



Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

Recommender Systems for Manual Testing

Breno Alexandro Ferreira de Miranda

Dissertação de Mestrado

Recife
22 de agosto de 2011

Universidade Federal de Pernambuco
Centro de Informática

Breno Alexandro Ferreira de Miranda

Recommender Systems for Manual Testing

*Trabalho apresentado ao Programa de Pós-graduação em
Ciência da Computação do Centro de Informática da Uni-
versidade Federal de Pernambuco como requisito parcial
para obtenção do grau de Mestre em Ciência da Com-
putação.*

Orientador: *Prof. Dr. Juliano Manabu Iyoda*
Co-orientador: *Prof. Dr. Eduardo Henrique da Silva Aranha*

Recife
22 de agosto de 2011

Catálogo na fonte
Biblioteca Jane Souto Maior, CRB4-571

Miranda, Breno Alexandre Ferreira de
Recommender systems for manual testing / Breno
Alexandre Ferreira de Miranda - Recife: O Autor, 2011.
xiii, 78 folhas : il., fig., tab.

Orientador: Juliano Manabu Iyoda.
Dissertação (mestrado) - Universidade Federal de
Pernambuco. CIn, Ciência da Computação, 2011.

Inclui bibliografia.

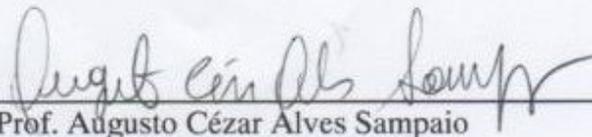
1. Engenharia de software. 2. Teste de software. 3. Sistemas
de recomendação. 4. Alocação de testes. I. Iyoda, Juliano
Manabu (orientador) II. Título.

005.1

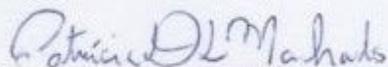
CDD (22. ed.)

MEI2011 – 119

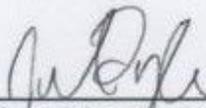
Dissertação de Mestrado apresentada por **Breno Alexandre Ferreira de Miranda** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Recommender Systems for Manual Testing**" orientada pelo **Prof. Juliano Manabu Iyoda** e aprovada pela Banca Examinadora formada pelos professores:



Prof. Augusto César Alves Sampaio
Centro de Informática / UFPE

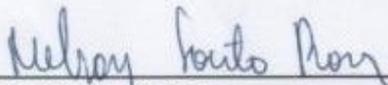


Profa. Patrícia Duarte de Lima Machado
Departamento de Sistemas e Computação / UFCG



Prof. Juliano Manabu Iyoda
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 22 de agosto de 2011.



Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

To my family.

Acknowledgements

First and foremost I would like to thank God for giving me the power to believe in myself and pursue my dreams.

I do sincerely want to express my gratitude to my advisor Juliano Iyoda. He was always available to help me and his ideas, insights, and inspiring advice smoothly guided me towards the accomplishment of this work. Thank you Juliano, I could not have done it without your support. I also want to thank my co-advisor Eduardo Aranha who, even being geographically distant, was always present to help me, especially during the experimental phase of this dissertation.

I want to thank my wife, Natália, for all the support, patience, and love. Her constant encouragements were fundamental to make this work a reality.

I want to express my gratitude to my parents, Bernardo and Margareth. I can barely find the words to express all the wisdom, love and support they have given to me and my siblings. I also want to thank my brothers, Bruno and Braulio, and my 7-year-old sister Bárbara (who asked me a million of times — *“when are you going to finish this work?”*) for the friendship and for understanding my absence during the realization of this work.

I want to thank the CIn-Motorola managers Rogério Monteiro and Luís Cláudio for their flexibility and comprehension during the times I had to be absent from my job to dedicate myself to the Masters. An especial thank you goes to Rogério Monteiro for his advice during the conception of this work’s ideas; and for all the reviews and suggestions that contributed to the continuous improvement of this dissertation.

Many thanks to all my coworkers from the CIn-Motorola project, especially the ones who actively participated in this work by providing information, given suggestions, or participating in the controlled experiment: João Paulo, Rodrigo Cursino, Eduardo Tavares, Franklin Mendes, Ana Ferraz, Luciano Mendes, Caroline Coelho, Isadora Mendonça, and José Fontes.

I also want to thank Jones Albuquerque for being one of the persons who most influenced my decision to pursue a master’s degree.

Finally, I would like to thank my family and everybody else who either directly or indirectly made this work possible.

*“The important thing is not to stop questioning.
Curiosity has its own reason for existing.”*

—ALBERT EINSTEIN

Resumo

A atividade de teste de software pode ser bastante árdua e custosa. No contexto de testes manuais, todo o esforço com o objetivo de reduzir o tempo de execução dos testes e aumentar a contenção de defeitos é bem-vindo. Uma possível estratégia é alocar os casos de teste de acordo com o perfil do testador de forma a maximizar a produtividade. Entretanto, otimizar a alocação de casos de teste não é uma tarefa trivial: em grandes companhias, gerentes de teste são responsáveis por alocar centenas de casos de teste aos testadores disponíveis ao início de uma nova execução. Neste trabalho nós propomos dois algoritmos para a alocação automática de casos de teste e três perfis para os testadores baseados em sistemas de recomendação (o mesmo tipo de sistema que recomenda, por exemplo, um livro na Amazon.com ou um filme no Netflix.com). Cada um dos algoritmos de alocação pode ser combinado com os três perfis de testador, resultando em seis sistemas de alocação possíveis: Exp-Manager, Exp-Blind, MO-Manager, MO-Blind, Eff-Manager, e Eff-Blind. Nossos sistemas de alocação consideram a efetividade (defeitos válidos encontrados no passado) e experiência do testador (habilidade em executar testes com determinadas características). Com o objetivo de comparar os nossos sistemas de alocação com a alocação do gerente e com alocações aleatórias, um experimento controlado, utilizando 100 alocações com pelo menos 50 casos de teste cada uma, foi realizado em um cenário industrial real. Os sistemas de alocação foram avaliados através das métricas de precisão, *recall* e taxa de não-alocação (percentual de casos de teste não alocados). Em nosso experimento, a aplicação da ANOVA (uma técnica estatística utilizada para verificar se as amostras de dois ou mais grupos são oriundas de populações com médias iguais) e do teste de Tukey (um procedimento de comparações múltiplas para identificar quais médias são significativamente diferentes entre si) mostraram que o Exp-Manager supera os demais sistemas de alocação com respeito às métricas de precisão e *recall*. Todos os sistemas de alocação mostraram-se superiores ao algoritmo randômico. A precisão média (entre os sistemas de alocação) variou de 39.32% a 64.83% enquanto o *recall* médio variou de 39.19% a 64.83%; para a métrica de não-alocação, três sistemas de alocação (Exp-Manager, Exp-Blind e MO-Blind) apresentaram um melhor desempenho alcançando taxa zero de não-alocação para todas as alocações de testes. A taxa média de não-alocação variou de 0% a 2.34% (para a métrica não-alocação, quanto menor, melhor). No cenário industrial real onde o nosso trabalho foi realizado, gerentes de teste gastam de 16 a 30 dias de trabalho por ano com a atividade de alocação de casos de teste. Nossos sistemas de alocação podem ajudá-los a realizar esta atividade de forma mais rápida e mais eficaz.

Palavras-chave: Teste de Software, Sistemas de Recomendação, Testes Manuais, Alocação de Casos de Teste

Abstract

Software testing is an arduous and expensive activity. In the context of manual testing, any effort to reduce the test execution time and to increase defect findings is welcome. One approach is to allocate test cases according to the testers profile in a way to maximise testing productivity. However, optimising the allocation of manual test cases is not a trivial task: in large companies, test managers are responsible for allocating hundreds of test cases among several testers. In this work we implemented 2 assignment algorithms for test case allocation and defined 3 tester profiles based on recommender systems (the same kind of system that recommends, for example, a book at Amazon.com or a movie at Netflix.com). Each assignment algorithm can be combined with 3 tester profiles, which results in six possible allocation systems: Exp-Manager, Exp-Blind, MO-Manager, MO-Blind, Eff-Manager, and Eff-Blind. Our allocation systems take into account the tester's effectiveness (valid defects found in the past) and expertise (ability to run tests with certain characteristics). We performed a controlled experiment that uses 100 test allocations, each one with at least 50 test cases, from a real industrial setting in order to compare our allocation systems to the manager's allocation and to random allocations. The allocation systems were evaluated in terms of precision, recall and unassignment (percentage of test cases the algorithm could not allocate). In our experiment, the application of ANOVA (a general technique that can be used to test the hypothesis that the means among two or more groups are equal) plus the Tukey's test (a multiple comparison procedure to find which means are significantly different from one another) showed us that the Exp-Manager system outperforms the other allocation systems with respect to the precision and recall metrics. All the allocation systems demonstrated to be superior to the random algorithm. The average precision (among the allocation systems) varied from 39.32% to 64.83% while the average recall ranged from 39.19% to 64.83%; For unassignment, three of our six allocation systems presented a better performance by achieving zero unassignment rate for all the allocation inputs, namely Exp-Manager, Exp-Blind and MO-Blind. The average unassignment varied from 0% to 2.34% (for unassignment, the lower the better). In the real industrial setting in which our work was developed, managers spend from 16 to 30 working days a year on test case allocation. Our algorithms help them do it faster and better.

Keywords: Software Testing, Recommender Systems, Manual Testing, Test Allocation

Contents

1	Introduction	1
1.1	Context, Objectives and Contributions	2
1.2	Dissertation Organisation	4
2	Background	5
2.1	Software Testing	5
2.1.1	Test Levels	6
2.1.1.1	Component Testing	6
2.1.1.2	Integration Testing	6
2.1.1.3	System Testing	7
2.1.1.4	Acceptance Testing	7
2.1.2	Test Types	7
2.1.2.1	Functional Testing	8
2.1.2.2	Non-functional Testing	8
2.1.2.3	Structural Testing (Testing of Software Structure/Architecture)	8
2.1.2.4	Re-testing and Regression Testing: Testing Related to Changes	9
2.1.3	Test Techniques: Black-box vs White-box	9
2.1.4	Testing Artifacts	10
2.1.5	Manual Testing	11
2.1.6	Test Allocation	12
2.1.7	The TTC Terms and Concepts	12
2.1.7.1	Test Case Keywords	13
2.1.7.2	The Anatomy of a Defect	13
2.1.7.3	The Lifecycle of a Defect Report	14
2.1.7.4	Raising / Referencing Defect Reports	15
2.2	TF-IDF	16
2.3	Recommender Systems	17
2.3.1	Symbolic Data Analysis (SDA)	18
2.3.2	Recommender Systems Supported by Symbolic Data Analysis	19
2.4	Statistical Tests	19
2.5	Controlled Experiments	21
2.6	Concluding Remarks	21

3	Recommender Systems and Testing	22
3.1	Recommender Systems Applied to the Testing Domain	22
3.2	Test Cases and Tester Profiles	24
3.2.1	The Expertise Profile	26
3.2.2	The Effectiveness Profile	28
3.2.3	The Multi-objective Profile	29
3.3	Calculating Similarity	30
3.4	Concluding Remarks	32
4	The Test Allocation Systems	33
4.1	The TCR Implementation and Architecture	33
4.2	The Assignment Algorithms	34
4.2.1	The Manager-Based Assignment	36
4.2.2	The Blind Assignment	39
4.3	On Performance	41
4.4	Concluding Remarks	42
5	Recommender Algorithms Evaluation	43
5.1	Introduction	43
5.1.1	Problem Statement	43
5.1.2	Research Objective	43
5.1.3	Context	44
5.2	Experimental Planning	44
5.2.1	Goals	44
5.2.2	Participants	46
5.2.3	Experimental Material	46
5.2.4	Tasks and Procedures	48
5.2.5	Statistical hypotheses	49
5.2.6	Experiment Design	49
5.3	Execution	50
5.4	Analysis	50
5.4.1	M_1 : Unassignment	50
5.4.2	M_2 : Strict Precision	51
5.4.3	M_3 : Strict Recall	53
5.4.4	M_4 : Approximate Precision	54
5.4.5	M_5 : Approximate Recall	55
5.5	Interpretation	56
5.5.1	Evaluation of results and implications	57
5.5.2	Threats to Validity	59
5.6	Concluding remarks	60

6	Related Work	61
6.1	Not All Classes are Created Equal: Toward a Recommendation System for Focusing Testing	61
6.2	Recommending Emergent Teams	61
6.3	Allocating Global Software Teams in Software Product Line Projects	63
6.4	Expertise Recommender	64
6.5	Who Should Fix this Bug?	65
6.6	Expertise Browser: A Quantitative Approach to Identifying Expertise	66
6.7	Predicting the Fix Time of Bugs	68
6.8	PR-Miner	69
6.9	Dynamine	69
6.10	Concluding Remarks	69
7	Conclusion and Future Work	71
7.1	Future Work	73

List of Figures

1.1	Test case allocation.	2
2.1	Example of a test case.	11
2.2	A sample of a Bugzilla defect report from Mozilla project.	14
2.3	The default lifecycle of a Bugzilla defect report.	15
3.1	Example of a test case.	25
3.2	Building the <i>expertise</i> profile.	27
3.3	Building the <i>effectiveness</i> profile.	29
3.4	Components for TC05 and Tester03.	31
4.1	The TCR architecture.	33
4.2	The TCR user interface.	34
4.3	TCR creating the tester profiles and the test case descriptions.	35
4.4	Similarity computation.	36
4.5	Manager-based recommendation.	37
4.6	Blind recommendation.	40
5.1	Amount of test cases and number of testers per allocation input.	47
5.2	Unassigned test cases per allocation system.	51
5.3	Strict precision per allocation system.	52
5.4	Strict recall per allocation system.	53
5.5	Approximate precision per allocation system.	55
5.6	Approximate precision - confidence intervals and histograms.	56
5.7	Approximate recall per allocation system.	58
5.8	Approximate recall - confidence intervals and histograms.	59
6.1	Evolution Cost and PageRank view.	62
6.2	EEL - ranked list of developers and the ways to initiate a communication.	62
6.3	Expertise Request dialog.	65
6.4	Recommendation response.	65
6.5	ExB user interface.	67

List of Tables

3.1	The allocation systems.	24
3.2	Content description of the test case TC25.	24
3.3	Test case TC25 with attributes, values and weights.	25
3.4	Test cases and their attributes.	26
3.5	Execution history.	27
3.6	Tester03's expertise profile.	28
3.7	Defect Report IDs per tester.	29
3.8	Defect Reports - Status and Resolution	30
3.9	Tester03's effectiveness profile.	30
3.10	Calculating similarities for Tester03.	32
4.1	Similarities.	36
4.2	Manager-based assignment: amount of test cases per tester.	38
4.3	Manager-based: first iteration.	39
4.4	Manager-based: second iteration.	39
4.5	Manager-based: third iteration.	39
4.6	Manager-based: fourth iteration.	39
4.7	Manager-based: fifth iteration.	40
4.8	Manager-based: final recommendation.	40
4.9	The Blind algorithm allocation.	41
4.10	Blind algorithm: final recommendation.	41
5.1	Test case unassignment.	51
5.2	Strict Precision - Comparisons for all pairs using Tukey's HSD	52
5.3	Strict Precision - Allocation systems classified in levels after Tukey's test.	53
5.4	Strict Recall - Comparisons for all pairs using Tukey's HSD	54
5.5	Strict Recall - Allocation systems classified in levels after Tukey's test.	54
5.6	Approximate Precision.	57
5.7	Approximate Recall.	60
6.1	Recommending Emergent Teams - Average precision and recall.	63
6.2	Performance of prediction models computed with initial attribute values.	68

List of Algorithms

1	Manager-based algorithm	38
2	Blind algorithm	41

CHAPTER 1

Introduction

Nowadays, almost everyone is aware of software systems. We encounter them in our homes, at work, and even while shopping. More and more, they are part of our lives. We use software in day-to-day business applications such as banking and in consumer products such as cars and washing machines [GVEB08]. However, software systems do not always work as expected: an error on a bill, a delay when waiting for a credit card to process and a website that did not load correctly are common examples of issues that may happen because of software problems.

Of course not all software systems carry the same level of risk and not all problems have the same impact in case they occur. Many software defects are merely annoying or inconvenient but some can have extremely serious consequences, either financially or as a threat to human well-being. The European Space Agency's *Ariane 5 Flight 501*, for example, was destroyed 40 seconds after takeoff (June 4, 1996). The US\$1 billion prototype rocket self-destructed due to a defect in the on-board guidance software [Dow97]. Another example is related to the *Therac-25*: a radiation therapy machine that was directly responsible for at least five patient deaths in the 1980s when it administered excessive quantities of X-rays due to a software defect [LT93].

As the consequences of a software defect can be either trivial or catastrophic, rigorous testing is necessary in order to reduce failures in the operational environment and increase the quality of the operational system. However, testing everything (all combinations of inputs and preconditions in a software system) is not feasible, so deciding how much testing is enough should take account of the level of risk, safety, and project constraints such as time and budget.

Testing can help us to measure the quality of software in terms of the number of defects found, the tests run, and the system covered by the tests. It can give confidence in the quality of the software if it finds few or no defects, provided that the testing is sufficiently rigorous. Of course, a poor test may uncover few defects and provide a false sense of security. A well-designed test will uncover defects if they are present and, if such a test passes, we will rightly be more confident in the software and be able to assert that the overall level of risk of using the system has been reduced. When testing does find defects, the quality of the software system increases when those defects are fixed, provided the fixes are carried out properly [GVEB08].

As we saw, testing is an essential process that assesses the behaviour and the quality of a software; and given that this process can be quite arduous and costly (testing is responsible for up to 40% of the total cost of a project [Som06]), all effort and tool support that optimises its costs are welcome.

In this work, we use recommender systems to help the test manager to allocate test cases to testers, thus reducing the time spent in the allocation process. Our approach can also reduce the time taken to run the tests and, hopefully, increase the amount of defects found at the end of a given test execution.

1.1 Context, Objectives and Contributions

This work is developed in the context of a company that outsources testing services to a mobile phone manufacturer. Due to contractual reasons the company's name cannot be disclosed and, from now on, we call it **TTC** (*"The Testing Company"*). TTC performs black-box testing, mostly executed *manually*. One important activity at TTC is the test case allocation (Figure 1.1). Typically, a test manager has to allocate hundreds of test cases to be executed by the testers available at that moment. The manager's input to this task is a list of test cases and a list of testers. We call these lists the *allocation input*. The outcome of an allocation is a list of pairs (test case, tester), which indicates who should run which test case. We call this list a *test plan*. In the example of Figure 1.1, the test cases TC01, TC10, and TC15 are allocated to tester T04.

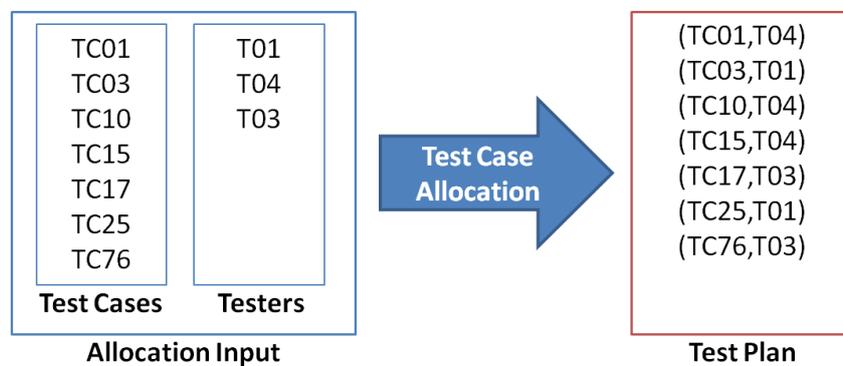


Figure 1.1 Test case allocation.

In order to perform this task, the test manager takes between 15 to 35 minutes, depending on the amount of tests to be allocated; and this task is executed routinely at TTC: around 10 times per week. This means that the time spent on this task per year varies from 16 to 30 working days. A bad allocation has an impact on the execution time of the test suite and the effectiveness of it (fewer defects found). By allocating a test to an experienced tester, which is someone who has executed that test (or similar tests) in the past, the execution time may reduce. Analogously, a tester who has found defects executing a particular test in the past is a good candidate to execute that same test again or other tests that are similar to that. At TTC the same test can be run several times over a month, either for regression testing or in different mobile phone models. Note that, in principle, defect finding should not depend on who is executing which test case. However, our context is that of *manual* execution. Test cases are written in English. They are usually ambiguous and open to different interpretations. So, we believe a wise choice of a tester can make a difference on defect finding.

We implemented 2 assignment algorithms for test case allocation and defined 3 tester profiles based on recommender systems (the same kind of system that recommends you a book at Amazon.com or a movie at Netflix.com) [Kon04]. Each assignment algorithm can be combined with 3 tester profiles, which results in six possible *allocation systems*: Exp-Manager, Exp-Blind, MO-Manager, MO-Blind, Eff-Manager, and Eff-Blind. The rationale behind these names will be explained in Chapter 3. In our case, our systems recommend a test case for a

given tester. We compared all allocation systems to the manual allocations of a test manager. We also developed a random allocation algorithm for control. The algorithms ran at 100 allocation inputs that had been previously allocated by hand by the managers. These allocations had been performed manually from November 2009 to September 2010. We compared the allocation systems among themselves and with respect to the manager's allocation and to a random algorithm. The results showed that the Exp-Manager system outperforms all the others with respect to the precision and the recall metrics. All the allocation systems demonstrated to be superior to the random algorithm. We compared the allocation systems with the manager's allocation in terms of precision, recall and unassignment (percentage of test cases the algorithm could not allocate). The average precision (among the allocation systems) varied from 39.32% to 64.83% while the average recall ranged from 39.19% to 64.83%; For unassignment, three of our six allocation systems presented a better performance by achieving zero unassignment rate for all the allocation inputs, namely Exp-Manager, Exp-Blind and MO-Blind. The average unassignment varied from 0% to 2.34% (for unassignment, the lower the better).

Recommender systems have been applied to almost all activities of Software Engineering. Several works describe recommender systems for allocating tasks to people [AHM06, JKZ09, MM07, MH02, PdSRE10], defect prevention and debugging [GPG10, LZ05a, LZ05b]. Only a few are directly related to testing [KRGA08], and none (as far as we know) is related to allocating test cases to testers. Even the book "Artificial Intelligence Methods In Software Testing" [LKB04], which collects a representative sample of artificial intelligence applications in the areas of software testing, does not mention any application of recommender systems applied to software testing.

