HASARD: A Model-Based Method for Quality Analysis of Software Architecture

Hong Zhu

Department of Computing and Communication Technologies, Oxford Brookes University, Oxford, UK; email: hzhu@brookes.ac.uk

Qian Zhang

School of Computer, The National University of Defense Technology, Changsha, China; email: qianr.zhang@gmail.com

Yanlong Zhang

Manchester Metropolitan University, Manchester, UK; email: y.zhang@mmu.ac.uk

Abstract. HASARD stands for Hazard Analysis of Software ARchitectural Designs. It is a model-based method for the analysis of software quality as entailed in software architectural designs. In this method, quality hazards are systematically explored and their causes and effects are identified. The results of hazard analysis are then transformed into a graphical quality model of the system under assessment. Queries about quality related properties of the system can be answered automatically through a set of algorithms with the quality model as the input. Such queries include: (1) the relationships between various quality attributes as manifested in the particular design, (2) the design decisions critical to a given quality concern, (3) the impacts of a given design decision to quality attributes, (4) trade-off points in the design that balancing conflict quality concerns, etc. The chapter will present a prototype tool called SQUARE to support the whole process of HASARD analysis. It will also report a case study with a real software system to demonstrate the feasibility and usability of the HASARD method.

1 Introduction

An extraordinarily powerful means of software engineering is architecture analysis, which enables the early prediction of a system's quality at design stage. As Bass, Clement and Kazman (1998; 2012) pointed out, "*without them, we would be reduced to building systems by choosing various structures, implementing the*

system, measuring the system's quality, and all along the way hoping for the best". This chapter presents such a method for analysing software architectural designs to predict their quality.

1.1 Motivation

Software architecture is a structural model of software system at a very high level of abstraction. In this structural model, a software system is represented as a collection of components interconnected through connectors (Shaw & Garlan, 1996), (Bass, Clements, & Kazman, 1998), (Bosch, 2000), (Hofmeister, Nord, & Soni, 2000), (Taylor, Medvidovic, & Dashofy, 2010). Being at a high level of abstraction, the implementation details of the components and the connectors are hidden whereas the focus is on their characteristic features and/or the assigned functionality. For a given software system, a number of architectural models can be constructed to represent different views of the system in order to achieve different engineering purposes.

For example, to demonstrate that a system correctly satisfies its functional requirements, a *conceptual view* depicts the system in terms of how functions are decomposed and assigned to conceptual components, and how these components are controlled and interact with each other through conceptual connectors. In order to provide instructions for further detailed design and implementation, a *module view* focuses on how the functionality of the system is mapped to the implementation environment, such as how the system uses the facilities and resources in the underlying software platform. It depicts software structures in terms of software modules and characterises them with implementation related features. In order to facilitate the construction, integration, installation and testing of the system, a *code view* describes how the source code that implements the system is organised. To assist organisational management of the development process, a *development view* depicts a software system in terms of a repository of software artefacts that are created, modified and managed by the programmers, maintainers, testers, etc. In order to understand the dynamic behaviour of a system, a *behaviour view* (or *execution view*) describes the system's execution in terms of its runtime platform elements, concurrency, communication and the physical resources used, etc.

One of the most important engineering purposes of software architecture is to predict the system's quality. To achieve this purpose, a *quality view* manifests the architectural design decisions made specifically to address quality requirements (Bass, Clements, & Kazman, 2012). Usually, a quality view builds on top of a conceptual view with the relevant elements characterized by properties that are either directly or indirectly related to quality attributes. Such properties are referred to as *quality-carrying properties* in the sequel. However, how to use such a quality view to predict software quality remains an open problem. This is the subject of this chapter.

1.2 Related Works and Open Problems

The approach reported in this chapter is model-driven: that is, we will first construct a quality model based on the architectural design, then to infer the quality of the system based on the model. In this section, we review the current state of art in the related areas and discuss the open problems.

1.2.1 Software Quality Models

Software quality is one of the most elusive concepts (Kitchenham & Pfleeger, 1996). A great amount of effort has been made since 1970s to define software quality models in order to understand the concept, to measure software systems' quality and to guide software development activities in order to improve software quality (Deissenboeck et al., 2009).

A software quality model defines a set of properties that are relevant to software quality and the relationships between them. Such a relationship can be either *quantitative* or *qualitative*. The former is the subject of research on software metrics. The latter is widely used in the literature on software quality models and often combined with metrics for the evaluation of software quality.

Existing qualitative software quality models fall into two types: hierarchical models and relational models.

A *hierarchical model* defines a set of aspects of software quality, then decomposes each aspect into a number of factors, which are in turn decomposed into a number of attributes, and further decomposed into several metrics, etc. In this way, the abstract notion of software quality is characterized by a set of quality related properties, which are organized into a hierarchical structure to express the positive relationships between them. Some properties are more abstract thus at a higher level of the hierarchical structure. Some are more concrete, even directly measurable in a quantitative way, thus at a lower level of the hierarchy. The lower level properties have a positive contribution to the parent property. Here, a positive relationship from property A to B in a hierarchical model means that improving a software system's quality on property A implies an improvement of its quality on property B. Typical examples of such hierarchical models include McCall model (1977), Boehm model (1978), ISO model (1992; 2012), Dromey model (1995), Bansiya-Davis model of OO software (2002), Goeb-Lochmann model for service-oriented systems (2011), Franke-Kowalewski-Weise model for mobile software (2012), etc. One of the main weaknesses of such models is that they are incapable of expressing negative relations between quality attributes.

A *relational model* overcomes this problem by defining a number of stereotypes of relationships between quality attributes, such as positive, negative and neutral relations. Typically, a positive relation between two quality attributes indicates that a software system that is good on one attribute will also be good on the other. A negative relation means that a software system is good on one

attribute implies that it is inevitably bad on the other. A neutral relation means that the attributes are not logically interrelated. Typical examples of relational quality models include Perry Model (1991) and Gillies Model (1992; 1997). There are also a number of such quality models of information systems (Zhang, 2005). Such models have significantly improved our understanding of software quality.

Both hierarchical and relational quality models can help software developers to improve software quality by providing guidelines to software development activities, such as in the elicitation of quality requirements. However, as Deissenboeck et al. (2009) pointed out, despite successes and standardization efforts, quality models do not live up to expectation and disappointed practitioners. In particular, they fail to take software structures into account (Dromey, 1995, 1996). Moreover, they are incapable of dealing with complicated relationships between quality attributes that are difficult to be stereotyped. They provide little help to the design of software systems, especially the architectural design. These problems will be addressed in this chapter by proposing a new quality modelling approach, that is, graphic quality modelling.

1.2.2 Quality Analysis of Software Architecture

Two types of quality analysis activities are often performed at software design stage: *quality assessment and evaluation* and *quality exploration*.

- *Quality assessment and evaluation* aims at evaluating software architecture with regard to a set of well-defined criteria and in a set of well defined usages or scenarios. It assesses the software to determine whether the design meets the given design goals.
- *Quality exploration* intends to explore the architecture in order to discover problems in a design, where the problems are normally unknown and there are no fixed criteria and/or usage scenarios.

Each of these types of quality analysis has its own uses in software development. The former has its value in software validation and verification against known users' quality requirements, while the latter manifests its importance to software engineers by discovering and predicting unintended side-effects of design decisions. This is particularly important for software that has large user bases and long lifespan such as systems software, utility software, etc., where usages are complicated and may change significantly in the future.

In the past decade, a significant progress has been made in the research on the quality analysis of software architectures. A number of methods have been advanced in the literature to evaluate and assess the quality of software architectural designs. The existing architecture quality analysis techniques and methods can be classified into two types: *scenario-based*, and *model-based*.

- *Scenario-based methods* examine software architecture in the context of a set of scenarios, although the ways that scenarios are elicited and used vary.

Among the most well-known scenario-based methods are SAAM (Bass, Clements, & Kazman, 1998, 2012; Kazman et al., 1996), ATAM (Clements, Kazman, & Klein, 2002), etc.; see (Dobrica & Niemela, 2002) for a survey, and (Babar & Gorton, 2004) for a comparison of them. They have been applied to the evaluation and assessment of architectural designs on various quality attributes, such as modifiability (Kazman et al., 1996; Kazman et al., 1994), usability (Folmer & Bosch, 2005), security (Alkussayer & Allen, 2010), etc. They have a number of advantages, including the examination of software behaviour in real situations, reduction of complexity of analysis through focusing on typical scenarios, etc. Each method is suitable for one specific quality attribute/factor. They are applicable to the evaluation and assessment type of quality analysis. However, there are difficulties to build an overall picture of the system's quality especially when there are intensive and complicated interactions between scenarios. The elicitation of a complete and representative set of scenarios is by no means a trial task, which currently still replies on brainstorming. The result of quality analysis heavily depends on the selection of the most representative scenarios as reported in practices, e.g. (Kostelijk, 2005). It was also perceived as complicated and expensive to use in industrial context (Woods, 2012).

- *Model-based methods* start the analysis of software architecture with the construction of a model of the software system and then systematically deduce the quality based on the model. Typical examples of such quality analysis techniques are used in performance evaluation, reliability estimation and security analysis. Such methods are good at exploring the unknown quality problems in a design. However, each of them aims at a single quality attribute, too.

Therefore, it is highly desirable to develop a method that enables software engineers to systematically explore software architecture in order to discover and predict quality problems, especially when multiple quality attributes are involved.

1.2.3 Hazard Analysis Methods and Techniques

For constructing quality models, we will adapt existing system hazard analysis methods and techniques that are widely used for safety engineering.

Hazard analysis techniques have been widely used in the development and deployment of safety critical systems. Originally, hazard analysis aims at systematically identifying, assessing and controlling hazards before a new work process, piece of equipment, or other activity is initiated. In such a context, a *hazard* is a situation in which there is actual or potential danger to people or to the environment. Associated with each hazard is a risk, which is the product of the likelihood of the event occurring and its consequences. Once the hazards are identified and analysed, safety requirements can be specified for each component. Risks can be avoided or reduced ultimately through technical design, management

and organizational means. Consequently, the safety of the system is improved (Leveson, 1995; Neumann, 1995; Storey, 1996).

Here, we adapt the methods of hazard analysis and extend the concept of hazard in order to cover all quality aspects besides safety. In our context, the word hazard has its widest meaning, which means any *situation that may cause harm as designed or due to a deviation from the design decision*. The more likely a hazard occurs and more serious is the consequences of the hazard, the higher risk, and thus the more important the corresponding quality attribute is with the system in question.

There are a number of hazard analysis techniques available in the literature of safety engineering. These techniques fall into two types: (a) *hazard identification* techniques and methods and (b) *cause-effect analysis* techniques and methods. The former aims at discovering hazards systematically while the later arms at finding out the causes and consequences of the hazards.

