



# **A Framework to enhance the process of potential faults detection at early stages of software development life cycle**

**By**

**Fatima Faraj Musbah Saeid**

**Supervisor**

**Dr.Mohamed Hagal**

**This Thesis was submitted in Partial Fulfillment of the  
Requirements for Master's Degree of Science in Software  
Engineering**

**University of Benghazi**

**Faculty of Information Technology**

**February 2022**

Copyright © 2022. All rights reserved, no part of this thesis may be reproduced in any form, electronic or mechanical, including photocopy, recording scanning, or any information, without the permission in writing from the author or the directorate of graduate studies and training of Benghazi university.

حقوق الطبع 2022 محفوظة. لا يسمح اخذ اي معلومة من اي جزء من هذه الرسالة على هيئة نسخة إلكترونية أو ميكانيكية بطريقة التصوير أو التسجيل أو المسح الضوئي أو المسح الضوئي من دون الحصول على إذن كتابي من المؤلف أو إدارة الدراسات العليا والتدريب بجامعة بنغازي.



# A Framework to enhance the process of potential faults detection at early stages of software development life cycle

By  
**Fatima Faraj Saeid**

This Thesis was Successfully Defended and Approved on . . **2022**

Supervisor  
**Dr. Mohammed Hagal**

Signature: .....

Dr..... ( **Internal examiner** )

Signature: .....

Dr..... ( **External examiner** )

Signature: .....

**(Dean of Faculty)**

**(Director of Graduate studies and training)**

## *Dedication*

I'd like to take this opportunity to express my heartfelt gratitude to my parents, sisters, and brothers for their unwavering love, encouragement, and patience over the years.

I also owe gratitude to all of my kind friends who have supported me at every turn.

Fatima Faraj Saeid

## **Acknowledgments**

First and foremost, I want to express my gratitude to ALLAH, who has provided everything I need to finish this project.

My profound gratitude goes to my supervisor, Dr. Mohammed Hagal, for his tremendous direction, unending support, and all of the fruitful discussions we had from the beginning of this project until the final thesis revisions..

I owe a great debt of appreciation to Mr. Bilal Jabour, who graciously assisted and advised me throughout my study period.

Fatima Faraj Saeid

# List of Contents

<b>Table</b>	<b>Page no</b>
Copyright©.....	ii
Examination Committee.....	iii
Dedication.....	iv
Acknowledgments.....	v
List of Contents.....	vi
List of Tables.....	viii
List of Figures.....	ix
List of Abbreviations and Symbols .....	x
Abstract.....	xi
Chapter 1: Introduction.....	1
1.1 Problem statement.....	3
1.2 Research questions.....	3
1.3 Aims and objectives.....	4
1.4 Importance of the study.....	4
1.5 Scope and limitation.....	4
1.6 Overview of the study.....	5
Chapter 2: Background and Literature Survey.....	6
2.1 Requirements validation activity.....	6
2.1.1 Requirements validation techniques.....	7
2.2 Coupling.....	8
2.3 The testing process.....	10
2.4 Literature survey.....	13
2.4.1 General background.....	13
2.4.2 Requirement issues .....	15
2.4.3 Coupling issues .....	19
Chapter 3: The Proposed Approach .....	23
3.1 Requirements validation process improvement.....	24
3.2 Coupling and its testing process optimization.....	29

3.2.1 Coupling precedence.....	29
3.2.2 Types of coupling.....	30
Chapter 4: Case Study.....	40
4.1 Scenario.....	40
4.1.1 Requirements characteristics traceability.....	41
4.1.2 Design coupling traceability.....	49
Chapter 5: Result and Discussion.....	55
Conclusion and future work.....	59
References.....	60

## List of Tables

Table	Page no.
Table 2.1. Studies and characteristics of the requirements addressed in them	19
Table 3.1 Requirement resource table	25
Table 3.2 An example of consistency traceability table	27
Table 3.3. Data coupling testing process	31
Table 3.4. Stamp coupling testing process	33
Table 3.5. Control coupling testing process	35
Table 3.6. Common coupling testing process	37
Table 3.7. Content coupling testing process	39
Table 4.1. Requirement resource table	41
Table 4.2. Consistency traceability table	47
Table 4.3. An example of data coupling between two components	49
Table 4.4 An example of stamp coupling between two components	50
Table 4.5. An example of control coupling between two components	52
Table 4.6. An example of common coupling between two components	53
Table 4.7. An example of content coupling between two components	54



## List of Figures

Figure	Page Number
Figure 1.1: Major phases of the SDLC models	1
Figure 1.2: Requirements engineering process	2
Figure 2.1: Types of coupling adapted	9
Figure 3.1: Overview of the proposed framework	23
Figure 3.2 : Activity diagram with swimlanes example	26
Figure 3.3 : Activity diagram with swimlanes to trace the consistency between requirements	28
Figure 3.4: An example of coupling precedence	30
Figure 3.5: Data coupling between different software components	30
Figure 3.6 : Stamp coupling between two components	32
Figure 3.7: Control coupling between two components example	34
Figure 3.8 : Common coupling between two components	36
Figure 3.9 : Content coupling between two components	38
Figure 4.1 : Activity diagram with swimlanes for "Add new book"	42
Figure 4.2 : Activity diagram with swimlane for "Borrow book"	43
Figure 4.3 : Activity diagram with swimlane for "Remove book"	44
Figure 4.4 : Activity diagram with swimlane for "Browse books info "	44
Figure 4.5: Activity diagram with swimlane for "Returns Book "	45
Figure 4.6 : Activity diagram with swimlane for" Book Booking"	46
Fig 4.7 : Activity diagram with swimlane to trace the consistency between two requirements	48

## LIST OF ABBREVIATIONS AND SYMBOLS

Abbreviation	Meaning
<b>BVA</b>	Boundary value analysis
<b>CA</b>	Afferent coupling
<b>CBO</b>	Coupling between objects
<b>CE</b>	Efferent coupling
<b>ECP</b>	Equivalence class partitioning
<b>OO</b>	Object oriented
<b>PSOs</b>	Property specifications patterns
<b>RE</b>	Requirement Engineering
<b>SDLC</b>	Software development Life cycle
<b>SRS</b>	Software requirements specification
<b>SVM</b>	Support vector machine

# **A Framework to enhance the process of potential faults detection at early stages of software development life cycle**

**By**

**Fatima Faraj Saeid**

**Supervisor**

**Dr. Mohamed Hagal**

## **ABSTRACT**

The process of developing high-quality software depends on the extent to which it meets what is required of it completely and correctly. As a result, the requirements validation process and the testing phase are considered as the most critical stages for ascertaining exactly what the product will offer. Many efforts have been made to prepare methods and techniques to facilitate the testing process and ensure its quality. However, there is a lack of focus on test cases which can lead to potential flaws such as requirements and design coupling difficulties.

As a result, this thesis has been working on providing a comprehensive framework that enables software developers to focus on the underlying errors in an organized documentation manner, as well as to be supportive and complementary to the various processes of validation and testing, by focusing on the requirements validation process and the design coupling testing. A case study was presented in this thesis to clarify the mechanism of the proposed framework, in which the framework demonstrates a clear mechanism for focusing on potential faults by following requirements in the requirements engineering stage and testing the interaction between software components (design coupling).

**Keywords:** Requirements engineering, software engineering, design coupling, SDLC, potential faults.

# Chapter 1

## Introduction

This chapter provides an overview of the testing process in the software development life cycle, at early stages, research problem and scope.

Software development has begun to control and organize large areas of our life activities, and this space expands every day. It is necessary to have a way in order to manage software development, from the moment an idea is conceived through the stages of development until it is released ( i.e., in order to arrive at a product that meets the requirements of its stakeholders).

The Software Development Life Cycle (SDLC) is a sequential process in which to create or maintain software. It includes various stages that start from the stage of requirements elicitation to the stage of software maintenance. There are a wide range of models and methodologies that development teams use to develop software systems, which provides a framework for planning and monitoring the software development from the outset (Leau et al., 2012; Akinsola et al., 2020).

SDLC includes a set of different phases, starting with the functional system requirements, followed by design and implementation phases, then testing comes in. Carefully organization of these phases improves the performance and efficiency of software projects, whereas disregarding inherited errors/bugs between these phases weakens their roles.(Tuteja & Dubey, 2012 ; Nidamanuri, 2021). The main steps of the SDLC are depicted in Figure 1.1



Figure 1.1:Major phases of SDLC models (Akinsola et al., 2020)

It is obvious from figure 1.1, that the requirements engineering stage is the first stage in the SDLC on which the other stages of software development rely. The importance of this stage is to understand the stakeholders' needs, and then document what their software system will do. This stage consists of several activities: requirements elicitation, requirements analysis, requirements documentation, requirements validation, and requirements management. This referred to as the requirements engineering(RE) process (Darwish, 2016). Figure 1.2 illustrates the activities involved of this stage.

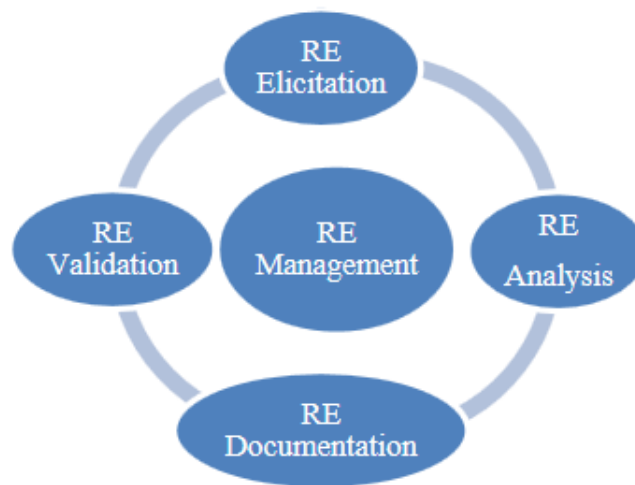


Figure 1.2: Requirements Engineering process (Darwish, 2016)

Furthermore, the requirements validation activity aims to guarantee that the requirements are correct and accurate, Thus writing the requirements in a clear and unambiguous manner is critical.

On the other hand, as known, the software testing process is a late stage in which the emphasis is on requirements specification and the software operation. This may overlook some inherent errors (potential faults) that testers may not notice. For example, the inconsistency in requirements or the complexity of the association between software components (i.e., high coupling between the components) can cause such errors. Therefore, by potential faults, we mean errors that the system testers may overlook, with regard to the inconsistency of the main components and subcomponents related or dependent on them, in a manner that ensures their consistency, accuracy and completeness, as well as design coupling between software components. Therefore, the proposed work aims to improve the software testing process by detecting potential

errors at early stages of the SDLC. This can be considered as a complementary process to the software testing stage.

## **1.1 Problem statement**

Software passes through many stages during its development process, from the requirements elicitation stage to its operation and maintenance stages. However, many software systems may fail to run or may encounter some problems while running. This may be due to some potential faults, such as requirements issues, or as a result of high coupling between software components. This will contribute to the spread of errors among these components, which may cost a lot of effort and time to maintain them. As a result, these effects on building clear and precise requirements. Hence, these require focusing on familiarity with many factors that help build robust requirements. Some of these factors are the requirements must be complete, correct, consistent, necessary, and traceable.

Furthermore, increasing the coupling process between the software components during the design stage is a negative factor in creating a coherent software design, since high-coupling software is more prone to the spread of errors among the components.

This study introduces a framework that helps to improve the software testing process. As a result, it minimizes potential faults that are often rooted in a number of stages of software development.

## **1.2 Research questions**

The primary research questions of the research problem can be summarized as follows:

Q1: What types of potential errors may contribute to software weakness or failure?

Q2: What are the most important methods to enhance the process of detecting potential errors during the SDLC's early stages?

Q3: What is the proposed approach for improving the process of detecting potential defects during the SDLC early stages?

Q4: Which parties benefits the most from enhancing the process of detecting potential defects during the SDLC early stages?

Q5: How the proposed framework work will be evaluated?

### **1.3 Aims and objectives**

1.3.1 This study attempts to achieve the following aims:

- Improve the software development process by reducing the potential errors at the early stages of the SDLC.
- Formulate a framework to enhance the process of detecting potential faults at early stages of the SDLC.

1.3.2 In the same context, this study has the following objectives:

- Supporting software testers in detecting the potential errors at early stages of the SDLC.
- Developing a framework to improve software-testing process, as well as to assist software testers.
- Identifying the most significant effects of improving the process of detecting potential faults at the early stages of the SDLC.
- Applying the proposed framework on case study.
- Evaluating the result obtained from the case study.

### **1.4 Importance of this study**

The importance of this study can be described as the follows:

- Emphasizing the need of improving the software testing process by focusing on potential flaws early at the SDLC.
- Determining the impact of some factors on enhancing the process of detecting potential faults at early stages of SDLC.
- Creating a framework that is complementary to the software-testing phase and assisting software testers in improving the testing process.

### **1.5 Scope and limitation**

1.5.1 Scope

The focus of this study is on the accuracy of requirements in terms of Complete, Necessary, Correct, and Consistency, as well as design coupling, which arises from the

interaction between the components, thus the field of this research focuses on the potential requirements faults and coupling testing in the design, which does not replace, but rather supports the testing process.

### 1.5.2 Limitation

This study only addresses the faults inherent at the early stage (i.e., the inherent problems of requirements and design coupling in the software design stage) and does not address the other stages of the SDLC.

## 1.6 Overview of this study

This thesis introduces a framework for potential faults testing as a complementary process to software testing, as the study focuses on the early stages (i.e., requirements and design) of the SDLC, which is one of the most essential stages of software development to limit the spread of errors. The framework describes the two-stage dilemmas that assist testers in detecting errors (faults) before they spread, saving time ,effort and lowering the burden on the final testing process.

This first chapter (Introduction) presents the study by outlining the general context of the SDLC, research problem, research questions , research aims and objectives.

Chapter 2 (Background and Literature survey) provides an overview of the two phases of the study (the requirements and the design phase), as well as a reviewing previous literature for the research topic through which the problems to be addressed in this study will be arrived at.

Chapter 3 (The proposed approach) presents the proposed framework to reduce the potential problems at the early stages of the SDLC.

Chapter 4 (Case Study) presents the application of the proposed framework to a case study of an electronic library system.

Chapter 5 demonstrates the findings of the study in greater depth.



## **Chapter 2**

### **Background and Literature survey**

This chapter includes four sections. The first section provides a brief background on the importance of requirements validation. The second section introduces coupling design process, which has an significant and triple effect in the software process that should be given more attention. The third section presents a brief general background on the software testing process. The fourth section focuses on the related work in the field of testing, which revolves around the study and efforts made in software testing, with regard to validating requirements and design coupling issues.

#### **2.1 Requirements validation activity**

The most essential activity in the requirements engineering (RE) stage is requirements validation, consequently, it is necessary to delve a little bit to give an overview of this stage, and thus the role and responsibility of this activity will become clear.

RE stage is the first and most important stage in the SDLC, as it focuses on collecting and understanding the requirements of the stakeholders in an appropriate way for collecting clear requirements about what the software system is expected to perform. As a result, it is thought to have a significant impact on all subsequent stages. Whereas engineering requirements focus on understanding the purpose of the software system to be built and gathering system requirements by collecting and extracting them from the stakeholders. Then thoroughly analyzing and documenting, which leads to building a software system that clearly performs what is required of it. Furthermore, the primary goal of Requirements Engineering is to guide development toward the production of a correct product. Hence, one of the problems with developing clear requirements at the time is the differing views of the stakeholders on them. So, if requirements are not specified properly, the system will cause a lot of problems that will be expensive to repair. Moreover, Effectiveness of requirements validation is the activity responsible for validating the requirements that have been documented, in order to ensure their correctness and accuracy, as well as resolving any ambiguities or inconsistencies therein. After ensuring the accuracy of these requirements, they are documented and forwarded to the next stage, which will focus on designing the system and how it will be

built (Nidamanuri, 2021). In addition, The cost of software testing will be reduced if testers are involved from the early stages of the development process (Graham, 2002) (Lawrence et al., 2001).

On the other hand, when testing is taken into account at the requirements stage, defects are detected early. As a result, software projects will be more robust in terms of design, implementation, and maintenance. Studies have shown that the goal of adding testing into the SDLC can be summarized as follows: (1) defects that are subsequently discovered have a root cause (i.e., poor requirements), and (2) if the error is detected early, it will not be too expensive to fix (Pandey and Batra, 2013). In conclusion, According to NASA findings, “problems that are not found until testing are at least 14 times more costly to fix than if the problem was found in the requirements phase” (Pandey & Batra, 2013).

Thus, the requirements stage should include the integration verification process later, in which the components of the software interact correctly with one another, additionally, the design verification process should not neglect checking the consistency of the software architecture and its requirements (Maia & Souza, 2018). Therefore, checking the coupling between components early must be taken in consideration.

Furthermore, detecting requirements conflicts causes problems in the development of software systems, delays their development, and exceeds the proposed cost of producing them. However, the process of conflict detection is critical for verifying requirements, and detecting the conflict is a significant challenge in and of itself. (Guo et al., 2021).

### **2.1.1 Requirements validation techniques**

The goal of using requirements validation techniques is to ensure the validity of those requirements and document them in a way that ensures the success of the software system later. There are many requirements validation techniques, so a brief look at a few of them will be given (Anas et al., 2016):

- Inspection

Inspections are a technique of manually checking requirements in order to ensure that they are correct and meet the needs of stakeholders.

- Prototyping

Prototyping facilitates the process of validating requirements, by attempting to simulate the system to be developed. It is an effective tool when there is uncertainty about the correctness and completeness of requirements, by creating an environment of understanding between developers and stakeholders.

- Requirements testing

The objective of this technique is to create test cases for all requirements that are documented in the software requirements specification (SRS). However, this technique is rather expensive, so it necessitates the assistance experts in the field of requirements engineering testing in order to be performed in a timely manner.

- Viewpoint-oriented requirements validation

The purpose of this technique is to facilitate the process of extracting requirements, by giving space to the different points of view, and then preparing an approach for negotiating these views in order to resolve the contradictions and ambiguities that surround the generation of valid requirements.

## **2.2 Coupling**

Functional independence of software components is required to reduce the propagation of software errors between them, and there are two criteria for measuring this: coupling and coherence. Where coupling determines the degree to which each component is dependent on the other. It also demonstrates the strength of the link between these two components, and the degree of complexity of the software product is determined (Shweta Sharma & Srinivasan, 2013).

