Requirements Trade-off Analysis for Test-First Development

Nien-Lin Hsueh, Kee-Wun Lee, Shi-Chuen Hwang

Feng Chia University
No.100, Wunhua Rd., Situn District, Taichung, Taiwan (R.O.C.)
{nlhsueh,m9601321,schwang}@fcu.edu.tw

***Abstract*—Test-first development requires tests before implementation and provides fast feedback after implementation. However, this development method emphasizes functional testing rather than non-functional testing. Furthermore, it does not provide any approach to handle requirements trade-off problems even requirements conflicts are inevitable during software development. Thus, in this research we design a Requirement Trade-off Analysis Framework (RTAF) to automatically explore conflicts between requirements. This framework allows developers to define functional and non-functional requirements, set the properties of each requirement, and specify the critical design point of the system. In our approach, a critical design point may be implemented by several designs. By evaluating the satisfaction degrees of all requirements with respective to the possible designs, RTAF will determine the best design according to the critical method of different designs. This research will introduce a process to apply RTAF. A sorting example is developed to describe our framework and process. This approach is implemented on the basis of JUnit and JUnitPerf.**

***Keywords*—Test-first development, requirements conflicts, trade-off analysis**

1. Introduction

Requirements play a crucial role in software development process. Requirements consist of functional (e.g., login, logout, and read file function) and non-functional requirements (e.g., performance, reliability, and maintainability). Well-defined requirements will lead to the success of a project. Failed to meet non-functional requirements may cause a system unusable. It is not easy to satisfy all non-functional requirements without affecting other functional requirements (FR) or non-functional requirements (NFR). When a non-functional requirement is satisfied, it may impact the others (e.g., a system may perform quickly but exhaust a lot of memories). Time-space trade-off occurred when a system is required to have the best performance and using the least space. Trade-off is a spot where enhancing one attributes decreases the others [9]. Thus, Best performance can be achieved but much space is demanded.

To achieve the greatest performance, different code designs may be required. Much of the quality aspects of a system or non-functional requirements are determined during design phase [6]. Therefore, programmers have to make different code design and test the performance of each design in order to satisfy the customers' expectations [17].

Test-first development is part of agile software development approaches. In test-first development, test cases are implemented before the coding phase. By doing test first, programmers will think about what to do before thinking about how to do it [4]. This development method improves software quality and helps programmers work faster. However, some problems were found on this development method. First, this approach emphasizes functional testing rather than non-functional testing [7, 13, 1]. Second, this approach does not handle requirements trade-off problems. Therefore, the interaction between functional requirements and non-functional requirements are difficult to identify in test-first development. Furthermore, conflicts between requirements were identified manually.

To resolve these problems, we propose an object-oriented framework – called Requirement Trade-off Analysis Framework (RTAF) – which utilizes the automation capability of JUnit and JUnitPerf, to investigate the trade-offs between requirements in an automatic way. RTAF obtains trade-off decision in the design phase. Our framework provides the following functionality:

- allowing developers to define system requirements and their satisfaction degrees

- providing a framework where developers can define different designs for evaluating the satisfaction degrees of non-functional requirements
- helping the developers to identify conflicts between non-functional requirements and designs

Our approach will use performance between time and load to illustrate the requirements trade-off analysis. The approach is iterative until the final decision is made.

The rest of this paper is organized as follows: section 2 provides the background of trade-off analysis, conflict, and Test-first Development. Section 3 introduces the proposed RTAF and its process. Section 4 provides a case study to demonstrate our approach. Section 5 is related work. Finally, section 6 concludes the thesis with summary and future work.

2. REQUIREMENTS TRADE-OFF ANALYSIS

2.1. Requirements Conflicts

User requirements usually conflict with each other. Conflicts are inevitable in requirements elicitation. Handling conflicts can improve productivity, satisfaction, quality and also understanding of the requirements.

Conflicts occur when an increase in the degree of satisfaction of a requirement cause a decrease in the degree of satisfaction of another requirements [19]. According to Yen and Tiao [19], conflicts can be divided into two types: completely conflicting and partially conflicting (Fig. 1). Two conflicting requirements are said to be *completely conflicting* if an increase of the satisfaction degree of one requirement *always* decreases the satisfaction degree of other requirements; two conflicting requirements are said to be *partially conflicting* if an increase of the satisfaction degree of one requirement affect the other requirements in some circumstance. For example, space and time are said to be *completely conflicting* because a system with good performance always exhausts much space. Security and space are *partially conflicting* because not all systems with high security need much memory.

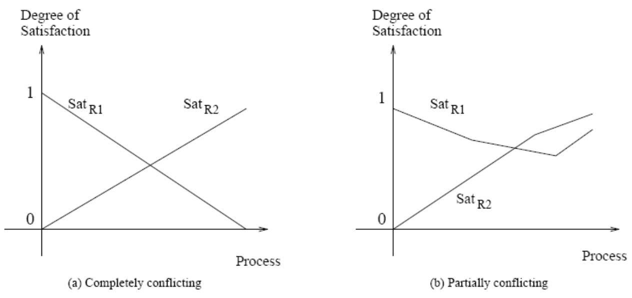


Fig. 1 Conflicting Imprecise Requirements [19]

2.2 Trade-off Analysis

Seems that conflicts are ineluctable in requirement elicitation, trade-off among requirements has become a very challenging issue. It is important to explore trade-offs between conflicting requirements. There are various approaches to trade-off analysis in the literature.

According to Kazman et al., all designs involve trade-offs [10]. If system attributes are not analyzed, trade-offs in the architecture may not be realized. Making good enough trade-offs between quality attributes is a crucial issue on quality assurance. Hence, researchers were working so hard to find out requirements trade-off solutions.

Poort and With develop a method called Non-Functional Decomposition (NFD) to resolve conflicts through non-functional decomposition [14]. This model splits requirements into primary and supplementary requirement to optimize the system structure by applying process, structural or functional strategies.

Yang et al. postpones trade-off analysis until runtime since they believe that information obtained during runtime is more accurate than the estimation at design phase [18]. They also found that it is very hard and impossible to make trade-off between designs during design phase. However, performance issues should be dealt earlier during development process, otherwise cost will increase and problems are hard to fix [7, 3, 9, 15, 10].

Kazman et al. proposed a method to resolve trade-offs in the software architecture during design phase [10]. They aimed in illuminating risks in the architecture designs. We believe resolve design trade-off problems earlier in the development process will assure the software product quality.