The main contributions of this dissertation are:

- The application of recommender systems to test case allocation;
- The proposal of 6 allocation systems for allocating test cases to testers;
- The implementation of a tool that mechanises this activity. Test case allocation with the support of a recommender system can be done faster than before. Moreover, new managers who are not familiar with the testers and the test cases can benefit from an initial allocation produced by the tool;
- An experiment in which all allocation systems are compared to the manager's performance in terms of unassignment, precision and recall;
- A guide for test managers to choose the right allocation system depending on the context.

Some techniques and equations presented in this dissertation are not original contributions of this work. The way we model the test cases and the tester profiles based on symbolic data (Section 3.2) as well as the equation we use to calculate the similarity between a test case and a tester (Section 3.3) were proposed by Bezerra and Carvalho [BdC10]. The allocation algorithms (Chapter 4), however, are an original contribution of this work. Part of the material available in Chapters 3 and 4 has already been published in [MIM10] which was developed in the context of the **National Institute of Science and Technology for Software Engineering (INES)**.

1.2 Dissertation Organisation

In this section we outline the structure of the subsequent chapters.

- **Chapter 2** introduces an overview of the main fundamental concepts related to this work. Initially we describe the different test levels, test types and test design techniques. We then explain how the test allocation process works at TTC as well as we present some terms and concepts that are part of TTC's routine. We conclude by introducing some concepts related to TF-IDF (a statistical measure used to evaluate how important a word is to a document), recommender systems, statistical tests and controlled experiments.
- **Chapter 3** presents how recommender systems can be applied to the testing domain. We start by presenting the tester profiles and the assignment ways we are proposing in this work. Then, we describe how we model test cases and tester profiles and we illustrate how the tester profiles are built. To conclude, we describe how we calculate the similarity between a test case and a tester.
- **Chapter 4** shows how the allocation systems assign test cases to testers. In this chapter we present the tool, which encompasses our allocation systems, that was developed as part of this work and we discuss some implementation details.
- **Chapter 5** reports a controlled experiment that was performed to evaluate the effectiveness of the allocation systems. We start by stating the problem investigated and defining our research objective. Then, we give some details about the experimental planning and its execution. We report the results of our experiment and we address the threats to validity.
- In **Chapter 6** , we present an overview of previous work on recommender systems applied to testing, debugging or people recommendation.
- **Chapter 7** summarises our work and discusses the limitations and contributions of our approach, and future work.

Background

In this chapter, we introduce the main fundamental concepts related to this work. We start by describing the different test levels, test types and test design techniques. We explain how the test allocation process works at TTC as well as we present some terms and concepts that are part of TTC's routine. Finally we introduce some concepts related to TF-IDF, recommender systems, statistical tests and controlled experiments.

2.1 Software Testing

Software systems have become an integral part of our lives, from business applications (e.g., banking) to consumer products (e.g., cars). Most people have had an experience with software that did not work as expected. Software that does not work correctly can lead to many problems, including loss of money, time or business reputation, and could even cause injury or death. A primary purpose of testing is to detect software failures so that defects may be discovered and corrected.

A human being can make an **error** (e.g., an incorrect instruction in a computer program), which produces a **defect** (fault, bug) in the program code, or in a document. If a defect in code is executed, the system may fail to do what it should do (or do something it should not), causing a **failure**. Defects in software, systems or documents may result in failures, but not all defects do so. These definitions are described in the IEEE Standard Glossary of Software Engineering Terminology [IEE90].

Defects occur because human beings are fallible and because there is time pressure, complex code, complexity of infrastructure, changing technologies, and/or many system interactions. However, failures are not caused by human beings only; they can happen by environmental conditions as well (for example, radiation, magnetism, and electronic fields can influence the execution of software by changing the hardware conditions).

Testing of systems and documentation can help to reduce the risk of failures occurring during operation and contribute to the quality of the software system. It cannot establish, however, that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions [KFN99].

A common perception of testing is that it consists only of running tests, i.e., executing the software. This is part of testing, but not all of the testing activities. Test activities exist before and after test execution. These activities include planning and control, choosing test conditions, designing and executing test cases, checking results, evaluating exit criteria, reporting on the testing process and system under test, and finalising closure activities after a test phase has been

completed. Testing also includes reviewing documents (including source code) and conducting static analysis.

2.1.1 Test Levels

Tests are frequently grouped by where they are used in the software development process, or by the level of specificity of the test. In general, they are divided into four groups: Component Testing, Integration Testing, System Testing and Acceptance Testing.

2.1.1.1 Component Testing

Component testing (also known as unit, module or program testing) searches for defects in software modules, programs, objects, classes, etc., that are separately testable. It may be done in isolation from the rest of the system, depending on the context of the development life cycle and the system. Test cases are derived from artifacts such as the specification of a component, the software design or the data model.

Frequently, component testing occurs with access to the code being tested and with the support of a development environment, such as a unit test framework or debugging tool. It usually involves the programmer who wrote the code. Defects are typically fixed as soon as they are found, without formally managing these defects.

Preparing and automating test cases before coding is one approach to component testing. This is called test-driven development or test-first approach [GVEB08]. This approach is highly iterative and is based on cycles of (i) developing test cases; (ii) building and integrating small pieces of code; (iii) executing the component tests; and (iv) correcting any defects found.

2.1.1.2 Integration Testing

The main goal of integration testing is to test the interfaces between components, the interactions with different parts of a system, and the interfaces between systems. Usually, the following levels of integration testing are considered:

- **Component integration testing** focuses on the interactions between software components and is done after component testing.
- **System integration testing** focuses on the interactions between different systems or between hardware and software and may be done after system testing.

The greater the scope of integration, the more difficult it becomes to isolate defects to a specific component or system, which may lead to increased risk and additional time for troubleshooting.

At each stage of integration, testers concentrate exclusively on the integration itself. If *module A* is being integrated with *module B*, for example, testers are interested in assessing the communication between the modules, not the functionality of the individual module as done during component testing.

2.1.1.3 System Testing

System testing is concerned with the behaviour of a whole system or product. That said, the test environment should correspond to the production environment as much as possible in order to minimise the risk of environment-specific failures not being found in testing.

Test cases are derived from requirements specifications, business processes, use cases, or other high level descriptions of the system behaviour, interactions with the operating system, and system resources.

System testing of functional requirements uses the most appropriate specification-based (black-box) techniques for the feature of the system to be tested. Structure-based techniques (white-box) may then be used to assess the thoroughness of the testing with respect to a structural element.

2.1.1.4 Acceptance Testing

The main goal of acceptance testing is to establish confidence in the system, parts of the system or specific non-functional characteristics of the system. Finding defects is not the main focus in acceptance testing. Although acceptance testing may assess the system's readiness for deployment and use, it is not necessarily the final level of testing: a large-scale system integration test may come after the acceptance test for a system, for example.

Acceptance testing may occur at various times in the life cycle. For example, acceptance testing of the usability of a component may be done during component testing whereas acceptance testing of a new functional enhancement may come before system testing. Typical forms of acceptance testing include the following:

- **User acceptance testing.** Verifies the fitness for use of the system by business users.
- **Operational (acceptance) testing.** Evaluates the acceptance of the system by the system administrators, including testing of backup/restore, user management, maintenance tasks, etc.
- **Contract and regulation acceptance testing.** Contract acceptance testing is performed against a contract's acceptance criteria for producing custom-developed software. Acceptance criteria should be defined when the parties agree to the contract. Regulation acceptance testing is performed against any regulations that must be adhered to, such as government, legal or safety regulations.
- **Alpha and beta (or field) testing.** Sometimes it is necessary to get feedback from potential or existing customers before putting up a software product for sale commercially. Alpha testing is performed at the developing organisation's site but not by the developing team. Beta testing, or field-testing, is performed by customers or potential customers at their own locations.

2.1.2 Test Types

A test type is focused on a particular test objective, which could be any of the following:

- Functional testing. A function to be performed by the software;
- Non-functional testing. A non-functional quality characteristic, such as reliability or usability;
- Structural testing. The structure or architecture of the software or system;
- Re-testing and Regression Testing. Confirming that defects have been fixed (confirmation testing) and looking for unintended changes (regression testing).

2.1.2.1 Functional Testing

The functions that a system, subsystem or component should perform are described in work products such as a requirements specification, use cases, or a functional specification. The functions describe “*what*” the system does. Functional tests are based on functions and features and their interoperability with specific systems, and may be performed at all test levels. Functional testing considers the external behaviour of the software (black-box testing) [Bei90].

There are different types of functional testing. For example, **security** testing investigates the functions relating to detection of threats, such as viruses, from malicious outsiders. Another type of functional testing, **interoperability** testing, evaluates the capability of the software product to interact with one or more specified components or systems.

2.1.2.2 Non-functional Testing

Non-functional testing includes, but is not limited to, **performance** testing, **load** testing, **stress** testing, **usability** testing, **maintainability** testing, **reliability** testing and **portability** testing. It is the testing of “*how*” the system works and it may be performed at all test levels [GVEB08].

The term non-functional testing describes the tests required to measure characteristics of systems and software that can be quantified on a varying scale, such as response times for performance testing. Non-functional testing considers the external behaviour of the software and, in most cases, uses black-box test design techniques to accomplish that.

2.1.2.3 Structural Testing (Testing of Software Structure/Architecture)

Structural (white-box) testing may be performed at all test levels. Structural techniques are best used after specification-based techniques, in order to help measure the thoroughness of testing through assessment of coverage of a type of structure (typically, the source code) [Bla02].

Coverage is the extent that a structure has been exercised by a test suite, expressed as a percentage of the items being covered. If coverage is not 100%, then more tests may be designed to test those items that were missed to increase coverage. Tools can be used at all test levels, but especially in component testing and component integration testing, to measure the code coverage of elements, such as statements or decisions.

2.1.2.4 Re-testing and Regression Testing: Testing Related to Changes

After a defect is detected and fixed, the software should be **re-tested** to confirm that the original defect has been successfully removed. This is called **confirmation** [GVEB08].

Regression testing is the repeated testing of an already tested program, after modification, to discover any defects introduced or uncovered as a result of changes. It may be performed at all test levels, and includes functional, non-functional and structural testing. As the regression test suites are run many times and generally evolve slowly, regression testing is a strong candidate for automation.

2.1.3 Test Techniques: Black-box vs White-box

From the point of view of test design techniques, software testing is traditionally divided into black-box and white-box testing.

- **Black-Box Testing:** Black-box testing assumes that the system is viewed as a closed box, hence the testers should have no information about how the code behaves or how it is structured. Test conditions and test cases are selected based on an analysis of the test basis documentation for a component or system without reference to its internal structure.

On this method, the testers are completely unconcerned about the internal behaviour and structure of the program. Instead, they concentrate on finding circumstances in which the program does not behave according to its specifications. In this approach, test data are derived solely from the specifications (i.e., without taking advantage of knowledge of the internal structure of the program) [MS04].

There are many advantages of using black-box testing. We can cite, for example, the fact that test cases are done from a user's point of view and they can be designed as soon as the specifications are complete. This test design technique also helps to expose any ambiguities or inconsistencies in the specifications. In addition to that, since the tester and the developer are independent of each other, testing is balanced and unbiased.

Despite these benefits, black-box testing has its drawbacks: test cases are challenging to design without having clear functional specifications (which is usually the case) and there is a high probability of repeating tests already performed by the programmer.

- **White-Box Testing:** White-box testing (a.k.a. clear box testing, glass box testing, transparent box testing, or structural testing) is a method of testing software that tests the internal structures of the component or the system [IEE90]. In white-box testing an internal perspective of the system, as well as programming skills, are required and used to design test cases. The tester chooses inputs to exercise paths through the code and determine the appropriate outputs. Though this method of test design can uncover many errors or problems, it might not detect unimplemented parts of the specification or missing requirements.

White-box testing is an important method for the early detection of errors during software development. In this process, test case generation plays a crucial role, defining

appropriate and error-sensitive test data. White-box testing strategies include designing tests such that every statement in the program is executed at least once, or requiring every function to be individually tested.

One of the main advantages of using white-box testing is that it becomes very easy to find out which type of input/data can help in testing a given system or component effectively as the knowledge of internal coding structure is known. However, such knowledge can also be considered as a disadvantage as a skilled tester is needed to carry out this type of testing.

Besides black-box and white-box, there are other testing design techniques considered in the literature such as the grey-box, which is a hybrid approach of black-box and white-box. It involves having knowledge of internal data structures and algorithms for purposes of designing the test cases, but testing at the user, or black-box level. Black-box, white-box and grey-box testing are complementary techniques and may be used for any given test activity, regardless of which level of testing is being performed.

2.1.4 Testing Artifacts

The software testing process produces several artifacts. Below we list the most common ones:

- **Test plan.** A document that prescribes the scope, approach, resources, and schedule of the testing activities. It identifies the items to be tested, the features to be tested, the testing tasks to be performed, the personnel responsible for each task, and the risks associated with the plan [IEE98].
- **Test suite.** A collection of test cases for a component or system under test. It is often associated with detailed instructions or goals for a particular test suite. In some test suites the post condition of one test case is often used as the precondition for the next one.
- **Test case.** A set of input values, execution preconditions, expected results and execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement [IEE90].

A test case is basically composed by an *id* (or *name*), a *description* (a short description that highlights the test case objective), and a sequence of *steps* and *expected results* (ideally, all the steps should have at least one expected result associated with it). Additional fields may include, but are not limited to: feature ID, component, test case category, test case level, test case type, preconditions, related requirements, cleanup, etc. Figure 2.1 shows a test case written on Testlink [Tes09] that illustrates very closely the structure of a test case used at TTC. We cannot show the actual test case as it is proprietary data of TTC. The example using Testlink captures the crucial information from the actual TTC test case.

Figure 2.1 shows a test case that verifies if it is possible to send a text message, also known as SMS (“*Short Message Service*”), to an e-mail address. In the first step, the tester launches the text messaging application and makes sure it is working fine. Then,

TEST CASE TC25		
Summary Ability to send SMS to e-mail address.		
#	Step actions	Expected Results
1	Launch text messaging application	Text messaging application is properly launched
2	Enter a valid e-mail address in the 'recipients' field, write some text in the body of the SMS and send it	User is notified that the message was successfully sent
3	From PC, login in the e-mail account used in the previous step	Verify that the SMS is properly received in the e-mail address
Keywords: Category: feature testing Component: e-mail Component: SMS Feature ID: 2801 - SMS to e-mail		

Figure 2.1 Example of a test case.

a new text message is composed and sent to a valid e-mail address. Finally, the tester access the e-mail account from a personal computer to make sure that the text message is received successfully.

- **Test summary report.** A document that summarises testing activities and results. It also contains an evaluation of the corresponding test items [IEE98].
- **Test phase.** A distinct set of test activities collected into a manageable phase of a project, e.g. the execution activities of a test level.
- **Traceability matrix.** A matrix that records the relationship between two or more products of the development process [IEE90]. In the testing context, it correlates requirements or design documents to test documents. It may be used to change test cases when the source documents are changed, or to verify that the test results are correct. The IEEE standard for software test documentation [IEE98] states that it should be available in the section “Approach” of a Test Plan.

2.1.5 Manual Testing

Manual testing is the process of manually testing software to detect failures. It is a laborious activity that requires the tester to possess a certain set of qualities: to be patient, observant, speculative, creative, innovative, open-minded, resourceful, and skillful. To ensure completeness of testing, the testers often follow a previously defined test plan that leads them through a set of important test cases.

A systematic approach focuses on predetermined test cases and generally involves the following steps [IEE98].

1. Define a test plan.
2. Write detailed test cases, identifying clear and concise steps to be taken by the tester, with expected outcomes.
3. **Assign the test cases to testers** (see Section 2.1.6), who manually follow the steps and record the test results.
4. Generate a test report, detailing the findings of the testers.

2.1.6 Test Allocation

In the context of manual testing, test allocation is the process of assigning the test cases, which were selected to be run, for the members of a test team. Below we describe where in the testing process the test allocation takes place at TTC.

1. A request for a new test execution arrives;
2. The test plan is released;
3. A test architect selects the test cases that should be run based on the information available in the request (e.g. features to be tested, features not to be tested, approach, environmental needs, schedule, etc);
4. The test manager reviews the test cases selected for that execution and assigns them (manually) to the testers available at that moment;
 - (a) During the allocation process, depending on the size and characteristics of a given test suite, the manager may opt to either assign it to a single tester or split it among several testers.
 - (b) During the test execution, if a tester that was busy in another previous activity becomes available, the test manager may ask that person to join the ongoing test execution and assign some test cases to her.
5. The test execution begins;
6. Test results are logged and defects are reported (if needed);
7. A test summary report is generated and the test execution finishes.

2.1.7 The TTC Terms and Concepts

Below we introduce some terms and concepts used at TTC that is referenced throughout this document. Some of them have the same meaning as defined in the classic literature for software testing whereas other terms have a slightly different meaning.

2.1.7.1 Test Case Keywords

Each test case is associated with some *keywords* to make easier the process of selecting test cases during the planning phase. These keywords are also useful to generate metrics such as “test cases distribution per feature”, “test coverage per component”, etc. In order to illustrate our techniques, we use only the keywords *feature*, *component* and *category* in this work (the actual database of TTC stores some other keywords).

- **Feature.** A clustering of individual requirements that describes a cohesive, identifiable unit of functionality [TWFL98]. For example, “SMS to e-mail” is a feature of *Text Messaging* application. In the TTC, each feature has a unique identifier and each test case is associated to one or more features.
- **Component.** A grouping of various features from the same domain. Each test case is also associated to one or more components. In Figure 2.1, the test case TC25 is associated to the components “e-mail” and “SMS”.
- **Category.** A category distinguishes a test case among the different types of test cases and can assume values like: “functional”, “feature testing”, “interaction”, “performance”, “stress”, “load”, “sanity”, “localisation”, “interoperability”, etc. Each test case can be associated to one or more categories. The test case in Figure 2.1 belongs to the “feature testing” category.

2.1.7.2 The Anatomy of a Defect

Figure 2.2 shows an example of a defect report stored in Bugzilla. TTC does not use Bugzilla. They use a proprietary software. However, as Bugzilla stores and manages defects in a similar way to the TTC defect management system, we can use it to illustrate our approach without any loss of information (in any case, the techniques described here are not dependent on any particular defect management system or test case repository). Each defect report includes pre-defined fields, free-form text, attachments, and dependencies. As we can see, there are lots of pre-defined fields (*Status*, *Keywords*, *Product*, *Component*, *Platform*, etc).

In order to fulfil the requirements of our approach, we are mainly concerned about the following fields:

- **Status / Resolution.** These fields define exactly what state the defect is in (from before being confirmed as a defect, through to fix and the fix confirmation). The different possible values for *Status* and *Resolution* fields will be discussed in more details later on.
- **Reporter.** The person who filed a given defect report.
- **Modified date.** By analysing this field we can get the date when a given defect achieved a final resolution such as “*fixed*”, for example.

Bug 628768 - mark as read toggle broken [Last Comment](#)

Status: RESOLVED FIXED
Whiteboard:
Keywords: regression, relnote
Product: Thunderbird
Component: Mail Window Front End
Version: Trunk
Platform: x86 Windows 7
Importance: -- major (vote)
Target Milestone: Thunderbird 3.3a3
Assigned To: Jim Porter (:squib)
QA Contact: front-end
URL:
Depends on: [649898](#)
Blocks: [TB2SM 575732](#)
[Show dependency tree / graph](#)

Reported: 2011-01-25 12:39 PST by Asa Dotzler [:asa]
Modified: 2011-04-13 20:12 PDT ([History](#))
CC List: 8 users ([show](#))

Flags:
 mbanner: in-testsuite+
See Also:

blocking-thunderbird3.3: alpha3+ ▾
status-thunderbird3.3: ---
blocking-thunderbird3.2: --- ▾
status-thunderbird3.2: ---
blocking-thunderbird3.1: --- ▾
status-thunderbird3.1: ---
blocking-thunderbird3.0: --- ▾
status-thunderbird3.0: ---

Attachments

Now with tests (8.71 KB, patch) 2011-01-31 14:44 PST, Jim Porter (:squib)	bwinton: review+ clarkbw: ui-review+	Details Diff Review (Splinter)
Add an attachment (proposed patch, testcase, etc.)		Show Obsolete (1) View All

[Summon comment box](#)

Asa Dotzler [:asa] 2011-01-25 12:39:53 PST [Description](#)

I use the "m" key to mark messages that I've just read as unread. That broke recently. Now it seems it can only unread messages as read which seems pretty useless to me. If I've got the message selected then it's mostly likely already marked read.

Keyboard shortcuts are critical for an email client. We can't let them break, even for a few days. This is really really painful and caused me to revert from nightly builds to the ill-performant previous release.

Figure 2.2 A sample of a Bugzilla defect report from Mozilla project.

2.1.7.3 The Lifecycle of a Defect Report

The *lifecycle* of a defect report, also known as workflow, can be customised by each company. In Figure 2.3 we show a graphical representation of the default workflow adopted by the Bugzilla system. According to its status, a defect can be classified either as an *open defect* or a *closed defect*.

Open defects do not have any resolution associated with them and they can be classified as:

- **Unconfirmed.** This defect report has recently been added to the database. Nobody has confirmed that the report is an actual defect.
- **Confirmed.** The report has been filled with the information of an actual defect.
- **In Progress.** This defect is not yet resolved, but is assigned to the proper person who is working on the defect.