Furthermore, because coupling is one of the basic characteristics (principles) of software systems, several standards and procedures for coupling have been proposed in order to support software development to ensure the quality and validity of the software product. Understanding the coupling (i.e. interaction between the components) is useful in many program development or maintenance activities in terms of quality, detection of errors and the effects of changes that may occur (Bavota, Dit, Oliveto, Penta, et al., 2013)

In addition, there are several types of coupling, which can be classified from high to low level. Figure 2.1 depicts such types (Shweta Sharma & Srinivasan, 2013).

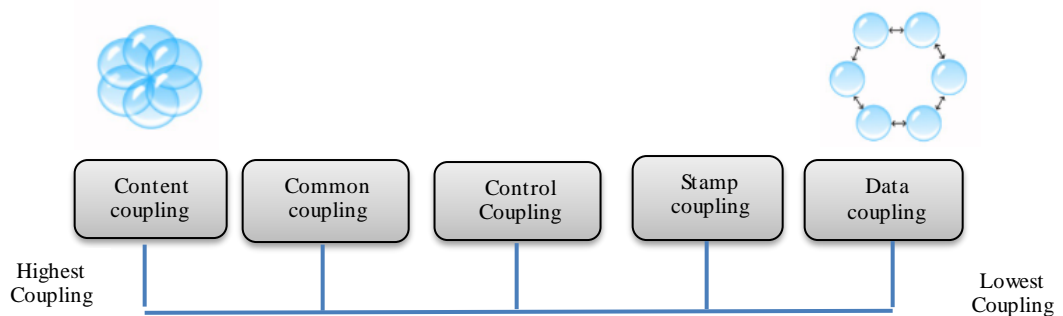


Figure 2.1: Types of coupling adapted from (Shweta Sharma & Srinivasan, 2013)

**Data coupling:** data coupling occurs when parameters are passed between more than one component.

**Stamp coupling:** stamp coupling occurs when an object or data structure is passed between components.

**Control coupling:** control coupling occurs when parameters are passed between two components causing an effect on the internal content of the second component.

**Common coupling:** common coupling occurs when many components use the same parameters.

**Content coupling:** content coupling occurs when a component changes the contents of another component.

Cohesion means that the less dependence between software components on each other's, the greater the strength of their cohesion and this contributed to the less spread of errors between them. As a result, the low of coupling leads to a higher cohesion and vice versa (Shweta Sharma & Srinivasan, 2013).

Moreover, in object oriented systems coupling and cohesion contribute to measure the strength of the interaction between the classes, methods, and attributes. And thus knowing the complexity of the program, which will negatively affect the interacting components in terms of the spread of errors (Shweta Sharma & Srinivasan, 2013).

To facilitate the software development process, the problem is divided into several problems (divide and conquer) which leads to controlling it, but on the other hand, the integration process between these parts is a challenge to its accuracy, due to the excessive interaction between them, (i.e., the high coupling) (Kamble et al., 2017). That is, when the divided problem was fragmented and several components were generated, it became easier to control. However, some components may aggravate interaction, and perhaps excessive interaction, resulting in the complexity of the interaction, which leads to high coupling, which is a major cause of the spread of errors between the interacting components.

The clarification of these types, as well as the proposed mechanism for tracking and testing these types, are presented in greater detail in Chapter 3.

### **2.3 The testing process**

Software testing is an essential phase of the software development cycle. Because the main objective of this process is to ensure that the software meets their specifications (Vanmali et al., 2002;Umar, 2020).So, a good testing process is an essential component of software development that is effective in terms of high quality and appropriate cost (Causevic et al., 2010;Cunningham et al., 2019) .It is a process whose function is to verify the validity, completeness, quality and fulfill the specifications required for the developed software. It consists of many activities that are carried out in order to detect and correct errors in the developed system before the product is delivered (Yin & Ding, 2012).

Software testing is a large area that mainly contains technical and non-technical testing fields. It includes, for example, requirements specifications, design, implementation, and administrative problems in software engineering (Nidhra & Dondeti, 2012). To guarantee the success of software objectives, software testing should concentrate on verification and validation (Sawant et al., 2012).

Moreover, due to its importance in the success of the software, there is a growing concern in improving the implementation of this practice (Shilpa Sharma et al., 2020). Hence, focusing on this process must take into account the potential errors mentioned in the preceding paragraphs.

Despite the fact that, corporations test their software, many of them nevertheless contain bugs that differ in their effect. Whereas, sometimes it can be difficult to imagine how a missed test might have a costly effect. This could be due to the fact that testers are underqualified to undertake software testing, particularly if the product is complicated.(Whittaker, 2000).

There are numerous types of software testing that are frequently used in software development, and among these techniques are black box testing and white box testing (Freeman, 2002).

### **A. Black box testing**

Black box testing is a technique in which the tester does not know the internal structure of the code. The test is performed by executing the software. It can be a test for a function such as (integration test) or non-functional (performance test). Test cases are built according to the requirements specifications. This test can be applied to all levels of software testing operations such as unit levels, system integration, and acceptance testing. Black box testing is also known as functional, specification-based, box closing, behavioral, and I / O tests(Umar, 2020).

The black box test plays an important role in the software testing process. It validates the system functions. This test is based on the system requirements extracted from the customer. Therefore, it is possible to identify incomplete or unexpected requirements and can be addressed later. It is a test from the viewpoint of users. One of the main tasks of the black box with all inputs, whether they are valid or invalid from the customer's point of view (Nidhra & Dondeti, 2012). However, potential errors may not be taken into account, especially in the absence of a mechanism to help testers detect them.

The black box testing process takes place from the beginning of the software development. So, testing team must be involved in all stages of software development, where test scenarios must be prepared to cover these stages (Nidhra & Dondeti, 2012).

The black box test has the advantage that testers do not need to have prior knowledge of a particular programming language nor knowledge about how to implement it. Another advantage that It helps test requirements for ambiguity or

inconsistency. Hence, it is preferable that the testing process be carried out by independent testers (Nidhra & Dondeti, 2012; Dashti & Basin, 2020). In addition to, equivalence class partitioning (ECP) and boundary value analysis (BVA) are two kinds of black box testing, where ECP assumes that system be tested using valid and invalid inputs while BVA focuses on the edge (boundary) of the inputs (Hedao & Khandelwal, 2017).

## **B. White-box testing**

White box testing is a technique in which a tester is familiar with the internal structure of the software to be tested. In this test, it is necessary to know the source code because the test cases depend on the implementation of the program entity. The internal structure of the software and the testing skills are used to design test cases, and thus to fix errors discovered in the tested code (Umar, 2020).

White box testing is done at a low design level to test the operation of the software. It also applies to all parts of system development, mainly unit testing, integration testing, and system testing (Nidhra & Dondeti, 2012).

Black and white boxes testing are important, thus, both specifications and code procedures must be covered to ensure that the intended goal of building the software is achieved (Nidhra & Dondeti, 2012).

## **C. Gray-box testing**

The gray box testing is a hybrid between the white box testing and the black box testing. The gray box testing technique is used to test software specifications as well as internal work. Also, the internal structure of the software must be understood, because it is considered more than a black box test and also less than being a white box test (Sawant et al., 2012; Umar, 2020).

In addition to the foregoing, “Discovering the design defects in software, is equally difficult, for the same reason of complexity. Because software and any digital systems are not continuous, testing boundary values are not sufficient to guarantee correctness. All the possible values need to be tested and verified, but complete testing is infeasible” (Tuteja & Dubey, 2012). This may cause some potential errors. Hence, it is not easy to assess the design quality of software, because design is not expressed by

strict rules but rather through guidelines and reasoning. A successful approach to assessing design quality depends on detection strategies (Wettel & Lanza, 2008). For example high coupling can be due to error propagation reasons. Thus, coupling is an important characteristic of software systems, which has a direct impact on the coherence of the software and for this the coupling between the components of the software will affect its quality, especially with regard to ease of discovering errors (Poshyvanyk et al., 2009).

Moreover, for ease of testing and maintenance at a later time, developers need accurate knowledge of the structure of the software components and their interactions (i.e., the degree of coupling between these components should be clearly understood). This will be useful for later maintenance (Poshyvanyk et al., 2009).

From the above, we could conclude that, the more the process of depending the components of the software on each other, the greater so-called Triple effect which may be due to a defect in the design of one of these components.

## **2.4 Literature survey**

The software testing phase is one of the most important phases in the software development cycle, as it ensures that the software has met the requirements correctly and in accordance with what was documented in the requirements engineering phase. However, this process always needs to develop and improve in techniques used for that, because it requires follow-up development in building software systems.

This section gives a view on the studies related to the issues of requirements and coupling design, as it progresses by giving a historical overview of the efforts made in this field, and then by giving a survey of the previous available studies that tried to find ways and methods to solve the dilemmas of these issues.

### **2.4.1 General background**

Tsai et al (1997) presented a model of software system development life cycle named as "Test design stages processed model" (TSP), and they stressed that iterative test design stages should be incorporated at each phase of the software development



lifecycle. However, a clear mechanism for detecting potential errors has not been explored.

Bose & Srinivasan (2005) conducted a study on how to diagnose software errors. Three artificial intelligence techniques were used: "spectrum kernel, SVM, and semantic latent analysis ", and these techniques showed encouraging results. Their focus was on detecting errors during the implementation phase only. This may neglect the potential errors, such as requirements issues and issues of design coupling.

Zheng et al(2006) explained that no single technology can detect all errors. They also pointed out that the techniques for analyzing errors represented in customer reports about the problems that appear in the software. Their study were conducted on three large software systems that were developed in Nortel Networks , and they stated that statistical analysis can be effective for identifying problem modules. Also, it can be considered as a complementary to the other fault-detection techniques. This encourages that stress on potential errors must be included in the testing techniques that undertake to ensure testing of the correctness of the software.

Eichinger et al(2008) conducted a study to discover errors in the software development process. An approach has been presented in order to locate errors that are not predefined or unfamiliar. As for known and familiar errors, the approach increases the accuracy of their identification. They have used graph mining to significantly locate errors. The approach achieved excellent results, but the focus was on the implementation phase. In order to ensure quality of software systems, all stages should be verified.

Tuteja & Dubey (2012) presented a study in which they identified a list of testing methods that could be applied at each stage of the SDLC, and recommended testing at each of these stages is necessary. This reinforces the intent of this thesis, since potential errors are emphasized at an early stage.

Kaur & Singh (2014) conducted a study on analyzing and comparing a number of testing techniques, in order to determine which is better at detecting errors. However, their conclusion is that not all errors can be found in software systems. Thus, this leaves open the possibility of further research to improve the testing process.

Dhanalaxmi et al(2015) conducted a general study on error detection techniques in an effort to build high quality software, and indicated that there is no general mechanism for testing that is applicable to all software systems. They concluded that, there is a need for testing techniques to help improve commercial software.

Wong et al(2016) presented an overview of some software techniques for tracking errors, by conducting a survey of many masters and doctoral studies from 1977 to 2014, and they used the questionnaire to find relevant studies in order to identify possible errors in the techniques used. Most of the studies focused on bugs during software implementation. However, this study did not address the issues of requirements and issues of coupling design.

Yusupbekov et al(2017) developed a framework for errors prediction using data mining and intelligent decision support system technique. Associative rules are used in data mining for the traceability of objects hierarchy can be used as a basis of better analysis and gaining additional knowledge to detect and analyze errors. Where they indicated that this work can contribute to the development of requirements for software systems, and to update them in general if the associated rules are well formulated. Nevertheless potential errors still not included.

## **2.4.2 Requirement issues**

Creating requirements with excellent characteristics is one of the most important factors that contribute to the success of the program later, and for this reason, efforts have been made, and continue to be made to introduce approaches and methods that contribute to improving the generation of requirements and thus ensuring that they flow well to later stages of development.

Hagal & H.Fazzani (2013) introduced an approach aimed to reduce contradictions and ambiguity in software requirements and increase requirements consistency. To capture the degree of requirement inconsistency, use case map (UCMS) is used to visually represent all requirements, and a UML use case diagram is used to represent system functions. The approach did not pay attention to other potential errors such as incompleteness and design coupling issues that may contribute to the software weakness or failure.

Patel & Gandhi (2014) proposed an algorithm to eliminate requirements inconsistency. The algorithm checks the rules that software requirements specification must follow. So if these rules are broken, inconsistency will arise, which can then be fixed. The focus in this approach was only on requirements consistency, while the other requirements issues were neglected.

Kamalrudin & Sidek (2015) presented a review to verify requirements and consistency in order to identify gaps in the requirements specifications. They discussed the different types of techniques used in requirements inconsistency. The models used for semi-formal specifications were discussed. Map representations for searching papers related to consistency determines the technique most commonly used. The presented approach is abstract, and did not address the other issues of requirements such as completeness and necessity.

Gigante, Gargiulo, & Ficco (2015) conducted a study was based on a survey of the basic concepts that must be considered to check the requirements verification, and they proposed an approach to illustrate requirement overlapping. According to the study, there is a great difficulty regarding the completeness of the requirements, and semantic web can be a promising approach for resolving this issue. In order to find methodologies and techniques that solve most of the concerns that may occur in requirements, such as completeness, contradiction, and others, in-depth research is still required.

Stachtari et al.(2018) a model-based approach was introduced to validate requirements and translate them into system design. In the requirements phase, they used instantiating textual templates, and user defined maps to get unambiguous requirements. In the design phase, the functions of the system were built on the basis of templates of components, and they proposed a phase for checking the design model. They pointed out that the accuracy of the requirements is the foundation for all of the preceding.

Riaz et al. (2019) conducted a survey on tools and techniques used to detect ambiguity in natural language requirements from 2003 to 2013. The study revealed the popularity of using these tools and techniques based on citations, and also identified a number of the most important techniques used in this field. This study inspired the work

of this thesis to present an approach that can help improve or complement the tools that have been proposed.

Yang et al. (2019) developed a tool named "Requirements validation through an automatic prototyping" that can automatically detect the inconsistencies in the requirements by generating a number of prototypes that are tested on four case studies where the results are satisfactory, and they concluded that, the tool can be improved later to verify the requirements. This research motivated the work in this thesis to improve and develop methods for solving requirements issues, which could help to support any requirement quality improvement process.

Langenfeld et al.(2019) introduced a real-time requirement analysis approach aims to transfer the analysis problem to real-time requirements. They translate the formal requirements into an executable program, and then analyze this program as an open source program by using the "ULTIMATE REQANALYZER". They indicated that numerous problems have been identified as serious flows that lead to major problems in subsequent stages of system development. The study did not go into great detail on the testing process, especially with regard to the potential errors that are the focus of this thesis.

Narizzano et al.(2019) conducted a study on an expansion of property specifications patterns (PSOs) that considers the internal consistency of functional requirements, and the results demonstrated that the proposed approach can check specification consistency. They stated that their experiments were carried out on nearly two thousand requirements, and that their future work will concentrate on translating natural requirements into patterns.

Hadar et al.(2019) presented an empirical study on requirement inconsistency, taking practitioners' perspectives on it and attempting to identify some dilemmas that contribute to requirement inconsistency. They indicated that the strategies for managing consistency in detecting errors will greatly enhance the consistency process. The research presented in this thesis aims to improve requirements challenges, rather than just consistency.

Sulaiman et al.( 2019) investigated the inconsistency between the activity diagram and the class diagram, pointing out that the activity diagram should consistently

describe the functions of the class, and emphasizing that the inconsistency occurs when the elements to be described overlap.

Mayr-Dorn et al. (2021) presented an approach to assisting and guiding engineers in resolving the inconsistency, and they note that prototypes may contribute to improving the deviation, but they also note that this is rarely addressed in practice. This prompted the emergence of several tools that may contribute to improving this process. As a result, further research into the requirements problems is required.

Guo et al. (2021) proposed algorithms for analyzing the characteristics of natural language requirements, and they used heuristic rules to determine a number of conflicts over a number of open requirements datasets. They pointed out that the proposed algorithms gave good results, and that this work requires further investigation.

As previously stated and indicated, most studies dealt with some aspects in a private or abstract manner, in contrast to what was done in this study, which provided an organized framework for tracking the mentioned issues in order to preserve the important characteristics that the requirements must have (Table 2.1 below shows a sample of the available studies and the characteristics that has been addressed, even partially or in an abstract way).

Table 2.1. Studies and characteristics of the requirements addressed in them

Study	Requirements characteristics			
	Necessity	Correctness	Completeness	Consistency
(Hagal & H.Fazzani, 2013)				√
(Patel & Gandhi, 2014)				√
(Kamalrudin & Sidek, 2015)				√
(Gigante, Gargiulo, & Ficco, 2015)			√	
(Stachtiari et al., 2018)				√
(Riaz et al., 2019)				√
(Yang et al., 2019)				√
(Narizzano et al., 2019)				√
(Hadar et al., 2019)				√
(Sulaiman et al., 2019)				√
(Mayr-Dorn et al., 2021)				√
(Guo et al., 2021)				√
The proposal approach	√	√	√	√

### 2.4.3 Coupling issues

High coupling tracking poses a challenge for software testers, because increasing coupling represents an increase in dependency between software components, which weakens the role of software modularity, where modularity is regarded as an important concept in designing high-quality software.

Shweta Sharma & Srinivasan (2013) conducted a review work on the types of static and dynamic coupling and coherence metrics in OO systems to capture their limitations and what improvements are needed. The study found that static metrics can be used in the early stages of software systems, but they do not support testability. Testing neglects the testing of potential errors at early stages of SDLC.

Bavota, Di, Oliveto, Di Penta, et al. (2013) presented an empirical study to help software developers to learn about class coupling mechanisms. It aims to capture the extent to which metrics capture structural, semantic, and dynamic coupling. The research was carried out on three open source Java programs. The results demonstrate that a large proportion of the couplings are captured from the semantic and structural

measures that complement each other. In general, the fundamental types of coupling are not addressed. Furthermore, they ignored the design stage and concentrated solely on the implementation phase, particularly in Java programs.

Geetika & Singh (2014) investigated the validation of static and dynamic metrics for object coupling. The test was carried out on open source Java programs, with the results classified into three levels: class, method, and message. Their conclusion is that static and dynamic metrics are not the same behavior in object-oriented programs. Emphasis is placed on later stage, where coupling errors should be considered early at the design stage in order to improve the quality of the software.