One of the most effective hazard identification technique is HAZOP (MoD, 2000), which has been adapted to the analysis of software safety. In order to deal with a wider range of quality attributes rather than just safety, in this chapter we will extend the method and interpret the technique in a wider context.

A typical example of cause-effect analysis techniques is the FMEA technique, which stands for *Failure Modes and Effects Analysis*. It is a mature and well-defined safety analysis technique. Its engineering principle and process are similar to what software engineers are familiar with. It progressively selects the individual components or functions within a system and investigates their possible modes of failure. It then considers possible causes for each failure mode and assesses their likely consequences. In the original FMEA, the effects of the failure of a component are first determined for the unit itself, and then for the complete system. Possible remedial actions are also suggested. It requires engineers to identify the potential failure modes, and to link each failure mode to their possible causes and the consequences. This process is also adapted with some minor changes in order to make it closer to what software engineers familiar with and more suitable for software engineering.

1.3 Overview of the Proposed Approach

This chapter presents a model-based method for exploratory quality analysis of software architecture. It aims at systematically analysing an architectural design through building a quality model for the system under scrutiny. The method consists of the following technical elements.

- *A graphical quality modelling notation*. In this graphical notation, detailed and complex relationships between quality attributes and/or design decisions can be represented in the context of a given architecture design.

- *A software hazard analysis method*. It identifies the potential hazards in the development, operation and maintenance of the software and the causes and effects of such hazards.
- *A quality model construction technique*. It transforms software hazard analysis results into a software quality model represented in the graphic notation.
- *A set of algorithms* for automatically analysing quality models. They are applied in order to identify critical quality issues, trade-off design decision points, the impacts of a design decision and the factors that influence a quality attribute, etc.
- *A software tool*. It supports the software hazard analysis process and the construction of graphic quality model, and implements quality analysis algorithms.

The proposed quality analysis process is illustrated in Figure 1, which consists of the following activities.

- *Hazard identification*: it identifies the potential quality hazards of the software system as the consequences of the software architectural design decisions. The result is a list of quality hazards.
- *Hazard cause-consequence analysis*: it recognises the cause-consequence relationships between software quality hazards and design decisions. The result is the causal relations between hazards. It may also result in additional hazards added into the list of hazards.
- *Hazard classification*: it associates each identified hazard to a quality attribute or a quality-carrying property that the hazard is concerned with. The result is a set of classified design hazards.
- *Quality model assembling*: it transforms the results of the above steps into a graphic quality model represented in the graphic quality modelling notation.
- *Quality concern analysis*: it analyses the graphic quality model constructed through the above steps to infer quality concerns of the whole system, such as to find the critical design decisions that affect certain quality attributes, to recognise the trade-off points for certain conflict quality attributes, to discover the consequences of a design decision on various quality attributes, etc.

\includegraphics{Figures/Fig01-ProcessModel}

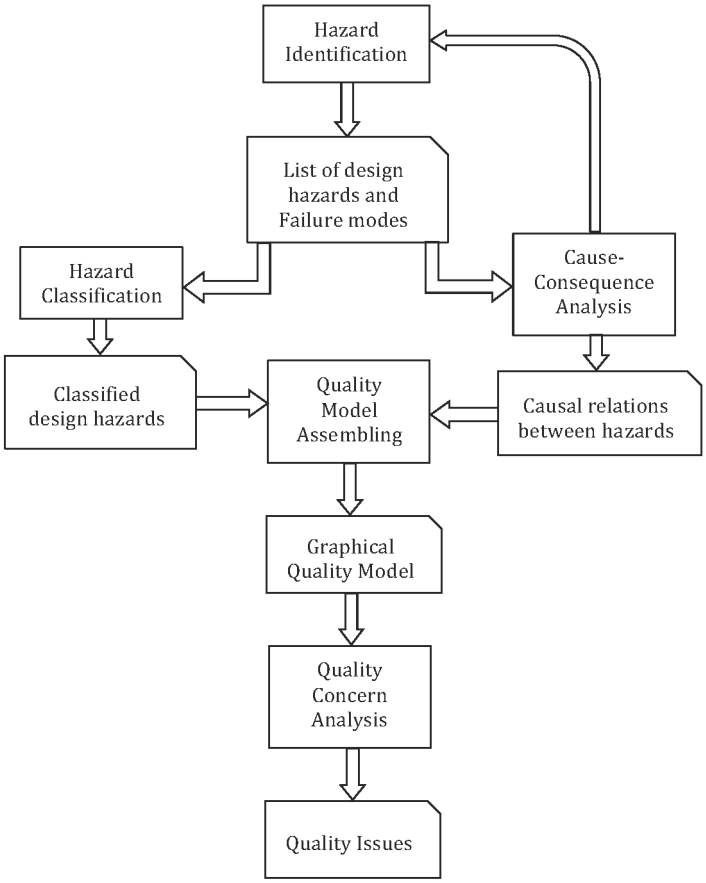


Figure 1. Process of Model-Based Quality Analysis

1.4 Organisation of the Chapter

The remainder of the chapter is organized as follows.

Section 2 is devoted to the hazard analysis of software systems. It is an adaptation and extension of the hazard analysis methods and techniques of system engineering.

Section 3 proposes the diagrammatic notations for graphical modelling of software quality and the technique for constructing such models from the results of hazard analysis of software architecture.

Section 4 presents the algorithms for the derivation of system quality features from graphic quality models.

Section 5 describes a software tool called SQUARE that supports the whole process of HASARD analysis and implements the quality analysis algorithms.

Section 6 reports a case study with the method and the tool.

Section 7 concludes the chapter with a comparison with related work and a discussion of future work.

2 Hazard Analysis of Software Architectural Designs

This section introduces the software hazard analysis method. It takes an architectural model of a software system as input. It systematically explores observable quality related phenomena of the system and then to establish the causal relationships between the phenomena. Here, a *phenomenon* is a property or event that is observable and concrete in the development, operation, maintenance and evolution of the software system. The process consists of two phases: the first is the identification of design hazards. The second is the discovery of causal relationships between such hazards and design decisions. The following subsections present each of these phases.

2.1 Identification of Design Hazards

The process of hazard analysis starts with the identification of hazards. In systems engineering disciplines, one of the most effective methods of hazard identification is the so called *Hazard and Operability Studies* (or *HAZOP* for short) (MoD, 2000). The method relies on asking and answering questions of *what-if* nature.

For example, in the analysis of a design of a chemical reactor, a typical what-if question is: *What would happen if temperature in the reaction container is higher than the designed range*? An answer to this question could be: "*The reactor will explode*". Consequently, the engineer will identify that controlling the temperature is a critical issue in the design of the reactor.

Similarly, such what-if questions are effective to identify quality issues in a software architectural design. For example, one may ask the what-if question: "*What would happen if the authentication of the user is not checked before granting an access to the database*?" An answer could be: "*The integrity of the database will be broken*". In such a way, the importance of authentication to system security can be identified.

The effectiveness of HAZOP method heavily depends on systematically asking such what-if questions. In HAZOP technique, this is achieved by designing and applying a set of *guide words* to cover all aspects of the design in question.

For example, "*more*" and "*less*" are typical examples of guide words that are applicable to a quantitative attribute in the design of a system. Applying such a guide word to a quantitative attribute prompts the engineer to ask the what-if

questions that what will happen if the actual value of the quantitative attribute is *more* than the designed value, or *less* than the designed value. For instance, suppose that a design decision of a chemical reactor is that "*the temperature of the reaction container is to be controlled in the range of 650ºC ±10ºC*". The what-if questions corresponding to these two guide words are:

- "*More*": What will happen if the temperature of the reactor container is *higher* than 650ºC + 10ºC?
- "*Less*": What will happen if the temperature of the reactor container is *lower* than 650ºC – 10ºC?

The same method can be applied to the identification of software hazards.

For example, the guide word *No* can be applied to a type of data produced by a component in the software system under analysis. In such a context, it means *no data* is produced by the component. In hazard identification, the analyst will ask the what-if question that "*what would happen if the data is not produced by the component*?"

Worth note: first, each guide word may be applied to more than one type of design features and architectural design decisions. Its meaning depends on the type of design decisions and the context in the system. For example, the same guide word "*No*" can also be applied to an architectural component. In such a context, it means that the component is not contained in the system. For example, a dynamic link library is not available when the system is initialised at run-time. A what-if question can be asked to identify the consequences of missing such a component.

Second, not all what-if questions will lead to a hazard. However, systematically asking all such what-if questions reduces the chance of omitting serious hazards during hazard identification.

In order to identify hazards hidden in a software architectural design, we have developed a set of guide words for software engineers to systematically develop a collection of what-if questions. They are applied to the elements in an architectural design, where a design element can be a property or function assigned to a component and/or a connector, the component or connector itself, the configuration of the system, etc. If a deviation from the design is credible, the corresponding behaviour of the element is considered as a potential hazard. Then, its causes, effects and recommendations of the further investigation are documented. Table 1 lists the guide words for analysing software architectural designs.

Table 1. Guide Words for Software Hazard Identification

Guide	Applicable Attribute	Interpretations
No	Data or control signals	No data or control signals are exchanged through a connector
		No data or control signals produced by a component
		No data or control signals received from input

	Property or function of a component or connector	The component (or connector) does not have the designed property (or function)
	Component or connector	The system does not contain the component (or connector)
More	Quantitative parameters of a component, connector or the whole system	The value of the parameter is too large
Less	Quantitative parameters of a component, connector or the whole system	The value of the parameter is too small
As well as	Event or activity	The intended event (or activity) occurs, but another event (or activity) also occurs in addition to this. For example, redundant data are sent to the designated receiver in addition to intended value.
		Data are sent to the designated receiver as well as an unintended receiver
	Property or function of a component or connector	In addition to the intended property (or function), the component (or connector) also has other additional properties (or functions)
	Component or connector	In addition to the intended component (or connector), the system contains other components and connectors
Part of	Structured data	Only a part of the data produced, stored or received
	Structured events	Only a part of the events happened
Reverse	Direction of information flow	The information flows in the opposite direction
	Event	The opposite event happened
Other than	Data or control signals	Incorrect data or control signals produced
	Quantitative and qualitative parameters	The parameter has a value different from the designed one
	Property or function of a component, connector or the whole system	The component (or connector, or the whole system) has a property (or function) different from the designed one
	Component or connector	The component (connector) is replaced by another kind of component (or connector). For example, a session bean is used instead of an entity bean
Early	Periodical events	The event happened earlier than expected
Late	Periodical events	The event happened later than expected
Before	Temporal orders between events	Two events happened in a different temporal order as designed

For example, consider the Internet connection between the client and server in a web-based application. As shown in Table 2, by applying these guide words, we can identify all of the most well-known hazards of web-based applications related to internet connection.