Closed defects, on the other hand, must have a resolution associated with them. They can be classified as:

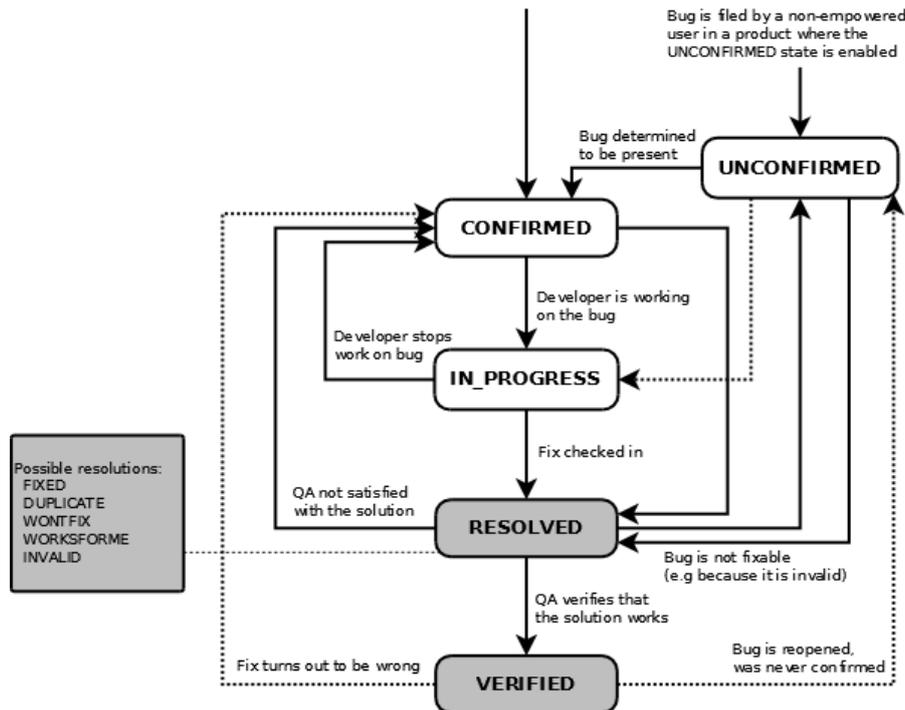


Figure 2.3 The default lifecycle of a Bugzilla defect report.

- **Resolved.** A resolution has been performed, and it is awaiting verification from the *Quality Assurance* (QA) team. From here defects are either reopened or are verified by the QA and marked as *Verified*. A defect report can be resolved in a number of ways:
 - **Fixed.** A fix for this defect is checked into the code and tested.
 - **Invalid.** The problem described is not a defect.
 - **Wontfix.** The problem described is a defect that will never be fixed.
 - **Duplicate.** The problem is a duplicate of an existing defect report.
 - **Worksforme.** Either all attempts of reproducing this defect were futile or the defect was present once, but is now not reproducible.
- **Verified.** Someone with QA privileges has looked at the defect and its resolution and agrees that the appropriate resolution has been taken. This is the final status for defect reports.

2.1.7.4 Raising / Referencing Defect Reports

When running a given test case, if any inconsistency between the actual result and the expected result is noticed, the tester changes the result of that test case to “*Fail*” and associates a defect report to it in order to track either a software fix or a test case update. In this case, the tester needs to access the defect management system to search for a defect report that describes the issue identified. If the tester does not find any defect report that describes precisely the issue

found, then a new one needs to be *raised*. Otherwise, the tester simply makes a *reference* to an existing defect report. When creating the effectiveness profile for a tester (Section 3.2.2) we are interested only in the *valid* defects a particular tester has raised/referred to. In our approach, we consider valid defects the ones that were closed as *Resolved* or *Verified* with resolution equal to *Fixed*.

2.2 TF-IDF

The *tf-idf* measure (**term frequency-inverse document frequency**) is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus (a large and structured set of texts). The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the corpus. For instance, suppose the outcome of *tf-idf* for a document d with respect to a set of documents D is (“the”,0.001), (“tester”,0.04), (“SMS”,1.5), etc. This means that the word “SMS” is the most important word in d with respect to D . “SMS” is not the word that occurs more often in D , but probably it occurs a lot in d (and in few other documents in D). In our context, the set D is the test suite and d is a test case.

Tf-idf combines two measures: term frequency (tf) and inverse document frequency (idf).

Term frequency counts the number of times a given term appears in a document. This counting is usually normalised to prevent a bias towards longer documents, which may have a higher term count regardless of the actual importance of that term in the document. The term frequency is denoted by $tf_{t,d}$, where t is the term being counted and d is the document [MRS08]. The term frequency suffers from a critical problem: all terms are considered equally important when it comes to assessing relevancy on a query. This is not a good thing as certain terms have little or no discriminating power in determining relevance. For instance, a collection of documents on the *software testing industry* is likely to have the term *test* in almost every document. For attenuating the effect of terms that occur too often in the collection, the *inverse document frequency* (*idf*) of a term t is used.

The inverse document frequency is a measure of the general importance of the term t obtained by dividing the total number of documents N by the number of documents containing the term t (denoted by df_t), and then taking the logarithm of that quotient.

$$idf_t = \log \frac{N}{df_t} \quad (2.1)$$

Thus, the idf_t of a rare term t is high, whereas the $idf_{t'}$ of a frequent term t' is likely to be low. For example, for $N = 100$ documents, suppose that the term “the” occurs in 98 of them, i.e. $df_{\text{“the”}} = 98$. Then, its $idf_{\text{“the”}} = 0.00877$. Now, suppose the term “software” occurs in just 10 documents, i.e. $df_{\text{“software”}} = 10$. The $idf_{\text{“software”}}$ is 1.

The concepts of *term frequency* and *inverse document frequency* can be combined to produce a composite weight for each term in each document. The *tf-idf* weighting scheme assigns to term t a weight in document d given by:

$$tf-idf_{t,d} = tf_{t,d} \cdot idf_t \quad (2.2)$$

In other words, $tf-idf_{t,d}$ assigns to term t a weight in document d that is:

1. Highest when t occurs many times within a small number of documents (thus lending high discriminating power to those documents);
 - Consider a document containing 100 words wherein the word “*company*” appears 50 times. Following the previously defined formulae, the term frequency (tf) for “*company*” is then $\frac{50}{100} = 0.50$. Now, assume we have 1,000 documents and “*company*” appears in 10 of those. Then, the inverse document frequency (idf) is calculated as $\log \frac{1,000}{10} = 2$. The $tf-idf$ score is the product of these quantities: $0.5 \cdot 2 = 1.0$.
2. Lower when the term occurs fewer times in a document, or occurs in many documents (thus offering a less pronounced relevance signal);
 - If, in the same document from the previous example, the word “*test*” appears 10 times, then $tf_{“test”} = \frac{10}{100} = 0.10$. Assuming that the word “*test*” appears in 100 documents out of 1,000, the $idf_{“test”} = \log \frac{1,000}{100} = 1$. Thus, $tf-idf_{“test”} = 0.10 \cdot 1 = 0.10$.
3. Lowest when the term occurs in virtually all documents.
 - Consider a scenario in which the word “*the*” appears 50 times in a document containing 100 words; and it appears in 990 documents out of 1,000. Then, the $tf_{“the”} = \frac{50}{100} = 0.5$ and $idf_{“the”} = \log \frac{1,000}{990} \approx 0.004$. The $tf-idf_{“the”}$ is, therefore, $0.5 \cdot 0.004 = 0.002$.

In this work we use the $tf-idf$ to analyse the relevant words occurring in a test case description in order to create the tester profiles. This process is described in Section 3.2.

2.3 Recommender Systems

Recommender systems are a sub-domain of information filtering system techniques that attempt to recommend information items (movies, TV programs, video on demand, music, books, news, images, web pages, scientific literature, etc.) or social elements (e.g. people, events or groups) that are likely to be of interest to the user. They can now be found in many modern applications that expose the user to a huge collection of items. Such systems typically provide the user with a list of recommended items they might prefer, or predict how much they might prefer each item. These systems help the user to decide on appropriate items, and ease the task of finding preferred items in the collection [SG09].

Since they help users to discover items they might not have found by themselves, recommender systems are considered a useful alternative to search algorithms. To perform this action, such system needs to know something about every user. Every recommender system must develop and maintain a user model, which we call a *user profile*, that captures the user preferences [JZFF10].

Recommender systems differ in the way they analyse data sources to develop notions of affinity between users and items, which can be used to identify well-matched pairs. There are two prevalent approaches:

- **Collaborative filtering.** Collaborative filtering is a method of making automatic predictions (filtering) about the interests of a user by collecting preferences or taste information from many users (collaborating). The underlying assumption of the collaborative filtering approach is that those who agreed in the past tend to agree again in the future. One approach is to collect user feedback in the form of ratings for items in a given domain and exploit similarities in rating behaviour amongst several users [MS10]. Thanks to collaborative filtering, Amazon.com can say, “People who bought *book A* also bought *book B*”. To generate this recommendation, all Amazon needs is a database of who has bought what (so it can calculate who are the “people like you”). Then any time a user that has tastes similar to yours buys a book that you have not bought, Amazon.com can recommend that book to you.
- **Content-based recommending.** Content-based approaches recommend items that are similar in content to items the user has liked in the past. Alternatively, the recommendation can be based on items that match attributes of the user. These approaches are sometimes also referred to as content-based filtering. By knowing more about a user or about an item, it is possible to make better personalised recommendations. For instance, suppose we are looking at the attribute “genre” of a movie. By knowing that a user liked “Star Wars” and “Blade Runner,” one may infer a predilection for the genre science fiction and, subsequently, recommend “Twelve Monkeys” [MS10].

2.3.1 Symbolic Data Analysis (SDA)

In many domains of human activities it is now quite common to record huge sets of data in large databases. It becomes a task of first importance to summarise these data in terms of their underlying concepts in order to extract new knowledge from them. These concepts can only be described by more complex type of data called “symbolic data” as they contain internal variation and are structured [BD00].

When large data sets are aggregated into smaller more manageable data sizes we need more complex data tables called **symbolic data tables**. The columns of such table are called *variables* and they are used in order to describe a set of units called *individuals*. Rows are called **symbolic descriptions** of these individuals because they are not vectors of single quantitative or categorical values. Actually, each cell of a symbolic data table contains either distributions, intervals, or values linked by a taxonomy and logical rules [Did03]. For instance, in a table that stores book prices from different bookstores, the price of a given book *A* could be represented as an interval $[15, 25]$, which means that the price of book *A* varies from \$15 to \$25 in different bookstores. In our work, the test cases and the testers are the individuals and the variables associated to these individuals are stored using histogram-valued symbolic data. For example, $\text{Tester03}_{\text{Category}} = \{(feature\ testing, 25\%), (feature\ interaction, 40\%), (performance, 35\%)\}$ denotes a symbolic description to the Tester03, with respect to the variable *category*, which reads:

Tester03 has a weight of 25% in “feature testing”, 40% in “feature interaction”, and 35% in “performance” with respect to the category attribute.

2.3.2 Recommender Systems Supported by Symbolic Data Analysis

The recommender systems presented in this work follow the content-based approach and they use symbolic data to model both the test cases and the tester profiles. We adopted the techniques proposed by Bezerra and Carvalho [BdC10] as their model of *item* and *user* captures very closely our test cases and testers, respectively. The advantage of using SDA techniques and structures is that the tester description synthesises the entire body of information taken from the test case descriptions related to the tester profile. Then, as both testers and test cases are described by histogram-valued symbolic data, one can compare tester interests and test cases through a dissimilarity function [BdC10]. An example of how SDA works for the testing domain is given in Section 3.2.

The following steps correspond to the recommendation process described by [BdC10] when using the content-based approach and symbolic data analysis:

1. Construction of the symbolic description of the user profile.
2. Comparison between the user profile and an item to be recommended according to a dissimilarity function.
3. Generation of a ranked list of items according to their comparison with the user profile.

How we apply these steps to build our similarity ranking is described in detail in Chapter 3.

2.4 Statistical Tests

In our experiment we used the following statistical tests:

- **Analysis of Variance (ANOVA):** ANOVA is a method of testing the equality of three or more population means by analysing sample variances [Tri09]. The method we use is called **one-way analysis of variance** because we use a single treatment (or property) to categorise the populations. The ANOVA is classified as a *parametric method* because it is based on sampling from a population with specific parameters, such as the mean μ , or the standard deviation σ . Parametric methods usually must conform to some fairly strict conditions, such as a requirement that the sample data come from a normally distributed population.

Before applying the ANOVA we need to make sure that the following assumptions are respected:

1. The populations have distributions that are approximately normal. If the population does have a distribution that is far from normal, it is recommended to use the Kruskal-Wallis test.
2. The populations have the same variance σ^2 (or standard deviation σ).

3. The samples are simple random samples. (That is, samples of the same size have the same probability of being selected.)
4. The samples are independent of each other. (The samples are not matched or paired in any way.)
5. The different samples are from populations that are categorised in only one way.

Triola suggests that the first step to interpret the ANOVA results is understanding that a small p -value (such as 0.05 or less) leads to rejection of the null hypothesis of equal means while a large p -value (such as greater than 0.05), fails to reject the null hypothesis of equal means [Tri09].

- **Kruskal-Wallis:** it is a *nonparametric test* that uses ranks of sample data from three or more independent populations [Tri09]. Differently from the parametric methods, the nonparametric tests do not require assumptions about the population distributions. The Kruskal-Wallis test is used to test the null hypothesis that the independent samples come from populations with the same distribution; the alternative hypothesis is the claim that the population distributions are different in some way:

H_0 : The samples come from populations with the same distribution.

H_1 : The samples come from populations with different distributions.

Before applying the Kruskal-Wallis test the following assumptions need to be observed:

1. We have at least three independent samples, all of which are randomly selected.
 2. Each sample has at least five observations.
 3. There is *no* requirement that the populations have a normal distribution or any other particular distribution.
- **Tukey's test:** the Tukey's test, also known as Tukey-Kramer method or Tukey's HSD (Honestly Significant Difference) test, is a single-step multiple comparison procedure and statistical test generally used in conjunction with an ANOVA to find which means are significantly different from one another. The test compares the means of every treatment to the means of every other treatment; that is, it applies simultaneously to the set of all pairwise comparisons and identifies where the difference between two means is greater than the standard error would be expected to allow [Nat10]. The assumptions of Tukey's test are:
 1. The samples are independent of each other. (The samples are not matched or paired in any way.)
 2. There is equal variation across observations.
 - **Anderson-Darling:** the Anderson-Darling test is used to test if a sample of data came from a population with a specific distribution. It makes use of the specific distribution in calculating critical values. This has the advantage of allowing a more sensitive test

and the disadvantage that critical values must be calculated for each distribution [Nat10]. When applied to testing if a normal distribution adequately describes a set of data, it is one of the most powerful statistical tools for detecting most departures from normality [Ste86].

2.5 Controlled Experiments

Controlled experiments offer several specific benefits. They allow us to conduct well-defined, focused studies, with the potential for statistically significant results. They allow us to focus on specific variables, measures, and the relationships between them. They help us formulate hypotheses by forcing us to clearly state the question being studied and allow us to maximise the number of questions being asked. Such studies usually result in well defined dependent and independent variables and well-defined hypotheses. They result in the identification of key variables and good proxies for those variables. They allow us to measure the relationships among variables [Bas07].

A controlled study provides good insights into why relationships and results do and do not occur. It forces us to analyse the threats to validity, leading to insights in the identification of where replications or alternate studies are needed and where variations might show different effects.

2.6 Concluding Remarks

In this chapter we presented a brief introduction to software testing and we described the different test levels, test types, and test design techniques. We explained how the test allocation process works at TTC and we presented some terms and concepts that are part of TTC's routine. We also introduced some concepts related to recommender systems, statistical tests and controlled experiments.

White-box testing is not the focus of this work as TTC only performs black-box testing in the levels of component integration test and system test. Structural testing is the only test type that is not performed on TTC: functional testing, non-functional testing, and testing related to changes (re-testing and regression testing) are part of the TTC's routine.

The recommender systems presented in this work follow the content-based approach (collaborative filtering is not in the scope of this work) and they use symbolic data to model both the test cases and the tester profiles. We adopted the techniques proposed by Bezerra and Carvalho [BdC10] as their model of *item* and *user* captures very closely our test cases and testers, respectively.

In the next chapter we explain in more details how recommender systems can be applied in the software testing domain.

Recommender Systems and Testing

In this chapter we present how recommender systems can be applied to the testing domain. We start by presenting the tester profiles and the assignment ways we are proposing in this work. Then, we describe how we model test cases and tester profiles using symbolic data analysis techniques. Finally, we illustrate how the tester profiles are built and we describe how we calculate the similarity between a test case and a tester.

3.1 Recommender Systems Applied to the Testing Domain

Recommender systems [Kon04] are information filtering systems that recommend *items* (books, music, movies, news, images, web pages, etc) that are likely to be of interest to a customer. Since the 1990s these systems have been heavily used by sites like Amazon.com, Yahoo!Music and Netflix. For instance, Amazon.com recommends books to customers based on their browsing and shopping history. In the case of Amazon.com, the customer *profile* is built based on the customer's browsing and shopping history. In our case, the tester profile is built based on the tester's test execution history. An item for Amazon.com is a book. An item in our context is a test case. We represent a recommendation as a pair (test case, tester).

When comparing the test case characteristics with the tester profile, those characteristics may come from the test case itself (the content-based approach [FP09]) or from the tester's social environment (the collaborative filtering approach [HKTR04]). In our work, we use the content-based approach and thus our recommendations are made by comparing a particular test case with the tester profile and by evaluating their similarity. Alternatively we could compare the tester profile with other testers before recommending test cases. This would be specially useful while making recommendations for newcomers testers as their profiles does not have much information yet. We intend to evaluate this approach as part of our future work.

In order to build the tester profile, the recommender systems need to collect some data from the tester. This process of data collection can be made either explicitly (e.g. by asking a tester to rate a test case or by asking testers to create a list of test cases that they like) or implicitly (e.g. by observing the test cases that a tester has executed and the outcome of this execution) [CBLW01, RP06]. Whichever data collection approach is used (explicit or implicit), the more data collected from the tester, the better the recommendations. In our approach the tester profile is built from implicitly collected data: the list of test cases ran in the past and the list of *valid* defects raised/referenced by a tester. A valid defect is one that can be reproducible by a developer and is later fixed. The concept of valid defect is introduced in Section 2.1.7.4.

We defined 6 allocation systems for assigning test cases to testers. They vary in the tester

profile we use and in the algorithm that assigns a test case to a tester.

We propose 3 profiles for a tester:

- The *expertise* profile takes into account which test cases have been executed by a given tester in the past. With this profile we try to infer whether or not a tester has knowledge and experience in running a given test case. Recall that our context is that of manual testing and the correct assignment of test cases to experienced testers may result in reduced execution time. In order to have a flavour of a typical manual test case, see Figure 3.1 on page 25.
- The *effectiveness* profile compiles all test cases in which the tester has found a valid defect. The idea behind this profile is to try to infer which tester would have more chances to reveal valid defects for a given test case. In principle, defect finding should not depend on who is executing which test case (at least, this is what happens to automatic testing). However, in the context of manual testing, where the test case might be ambiguous, having found a valid defect with a particular test case in the past gives the tester some valuable experience for running similar test cases in the present.
- The *multi-objective* profile (*MO* for short) combines *expertise* and *effectiveness* by assigning a different weight to each. Note that *expertise* and *effectiveness* can be seen as two extremes of the same spectrum, which can be fine tuned through the *MO* profile (the manager may decide to use 60% on *expertise* and 40% on *effectiveness*, for example). In our experiment, reported in Chapter 5, we used our recommender systems with 50% weight to *expertise* and 50% weight to *effectiveness* as an attempt to capture the manager's intentions while allocating test cases to testers: test execution should be fast (*expertise*) and effective (*effectiveness*).

We defined 2 different ways of assigning test cases to testers. Suppose we have to execute 50 test cases. And suppose that we have 5 testers available. It is often the case that the test manager does not distribute test cases to testers uniformly, i.e. it is often the case that the test manager will *not* allocate 10 test cases to each tester. Some testers are more experienced than others, some are busier (already allocated to another activity) than others, some are faster than others etc. Therefore, the distribution of test cases among testers is usually *not* uniform. One variation of our allocation algorithm takes as input (from the test manager) the amount of test cases each tester should execute (the tester's workload). We call this algorithm *Manager-based*. Alternatively, the recommender system could assign freely test cases to testers without taking into account the manager's distribution. We call this assignment *Blind*. Notice that the *Blind* assignment also does not distribute test cases to testers evenly. It depends on the similarity between a test case and a tester (similarity is explained in Section 3.3).

All possible combinations of profiles and assignment algorithms are shown in Table 3.1. The 6 allocation systems are named after their profile and assignment algorithms: Exp-Manager, Eff-Manager, MO-Manager, Exp-Blind, Eff-Blind and MO-Blind.

All allocation systems shown in Table 3.1 share the same kernel for building the profile and checking similarity between a test case and a tester (details are described in sections 3.2

Table 3.1 The allocation systems.

Allocation System	Profile	Assignment
Exp-Manager	<i>Expertise</i>	Manager-based
Eff-Manager	<i>Effectiveness</i>	Manager-based
MO-Manager	<i>MO</i>	Manager-based
Exp-Blind	<i>Expertise</i>	Blind
Eff-Blind	<i>Effectiveness</i>	Blind
MO-Blind	<i>MO</i>	Blind

and 3.3). In this work we adopt the recommender system proposed by Bezerra and Carvalho [BdC10] based on Symbolic Data Analysis (SDA). In the next section we give an example of how SDA works for the testing domain.

3.2 Test Cases and Tester Profiles

This section describes how we model test cases and tester profiles. We start by introducing the test case structure and the terms/concepts used in the TTC through simple examples.

A test case is basically composed by an *id* (or *name*), a *description* (a short description that highlights the test case objective), and a sequence of *steps* and *expected results* (ideally, all the steps should have at least one expected result associated with it). Additional fields may include, but are not limited to: feature ID, component, test case category, test case level, test case type, preconditions, related requirements, cleanup, etc. Figure 3.1 shows a test case written in Testlink [Tes09] that illustrates very closely the structure of a test case used at TTC. The test case of Figure 3.1 verifies if it is possible to send a text message to an email address. This test case is the same test case shown in Section 2.1.4. We show it again in order to describe details relevant to the construction of the tester profile.

Figure 3.1 shows that the Test Case 25: tests the feature “SMS to e-mail”; is associated to the components “e-mail” and “SMS”; and belongs to the “feature testing” category. There are other test case attributes but, to illustrate our approach in the context of TTC, we will use only *feature*, *component* and *category*. It is important to emphasise that our technique can be customised to fit any other company’s attributes.

Table 3.2 presents how we abstract the test case TC25 in terms of its features, components, categories, and description.

Table 3.2 Content description of the test case TC25.

Attribute	Value
Feature	<i>2801</i>
Component	<i>e-mail, SMS</i>
Category	<i>feature testing</i>
Description	<i>“Ability to send SMS to e-mail address”</i>
...	...