Kumar & Chauhan (2015) introduced a method for prioritizing test cases, as this method focuses on the coupling information between program units in order to identify the critical unit that may affect the rest of the units and cause errors, as well as to prioritize the test cases. This approach was applied to a software case study containing ten components, where the results indicated that the approach is capable of detecting the critical component. Focusing on the priority of the test case is good in order to reduce the spread of errors, but because this study focused on the implementation phase, the potential errors at early stages of the SDLC were overlooked.

Razafimahatratra et al.( 2017) presented a method for detecting coupling types in a sequence diagram and re-designed a component with a high coupling. Furthermore, they stated that, an algorithm for coupling detection was introduced. The approach was tested in a case study, and the results were adopted in a fuzzy architecture for validation. They concluded that, their results demonstrated that the approach assist software developers to obtain high-quality software. The algorithm dealt with a definition of the types of coupling in abstract manner, without taking the detection process into account.

Kamble et al.( 2017) presented the identification of a coupling pattern when trying to integrate parts of software system. They added some features to the pattern to show the complexity of coupling resolving. They recommended that more research in this field should be conducted.

Alenezi & Magel (2017) proposed a new coupling metric for software entities that combines structural and semantic relationships. An empirically study was conducted on three different applications, and they concluded that the new coupling metric is useful

for classes impact changes. They stated that more studies on the proposed metric are required on software re-modularization and refactoring.

Anwer et al. (2017) carried out an empirical study on the effect of coupling on fault prediction errors, which was conducted on seven open source Java programs. Three of the coupling metrics were chosen: afferent coupling (CA), efferent coupling (CE) and coupling between objects (CBO). The results showed that, the CE has the best correlation with the defects among the selected metrics. This study does not include design coupling in more details, especially in testing.

Fregnan et al. (2019) conducted a survey on most of the coupling relationships and metrics that were proposed in studies from 2002 to 2017. They introduced a complete classification of these relationships and categorized the coupling relationships into four groups: structural, dynamic, semantic, and logical. They also added a fifth classification that includes coupling metrics, but it is not incorporated in other classifications. In addition to that they also clarified the tools through which the coupling relationships were discovered. This effort did not focus on design stage testing, especially design coupling.

Rizwan et al. (2020) conducted a study based on evaluating seven coupling metrics on their impact on software fault prediction. Support vector machine is used to classify errors prediction. Experiments were performed on 87 different data sets to evaluate these metrics of errors detection . The results of these experiments demonstrated that coupling metrics are effective in detecting errors. The testing process does not indicate to the known types of coupling that take place between the components, (i.e. the testing issues of coupling design not clearly included).

Yusuf & Hammad (2020) suggested an approach to measure coupling between classes by tokens extraction for the classes, and then matching these tokens with other classes according to coupling measures metrics. This work encouraged the preparation of a more detailed framework to address this abstract work.

Furthermore, Miholca & Onet-Marian (2020) stated that, several studies have attempts to develop metrics to measure dependence between software source files. The focus was on the implementation phase, and they indicated the difficulty of tracking all types of coupling. However, more effort is required to move from abstract work to



simplified implementation work that makes it easier for developers to track the process of coupling between software components.

In light of the foregoing, the majority of studies in the existing literature have concentrated on the implementation phase or have addressed these challenges in an abstract manner. This may not take into account the errors that occur as a result of neglecting potential errors. Therefore, introducing a framework that enhances the testing process early in the SDLC, especially with regard to potential errors is required. This framework can be considered as complement to the testing process rather than a substitute for existing testing techniques.

## Chapter 3

### The proposed approach

As mentioned before, potential fault testing is a crucial process in the SDLC that effects the cost and development time of the software. Where the scope of this study will focus on the early stages of SDLC: Requirements and Design. At the requirement stage, the work idea concentrates on the validity of the requirements by applying some solutions for testing requirements such as necessity, correctness, completeness and consistency. So the discovery of errors at the early stage reduces the spread of errors during the subsequent stages. On the other side, the design coupling issues are the primary cause for potential errors at the design stage because the strong interconnection between software components which causes scattered errors between these components. The proposed framework is depicted in Figure 3.1.

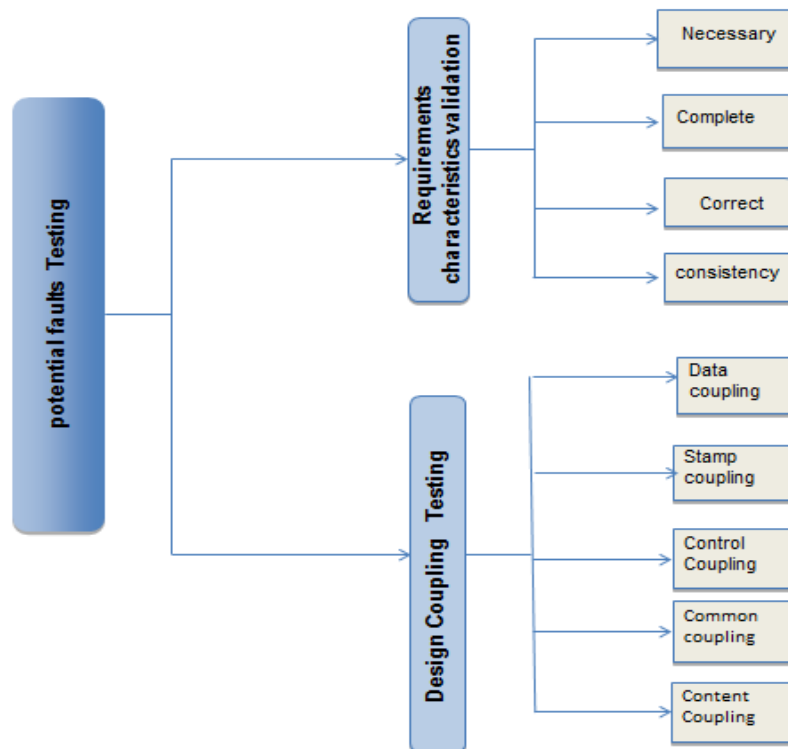


Figure3.1: Overview of the proposed framework

### 3.1 Requirements validation process improvement

The requirements stage, as is well known, is the first and most important phase in the SDLC. Completing this stage successfully improves software quality and decreases development time and cost. Requirements success must take into account that it may pose a significant challenge, especially with regard to validating requirements properties, which if neglected will result in potential errors. Completeness, necessity, correctness, and consistency are examples of these characteristics. Therefore, successful requirements validation is an integrated process for improving potential faults testing. Hence, requirements cannot be considered excellent unless they meet the mentioned characteristics. Therefore, enhancing these characteristics is one of this study's objectives.

- **Necessity**

"Necessary" is defined by Merriam-Webster as "so important that you must do it or have it absolutely necessary" (Merriam-Wbster, 2021). Therefore, a requirement to be necessary means that there will be violation in the system's functionality if this characteristic is missed. As previously stated, the requirements stage is an integrated process, so it will be misleading if the requirements engineer puts some requirements which he assume necessary, and therefore he documented them by chance or consider that they are important to the system. While the mistake is to add them without referring to their sources (i.e. stakeholders or documents)(Saavedra et al, 2013) .

In addition, determining requirements and knowing their sources is an essential and important process. This process will not be complete without clarity and correctness of these requirements, which is a difficult process that requires more effort to maintain what is mentioned in several researches..

The relationship between requirements and their sources is depicted in Table 3.1.

Table 3.1: Requirement resource table

	Req <sub>1</sub>	Req <sub>2</sub>	Req <sub>3</sub>	.....	Req <sub>n</sub>
Source <sub>1</sub>	x				
Source <sub>2</sub>			x		X
⋮					
Source <sub>n</sub>					

The table, for example, illustrates the sources 1,2, and n, as well as their related requirements, where the character "x" denotes the relationship between the requirement and its resource.

- **Completeness**

When something is said to be complete, it usually means that no necessary information are missing to be fit for use or serve its intended purpose. The Merriam-Webster dictionary defines complete as “having all necessary parts, elements, or steps” (Merriam-Wbster, 2021). This is consistent with our intuitive understanding. Moreover, for a set of requirements, this means that there are no other requirements necessary for the set of related requirements to fulfill their collective purpose or mission of defining what a given system must be and do, with respect to some specified context(Carson et al., 2004) (Marques & Yelisetty, 2019)

Hence, completeness totally depends on the understanding of stakeholders’ requirements and meeting these requirements in the way that they need nothing else.

To confirm the completeness of the requirements, the process of clarifying them is important. So, UML activity diagram with swimlanes is proposed to clarify the requirements, where the interaction between the activities in each requirement, and who are responsible for each zone are illustrated. Figure.3.2 depicts an activity diagram with swimlanes.

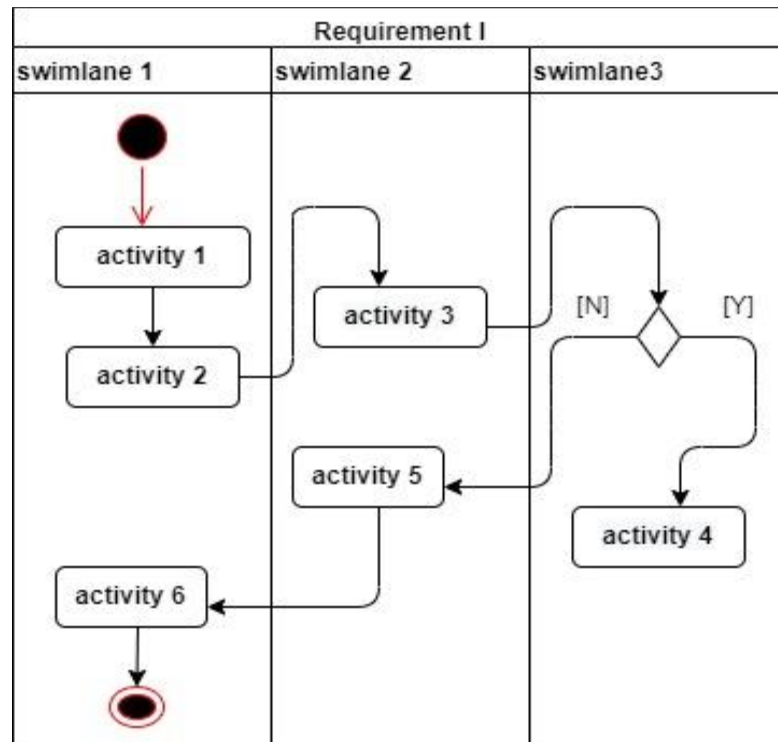


Figure 3.2: Activity diagram with swimlanes example

- **Correctness**

It means that the correctness of the requirements matches (reflects) the user's real needs, and it emphasizes that they are correct, unambiguous and not repetitive. So, its correctness will affect the part of the program related to it, and then a program is considered correct if it behaves as expected on each element of its input domain (Zowghi & Gervasi, 2003;Kamaludin & Sidek, 2015). Furthermore, the word “correct” may have many interpretations (i.e. need more clarification) (Gigante, Gargiulo, Ficco, et al., 2015) .This gives us the motivation to study the requirements correctness in more depth.

Hence, this characteristic can be considered as a complementary characteristic to the two processes of Necessity and Completeness. So each requirement will be improved if it has been declared as in the figure 3.2 and table 3.1. Moreover, to complement the necessity table, its validity must be confirmed by the relevant stakeholder.

- **Consistency**

The process of requirements consistency is regarded as critical in order to guarantee accuracy. This means that no requirements in a specification contradict each other, where all terms have the same interpretation (Zowghi & Gervasi, 2003). This requirement uses terms consistently with their specified meanings, so requirement should not contradict with other requirement, as well as be understood precisely in the same way by every person who reads (Sommerville, 2011; Acharya et al., 2005).

Requirement descriptions must be complete, with no ambiguity or carelessness for any of the governing conditions. Furthermore, some requirements must be performed after others or have some common parts. Therefore, the dependency between these requirements and the common part in them should be considered and not violated. These parts can be activity(s), conditions, or rules. For example, in order for a student to register his subjects, he must first complete his admission process by considering the required conditions. This means that the process of tracking the dependency of the requirements on each other is necessary, as well as knowing the common parts between them, which should be considered and not violated in any of them. Table 3.2 illustrates a requirement consistency example.

Table 3.2: An example of consistency traceability table

Requirements	Requirements			common activity, condition or consistency rules
	Req <sub>1</sub>	Req <sub>2</sub>	Req <sub>3</sub>	
Req <sub>1</sub>		x	x	
Req <sub>2</sub>	x			
Req <sub>3</sub>		x		

The common activity, condition, or consistency rules column in the preceding table illustrates the consistency action(s) that should not be neglected in the related requirements.

In addition, requirements gathering in a complete form will ensure that requirements are excellent and free of violations. Here, are some factors that increases the degree of requirements violations:

1. Stakeholders have different views.

2. Contradiction between the original (root) and its dependent requirements is not allowed. For example "Req1" must be done before "Req2" and must not violate any of the rules that "Req1" subjected to.

Figure 3.3 below clarifies the activities that are common in more than one requirement (i.e., what activities or conditions must be contained in each of the two requirements). The dotted activity shows the common process that must be included in each of the two requirements. For instance, activity "a4" in "requirement<sub>i</sub>" is the same as the activity "b7" in "requirement<sub>j</sub>".

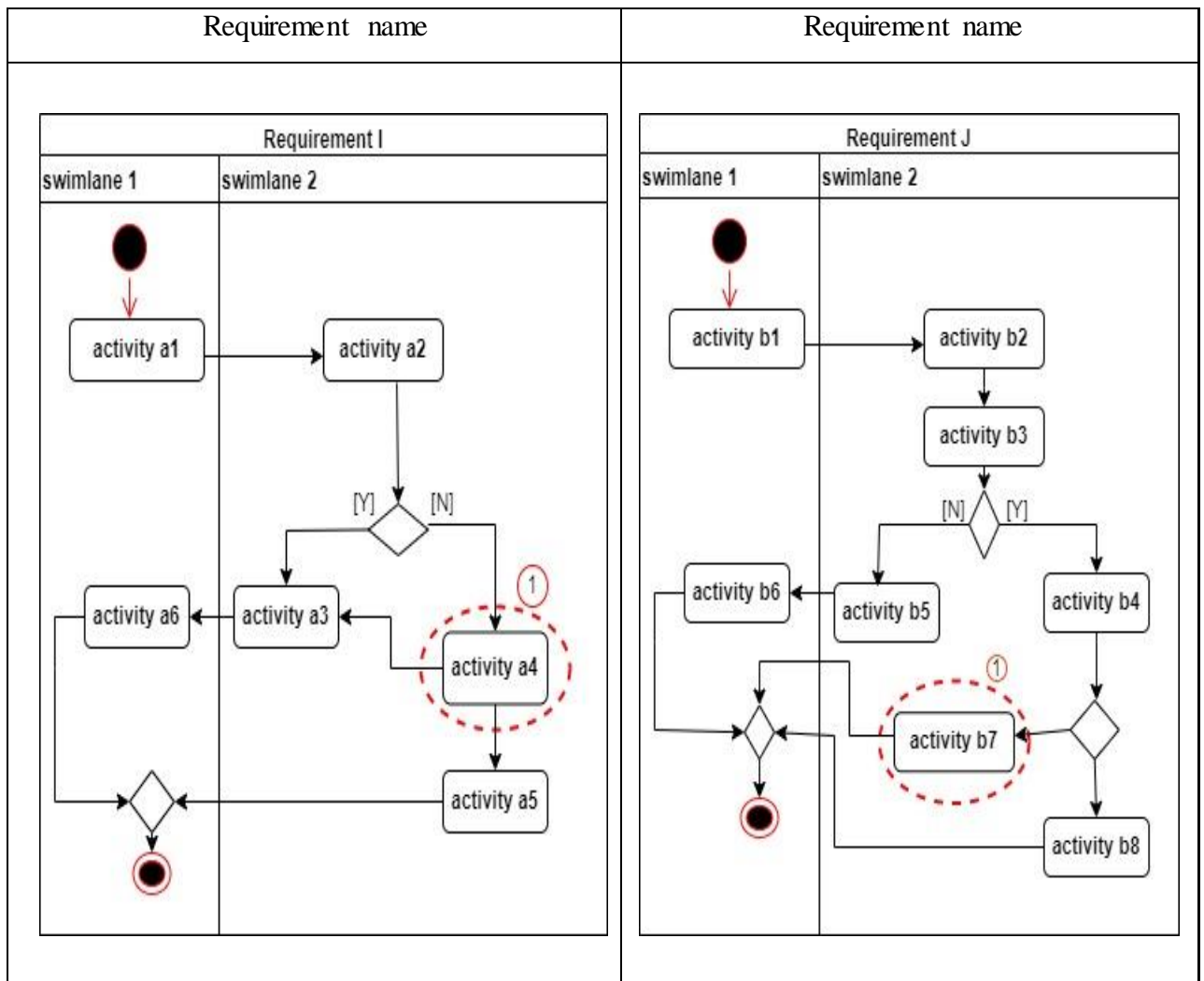


Figure 3.3: Activity diagram with swimlane to trace the consistency between requirements

Where the dotted circle in the preceding diagram depicts the consistency activity, which represents the process or rules in both components that should not be ignored(common).

## 3.2 Coupling and its testing process optimization

Due to the difference between the requirements stage and the design stage, the design stage as it is known focuses on how the system will be built. The coupling testing process should be done on each module (component), where components represent parts of a system or application. The high coupling between system components leads to a marked increase in cost as well as errors propagation. While low coupling (i.e. high cohesion) leads to high quality, cheaper developed, and easier maintained system (Razafimahatratra et al., 2017).

In addition, coupling is somehow important in software as it builds the internal interactions between software components which allows it to do what it was designed to do. But the increase in coupling will play an essential role in faults spreading between the coupled components. However, some software developers see that coupling is necessary, so the problems result from these interactions must be thoroughly validated. Coupling can come in many types: Content coupling, Common coupling, External coupling, Control coupling, Stamp coupling, and Data coupling (Razafimahatratra et al., 2017).

### 3.2.1 Coupling precedence

To study coupling, a precedence is needed in order to simplify tracking of coupling testing process. Figure 3.4 shows an example of components interactions and levels of precedence. Where a system is the root and the first level branches represents the main components, then their dependent branches and so on. So every higher level component will affect its dependent processes. Beside that it can be seen from the hierarchy that there is an interaction dependency between component "A" and component "B". Hence any fault in component "B" will affect component "A", as well as its dependent components, because as shown in the figure, the dotted arrow from component "A" to component "B" means that component "A" depends on the component "B".