Table 2. Example of HAZOP analysis: The Application of Guide Words to Identify the Hazards of the Internet Connection between Client and Server

Guide	Hazard	Causes	Consequences
No	The internet connection passes no messages between the client and server.	Physically disconnected; Traffic jam; Software failure; Network server is down.	Client cannot communicate with the server.
More	More messages are delivered to the server than what the clients sent out, e.g. duplicated messages.	Hacker's attack; Heavy traffic on the Internet caused resending packages.	System clash; Overload on the server.
Less	Fewer messages are delivered than what the server (or the client) sent out, i.e. lost messages.	Discontinued Internet connection; Heavy traffic on the Internet; Software failure.	Incomplete transactions; System crash; Damage the data integrity of the server (and/or client).
As well as	Messages are delivered to other destinations in addition to the designated receiver.	Hacker's attack; Software failure.	Leak of sensitive information.
Part of	Only a part of the packets of a message is delivered to the destination client (or server).	Discontinued Internet connection; Heavy traffic on the Internet; Software failure.	Software failure; Production of incorrect computation results if incompleteness is not detected.
Other than	A message not from the client (or the server) is passed to the server (or client).	Hacker's attack; Other software system's failure	System failure; Damage the integrity of the data and the program.
Other than	The message is in a different format.	The client (or the server) is modified; Fault in the software.	System failure.

HAZOP study requires the analyst identifying not only the hazards, but also their effects and causes. Table 2 also shows the possible causes and the consequences of each hazard. Such analysis is preliminary, but indicates which hazard is important and deserves further investigation in cause-consequence analysis.

2.2 Cause-Consequence Analysis

The cause-consequence analysis of hazards aims at deepening the understanding of the hazards. For example, consider the hazard *more messages are delivered to the server than what the clients sent out* in Table 2. There are several possible causes of the hazard, which include that (a) the traffic on the Internet results in duplicated packages being generated, (b) a malicious source generates false request messages on purpose, etc. A direct consequence of a large number of false request messages is a high workload on the server. This, in turn,

could lead to (a) a poor performance of the system, even (b) the clash of the system. Consequently, the clients cannot get the requested services responsively. In the above analysis, we can see a sequence of hazardous phenomena or events is identified as a cause-effect chain.

The cause-consequence analysis can be performed in the backward or forward direction, or a combination of both. Forward analysis is from a hazard to search for potential effects, i.e. the consequences. Backward analysis starts with a hazard to search for its causes.

In forward analysis, the consequences of a hazard are identified and added into the list of identified hazards until the consequence is *terminal*. A hazard is terminal if it does not affect any other component of the system or does not cause any other hazards/failures. In many cases, we consider a hazard as terminal simply because we are not interested in its further consequence. For example, the phenomenon that a user cannot find required information could be considered as terminal, if we are not interested in what would happen afterwards. However, in certain context, what happens afterwards may become a serious problem. For example, suppose that the system store patients' medical records. The situation that a doctor cannot find a patient's record may lead to serious consequences in the treatment of the patient.

In backward analysis, the causes of a hazard are identified and added into the list of identified hazards until the hazard is *primitive*. A hazard is primitive if its causes cannot be further identified without additional knowledge of the system. A hazard can also be considered as primitive if we are not interested in its causes. For example, in most cases a broken link in a HTML file can be considered as primitive. However, in a different context, we may well be interested in its causes and want to find out why the hyperlink is broken.

The cause-consequence relationships between hazards can be recorded in a form as the results of cause-consequence analysis. They are used for the construction of a graphic quality model at the next step. Figure 2 shows the structure of the form with some examples of the hazards associated to web-based applications.

\includegraphics{Figures/Fig02-CauseConsequenceForm}

System: Webpage							
Analyists: HZ, QZ, YZ							
Date: April 14, 2009				**Version:** 1.0			
Ref	**Cause**			**Consequence**			**Explanation**
	Hazard Ref.	Description		Hazard Ref.	Description		
		Comp.	Phenomenon		Comp.	Phenomenon	
R1	W2	HTML file	Contains a broken link	U8	The user	Cannot download required HTML file	No file associated to the hyperlink
R2	S1	Web server	Server is down	U8	The user	Cannot download required HTML file	The file cannot be retrieved and transmitted
R3	U8	The user	Cannot download required HTML file	Y1	System	Cannot be used	The user is frustrated in finding required information

Figure 2. Cause-Consequence Analysis Form

Most hazards should have already been identified in the initial hazard identification step. However, some new hazards may still be discovered in cause-consequence analysis. These new hazards are then added into the hazard list and assigned with a unique identification number.

3 Graphical Modelling of Software Quality

In this section, we present the graphic notation for modelling software quality and the process of constructing such quality models based on hazard analysis.

3.1 Graphic Notation of Quality Models

As shown in Figure 3, our proposed diagrammatic representation of quality models is a directed graph, which consists of two principal elements: the *nodes* and *links*. Each node contains three basic elements:

1. a design element in the architectural design;
2. a quality-carrying properties of the design element; and
3. an observable phenomenon of the property.

where a design element can be a component, a connector, or any design decision. An observable phenomenon could be a *positive indicator* of the quality carrying property or a *negative indicator*. When it is a negative indicator, the property is marked with the symbol "−". The positive indicator is omitted and it is the default.

It is worth noting that although there may be more than one observable phenomenon that reveals a quality-carrying property for a design element, each node can only have one such phenomenon. This is to enable different causal relationships between phenomena to be easily represented without transforming the graphic model, and to enable the causal relationships to be validated separately.

The links are directed arcs between the nodes. Each link must contain an *impact factor*, which can be either *positive* or *negative*. A positive link from node *A* to node *B* means that the occurrence of the phenomenon of node *A* implies the occurrence of the phenomenon on node *B*. A negative link means that the occurrence of the phenomenon of node *A* prevents the occurrence of the phenomenon on node *B*. When a link has a negative impact, the arc is marked with the symbol "−" on the arrow. It is omitted when the impact factor is positive, and it is the default.

Each link can also contain an optional annotation for the reasons why the two nodes are related. This enables manual validation of the model.

\includegraphics{Figures/Fig03-GraphicalNotation}

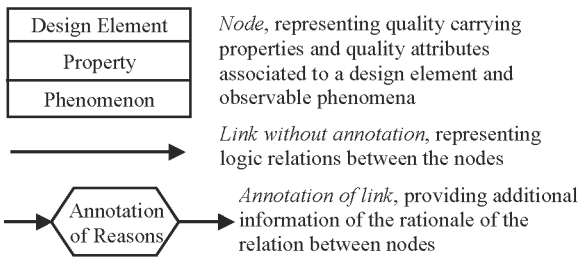


Figure 3. Graphic Notation of Quality Models

Figure 4 shows a fragment of a quality model of web-based information systems. This fragment of a quality model shows that the usability of a web-based system is related to the correctness of the HTML files, the load and performance of the server, the sizes of HTML files, the compatibility of client-side platform, and the availability of the online helps. It described in detail how these properties are related to each other and affect whether the user can find the required information.

\includegraphics{Figures/Fig04-ExampleOfQualityModel}

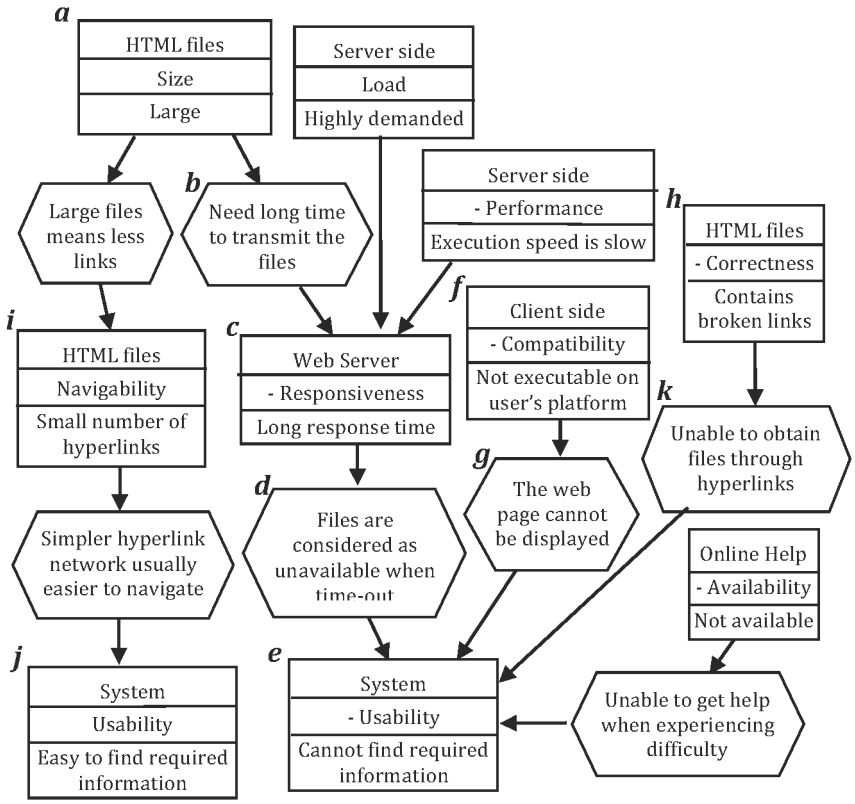


Figure 4. Example of Quality Model

For example, if the sizes of HTML files are large as shown in node *a* of Figure 4, the Web Server will need a long time to transmit a file from the server side to the client side. This results in Web Server's long response time as link *b* and node *c* shown. When the response time is longer than the time-out setting, the browser will regard the requested file as unavailable. This implies that the required information cannot be found, which is a negative phenomenon of the system's usability. This is depicted in Figure 4 by link *d* and node *e*. Therefore, using this quality model, we can infer that to achieve a good usability, the software designer should make each web page in a reasonable size to avoid excessive response time.

Similarly, Figure 4 also shows how the compatibility of the client side (node *f*) and the broken links in the HTML files (node *h*) will affect the usability of the system through links *g* and *k*, and so on.

It is worth noting that, the links between the nodes must be understood as the implications of one phenomenon to another, rather than simply the relationship between two quality attributes. For example, a collection of large sized HTML files may contain fewer hyperlinks between them than a collection of smaller sized files that contain the same information. This makes the navigation between

the files easier, which is a positive observable phenomenon of the navigability of the HTML files (node *i*). Consequently, the user may feel easier to find required information, which is a positive phenomenon of system's usability (node *j*). Therefore, the property that the HTML files are of large sizes is positively related to the usability of the system. On the other hand, as discussed above, large sized HTML files will increase the response time and in extreme cases it may cause poor usability. Such complexity cannot be represented in a quality model that only relates two abstract quality attributes as in hierarchical and relational models.