TEST CASE TC25		
Summary		
Ability to send SMS to e-mail address.		
#	Step actions	Expected Results
1	Launch text messaging application	Text messaging application is properly launched
2	Enter a valid e-mail address in the 'recipients' field, write some text in the body of the SMS and send it	User is notified that the message was successfully sent
3	From PC, login in the e-mail account used in the previous step	Verify that the SMS is properly received in the e-mail address
Keywords:		
Category: feature testing		
Component: e-mail		
Component: SMS		
Feature ID: 2801 - SMS to e-mail		

Figure 3.1 Example of a test case.

Each test case is represented by attributes whose values have a particular *weight*. In the case of a textual variable, such as the *description*, it is pre-processed by the TF-IDF method. Table 3.3 shows how the weights are calculated for the test case TC25. Each attribute has a total weight equal to 1 and each value of a non-textual attribute has its weight evenly distributed among their values. For example, components *e-mail* and *SMS* have, each one, $\frac{1}{2}$ weight. In the case of the textual variable *description*, the weight of each word computed from the textual value comes from the TF-IDF. Before applying the TF-IDF technique, we perform the following *pre-processing* in the test case descriptions:

1. *Extract terms*. Divide the test case description into terms (words) and remove punctuation as well as non alphanumeric characters.
2. *Terms normalisation*. Normalise each word to make it lowercase.
3. *Stop words removal*. The objective of this step is to remove very common words which are not significant to characterise a given document. We use a list that contains all *stop words* that must be excluded. This is the reason why words such as *the*, *is*, *at*, *of*, and *on*, are not considered when calculating the TF-IDF for the test case description.

Table 3.3 Test case TC25 with attributes, values and weights.

Attribute	Values and Weights
Feature	(2801, 1)
Component	(e-mail, 1/2), (SMS, 1/2)
Category	("feature testing", 1)
Description	("ability", 0.238), ("send", 0.166), ("sms", 0.143), ("e-mail", 0.233), ("address", 0.220)
...	...

As an additional step, we could apply a *stemming* algorithm in the remaining terms after step 3. Stemming is the process of reducing inflected (or sometimes derived) words to their stem, base or root form [KMM00]. For example, after applying a stemming algorithm for the words “fishing”, “fished”, “fish”, and “fisher”, all of them would be reduced for “fish”.

Our recommender systems store all test cases in a structure similar to that of test case TC25 (see Table 3.4). For simplicity, from now on, we use values like F1, F2, ... for *features*, CMP1, CMP2, ... for *components*, CTG1, CTG2, ... for *categories*, and W1, W2, ... for words present in the test case *description*.

Table 3.4 Test cases and their attributes.

Test Case	Feature	Component	Category	Description
TC01	(F1, 1/2), (F5, 1/2)	(CMP1, 1/3), (CMP3, 1/3), (CMP4, 1/3)	(CTG2, 1)	(W2, 0.245), (W4, 0.236), (W5, 0.419)
TC02	(F2, 1/3), (F4, 1/3), (F8, 1/3)	(CMP3, 1/2), (CMP5, 1/2)	(CTG4, 1/2), (CTG6, 1/2)	(W3, 0.425), (W7, 0.575)
TC03	(F3, 1/2), (F5, 1/2)	(CMP4, 1)	(CTG1, 1)	(W1, 0.197), (W6, 0.333), (W8, 0.470)
TC04	(F3, 1/3), (F6, 1/3), (F7, 1/3)	(CMP2, 1/2), (CMP5, 1/2)	(CTG4, 1)	(W1, 0.531), (W3, 0.243), (W8, 0.226)
TC05	(F3, 1/2), (F7, 1/2)	(CMP2, 1/3), (CMP5, 1/3), (CMP7, 1/3)	(CTG1, 1/2), (CTG5, 1/2)	(W1, 0.631), (W9, 0.369)
...

Now that we have defined how test cases are modelled, we describe how we build the tester profiles. As mentioned in Section 3, we work with three profiles: *expertise*, *effectiveness*, and *multi-objective*.

3.2.1 The Expertise Profile

The expertise profile captures how much experience a tester has with respect to a set of features, components, categories, and descriptions. Based on this profile we can calculate how similar a given test case is in comparison to a given tester. The process of building the expertise profile is achieved by performing the following steps:

1. Access the test case repository to get the list of all test cases ran by a given tester.
2. Create symbolic descriptions for all test cases collected on step 1.
3. Build the tester profile by consolidating the weights of each *feature*, *component*, *category*, and *description* present in the test cases ran by that tester. Figure 3.2 illustrates this process.

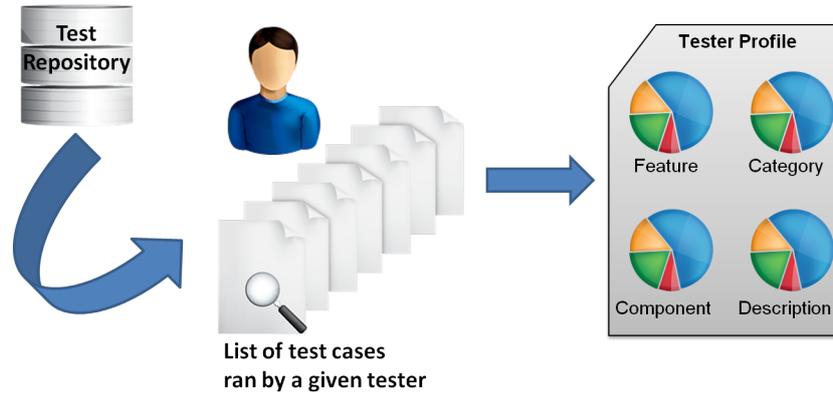


Figure 3.2 Building the *expertise* profile.

In what follows we describe in more details how we construct the profile of a given tester. Table 3.5 shows who has run each test case. We illustrate the construction of the profile of Tester03. We can see that Tester03 has run test cases TC02, TC03 and TC04 in the past.

Table 3.5 Execution history.

	TC01	TC02	TC03	TC04	TC05
Tester01	X		X	X	
Tester02	X	X			X
Tester03		X	X	X	
Tester04	X		X	X	X
Tester05		X		X	X
...

Table 3.6 shows the Tester03's *expertise* profile, which was built taking into account the execution history displayed in Table 3.5 and the test cases displayed in Table 3.4. As Tester03 executed test cases TC02, TC03 and TC04, we calculate the weights of each *feature*, *component*, *category*, and *description* associated with those test cases. We have to calculate, for example, the weights for the features F2, F3, F4, F5, F6, F7 and F8, as they are present in test cases TC02, TC03 and TC04 (see Table 3.4). The weight of a given feature for a tester is the average of the weights of all features present in the test cases performed by that tester. For example, the weights for F3 are 0 (its weight in TC02), $\frac{1}{2}$ (its weight in TC03) and $\frac{1}{3}$ (its weight in TC04). Therefore, the weight of feature F3 for Tester03 is the average of those weights:

$$\frac{1}{3} \cdot \left(0 + \frac{1}{2} + \frac{1}{3} \right) = \frac{1}{3} \cdot \frac{5}{6} = \frac{5}{18}.$$

The weight of the remainder features, components, categories, and descriptions are calculated in the same way.

Table 3.6 Tester03's expertise profile.

Tester	Feature	Component	Category	Description
...
Tester03	(F2, 1/9), (F3, 5/18), (F4, 1/9), (F5, 1/6), (F6, 1/9) (F7, 1/9) (F8, 1/9)	(CMP2, 1/6) (CMP3, 1/6), (CMP4, 1/3), (CMP5, 1/3)	(CTG1, 1/3), (CTG4, 1/2), (CTG6, 1/6)	(W1, 0.243), (W3, 0.223), (W6, 0.111), (W7, 0.191), (W8, 0.232)
...

3.2.2 The Effectiveness Profile

The steps for creating the *effectiveness* profile are very similar to the ones required to build the expertise profile. The main difference is that, in addition to accessing the test case repository to get the execution history information, we need to access the defect management systems to get the information related to the defects raised/referenced by a given tester. The steps for constructing this profile are described below:

1. Access the test case repository to get the list of all test cases in which the tester has reported a defect. Note that, for the effectiveness profile, we are concerned only about the “*failed*” test cases.
2. Connect to the defect management systems to get the final resolution of all defects raised/referenced by a given tester in the test cases collected in the previous step. Only valid defects are considered when creating the tester profile (defects that were terminated as “*wontfix*” or “*work as designed*”, for example, are not considered).
3. Create symbolic descriptions for the test cases in which the tester has raised/referenced a valid defect.
4. Build the tester profile by calculating the weights of each *feature*, *component*, *category*, and *description* present in the test cases arising from the previous step. The profile creation process is illustrated in Figure 3.3.

After performing the step 1, we have a table that associates each test case with the respective defect report ID raised/referenced by a given tester. Table 3.7 illustrates the outcome of this step. As we can see, this table is very similar to Table 3.5, except that crosses are replaced by defect report IDs.

Table 3.8 contains the final status and resolution of all defects raised/referenced by a given tester, which illustrates an outcome of step 2.

Now that we have tables 3.7 and 3.8, we are ready to build the effectiveness profile. As Tester03 found valid defects while running the test cases TC02 and TC04, we calculate the weights of each feature, component, category, and description associated with those test cases.

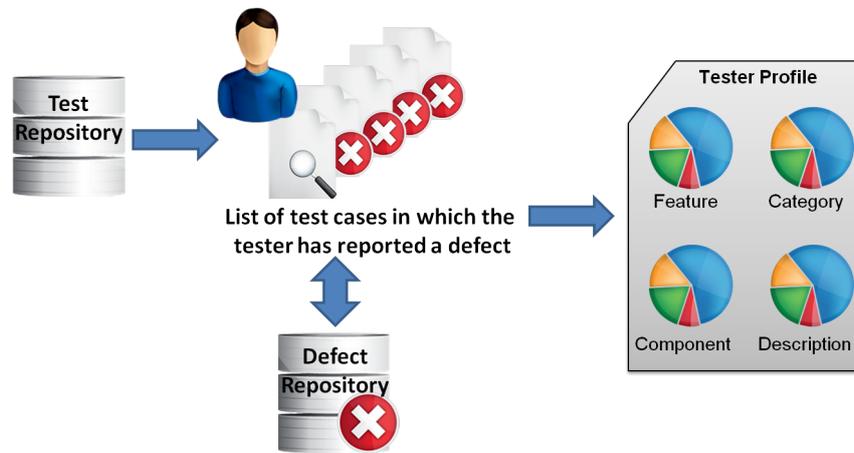


Figure 3.3 Building the *effectiveness* profile.

Table 3.7 Defect Report IDs per tester.

	TC01	TC02	TC03	TC04	TC05
Tester01	BUG1403			BUG1987	
Tester02					BUG0904
Tester03		BUG2801	BUG2011	BUG1985	
Tester04	BUG1234				
Tester05				BUG1412	BUG2005
...

We have to calculate, for example, the weights for the components CMP2, CMP3, and CMP5 (see Table 3.4). The calculation of the weights is performed in the same way described above for the *expertise* profile. The resulting effectiveness profile for Tester03 is displayed in Table 3.9.

3.2.3 The Multi-objective Profile

As mentioned earlier, the *multi-objective* profile is built by combining the expertise and effectiveness profiles by assigning a weight to each. The weight is assigned only after calculating the similarity between a tester and a test case (the way we calculate the similarity is explained in Section 3.3). The similarity between a tester and a test case is a number between 0 and 1: a similarity = 1 means that the tester and the test case have profiles that match perfectly.

For instance, let us assume the manager assigns weight 60 to the expertise profile and weight 40 to the effectiveness profile. This means that expertise is more relevant than effectiveness. If the similarity between a tester and a test case is 0.65 for the expertise profile and 0.73 for the effectiveness profile, then the similarity between them for the *multi-objective* profile is the weighted mean of the two: $\frac{(0.65 \cdot 60 + 0.73 \cdot 40)}{(60 + 40)} = 0.68$. In this work, we assigned equal weights of 50 to both profiles.

Table 3.8 Defect Reports - Status and Resolution

Tester	Defect Report ID	Status	Resolution
Tester01	BUG1403	Verified	Invalid
Tester01	BUG1987	Resolved	Fixed
Tester02	BUG0904	Resolved	Duplicate
Tester03	BUG2801	Resolved	Fixed
Tester03	BUG2011	Verified	Invalid
Tester03	BUG1985	Verified	Fixed
Tester04	BUG1234	Resolved	Invalid
Tester05	BUG1412	Verified	Wontfix
Tester05	BUG2005	Verified	Fixed

Table 3.9 Tester03's effectiveness profile.

Tester	Feature	Component	Category	Description
...
Tester03	(F2, 1/6), (F3, 1/6), (F4, 1/6), (F6, 1/6), (F7, 1/6), (F8, 1/6)	(CMP2, 1/4), (CMP3, 1/4), (CMP5, 1/2)	(CTG4, 3/4), (CTG6, 1/4)	(W1, 0.265), (W3, 0.334), (W7, 0.288), (W8, 0.113)
...

3.3 Calculating Similarity

This section describes how to calculate the similarity between a test case and a tester [BdC10]. The similarity degree is a number between 0 and 1. The closer to 1 the similarity is, the more advisable it is to allocate a given test case to a tester. Let us calculate the similarity between the test case TC05 (Table 3.4 on page 26) and the Tester03 (Table 3.6 on page 28) with respect to the expertise profile.

Let $C = \{CMP2, CMP5\}$ be the set of common components between TC05 and Tester03. Let $TC = \{CMP7\}$ and $T = \{CMP3, CMP4\}$ be the sets of components that belong exclusively to TC05 and Tester03, respectively (see Figure 3.4).

We will calculate four sums of weights: α , β , γ and δ . The sum α is the sum of weights from TC05 for the elements in C and the sum β is the sum of weights from Tester03 for the elements in C . Note that α and β are common components that capture the *agreements* between the Tester03's profile and the test case TC05. As CMP2 and CMP5 have 1/3 weight each for TC05, α is the sum of these values. For Tester03, the weights for CMP2 and CMP5 are 1/6 and 1/3, respectively. Then, β is the sum of 1/6 + 1/3.

$$\alpha = \frac{1}{3} + \frac{1}{3} = \frac{2}{3} \quad \beta = \frac{1}{6} + \frac{1}{3} = \frac{1+2}{6} = \frac{1}{2}$$

The sum γ is the sum of weights from Tester03 for the elements in T and the sum δ is

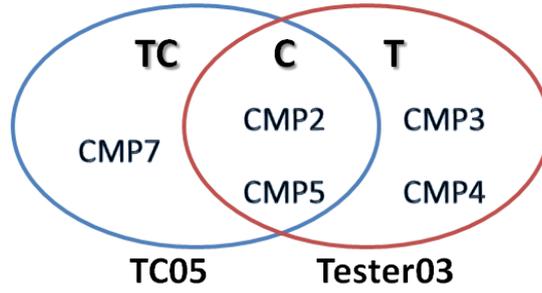


Figure 3.4 Components for TC05 and Tester03.

the sum of weights from TC05 for the elements in TC . As CMP3 and CMP4 are exclusive components of Tester03 profile (TC05 does not assess those components), γ is the sum of these components' weights: $1/6 + 1/3$ (note that this is the same of complement of β : $1 - 1/2 = 1/2$). CMP7 is the only exclusive component of TC05, thus the value of δ is equal to CMP7 weight for TC05: $1/3$ (or the complement of α : $1 - 2/3 = 1/3$). The sums γ and δ capture the *disagreements* between the Tester03's profiles and the test case TC05.

$$\gamma = \frac{1}{6} + \frac{1}{3} = \frac{1+2}{6} = \frac{1}{2} \quad \delta = \frac{1}{3}$$

The *dissimilarity* with respect to components is calculated through the following equation:

$$\frac{1}{2} \cdot \left(\frac{\gamma + \delta}{\alpha + \gamma + \delta} + \frac{\gamma + \delta}{\beta + \gamma + \delta} \right) \quad (3.1)$$

The dissimilarity measures how much disagreement exists between the tester profile and a given test case. The term $\frac{\gamma + \delta}{\alpha + \gamma + \delta}$ captures the proportion of dissimilarity with respect to α while the term $\frac{\gamma + \delta}{\beta + \gamma + \delta}$ calculates the proportion of dissimilarity with respect to β . The dissimilarity equation is the average of these two measures.

In the case of test case TC05 and Tester03, with respect to the attribute *component*, we have:

$$\frac{1}{2} \cdot \left(\frac{\frac{1}{2} + \frac{1}{3}}{\frac{2}{3} + \frac{1}{2} + \frac{1}{3}} + \frac{\frac{1}{2} + \frac{1}{3}}{\frac{1}{2} + \frac{1}{2} + \frac{1}{3}} \right) \cong 0.59.$$

The dissimilarities for feature, category, and description are calculated in the same way. In this case, the results (omitted here for simplicity), are approximately 0.50 for feature, 0.74 for category, and 0.73 for description. In Table 3.10 we show the *partial dissimilarities*, the *total dissimilarity*, and the *similarity* between Tester03 and each test case displayed in Table 3.4.

The *total dissimilarity* is the average of the partial dissimilarities: $(0.59 + 0.50 + 0.74 + 0.73)/4 = 0.64$. The *similarity degree* is the complement of this value: $1 - 0.64 = 0.36$. Thus, the *similarity* between the Tester03 and the test case TC05 is 36%. The expertise profile of tester T03 shows highest similarity with test case TC04 with a 62% similarity.

In cases when there is no common elements between the tester profile and a given test case, then Equation 3.1 will return dissimilarity 1.0, which means similarity = 0 between the tester

Table 3.10 Calculating similarities for Tester03.

Test Case	Partial Dissimilarities				Total Dissimilarity	Similarity
	Feature	Component	Category	Description		
TC01	0.81	0.59	1.00	1.00	0.85	15%
TC02	0.53	0.42	0.29	0.48	0.43	57%
TC03	0.46	0.53	0.53	0.35	0.47	53%
TC04	0.42	0.42	0.42	0.27	0.38	62%
TC05	0.50	0.59	0.74	0.73	0.64	36%

and that attribute. In Table 3.10 this situation happened twice for test case TC01, attributes *category* and *description* (see values in boldface).

Too many common elements, on the other hand, tend to result in high similarity values. If we look at Tables 3.4 and 3.6, we can see that the Tester03 profile contains all the *words* available in the *description* of TC04 (W1, W3, and W8). This is why they achieved the lowest dissimilarity value (0.27) in Table 3.10 for Description.

The work by Bezerra and Carvalho [BdC10] describes in details all the equations used in our recommender systems.

3.4 Concluding Remarks

In this chapter we described how Recommender Systems can be used for assigning test cases to testers. We started by introducing some key terms and concepts related to the test cases and the defect reports at TTC, and then, we presented the 3 tester profiles proposed on this work: *expertise*, *effectiveness*, and *multi-objective*.

We explained how we get information from the test repository and from the defect management systems to build the tester profiles. We chose to use SDA techniques and structures to model our testers and test cases and we presented the main advantages of using such a model.

We demonstrated how we calculate the similarity between a tester and a test case. In short, the similarity is 1-dissimilarity, and the dissimilarity is calculated by the proportion of *agreements* and *disagreements* between the tester and the test case. The computation of dissimilarity was introduced by Bezerra and Carvalho [BdC10], whose work proposes a recommender system based on SDA. Bezerra and Carvalho did not, however, applied their recommender system to testing.

Having the similarities between the testers and the test cases of an allocation input is half-way to our approach as the assignment of test cases will depend on either the workload of the testers (Manager-based assignment) or the similarity degree itself (Blind assignment).

In the next chapter we will describe in more details how our allocation systems were developed and how they actually assign test cases to testers.

The Test Allocation Systems

In this chapter we describe how the test allocation systems assign test cases to testers as well as we discuss some implementation details. All allocation systems are implemented in a tool called “*Test Case Recommender*” (TCR).

4.1 The TCR Implementation and Architecture

Our allocation systems were developed in *Python* [Lut06] using the *Eclipse* Integrated Development Environment [Hol04] and *Pydev* (a third-party plug-in for Eclipse) [LMS10]. *Rapid development* and a *powerful standard library* were key characteristics for the choice of this programming language. Besides that, the ease of communication with the test case repository through SQL queries was also a deciding factor. Although we have only ran our recommender system in Windows, as the system is developed in Python, it is compatible with many operating systems like Mac OS X and Linux.

Figure 4.1 shows the TCR architecture: the tool receives as input the *cutoff* value, the *profile* and the *assignment algorithm* to be used. The cutoff is the minimum acceptable similarity percentage. Its usage is explained in details below.

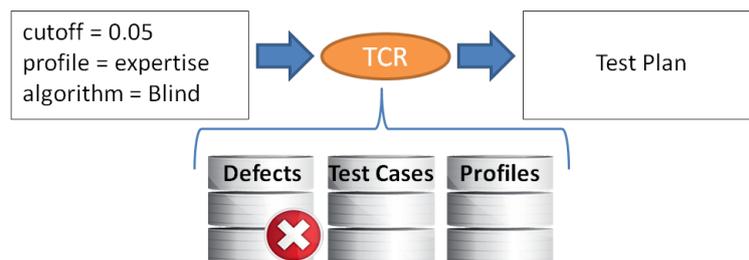


Figure 4.1 The TCR architecture.

Figure 4.2 shows a snapshot of *Test Case Recommender* main screens. The left screen shows how the manager enters the necessary data to create a new *test plan*. The manager provides the list of *test cases* to be allocated (1); and selects the *testers* that are available to participate in that test execution (2). Additionally, the manager may provide the maximum amount of test cases (workload) that can be allocated to each tester. At this point, the manager is implicitly choosing between the assignment algorithms *Manager-based* or *Blind*. If the manager leaves the workload field empty for all the testers selected, then the Blind algorithm is assumed. Otherwise, the Manager-based algorithm is used and the workloads defined are observed. The

next step is the profile selection and the manager can choose between *Expertise*, *Effectiveness*, and *Multi-objective* (3). If the *Multi-objective* profile is chosen, additional fields are displayed so the manager can define the weights to be used by each profile (*Expertise* and *Effectiveness*). Finally, the manager defines the *cutoff*. The cutoff defines a similarity percentage that makes the algorithms to ignore allocations whose similarity is below that value. In Figure 4.2, any pair (test case, tester) with similarity below 5% is not allocated. This prevents the algorithm to recommend pairs that have very low similarity. TCR is now ready to start allocating test cases to testers.