2.3 Test-first Development

Test-first Development improves code quality and productivity. It is a software development method consisting of short iterations where new test cases covering the desired improvements or new functions. Using this method, tests have to be prepared before coding to facilitate rapid feedback changes. Story cards, task cards and test cards are used to represent requirements in Test-first Development [16].

In conventional development method like Waterfall model (Fig. 2), testing is done after coding phase. Waterfall model is a sequential development process. Programmers have to develop the system from on phase to another phase in a purely sequential manner. Thus, when conflicts occur, it is infeasible to change the design.

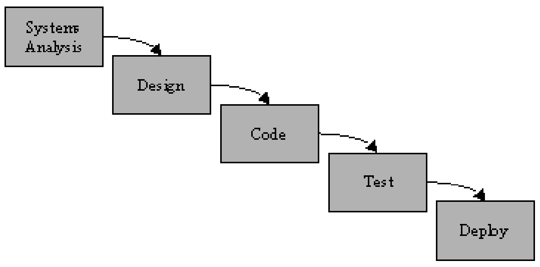


Fig. 2 Waterfall model.

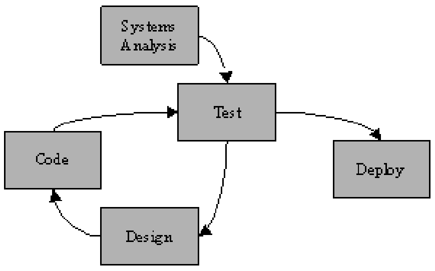


Fig. 3 Test-first development process.

Test-first Development is an iterated development process (Fig. 3). Hence, programmers may refactor the code to accommodate changes. One of the main key features of Test-first Development is that developers are required to create automated unit tests before writing the code. Programmers running the tests rapidly throughout the developing process to confirm the correct behavior of the code as programmers evolve and refactor the code.

RTAF aims at resolve trade-offs among requirements. Changes have to be made during the developing process. Thus, our framework is conducted based on Test-first Development. In this manner, trade-off will be revealed earlier in coding phase Moreover, different designs can be created and conflicts can be solved on time. This will guarantee the quality of the system and reduce the development cost.

3. REQUIREMENTS TRADE-OFF ANALYSIS FRAMEWORK (RTAF)

This section will introduce the proposed Requirements Trade-off Analysis Framework (RTAF) and the trade-off analysis process in Test-first Development.

3.1. Requirements Taxonomy

Fig. 4 shows the taxonomy of the requirements. The requirements taxonomy helps us realize the way to represent the requirements and the relationships among requirement, story, task and test.

It is important to categorize the requirements. According to the requirements taxonomy, requirements are divided into Functional and Non-functional Requirements. Non-functional requirements are quality attributes: reliability, performance and security. Other quality requirements such as portability, maintainability, and reusability may also adherence to standards and guidelines to meet the system quality metrics.

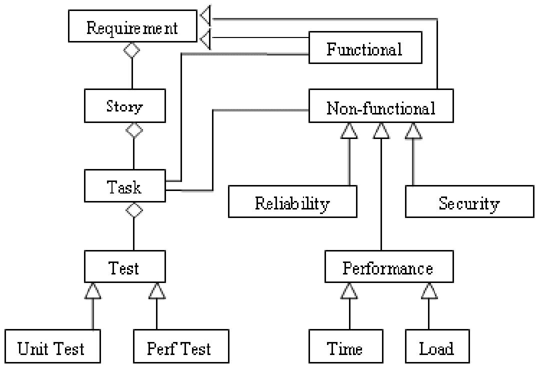


Fig. 4 Requirements Taxonomy.

Agile method like Extreme Programming expresses requirements as story or scenario [16]. Each story can be decomposed in to tasks. Tasks represent the discrete features of the system and unit test can then be designed for each task. Our framework exploits JUnit for unit test and JUnitPerf for performance testing.

Our trade-off analysis process is developed based on the requirements taxonomy depicts in Fig. 4. The details of the trade-off analysis process are elaborated on section 3.3.

Relationships between FR, NFR, design, and satisfaction degree are illustrated in Fig. 5. The upper part lists the functional and non-functional requirements and the dependence relationship between them. Usually, non-functional requirements are dependent on some functional requirements (e.g., *NFR_{11}* depends on *FR_1*); however, some of the non-functional are independent (e.g., *NFR_{ij}*). The independent non-functional requirements are system performance requirements, such as system performance, reliability and maintainability. Each non-functional requirement will be given a satisfaction degree according to users needs.

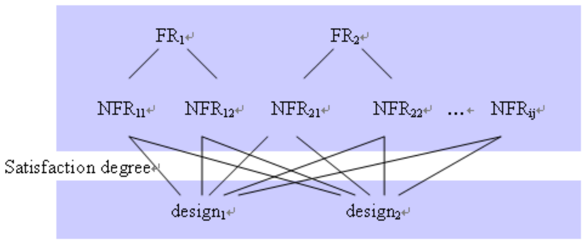


Fig. 5 Relationships between FR, NFR, design and satisfaction degree.

During system development process, developers propose the first design (*$design_1$*). In *$design_1$*, *NFR_{11}* may be satisfied with a high degree, but *NFR_{12}* is satisfied with low degree. The developer then tries another design, said *$design_2$*, to increase the satisfaction degree of *NFR_{12}*. When *$design_2$* satisfies both *NFR_{11}* and *NFR_{12}* with a high degree, it might impact the other requirements (e.g., *NFR_{22}* and *NFR_{ij}*). Without any indication, the developer may think "*$design_2$ is better than $design_1$*". However, some implicit conflicts might be occurred between other requirements.

In short, each NFR will be given a satisfaction degree and different designs is generated to approaching the given satisfaction degree. Implicit and explicit conflicts among non-functional requirements might occur. Our framework can help developers identify the conflicts among requirements and designs whenever a new design is proposed.