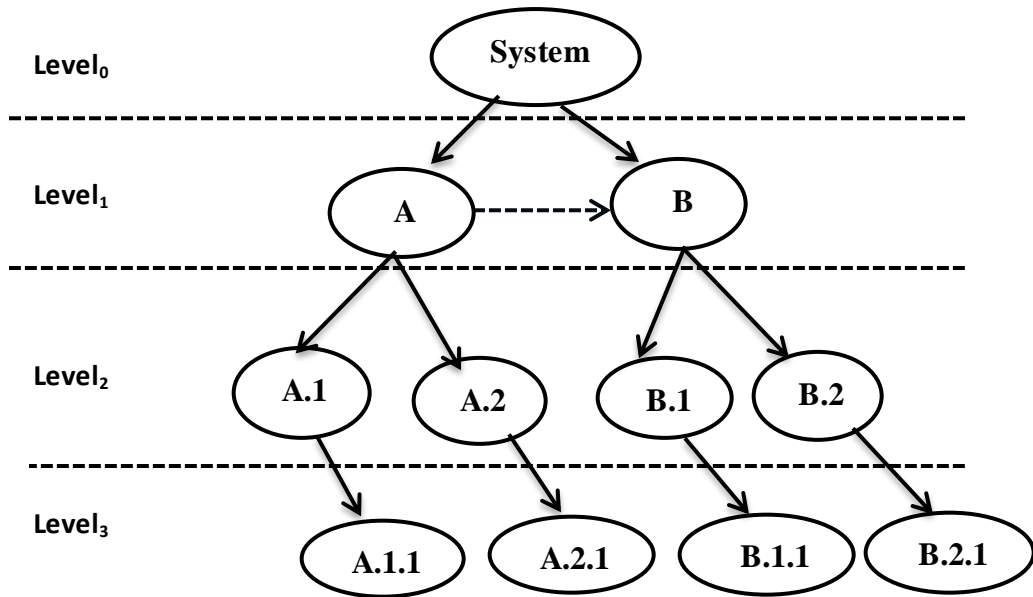


Figure 3.4: An example of coupling precedence

### 3.2.2 Types of coupling

#### A. Data coupling

Whereas data coupling represents the process of interacting between components by passing data between them (i.e. the dependence of a component on data passed to it from another component) (Schach, 2011). Figure 3.5 illustrates such process.

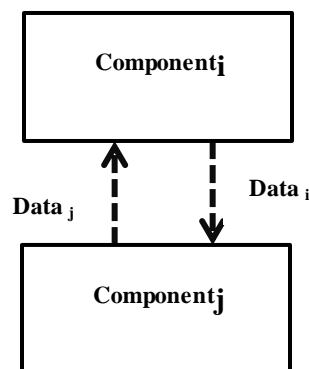


Figure 3.5: Data coupling between different software components

To follow the process of data coupling and reduce its impact on the propagation of errors, table 3.3 below is proposed to track the interaction process based on this type of coupling. It shows the traceability of the data exchange process between two

components (for example, Component<sub>i</sub> and Component<sub>j</sub>), and the testing process required in that exchange.

Table 3.3: Data coupling testing process

Testing Date	Testing Time	Tester	Type of test Pass/fail																																					
			<input type="radio"/> Test to pass <input type="radio"/> Test to fail																																					
<b>Component<sub>i</sub></b>		<b>Component<sub>j</sub></b>																																						
<b>parameter (s )</b> <table border="1" style="width: 100%; height: 20px; border-collapse: collapse;"> <tr> <td style="width: 25%;"></td> <td style="width: 25%;"></td> <td style="width: 25%;"></td> <td style="width: 25%;"></td> </tr> </table>					→	<p><b>Testing</b>  <i>Boundary Value Analysis:</i>  <i>Inputs edge: -----</i></p> <div style="text-align: center;"> <math>\leftarrow</math> <table style="display: inline-table; border: none;"> <tr> <td style="text-align: center;"><math>&lt;min</math></td> <td style="text-align: center;"><math>\geq min</math> and <math>\leq max</math></td> <td style="text-align: center;"><math>&gt;max</math></td> </tr> <tr> <td style="text-align: center;"><i>Invalid</i></td> <td style="text-align: center;"><i>Valid</i></td> <td style="text-align: center;"><i>Invalid</i></td> </tr> </table> <math>\rightarrow</math> </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 33%;">partitions</th> <th style="width: 33%;">Input value</th> <th style="width: 33%;">Expected result</th> </tr> </thead> <tbody> <tr> <td><math>&lt; min</math></td> <td></td> <td>fail</td> </tr> <tr> <td><math>min</math></td> <td></td> <td>pass</td> </tr> <tr> <td><math>max</math></td> <td></td> <td>pass</td> </tr> <tr> <td><math>&gt;max</math></td> <td></td> <td>fail</td> </tr> </tbody> </table> <p><i>Equivalence Class Partitioning:</i>  <i>valid type-----</i></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2" style="text-align: center;"><i>valid</i></th> <th colspan="2" style="text-align: center;"><i>Invalid</i></th> </tr> <tr> <th style="width: 25%;"><i>Partition1</i></th> <th style="width: 25%;"><i>Partition2</i></th> <th style="width: 25%;"><i>Partitions1</i></th> <th style="width: 25%;"><i>Partitions2</i></th> </tr> </thead> <tbody> <tr> <td> </td> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>		$<min$	$\geq min$ and $\leq max$	$>max$	<i>Invalid</i>	<i>Valid</i>	<i>Invalid</i>	partitions	Input value	Expected result	$< min$		fail	$min$		pass	$max$		pass	$>max$		fail	<i>valid</i>		<i>Invalid</i>		<i>Partition1</i>	<i>Partition2</i>	<i>Partitions1</i>	<i>Partitions2</i>				
$<min$	$\geq min$ and $\leq max$	$>max$																																						
<i>Invalid</i>	<i>Valid</i>	<i>Invalid</i>																																						
partitions	Input value	Expected result																																						
$< min$		fail																																						
$min$		pass																																						
$max$		pass																																						
$>max$		fail																																						
<i>valid</i>		<i>Invalid</i>																																						
<i>Partition1</i>	<i>Partition2</i>	<i>Partitions1</i>	<i>Partitions2</i>																																					
<b>Summary of failures :</b> ----- ----- -----																																								
<b>Test Result:</b> <input type="radio"/> Pass <input type="radio"/> Fail																																								

Testing date: The date of the test.

Testing time: Time of the test.

Tester: Tester name.

Pass/Fail: The type of test being performed.

Test Result: The result of the test.

Summary of failures: Write down the found faults.

Component: Component name.

Testing: the required test types.

Boundary Value Analysis (BVA): Boundary testing of data sent from one component to another (permissible range/inputs edge). It represents the inputs' edge, which can be the length of the parameter, the value of the parameter, etc.

Equivalence Class Partitioning: Splitting exchanged data into different equivalence data classes (valid and invalid data inputs).

## B. Stamp coupling

Stamp coupling occurs between components when data are passed by parameters using a data structure containing arguments (data items), where the called component may not operate on all the data items of the received data structure (Fregnan et al., 2019).

Figure 3.6 below illustrates an example of the data structure that is passed from one component to another. Where the first component sends student data to the second component, and the second component retrieves the student's GPA from the data structure (student record) sent to it.

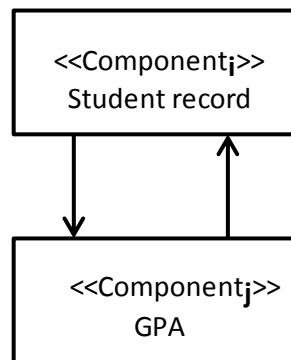


Figure 3.6: Stamp coupling between two components

The proposed stamp coupling is illustrated in Table 3.4 below. Furthermore, the passed data structure may contain fields that are unnecessary for the receiving component. The table contains the required test processes, which were also explained in the previous type of coupling. This type of coupling testing requires testing the data elements that will be used by the destination component, as shown in the table as "Data structure/object contents."

Table 3.4: Stamp coupling testing process

Testing Date	Testing Time	Tester	Type of test Pass/fail																										
			<input type="radio"/> Test to pass <input type="radio"/> Test to fail																										
Component <sub>i</sub>		Component <sub>j</sub>																											
<b>Data structure   object name...</b> <b>Contents</b> <div style="border: 1px solid black; width: 100px; height: 15px; margin: 5px 0;"></div> <div style="border: 1px solid black; width: 100px; height: 15px; margin: 5px 0;"></div> <div style="border: 1px solid black; width: 100px; height: 15px; margin: 5px 0;"></div> <div style="border: 1px solid black; width: 100px; height: 15px; margin: 5px 0;"></div> <div style="border: 1px solid black; width: 100px; height: 15px; margin: 5px 0;"></div>	→	<b>Testing (For each data item)</b> <b>Boundary Value Analysis</b> <b>Inputs edge: -----</b>  <div style="text-align: center;"> <math>&lt;min</math>    <math>\geq min</math> and <math>\leq max</math>    <math>&gt;max</math>  </div> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th><i>partitions</i></th> <th><i>Input value</i></th> <th><i>Expected result</i></th> </tr> </thead> <tbody> <tr> <td><i>&lt; min</i></td> <td></td> <td>fail</td> </tr> <tr> <td><i>min</i></td> <td></td> <td>pass</td> </tr> <tr> <td><i>max</i></td> <td></td> <td>pass</td> </tr> <tr> <td><i>&gt; max</i></td> <td></td> <td>fail</td> </tr> </tbody> </table> <b>Equivalence Class Partitioning:</b>  <b>Valid Type: -----</b>  <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2"><i>valid</i></th> <th colspan="2"><i>Invalid</i></th> </tr> <tr> <th><i>Partition1</i></th> <th><i>Partition2</i></th> <th><i>Partitions1</i></th> <th><i>Partitions2</i></th> </tr> </thead> <tbody> <tr> <td> </td> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>	<i>partitions</i>	<i>Input value</i>	<i>Expected result</i>	<i>&lt; min</i>		fail	<i>min</i>		pass	<i>max</i>		pass	<i>&gt; max</i>		fail	<i>valid</i>		<i>Invalid</i>		<i>Partition1</i>	<i>Partition2</i>	<i>Partitions1</i>	<i>Partitions2</i>				
<i>partitions</i>	<i>Input value</i>	<i>Expected result</i>																											
<i>&lt; min</i>		fail																											
<i>min</i>		pass																											
<i>max</i>		pass																											
<i>&gt; max</i>		fail																											
<i>valid</i>		<i>Invalid</i>																											
<i>Partition1</i>	<i>Partition2</i>	<i>Partitions1</i>	<i>Partitions2</i>																										
<b>Summary of failures :-----</b> <div style="border-top: 1px dashed black; height: 20px; width: 100%;"></div>																													
<b>Test Result:</b> <input type="radio"/> pass <input type="radio"/> fail																													

Note: The table contents description is similar to that found in table 3.3

### C. Control coupling

This type of coupling is meant to provide one component control over another's implementation (i.e., the component is affected by the data sent by the other component, and any change will affect the controller and controlled components) (Maia & Souza, 2018). In an example of control coupling, a component that retrieves either student name or GPA depending on the value of a flag/control data (Search criteria) is illustrated (Figure3.7).

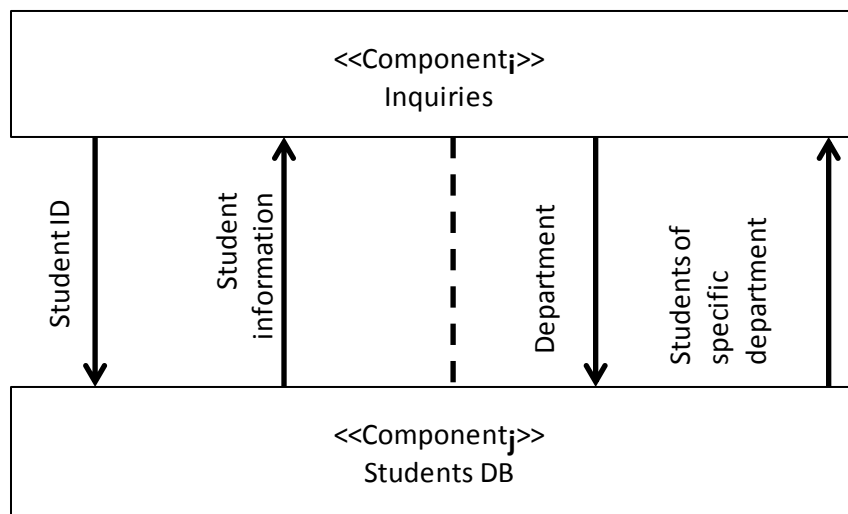



Figure 3.7: Control coupling between two components example

Table3.5 bellow demonstrates the proposed form of control coupling, where the process of interaction between two components depends on the flag (control data) passed from the one component to another (for example, between "component<sub>i</sub>" and "component<sub>j</sub>"). All paths of the second component will be tested to ensure that the interaction does not cause any errors.

Table 3.5: Control coupling testing process

Testing Date	Testing Time	Tester	Type of test Pass/fail
			<input type="radio"/> Test to pass <input type="radio"/> Test to fail
<b>Component<sub>i</sub></b>		Component <sub>j</sub>	
<b>Flag/control data</b> <div style="border: 1px solid black; width: 80px; height: 20px; margin-left: 100px;"></div>		<i>Path 1: Path name</i>  <b>Test Result:</b> <input type="radio"/> Pass <input type="radio"/> fail  .....  <b>Path 2: Path name</b>  <b>Test Result:</b> <input type="radio"/> Pass <input type="radio"/> fail  .....  <i>Path N: Path name</i>  <b>Test Result:</b> <input type="radio"/> Pass <input type="radio"/> fail	
<b>Summary of failures :</b>			
<b>Test Result:</b> <input type="radio"/> pass <input type="radio"/> fail			

Where the control coupling process test represents the necessity of executing the required command from the first component. This may contain a flag that determines the execution process, in which the second component receives the command and the ability to execute is confirmed, then the first component is informed of the result of the process, whether it was done correctly or not.

#### D. Common coupling

In this type of coupling interactions between coupled components comes from using the same common variable (Schach, 2011 ; Fregnan et al., 2019). Therefore, any modification in this common variable will be spread inadvertently across all the interacted components. the test should focus on changing or modifying that variable in order to avoid having a wrong influence on the interacting components. Figure 3.8 depicts the idea of such a coupling.

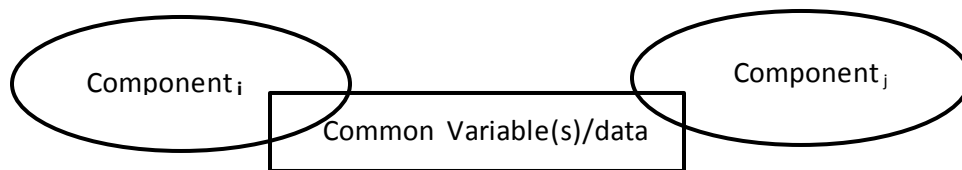
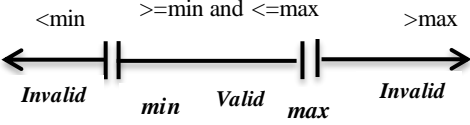
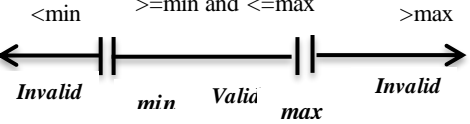


Figure 3.8: Common Coupling between two components

Table 3.6 bellow shows the precedence in modifying the shared data and it may happen for a component to deal with shared data more than once. So, after the interaction process between the two components, the common variable(s)/data should be tested to see if they are changed in an undesirable way or not working properly. Furthermore, this traceability must be performed on all interacting and shared components of this variable(s)/data.

Table 3.6: Common coupling testing process

	Testing Date	Testing Time	Tester	Type of test Pass/fail																																																												
				<input type="radio"/> Test to pass <input type="radio"/> Test to fail																																																												
<b>Common(Shared) variable(s)</b>	<b>Component<sub>i</sub></b>		<b>Component<sub>j</sub></b>																																																													
<div style="border: 1px solid black; width: 30px; height: 15px; display: inline-block;"></div>	<p><b>Testing</b>  <i>Boundary Value Analysis:</i>                      Inputs edge: -----</p>  <table border="1" data-bbox="486 784 949 952"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt; min</td> <td></td> <td>fail</td> </tr> <tr> <td>min</td> <td></td> <td>pass</td> </tr> <tr> <td>max</td> <td></td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td></td> <td>fail</td> </tr> </tbody> </table> <p><i>Equivalence Class Partitioning:</i>                      Valid type: -----</p> <table border="1" data-bbox="486 1041 949 1209"> <thead> <tr> <th></th> <th>valid</th> <th>Invalid</th> </tr> </thead> <tbody> <tr> <td>Partitions1</td> <td></td> <td></td> </tr> <tr> <td>Partitions2</td> <td></td> <td></td> </tr> <tr> <td>Partitions3</td> <td></td> <td></td> </tr> <tr> <td>Partitions4</td> <td></td> <td></td> </tr> </tbody> </table> <p>Common variable test(s):  <input type="radio"/> pass  <input type="radio"/> fail</p>		partitions	Input value	Expected result	< min		fail	min		pass	max		pass	>max		fail		valid	Invalid	Partitions1			Partitions2			Partitions3			Partitions4			<p><b>Testing</b>  <i>Boundary Value Analysis:</i>                      Inputs edge: -----</p>  <table border="1" data-bbox="981 862 1460 1030"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt; min</td> <td></td> <td>fail</td> </tr> <tr> <td>min</td> <td></td> <td>pass</td> </tr> <tr> <td>max</td> <td></td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td></td> <td>fail</td> </tr> </tbody> </table> <p><i>Equivalence Class Partitioning:</i>                      Valid type: -----</p> <table border="1" data-bbox="981 1120 1444 1288"> <thead> <tr> <th></th> <th>valid</th> <th>Invalid</th> </tr> </thead> <tbody> <tr> <td>Partitions1</td> <td></td> <td></td> </tr> <tr> <td>Partitions2</td> <td></td> <td></td> </tr> <tr> <td>Partitions3</td> <td></td> <td></td> </tr> <tr> <td>Partitions4</td> <td></td> <td></td> </tr> </tbody> </table>		partitions	Input value	Expected result	< min		fail	min		pass	max		pass	>max		fail		valid	Invalid	Partitions1			Partitions2			Partitions3			Partitions4		
partitions	Input value	Expected result																																																														
< min		fail																																																														
min		pass																																																														
max		pass																																																														
>max		fail																																																														
	valid	Invalid																																																														
Partitions1																																																																
Partitions2																																																																
Partitions3																																																																
Partitions4																																																																
partitions	Input value	Expected result																																																														
< min		fail																																																														
min		pass																																																														
max		pass																																																														
>max		fail																																																														
	valid	Invalid																																																														
Partitions1																																																																
Partitions2																																																																
Partitions3																																																																
Partitions4																																																																
<p><b>Summary of failures :</b> -----                      -----</p> <p><b>Test result:</b>  <input type="radio"/> pass  <input type="radio"/> fail</p>																																																																

Note: The table contents description is the same as expressed previously in table 3.3.