3.2 Construction of Quality Model

The construction of a quality model takes the information charted in the cause-consequence analysis form as input. Each hazard in the chart forms a node with the component and phenomenon as specified in the form. Each row in the chart forms a link from the node that represents the cause to the node that represents the hazard. The explanation column of the row forms the reason of the link. For example, from the first row in Figure 2, the nodes and link in Figure 5 are generated.

\includegraphics{Figures/Fig05-DeriveQualityModel}

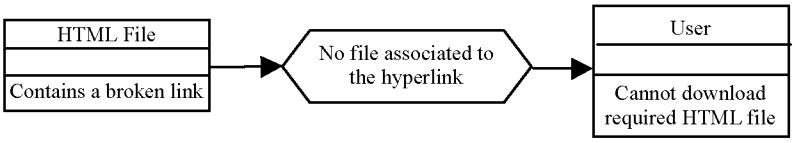


Figure 5. Example of Deriving Quality Model from Hazard Analysis

The nodes and links represent the hazards and their causal relationships are thus assembled together to form a diagram. However, such a diagram generated from a hazard analysis chart may be incomplete. The property slots of the nodes need to be filled in. In some cases, the quality-carrying property is the same as the property that the guide word is applied to. Otherwise, the observable phenomenon is compared with the definitions of a set of quality attributes and quality-carrying properties. The property that a phenomenon demonstrates is, then, identified; otherwise, a new attribute or property is recognized. This property is filled into the slot of the node.

For example, ``*An HTML file contains a broken hyperlink*" is a correctness issue of the HTML file, thus the quality attribute of the hazard is *correctness*. ``*Server is down*" is a problem of the reliability of the server, or more precisely, *availability*. ``*User cannot find required information*" is associated to the *usability* of the system. Therefore, from the hazard causal relationships given in Figure 2, we can derive the quality model shown in Figure 6.

\includegraphics{Figures/Fig06-SmallExampleQualitModel}

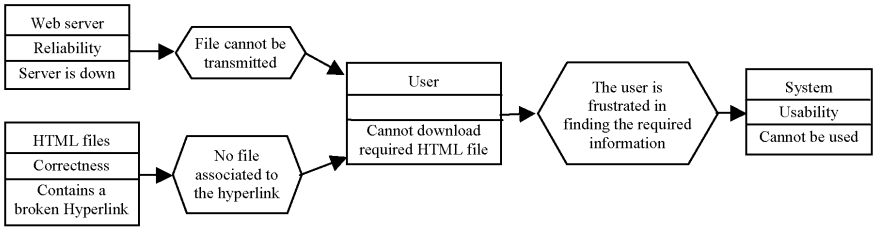


Figure 6. The Quality Model Derived from Figure 2.

4 Reasoning about Software Quality

In this section, we discuss various tasks of quality analysis at software design stage and how they can be supported by automated tools. The algorithms for such tool support are presented.

4.1 Contribution Factors of a Quality Attribute

In the analysis of a software architectural design, we often want to know how a quality issue is addressed. We want to know which components, connectors or the properties of the configuration are related to the quality issue and how they collectively provide the solution to meet quality requirements. The contribution factors of a quality attribute is a set of properties of the components and/or connectors or other design decisions such as the configuration of the architecture that affect the quality issue according to the design.

For example, consider the quality model given in Figure 4. We can derive the sub-graph shown in Figure 7 for the contribution factors of a server's responsiveness.

\includegraphics{Figures/Fig07-ContributionFactors}

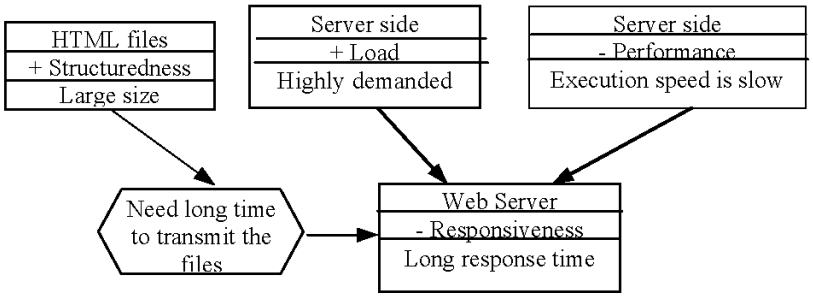


Figure 7. Factors Contributing to Server's Responsiveness.

This quality analysis task can be automatically performed by using the following algorithm. Note that, given a link *L* from node *A* to node *B* in a quality model, we say that node *A* is the *tail* of the link *L* and *B* is the *head* of *L*.

```
ALGORITHM  A1
    INPUT:
        QualityModel (* which is < NodeList, LinkList > *);
        Component; (* the name of the component *)
        QualityAttribute; (* the quality attribute *)
    OUTPUT:
        RelatedNodeList;
        RelatedLinkList;
    BEGIN
        RelatedNodeList := { };
        RelatedLinkList := { };
        FOR each node N in NodeList DO
            IF (N's component name ==Component) AND (N's property ==QualityAttribute)
            THEN add N into RelatedNodeList;
        END_FOR;
        REPEAT
            FOR each link L in LinkList DO
                IF (L's head is in RelatedNodeList) AND (L's head is not equal to L's tail)?
                THEN
                    IF L is not in RelatedLinkList
                    THEN Add link L to RelatedLinkList;
                    IF L's tail is not in RelatedNodeList
                    THEN Add L's tail to RelatedNodeList;
                END_IF
            END_FOR;
        UNTIL no more element is added into RelatedLinkList or RelatedNodeList;
        OUTPUT RelatedLinkList and RelatedNodeList;
    END
END_ALGORITHM.
```

Informally, Algorithm A1 searches for all the nodes and the links in the quality model that link to the nodes containing the component and its quality attribute that the user is interested in.

4.2 Impacts of Design Decisions

Another frequently asked question in the analysis of a software architectural design is ``*what are the consequences of a design decision*?" In such cases, we need to find out what are the quality attributes that are affected by the design decision. Such information can also be derived from a well constructed quality model.

For example, consider the quality model depicted in Figure 4. We can obtain the sub-graph shown in Figure 8 that represents the impacts of the quality-carrying property of HTML file's size on other quality attributes. It shows that the size of HTML files affects the navigability and responsiveness of the system, which in turn affects the usability of the whole system.

\includegraphics{Figures/Fig08-ImpactOfHazard}

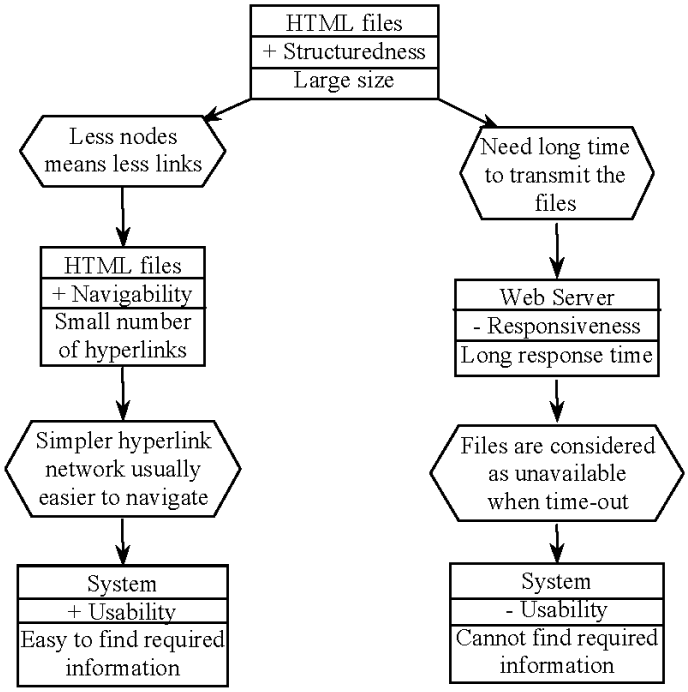


Figure 8. Example of the Impacts of a Design Decision on System Quality.

This analysis task can be automated by using the following algorithm.

```
ALGORITHM A2
    INPUT:
        QualityModel (* which is < NodeList, LinkList > *);
        Component (* the name of the component *),
        QualityAttribute (* the property of the component *),
    OUTPUT:
        EffectedNodeList;
        EffectedLinkList;
    BEGIN
        EffectedNodeList := { };
        FOR each node N in NodeList DO
            IF (N's component name ==Component) AND (N's property== QualityAttribute)
            THEN add N into EffectedNodeList;
        END_FOR;
        EffectedLinkList := { };
        REPEAT
            FOR each link L in LinkList DO
                IF (L's tail is in EffectedNodeList) AND (L's head is not equal to L's tail)?
                THEN
                    IF L is not in EffectedLinkList THEN Add link L to EffectedLinkList;
                    IF L's head is not in EffectedNodeList
                    THEN Add L's head to EffectedNodeList;
                END_IF
            END_FOR;
        UNTIL no more element is added into EffectedLinkList  or EffectedNodeList;
        OUTPUT EffectedLinkList and EffectedNodeList;
    END
END_ALGORITHM.
```

Informally, Algorithm A2 searches for the subset of nodes and links in the quality model to which the node(s) with user interested component and quality are directly or indirectly linked.

4.3 Quality Risks

A design decision may have positive as well as negative effects on a quality attribute. The negative effects may impose quality risks to the system. Therefore, it is often desirable to know what are the quality risks. This can also be derived from a quality model.

A negative effect of a design decision can be recognised by searching for the links and nodes in the quality model that have a negative effect on the quality attribute. Such a negative effect could be in one of the following two forms.

First, there is a negative indicator in a node while there is a positive influence factor on the link. In this case, if the phenomenon of the node is observed, the quality will be worse on that attribute.

Second, there is a negative influence factor on the link while there is a positive indicator on the node. In this case, the phenomenon that indicates a better value of the quality attribute will be prohibited to happen.

For example, in the quality model depicted in Figure 4, there is a link from the node HTML files with the property of large size to the node Web Server with a property of responsiveness. There is a positive influence factor marked on the link between the large size of HTML file and the phenomenon of long response time. This is because that the larger the HTML file size is, the longer the response time will be. Because the phenomenon of long response time is a negative indicator on usability, a large file size has a negative effect on usability. Therefore, a design decision of large file size is a risk to the quality attribute of responsiveness. The further effects of a quality risk can be identified and analysed. In certain cases, a negative effect, i.e. a quality risk, is not the consequence of a single design decision. Instead, it can be the consequence of a number of design decisions. In that case, all the causes must be identified so that a better design can be made. These causes can also be automatically derived from graphic quality models using the following algorithm.

```
ALGORITHM A3
    INPUT:
        QualityModel (* which is < NodeList, LinkList > *).
    OUTPUT:
        RelatedNodeList;
        RelatedLinkList;
    BEGIN
        RelatedNodeList := { };
        RelatedLinkList := { };
        FOR each node N in NodeList DO
            IF (N's influence indicator is negative)
            THEN add N into RelatedNodeList;
        END_FOR;
        FOR each link L in LinkList DO
            IF (L's influence factor is negative) AND (The indicator of L's head is positive)
            THEN add L into RelatedLinkList
        END_FOT;
        OUTPUT RelatedNodeList, RelatedLinkList;
    END
END_ALGORITHM.
```

Informally, Algorithm A3 searches for all nodes that have negative phenomena of quality attributes and implications that prevent positive quality phenomena.

