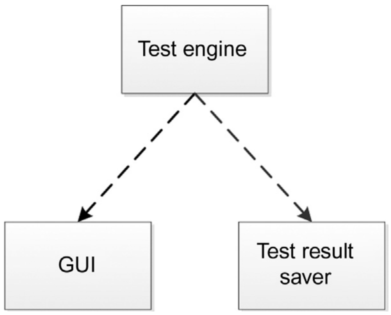
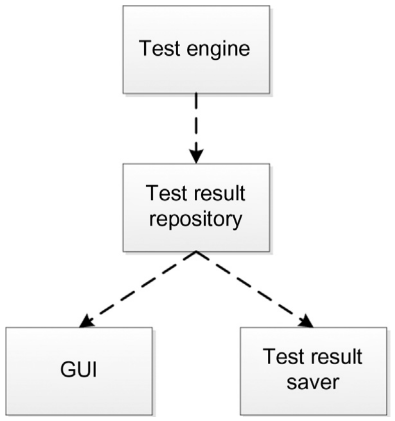
Change Scenarios	#	Scenario description	Consequence	Scenario interest	Prob.
	1	Add a Unit Under Test (UUT) visualization component	Have to modify the code in the *Test Engine* and test all functionalities depending on the *Test Engine*	10 person-days	0.9
	2	Generate a report for the just-finished test	Have to store the test results somewhere, such as a buffer	6 person-days	0.9
Architecture Diagram					
History	Created: 26-10-2012 by Zengyang				

Table 9.6 Resolved ATD Item ATD-3

ID	ATD-3
Date	26-11-2012
Status	Resolved
Responsible	Zengyang
Rationale	The solution to resolving ATD-3 is to add a *Test Result Repository* that stores the test results temporarily, as shown in the following architecture diagram. The *Test Engine* updates the latest test results to the *Test Result Repository*, and all the components that use test results will request the test results from the *Test Result Repository* instead of the *Test Engine*. In this way, the *Test Engine* is free from the direct uses of test results.
Architecture Diagram	 Test engine Test result repository GUI Test result saver
History	Created: 26-10-2012 by Zengyang Revised: 26-11-2012 by Zengyang

9.6 Related work

Value-oriented software architecting is an important area in value-based software engineering (Boehm, 2006), especially for architecture practitioners, since it explicitly considers economic aspects as a driven factor within the whole architecting process. Practitioners and researchers in the software architecture community have already put considerable effort into this area and have investigated value and economic impact in architecture design. Kazman et al. proposed the Cost-Benefit Analysis Method (CBAM) for architecture evaluation (Kazman et al., 2001), which models and calculates the costs and benefits of architecture decisions to assist architecture evaluation in a cost and benefit perspective. Both CBAM and architecture evaluation with DATDM evaluate architectural strategies from a cost-benefit perspective based on scenarios. The major differences between CBAM and architecture evaluation with DATDM are as follows: (1) CBAM evaluates the quality attribute benefit of an architectural strategy, while our approach evaluates both the nontechnical benefit (e.g., organizational benefit) and the quality attribute benefit of an architecture decision; (2) CBAM estimates the cost of implementing an architectural strategy, but our approach estimates the future cost of maintenance and evolution tasks, plus the implementation cost of an architecture decision; and (3) our approach considers the probability of a change scenario in the next release as a parameter when estimating the cost of an ATD item. Martínez-Fernández et al. presented a reuse-based economic model for software reference architectures (Martínez-Fernández et al., 2012). This economic model provides a cost-benefit analysis for the adoption of reference architectures to optimize architectural decision making. This model also estimates the development and maintenance benefits and costs of a specific product based on reuse of a candidate reference architecture, and the reference architecture with highest ROI (return on investment) is selected. With this model, the benefits and costs of a software architecture as a whole are calculated, while in our DATDM approach, benefits and costs are measured based on architecture decisions and incurred ATD items.

Architectural technical debt management is an emerging research area in software architecture. To date, little research has been conducted on technical debt management at the architecture level, and the scope of architectural technical debt is not clear (Kruchten et al., 2012). Nord et al. employed an architecture-focused and measurement-based approach to develop a metric to quantify and manage architectural technical debt (Nord et al., 2012). In their approach, architectural technical debt is modeled as rework, and the amount of rework caused by a suboptimal architecture design strategy is considered as the metric for architectural technical debt measurement. This approach "can be used to optimize the cost of development over time while continuing to deliver value to the customer" (Nord et al., 2012, p. 91). Measuring ATD incurred by different design paths in this approach provides a good way to estimate the ATD incurred by a group of architecture decisions.

9.7 Conclusions and future work

Architectural technical debt is an important element that needs to be considered in the architecting process, especially for value-oriented architecting, but currently it is seldom addressed. This chapter proposes an ATD conceptual model with an ATD item template for ATD management and integrates this conceptual model into the ATDM process in order to facilitate decision making and decision evaluation in a value-oriented perspective in architecture design. Working examples with a template in using ATDM in architecture synthesis and evaluation also provides architecture practitioners a ready-made solution for managing ATD in their architecting contexts. In a methodology perspective, the contribution of this work provides a controllable and predictable balance between the value and cost of architecture design in the long term.