## E. Content coupling

This type of coupling occurs when one component modifies or depends on the internal work of another component (Fregnan et al., 2019). The aggregation relationship in an object oriented can be considered as an example of content association, where the base class affects the aggregated class. In the composition aggregation, for example, destroying a base class means destroying its aggregated classes. Figure 3.9 depicts the idea of content coupling (where the direction head of dashed arrow indicates to the controller component). Moreover, if any change or modification is made to the controlled component, it may play the role of controller component.

Therefore, the dependent component must have the necessary protection to ensure that it is not affected by the component interacting with it, and does not allow it to be changed in it unless there is a permissible necessity.

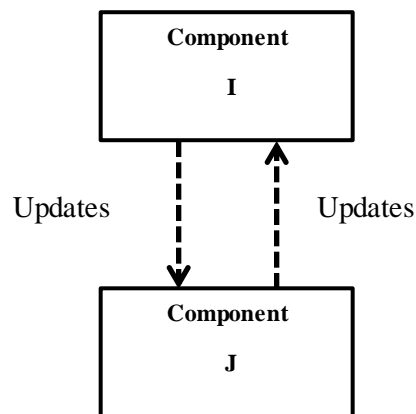


Figure 3.9: content coupling between two components

The process of the effect of one component on the internal composition of the other component is depicted in Table 3.7 below (i.e., In other words, the negative influence between the interacting components is not allowed). So, it must be ensured that this interaction does not negatively affect the interacted components, whether by modification or deletion.

Table 3.7: Content coupling testing process

Testing Date	Testing Time	Tester	Type of test Pass/fail																											
			<input type="radio"/> Test to pass <input type="radio"/> Test to fail																											
<b>Component<sub>i</sub></b>		<b>Component<sub>j</sub></b>																												
<b>Command</b> <div style="border: 1px solid black; width: 100px; height: 30px; margin: 10px auto;"></div>																														
		<p><b>Testing</b>  <i>Boundary Value Analysis:</i>  <i>Inputs edge: -----</i></p> <p style="text-align: center;"> <math>&lt;min</math>    <math>\geq min</math> and <math>\leq max</math>    <math>&gt;max</math>  </p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th><i>partitions</i></th> <th><i>Input value</i></th> <th><i>Expected result</i></th> </tr> </thead> <tbody> <tr> <td><i>&lt;min</i></td> <td></td> <td>fail</td> </tr> <tr> <td><i>min</i></td> <td></td> <td>pass</td> </tr> <tr> <td><i>max</i></td> <td></td> <td>pass</td> </tr> <tr> <td><i>&gt;max</i></td> <td></td> <td>fail</td> </tr> </tbody> </table> <p><i>Equivalence Class Partitioning:</i>  <i>valid type-----</i></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th colspan="2"><i>valid</i></th> <th colspan="2"><i>Invalid</i></th> </tr> <tr> <th><i>Partition1</i></th> <th><i>Partition2</i></th> <th><i>Partitions1</i></th> <th><i>Partitions2</i></th> </tr> </thead> <tbody> <tr> <td> </td> <td> </td> <td> </td> <td> </td> </tr> </tbody> </table>		<i>partitions</i>	<i>Input value</i>	<i>Expected result</i>	<i>&lt;min</i>		fail	<i>min</i>		pass	<i>max</i>		pass	<i>&gt;max</i>		fail	<i>valid</i>		<i>Invalid</i>		<i>Partition1</i>	<i>Partition2</i>	<i>Partitions1</i>	<i>Partitions2</i>				
<i>partitions</i>	<i>Input value</i>	<i>Expected result</i>																												
<i>&lt;min</i>		fail																												
<i>min</i>		pass																												
<i>max</i>		pass																												
<i>&gt;max</i>		fail																												
<i>valid</i>		<i>Invalid</i>																												
<i>Partition1</i>	<i>Partition2</i>	<i>Partitions1</i>	<i>Partitions2</i>																											
<b>Summary of failures :</b> ----- -----																														
<b>Test Result</b> <input type="radio"/> pass <input type="radio"/> fail																														

Note: The table contents description is the same as expressed previously in table 3.3.

## Chapter 4

### Case Study

In this chapter, a simple library system is used to demonstrate how the framework is applied.

#### 4.1 Scenario

This system aims to replace the usage of manual procedures in a library, such as Book borrowing records, List of participants, Books records, and etc. What makes it easier to manage the library and keep resources in the same time. This system will be operated by an admin who can access using his credentials with full permissions to add, remove, lend, and organize books. Also, keep informed about all records without any extra effort.

##### Admin responsibilities

- a. Add/Remove books with all related information.
- b. Lend/Return books and update system data.
- c. Add/Remove participants and organize participants needs.
- d. Help participants to find their needed books.

##### Book lending policy

- e. Books lent to participants only.
- f. Participant can borrow only three books simultaneously.
- g. No books can be borrowed for more than two weeks.
- h. There will be a penalty if the participant kept the book for more than two weeks.

##### Operational procedures

- a. Anyone who registers on the systems is considered a participant, and thus has access to all library services. Registration begins with the filling of a registration form, after which the administrator enters these data into the system, makes a user profile, then issues a participation card.
- b. Once he get his participation card he can borrow books as needed following book lending policy.
- c. The admin enters the borrowing information (participant, book, borrowing date).
- d. Participant can renew borrowing date for the book once more before retrieval.

- e. There will be a penalty if the participant keeps the book for more than two weeks without renewing the borrowing.
- f. Participant can fill Suggestions/Notes form if he had any.

#### 4.1.1 Requirements characteristics traceability

The requirements phase is the first and most important phase in the SDLC, as is well known. Successful completion of this phase increases software quality while also reducing development time and expense. In this section, the stages of the proposed framework will be applied in relation to the issues of the requirements stage aimed at improving the requirements in terms of Completeness, Necessity, Correctness, and Consistency.

- Necessary

The relationships between requirements and their sources are included in table 4.1

Table 4.1: requirement resource table

	Add new Book	Borrow Book	Search Book	Remove Book	Browse books info	Book returns
Librarian	x	x	x	x	x	x
Participant		x	x		x	x

- **Completeness**

Figure 4.1 to Figure 4.6 depict the description of the above requirements, and illustrate the completion of activities flow .

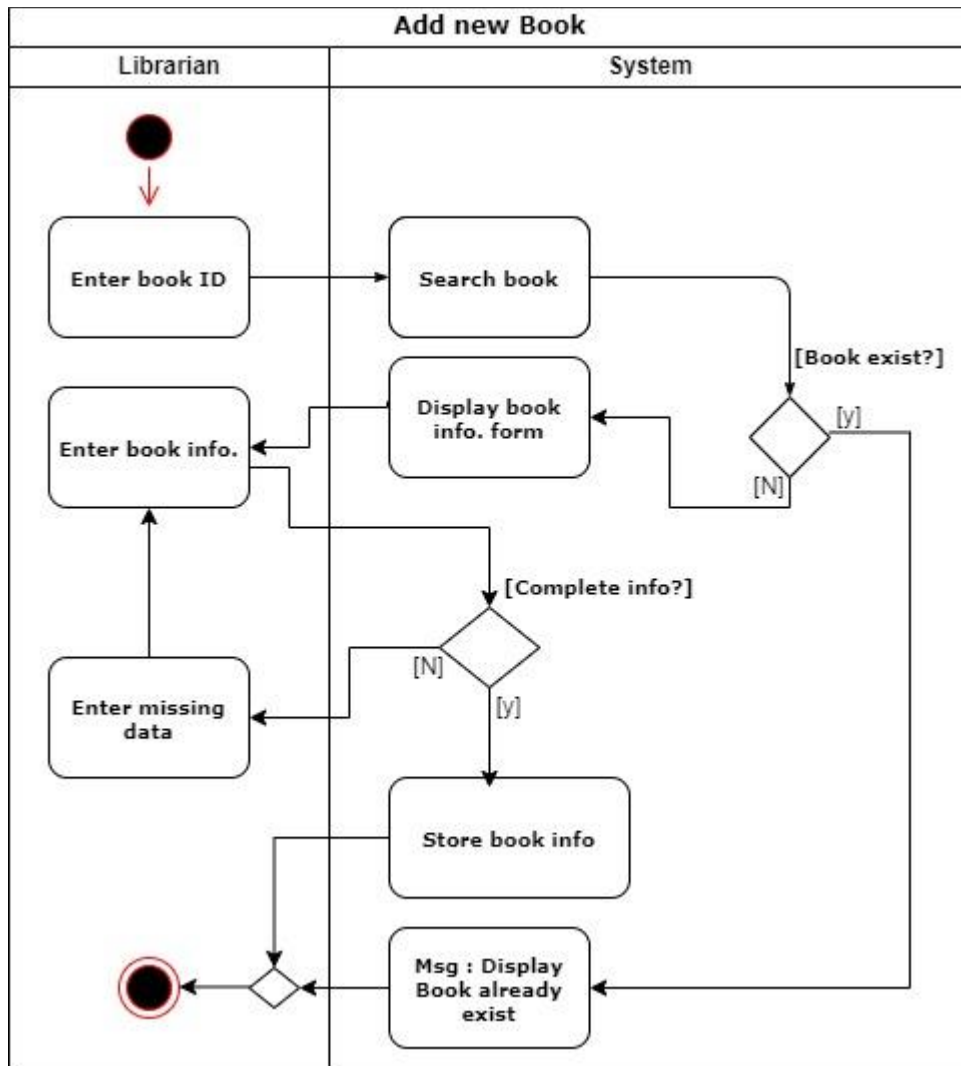


Figure 4.1: Activity diagram with swimlane for "Add a new book"

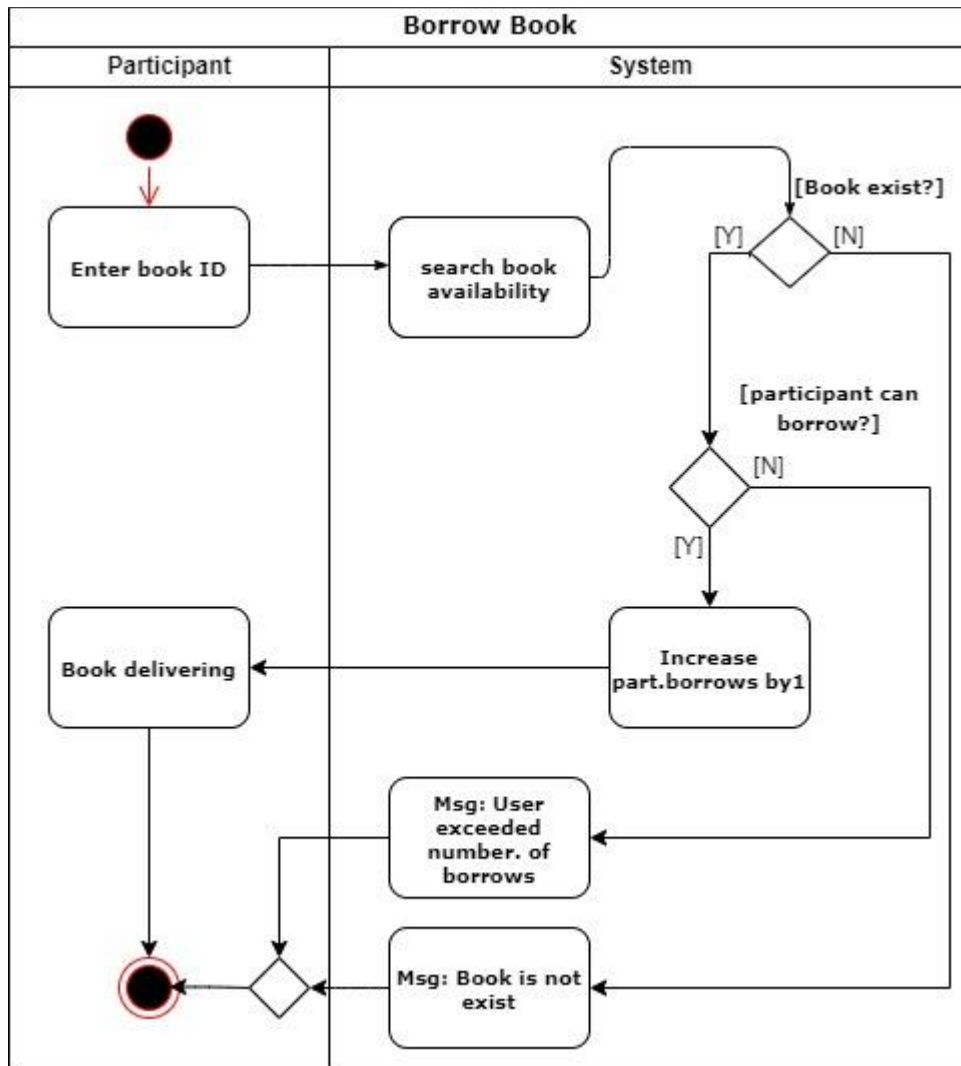


Figure 4.2: Activity diagram with swimlane for "Borrow a book"

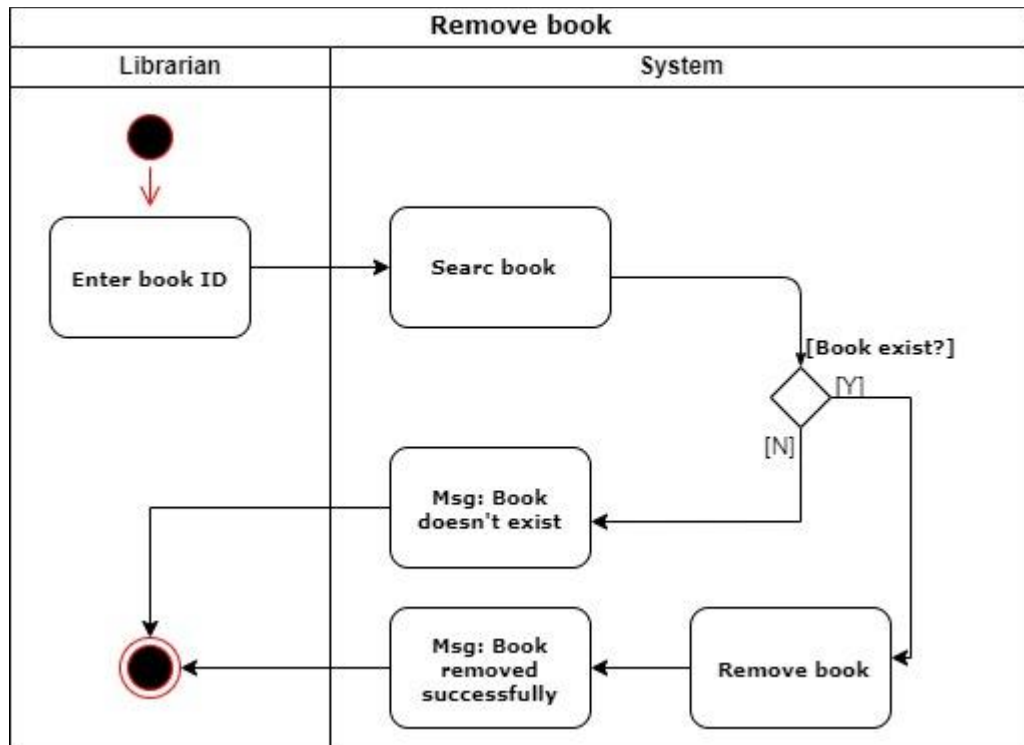


Figure 4.3: Activity diagram with swimlane for "Remove a book"

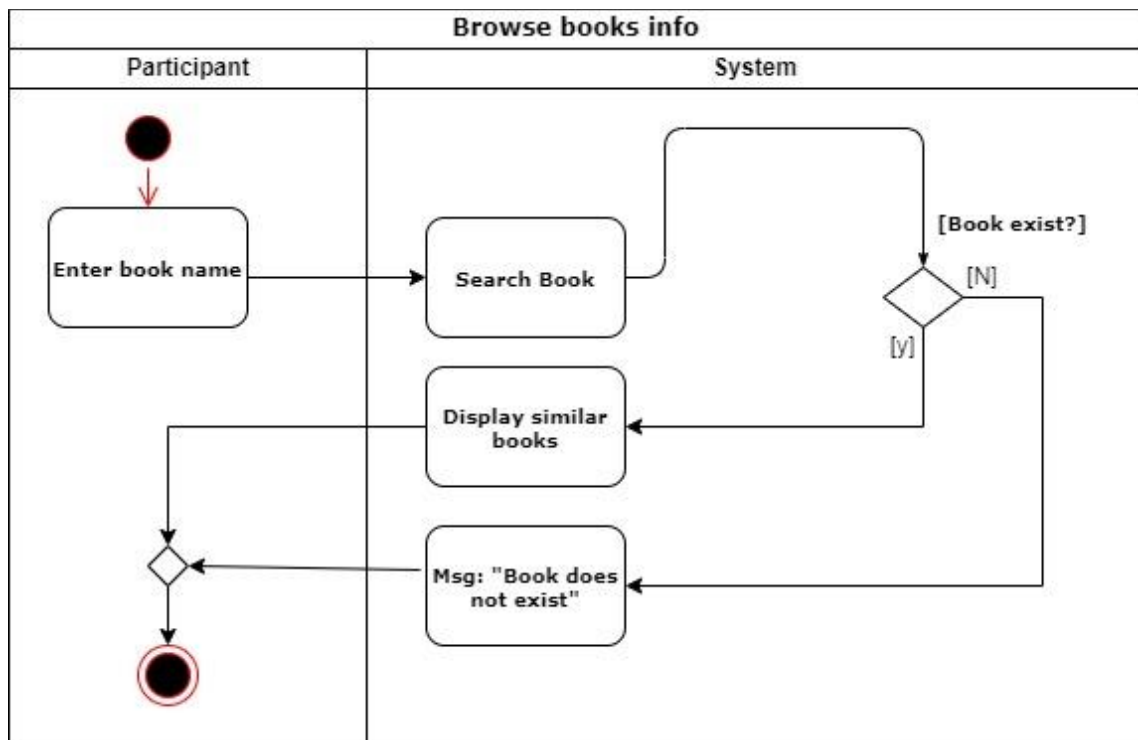


Figure 4.4: Activity diagram with swimlane for "Browse books info"