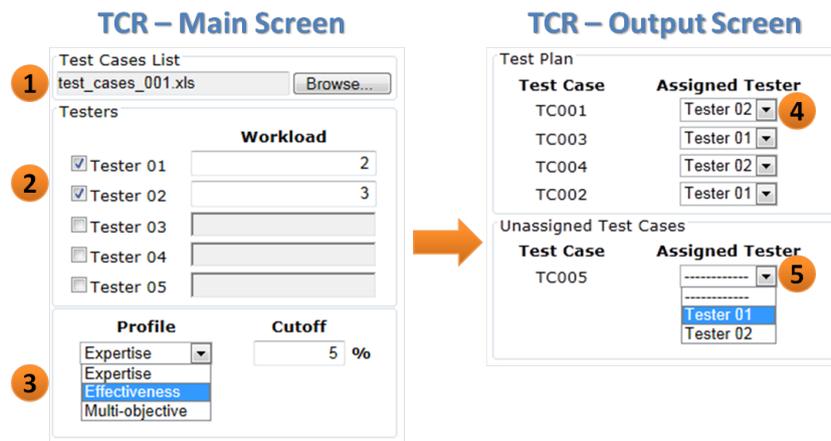


Figure 4.2 The TCR user interface.

The right side screen in Figure 4.2 illustrates an output from the TCR: a *test plan* containing the pairs (test case, tester) recommended by the algorithm. In this example, test cases TC001 and TC004 were assigned to Tester02 while the test cases TC003 and TC002 were assigned to Tester01. The pairs are ordered by the similarity rate (descending order). Notice that the manager has the possibility to change the tester, which was suggested by the algorithm, assigned to a given test case. If the manager judges that the tester suggested by the TCR is not suitable for a given test case, a different tester can be selected from the list of testers available that is displayed in the *combo box* (4). As a future work, we plan to watch the changes performed by the manager and feed the TCR with them so it can make the necessary adjustments in the tester profiles to avoid that “*inappropriate allocation*” in future test plans. The test cases that were not assigned to anyone due to either *low similarity* or *workload limitation* (in the case of Manager-based systems) are also displayed on this screen (5) so the manager can manually assign testers to run the remaining test cases.

4.2 The Assignment Algorithms

When the TCR is run for the first time, it builds up the testers profiles (tables 3.6 and 3.9 on pages 28 and 30, respectively) and the test case descriptions (Table 3.4 on page 26). As described in Chapter 3, this step is achieved by communicating with the test case repository

(in order to collect data from previous test executions) and the defect management systems (to collect information about the defects raised/referenced by the testers). Figure 4.3 illustrates this process. Those repositories allow us to create both the expertise and the effectiveness profiles for each tester. The multi-objective profile is built later on by combining the expertise and effectiveness profiles. In the multi-objective profile, the similarity value between a tester and a test case is the weighted mean of the similarities, calculated according to the weights defined by the manager, for the expertise and effectiveness profiles. This is why this profile is not built at this first moment.

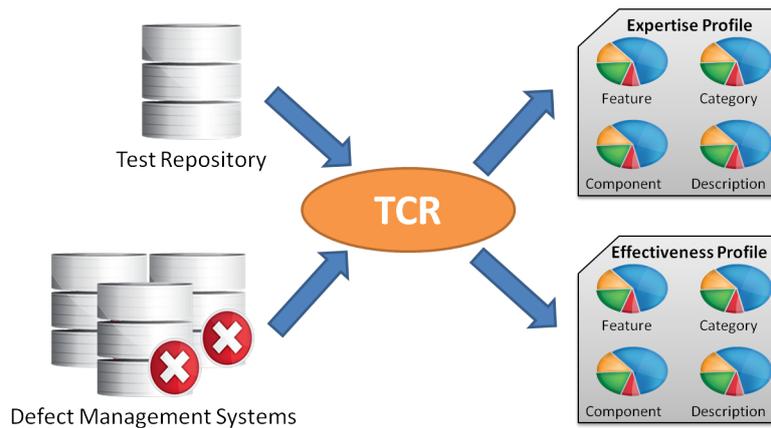


Figure 4.3 TCR creating the tester profiles and the test case descriptions.

The expertise and the effectiveness profiles are generated from scratch in the first run (at the installation of the TCR). This process takes 3 minutes for the particular configuration of TTC (details are explained in Section 4.3 below). Subsequent runs update the profiles instead of rebuilding them from scratch. The update takes no more than 30 seconds.

Having the initial setup done, the TCR is ready to assign test cases. For the next step, it takes as input (from the test manager) the *allocation input*: a list of test cases to be executed and a list of testers available for that execution. Then, the test manager selects the profile (*Expertise*, *Effectiveness* or *Multi-objective*) to be used in that test allocation. With this information, the TCR calculates the relevance (through the similarity degree) of each test case for all the testers available (see Figure 4.4).

An example of the outcome of this step is shown in Table 4.1. We give as allocation input to the algorithms 5 test cases (TC01, TC02, TC03, TC04 and TC05) and 2 testers (Tester01 and Tester02). The algorithm computes the similarity of all pairs (test case, tester) and orders them by similarity. Our algorithms discard from this list all pairs (test case, tester) whose similarity is smaller than 5%. Therefore, the pairs (TC02, Tester02), (TC04, Tester01) and (TC05, Tester02) are not taken into account during the production of the recommendation list.

The last step is the test allocation itself. The way this step is performed depends on the assignment algorithm (*Manager-based* or *Blind*) selected by the manager. In the next sections we explain in details how the assignment algorithms work.

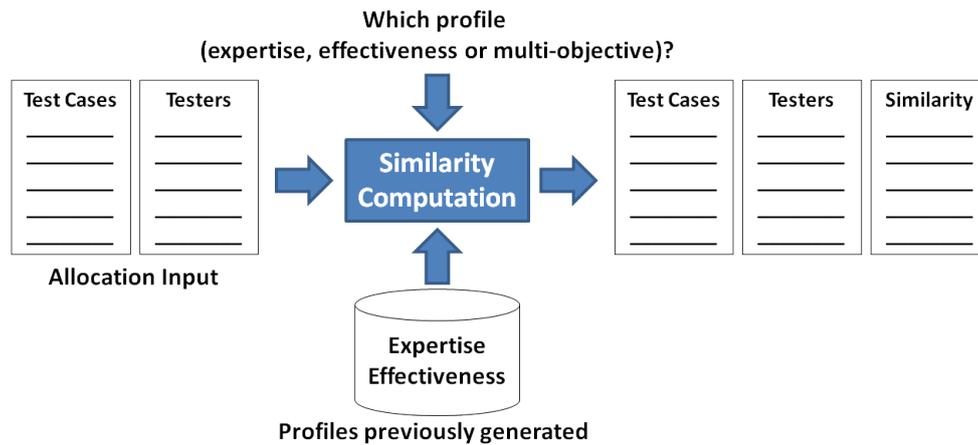


Figure 4.4 Similarity computation.

Table 4.1 Similarities.

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

4.2.1 The Manager-Based Assignment

Figure 4.5 shows the overall process for the construction of the recommendation when using the Manager-based assignment. The manager provides as input the workload of each tester: how many test cases each tester is supposed to run. Note that, in this case, the test case assignment does not depend on the similarity degree between a test case and a tester alone; it also depends on the workload defined by the manager for each tester. The algorithm takes the outcome of the previous step (Figure 4.4) and computes the recommendation as described in what follows.

Algorithm 1 shows how the Manager-based assignment work. The actual code in Python is obviously more elaborate than Algorithm 1, but for conciseness we describe here a simplified version. Algorithm 1 receives as input a list of the testers to be allocated, a list containing the maximum amount of test cases that can be assigned to each tester (*workload*), the *cutoff*, and a table containing the similarities calculated for each test case and each tester. The output of this algorithm is the *test plan* containing the list of all pairs (test case, tester) suggested by the algorithm. In this algorithm, an empty list is represented by `[]`, an element from the list *L* in

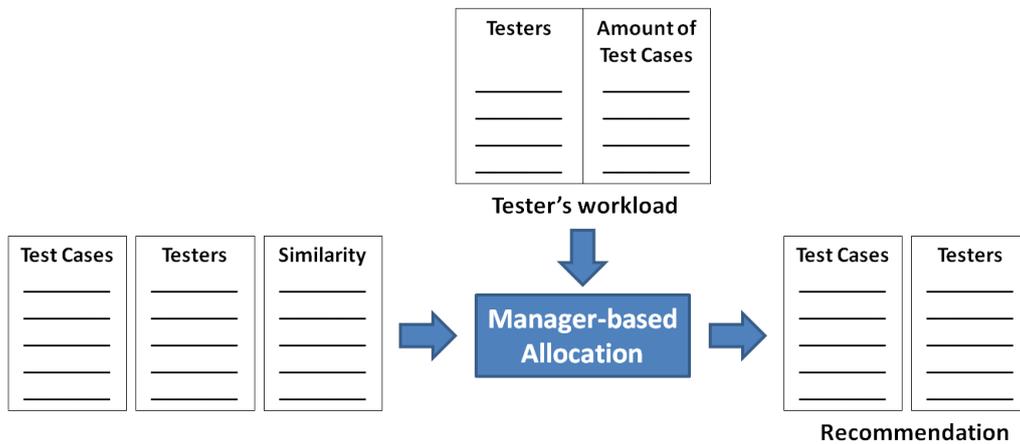


Figure 4.5 Manager-based recommendation.

the n -th position is $L[n]$, the assignment of a variable v by the expression e is $(v \leftarrow e)$, \wedge is the logical *AND* and \neg represents logical negation.

At first, the *testPlan* is empty and the counter i starts with 0. The variable i will iterate over the *testers*. The function *thereArePossibleAllocations()* verifies if it is possible to allocate test cases to testers, i.e., it verifies if there are testers whose workload have not been reached yet. If this is the case, it verifies if there are test cases not allocated to *testPlan* and are available for available testers. If these test cases and testers exist, and their similarity is greater than or equal to the *cutoff*, then the loop continues. Otherwise, the loop ends. The list *testers* is handled as a circular list. So, at each iteration, a different tester will have the opportunity to “receive” a test case. *aTester* receives the tester of the turn and *aTestCase* receives the test case that has the highest similarity with *aTester* and was not allocated yet. If the maximum workload of *aTester* has been achieved, then this tester is done and is removed from the allocation. Otherwise, if the similarity between *aTestCase* and *aTester* is above the *cutoff*, then we allocate *aTestCase* to *aTester*. We make sure this test is not allocated anymore by removing it. If the similarity is below the *cutoff*, then we remove the pair $(aTestCase, aTester)$ from the similarity table. The next tester takes the turn ($i++$) and the loop continues.

To illustrate Algorithm 1, let us assume the manager decides to allocate the amount of test cases to each tester (workload) according to Table 4.2. The test team always has to run all test cases of the allocation input. Therefore, it is usually the case that the manager will provide a workload whose sum is the number of test cases to be executed. This is not a restriction of our tool, though. If the manager decides to provide a workload whose sum is less than the total number of test cases to be executed, then some test cases will remain unassigned at the end of our tool execution.

The Manager-based algorithm tries to allocate one test case for each tester in each iteration over Table 4.1. Let us assume that *testers* is the list $[Tester02, Tester01]$. So, in the first iteration ($i = 0$), the test case with the highest similarity to Tester02 is allocated: in our case, test case TC01 is allocated to Tester02. As TC01 cannot be allocated anymore, it is removed from the similarity table. Table 4.3 shows the outcome of the first iteration. We show in boldface the

Algorithm 1: Manager-based algorithm

```

Input: testers          /* list of the testers to be allocated */
        workload        /* maximum amount of test cases per tester */
        cutoff           /* minimum acceptable similarity */
        similarities    /* similarities between test case and testers */
Output: testPlan      /* A list of pairs (test case, tester) */

testPlan  $\leftarrow$  []      /* testPlan starts with the empty list */
i  $\leftarrow$  0             /* iterator over testers */
while thereArePossibleAllocations(similarities, testPlan, workload, cutoff) do
  aTester  $\leftarrow$  testers[i mod length(testers)]
  aTestCase  $\leftarrow$  getHighestSimilarity(aTester, similarities)
  if workloadSoFar(aTester, testPlan) = maximumWorkload(aTester, workload) then
    | removeTester(aTester, testers)
  else
    | if getSimilarity((aTestCase, aTester), similarities)  $\geq$  cutoff then
      | | add((aTestCase, aTester), testPlan)
      | | remove(aTestCase, similarities)
    | else
      | | remove((aTestCase, aTester), similarities)
    | end
  end
  i ++
end

```

Table 4.2 Manager-based assignment: amount of test cases per tester.

Tester	Amount of Test Cases
Tester01	2
Tester02	3

pair that has been allocated, and we strike through those that have been removed.

In the second iteration ($i = 1$), it is the turn of Tester01. The test case with the highest similarity with this tester is TC03. Then, Tester01 receives TC03 and this test case is removed from the similarity table (see Table 4.4).

The loop continues to the third iteration ($i = 2$) and Tester02 has the turn again. TC04, which is the test case with the highest similarity to Tester02, is allocated and subsequently removed from similarity table (see Table 4.5).

In the fourth iteration ($i = 3$), Tester01 receives TC02. Notice that Tester01 is now allocated to run 2 test cases, which is the maximum workload defined for this tester. In the next time Tester01 receives the turn, the algorithm will remove this tester from the circular list *testers* (see Table 4.6).

Tester02 has the turn in the fifth iteration ($i = 4$) and the test case with the highest similarity with this tester is TC05. Although TC05 is still available (not allocated) and Tester02 has not reached the maximum workload yet (the current workload for Tester02 is 2, while the

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

Table 4.3 Manager-based: first iteration.

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

Table 4.4 Manager-based: second iteration.

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

Table 4.5 Manager-based: third iteration.

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

Table 4.6 Manager-based: fourth iteration.

maximum workload defined by the manager is 3), this pair is not added to the *testPlan* because the similarity between Tester02 and TC05 is 1.45%. As it is below the *cutoff*, the pair (TC05, Tester02) is removed from the similarity table (see Table 4.7).

In the sixth (and last) iteration ($i = 5$), Tester01 has the turn. However, as the maximum workload defined for this tester (2 test cases) has been already reached, Tester01 is removed from the circular list *testers* and the loop finishes as there are no more allocations to be done. Note that not all test cases are allocated: test case TC05 is not assigned to anyone.

In summary, the Manager-based algorithm iterates over the testers and recommend test cases to testers based on the following restrictions: (i) a tester is allocated at most once per iteration; (ii) previously allocated test cases are disregarded; (iii) the workload of each tester must not exceed the manager's input; and (iv) the similarity must be above a certain value (in our example, we used a 5% cutoff). The final recommendation can be seen at Table 4.8.

4.2.2 The Blind Assignment

For the Blind systems (*Exp-Blind*, *Eff-Blind* and *MO-Blind*), the workload of each tester is not given as input. Figure 4.6 depicts the overall process for the computation of Blind assignment.

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

Table 4.7 Manager-based: fifth iteration.

Test Case	Tester
TC01	Tester02
TC03	Tester01
TC04	Tester02
TC02	Tester01

Table 4.8 Manager-based: final recommendation.

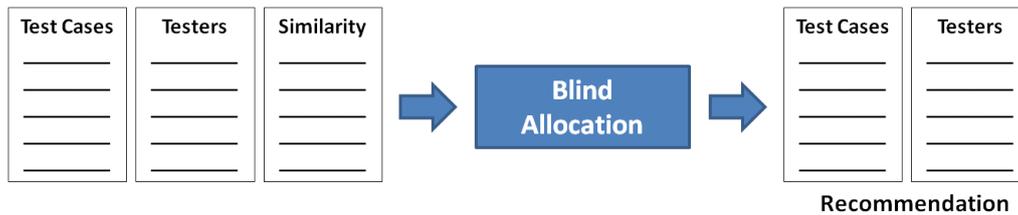


Figure 4.6 Blind recommendation.

Algorithm 2 shows how the Blind algorithm works. It receives the list of testers to be allocated, the *cutoff point*, and the similarity table. The output of this algorithm is the same as the output of Algorithm 1: a *test plan* containing the list of all pairs (test case, tester) allocated by the algorithm. The *test plan* starts as an empty list and the function *thereArePossibleAllocations()* does a similar work as the one in Algorithm 1: it verifies if it is still possible to allocate test cases to testers. The only difference is that no workload is taken into account. The function *getPairWithHighestSimilarity()* returns the pair (*aTestCase*, *aTester*) that has the highest similarity amongst all the possible pairs. If the similarity between the test case and the tester from the pair returned by *getPairWithHighestSimilarity()* is greater than or equal to the *cutoff*, then that pair (*aTestCase*, *aTester*) is added to the *test plan*. Analogously to Algorithm 1, once a test case is allocated to a given tester, all the instances of that test case are removed from the similarity table as the same test case is allocated to at most one tester.

To illustrate the Blind algorithm in action, for the similarities given in Table 4.9, the first allocation is for the pair (TC01, Tester02). TC01 is then removed from the similarity table as it cannot be assigned anymore. After that, the Blind algorithm allocates the pair (TC03, Tester02) and removes TC03 from the similarity table. Later, the pairs (TC04, Tester01), (TC02, Tester01) and (TC05, Tester01) are allocated. The final recommendation is shown in Table 4.9. Although in this example all test cases have been allocated, it is not the case in general. It is possible that, for a particular test case, its similarity is below the cutoff with respect to any tester. In this case, this test case is never allocated.

Algorithm 2: Blind algorithm

```

Input: testers          /* list of the testers to be allocated */
        cutoff           /* minimum acceptable similarity */
        similarities     /* similarities between test case and testers */
Output: testPlan      /* A list of pairs (test case, tester) */

testPlan ← []           /* testPlan starts with the empty list */
while thereArePossibleAllocations(similarities, testPlan, cutoff) do
  (aTestCase, aTester) ← getPairWithHighestSimilarity(similarities)
  if getSimilarity(aTestCase, aTester, similarities) ≥ cutoff then
    add(aTestCase, aTester, testPlan)
    remove(aTestCase, similarities)
  end
end

```

Test Case	Tester	Similarity
TC01	Tester02	65,78%
TC03	Tester02	62,75%
TC03	Tester01	59,65%
TC04	Tester02	57,19%
TC02	Tester01	55,17%
TC01	Tester01	33,63%
TC05	Tester01	13,46%
TC02	Tester02	2,44%
TC04	Tester01	1,45%
TC05	Tester02	1,45%

Table 4.9 The Blind algorithm allocation.

Test Case	Tester
TC01	Tester02
TC03	Tester02
TC04	Tester02
TC02	Tester01
TC05	Tester01

Table 4.10 Blind algorithm: final recommendation.

So, basically the Blind algorithm iterates over Table 4.9 and the restrictions to be met are: (i) the highest similarities are chosen; (ii) the same test case is allocated to at most one tester; and (iii) pairs whose similarities are below the cutoff value of 5% are discarded. The final recommendation of the Blind algorithm is shown in Table 4.10.

4.3 On Performance

The first step performed by the TCR is the construction of the testers profiles (Table 3.6 and Table 3.9 on pages 28 and 30, respectively) and the test case descriptions (Table 3.4 on page 26). This step takes about 3 minutes considering the particular configuration of TTC (a database with approximately 120,000 entries) running on an Intel T2050, 1.6GHz, 1GB of RAM. Notice that this step needs to run only in the first time we run the TCR (say, at installation time). In subsequent runs, the system simply updates the tester's profiles that was built from scratch in the first run. In this case, the time to update the profiles is negligible. The second and third step

generates Table 4.1 (on page 36), and Table 4.8 (on page 40) or Table 4.10, page 41, depending on the assignment algorithm chosen by the manager. This step takes no more than 30 seconds considering an allocation input with 300 test cases and 10 testers. Therefore, performance does not seem to be an issue even in an industrial context.

4.4 Concluding Remarks

In this chapter we presented the TCR architecture and discussed some of its implementation details. We described how the assignment algorithms work and we illustrated how they assign test cases to testers through simple examples.

Both Manager-based and Blind algorithms do *not* necessarily allocate all test cases. Recall that pairs (test case, tester) whose similarity are smaller than 5% are discarded. There are test cases that are poorly documented (attributes are missing) and therefore their similarity to all testers are below 5%. In addition to that, in the case of Manager-based systems, the workload for each tester may prevent some test cases to be allocated despite having a similarity above 5% (see the pair (TC05, T01) in the example of Section 4.2.1).

Manager-based and Blind assignment have their advantages and disadvantages. On the one hand, the Manager-based algorithm produces allocations which are consistent with the *tester's availability*. On the other hand, it requires the manager to know in advance, and provide as input, the workload of all testers to achieve such a result. Furthermore, as the similarity is not the only factor considered during the allocation process, the suggested allocation may not be the most appropriate one. The Blind algorithm, on its turn, may produce better allocations as it is based solely on the similarity between test case and testers. However, it can suggest workloads that are totally misaligned with the actual tester's availability.

In the next chapter we present a controlled experiment that was performed in order to evaluate the effectiveness of our allocation systems.

Recommender Algorithms Evaluation

This chapter presents a controlled experiment designed to evaluate the effectiveness of the allocation systems. We start by stating the problem investigated and defining our research objective. Then, we give some details about the experimental planning and its execution. Finally, we report the results of our experiment and address the threats to validity.

5.1 Introduction

The purpose of this section is to set the scope of our experiment. In the next sections we *state the problem* investigated in our experiment, we define our *research objective*, and we present the *context* in which the experiment was conducted.

5.1.1 Problem Statement

One important activity at TTC is the test case allocation. Typically, a test manager has to allocate hundreds of test cases to be executed by the testers available at that moment. In order to perform this task, the test manager takes between 15 to 35 minutes, depending on the amount of tests to be allocated. This task is executed routinely at TTC: around 10 times per week, which means that the time spent on this task per year varies from 16 to 30 working days. A bad allocation has also an impact on the execution time of the test suite and the effectiveness of it (fewer defects found).

By allocating a test to an experienced tester, which is someone who has executed that test (or similar tests) in the past, the execution time may reduce. Analogously, a tester who has found defects executing a particular test in the past is a good candidate to execute that same test again or other tests that are similar to that.

The introduction of the allocation systems presented in this work can help the managers to perform better allocations faster. To identify which allocation system achieve better results, we need to evaluate the performance of each one of them in a controlled experiment.

5.1.2 Research Objective

The research objective of this experiment is to:

Analyse the performance of the proposed allocation systems for the purpose of comparison with respect to the rate of unassigned test cases and the adequacy of the achieved test allocation from the viewpoint of test managers in the context of manual test execution at

TTC.