3.2. Requirements Trade-off Analysis Framework

The architecture of our framework is shown in Fig. 6. In our framework, all requirements comprise requirement id (*rid*), description, owner, and priority. FR and NFR both extend an abstract class – *Requirement*. Each FR object consists of:

- *rid* – requirement's id
- *description* – requirement's description
- *priority* – priority of the requirement
- *owner* – requirement's related stakeholder
- *input* – requirement input description
- *output* – requirement output description

Functional and non-functional requirements are tightly correlated. It is hard to state non-functional requirements separately from the functional requirements [16]. Thus, each NFR is related to one or more FRs. As illustrated in Fig 6, there is a relationship between FR and NFR. Besides having the same attributes as FR, NFR has a reference and some particular attributes:

- *sd* – satisfaction degree of NFR (a double type number between 0 and 1)
- *type* – NFR type (e.g., time, load, space)
- *unit* – unit of the NFR type (e.g., unit for time is ms, unit for space is byte)
- *value* – value of the expected result (e.g., time less than 30 ms, value=30)

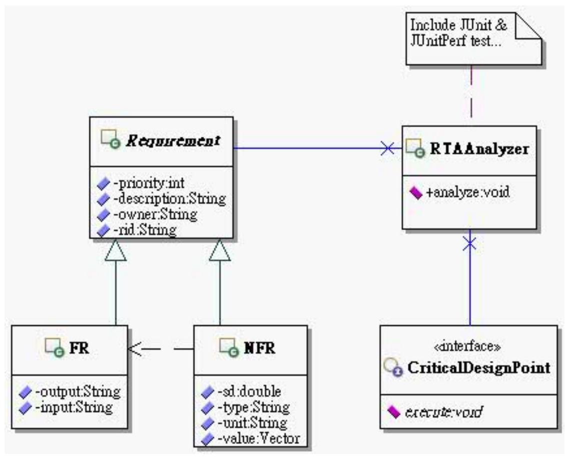
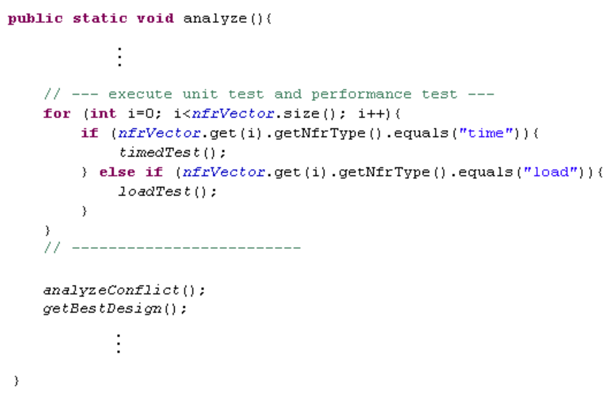


Fig. 6 Architecture of RTAF

In our framework, all requirements have to be modeled as FR or NFR objects. All Designed classes with critical method have to extend *CriticalDesignPoint* and override *execute()*. After modeling all requirements and setting the *CriticalDesignPoint*, the *RTAAnalyzer* will analyse conflicts between designs and requirements according to the FR and NFR attributes, and also the critical methods of different designs.

The *analyze()* in *RTAAnalyzer* is the core of RTAF. Conflicts analysis is perform in this method. An extract of source code from *RTAAnalyzer.analyze()* is shown in Fig. 7. Test results (unit test and performance test result) must be obtained before analyzing process. In Test-first development, programmers will write tests for system test. However, using RTAF, programmers do not have to write extra testing code for trade-off analysis. Unit test and performance test are included in our framework.



```
public static void analyze(){

        ⋮

    // --- execute unit test and performance test ---
    for (int i=0; i<nfrVector.size(); i++){
        if (nfrVector.get(i).getNfrType().equals("time")){
            timedTest();
        } else if (nfrVector.get(i).getNfrType().equals("load")){
            loadTest();
        }
    }
    // -------------------------

    analyzeConflict();
    getBestDesign();

        ⋮

}
```

Fig. 7 An extract of *RTAAnalyzer.analyze()*.

Performance test (JUnitperf) is a collection of JUnit test decorator. Thus, no matter which performance test (*timeTest()* or *loadTest()*) is invoked, unit test will be performed first, then followed by performance test. In RTAF, Unit test will only test the overridden execute() in different designs; JUnitPerf tests only time and load. When *timeTest()* is invoked, system elapsed time will be measured; when *loadTest()* is invoked, elapsed time of simulated number of concurrent users and iterations will be measured. The result of performance test will be recorded in a text file.

The *getBestDesign()* will show the best design according to the expected results and results generated by performance test. Design's satisfaction degree approximate the most to the expected satisfaction degree will be chosen as the best design. If all designs' satisfaction degrees are higher that the expected satisfaction degrees, the design with the highest satisfaction degree will be chosen as the best one.

In *analyzeConflict()*, satisfaction degree of each NFR of different designs will be measured. Conflicts between two NFRs occur when an increase in the satisfaction degree of one NFR decreases the satisfaction degree of the other, and vice versa. If conflicts between two NFRs occurred in different designs, we assumed that the designs are conflict to each other. This means that each design can only satisfy one NFR.

3.3 The Process Using RTAF

Fig. 8 introduces the trade-off analysis process. The process consists of the following six phases:

1. System analysis. Requirements gathered from the users will be expressed as stories and recorded in story cards. Each story will be broken down into tasks. Tasks are the basis of implementation. Then, programmer have to model each task's functional requirements as a FR object. Besides, each non-functional requirement will be modelled as a NFR object and each NFR's satisfaction degree will be set according to stakeholders' needs.

2. Initial design. Simple design is made to meet the current requirements. Programmers choose the class to implement each requirement.

3. Test. In RTAF, test cases are generated by the framework based on the requirements. The tests content are written by the tester according to the test case descriptions.

4. Design. This phase includes architecture and detailed designs. First, system architecture diagram will be created by the system analysts. Next, detailed design will be constructed using UML diagrams such as class diagrams. The class diagrams will be generated by the analysts based on the architecture design. The programmers have to determine the Critical Design Point in this phase and model the critical class as a child class of the *CriticalDesignPoint* class. Critical design point is the part of the system that affects the whole system performance the most. Programmers generate different designs, and test each design's critical point to obtain the best performance.

5. Code. Programmers implement the designs based on the task cards and class templates generated.

6. Trade-off analysis. When all tests passed, RTAF will analyses trade-offs of critical design point according to the test results. Trade-offs or conflicts between requirements and different designs will be revealed. Moreover, the best design will be shown. RTAF generates best design based on the satisfaction degrees and the priority of the requirements. Design with the closest satisfaction degree to the expected satisfaction degree and the highest priority will be chosen as the best result. Programmers can choose the suitable design according to the results generated in trade-off analysis phase. If the results do not meet the stakeholders' satisfaction degree, the programmers or system analysts have to make changes to the design according to the requirements and return to the System Analysis phase.