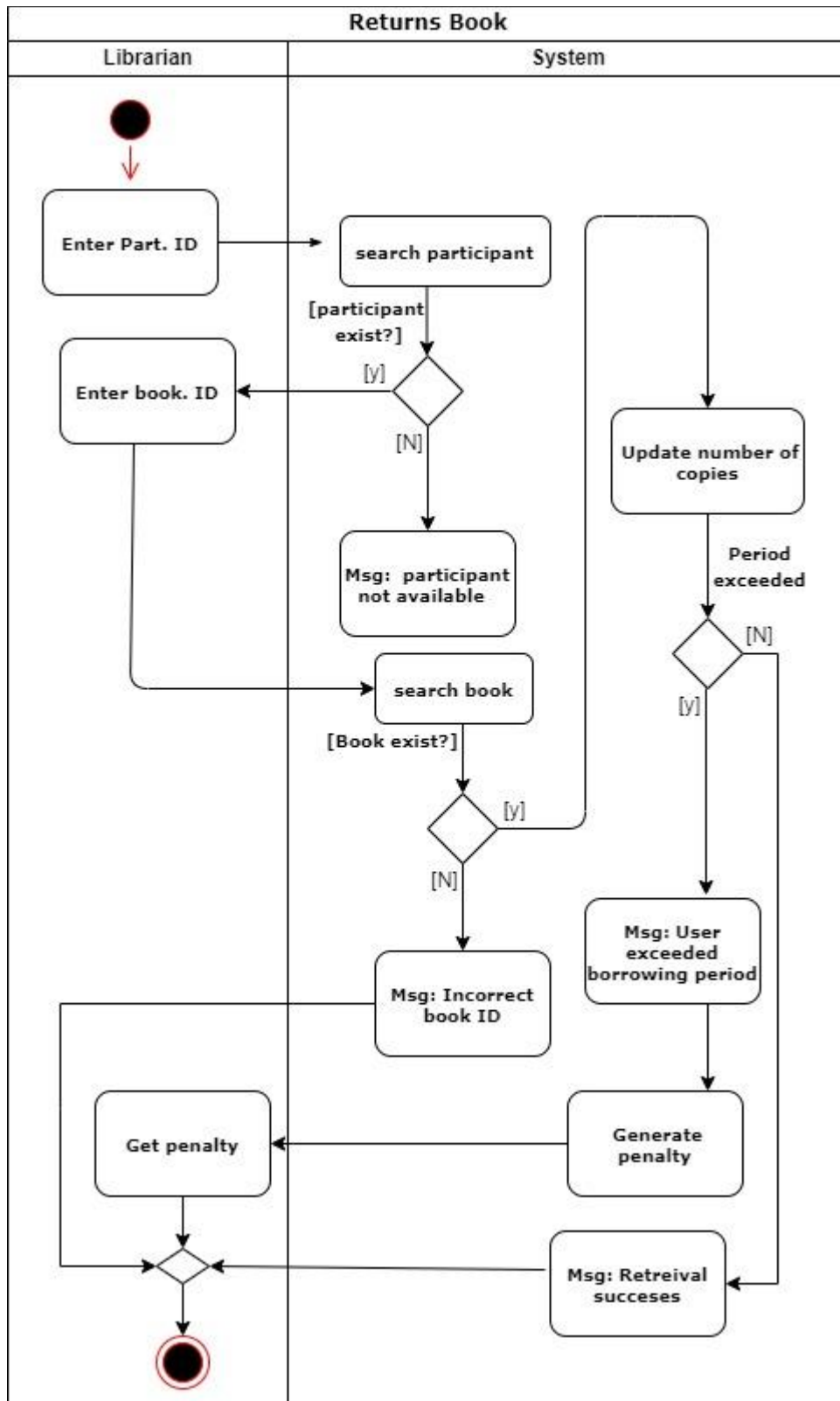


Figure 4.5: Activity diagram with swimlane for "Return a Book"



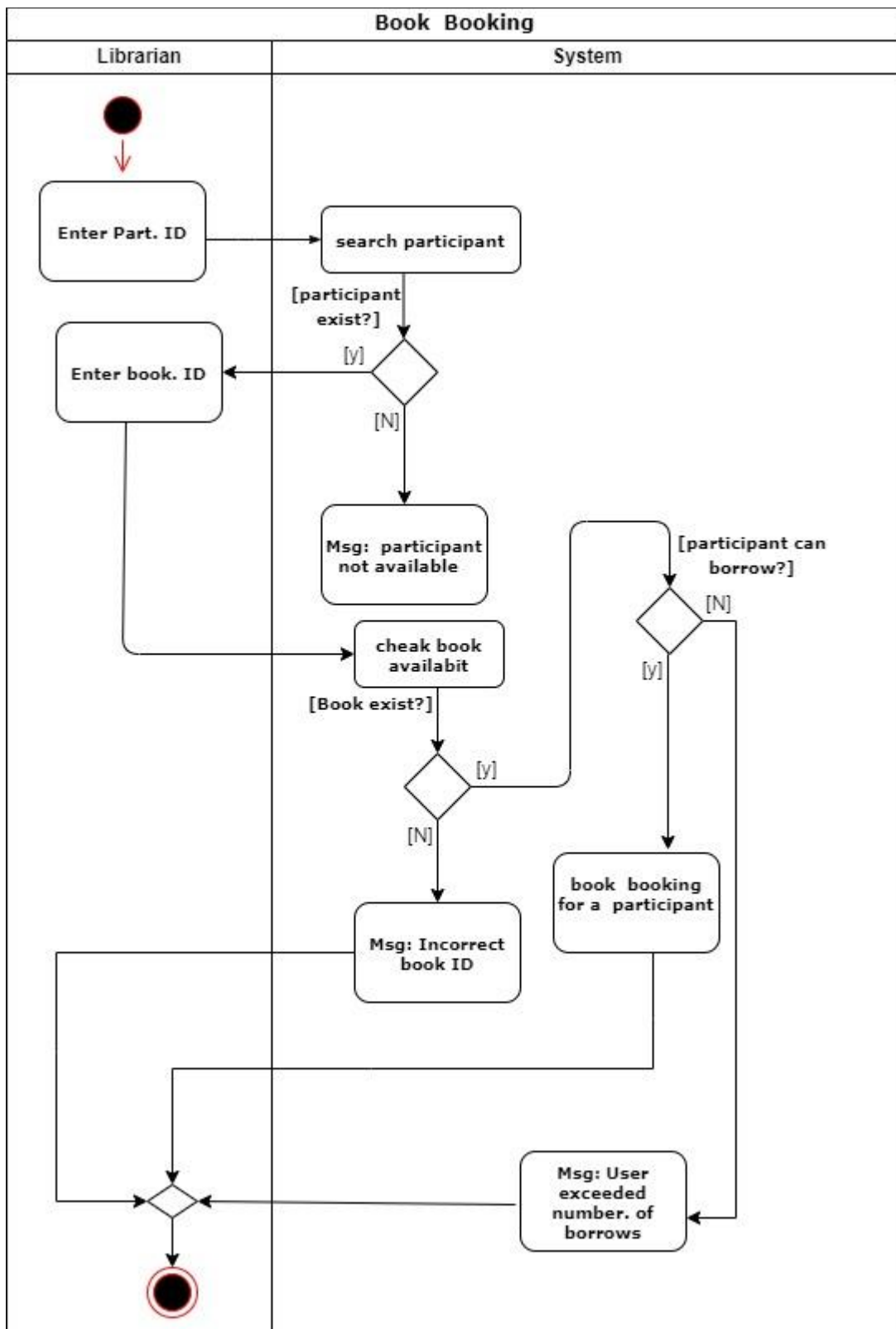


Figure 4.6: Activity diagram with swimlane for " Booking a Book "

- **Consistency**

The process of tracking the consistency between the requirements is depicted in Table 4.2 below.

Table 4.2: . Consistency traceability table

	Requirements							Common activity, condition or consistency rules
	Add a new book	Book a book	Browse books info	Retrieve a book	Remove a book	Borrow a Book		
Requirements	Add a new book							
	Book a book						X	Check number of borrowed books
	Browse books info							
	Retrieve a book							
	Remove a book							
	Borrow a book							

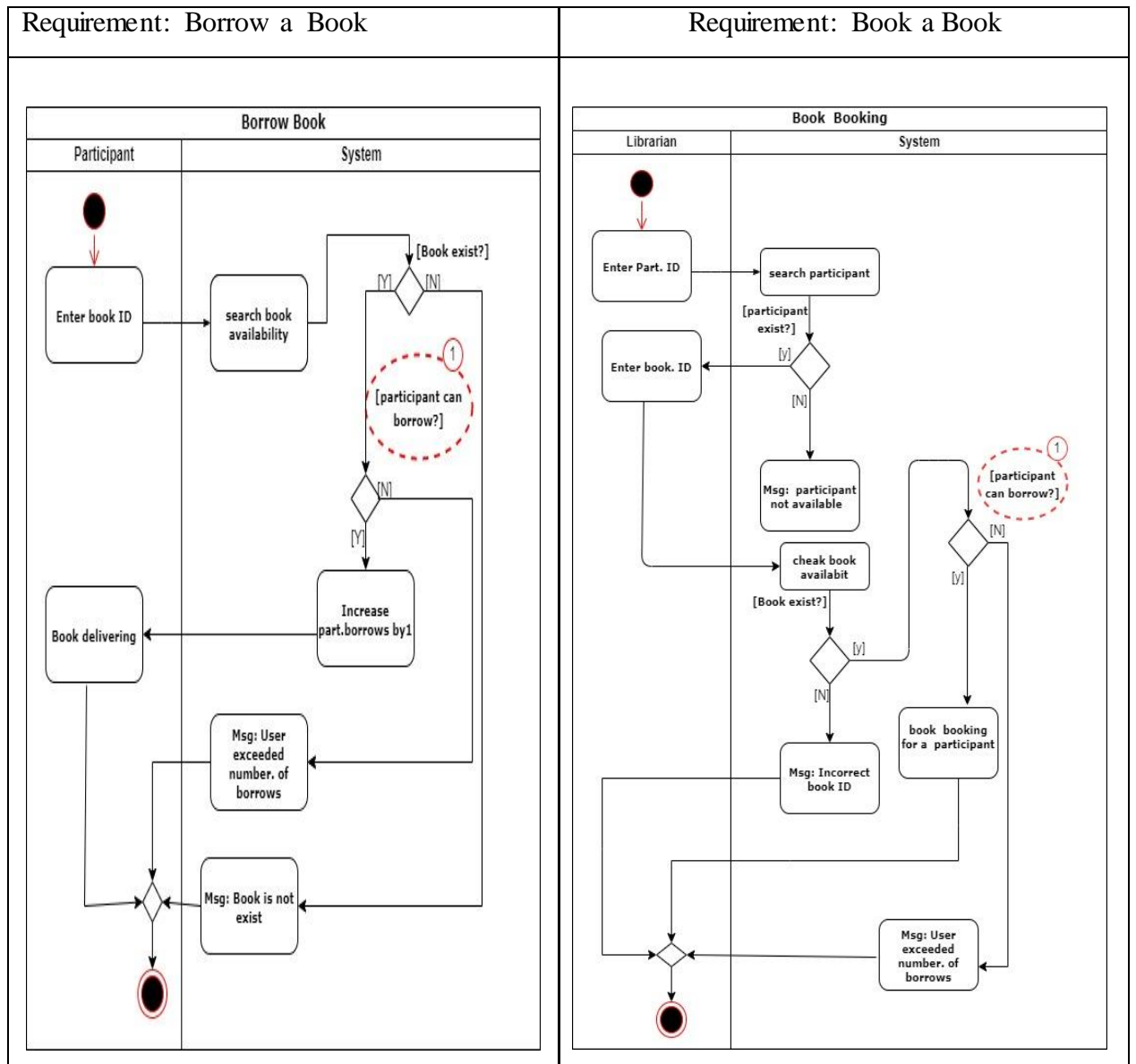


Fig 4.7: Activity diagram with swimlane to trace the Consistency between two requirements


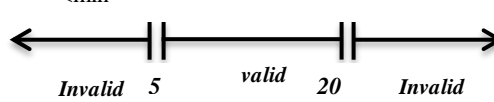
In the above figure, "Check number of borrowed books" (i.e. "Participant can borrow?") represents the rule or condition that should be considered.

### 4.1.2 Design coupling traceability

The stages of the proposed framework will be applied to tracking component coupling types in this section.

- **Data coupling**

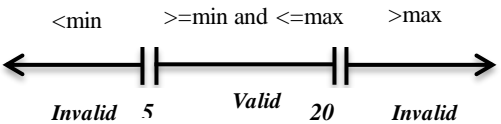
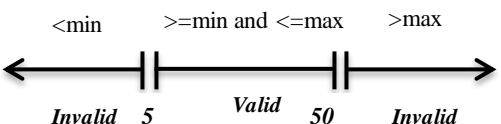
Table 4.3: An example of data coupling between two components

Testing Date/Time	Testing Time	Tester	Type of test Pass/fail																													
12/11/2021	15:00	Fatima	<input checked="" type="radio"/> Test to pass <input type="radio"/> Test to fail																													
Componenti: Search a Book		Componentj: Browse Books																														
Search a book (BookID )		Parameter : BookID  <b>Boundary Value Analysis:</b> <i>Inputs edge: length from 5 to 20 integer digits</i>  <div style="text-align: center;"> <p>&lt;min      &gt;=min and &lt;=max      &gt;max</p>  </div> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt;min</td> <td>4</td> <td>Fail</td> </tr> <tr> <td>min</td> <td>5</td> <td>pass</td> </tr> <tr> <td>max</td> <td>20</td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td>21</td> <td>Fail</td> </tr> </tbody> </table> <b>Equivalence Class Partitioning:</b> <i>Valid type: integer value</i>  <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="2">valid</th> <th colspan="3">invalid</th> </tr> <tr> <th>Partitions1</th> <th>Partitions1</th> <th>Partitions2</th> <th>Partitions3</th> <th>Partitions4</th> </tr> </thead> <tbody> <tr> <td>0-9</td> <td>a-z</td> <td>A-Z</td> <td>Special Chars</td> <td>Blank field</td> </tr> </tbody> </table>	partitions	Input value	Expected result	<min	4	Fail	min	5	pass	max	20	pass	>max	21	Fail	valid		invalid			Partitions1	Partitions1	Partitions2	Partitions3	Partitions4	0-9	a-z	A-Z	Special Chars	Blank field
partitions	Input value	Expected result																														
<min	4	Fail																														
min	5	pass																														
max	20	pass																														
>max	21	Fail																														
valid		invalid																														
Partitions1	Partitions1	Partitions2	Partitions3	Partitions4																												
0-9	a-z	A-Z	Special Chars	Blank field																												
<b>Summary of failures : No failure found</b>  <b>Test Result:</b> <input checked="" type="radio"/> Pass <input type="radio"/> fail																																

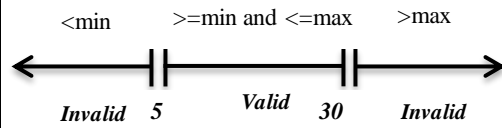
As the "Search a Book" component has passed the book number ("BookID") to the component "Browse book", where this pass shows the data association. Thus, the sent data must be verified for its integrity to avoid the negative affect on the required result. In this example, the boundary concentrated on the length of the parameter (i.e., the number of digits that the parameter "Book ID" consists of).