An *unassigned* test case is a test case that the allocation system did not allocate to any tester. This happens either because the similarities of this test case with all testers were smaller than the *cutoff point* (in our experiment, we used 5%) or because of the workload of the testers (in the case of Manager-based systems).

5.1.3 Context

The experiment was conducted in the context of TTC, a company that outsources testing services to a mobile phone manufacturer. TTC performs black-box testing and most of them are executed manually.

5.2 Experimental Planning

We designed an experiment to evaluate the effectiveness of our six allocation systems against of the manager's allocation and random allocations. We compared the performance of our systems with respect to the history of allocations done by the managers of TTC from November 2009 to September 2010, over a subset of 9 products (a product at TTC is a mobile phone model).

5.2.1 Goals

In order to achieve our research objective, we defined the following goals for this study, according to the Goal/Question/Metric method [BCR94]:

- G₁**: Evaluate the unassignment rate of each allocation system.
- G₂**: Evaluate and compare the effectiveness of each allocation system with respect to the following approaches: real manager's allocation and random allocation.

For assessing the goal **G₂**, the managers' allocations are considered to be the *golden-standard* for comparing the results obtained by our allocation systems. Comparing the performance of recommender algorithms to a human being is a well established practise in the Artificial Intelligence domain as such systems typically attempt to mimic the human being choices. Before we introduce the questions to be answered by this experiment, we define some concepts that are used in our metrics.

Let TC_S and TC_M be test cases, and T_S and T_M be testers. Let (TC_S, T_S) and (TC_M, T_M) be allocations recommended by our system and by the manager, respectively. Whenever our system recommends an allocation exactly as the manager did, we say that our system produced a *strictly correct pair*. In other words, whenever $TC_S = TC_M$ and $T_S = T_M$, (TC_S, T_S) is a strictly correct pair.

The concept of a strict correct pair can be relaxed. Instead of expecting our allocation systems to get answers exactly the same as the managers, we could instead define the concept of an *approximately correct pair*. Let us assume that the testers can be grouped according

to their skills. Suppose that G_1 is the group of the most talented testers, G_2 is the group of the second most talented testers, and so on. The pair (TC_S, T_S) is an approximately correct pair whenever $TC_S = TC_M$ and T_S belongs to the same group as T_M . These groups have been defined by two test managers from TTC (Section 5.2.4 describes how these groups were defined).

With the strict metrics, we take the risk of an approximately correct pair being regarded as incorrect. On the other hand, the approximate metrics assume that the groups of equivalent testers are the same for *all* allocation inputs. This assumption may not be always true, as the groups may change depending on the given allocation input (for example, the group of most talented testers may depend on the *component* being tested).

We intend to assess our allocation systems in terms of strictly correct pairs and approximately correct pairs. The assessment of the goals G_1 and G_2 is performed by answering the following questions.

Q_a [unassignment] Which allocation system leaves more test cases unassigned?

Q_b [strict precision] How many pairs (test case, tester) recommended by the allocation system are correct?

Q_c [strict recall] How many pairs (test case, tester) recommended by the manager are also recommended by the allocation system?

Q_d [approximate precision] How many pairs (test case, tester) recommended by the allocation system are approximately correct?

Q_e [approximate recall] How many pairs (test case, tester) recommended by the manager are also recommended (in an approximate way) by the allocation system?

Questions **Q_b** to **Q_e** are answered by well-established metrics from the Artificial Intelligence domain called *precision* and *recall*. The questions **Q_a** to **Q_e** are directly answered through the following metrics, respectively.

M_a [unassignment]

$$U = \frac{\# \text{ of unassigned test cases}}{\# \text{ of tests in the allocation input}} \quad (5.1)$$

M_b [strict precision]

$$P = \frac{\# \text{ of strictly correct pairs}}{\# \text{ of recommendations of the algorithm}} \quad (5.2)$$

M_c [strict recall]

$$R = \frac{\# \text{ of strictly correct pairs}}{\# \text{ of recommendations of the manager}} \quad (5.3)$$

M_d [approximate precision]

$$AP = \frac{\# \text{ of approximately correct pairs}}{\# \text{ of recommendations of the algorithm}} \quad (5.4)$$

M_e [approximate recall]

$$AR = \frac{\# \text{ of approximately correct pairs}}{\# \text{ of recommendations of the manager}} \quad (5.5)$$

The unassignment metric M_a computes the amount of unassigned test cases for each allocation system with respect to the total amount of test cases given in the allocation input. For instance, if the allocation system manages to allocate 75 test cases out of 100 test cases given in the allocation input, its unassignment metric is 25%.

Strict precision M_b computes the total amount of strictly correct pairs with respect to the total amount of recommendations made. If, for example, 75 pairs are recommended by the allocation system (i.e. given as *output*) and only 45 pairs are strictly correct, then its strict precision M_b is 60%.

The strict recall metric M_c computes the percentage of strictly correct pairs recommended by the allocation system with respect to all correct pairs, which are those pairs allocated by the manager. If, for instance, 100 test cases are allocated by the manager and, out of those recommended by the allocation system, only 45 test cases are strictly correct pairs, then $M_c = 45\%$. Note that the manager always allocates all test cases. So, out of 100 test cases given in the allocation input, the manager always allocates 100 test cases. The manager does not leave test cases unassigned, only the allocation systems do.

The approximate precision and approximate recall, M_d and M_e respectively, are analogous to the strict precision and the strict recall. The only difference is that, in an approximate metric, a correct pair takes into account a group (or class) of equivalent testers.

For unassignment, the lower the rate, the better. For the other metrics, the higher the percentage, the better.

5.2.2 Participants

We run our allocation systems using historical data. There were no participants on that phase of the experiment. However, in order to calculate the approximately correct pairs, we needed to know the groups or classes of equivalent testers. To produce these groups, we asked two TTC's test managers to define them in cooperation. They are both equally experienced and they each happen to know a significant subset of the testers involved in the past allocations. The fact that no one knew all the testers was not a problem, as their subsets intersect substantially, and the union of their subsets comprises all testers.

5.2.3 Experimental Material

The objects used in the experiment are the allocation inputs, the Test Case Recommender, and the manager's allocation.

- **Allocation Inputs.** Recall that an *allocation input* is a list of test cases and a list of testers to be allocated. Basically, an allocation input is the input of our allocation systems. The *test plan* is the outcome of the allocation: a list of pairs (test case, tester). In order to carry out our experiment, we selected 100 allocation inputs from the past that have been

subsequently transformed into test plans by TTC managers in the period from November 2009 to September 2010. The allocation inputs were chosen according to the following criteria:

- The allocation inputs should have at least 50 test cases to be allocated. We chose 50 test cases because we think that the allocation of 50 test cases (or more) is a problem hard enough for a human being to deal with by hand. The allocation input sizes varied from 50 to 585 test cases (our largest allocation input).
- They should have at least 2 testers available for each test execution. At TTC some test plans are performed by a single tester either because the purpose of the test plan requires that it should be run by a single tester following a predefined order to perform each test case or because the size of the allocation input is too small. In the case of a single tester available, it would make no sense to compare the automatic allocations with the manager’s allocations as they would have always the same pairs (test case, tester). The highest number of testers involved in our allocation inputs was 9.

The scatter plot displayed on Figure 5.1 shows the amount of test cases (horizontal axis) and the number of testers (vertical axis) of each allocation input. This figure captures the main features of the allocation inputs we used in our experiment.

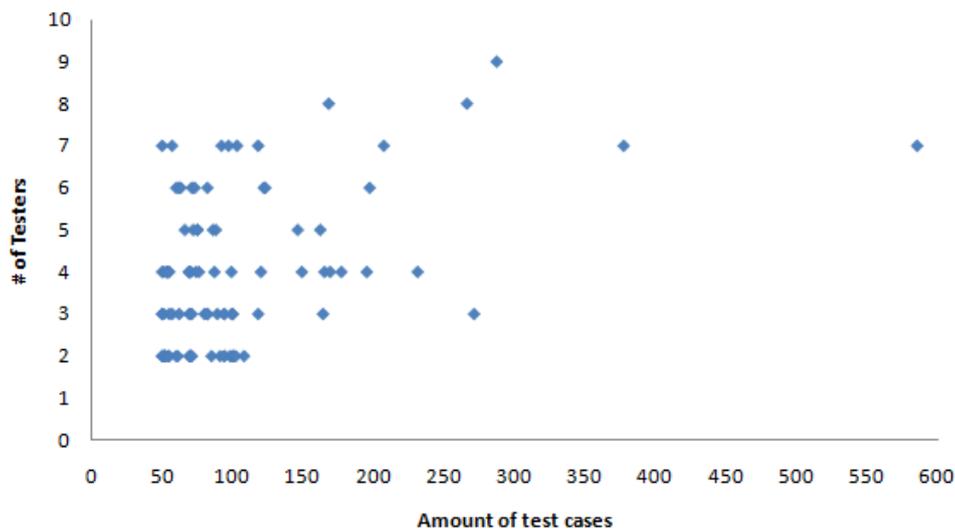


Figure 5.1 Amount of test cases and number of testers per allocation input.

The majority of the allocation inputs have between 50 and 200 test cases and 2 to 7 testers; just a few have more than 200 test cases and only one contains more than 500 test cases. Only 3 allocation inputs were performed by more than 7 testers.

- **Test Case Recommender.** We used the tool described in Chapter 4 to generate the automatic allocations based on our allocation systems as well as the random allocations.

- **Manager's allocations.** We collected the manager's allocation for each one of the 100 allocation inputs for further comparison with the automatic allocations generated by our allocation systems.

5.2.4 Tasks and Procedures

For each allocation input previously selected, we performed the following tasks:

- 1 We collected the allocation inputs from TTC repository from November 2009 to September 2010;
- 2 For each allocation input, we saved the date in which that allocation input was used: only data prior to that date were used in order to create the tester profiles. This way we guarantee that the allocation systems did not have more information than the manager had at that moment;
- 3 We used each one of the 6 allocation systems to generate automatic allocations;
- 4 For each automatic allocation made, we calculated the metrics M_a to M_e described earlier.

Just before executing task 4 above, we had to define the groups of equivalent testers. In order to define these equivalence groups we asked two test managers to classify the testers who were involved in executing all the allocation inputs used in our experiment. Separately, we gave the managers cards containing the testers' names (one card for each tester) and we asked them to perform the following procedure:

1. Put away the cards from the testers they would not be able to classify due to lack of knowledge of the skills of that particular tester.
2. Group the remaining cards by putting together testers with similar skills.
 - a. We emphasised that testers in the same group should achieve the same results in terms of efficiency (execution time) and effectiveness (defects detected).
 - b. We made it clear that people with unique profile (in a good or bad sense) should not be grouped. They should stay alone.
3. The managers then sorted the groups they were able to create in order of performance (in terms of efficiency and effectiveness). This way we were able to know which group was the best group, second best group, and so on.
4. Managers got together in order to solve any inconsistency over their groups. There was only one inconsistency easily solved by the managers.

5.2.5 Statistical hypotheses

In this section, we present the statistical hypotheses that we want to test in our experiment. We define the following null statistical hypotheses H_{0i} and their alternative hypotheses H_{1i} , where i is the index of their related questions (Q_i).

H_{0a} : All allocation systems have the same *unassignment* rate, that is, there is no statistically significant difference between their rates.

H_{1a} : There is at least one allocation system with a different *unassignment* rate.

H_{0b} : All allocation systems have the same *strict precision* rate, that is, there is no statistically significant difference between their rates.

H_{1b} : There is at least one allocation system with a different *strict precision* rate.

H_{0c} : All allocation systems have the same *strict recall* rate, that is, there is no statistically significant difference between their rates.

H_{1c} : There is at least one allocation system with a different *strict recall* rate.

H_{0d} : All allocation systems have the same *approximate precision* rate, that is, there is no statistically significant difference between their rates.

H_{1d} : There is at least one allocation system with a different *approximate precision* rate.

H_{0e} : All allocation systems have the same *approximate recall* rate, that is, there is no statistically significant difference between their rates.

H_{1e} : There is at least one allocation system with a different *approximate recall* rate.

5.2.6 Experiment Design

We generated, in a random order, 12 automatic allocations for each one of the 100 allocation inputs available. Six of these allocations were generated by our allocation systems and the other six were generated by the random algorithms. The random algorithms differ from each other only in the value of their seeds and they randomly allocate a tester for a given test case. All test cases are allocated. Therefore random algorithms always have zero unassignment rate.

Since the characteristics of each allocation input (number of tests, testers available, etc.) can influence the performance of the algorithms, we control this confounding factor by considering each allocation input as a block, in a randomised complete block design.

5.3 Execution

During the experiment, we noticed that the allocation of some plans seems not to follow a typical allocation done by TTC's test managers. For example, an allocation input with more than 200 test cases had a particular tester allocated to a single test case. We investigated these scenarios together with the test managers and they confirmed that such allocations did not reflect the initial planning. For instance, a particular tester was busy in some other activity and became available at the last minute during test *execution*. That tester was then asked to rerun one particular test case. This allocation was stored in the system as if it was the manager's initial plan. Whenever such cases happened, both the tester and the test case associated to that tester were removed from the allocation input. This exceptional situations happened in 11 allocation inputs.

5.4 Analysis

This section presents the outcome of our experiment with respect to the metrics M_1 to M_5 presented in Section 5.2.1. We use these metrics to test our statistical hypotheses through graphical analyses, ANOVA (Analysis of Variance) + Tukey's test or Wilcoxon/Kruskal-Wallis, as detailed next.

5.4.1 M_1 : Unassignment

Figure 5.2 shows the boxplot for the rate of unassigned test cases per allocation system. The random algorithm is not included in this analysis as it always allocates all test cases. The horizontal lines crossing the boxplot indicates the average, while the dots show how the values spread over the Y-axis. Notice that Exp-Manager, Exp-Blind, and MO-Blind achieved zero unassignment rates for all the allocation inputs, which means that every single test case, from each one of the 100 allocation inputs, was assigned to a tester. MO-Manager also had a good performance: out of 100 allocation inputs, only 4 had unassignment rates larger than zero; and its maximum unassignment rate was 2.70%.

The two allocation systems based on the Effectiveness profile (Eff-Manager and Eff-Blind) did not perform very well when compared to the other systems. The maximum unassignment rate achieved was 10.81% and 25% for Eff-Blind and Eff-Manager, respectively. Although they had achieved high unassignment rates, both systems had very low average values: 0.48% and 2.34% for Eff-Blind and Eff-Manager, respectively.

Table 5.1 shows the average, standard deviation, median, maximum and minimum values for unassignment. Notice that all allocation systems had zero unassignment rate for at least one allocation input. The average unassignment rate was never higher than 2.34%, and the maximum unassignment rate was never higher than 25%.

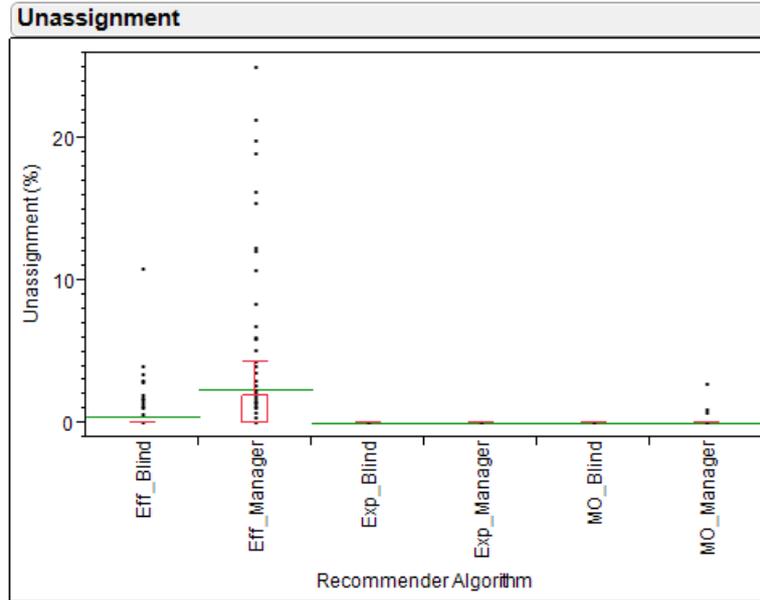


Figure 5.2 Unassigned test cases per allocation system.

Table 5.1 Test case unassignment.

	Avg.	Std. Dev.	Median	Max.	Min.
Exp-Manager	0%	0%	0%	0%	0%
Eff-Manager	2.34%	5.06%	0%	25.00%	0%
MO-Manager	0.05%	0.30%	0%	2.70%	0%
Exp-Blind	0%	0%	0%	0%	0%
Eff-Blind	0.48%	1.37%	0%	10.81%	0%
MO-Blind	0%	0%	0%	0%	0%

5.4.2 M_2 : Strict Precision

Figure 5.3 shows the boxplot for the metric M_2 (strict precision). The random algorithms are labelled as Rnd1, Rnd2, ..., Rnd6. The random algorithms differ only in the seed used. The average strict precision among our allocation systems varied from 39.32% to 45.41%. The median values ranged from 39.13% to 42.86%. The maximum strict precision value rate achieved was 97.85% (Eff-Blind). Overall, it is clear from Figure 5.3 that the random algorithms had a lower performance in comparison to our allocation systems.

In order to evaluate if any of the allocation systems is significantly different from the others, we removed the results from the random allocations and performed an analysis of variance. The p -value for the ANOVA was 0.00001, which means that we can reject the null hypothesis of equal means at 5% of significance level, supporting the alternative hypothesis that at least one allocation system has performance (strict precision) significantly different from the others. We performed a residual analysis and confirmed that all ANOVA assumptions were met.

Although there is sufficient evidence to reject the claim of equal population means, we

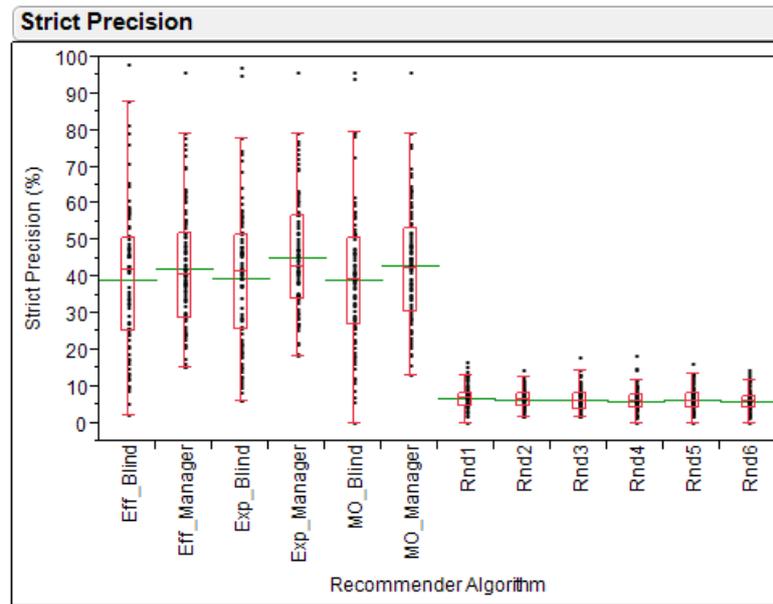


Figure 5.3 Strict precision per allocation system.

cannot conclude which allocation systems are different from the others by considering only the results from ANOVA and the graphical analysis. In order to classify the allocation systems (better or worse), we performed a multiple comparison procedure called Tukey's test. Table 5.2 shows the result of the comparisons for all pairs using the Tukey's test. Cells with positive values refer to pairs whose means are significantly different. In this case, the systems Exp-Blind, Eff-Blind and MO-Blind are significantly different from Exp-Manager.

Table 5.2 Strict Precision - Comparisons for all pairs using Tukey's HSD

	Exp-Manager	MO-Manager	Eff-Manager	Exp-Blind	Eff-Blind	MO-Blind
Exp-Manager	-4.35214	-1.84874	-1.19774	1.33136	1.67316	1.74536
MO-Manager	-1.84874	-4.35214	-3.70114	-1.17204	-0.83024	-0.75804
Eff-Manager	-1.19774	-3.70114	-4.35214	-1.82304	-1.48124	-1.40904
Exp-Blind	1.33136	-1.17204	-1.82304	-4.35214	-4.01034	-3.93814
Eff-Blind	1.67316	-0.83024	-1.48124	-4.01034	-4.35214	-4.27994
MO-Blind	1.74536	-0.75804	-1.40904	-3.93814	-4.27994	-4.35214

Table 5.3 presents the allocation systems classified in levels after the Tukey's test execution. Levels not connected by same letter are significantly different. So, we can read the data displayed in Table 5.3 as follows:

1. **A** (Exp-Manager) is definitely better than any system classified as **B** (all Blinds); and it is better than or equal to the systems classified as **AB** (MO-Manager and Eff-Manager).
2. **AB** is better than or equal to the systems classified as **B**; also, it can be as good as any system classified as **A** (more data is needed to better identify these differences).

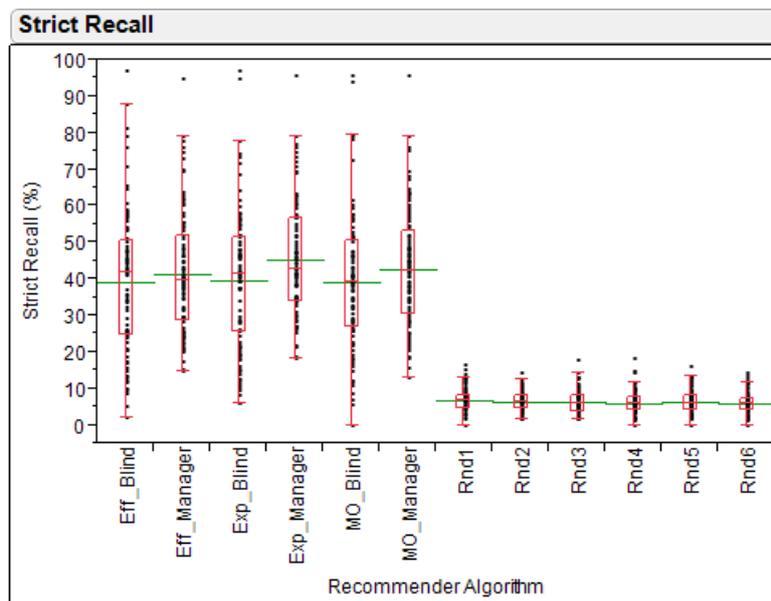
Table 5.3 Strict Precision - Allocation systems classified in levels after Tukey's test.