4.4 Relationships between Quality Issues

An important question to be answered in quality analysis is the interrelationship between two quality issues. For example, how a server's performance is related to the system's usability? Answers to such questions can be found from the quality model by searching for all paths from a node that represents one quality issue to the nodes that represents the other quality issue. The algorithm for this purpose is given below.

```
ALGORITHM A4
    INPUT:
        QualityModel (* which is < NodeList, LinkList > *),
        Component1 (* the name of the first component *),
        Component2 (* the name of the second component *),
        QualityAttribute1 (* the first quality attribute *),
        QualityAttribute2 (* the second quality attribute *),
    OUTPUT:
        RelatedNodeList;
        RelatedLinkList;
    BEGIN
        RelatedNodeList := { };
        RelatedLinkList := { };
        Node1 := NULL;
        Node2 := NULL;
        TemptNodeList := { };
        TemptNode := NULL;
        FOR each node N in NodeList DO
            IF   (N's component name == Component1)
                 AND (N's property == QualityAttribute1)
            THEN Node1=N;
            ELSE IF  (N's component name == Component2)
                     AND (N's property == QualityAttribute2)
            THEN Node2 := N
            END_IF;
        END_FOR;
        Add Node1 to TemptNodeList;
        CurrentNode := Node1;
        Search(CurrentNode, Node1, Node2, QualityModel,
                TemptNodeList, RelatedLinkList, RelatedNodeList);
        OUTPUT RelatedLinkList and RelatedNodeList;
```

```
    END
END_ALGORITHM.
```

In Algorithm A4, the following function of depth first search is used.

```
FUNCTION Search (Component, Component1, Component2, QualityModel,
                    CurrentNodeList, ResultLinkList, ResultNodeList)
(* Depth-First Search *)
    BEGIN
        TemptNode=NULL;
        FOR each link L in LinkList that  L's head == Component DO
            Add L's tail to CurrentNodeList;
            IF L's tail == Component2
            THEN (* Find a path and record it*)
                TemptNode := L's tail;
                REPEAT
                    Add TemptNode to ResultNodeList;
                    TemptNode := TemptNode's previous node of CurrentNodeList;
                    Add link TemptL( whose head == TemptNode's Next node of
                        CurrentNodeList  AND whose tail == TemptNode)
                        to ResultLinklist;
                UNTILL TemptNode == Component1;
                Remove L's tail From CurrentNodeList;
            ELSE (* Depth first *)
                Search (L's tail, Component1, Component2, QualityModel,
                        CurrentNodeList, ResultLinkList, ResultNodeList);
            END_IF
        END_FOR;
        remove Component from TemptList;
    END_FUNCTION
```

Informally, Algorithm A4 uses a depth-first search algorithm to search for a sub-graph of the quality model that contains all paths between two input nodes.

4.5 Trade-Off Points

In many situations, a quality risk cannot be resolved without compromising on another or a number of other quality issues because these quality issues are conflicting with each other. In such cases, a trade-off between the quality attributes must be made and a balance between them must be achieved through appropriate design decisions.

For example, consider the quality model depicted in Figure 4. The size of HTML files positively affects the navigability of the hypertext network, but

negatively affects responsiveness of the web server. Therefore, navigability is in conflict with responsiveness. A trade-off between them must be made so that responsiveness is within a tolerable range while navigability is also acceptable. Such a trade-off occurs in the form of deciding on a suitable size of HTML file. In other words, HTML file size is a *trade-off point*.

From this example, we can see that a trade-off point is a node in the quality model that has a negative effect to one or more quality attributes and at the same time it has positive effects on one or more other quality attributes. Trade-off points can also be derived from quality models automatically. The algorithm is given below.

```
ALGORITHM A5
    INPUT:
        QualityModel (* which is < NodeList, LinkList >*);
     OUTPUT:
        RelatedNodeList (* the set of trade-off points *);
    BEGIN
        RelatedNodeList := { };
        TemptNodeList := result from calling A3;
        FOR each node N in TemptNodeList DO
            FOR each link L in LinkList AND ( L's head == N OR L's tail ==N) DO
                IF ( L's head != N AND (L's head's indicator == L's influence factor))
                THEN Add L's head to RelatedNodeList;
                IF (L's tail != N AND (L's tail's indicator == L's influence factor))
                THEN Add L's tail to RelatedNodeList;
            END_FOR
        END_FOR
        OUTPUT RelatedNodeList;
    END
END_ALGORITHM
```

Informally, Algorithm A5 first searches for those nodes of negative quality by employing Algorithm A3. Then for each of such nodes of negative quality, search for the nodes that links to this node of negative quality as well as a node of positive quality.

5 Support Tool SQUARE

To support the construction of quality model and the quality analysis of software architectural designs using HASARD method, we have developed a software tool called SQUARE, which stand for Software QUality and ARchitecture modeling Environment. It provides the following functions.

- Modelling software architecture in the visual notation proposed by Bass, Clement and Kazman (1998);
- Analysing software architecture models using HASARD method;
- Constructing software quality models in the graphical notation; and
- Reasoning about the system's quality using the quality model.

As shown in Figure 9, the SQUARE tool consists of the following components.

\includegraphics{Figures/Fig09-SQUAREStructure}

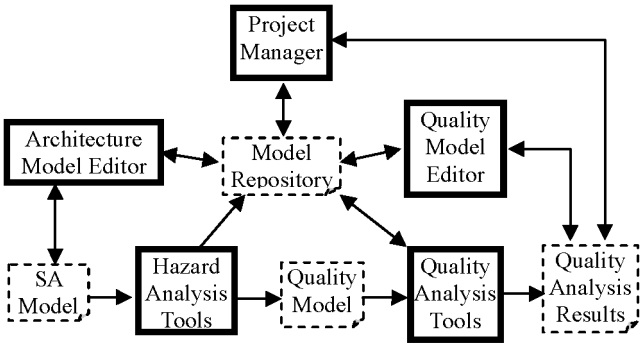


Figure 9. Architecture of SQAURE

- The *Architecture Model Editor* supports software architecture modelling through an interactive graphical user interface and represents software architectural models in the *Software Architecture Visual Notation* proposed by Bass, Clements, and Kazman in (1998). Figure 10 shows the graphic user interface of the architectural modelling tool.

\includegraphics{Figures/Fig10-SAModelingInterface}

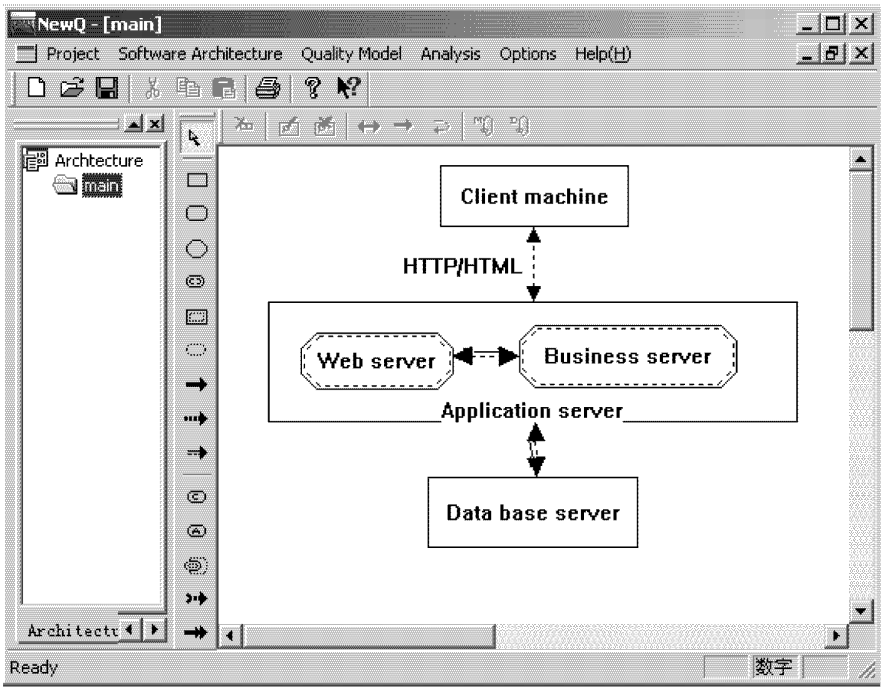
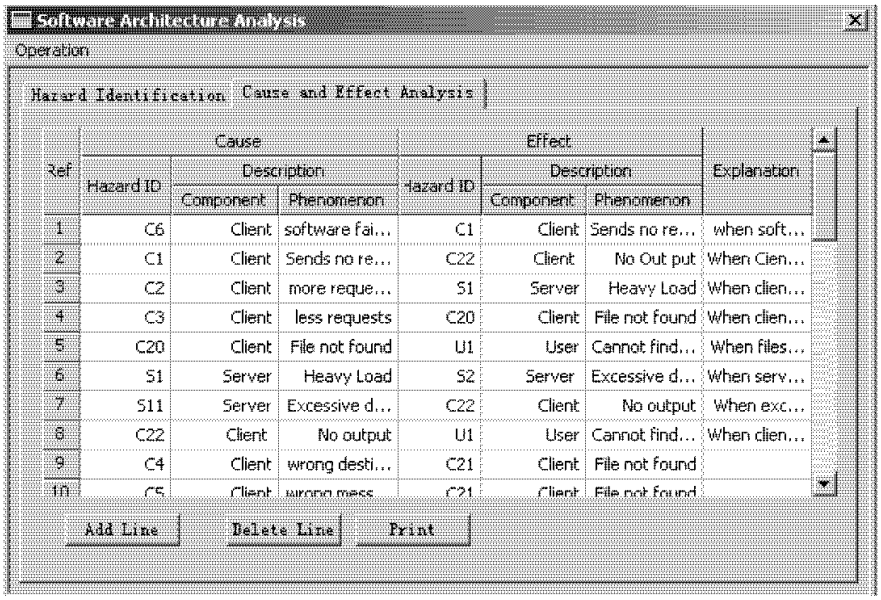


Figure 10. Interface of Software Architectural Modelling

- The *Hazard Analysis Tools* help the developers to analyse software architecture using HASARD method. It records the analysis results and automatically transforms them into the graphic representation of quality models. It consists of three tools. The *hazard identification tool* helps the users to apply guide words to various attributes of components/connectors in software architecture models so that hazards are systematically identified. The *cause-consequence analysis tool* helps the user to identify the causal relationships between the hazards. The *quality model generation tool* automatically transforms the results of hazard analysis into a quality model in graphical notation. Figure 11 shows the interfaces of the hazard analysis tools.