- Stamp coupling

Table4.4. An example of stamp coupling between two components

Testing Date	Testing Time	Tester	Type of test Pass/fail																																																												
12/11/2021	15:15	Fatima	<input checked="" type="radio"/> Test to pass <input type="radio"/> Test to fail																																																												
Component I : : Add Book( UI)		Component J: Store Book																																																													
AddNewBook ( Book)	→	<p>Data structure : Book            Parameters : IDBook ,Title , Author , Year</p> <p><b>Parameter : IDBook</b></p> <p><b>Boundary Value Analysis:</b>  <b>Inputs edge: length from 5 to 20 integer digits</b></p> <p style="text-align: center;">             &lt;min      &gt;=min and &lt;=max      &gt;max   </p> <table border="1"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt;min</td> <td>4</td> <td>Fail</td> </tr> <tr> <td>min</td> <td>5</td> <td>pass</td> </tr> <tr> <td>max</td> <td>20</td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td>21</td> <td>Fail</td> </tr> </tbody> </table> <p><b>Equivalence Class Partitioning:</b>  <b>Valid type: integer value</b></p> <table border="1"> <thead> <tr> <th>valid</th> <th colspan="4">invalid</th> </tr> <tr> <th>Partitions1</th> <th>Partitions1</th> <th>Partitions2</th> <th>Partitions3</th> <th>Partitions4</th> </tr> </thead> <tbody> <tr> <td>0-9</td> <td>a-z</td> <td>A-Z</td> <td>Special Chars</td> <td>Blank field</td> </tr> </tbody> </table> <p><b>Parameter : Title</b></p> <p><b>Boundary Value Analysis:</b>  <b>inputs edge: string value from to 50 characters</b></p> <p style="text-align: center;">             &lt;min      &gt;=min and &lt;=max      &gt;max   </p> <table border="1"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt;min</td> <td>4 Characters</td> <td>Fail</td> </tr> <tr> <td>min</td> <td>5 Characters</td> <td>pass</td> </tr> <tr> <td>max</td> <td>50 Characters</td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td>51 Characters</td> <td>Fail</td> </tr> </tbody> </table> <p><b>Equivalence Class partition (Valid Type: Alphabetic characters (small/capital or mixed))</b></p> <table border="1"> <thead> <tr> <th colspan="2">valid</th> <th colspan="3">Invalid</th> </tr> <tr> <th>Partitions1</th> <th>Partitions2</th> <th>Partitions1</th> <th>Partitions2</th> <th>Partitions3</th> </tr> </thead> <tbody> <tr> <td>A-Z</td> <td>a-z</td> <td>0-9</td> <td>Special Chars</td> <td>Blank field</td> </tr> </tbody> </table>		partitions	Input value	Expected result	<min	4	Fail	min	5	pass	max	20	pass	>max	21	Fail	valid	invalid				Partitions1	Partitions1	Partitions2	Partitions3	Partitions4	0-9	a-z	A-Z	Special Chars	Blank field	partitions	Input value	Expected result	<min	4 Characters	Fail	min	5 Characters	pass	max	50 Characters	pass	>max	51 Characters	Fail	valid		Invalid			Partitions1	Partitions2	Partitions1	Partitions2	Partitions3	A-Z	a-z	0-9	Special Chars	Blank field
partitions	Input value	Expected result																																																													
<min	4	Fail																																																													
min	5	pass																																																													
max	20	pass																																																													
>max	21	Fail																																																													
valid	invalid																																																														
Partitions1	Partitions1	Partitions2	Partitions3	Partitions4																																																											
0-9	a-z	A-Z	Special Chars	Blank field																																																											
partitions	Input value	Expected result																																																													
<min	4 Characters	Fail																																																													
min	5 Characters	pass																																																													
max	50 Characters	pass																																																													
>max	51 Characters	Fail																																																													
valid		Invalid																																																													
Partitions1	Partitions2	Partitions1	Partitions2	Partitions3																																																											
A-Z	a-z	0-9	Special Chars	Blank field																																																											

**Parameter : Author**  
**Boundary Value Analysis(BVA)** (inputs edge: string value from 5 to 30 characters)

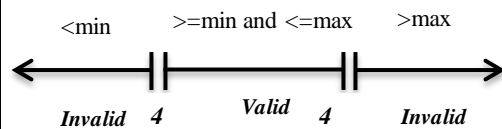


partitions	Input value	Expected result
<min	4 Characters	Fail
min	5 Characters	pass
max	30 Characters	pass
>max	31 Characters	Fail

**Equivalence Class partition (Valid Type: Alphabetic characters (small/capital or mixed))**

valid		Invalid		
Partitions1	Partitions2	Partitions1	Partitions2	Partitions3
A-Z	a-z	0-9	Special Chars	Blank field

**Parameter : Year**  
**Boundary Value Analysis:**  
**Inputs edge: length 4 integer digits**



partitions	Input value	Expected result
<min	3	Fail
min	4	pass
max	4	pass
>max	5	Fail

**Equivalence Class partition**  
**Valid type: integer value**

valid	Invalid			
Partitions1	Partitions1	Partitions2	Partitions3	Partitions4
0-9	a-z	A-Z	Special Chars	Blank field

**Summary of failures: No failure found**

**Test Result:**

- pass
- fail

The object is passed from the "Add Book" component to the "Store Book" component. Then, the "Store Book" perform the required tests to ensures that the data is correct before it is stored in the database.

- **Control coupling**

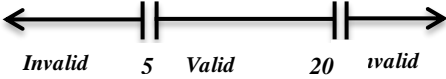
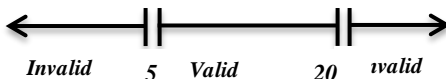
Table 4.5 :An example of control coupling between two components

Testing Date	Testing Time	Tester	Type of test Pass/fail
12/11/2021	15:30	Fatima	<input checked="" type="radio"/> Test to pass <input type="radio"/> Test to fail
Component I : Browse a Book		Component J: Search Book	
Browse a Book (book title)	→	<b>Path 1: search by book title</b>  Test Result:  <input checked="" type="radio"/> Pass <input type="radio"/> fail	
Browse a Book (bookID)		<b>Path 2: search by bookID</b>  Test Result:  <input checked="" type="radio"/> Pass <input type="radio"/> fail	
<b>Summary of failures: No failure found</b>  <b>Test Result:</b>  <input checked="" type="radio"/> pass <input type="radio"/> fail			

The "Browse Book" component controls the path of the search component, where the search criterion (for example, "Book title") is passed and the appropriate path (here, search by "BookID") is selected, despite there are many paths to search (for example, the search can be done using BookID , Book title and so on).

• Common coupling

Table 4.6. An example of common coupling between two components.

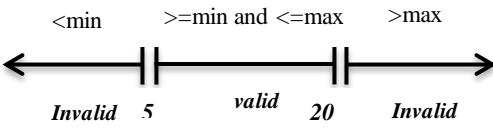
	Testing Date	Testing Time	Tester	Type of test Pass/fail																														
	12/11/2021	15:30	Fatima	● Test to pass ○ Test to fail																														
<b>Common(Shared) variable(s)</b>	<b>Component<sub>i</sub>: Browse book</b>																																	
BookID	<p>Parameter : BookID <b>Boundary Value Analysis:</b> <i>Inputs edge: length from 5 to 20 integer digits</i></p> <p style="text-align: center;">&lt;min    &gt;=min and &lt;=max    &gt;max</p>  <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt; min</td> <td>4</td> <td>Fail</td> </tr> <tr> <td>min</td> <td>5</td> <td>pass</td> </tr> <tr> <td>max</td> <td>20</td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td>21</td> <td>Fail</td> </tr> </tbody> </table> <p><b>Equivalence Class Partitioning</b> <i>Valid type: integer value</i></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th>valid</th> <th>Invalid</th> </tr> </thead> <tbody> <tr> <td>Partitions1</td> <td>0-9</td> <td>a-z</td> </tr> <tr> <td>Partitions2</td> <td></td> <td>A-Z</td> </tr> <tr> <td>Partitions3</td> <td></td> <td>Special Chars</td> </tr> <tr> <td>Partitions4</td> <td></td> <td>Blank field</td> </tr> </tbody> </table> <p>Common variable test(s): ● Pass ○ Fail</p>				partitions	Input value	Expected result	< min	4	Fail	min	5	pass	max	20	pass	>max	21	Fail		valid	Invalid	Partitions1	0-9	a-z	Partitions2		A-Z	Partitions3		Special Chars	Partitions4		Blank field
partitions	Input value	Expected result																																
< min	4	Fail																																
min	5	pass																																
max	20	pass																																
>max	21	Fail																																
	valid	Invalid																																
Partitions1	0-9	a-z																																
Partitions2		A-Z																																
Partitions3		Special Chars																																
Partitions4		Blank field																																
	<b>Component: Update</b>																																	
	<p>Parameter : BookID <b>Boundary Value Analysis:</b> <i>Inputs edge: length from 5 to 20 integer digits</i></p> <p style="text-align: center;">&lt;min    &gt;=min and &lt;=max    &gt;max</p>  <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th>partitions</th> <th>Input value</th> <th>Expected result</th> </tr> </thead> <tbody> <tr> <td>&lt; min</td> <td>4</td> <td>Fail</td> </tr> <tr> <td>min</td> <td>5</td> <td>pass</td> </tr> <tr> <td>max</td> <td>20</td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td>21</td> <td>Fail</td> </tr> </tbody> </table> <p><b>Equivalence Class Partitioning</b> <i>Valid type: integer value</i></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th></th> <th>valid</th> <th>Invalid</th> </tr> </thead> <tbody> <tr> <td>Partitions1</td> <td>0-9</td> <td>a-z</td> </tr> <tr> <td>Partitions2</td> <td></td> <td>A-Z</td> </tr> <tr> <td>Partitions3</td> <td></td> <td>Special Chars</td> </tr> <tr> <td>Partitions4</td> <td></td> <td>Blank field</td> </tr> </tbody> </table>				partitions	Input value	Expected result	< min	4	Fail	min	5	pass	max	20	pass	>max	21	Fail		valid	Invalid	Partitions1	0-9	a-z	Partitions2		A-Z	Partitions3		Special Chars	Partitions4		Blank field
partitions	Input value	Expected result																																
< min	4	Fail																																
min	5	pass																																
max	20	pass																																
>max	21	Fail																																
	valid	Invalid																																
Partitions1	0-9	a-z																																
Partitions2		A-Z																																
Partitions3		Special Chars																																
Partitions4		Blank field																																
<b>Summary of failures: No failure found</b>																																		
<p><b>Test result:</b> ● pass ○ fail</p>																																		

In this example, "BookID" parameter which is the common variable, should be controlled and kept constant in both components.



- **Content coupling**

Table 4.7 :An example of content coupling between two components

Testing Date	Testing Time	Tester	Type of test Pass/fail																														
12/11/2021	15:39	Fatima	<input checked="" type="radio"/> Test to pass <input type="radio"/> Test to fail																														
Component <sub>i</sub> : Remove a Book		Component <sub>j</sub> : RemoveBorrow (BookID)																															
Remove a book (BookID )	→	Parameter : BookID <i>Valid type: integer value</i>  <b>Boundary Value Analysis:</b> <i>Inputs edge: length from 5 to 20 integer digits</i>  <div style="text-align: center;"> <math>\leftarrow</math> &lt;min      &gt;=min and &lt;=max      &gt;max <math>\rightarrow</math>    <i>Invalid</i> 5      <i>valid</i> 20      <i>Invalid</i> </div> <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th><i>partitions</i></th> <th><i>Input value</i></th> <th><i>Expected result</i></th> </tr> </thead> <tbody> <tr> <td>&lt;min</td> <td>4</td> <td>Fail</td> </tr> <tr> <td>min</td> <td>5</td> <td>pass</td> </tr> <tr> <td>max</td> <td>20</td> <td>pass</td> </tr> <tr> <td>&gt;max</td> <td>21</td> <td>Fail</td> </tr> </tbody> </table> <b>Equivalence Class Partitioning:</b> <i>Valid type: integer value</i>  <table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th colspan="2"><i>valid</i></th> <th colspan="3"><i>invalid</i></th> </tr> <tr> <th><i>Partitions1</i></th> <th><i>Partitions1</i></th> <th><i>Partitions2</i></th> <th><i>Partitions3</i></th> <th><i>Partitions4</i></th> </tr> </thead> <tbody> <tr> <td>0-9</td> <td>a-z</td> <td>A-Z</td> <td>Special Chars</td> <td>Blank field</td> </tr> </tbody> </table>		<i>partitions</i>	<i>Input value</i>	<i>Expected result</i>	<min	4	Fail	min	5	pass	max	20	pass	>max	21	Fail	<i>valid</i>		<i>invalid</i>			<i>Partitions1</i>	<i>Partitions1</i>	<i>Partitions2</i>	<i>Partitions3</i>	<i>Partitions4</i>	0-9	a-z	A-Z	Special Chars	Blank field
<i>partitions</i>	<i>Input value</i>	<i>Expected result</i>																															
<min	4	Fail																															
min	5	pass																															
max	20	pass																															
>max	21	Fail																															
<i>valid</i>		<i>invalid</i>																															
<i>Partitions1</i>	<i>Partitions1</i>	<i>Partitions2</i>	<i>Partitions3</i>	<i>Partitions4</i>																													
0-9	a-z	A-Z	Special Chars	Blank field																													
<b>Summary of failures: No failure found</b>  <b>Test Result:</b>  <input checked="" type="radio"/> pass <input type="radio"/> fail																																	

For example, if you delete the book data, you will also delete the borrowing data, because the borrowing data is entirely dependent on the book data.

## **Chapter 5**

### **Result and Discussion**

This chapter will summarize the results of this research, by providing an overview of the effort and contributions made in this research, and also compared with the previous efforts. This is an attempt to reduce the potential errors in the phase of requirements building and coupling the design of software components where this work will aid developers in taking these issues into account and not ignoring them in the future.

The software testing process requires a great effort in order to obtain an accurate system that meets what is required of it, and for this reason it is regarded as one of the most essential stages of the SDLC, where researchers are constantly striving to improve.

However, the emphasis in the testing phase does not preclude comprehensive testing of the preceding stages, as the potential errors are considered among the most important of these problems that did not receive great attention at an early stage. Hence, the requirements stage contains a number of issues that impede software success, as ambiguity and inaccuracy of requirements, many efforts have been made in this area, and it still needs further study and improvement. According to available studies, there is no comprehensive framework that includes trying to identify and then solve these problems as early as possible.

This thesis investigated a number of potential faults at the early stages of SDLC (i.e., requirements issues and design coupling), with the goal of developing a general framework to track these errors according to an organized mechanism, which can later be considered a supplementary process to the testing phase and contribute to easing software developers' work in capturing these issues. In its first part, the framework focuses on contributing to the improvement of the software requirements tracking process by defining a set of tracking tables specially designed for this purpose. These tables will contribute to reducing these issues. Compared to previous efforts made in this subject, most of them focused on some issues and neglected others, and many of those efforts dealt with these issues in an abstract manner. Such efforts are expressed in (Yang et al., 2019 ; Riaz et al., 2019 ; Kamalrudin & Sidek, 2015; Gigante, Gargiulo, &

Ficco, 2015; Mayr-Dorn et al., 2021 ; Sulaiman et al., 2019;Hadar et al., 2019; Patel & Gandhi, 2014).

In addition, potential bugs inherited from the requirements stage may hinder successful software design. Furthermore, due to a lack of focus on design coupling tracking, the design stage itself may contain potential problems. As is well known, software component dependence on one another (i.e., coupling) is undesirable, especially if some mistakes occur in one of those components and subsequently spread to other software components. Efforts in this area are still limited and that additional research and study are needed (Razafimahatratra et al., 2017; Shweta Sharma & Srinivasan, 2013). Furthermore, no comprehensive mechanism has been identified to address these dilemmas, as compared to what has been proposed in this thesis, some of these studies are shown in (Bavota, Dit, Oliveto, Di Penta, et al., 2013;Geetika & Singh, 2014; Alenezi & Magel, 2017; Anwer et al., 2017; Kumar & Chauhan, 2015) which focused only on coupling detection in the implementation phase, as these efforts did not limit the spread of potential faults that are addressed in this thesis.

Based on the foregoing, the contributions of the thesis can be summarized as follows:

- Contribute to introduce a mechanism for detecting a number of potential faults at early stage of the SDLC (according to what was inquired in Research question No.1). Where the focus was on studying what the excellent requirements require in terms of conditions ,as well as an in-depth study of previous studies, especially with regard to those errors that are usually ignored or not focused on deeply in the process of validating requirements and designing coupling. In order to clarify what was mentioned in Research question No. 2, the most important proposed ways to solve the problem of potential errors were addressed in parts of it in an abstract or custom manner, such as what was done in the study (Sulaiman et al., 2019), which commented on finding the contradiction (ambiguity) between the activity diagram and the class diagram. Compared to the study in this thesis, the suggested framework will help to facilitate the process of tracking these dilemmas. Where this study adopts a hierarchical mechanism that begins by making sure of the accuracy of the requirements, in terms of their necessity first. Table 3.1 was proposed to track the requirements and the sources belonging to them. Where this study

aims that the requirements are not created or captured except according to the existence of their sources. Then, follows these requirements in order to ensure their validity, accuracy, completeness and unambiguity. Also, an activity diagram with swimlanes is proposed to illustrate the process of completing each requirement in order to indicate that these requirements have been accurately understood (Figure 3.2). In addition, the process of requirements consistency is a difficult process, and therefore adequate clarification of it is important, so that any dilemmas leading to inconsistency are captured. From here, and to complement the role played by the previous figure (Figure 3.2) in the process of clarifying the requirements. Failing may occur in part or (parts) common in the requirements that must be considered in all of them (table 3.2). Also, an activity diagram with swimlanes was proposed as a parallel process to clarify the common requirements in a specific part or parts. This means that the conditions or activities that should be inclined in both should be tracked, thus not missed into one of them.

Where most of the studies dealt with one aspect of the problems, for example, consistency or dealing with them in an abstract way is difficult for developers to focus on these dilemmas (Kamaludin & Sidek, 2015).

- In addition to the foregoing, the emphasis in the design phase has been on tracking the design coupling of software components and trying to capture the interaction between software components. A number of different of coupling types have been studied, and tracking tables for these types of coupling have been proposed to supplement the answer to research question No.2. These tables attempted to track the interaction of software components, with a focus on testing that interaction and capturing errors that may arise. Good tracking of that interaction will help to ensure that errors do not spread among those components when an error arise in any of the interacting components. Among the testing mechanisms that have been adopted are Boundary value analysis and Equivalence class partitioning, in addition to the tests complementary to that process.
- As an organizational process for unifying the framework's contents (Figure 3.1), and as an explanation for the question contained in Research question NO.3,

which includes a query about how to detect potential error problems. This thesis introduced an integrated general framework that helps to reduce errors in the requirements and design stages. It also highlights the significance of addressing dilemmas with a piercing eye that tracks the emergence of these dilemmas and alerts them.

- With regard to the parties that will benefit from the proposed framework (Research question No.4), the proposed framework seeks to be a complementary mechanism for the software testing phase in the SDLC, with the aim of improving it. Where stakeholders (requesters of the required software) will benefit. This is because their software will perform what is required of it without potential errors that appear later and hinder the continued success of their software. In addition, software development teams themselves will be encouraged to consider potential errors, due to a simplified framework that will help them easily track the quality of the software they are developing, in terms of correctness and accuracy.
- As for ascertaining the possibility of applying the proposed framework (Research question No.5), which was discussed by a number of specialists. Then building a case study in which all the items of the proposed framework were applied (i.e., starting with tracking requirements and ending with the process of tracing the coupling between software components). Where the differences between the dilemmas involved in the requirements problems and the design of the coupling have been clarified with examples for each of the cases included in the framework.

Finally, as a general recommendation, ignoring requirements issues and not tightly controlling the coupling of software components will significantly decrease the reliability and scalability of the software later on, as well as cause latent faults that are expensive to maintain. As a result. Addressing these issues early at SDLC will reduce the cost of reworking or maintaining software systems later on.

## **Conclusion and Future work**

Requirements issues and excessive software coupling are among the most important causes of potential faults that disrupt building successful software. An organized framework has been proposed in this thesis that contributes to tracking requirements in terms of complete, correct, consistent, and necessary in order to reduce the resulting errors that may be passed to later stages. Some tracking tables and graphical diagrams have been suggested to make it easier to track these problems. In addition, as high coupling between components is considered undesirable, especially excessive interaction, it is mainly considered as a significant contributor to the propagation of errors between software components, which if not taken into account will lead to week software development. From this point of view, the proposed framework included an organized mechanism for tracking coupling design, where special tables are prepared to track some of the coupling types such as: content coupling, common coupling, control coupling, stamp coupling, and data coupling. From here, it can be concluded, that the proposed framework will support developers in improving their software. Finally, evaluating the results of this framework was carried out based on the preparation of a simplified case study to clarify the mechanism of its work, which we hope would have accomplished the desired result.