Allocation System	Level	Mean
Exp-Manager	A	45.41
MO-Manager	A B	42.91
Eff-Manager	A B	42.26
Exp-Blind	B	39.73
Eff-Blind	B	39.39
MO-Blind	B	39.32

3. **B** is definitely worse than any system classified as **A**; also, it can be equal to any system classified as **AB**.

5.4.3 M_3 : Strict Recall

Figure 5.4 shows the boxplot for the strict recall metric. For the allocation systems, the average strict recall varied from 39.19% to 45.41%. The medians were also very consistent across the allocation systems: they range from 39.13% to 42.86%. The maximum strict recall was 96.97% (Exp-Blind). Similarly to M_2 , the boxplot suggests that all allocation systems are better than the random algorithms.

**Figure 5.4** Strict recall per allocation system.

Note that the random algorithm always allocates all test cases. Because of that, the figures for strict precision and strict recall are the same for the random algorithms (the denominators are the same in the equations for M_2 and M_3). That also happens to approximate precision and approximate recall.

Again, we removed the results from the random allocations and ran an analysis of variance to evaluate if any of the allocation systems have performance (strict recall) significantly different from the others. As none of the assumptions required by ANOVA was violated, we used this statistical test.

The p -value for the ANOVA was 0.0001, giving us sufficient evidence to reject the null hypothesis of equal population means and to conclude that at least one allocation system has performance significantly different from the others. In order to identify the specific means that are different we applied the Tukey's test.

Table 5.4 shows the result of the comparisons for all pairs using the Tukey's test with respect to the metric M_3 (strict recall). Positive values show pairs of means that are significantly different. Again, the Blinds are different from Exp-Manager.

Table 5.4 Strict Recall - Comparisons for all pairs using Tukey's HSD

	Exp-Manager	MO-Manager	Eff-Manager	Exp-Blind	Eff-Blind	MO-Blind
Exp-Manager	-4.36193	-1.83593	-0.35553	1.32157	1.73557	1.86487
MO-Manager	-1.83593	-4.36193	-2.88153	-1.20443	-0.79043	-0.66113
Eff-Manager	-0.35553	-2.88153	-4.36193	-2.68483	-2.27083	-2.14153
Exp-Blind	1.32157	-1.20443	-2.68483	-4.36193	-3.94793	-3.81863
Eff-Blind	1.73557	-0.79043	-2.27083	-3.94793	-4.36193	-4.23263
MO-Blind	1.86487	-0.66113	-2.14153	-3.81863	-4.23263	-4.36193

Table 5.5 presents the allocation systems classified in levels after the Tukey's test execution. The only difference between Table 5.5 and Table 5.3 (on page 53) is that MO-Blind and Eff-Blind exchanged positions because of their mean values. As both remained in the same group (**B**), the interpretation is the same as described for metric M_2 .

Table 5.5 Strict Recall - Allocation systems classified in levels after Tukey's test.

Allocation System	Level	Mean
Exp-Manager	A	45.41
MO-Manager	A B	42.89
Eff-Manager	A B	41.41
Exp-Blind	B	39.73
MO-Blind	B	39.32
Eff-Blind	B	39.19

5.4.4 M_4 : Approximate Precision

Figure 5.5 presents the boxplot for the metric M_4 (approximate precision). We can observe that both averages and medians are larger than those for strict precision metric. This was expected, as approximate precision (and approximate recall) do not require the systems to recommend exactly the same pair as allocated by the manager. Notice that the major difference from the results for strict precision is that the performance of the random algorithm has also improved.

The average approximate precision for our allocation systems varied from 62.01% to 64.83%. Medians varied from 54.73% to 59.18%. The minimum value for approximate precision was 10% and the maximum value was 100%. The boxplot still gives some indication that our allocation systems are superior to the random ones.

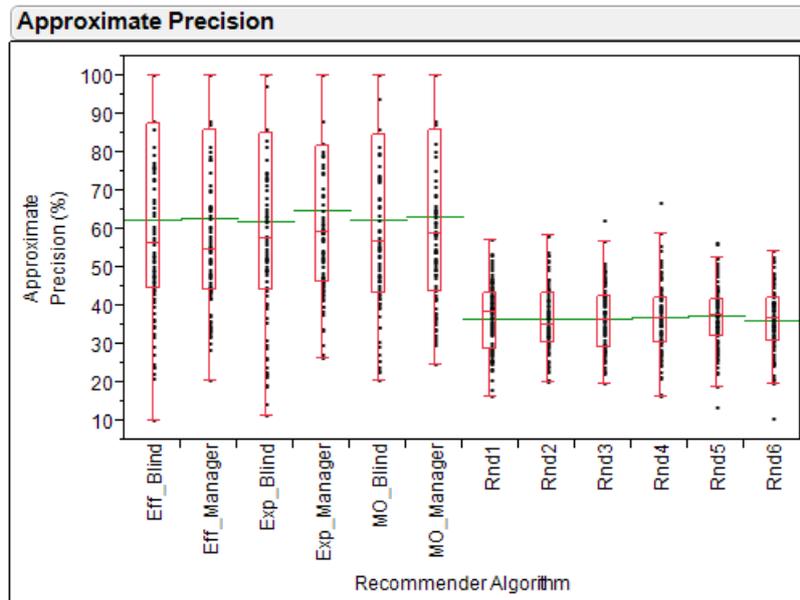


Figure 5.5 Approximate precision per allocation system.

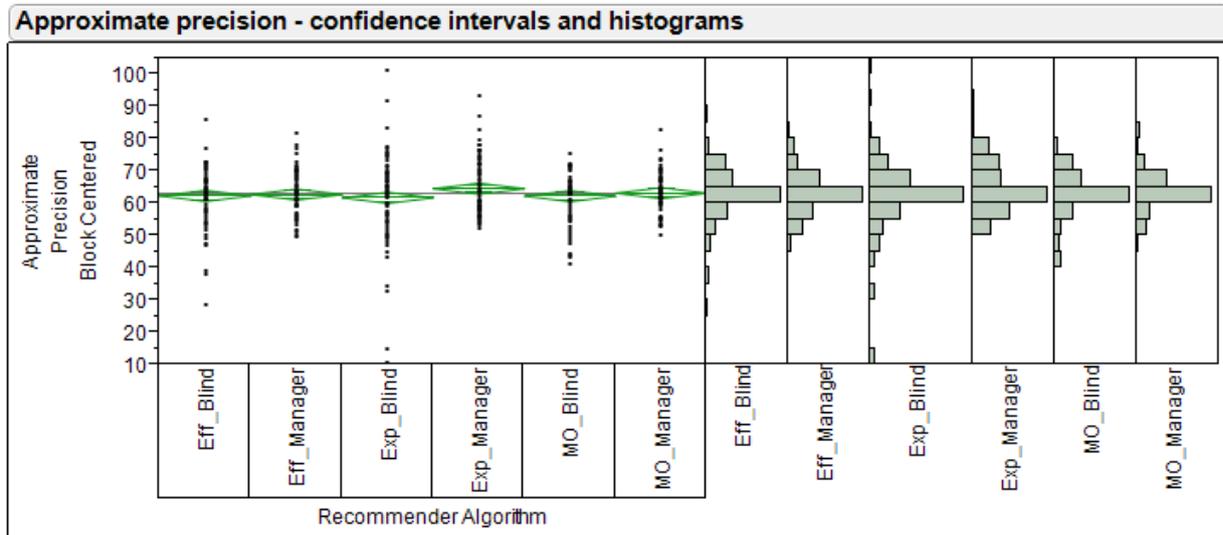
In order to evaluate if any of the allocation systems was significantly different from the others, we ran the analysis of variance. This time, the Anderson-Darling test showed that the residuals from ANOVA did not follow a normal distribution (or a distribution close enough to a normal one) that allowed us to apply the ANOVA. Hence, we also performed the Wilcoxon/Kruskal-Wallis test as it does not assume a normal distribution. Figure 5.6 shows the confidence intervals as well as the histograms of each allocation system. A quick glance at the confidence intervals gives already an indication that there is an overlap between all the allocation systems.

By running the Wilcoxon/Kruskal-Wallis test considering only our allocation systems, we found that all allocation systems are similar to each other at $\alpha = 0.05$ (p -value = 0.8509). Therefore, we can conclude that they are all similar among themselves, but different from the random algorithms (an outcome in agreement to our intuition when looking at the boxplot).

Table 5.6 shows the average, standard deviation, median, maximum and minimum values for approximate precision. Notice that all allocation systems achieved 100% approximate precision for at least one allocation input. The standard deviation rate was never higher than 25.57%, and the minimum approximate precision rate was never lower than 10%.

5.4.5 M_5 : Approximate Recall

As expected, approximate recall has also improved in comparison to strict recall (see Figure 5.7). Both average and median increased and all allocation systems reached 100% for some



Block Test Plan

Figure 5.6 Approximate precision - confidence intervals and histograms.

cases. Similarly to approximate precision, the random algorithm also improved markedly in comparison to strict recall. Both the average and the median for approximate recall were very consistent. The average varied from 61.85% to 64.83% and the median varied from 54.53% to 59.18%. The maximum value was 100% and minimum value was 9.86%. The boxplot still indicates a superior performance of our allocation systems.

Figure 5.8 shows the confidence intervals as well as the histograms of each allocation system with respect to the metric approximate recall. By looking at the histograms, we can notice that the distributions do not seem to follow a normal distribution (this intuition was confirmed after performing both the Anderson-Darling test and the ANOVA residual analysis).

By running the Wilcoxon/Kruskal-Wallis test over the allocation systems, we found that they are all similar to each other at $\alpha = 0.05$ (p -value = 0.1666). Again, this means that the allocation systems are similar among themselves, but different with respect to the random algorithms.

Table 5.7 shows the average, standard deviation, median, maximum and minimum values for approximate recall. Similarly to approximate precision, all allocation systems achieved 100% approximate recall for at least one allocation input. The average approximate recall ranged from 61.85% to 64.83% and the standard deviation rate was never higher than 26.61%.

5.5 Interpretation

This section discusses the analysis presented in Section 5.4 and addresses the threats to validity.

Table 5.6 Approximate Precision.

	Avg.	Std. Dev.	Median	Max.	Min.
Exp-Manager	64.83%	22.68%	59.18%	100.00%	26.09%
Eff-Manager	63.02%	24.17%	54.73%	100.00%	20.29%
MO-Manager	63.44%	23.94%	58.71%	100.00%	24.64%
Exp-Blind	62.01%	26.61%	57.40%	100.00%	11.11%
Eff-Blind	62.59%	25.57%	56.12%	100.00%	10.00%
MO-Blind	62.54%	25.52%	56.86%	100.00%	20.21%
Rnd1	36.87%	8.91%	38.38%	56.90%	16.42%
Rnd2	36.51%	8.74%	35.05%	58.33%	20.00%
Rnd3	36.80%	8.41%	36.43%	62.00%	19.40%
Rnd4	36.98%	9.08%	36.59%	66.67%	16.22%
Rnd5	37.36%	8.33%	37.38%	56.25%	13.51%
Rnd6	36.36%	8.41%	36.55%	54.00%	10.53%

5.5.1 Evaluation of results and implications

- **Unassignment.** Three allocation systems (Exp-Manager, Exp-Blind, and MO-Blind) achieved zero unassignment rates: all test cases from the allocation input were assigned to a tester. MO-Manager also performed very well as it had unassignment rates greater than zero in 4 allocation inputs out of 100. The two allocation systems based on the effectiveness profile (Eff-Manager and Eff-Blind) did not perform very well when compared to the other systems. The maximum unassignment rate achieved was 10.81% and 25% for Eff-Blind and Eff-Manager, respectively.

High unassignment rates (25%) show that many test cases are poorly documented. Test cases without any information about *component*, *feature* or *category* will have its symbolic description based solely on the the *test case summary* (or *description*). This scenario can make the allocation systems to assign very low similarities to all testers. If the similarity of a test case is smaller than the *cutoff* point for all testers, then the test case is not assigned to anyone.

It was already expected that the allocation systems based on the effectiveness profile would not perform as well as the ones based on the expertise profile. The preconditions for creating the effectiveness profile are very restrictive: only test cases that failed with valid defects are used for creating the tester profile (Section 3.2.2). Considering a hypothetical situation in which a tester has run 100 test cases in the past and failed only 2 with valid defects, the expertise profile will be created considering the 100 test cases, while the effectiveness profile will consider only 2 test cases. Little information on the tester profile can result in low similarity rates and, consequently, higher unassignment rates.

- **Strict Precision.** The ANOVA showed that at least one allocation system was significantly different from the others. After running the Tukey's test, we noticed that the Exp-Manager stood out when compared to the other allocation systems.

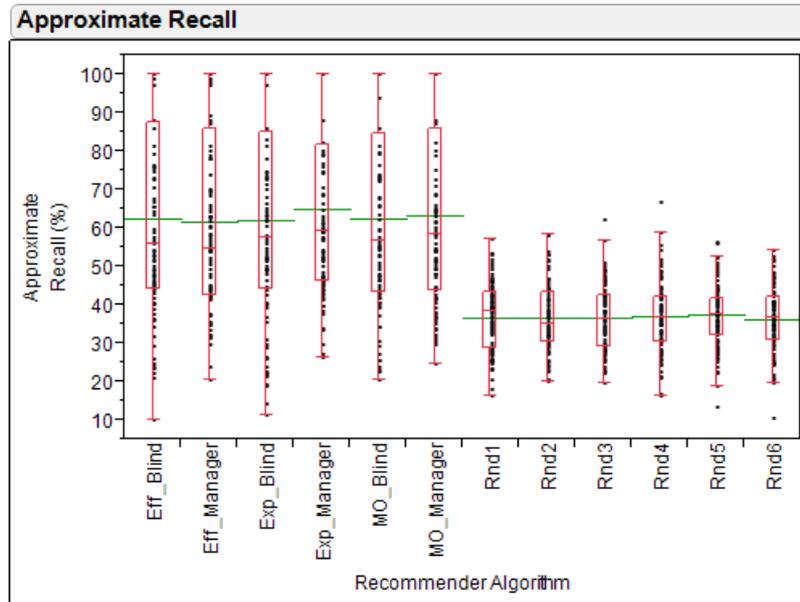


Figure 5.7 Approximate recall per allocation system.

The Manager-based systems (Exp-Manager, Eff-Manager, and MO-Manager) achieved better results than the Blind ones with respect to the mean values: this was somehow expected as the Manager-based systems receive more information from the manager before creating the automatic allocations. In addition to the list of test cases to be assigned, the list of testers available, and the cutoff point, they also receive the amount of test cases to be assigned to each tester.

Although the Manager-based systems performed very well in some cases (95.74%), the average was never above 45.41%, and the median never above 42.86%. An allocation system does not seem to be capable of identifying the exact person chosen by the manager. This is expected as even a different manager would have difficulties in reproducing the exact results of another manager as there are many possible alternatives for a given test case (the equivalence groups showed that there are many testers with equivalent skills).

- **Strict Recall.** ANOVA and Tukey's test were performed and again the Exp-Manager stood out when compared to the other allocation systems. The results for strict recall are in agreement with the results for the metric M_2 (strict precision). The average strict recall are never above 45.41% and the median, never above 42.86%. Strict correct pairs seems to be a too restrictive request for a recommender system to detect.
- **Approximate Precision.** By using approximate precision, on the one hand, the performance improved to an extent in which all allocation systems reached, for some cases, 100%. The average approximate precision varied from 62.01% to 64.83%. On the other hand, the random algorithm also improved noticeably. Despite that, the average approximate precision for the random algorithm was never higher than 37.36%. By inspecting

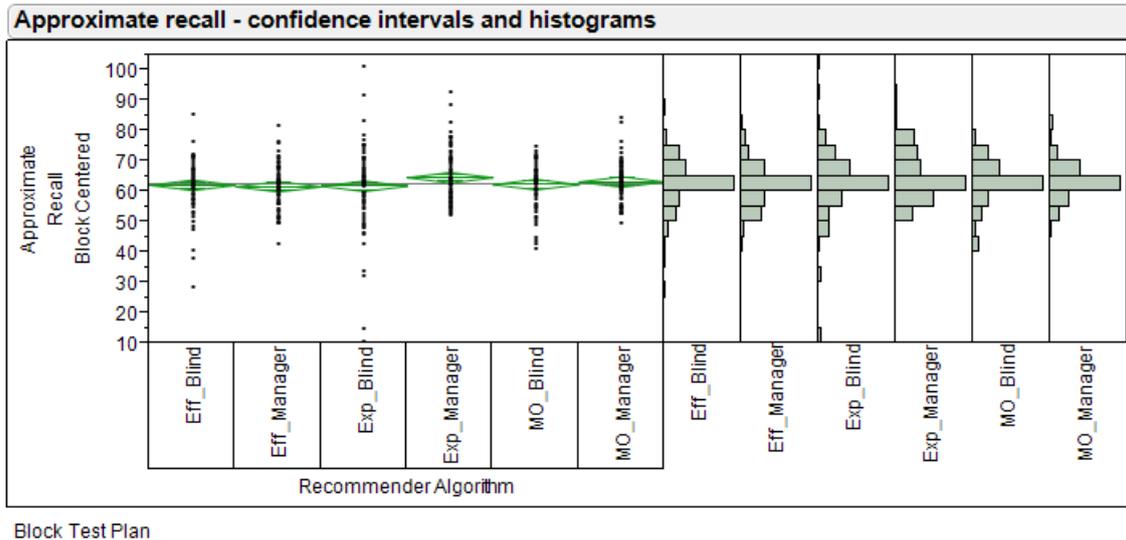


Figure 5.8 Approximate recall - confidence intervals and histograms.

the boxplot we could conclude that the allocation systems performed better than the random algorithm; and the Wilcoxon/Kruskal-Wallis test showed that the allocation systems are similar among themselves.

- **Approximate Recall.** As expected, approximate recall performed better than strict recall. This was valid for both allocation systems and random algorithm. The average recall for the allocation systems varied from 61.85% to 64.83%. Similarly to metric M_4 (approximate precision), the Wilcoxon/Kruskal-Wallis test showed that the allocation systems are similar among themselves. Again, the visual analysis of the boxplot indicates that the allocation systems are better than the random algorithm.

5.5.2 Threats to Validity

We identified the following threats to validity:

- **Analysis of compromised data.** The pressure for meeting the deadlines and other daily problems at TTC may have resulted in an ineffective allocation (by the managers) for some allocation inputs. As a consequence, the results can show a lower effectiveness of our approach, as our base of comparison (the manual allocation) may be partially compromised. This threat was controlled by careful filtering the historical data to remove inconsistent test plans.
- **Test cases poorly documented.** The existence of poorly documented test cases (attributes missing) in the TTC's database may have resulted in low similarity rates between the test cases and testers, thus leading to higher unassignment rates for some allocation inputs.

Table 5.7 Approximate Recall.

	Avg.	Std. Dev.	Median	Max.	Min.
Exp-Manager	64.83%	22.68%	59.18%	100.00%	26.09%
Eff-Manager	61.85%	24.61%	54.53%	100.00%	20.29%
MO-Manager	63.42%	23.95%	58.44%	100.00%	24.64%
Exp-Blind	62.01%	26.61%	57.40%	100.00%	11.11%
Eff-Blind	62.34%	25.62%	55.64%	100.00%	9.86%
MO-Blind	62.54%	25.52%	56.86%	100.00%	20.21%
Rnd1	36.87%	8.91%	38.38%	56.90%	16.42%
Rnd2	36.51%	8.74%	35.05%	58.33%	20.00%
Rnd3	36.80%	8.41%	36.43%	62.00%	19.40%
Rnd4	36.98%	9.08%	36.59%	66.67%	16.22%
Rnd5	37.36%	8.33%	37.38%	56.25%	13.51%
Rnd6	36.36%	8.41%	36.55%	54.00%	10.53%

- **Validation cannot be generalised.** We analysed test data from a single test company for a particular industry (mobile phone testing). This can be considered a threat to generalise our results in a different industrial setting.

5.6 Concluding remarks

Unassignment metric showed that three allocation systems (Exp-Manager, Exp-Blind, and MO-Blind) performed clearly better than the others by achieving zero unassignment rates for each one of the 100 allocation inputs. The maximum unassignment rate achieved was 10.81% and 25% for Eff-Blind and Eff-Manager, respectively.

For strict precision and strict recall metrics, the ANOVA showed that at least one allocation system was significantly different from the others. After running the Tukey's test, we noticed that the Exp-Manager stood out when compared to the other allocation systems and achieved mean values of 45.41% for both metrics.

As expected, approximate precision and approximate recall performed better than strict precision and strict recall, respectively. After running the Wilcoxon/Kruskal-Wallis test over the allocation systems only, we found that all allocation systems are similar to each other but different from the random algorithm and, by looking at the boxplots, we can conclude that different means better.

Related Work

In this chapter we present an overview of previous work on recommender systems applied to testing, debugging or people recommendation. As it was not easy to find works closely related to ours, we also describe a few projects that apply recommender systems and other artificial intelligence techniques to the software engineering domain in general.

6.1 Not All Classes are Created Equal: Toward a Recommendation System for Focusing Testing

Kpodjedo *et al.* [KRG08] proposed a metric to evaluate, in an object oriented system, which classes deserve more attention from the tester (or from the test manager) to distribute resources and assign testing effort.

Based on the following assumptions: (i) “the most important classes” must be tested more deeply and (ii) frequently changed classes are the most complex and therefore the most fault-prone, their algorithm computes, for each class, a pair of values representing the frequency of changes (Evolution Cost) and the class overall connectivity (PageRank). Random walks are implemented using a basic PageRank algorithm, which, considering the relations among classes, measures, for each class, its relative importance in a system and assigns it a numerical weight. The Evolution Cost quantifies how much the class and its relations changed in a time frame. The recommendation to the product manager and/or tester is a scatter-plot indicating the “critical classes” (i.e., important classes that have been changing frequently in the past).

The system was applied to the development of Mozilla and identified 9 critical classes from a total of 9,000 classes with 23,000 relations. An example of output from their approach (the scatter-plot) is shown in Figure 6.1. Points near to the origin represent classes “not so important” that have changed rarely. Points far from the origin (upper right corner) represent potentially “critical classes” and they should be carefully considered by the tester/manager to distribute resources and assign test effort. The point is that a high ranked class plays an important role in the system and if it changes quite often or dramatically, there are more chances to have residual defects potentially impacting various system functionalities.