\includegraphics{Figures/Fig11-CauseEffectToolGUI}



Ref	Cause			Effect			Explanation
	Hazard ID	Description		Hazard ID	Description		
		Component	Phenomenon		Component	Phenomenon	
1	C6	Client	software fai...	C1	Client	Sends no re...	when soft...
2	C1	Client	Sends no re...	C22	Client	No Out put	When Clen...
3	C2	Client	more reque...	S1	Server	Heavy Load	When clien...
4	C3	Client	less requests	C20	Client	File not found	When clien...
5	C20	Client	File not found	U1	User	Cannot find...	When files...
6	S1	Server	Heavy Load	S2	Server	Excessive d...	When serv...
7	S11	Server	Excessive d...	C22	Client	No output	When exc...
8	C22	Client	No output	U1	User	Cannot find...	When clien...
9	C4	Client	wrong desti...	C21	Client	File not found	
10	C5	Client	wrong mess	C21	Client	File not found	

Figure 11. Graphic User Interface of the Cause-Consequence Analysis Tool.

- The *Quality Model Editor* provides an interactive graphical user interface to the users for the display and modification of software quality models.
- The *Quality Model Analysis Tools* automatically recognize and identify the quality features of the software designs from a quality models when invoked by the user. The results of the analysis are also displayed as a diagram in the graphical notation of software quality models.
- The *Model Repository* stores the information about the architecture and quality models as well as the quality analysis results to enable them to be managed within a project and reused across different development projects.
- The *Project Manager* provides a graphic user interface to software engineers for managing the artefacts used and generated in the quality analysis process through accesses to the Model Repository.

6 Case Study

A case study has been conducted with a real e-commerce application to evaluate the usability of the approach. This section reports the main results of the case study.

6.1 Research Questions

The research question addressed in this case study is to test whether the proposed method and the supporting tools are practically applicable. This research question is decomposed into the following sub-questions:

- Can a quality model be constructed for a real-world software system with acceptable effort?
- How complex will a quality model be for a real software system?
- Can the quality model for such a real software system adequately cover the quality issues that a developer would be interested in?
- Can the automated quality analysis technique and the support tools derive the right quality predictions from the quality model?
- How well will the predictions of quality problems by the HASARD method match the real situation?

6.2 The Object System

The object of the case study is an e-commerce system for online trading of medicine. The system is operated by the Medicine Trading Regulation Authority of the Hunan Province, P. R. China, to supply medicines to all state-owned hospitals in the province. Its main functions include (a) customer relationship management, (b) product catalogue management, (c) online trade management, (d) online auction of medicine supply bids, (e) order tracking and management, (f) advertisement release, (g) a search engine for medicine information, and so on. The system was implemented in the J2EE technology.

The system includes the following functional components.

- *Management Component*: it supports the management activities, including the management of information release, trading centres, users' membership, manufacture membership, permission of trade and/or production of medicine, and log information of online activities.
- *Content Management*: it manages information contents stored, processed and displayed by the system, such as medicine catalogues, prices, geographical information, and sales information, etc.
- *Online trading*: it provides an interface and facilities for online trading activities and the links to other information contents such as catalogue, products information, and contract templates.
- *Public Relationship*: it maintains the public relationship between the organisation and its various types of customers, including sending out invitations to the public to bid on auctions, and so on.
- *Order tracking*: it provides the interface and facilities to track the business process of each deal.

- *Communication Management*: it provides the secure communications facilities for sending messages, and manages the mails sent and received by the system.
- *Report generation*: it answers queries from managers about various statistical data of the online trading and generates financial reports.

The case study was conducted after the object system was released and in operation for more than 1 year. However, the problems in the operation of the system were not revealed to the analysts involved in the case study before the predictions of the system's problems were made. This enables us to see how well the result of quality analysis matches the reality.

6.3 Process of the Case Study

The case study consists of the following activities.

- *Construction of the architectural model of the system through reverse engineering*. The system's design and implementation were fairly well documented. The accesses to the chief developers were available. The design documents as well as parts of the source code were reviewed. An architectural model of the system was constructed, which was reviewed by some of the chief developers of the system for approval of its accuracy. Figure 12 shows a part of the architectural model for the user management sub-system.

\includegraphics{Figures/Fig12-CaseStudySA}

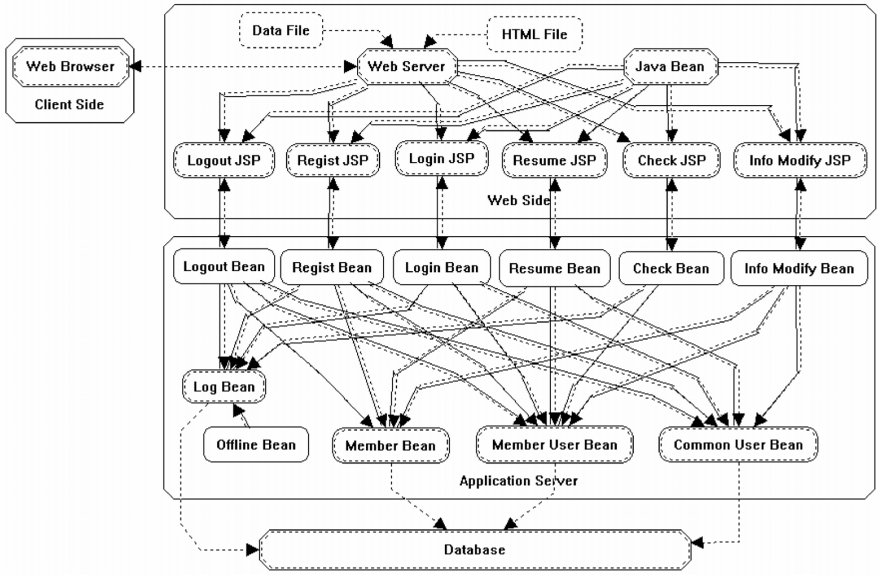


Figure 12. The architecture of the Case Study System

- *Application of HASARD method and construction of quality model*. The architectural model of the system was then analysed using the HASARD method. The quality hazards of the system were identified. The cause-consequence relationships between the hazards were recognised. The information was then transformed into a quality model in the graphical notation. The quality model contains 70 nodes and 64 links between the nodes. For the sake of space, the details of the quality model are omitted in this paper.
- *Analysis of the quality model*. The quality model developed in the previous step was analysed by applying the SQUARE analysis tools to identify quality risks, quality trade-off points, and to derive the impacts of design decision on certain quality attributes and the contribution factors to certain quality attributes. More details are given in the next subsection.
- *Validation of analysis results*. The results obtained from quality analysis of the system were fed back to the developers of the e-commerce system. A workshop was run to validate whether the outcomes of the quality analysis matches the reality in the development and operation of the system. It was found that all our findings were consistent with what has observed in the operation of the system. Some of the phenomena observed in the operation of the system were first time satisfactorily explained through the architecture and quality models of the system. Based on the analysis results, a number of specific suggestions on the improvement of the system's architecture were made. Some of them were taken by the development team in the development of the new release of the system. Some would result in major changes of system's architecture and regrettably cannot be implemented within the budget of the new releases.

The following provides some details of the main findings of the case study to illustrate the kinds of quality issues that the method can discover and the kinds of analysis activities that the automated tools can support. Some sensitive issues related to the privacy of the system are carefully removed because the main purpose here is to validate our quality analysis method and tool.

6.4 Main Results of Quality Analysis

In the case study, we discovered a number of quality issues of the system. The following are some examples of the discovered quality issues.

- *Critical quality attributes*. One observation made during the operation of the system is that the some users complained that they cannot find desired information. In the case study, we analysed the causes of the problem by finding the factors that affect the system's usability. The tool generated a sub-diagram that contains 35 nodes out of the 70 nodes in the quality model. This means that most components affect usability of the system. Consequently, we concluded that usability is a very sensitive quality issue in the design of the

system. The generated sub-diagram provided detailed information about how properties of various components affect the usability of the whole system. Our case study provided useful guides to the developers for how to enhance the usability.

- *Contribution factors of a quality attribute*. Intuitively, the server's availability is of particular importance to a number of other quality attributes. To find out what are the factors that affect server's availability, we applied the tool and generated the sub-diagram shown in Figure 13. The diagram shows that the factors that affect this quality attribute include hardware reliability, software reliability, power supply, system security, and maintenance. Therefore, we can conclude that necessary measures must be adopted to prevent hackers from attacking the server, to ensure a reliable power supply and the stability of server's hardware and software system to avoid the server crashes, and to implement tools to enable online maintenance in order to reduce the time that the system has to be shut down for maintenance tasks.

\includegraphics{Figures/Fig13-CaseStudyQAResult1}

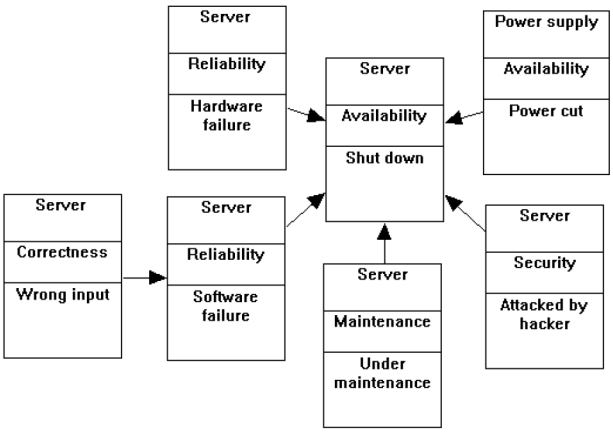


Figure 13. Example 1 of QA Results in the Case Study: factors affecting server's availability

- *Relationships between two quality attributes*. Our quality model can help us to understand the relationships between quality attributes. For example, the quality model demonstrated that usability of the client side is affected by performance of the web server. So we must consider carefully about the system's hardware configuration and the deployment of software components onto the hardware cluster to balance the communication and computation workload according to the operation profiles.
- *Quality trade-off points*. In the case study, quality trade-off points were also identified. For example, we found that the size of HTML files is a trade-off point. Because when the size is large, it has two different impacts on other quality attributes. One side, the HTML files of large sizes will make users find necessary information through fewer clicks. On the other side, the HTML files

of large sizes also make the response time longer. Both of these are related to the usability of the system, but one has positive impact while the other is negative. Therefore, it is a trade-off point. Another trade-off point identified in the case study is the granularity of session beans. A small-sized session bean can only implement relatively simpler functions in comparison to larger sized session beans. Therefore, to complete a task, smaller session beans need to invoke more methods of other beans. This results in more execution time to complete a task. Consequently, the performance of the whole system declines due to the time spent on creating instances of session beans. On the other hand, if session beans are of a larger size, to serve the same number of clients, more memory will be consumed. Therefore, the granularity of session beans is a trade-off point between the response time of the system and the consumption of the memory space.