7. Deploy. Finally, if the best design has been decided, programmers will deploy the selected design to the system.

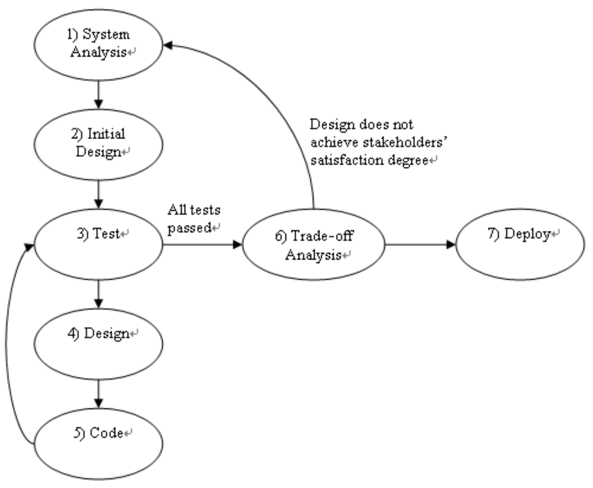


Fig. 8 Trade-off analysis process for test-first development.

4. A CASE STUDY

In this section, we present a simple case study – the implementation of *Student Grading System (SGS)*, to demonstrate the way to develop a subsystem using RTAF. The requirements include:

- sort more than 2000 data.
- read data from text file.
- sort 2000 data should be done within 30ms.
- allow 10 users access at the same time, time should not exceed 100ms each.

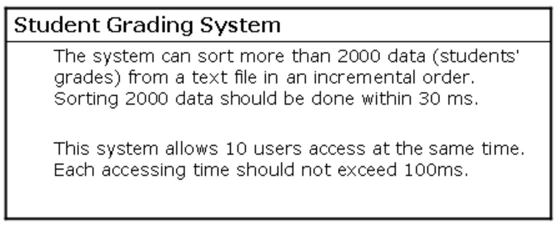
Phase 1: System analysis.

In this phase, we divided requirements analysis process into 3 steps:

1. Gather requirement.
2. Model each task's functional requirements as a FR object in RTAF.
3. Model each task's non-functional requirement as a NFR object and set each NFR's satisfaction degree in RTAF.

Step 1: Gather requirements

Story cards are used to represents the system requirements in Test-first Development. The story of the case study is presented as a story card in Fig. 9.

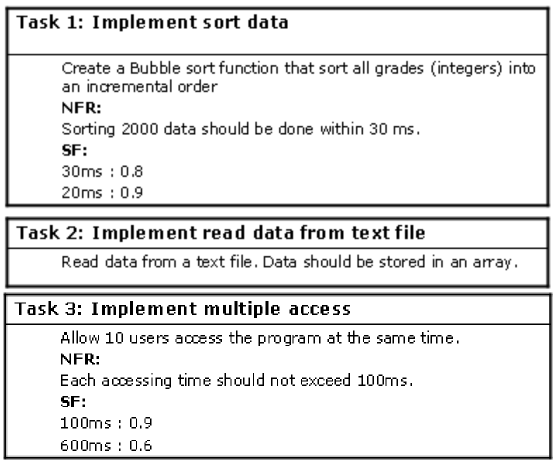


Student Grading System

The system can sort more than 2000 data (students' grades) from a text file in an incremental order.
Sorting 2000 data should be done within 30 ms.

This system allows 10 users access at the same time.
Each accessing time should not exceed 100ms.

Fig. 9 Story card for Student Grading System.



Task 1: Implement sort data

Create a Bubble sort function that sort all grades (integers) into an incremental order
NFR:
Sorting 2000 data should be done within 30 ms.
SF:
30ms : 0.8
20ms : 0.9

Task 2: Implement read data from text file

Read data from a text file. Data should be stored in an array.

Task 3: Implement multiple access

Allow 10 users access the program at the same time.
NFR:
Each accessing time should not exceed 100ms.
SF:
100ms : 0.9
600ms : 0.6

Fig. 10 Task Cards for Student Grading System.

The requirements in the story card will be decomposed into tasks. Each task is the principle unit of implementation [16]. Some of the tasks may concern about the quality attributes like performance and security. Extreme Programming only included task description in the task card. Considering that non-functional requirements are related to the specific functions of the system, we enclosed the non-functional requirements in the task card. This can be seen from *Task 1* and *Task 3* in Fig. 10. In our approach, the stakeholders are required to state the expected result and their satisfaction degree for the later conflict analysis.

Step 2: Model each task's functional requirements as a FR object in RTAF

The functional requirements gathered from the story of *Student Grading System* are *sort*, *read text file*, and *multiple access*. The relationship among these requirements is represented as an object diagram in Fig. 11. All requirements are treated as object. The *ReqReadFile* reads a text file and output an unsorted array to the *ReqSort*. The *ReqSort* will output an array sorted in an incremental order. This application allows multiple users access at the same time.

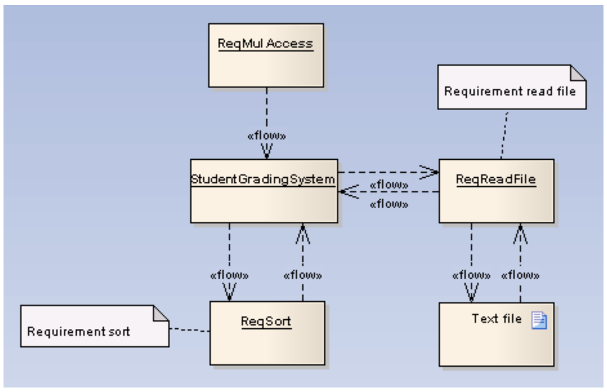


Fig. 11 Object Diagram of the Student Grading System.

In RTAF, each task's functional requirement must be modelled as a FR object. Doing so will provide useful information for trade-off analysis in later phase.

Fig 12 demonstrates the FR objects of the *Student Grading System*. The attributes of requirement sort in row one are *rid*, *description*, *priority* and *owner*. The methods *setInput* and *setOutput* are used to set input and output description respectively.

Step 3: Model each non-functional requirement as a NFR object and set each NFR's satisfaction degree in RTAF

According to the task cards in Fig. 10, *Task 1* and *Task 3* contain non-functional requirement. Each non-functional requirement has to model as a NFR object. This can be demonstrated as Fig. 13. The attributes set to the NFR objects in Fig. 13 are *reference*, *description*, *type*, and *priority*. Priority must be provided to enable programmers make the final decision.

```
NFR sortTime = new NFR(rSort, "time < 30", "time", 1);
NFR mulAccessLoad = new NFR(rMulAccess, "time < 1000", "load", 2);
```

Fig. 13 Model non-functional requirement as object.