Using ATDM in value-oriented architecting is a new research area, and the following directions need further exploration:

1. *ATDM theory*. A number of research questions remain: How to measure quantitatively the benefits of an ATD item? Is it possible to measure the benefits of an ATD item in a uniform metric, or is measuring ATD items necessary and helpful for architects to make decisions? How to define the criteria used to decide whether a specific ATD item should be resolved intermediately or left unresolved until later releases? Besides the cost of resolving an ATD item

and its interest related to change scenarios, how do other value considerations such as aesthetics, social, societal, and governance concerns influence the cost of an ATD item? What is the correlation between the metrics of ATD and risk, such as cost for ATD items and impact for risk?

2. *ATDM tool support*. What features should an ideal ATDM tool have? For example, an ATDM tool may better support architects with ATD item documentation, ATD monitoring in a dashboard, and visualization of the relationships between incurred ATD items and architecture decisions.
3. *Evidence*. We currently lack scientific evidence (e.g., academic or industrial studies through controlled experiments) on how ATDM can facilitate architecting. Empirical studies on using ATDM in architecting activities are needed.

Acknowledgments

This work is partially supported by AFR-Luxembourg under the contract No. 895528 and the NSFC under the grant No. 61170025. We would like to thank the anonymous reviewers for their valuable comments on the earlier version of this chapter.

References

Boehm, B., 2006. Value-based doftware rngineering: overview and sgenda. In: Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grünbacher, P. (Eds.), Value-nased Doftware Rngineering. Springer, Berlin, pp. 3–14.

Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., et al., 2010. Managing technical debt in software-reliant systems. Paper presented at the Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER′10), Santa Fe, New Mexico.

Buschmann, F., 2011. To pay or not to pay technical debt. IEEE Softw. 28 (6), 29–31.

Cunningham, W., 1992. The WyCash portfolio management system. Paper presented at the Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum). Vancouver, British Columbia, Canada.

Hofmeister, C., Kruchten, P., Nord, R.L., Obbink, H., Ran, A., America, P., 2007. A general model of software architecture design derived from five industrial approaches. J. Syst. Softw. 80 (1), 106–126.

ISO/IEC, 2011. Systems and Software Engineering—Systems and Software Quality Requirements and Evaluation (SQuaRE)—System and Software Quality Models. ISO/IEC FDIS 25010:2011, pp. 1–34.

ISO/IEC/IEEE, 2011. Systems and software engineering—Architecture description. ISO/IEC/IEEE 42010:2011 (E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471–2000), 1–46.

Jansen, A., Bosch, J., 2005. Software architecture as a set of architectural design decisions. Paper presented at the Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture (WICSA’05), Pittsburgh, Pennsylvania.

Kazman, R., Asundi, J., Klein, M., 2001. Quantifying the costs and benefits of architectural decisions. Paper presented at the Proceedings of the 23rd International Conference on Software Engineering (ICSE’01), Toronto, Ontario, Canada.

Kruchten, P., Nord, R.L., Ozkaya, I., 2012. Technical debt: from metaphor to theory and practice. IEEE Softw. 29 (6), 18–21.

Li, Z., Liang, P., Avgeriou, P., 2013. A systematic mapping study on technical debt. Under submission.

Martínez-Fernández, S., Ayala, C., Franch, X., 2012. A reuse-based economic model for software reference architectures. Departament d'Enginyeria de Serveis i Sistemes d'Informació, Universitat Politècnica de Catalunya, Barcelona, Spain.

Nord, R.L., Ozkaya, I., Kruchten, P., Gonzalez-Rojas, M., 2012. In search of a metric for managing architectural technical debt. Paper presented at the Proceedings of the 10th Working IEEE/IFIP Conference on Software Architecture (WICSA ′12), Helsinki, Finland.

Ozkaya, I., Kruchten, P., Nord, R.L., Brown, N., 2011. Managing technical debt in software development: report on the 2nd international workshop on managing technical debt, held at ICSE 2011. SIGSOFT Softw. Eng. Notes 36 (5), 33–35.

Seaman, C., Guo, Y., 2011. Measuring and monitoring technical debt. In: Zelkowitz, M. (Ed.), *Advances in Computers*, vol. 82. Elsevier Science, London, UK, pp. 25–45.

van Heesch, U., Avgeriou, P., Hilliard, R., 2012a. A documentation framework for architecture decisions. J. Syst. Softw. 85 (4), 795–820.

van Heesch, U., Avgeriou, P., Hilliard, R., 2012b. Forces on architecture decisions—a viewpoint. Paper presented at the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA ′12).

Zazworka, N., Seaman, C., Shull, F., 2011. Prioritizing design debt investment opportunities. Paper presented at the Proceedings of the 2nd International Workshop on Managing Technical Debt (MTD ′11), Waikiki, Honolulu, Hawaii.