- *Impacts of a design decision*. As discussed in the previous sections, the impacts of a design decision can be easily identified by using our quality model. In the case study we derived a large amount of such information. For example, if the component of internet has heavy traffic, the usability and performance of the whole system will be affected.
- *Key quality issues*. In the analysis of the impact of a quality attribute, quality risk points and critical quality issues can be recognised if the quality attribute has significant impacts on a wide range of other quality attributes. For example, in the case study, we found that the impacts of database's performance are extensive as shown in the sub-diagram in Figure 14, which is created by the SQUARE tool. It has the impact on a wide range of issues ranging from business layer to presentation layer. So it is necessary to take some measures to avoid vicious attack and to ensure the stability of hardware and software of the database server.

\includegraphics{Figures/Fig14-CaseStudyQAResult2}

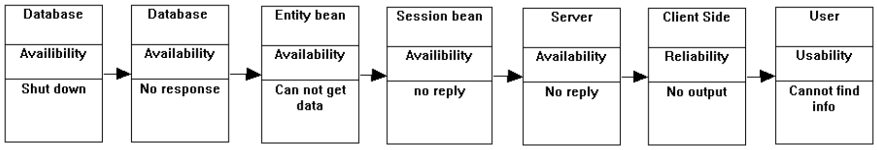


Figure 14. Example 2 of QA Analysis in the Case Study: Impact of Database Failure

6.5 Conclusions of the Case Study

From the findings of the case study, we answer the research questions asked in Subsection 6.1 as follows.

- *Can a quality model be constructed for a real-world software system with acceptable effort?*

In the case study, a quality model was successfully constructed by applying the HASARD method with the assistance of the automated tool SQUARE. Two persons (one PhD student and one MSc student of Computer Science) worked on this quality model construction task. It took them 1 month, including the reverse engineering effort. These two analysts are familiar with the method and the uses of the tools before they started the case study. We can draw the conclusion that quality models for such real-work software can be constructed with reasonable efforts.

It is worth noting that the reverse engineering including the construction of the architectural model of the system takes about half of the time. It lays a foundation for the construction of the quality model. It is necessary to have a good understanding of the design of the system and to have good domain knowledge of the application. Hazard analysis of a design heavily depends on such knowledge as well as the knowledge about how the software system operates.

- *How complex a quality model will be for a real world software system?*

The quality model of the system consists of 70 nodes, which is the largest and the most complicated quality model that we have ever seen in the literature. However, we found that the model is readable and its validity was validated by the developers of the system by checking the validity of the information contained in each node and link. It is worth noting that we believe that a quality model of such a scale cannot be constructed without a structured method. Our HASRAD method is capable of handling such a complexity and scale due to the structured approach based on hazard analysis techniques and the tool support.

- *Can the quality model for such a real world software system adequately cover the quality issues that a developer would be interested in?*

Our case study found that the quality model adequately covered all quality attributes in existing quality models. Quantitative analysis of the quality issues were not covered because the original requirements and design document do not contain such information. Whether the proposed method can deal with quality issues quantitatively remains an open question that is not tested by the case study. It is one of the main limitation of the case study.

- *Can the automated quality analysis technique and the support tool derive the right quality concerns from the quality model?*

As reported in the previous subsection, the tool SQUARE was used to derive the quality concerns for each type of the quality concerns discussed in Section 4 and produced meaningful output. The use of SQUARE tool to derive quality concerns after the construction of the quality model took little effort, much less than 1 week by two analysts. For each query input to the tool, SQUARE responses

within 1 second. The main time that the analysts spent on is reading and interpreting the output.

- *How well the predictions of quality problems by the HASARD method match the real situation?*

The main findings reported in the previous subsection in the quality analysis of the system were all checked by the developers of the system. All these findings were confirmed and agreed by the development team who are also responsible for the maintenance of the e-commerce system. Note that the quality problems of the system in the operation were unknown to the analysts before the predictions were made. Therefore, we can conclude from the case study, the predictions made by analysing the quality model by employing our method were correct.

During the validation workshop of the case study, the development team were also asked if they have any quality problem observed in the operation of the system that were not predicted by the analysts. They answer was negative. Therefore, we can conclude that the predictions were also precise in the sense there were no serious issue missed in the quality analysis.

7 Conclusion

In this section, we conclude the chapter with a comparison of our approach with the related work and a discussion of the limitations and open problems that deserve further research.

7.1 Comparison with Related Work

The work reported in this chapter is related to three groups of research: (a) software quality models, (b) hazard analysis methods and techniques, and (c) software architecture evaluation and assessment. The following subsections compare our approach to the related work in these three areas, respectively.

7.1.1 Software Quality Models

The development of software quality models can be back dated to 1970s such as Boehm model (1978) and McCall model (1977). Research on traditional quality models has been carried out in more recent years. Al-qutaish (2010) studied 5 hierarchical quality models, which are the McCall model, Boehm model, Dromey model, FURPS model (Grady, 1992) and ISO 9126 model. He compared the structure as well as the coverage of quality attributes in these models. Kayed *et al*. (2009) applied ontology extraction and analysis techniques to the definitions of

software product quality attributes. They studied 67 most commonly discussed software product quality attributes and concluded that ``there is a lack of consensus on the concepts and terminologies used in this field".

The weaknesses of such quality models discussed in Section 1.2 have been addressed successfully in our graphical quality modelling approach. In particular, graphic quality models take full uses of the knowledge of the system's structure, where a node in the quality model associates an architectural design element, including the components, connectors and configuration features, with an observable phenomenon of its quality carrying property. The complicated relationships between various quality attributes can be represented by multiple links between the nodes. Our case study shows that such a quality modelling approach can represent complicated quality models of real software system adequately.

In this paper, we demonstrated our graphical quality models as qualitative models. How to combine our graphical model with quantitative metrics is an interesting topic for further research. However, we believe that it should not be a major problem to include quantitative information in our graphic quality models. How to obtain and use such quantitative data in the analysis of software architecture is the key problem to be solved.

A closely related work on software quality modelling is the so-called activity-based approach proposed by Deissenboeck et al. (2007). In the activity-based approach, quality models are constructed based on two notions: the *facts* and the *activities*. A *fact* is a property of an entity in the system under consideration. It is represented in the form of [*entity* | *attribute*]. For example, the fact that "*a class C is complex*" can be represented as [*C* | *Complex*]. An *activity* is an action that can be performed on or with the support of the system under consideration. Typical examples of activities are "*attacking the system*" related to system's security, "*modifying the code of a class*" related to the modifiability, etc. In this approach, the quality of the system is manifested by how facts affect activities, where the impact of a fact on an activity can be either *positive* or *negative* depending on whether the fact contributes to the action positively or negatively. Therefore, the elements of a quality model are represented in the form of

$$[\text{entity} \mid \text{attribute}] \xrightarrow{+\mid-} [\text{activity}].$$

A model can therefore be depicted in the form of a two dimensional matrix where the entity-attribute pairs are the rows and the activities are the columns, the impacts are the cells in the matrix. The activity-based quality modelling approach was first proposed by Deissenboeck et al. (2007) for the quality of maintainability. It is generalised by Wagner and Deissenboeck (2007) and Lochmann and Goeb (2011), applied to security by Luckey et al. (2010), to usability by Winter et al. (2008), to service-oriented architecture by Goeb and Lochmann (2011), and combined with Bayesian network to assess and predict software quality by Wagner (2010). In 2011, Deissenboeck and Wagner et al. (2011; 2012) reported a

tool called Quamoco that supports the construction of such quality models. An evaluating of the Quamoco meta-model and the tool was reported in (Klas, Lampasona, & Munch, 2011).

Both activity-based approach (and its extensions) and our approach are concerned with the properties of entities in a software system. However, we further include phenomena as an important part of quality models. The main difference between the activity-based approach and our approach is that we emphasise the relationships between quality carrying properties while activity-based approach is concerned with how such properties affect the actions to be performed on the system. Thus, the complicated relationships between the quality attributes cannot be modelled in the activity-based approaches. More importantly, our method covers the model construction process and automated analysis of the models, while their work does not.

7.1.2 Hazard Analysis

Our software hazard analysis method is an adaptation of existing system hazard analysis methods and techniques for safety engineering (Leveson, 1995; Neumann, 1995; Storey, 1996).

In particular, we extended the concept of hazard in order to cover all quality factors besides safety in the construction of quality models of software systems. In our context, the word hazard has its widest meaning, which means any situation that may cause harm as designed or due to a deviation from the design decision.

One of the most effective hazard identification technique is HAZOP (MoD, 2000), which has already been adapted to analysis of software safety. Here, we further extended it for a wider range of software quality by re-defining the guide words.

From software engineering point of view, the original FMEA method has a number of weaknesses when applied to software systems. First, the original FMEA chart is ambiguous about which component causes the failure. However, it is important to clearly identify the component that causes the failure in order to enable the construction of a quality model of the software. Therefore, we modify the format of FMEA chart to include information about the component that causes the failure. Note that, by the word component we meant both software architectural components and connectors.

Our second modification to FMEA is that the indirect effects of a failure mode are not charted. There are two reasons for indirect effect not to be charted. First, we found that for a complicated software system, the indirect effects such as those at system level may not be so clear when a component fails. Second, information about indirect effects are redundant because they will be analysed subsequently as the effect of other failures. The system level effects of a component failure will eventually emerge from such a chain of cause-effect.

Our third modification to FMEA is that we also included an explanation column in the chart so that the reasons why a failure mode causes another can be

provided. This is for the purpose to enable the validation of the analysis results. Such reasons are usually obvious in physical systems, but we find that they are less obvious sometimes for software systems.

Finally, in addition to what hazard analysis techniques and method do, our approach also links the results of hazard analysis to quality attributes. In particular, each cause of a failure mode indicates a quality attribute that the developers are usually concerned with. The corresponding consequences of the failure mode indicate what quality attributes that the users are usually most concerned with. Note that, both causes and consequences of a failure mode are stated as observable phenomena of the system. The abstract quality attributes that a phenomenon manifested must be identified. Consequently, the relationships between the quality attributes or quality-carrying properties can be established through such concrete phenomena. This enables engineers to reason about quality at a high level of abstraction.

7.1.3 Evaluation and Assessment of Software Architecture

In the past decade, a significant progress has been made in the research on the analysis of software architectures. A number of methods have been advanced in the literature to evaluate and assess the quality of software architectural designs. Among the most well-known are SAAM (Kazman et al., 1996) and ATAM (Clements, Kazman, & Klein, 2002), etc.; see (Dobrica & Niemela, 2002) for a survey.