After modelled the non-functional requirements as NFR object, each satisfaction degree of the NFR object has to be set. The result of JUnit test is either pass or fail. However, performance and the other NFRs test can not be justified by only right or wrong. According to IEEE-1061, 1998 [8], quality means "the degree to which software possesses a desired combination of quality attributes." Thus, our approach allows users to set expected response value and satisfaction degree.

```
sortTime.setSF(30, 0.8);
sortTime.setSF(20, 0.9);

mulAccessLoad.setSF(100, 0.9);
mulAccessLoad.setSF(600, 0.6);
```

Fig. 14 Set Satisfaction Degrees.

The satisfaction degree of each NFR is set as Fig. 14. It is a must to set the satisfaction degree because the result of trade-off analysis is relying on the *sd*. The arguments of *setSF* are the expected response value and satisfaction degree (*sd*). The first line of Fig. 14 indicates the satisfaction degree of sort is 0.8 if the execution time is 30ms.

```
FR rSort = new FR("s001", "sort all integers into an incremental order", 1, "user1");
rSort.setInput("int array");
rSort.setOutput("incremental sorted array");

FR rReadFile = new FR("s002","read text file");
rReadFile.setInput("numbers text file");
rReadFile.setOutput("--");

FR rMulAccess = new FR("s003","allow 10 users access at the same time", 2);
rMulAccess.setInput("--");
rMulAccess.setOutput("--");
```

Fig. 12 FR objects in RTAF.

In our approach, users have to insert two data for each NFR object in order to obtain the linear relationship of the value and the satisfaction degree. The linear relationship between value and satisfaction degree of the *sortTime* in Fig. 14 can be demonstrated in Fig. 15. In RTAF, the satisfaction degree is limited between 0 and 1. If the satisfaction degree approaching 1.0 indicates that the program achieves the greatest performance. We wish that our future research will adapt the fuzzy analysis for our framework. This will provide more precise analysis result for trade-off analysis.

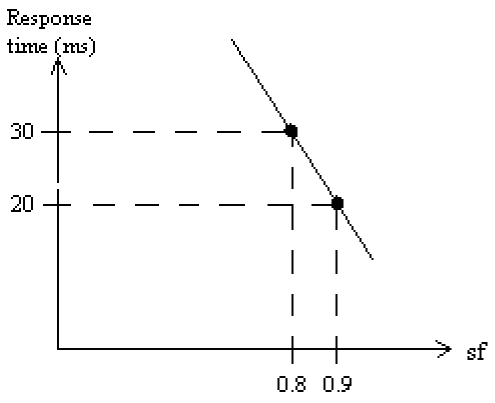


Fig. 15 Linear Relationship between Response Time and Satisfaction Degree.

Phase 2: Initial Design

In Initial Design phase, a simple design or a draft design has to be made for creating the class templates and test templates. Therefore, programmers have to choose the class to implement each requirement. Table 1 reveals the unimplemented relationships between classes and requirements. The requirement *rSort* will be implemented by sort() in the Sort class while the *rReadFile* will be implemented by *readFile()* in the *FileProcessor* class. In RTAF, this can be realized as Fig. 16(a).

TABLE 1

UNIMPLEMENTED RELATIONSHIPS BETWEEN CLASSES AND REQUIREMENTS.

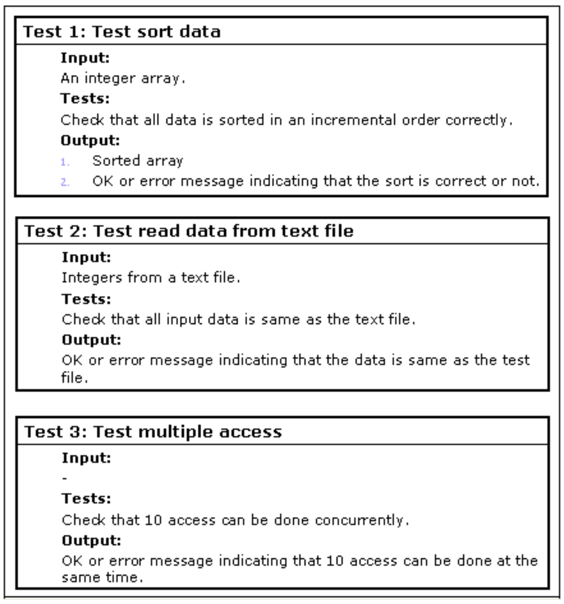
Class	Sort	FileProcessor	UserAccess
Requirement	Method		
rSort	sort()		
rReadFile		readFile()	
rMulAccess			mulAccess()

The method implement in Fig. 16(a) contains two arguments: FR object and class method name. According to Table 1, the requirement *rSort* will be realized in the sort() method of the Sort class. Hence, the second argument "*Sort.sort*" is set. The "*Sort*" before the "." represents the class name; the "*sort*" after the "." symbolize the method.

```
(a) implement(rSort, "Sort.sort");
    implement(rReadFile, "FileProcessor.readFile");

(b) implement(sortTime);
    implement(mulAccessLoad);
```

Fig. 16 (a) Implement the requirements to the Related Class in RTAF, (b) Implement the NFR objects.

Fig. 16(b) shows that the RTAF implement the NFR objects. After implementing the FR and NFR objects, class and test templates will be generated. Programmers can specify the package for the program. This can be done by invoking *addDesign("designName")*. For example, if the *addDesign("Design 1")* is implemented, all the class templates will be generated in the Design 1 package. However, unit test and performance test templates will always created in the Tests package.


Test 1: Test sort data

Input:
An integer array.
Tests:
Check that all data is sorted in an incremental order correctly.
Output:
1. Sorted array
2. OK or error message indicating that the sort is correct or not.

Test 2: Test read data from text file

Input:
Integers from a text file.
Tests:
Check that all input data is same as the text file.
Output:
OK or error message indicating that the data is same as the test file.

Test 3: Test multiple access

Input:
-
Tests:
Check that 10 access can be done concurrently.
Output:
OK or error message indicating that 10 access can be done at the same time.

Fig. 17 Test Description.

Phase 3: Test

In our framework, JUnit is used for functional testing and JUnitPerf is used for non-functional testing. In this phase, programmers implement test cases using the test templates created by RTAF based on the test description cards. Every task in the task cards generates one or more unit test. Test descriptions are illustrated in Fig. 17.

Phase 4: Design

This phase includes three steps: architecture design, detailed design, and set critical design point.