As a future work, several case studies can be applied on this framework, and the recommendations received through the application can be taken into account; as a contribution to its improvement later. In addition, Applying Natural Language Processing (NLP) concepts can be considered as another trend through which the framework can be improved later by analyzing requirements scenarios and trying to help capture their issues early.

## References

- Acharya, S., Mohanty, H., & George, C. (2005). Domain consistency in requirements specification. *Proceedings - International Conference on Quality Software*, 2005(9), 231–238. <https://doi.org/10.1109/QSIC.2005.24>
- Akinsola, J. E. T., Ogunbanwo, A. S., Okesola, O. J., Odun-Ayo, I. J., Ayegbusi, F. D., & Adebisi, A. A. (2020). Comparative Analysis of Software Development Life Cycle Models (SDLC). *Computer Science On-Line Conference*, 310–322. [https://doi.org/10.1007/978-3-030-51965-0\\_27](https://doi.org/10.1007/978-3-030-51965-0_27)
- Alenezi, M., & Magel, K. (2017). Empirical evaluation of a new coupling metric: Combining structural and semantic coupling. *International Journal of Computers and Applications*, 36(1), 34–44. <https://doi.org/10.2316/Journal.202.2014.1.202-3902>
- Anas, H., Ilyas, M., Tariq, Q., & Hummayun, M. (2016). Requirements Validation Techniques: An Empirical Study. *International Journal of Computer Applications*, 148(14), 5–10. <https://doi.org/10.5120/ijca2016910911>
- Anwer, S., Adbellatif, A., Alshayeb, M., & Anjum, M. S. (2017). Effect of coupling on software faults: An empirical study. *Proceedings of 2017 International Conference on Communication, Computing and Digital Systems, C-CODE 2017*, October 2018, 211–215. <https://doi.org/10.1109/C-CODE.2017.7918930>
- Bavota, G., Dit, B., Oliveto, R., Di Penta, M., Poshyvanyk, D., & De Lucia, A. (2013). An empirical study on the developers' perception of software coupling. *Proceedings - International Conference on Software Engineering*, 692–701. <https://doi.org/10.1109/ICSE.2013.6606615>
- Bavota, G., Dit, B., Oliveto, R., Penta, M. Di, Poshyvanyk, D., & Lucia, A. De. (2013). An Empirical Study on the Developers' Perception of Software Coupling. In *2013 35th International Conference on Software Engineering (ICSE)*, 692–701.
- Bose, R. P. J. C., & Srinivasan, S. H. (2005). Data Mining Approaches to Software Fault Diagnosis. In *15th International Workshop on Research Issues in Data Engineering: Stream Data Mining and Applications (RIDE-SDMA'05)*, 45–52. <https://doi.org/10.1109/RIDE.2005.9>

- Carson, R. S., Aslaksen, E., Caple, G., Davies, P., Gonzales, R., Kohl, R., & Sahraoui, A.-E.-K. (2004). 5.1.3 Requirements Completeness. *INCOSE International Symposium*, 14(1), 930–944. <https://doi.org/10.1002/j.2334-5837.2004.tb00546.x>
- Causevic, A., Sundmark, D., & Punnekkat, S. (2010). An Industrial Survey on Contemporary Aspects of Software Testing. In *2010 Third International Conference on Software Testing, Verification and Validation*, 393–401. <https://doi.org/10.1109/ICST.2010.52>
- Cunningham, S., Gambo, J., Lawless, A., Moore, D., & Yilmaz, M. (2019). Software Testing: A Changing Career. In *European Conference on Software Process Improvement*, 731–742.
- Darwish, N. R. (2016). Requirements Engineering in Scrum Framework Requirements Engineering in Scrum Framework. September. <https://doi.org/10.5120/ijca2016911530>
- Dashti, M. T., & Basin, D. (2020). A Theory of Black-Box Tests. *ArXiv Preprint ArXiv: 2006*, 1–30.
- Dhanalaxmi, B., Naidu, G. A., & Anuradha, K. (2015). A Review on Software Fault Detection and Prevention Mechanism in Software Development Activities. *17(6)*, 25–30. <https://doi.org/10.9790/0661-17652530>
- Eichinger, F., Klemens, B., & Huber, M. (2008). Improved Software Fault Detection with Graph Mining. *Proceedings of the 6th International Workshop on Mining and Learning with Graphs (MLG) at ICML*, c, 1–3. <https://doi.org/10.5445/IR/1000008547>
- Freeman, H. (2002). Software Testing. *IEEE Instrumentation & Measurement Magazine*, September, 48–50.
- Fregnan, E., Baum, T., Palomba, F., & Bacchelli, A. (2019). A survey on software coupling relations and tools. *Information and Software Technology*, 107(November 2018), 159–178. <https://doi.org/10.1016/j.infsof.2018.11.008>
- Geetika, R., & Singh, P. (2014). Empirical investigation into static and dynamic coupling metrics. *ACM SIGSOFT Software Engineering Notes*, 39(1), 1–8. <https://doi.org/10.1145/2557833.2557847>



- Gigante, G., Gargiulo, F., & Ficco, M. (2015). A semantic driven approach for requirements verification. *Studies in Computational Intelligence*, 570, 427–436. [https://doi.org/10.1007/978-3-319-10422-5\\_44](https://doi.org/10.1007/978-3-319-10422-5_44)
- Gigante, G., Gargiulo, F., Ficco, M., & Pascarella, D. (2015). A semantic driven approach for consistency verification between requirements and FMEA. *Studies in Computational Intelligence*, 616, 403–413. [https://doi.org/10.1007/978-3-319-25017-5\\_38](https://doi.org/10.1007/978-3-319-25017-5_38)
- Graham, D. (2002). Requirements and Testing: *IEEE Software*, 19(5), 15–17.
- Guo, W., Zhang, L., & Lian, X. (2021). Automatically detecting the conflicts between software requirements based on finer semantic analysis. *Information and Software Technology*, 1–18.
- Hadar, I., Zamansky, A., & Berry, D. M. (2019). The inconsistency between theory and practice in managing inconsistency in requirements engineering. *Empirical Software Engineering*, 24(6), 3972–4005. <https://doi.org/10.1007/s10664-019-09718-5>
- Hagal, M. A., & H.Fazzani, F. (2013). A Use Case Map as a Visual Approach to Reduce the Degree of Inconsistency. *International Conference on Computer Systems and Industrial*, 0–3. <https://doi.org/10.1109/iccsii.2012.6454384>
- Hedao, A. H., & Khandelwal, A. (2017). Study of Dynamic Testing Techniques. *International Journal of Advanced Research in Computer Science and Software Engineering*, 7(4), 322–330. <https://doi.org/10.23956/ijarcsse/v7i4/0136>
- Kamalrudin, M., & Sidek, S. (2015). A review on software requirements validation and consistency management. *International Journal of Software Engineering and Its Applications*, 9(10), 39–58. <https://doi.org/10.14257/ijseia.2015.9.10.05>
- Kamble, S., Jin, X., Niu, N., & Simon, M. (2017). A Novel Coupling Pattern in Computational Science and Engineering Software. *Proceedings - 2017 IEEE/ACM 12th International Workshop on Software Engineering for Science, SE4Science 2017*, 9–12. <https://doi.org/10.1109/SE4Science.2017.10>

- Kaur, M., & Singh, R. (2014). A Review of Software Testing Techniques. *International Journal of Electronic and Electrical Engineering*, 7(5), 463–474.
- Kumar, H., & Chauhan, N. (2015). A Coupling Effect Based Test Case prioritization technique. 2015 2nd International Conference on Computing for Sustainable Global Development(INDIACom), 1341–1345.
- Lawrence, B., Wieggers, K., & Ebert, C. (2001). The Top Risks of Requirements Engineering. *IEEE Software*, 18(62–63).
- Leau, Y., Loo, W. K., Tham, W. Y. T., & Fun, S. (2012). Software Development Life Cycle AGILE vs Traditional Approaches. 2012 International Conference on Information and Network Technology (ICINT 2012), 37(Icint), 162–167.
- Maia, T., & Souza, M. (2018). A Practical Methodology for DO-178C Data and Control Coupling Objective Compliance. 236–240.
- Marques, J., & Yelisetty, S. (2019). An Analysis of Software Requirements Specification Characteristics In Regulated Environments. *International Journal of Software Engineering & Applications*, 10(6), 1–15. <https://doi.org/10.5121/ijsea.2019.10601>
- Mayr-Dorn, C., Kretschmer, R., Egyed, A., Heradio, R., & Fernandez-Amoros, D. (2021). Inconsistency-tolerating guidance for software engineering processes.
- Miholca, D. L., & Onet-Marian, Z. (2020). An analysis of aggregated coupling's suitability for software defect prediction. *Proceedings - 2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2020*, 141–148. <https://doi.org/10.1109/SYNASC51798.2020.00032>
- Narizzano, M., Pulina, L., Tacchella, A., & Vuotto, S. (2019). Property specification patterns at work: verification and inconsistency explanation. *Innovations in Systems and Software Engineering*, 15(3–4), 307–323. <https://doi.org/10.1007/s11334-019-00339-1>

- Nidamanuri, S. R. (2021). Requirements Validation Techniques and Factors Influencing them Santosh Kumar Reddy Peddireddy (Issue February).
- Nidhra, S., & Dondeti, J. (2012). BLACK BOX AND WHITE BOX TESTING TECHNIQUES – A LITERATURE REVIEW. *International Journal of Embedded Systems and Applications (IJESA)* Vol.2, No.2, June 2012 BLACK, 2(2), 29–50. <https://doi.org/10.5121/ijesa.2012.2204>
- Pandey, S. K., & Batra, M. (2013). Security Testing in Requirements Phase of SDLC. *International Journal of Computer Applications*, 68(9), 31–35. <https://doi.org/10.5120/11609-6985>
- Patel, K., & Gandhi, P. S. (2014). Inconsistency Measurement and Remove from Software Requirement Specification. IJEDR1402214 *International Journal of Engineering Development and Research (Www.Ijedr.Org)*, 2(2), 2655–2659.
- Poshyvanyk, D., Marcus, A., Ferenc, R., Gyimóthy, T., & Published. (2009). Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1), 5–32. <https://doi.org/10.1007/s10664-008-9088-2>
- Razafimahatratra, H., Mahatody, T., Razafimandimby, J. P., & Simionescu, S. M. (2017). Automatic detection of coupling type in the UML sequence diagram. 2017 21st International Conference on System Theory, Control and Computing, ICSTCC 2017, 635–640. <https://doi.org/10.1109/ICSTCC.2017.8107107>
- Riaz, M. Q., Butt, W. H., & Rehman, S. (2019). Automatic Detection of Ambiguous Software Requirements: An Insight. 5th International Conference on Information Management, ICIM 2019, March, 1–6. <https://doi.org/10.1109/INFOMAN.2019.8714682>
- Saavedra, R., Ballejos, L., & Ale, M. (2013). Software Requirements Quality Evaluation: State of the art and research challenges. 1850–2792.
- Sawant, A. A., Bari, P. H., & Chawan, P. M. (2012). Software Testing Techniques and Strategies. *International Journal of Engineering Research and Applications (IJERA)*, 2(3), 980–986.

- Schach, S. R. (2011). Object-oriented and classical software engineering. In 8 (Ed.), *Ruptures in the Everyday: Views of Modern Germany from the Ground* (8th ed.). <https://doi.org/10.5749/j.ctvtv93bw.16>
- Sharma, Shilpa, Raja, L., & Bhatt, D. P. (2020). Role of ontology in software testing. *Journal of Information and Optimization Sciences*, 41(2), 641–649. <https://doi.org/10.1080/02522667.2020.1733196>
- Sharma, Shweta, & Srinivasan, S. (2013). A review of Coupling and Cohesion metrics in Object Oriented Environment. *International Journal of Computer Science & Engineering Technology (IJCSET)*, 4(8), 1105–1111.
- Sommerville, I. (2011). Software Engineering. In *Clinical Engineering: A Handbook for Clinical and Biomedical Engineers* (9th ed.). <https://doi.org/10.1016/B978-0-12-396961-3.00009-3>
- Stachtari, E., Mavridou, A., Katsaros, P., Bliudze, S., & Sifakis, J. (2018). Early validation of system requirements and design through correctness-by-construction. *Journal of Systems and Software*, 145(September 2017), 52–78. <https://doi.org/10.1016/j.jss.2018.07.053>
- Sulaiman, N., Ahmad, S. S. S., & Ahmad, S. (2019). Logical approach: Consistency rules between activity diagram and class diagram. *International Journal on Advanced Science, Engineering and Information Technology*, 9(2), 552–559. <https://doi.org/10.18517/ijaseit.9.1.7581>
- Tsai, B., Stobart, S., Parrington, N., & Thompson, B. (1997). Iterative Design and Testing within the Software Development Life Cycle. 6(December).
- Tuteja, M., & Dubey, G. (2012). A Research Study on importance of Testing and Quality Assurance in Software Development Life Cycle ( SDLC ) Models. *International Journal of Soft Computing and Engineering (IJSCE)*, 2(3), 251–257.
- Umar, M. A. (2020). A Study of Software Testing: Categories , Levels , Techniques , and Types. 1–10.

- Vanmali, M., Last, M., & Kande, A. (2002). Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17(1), 45–62. <https://doi.org/10.1002/int.1002>
- Wettel, R., & Lanza, M. (2008). Visually Localizing Design Problems with Disharmony Maps. In *Proceedings of the 4th ACM Symposium on Software Visualization*, 155–164.
- Whittaker, J. A. (2000). What is software testing? And why is it so hard? *IEEE Software*, 17(1), 70–79. <https://doi.org/10.1109/52.819971>
- Wong, W. E., Gao, R., Li, Y., Abreu, R., Wotawa, F., Pan, H., Gregory, W. B., Liblit, B. R., Peichl, B., He, H., Renieris, E., Riaz, N., Abreu, R., & Wang, X. (2016). Transactions on Software Engineering A Survey on Software Fault Localization Transactions on Software Engineering. 5589(November 2014), 1–41. <https://doi.org/10.1109/TSE.2016.2521368>
- Yang, Y., Ke, W., & Li, X. (2019). RM2PT: Requirements validation through automatic prototyping. *Proceedings of the IEEE International Conference on Requirements Engineering*, 2019-Septe, 484–485. <https://doi.org/10.1109/RE.2019.00067>
- Yin, R., & Ding, X. M. (2012). How to improve the quality of software testing. 2012 *International Conference on Systems and Informatics (ICSAI 2012)*, Icsai, 2533–2536.
- Yusuf, A., & Hammad, M. (2020). An Automatic Approach to Measure and Visualize Coupling in Object-Oriented Programs. 2020 *International Conference on Innovation and Intelligence for Informatics, Computing and Technologies*, 3ICT 2020. <https://doi.org/10.1109/3ICT51146.2020.9311962>
- Yusupbekov, N. R., Gulyamov, S. M., Usmanova, N. B., & Mirzaev, D. A. (2017). Challenging the ways to determine the faults in software: Technique based on associative interconnections. *Procedia Computer Science*, 120, 641–648. <https://doi.org/10.1016/j.procs.2017.11.290>
- Zheng, J., Member, S., Williams, L., Nagappan, N., & Snipes, W. (2006). On the Value of Static Analysis for Fault Detection in Software. *IEEE Transactions on Software Engineering*, 32(4), 240–253. <https://doi.org/10.1109/TSE.2006.38>

Zowghi, D., & Gervasi, V. (2003). On the interplay between consistency, completeness, and correctness in requirements evolution. In *Information and Software Technology* (Vol. 45, Issue 14). [https://doi.org/10.1016/S0950-5849\(03\)00100-9](https://doi.org/10.1016/S0950-5849(03)00100-9)

Merriam-Webster. (n.d.). Necessary. In *Merriam-Webster.com dictionary*. Retrieved May 4, 2021, from <https://www.merriam-webster.com/dictionary/necessary>

Merriam-Webster. (n.d.). Complete. In *Merriam-Webster.com dictionary*. Retrieved May 4, 2021, from <https://www.merriam-webster.com/dictionary/complete>

# إطار عمل لتحسين عملية اكتشاف الأخطاء الكامنة في المراحل المبكرة من دورة حياة تطوير البرمجيات

إعداد

فاطمة فرج مصباح سعيد

المشرف

د. محمد حبل

الملخص

تعتمد عملية بناء برمجيات ذات جودة عالية على مدى تلبيتها لما هو مطلوب منها بصورة كاملة وصحيحة. ومن هنا تعتبر عمليتا: تدقيق المتطلبات ومرحلة الاختبار هما العمليتان المسؤولتان بشكل رئيسي على التأكد من أداء البرمجية وبصورة دقيقة. العديد من الجهود بذلت من أجل إعداد أساليب وتقنيات لتسهيل عملية الاختبار وضمان جودتها. لكننا من جانب آخر لاحظنا قصوراً في التركيز على اختبار الحالات التي قد تؤدي الى ظهور أخطاء كامنة، ومن أمثلتها مشاكل المتطلبات وتصميم الاقتران.

ولهذا؛ عكفت هذه الدراسة على تقديم إطار شامل يسهل على مطوري البرمجيات التركيز على الأخطاء الكامنة بأسلوب توثيقي منظم في المراحل المبكرة من دورة حياة البرمجيات، ويكون داعماً ومكملاً لمراحل التدقيق والاختبار المختلفة. حيث تكون عمليتا: تدقيق المتطلبات، و اختبار تصميم الاقتران، هما بؤرة التركيز.

قدمت خلال هذا الرسالة حالة دراسية لتوضيح آلية عمل الاطار المقترح, حيث يبين الاطار آلية واضحة للتركيز على الاخطاء الكامنة من خلال تتبع المتطلبات في مرحلة هندسة المتطلبات واختبار التفاعل بين المكونات البرمجية (التصميم الاقتران).

**الكلمات المفتاحية :** هندسة المتطلبات, هندسة البرمجيات, تصميم الاقتران, دورة حياة تطوير البرمجيات.





# إطار عمل لتحسين عملية اكتشاف الأخطاء الكامنة في المراحل المبكرة من دورة حياة تطوير البرمجيات

قدمت من قبل:

فاطمة فرج مصباح سعيد

تحت إشراف

د.محمد حجل

قدمت هذه الرسالة استكمالاً لمتطلبات الحصول على درجة الماجستير في هندسة

البرمجيات

جامعة بنغازي

كلية تقنية المعلومات

فبراير 2022