Existing scenario-based methods are for the assessment and evaluation of software quality as exhibited in architectural design. They examine software architectures in the context of a set of scenarios, although the ways that scenarios are elicited and used vary. The set of scenarios serves as the criteria or benchmark of the assessment and evaluation. A remarkable advantage of such methods is that the examination of software behaviour can be performed in realistic situations. Moreover, the complexity of analysis can be reduced through focusing on typical scenarios, etc. However, it is difficult to build an overall picture of the system's quality especially when there are intensive and complicated interactions between scenarios. The elicitation of a complete and representative set of scenarios is by no means a trivial task, which is currently still a brainstorming and negotiation process. The result of quality analysis may heavily depend on the selection of scenarios as reported in practical experiences (Kostelijk, 2005).

Existing model-based techniques focus on one single quality attribute, such as performance and reliability, etc., rather than relationships between various quality attributes. Moreover, they are mostly quantitative models.

Our method is fundamentally different from these existing works.

7.2 Limitations and Future work

The work reported in this chapter has some limitations and there are a number of open problems deserve further studies.

First, the quality models exemplified in this chapter are qualitative. We believe that the quality models can be extended to include quantitative data. For example, a phenomenon may occur with certain probability and the impact of one phenomenon on another can also be quantitative as in quantitative hazard analysis in systems engineering. The key open problem for quantitative quality modelling and quality analysis is how to use such quantitative data to predict software quality. This deserves further research and will significantly improve power of quality analysis of software architectures.

Second, architectural designs can be represented in two different approaches: (a) in the form of an architectural model, or (b) as a set of design decisions (Jansen & Bosch, 2005; Shahin, Liang, & Khayyambashi, 2009). The case study was carried out with an architectural model. Further investigation could be conducted to see how the approach can be adapted to analysis architecture represented in the form of a set of design decisions.

Another problem that is worthy investigating is to combine the activity-based approach with our approach. It will be interesting to see how hazard analysis can be applied to a software process model to derive the activities. Deriving activities related to a quality issue is a problem that has not been solved in the existing work of the activity based approach.

Moreover, the automation techniques can only be applied once a quality model is constructed. The quality model construction process depends on: (a) the knowledge of analysts; and (a) good definition/specification of the architecture. This model construction process is a structure process, but still a manual activity. Further tool support to this process is worth further investigating.

It is also worth further investigating to integrate existing architectural analysis methods and techniques with the approach presented in this chapter. For example, a hazard can be considered as a scenario. Can scenario-based approach be adapted and integrated with our approach? For example, can scenario identification and specification be used to replace hazard analysis? Or, on the other hand, can hazard analysis be used to replace scenario identification in scenario-based techniques? Hazard analysis techniques have the advantages of being a structured and systematic process, while scenarios are concrete and easy to understand.

Finally, when quality analysis of software architectural design is performed in the context of software evolution, it is of great importance to study how the method presented in this chapter applies when the system is restructured, re-engineered, or integrated to another system. The particular research questions that deserve further investigation include: Can quality models be incrementally modified? Can analysis algorithms be revised to be an incremental algorithm? etc.

Acknowledgments. The work reported in this chapter is partly supported by China High-Technology R\&D Programme (863 Programme) under the grant 2002AA11607. The authors are grateful to our colleague Dr. Sue Greenwood and Mr. Qingning Huo at Oxford Brookes University, UK, for their contribution to the research and Mr. Jian Wu of the National University of Defence Technology, China, for the participation in the development of the tools.

References

Al-qutaish, R. E. (2010). Quality Models in Software Engineering Literature: An Analytical and Comparative Study. *Journal of American Science, 6*(3), 166--175.

Alkussayer, A., & Allen, W. H. (2010). *A scenario-based framework for the security evaluation of software architecture*. In Proc. of The 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT 2010).

Babar, M. A., & Gorton, I. (2004). *Comparison of scenario-based software architecture evaluation methods*. In Proc. of The 11th Asia-Pacific Software Engineering Conference (APSEC 2004).

Bansiya, J., & Davis, C. G. (2002). A Hierarchical Model for Object-Oriented Design Quality Assessment. *IEEE Transactions on Software Engineering, 28*(1), 4--17.

Bass, L., Clements, P., & Kazman, R. (1998). *Software Architecture in Practice* (1st edition ed.): Addison Wesley.

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd Edition ed.): Addison Wesley.

Boehm, B. W., Brown, J., Kaspar, H., Lipow, M., MacLeod, G., & Merrit, M. (1978). *Characteristics of Software Quality*. New York: North-Holland.

Bosch, J. (2000). *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. London, UK: Addison-Wesley.

Clements, P., Kazman, R., & Klein, M. (2002). *Evaluating Software Architectures-Methods and Case Studies*: Addison Wesley.

Deissenboeck, F., Heinemann, L., Herrmannsdoerfer, M., Lochmann, K., & Wagner, S. (2011). *The QUAMOCO tool chain for quality modeling and assessment*. In Proc. of The 33rd International Conference on Software Engineering (ICSE'11).

Deissenboeck, F., Juergens, E., Lochmann, K., & Wagner, S. (2009). *Software quality models: Purposes, usage scenarios and requirements*. In Proc. of The 2009 ICSE Workshop on Software Quality (WOSQ '09).

Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., & Girard, J. F. (2007). *An Activity-Based Quality Model for Maintainability*. In Proc. of The 2007 IEEE International Conference on Software Maintenance (ICSM'07).

Dobrica, L., & Niemela, E. (2002). A survey on software architecture analysis methods. *IEEE Transactions on Software Engineering, 28*(7), 638--635.

Dromey, R. G. (1995). A Model for Software Product Quality. *IEEE Transactions on Software Engineering, 21*(2), 146--162.

Dromey, R. G. (1996). Cornering the Chimera. *IEEE Software, 13*(1), 33--43.

Folmer, E., & Bosch, J. (2005). *Case studies on analyzing software architectures for usability*. In Proc. of The 31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO 2005).

Franke, D., Kowalewski, S., & Weise, C. (2012). *A Mobile Software Quality Model*. In Proc. of The 12th International Conference on Quality Software (QSIC'12).

Gillies, A. (1992). Modelling Software Quality in The Commercial Environment. *Software Quality Journal, 1*, 175-191.

Gillies, A. (1997). *Software Quality: Theory and Management*: International Thomson Computer Press.

Goeb, A., & Lochmann, K. (2011). *A software quality model for SOA*. In Proc. of The 8th International Workshop on Software Quality (WoSQ'11).

Grady, R. B. (1992). *Practical software metrics for project management and process improvement*: Prentice-Hall, Inc.

Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied Software Architecture*: Addison-Wesley.

ISO. (1992). *Information Technology--Software Product Evaluation - Quality Characteristics and Guidelines for Their Use* (ISO 9126): International Organisation for Standardization.

ISO/IEC. (2012). *Systems and Software Quality Requirements and Evaluation (SQuaRE)*: ISO/IEC Standard, BS ISO/IEC 25021.

Jansen, A., & Bosch, J. (2005). *Software Architecture as a Set of Architectural Design Decisions*. In Proc. of The 5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05).

Kayed, A., Hirzalla, N., Samhan, A. A., & Alfayoumi, M. (2009). *Towards an Ontology for Software Product Quality Attributes*. In Proc. of The Fourth International Conference on Internet and Web Applications and Services (ICIW'09).

Kazman, R., Abowd, G., Bass, L., & Clements, P. (1996). Scenario-based analysis of software architecture. *IEEE Software*.

Kazman, R., Bass, L., Abowd, G., & Webb, M. (1994). *SAAM: a method for analyzing the properties of software architectures*. In Proc. of The 16th International Conference on Software Engineering (ICSE'94).

Kitchenham, B., & Pfleeger, S. L. (1996). Software Quality: The Elusive Target. *IEEE Software, 13*(1), 12-21.

Klas, M., Lampasona, C., & Munch, J. (2011). *Adapting Software Quality Models: Practical Challenges, Approach, and First Empirical Results*. In Proc. of Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA'11).

Kostelijk, T. (2005). Misleading Architecting Tradeoffs. *IEEE Computer, 38*(5), 20--26.

Leveson, N. G. (1995). *Safeware: System Safety and Computers*: Addison Wesley.

Lochmann, K., & Goeb, A. (2011). *A unifying model for software quality*. In Proc. of The 8th international workshop on Software quality (WoSQ'11).

Luckey, M., Baumann, A., Mendez, D., & Wagner, S. (2010). *Reusing security requirements using an extended quality model*. In Proc. of The 2010 ICSE Workshop on Software Engineering for Secure Systems (SESS'10).

McCall, J., Richards, P., & Walters, G. (1977). *Factors in Software Quality* (Technical Report CDRL A003, Vol 1.): US Rome Air Development Center.

MoD. (2000). *HAZOP Studies on Systems Containing Programmable Electronics, Part 1 Requirements; Part 2: General Application Guidance* (MoD 0058): Ministry of Defence.

Neumann, P. G. (1995). *Computer-Related Risks*: ACM Press.

Perry, W. E. (1991). *Quality Assurance for Information Systems: Methods, Tools and Techniques*: John Wiley and Sons.

Shahin, M., Liang, P., & Khayyambashi, M. R. (2009). *Architectural design decision: Existing models and tools*. In Proc. of The Joint Working IEEE/IFIP Conference on Software Architecture, 2009 and European Conference on Software Architecture (WICSA/ECSA 2009).

Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*: Pretince Hall.

Storey, N. (1996). *Safety-Critical Computer Systems*: Addison.

Taylor, R. N., Medvidovic, N., & Dashofy, E. M. (2010). *Software Archiecture: Foundations, Theory, and Practice*: Wiley.

Wagner, S. (2010). A Bayesian network approach to assess and predict software quality using activity-based quality models. *Information and Software Technology, 52*(11), 1230--1241.

Wagner, S., & Deissenboeck, F. (2007). *An Integrated Approach to Quality Modelling*. In Proc. of The 5th International Workshop on Software Quality (WoSQ'07).

Wagner, S., Lochmann, K., Heinemann, L., Klas, M., Trendowicz, A., Plosch, R., et al. (2012). *The QUAMOCO product quality modelling and assessment approach*. In Proc. of The 2012 International Conference on Software Engineering (ICSE'12).

Winter, S., Wagner, S., & Deissenboeck, F. (2008). A Comprehensive Model of Usability. In J. Gulliksen, et al. (Eds.), *Engineering Interactive Systems* (pp. 106--122).

Woods, E. (2012). Industrial architectural assessment using TARA. *Journal of Systems and Software, 85*(9), 2034-2047.

Zhang, Y. (2005). *Quality modelling and metrics of Web Information Systems*. PhD. Thesis, Oxford Brookes University, Oxford, UK.