Step 1: Architecture Design

Architecture design is concerned with the high-level software structures, such as subsystems, packages, and tasks [5]. There are two kinds of architecture design – logical and physical. Logical architecture refers to the organization of classes and data types at design time; physical architecture refers to the system element that occurs at runtime. We use domain diagram to demonstrate the logical architecture of *Student Grading System* (Fig.18).

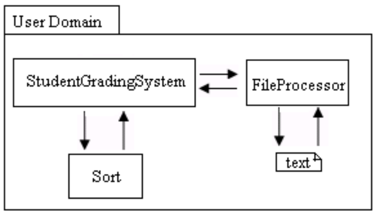


Fig. 18 Architecture design of Student Grading System.

Step 2: Detailed Design

According to Poort and With [6], conflicts among NFR occur when the solution is applied to a subsystem. This can be solved by separating the subsystem, and applying different solutions to the respective parts. Analysts or programmers have to find out critical design point according to the detailed design. Detailed design specifies the details of data members and function members within individual classes [5]. Fig. 19 illustrates the detailed design of the *Student Grading System*.

Different detail design is required for the trade-off analysis. To generate variety of designs to improve the performance, we provide a design guideline as follow.

- Intuitive design using array or variable
- Design using Data structure like linked-list
- Design using algorithm
- Using design pattern

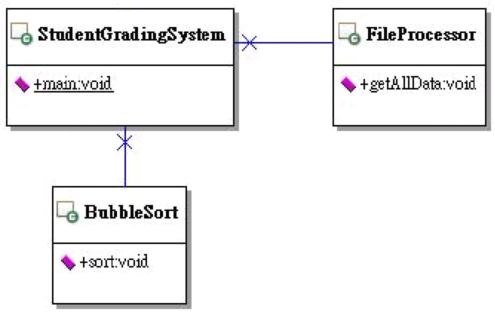


Fig. 19 Detailed Design of Student Grading System.

Step 3: Set critical design point

In this phase, the programmers have to set the Critical Design Point (CDP) for the trade-off analysis. After the architecture and detailed designs are provided, the CDP has to be set. CDP affects the most system performance. As such, the critical part of the system will be isolated and can be optimized by applying variety of designs. The least time spent by the CDP, the better satisfaction degree is obtained.

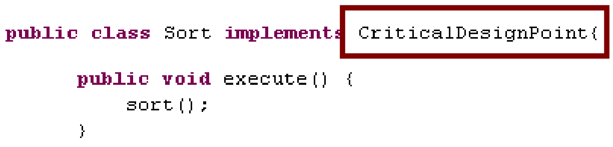
TABLE 2

MARKED CDP.

Class	Sort	FileProcessor	UserAccess
Requirement	Method		
rSort	sort()		
rReadFile		readFile()	
rMulAccess			mulAccess()

🟧 Critical Design Point

Programmers can find out CDP according to the table that implements the requirements (e.g., Table 1). If CDP is found, the method can be marked. We assume *sort()* will affect the system performance. Thus, *sort()* is marked as the CDP (Table 2). Then, model the critical class as a child class of the CDP class and override the *execute()* as Fig. 20. The content of the *execute()* should be the CDP method.



```
public class Sort implements CriticalDesignPoint{

      public void execute() {
          sort();
      }
```

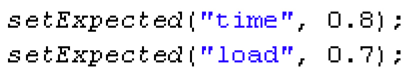
Fig. 20 Model the Critical Class as a Child Class of the CDP class

Phase 5: Code

Programmers start coding using the class templates created by RTAF in the System Analysis phase. Class templates created by RTAF is according to the requirements set by the analysts. Programmers can change the parameters or functions return type according to their needs.

Phase 6: Trade-off Analysis

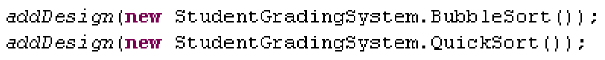
When programmers finish coding and testing, the expected satisfaction degree for each NFR has to be set. This can be seen from Fig. 21. The *setExpected* pass two arguments: NFR type and expected satisfaction degree of the NFR type.



```
setExpected("time", 0.8);
setExpected("load", 0.7);
```

Fig. 21 Set expected satisfaction degree of each NFR.

Besides setting the satisfaction degree, programmers have to set the designs that are going to be analyzed. In our case study, we assume *sort()* as the CDP. We have made two designs (*BubbleSort* and *QuickSort*) to analyze the satisfaction degree of time and load. As shown in Fig. 22, two designs (*StudentGradingSystem.BubbleSort()* and *StudentGradingSystem.QuickSort()*) is added. The *StudentGradingSystem* is the package where *BubbleSort* class is located.



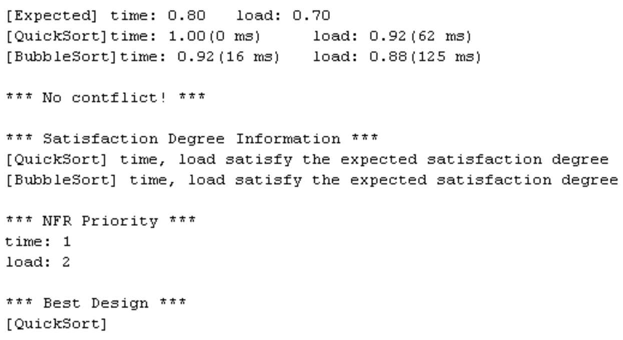
```
addDesign(new StudentGradingSystem.BubbleSort());
addDesign(new StudentGradingSystem.QuickSort());
```

Fig. 22 Set Designs for Trade-off Analysis.

The *analyze()* in *RTAAnalyzer* will analyze conflicts among different designs according to the CDP and the expected satisfaction degrees set. The trade-off analysis result will be printed as Fig. 23. The expected satisfaction degree of each NFR will be shown in the first line. Line 2 and Line 3 of Fig. 23 show the satisfaction degree results of each design.

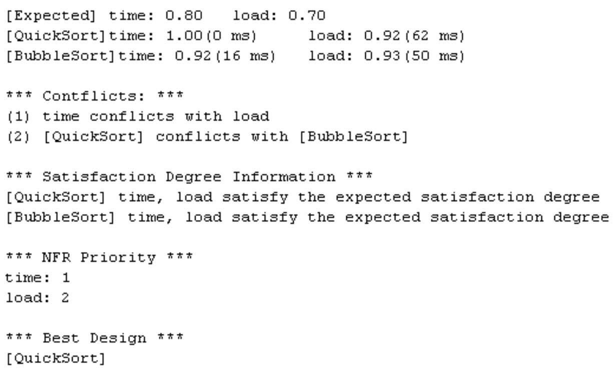
The Satisfaction Degree Information shows if the results achieve the satisfaction degree of each NFR. In Fig. 23, the result of *QuickSort* (time: 1.00; load: 0.92) and *BubbleSort* (time: 0.92; load: 0.88) reveal that both design achieve the expected satisfaction degree (time: 0.80; load: 0.70).

In our example, time and load of *sort()* is tested. Result in Fig. 23 reveals that no conflicts occur. In RTAF, if one satisfaction degree increase while the other decrease, and vice versa, we consider conflict occur. If conflicts occur, it will be revealed as Fig. 24. Time for *QuickSort* is 1.00 while time for *BubbleSort* decrease 0.08 (0.92); load for *QuickSort* is 0.92 while load for *BubbleSort* increase 0.01 (0.93). Satisfaction degree for time decrease while increase for load. Apparently, conflicts occurred, they are: 1) conflict among NFRs (time conflicts with load), and 2) conflict among design ([QuickSort] conflicts with [BubbleSort]).



```
[Expected] time: 0.80   load: 0.70
[QuickSort]time: 1.00(0 ms)      load: 0.92(62 ms)
[BubbleSort]time: 0.92(16 ms)    load: 0.88(125 ms)

*** No contflict! ***

*** Satisfaction Degree Information ***
[QuickSort] time, load satisfy the expected satisfaction degree
[BubbleSort] time, load satisfy the expected satisfaction degree

*** NFR Priority ***
time: 1
load: 2

*** Best Design ***
[QuickSort]
```

Fig. 23 Results with no Conflicts.



```
[Expected] time: 0.80   load: 0.70
[QuickSort]time: 1.00(0 ms)      load: 0.92(62 ms)
[BubbleSort]time: 0.92(16 ms)    load: 0.93(50 ms)

*** Contflicts: ***
(1) time conflicts with load
(2) [QuickSort] conflicts with [BubbleSort]

*** Satisfaction Degree Information ***
[QuickSort] time, load satisfy the expected satisfaction degree
[BubbleSort] time, load satisfy the expected satisfaction degree

*** NFR Priority ***
time: 1
load: 2

*** Best Design ***
[QuickSort]
```

Fig. 24 Results with Conflicts.

Besides showing the conflicts among NFRs and designs, priority of each NFR and the best design will be shown to help programmers in decision making. The lowest part of Fig. 23 and Fig. 24 show that [QuickSort] is the best design because the satisfaction degree of [QuickSort] is the highest.

If the programmers do not satisfy with the design, they can back to the System Analysis phase, review stakeholders' requirements and make new design. This process can iterate until the most satisfy design is found. After certain iteration, programmers can determine which design provides the best performance and is appropriate to be used in the developing system.

Phase 7: Deploy

If the programmers have found the most suitable design for the system, the final step is to deploy the chosen design to the system.

The sequence of trade-off analysis in RTAF can be demonstrated in Fig. 25. First, model functional requirements and non-functional requirements as FR and NFR objects. Next, set critical design point and override *execute()*. *RTAAnalyzer* will analyze conflicts based on the FR attributes and the expected result set in NFR. Finally, conflicts results and the best design will be shown.

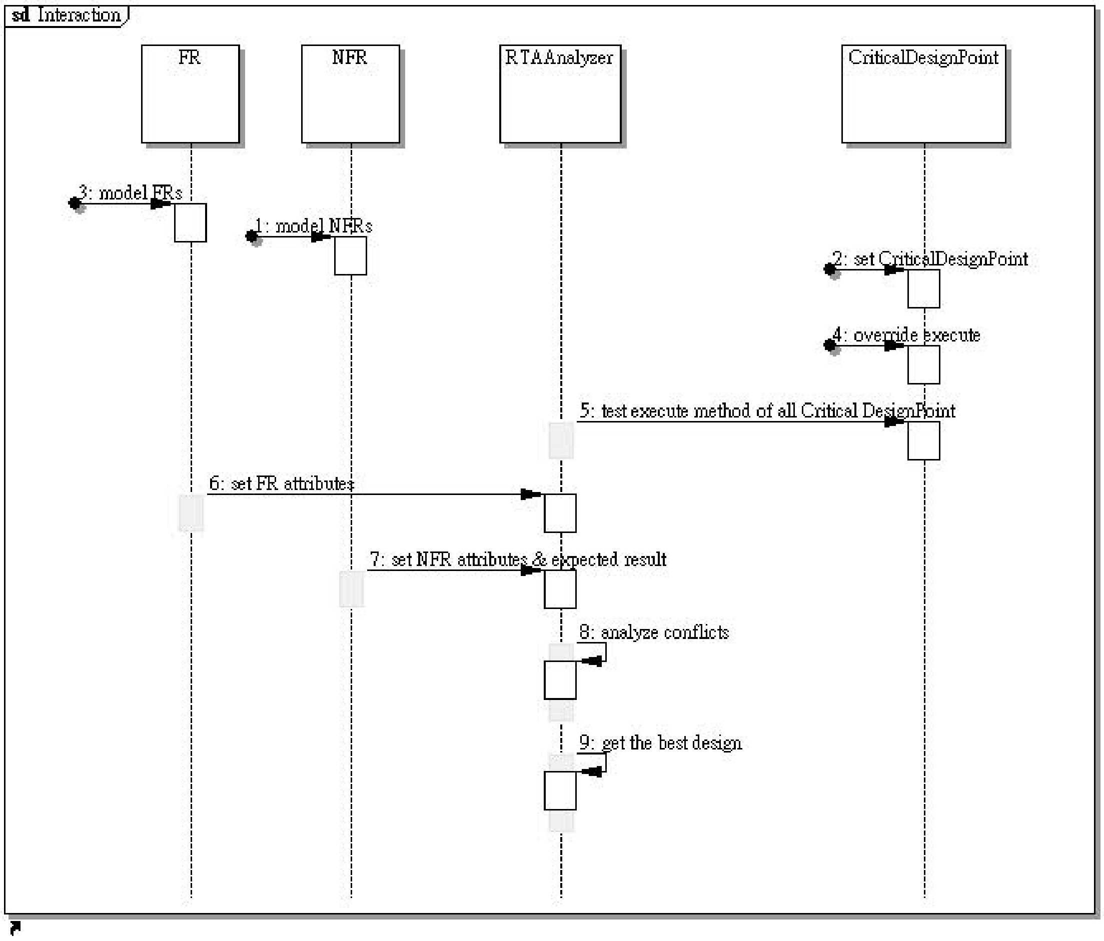


Fig. 25 Sequence Diagram of Trade-off Analysis in RTAF

5. Related Work

Requirement Trade-off Analysis technique published by Lee and Kuo analyzes the conflicts of a system in mathematical way [11]. This approach extended hierarchical aggregation structure with fuzzy and/or operators to facilitate requirements. Furthermore, they provide a requirements classification scheme to classify requirements.

Poort and With provided a Non-Functional Decomposition (NFD) framework that provides a model to resolve conflicts of the requirement [14]. They believe that functional requirements are never conflicting; conflicts might emerge in the non-functional requirements. The authors split requirements into primary and supplementary requirements. Relations between requirements are created. The NFD process isolates conflicting requirements. The isolated requirements will be optimized by applying process, structural or functional solution strategies.

Another trade-off analysis approach was presented by Yang et al [18]. To guarantee the quality of the system at runtime, their approach resolves trade-off solution according to the runtime context. They believe runtime information could only be acquired during the execution. It is not always possible to acquire desire quality during design phase. Thus the trade-off decision will be done during system execution.

The comparison results of those related works and RTAF are shown in Table 3. Researches done by Lee and Kuo, Poort and With identified conflicts manually. Although Yang et al resolve non-functional requirement conflicts automatically; the trade-off solution is done during system execution.

Trade-off analysis method proposed by Kazman et al. resolve trade-offs in the software architecture during design phase [10]. They aimed in detecting potential risk within the system architecture. The analysis process is done step by step manually.

Obviously, there is a lack of automatic trade-off analysis in practice. Our framework, RTAF, shows trade-off automatically during design phase. This will help developers resolve conflicts problems earlier during development process and improve project success rate.

TABLE 3

COMPARISON OF RELATED WORKS

	Resolve trade-off manually	**Resolve trade-off automatically**	**Trade-off during design**	**Trade-off during execution**
Lee & Kuo [11]	●			
Poort & With [14]	●			
Yang et al [18]		●		●
Kazman et al. [10]	●		●	●
RTAF		●	●	

6. CONCLUSIONS

Our Requirement Trade-off Analysis framework (RTAF) automatically investigates trade-off among requirements during design phase in Test-first Development. In RTAF, trade-off among each non-functional requirements and different designs will be shown. However, the final decision has to be made by the designers.

Our research concern about performance because JUnitPerf is the only performance tool that exist for time test. Our envision is that the other NFR testing components like security and reliability can be invented and be used in our framework.

There are two main restriction of our proposed framework. First, our approach is not applicable to all designs. This framework is best used for system with detail design. With detail design, the programmer is able to determine the critical design point. Our approach relies on the critical design point to resolve trade-off problem

Second, RTAF is not applicable to test frameworks. A framework consists of a large number of functions. There might be a lot of critical design points among these functions. RTAF can only analyse trade-off of one critical design point.

In future research, we wish our framework can be used to test performance of different design patterns. Furthermore, we wish that fuzzy logic can be adapted to our conflict analysis instead of the linear calculation.

ACKNOWLEDGMENT

This work was supported by the National Science Council, Taiwan, under Contract No. NSC97-2218-E-035-004.

REFERENCES

[1] Andrea, Jennitta, "Envisioning the Next-Generation of Functional Testing Tools,"

Software, IEEE, vol.24, no.3, pp.58-66, May-June 2007.

[2] Astels, David. *Test-driven Deveopment: A Practical Guide*, Chapter 3, 2003.

[3] Chung, L., B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements In Software Engineering*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 2000.

[4] Decker, B.; Ras, E.; Rech, J.; Jaubert, P.; Rieth, M., "Wiki-Based Stakeholder Participation in Requirements Engineering," *Software, IEEE*, vol.24, no.2, pp.28-35, March-April 2007.

[5] Douglass, B. P., *Real Time UML Third Edition: Advances in the UML for Real-Time Systems*. Addison Wesley Publishing Company, 2004.

[6] Gross, D. and E.S.K. Yu, From non-functional requirements to design through patterns, *Requir. Eng*. 6 (2001) (1), pp. 18–36.

[7] Ho, C. W.; Johnson, M.J.; Williams, L.; Maximilien, E.M., "On agile performance requirements specification and testing," *Agile Conference, 2006*, vol., no., pp. 6 pp.-, 23-28 July 2006.

[8] "IEEE standard for a software quality metrics methodology," *IEEE Std 1061-1998*, 31 Dec 1998.

[9] Kassab, M.; Daneva, M.; Ormandjieva, O., "Scope Management of Non-Functional Requirements," *Software Engineering and Advanced Applications, 2007*, pp.409-417, 28-31 Aug. 2007.

[10] Kazman, R.; Barbacci, M.; Klein, M.; Jeromy Carriere, S.; Woods, S.G., "Experience with performing architecture tradeoff analysis," Software Engineering, 1999. *Proceedings of the 1999 International Conference*, pp.54-63, 22-22 May 1999.

[11] Lee, J.; Kuo, J. Y., "New approach to requirements trade-off analysis for complex systems," *Knowledge and Data Engineering, IEEE Transactions*, vol.10, no.4, pp.551-562, Jul/Aug 1998.

[12] Metsker. S. J. and Wake W.C., *Design Patterns in Java*, Addison-Wesley, Boston, 2006.

[13] Paetsch, F.; Eberlein, A.; Maurer, F., "Requirements engineering and agile software development," *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003. WET ICE 2003. Proceedings. Twelfth IEEE International Workshops*, pp. 308-313, 9-11 June 2003.

[14] Poort, E.R.; de With, P.H.N., "Resolving requirement conflicts through non-functional decomposition," *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference*, pp. 145-154, 12-15 June 2004.

[15] Smith C. U. and Williams, L. G.., *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*, Addison-Wesley, Boston, MA, 2004.

[16] Sommerville, I. *Software Engineering*, 8th ed, 2007.

[17] Tate, B., Clark, M., Lee, B., and Linskey, P., *Bitter EJB*, Manning, 2003.

[18] Yang, J.; Huang, G.; Zhu, W.; Cui, X.; Mai, H., "Quality attribute tradeoff through adaptive architectures at runtime," *The Journal of Systems and Software* (2008).

[19] Yen, J.; Tiao, W.A., "A systematic tradeoff analysis for conflicting imprecise requirements," *Requirements Engineering, 1997., Proceedings of the Third IEEE International Symposium*, pp.87-96, 6-10, Jan 1997.