6.2 Recommending Emergent Teams

Minto and Murphy [MM07] introduced the Emergent Expertise Locator (EEL), a tool that uses emergent team information to propose experts to a developer within their development

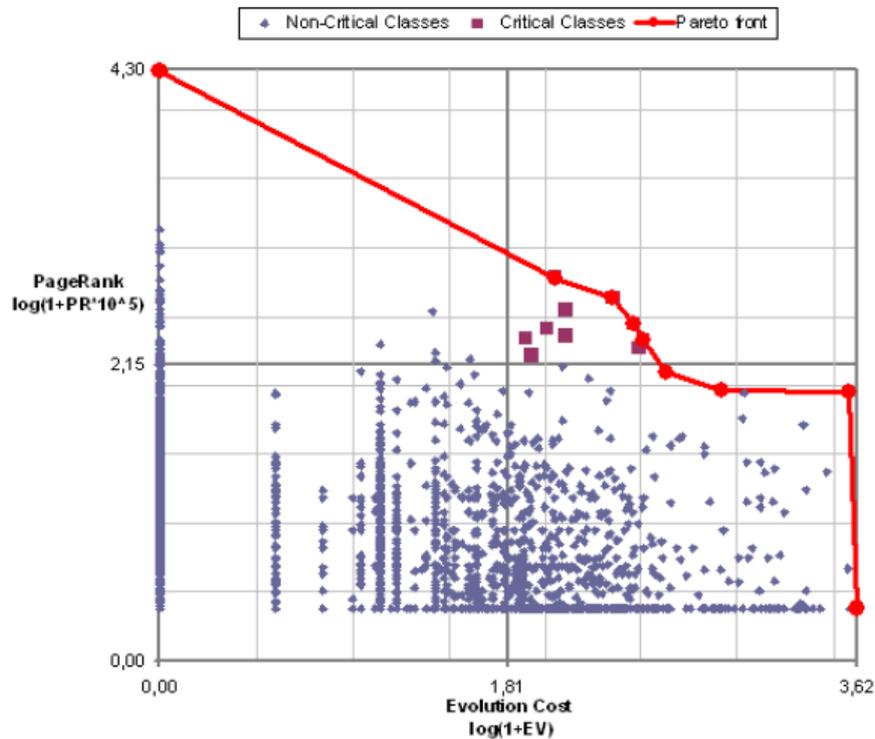


Figure 6.1 Evolution Cost and PageRank view.

environment as the developer works. Based on the history of how files have changed in the past together and who has participated in the changes, EEL can recommend members of an emergent team for the current problem of interest (an expert is recommended by the system to help others to solve a particular problem).

EEL is implemented as a Java plug-in for Eclipse and it displays a ranked list of other developers with expertise on the set of files that the current user of EEL has recently edited or selected. To use EEL, a developer access a menu on a source file that displays a ranked list of developers along with ways to initiate a communication. This scenario is illustrated in Figure 6.2.



Figure 6.2 EEL - ranked list of developers and the ways to initiate a communication.

EEL's approach is based on the mechanism of using matrices to compute coordination re-

quirements introduced by Cataldo *et al.* [CWHC06]. It involves two matrices, the file dependency matrix and the file authorship matrix, and produces a third, the expertise matrix.

To evaluate the approach, historical data from three existing open-source software projects was used: Eclipse, Firefox, and Bugzilla. These projects were chosen because they have sufficient amount of data to run the validation. After choosing the projects, they compared the results of EEL against “Line 10” rule that was used in the Expertise Recommender (ER) [MA00]. The “Line 10” rule states when the last person that modified the file is considered as the expert in that file.

To investigate the impact of the size of the recommendations on the performance of EEL, precision and recall were computed for three different sized lists of potential team members, namely three, five and seven recommendations. These lists were obtained by taking the top parts of the ordered list produced by EEL and the “Line 10” rule.

The best precision achieved by EEL was 37% (Eclipse project) against 28% achieved by “Line 10” whereas the worst precision values were 16% (Firefox) against 13% achieved by “Line 10”. Regarding recall, the best result achieved by EEL was 49% (Eclipse project) against 35% achieved by “Line 10”. The worst recall values were achieved on Firefox project with 21% (EEL) and 16% (“Line 10”). Detailed results can be seen in Table 6.1.

Table 6.1 Recommending Emergent Teams - Average precision and recall.

Project	Precision		Recall	
	EEL	Line 10	EEL	Line 10
Eclipse	37%	28%	49%	35%
Bugzilla	28%	23%	38%	28%
Firefox	16%	13%	21%	16%

Two limitations of EEL are presented in the paper: the first limitation is that EEL works only in the newer version control systems such as Subversion [PCSF08]; it does not work with many traditional version control systems like CVS [Ves03] and RCS [Tic85] as they maintain commit information on a per file basis. The second limitation is that if a new developer is added to the team, and that developer is already an expert, there is no support to ensure that this person is correctly recommended. Despite those limitations, it is clear that the Emergent Expertise Locator (EEL) eases collaboration for dynamic teams by determining the composition of the team automatically.

6.3 Allocating Global Software Teams in Software Product Line Projects

Following the same line of research developed by Minto and Murphy [MM07], Pereira *et al.* [PdSRE10] proposed a framework to allocate people geographically distributed for developing components together. The allocation of teams takes place in the context of Software Product Lines (SPL) development.

As global software development (GSD) approaches have their limitations, mainly related to communication between dispersed development teams, it is important to consider, besides

the technical skills of the teams (or people), some nontechnical attributes, such as language, time-zone, and cultural principles during the human resources allocation process.

The proposed framework analyses technical and non-technical aspects of the SPL project and the available software development teams in order to provide recommendations for allocating global software teams involved in the implementation phase of software components of a SPL project. Considering that technical and non-technical features can impact on communication requirements, the framework intends to provide a systematic way of recommending teams allocation based on the mitigation of communication needs among global software teams.

We do not have information about the framework's performance as the assessment of the presented technique was out of paper scope.

6.4 Expertise Recommender

McDonald and Ackerman [MA00] proposed the Expertise Recommender (ER) to assist the natural expertise locating behaviour in organisations. By relying on organisationally relevant sources of information and heuristics that are naturally used, ER can assist in finding people with desired expertise.

The ER was deployed on a medium sized software company that builds, sells, and supports medical and dental practise management software: the Medical Software Corporation (MSC).

For MSC, ER is implemented based on two specific identification heuristics that were used by MSC's participants:

1. ***Change History Heuristic (or "Line 10 Rule")***. It is a heuristic designed to augment an expertise locating behaviour common to MSC's developers: Given a problem with a module a developer looks into the version control system to see who last modified the code and then approaches that person for help. The idea behind this heuristic is that the person who last made a change has the code "freshest" in mind.
2. ***Tech Support Heuristic***. The heuristic augments a behaviour of technical support representatives when faced with an unfamiliar or difficult support problem: When faced with a difficult problem, a representative will perform multiple, separate queries over the support database using the symptoms, customer, or program module involved. The support rep then scans the records sequentially looking for similarities between the current problem and any past problems as returned by the different queries. In scanning records a support rep looks to identify people who have previously solved similar problems.

Besides these two heuristics, the MSC implementation includes three selection techniques:

1. ***No Filter***. This technique removes individuals who no longer work for MSC.
2. ***Departmental***. This technique selects based on the organizational distance between the department of the person making the request and the department of each person recommended. The filter module maintains a small graph structure that represents the "distance" between each of the departments at MSC.

3. **Social Network.** Recommendations are filtered based on an aggregate social network of the MSC workplace. This social network is a graph structure where the nodes represent individuals at the workplace and the edges represent some relation that links two individuals.

According to the authors these selection mechanisms can be generalised to other organisations other than MSC.

Figure 6.3 illustrates the expertise request dialog which allows the user to find experts for a given subject in a specific “Topic Area”.

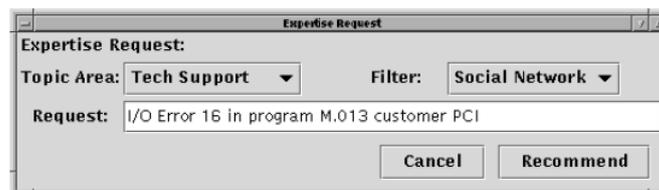


Figure 6.3 Expertise Request dialog.

Figure 6.4 shows the response dialog which displays the request followed by a list of recommended people. Each person is listed in a single pane containing contact information that includes office number, phone number, phone extension, and email address.

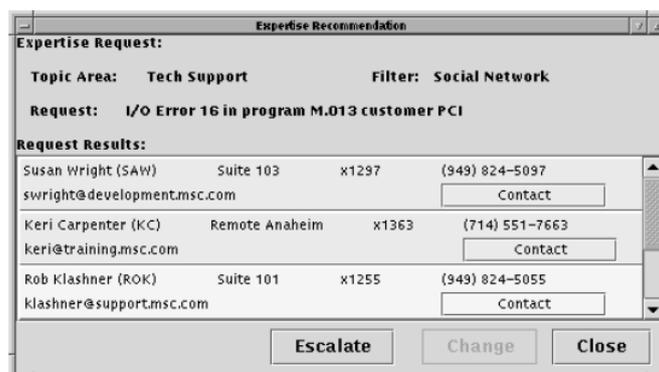


Figure 6.4 Recommendation response.

6.5 Who Should Fix this Bug?

Anvik *et al.* [AHM06] developed a semi-automated approach to assign defect reports to a developer with appropriate expertise to fix that defect. Their approach uses a supervised machine learning algorithm that is applied to the information available in the defect management system.

A supervised machine learning algorithm takes as input a set of instances with known labels and generates a classifier that can be used later to assign a label to an unknown instance. In their work, an instance is a defect report and the label is the name of the developer who either

was assigned to the report or resolved it. They evaluated three different machine learning algorithms: Naïve Bayes (a probabilistic classification algorithm), Support Vector Machines or SVM (a non-linear classification algorithm), and C4.5 (a decision tree algorithm). The SVM was chosen after having shown the best precision and recall rates.

When a new defect arrives in the repository, the supervised machine learning algorithm suggests developers who may be qualified to resolve the defect. A small list of potential resolvers is recommended because groups of developers often work in similar kinds of problems. The recommendations are made based on defect reports that the developers have been previously assigned or resolved for the system. The approach is semi-automated because the user (called triager) still needs to select, from the recommended set, the actual developer to handle the defect. The triager may make this choice based on knowledge other than that available in the defect management system, such as the workload of the developers, or the vacation schedule.

To evaluate their approach, they applied the algorithm against three open source defect repositories: Eclipse, Firefox, and GCC compiler project. They were able to achieve good precision results for the Eclipse and Firefox projects with 57% and 64%, respectively. The result for the GCC, however, was very different with precision rates hovering around 6%. Regarding recall, the results reported were quite low: when reporting only one recommendation, the highest recall achieved on average was 10% (Eclipse), 3% (Firefox), and 8% (GCC).

6.6 Expertise Browser: A Quantitative Approach to Identifying Expertise

Mockus and Herbsleb [MH02] proposed a web-based tool that uses data from change management systems to assist developers, testers, and managers in identifying experts for a number of software development tasks: the Expertise Browser (ExB). It uses a quantification of experience, and presents evidence to validate this quantification as a measure of expertise. The tool enables developers, for example, to easily distinguish someone who has worked only briefly in a particular area of the code from someone who has more extensive experience, and to locate people with broad expertise throughout large parts of the product, such as module or even subsystems.

With this tool, they try to eliminate the need for people to write and maintain descriptions of their expertise as such descriptions are generally difficult to use because different people often describe similar expertise differently (people have very different standards for judging the degree of their expertise). Also, such descriptions go out of date very quickly and are very difficult to maintain.

The expertise is measured through Experience Atoms (EAs), which are elementary units of experience. According to their definition, experience may pertain to a person, organisation, or a work product such as a piece of code. The simplest unit of experience that could be observed in projects using change management systems is the atomic change (delta) made to the source code or to documentation. The person (and the person's organisation) implementing the change gain a certain amount of experience by doing work required to change a particular part of a file. The changed work product provided a particular type of experience to a specific person.

The ExB is basically designed to answer the following questions:

1. Who has appropriate expertise for a particular part of code, documentation, functionality, or delivery?
2. What is the expertise profile of a particular part of an organisation: a person, a group of people, or a group of organisations?

To answer these questions the ExB displays aggregations of EAs into product and organisation units and displays the relationships among them. It also provides details on product and organisation units including contact information for individuals.

Figure 6.5 illustrates a typical interaction with ExB. The three views on the left display organizational units (supervisors, individual developers, and organisations) with EAs in the subsystem chosen by the user. The code view on the right shows part of the source code tree. The vertical size of a product or organizational unit represents the number of EAs gained by that unit. The horizontal size of product units represents the number of people who contributed deltas to that subsystem, module, or file. Both vertical and horizontal sizes are related to the box that wraps the components (supervisors, developers, organisations, and modules). When the user selects a potential expert in a given product unit, the contact information of that person is displayed on the bottom left corner.

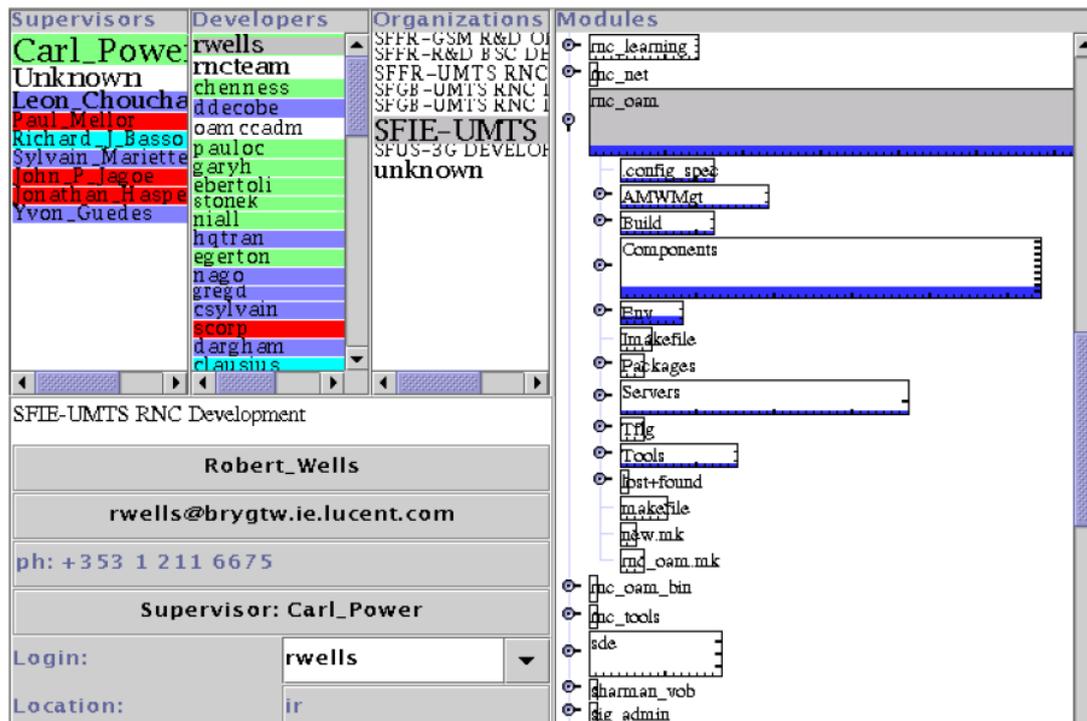


Figure 6.5 ExB user interface.

The ExB was deployed in two organisations, focused on different projects, with different code bases. After analysing the usage data they found out that satellite sites that are either new

to the project or do not have the sufficient breadth of the expertise are likely to be the most active users. They also noticed that the larger and more established main sites not only used the ExB less, but used it differently. In the newer, smaller, less established sites, the tendency was to begin with some part of the product and look for experts, or to start with a person to see where that person worked in the product. In the older, larger, more established sites, the tendency was to go from organisations and find whether they have worked in the product.

6.7 Predicting the Fix Time of Bugs

Giger *et al.* [GPG10] use prediction models to support developers in the cost/benefit analysis by recommending which defects should be fixed first. They investigated the relationships between the fix-time of defect reports and their attributes with six sub-systems taken from open source software projects: Eclipse JDT, Eclipse Platform, Mozilla Core, Mozilla Firefox, Gnome GStreamer, and Gnome Evolution.

By analysing the change history information from the defect reports, they compute the measures of some of the defect attributes (e.g.: # of comments made to a defect report, # of people in CC list, etc). In a second step, decision trees using Exhaustive CHAID algorithm are applied to classify the defect reports as “Fast” or “Slow” using the median of the defect fix-time, in hours, from opened to last fix.

To evaluate the approach, they used 10-fold cross validation: the data set is broken into 10 sets of equal size. The model is trained with 9 data sets and tested with the remaining tenth data set. This process is repeated 10 times with each of the 10 data sets used exactly once as the validation data. The results of the 10 folds then are averaged to produce the performance measures. Besides precision and recall, they also used the area under the receiver operating characteristic curve (AUC) statistic for measuring the performance of prediction models. The authors identified that *assignee*, *reporter*, and *monthOpened* (the month in which the defect report was opened) are the attributes that have the strongest influence on the fix-time of defects. The average precision varied from 60.8% (Mozilla Firefox) to 65.4% (Eclipse Platform); the recall ranged from 48.5% (Eclipse JDT) to 73.2% (Mozilla Firefox); and the AUC varied from 64.9% (Eclipse JDT) to 74.3% (Eclipse Platform). Detailed results from this work can be seen in Table 6.2.

Table 6.2 Performance of prediction models computed with initial attribute values.

Project	Precision	Recall	AUC
Eclipse JDT	63.5%	48.5%	64.9%
Eclipse Platform	65.4%	69.2%	74.3%
Mozilla Core	63.9%	64.1%	70.1%
Mozilla Firefox	60.8%	73.2%	70.1%
Gnome GStreamer	64.6%	69.4%	72.4%
Gnome Evolution	62.8%	69.5%	69.4%

After the first evaluation, they measured the performance of prediction models again. This

time, using post-submission data: in addition to the initial values they obtain the attribute values for 1 day, 3 days, 1 week, 2 weeks, and for 1 month after a defect report was opened. Using this approach, they managed to improve the performance of the prediction models by 5% to 10%.

The approach developed by Giger et al. can be very useful to new developers as it gives an insight in how defects are prioritised in a software project.

6.8 PR-Miner

PR-Miner (Programming Rule Miner) [LZ05a] is a tool that uses a data mining technique called frequent itemset mining to identify patterns on implicit programming rules. For example, the functions `lock()` and `unlock()` are always used in pairs. Besides such a well-known programming rule, there are many other implicit rules in large software. Such rules are useful information for software development. Unfortunately, they usually exist only in programmers' minds as they are too tedious to be documented manually. In addition, rule maintenance is a hard task since some rules can change in new versions. Violations to these rules are easy for programmers to introduce, especially for new programmers who are unaware of these rules.

The tool not only identifies these recurrent patterns but also detects their violations. The authors report thousands of occurrences of such patterns in projects like Linux and PostgreSQL. Such violations are potential defects being introduced. Once the list of suspects is generated, a manual inspection must be done to eliminate the false positives. By analysing the top 60 violations, 16 defects were found in Linux, 6 in PostgreSQL and 1 in Apache. Most of these defects violate complex rules that contain more than 2 elements and are thereby difficult to be detected by previous tools.

6.9 Dynamine

DynaMine [LZ05b] analyses source code check-ins and, just like PR-Miner [LZ05a], extracts coding patterns and their violations. Its method can learn both simple and complex patterns and scales to millions of lines of code.

The tool combines revision history information with dynamic analysis for the purpose of finding software errors. It largely automates the mining and dynamic execution steps and makes the results of both steps more accessible by presenting the discovered patterns as well as the results of dynamic checking to the user in custom Eclipse views.

The authors applied the tool on two real software systems (Eclipse [Hol04] and jEdit) and found 57% of patterns' occurrences, in which 66% of those were violations.

6.10 Concluding Remarks

In this chapter we presented some works that apply recommender systems and other artificial intelligence techniques to the software engineering domain. It was very difficult to find works closely related to ours. Most of the works are related to recommender systems

being applied to software engineering in general: recommending people with desired expertise [MM07, MA00, MH02]; recommending developers to fix defects [AHM06]; allocating people geographically distributed for developing components together [PdSRE10]. We found only one work related to recommender systems being applied to testing: Kpodjedo *et al.* [KRGA08] developed a recommendation system to suggest which classes, in an object oriented system, should be tested more deeply. Even the book “Artificial Intelligence Methods In Software Testing” [LKB04], which collects a representative sample of artificial intelligence applications in the areas of software testing, does not mention any application of recommender systems applied to software testing. As far as we know there are no works being developed that make use of recommender systems to assign test cases to testers.

If, on the one hand, the lack of closely related works did not allow us to perform a comparative analysis to confront prior approaches to ours and eventually show any improvements brought by our method, on the other hand, it gives a strong evidence that we are addressing a new problem. The adoption of recommender systems for helping test managers to assign test cases to testers seems to be a new contribution.

Conclusion and Future Work

We proposed 6 allocation systems to allocate test cases to be run manually by testers. Three of them are Manager-based ones, which take as input the number of test cases to be allocated to each tester. The other three are the Blind systems that allocate the amount of test cases to each testers solely based on similarity. For each of those two categories (Manager-based and Blind), the algorithms run over three different tester's profiles: the effectiveness profile, the expertise profile and the combination of the two (multi-objective).

Test case allocation is a task routinely done by the test managers without any support of tools. A manual allocation takes from 15 to up to 35 minutes of the manager's time. And allocations are done more or less 10 times per week. This accounts to 16 to 30 working days per year. The introduction of a tool can help in two ways:

1. The allocations can be performed in a much faster way as the managers can now use the automatic recommendation as a starting point of their allocations;
2. The arrival of new managers is not impacted as the tool provides them with some knowledge of test cases, testers and previous allocations. With the aid of a recommender tool, managers can perform better allocations faster.

We implemented six allocation systems plus one random algorithm for control. We ran all allocation systems as well as the random algorithm over 100 allocation inputs used by the TTC managers from November 2009 to September 2010. In our experiment, the ANOVA plus the Tukey's test showed us that the Exp-Manager outperforms the other systems with respect to the strict precision and the strict recall metrics. The average strict precision (among the allocation systems) varied from 39.32% to 45.41% while the average strict recall ranged from 39.19% to 45.41%. When considering the concept of *approximately correct pair* (explained in Section 5.2.1), the Wilcoxon/Kruskal-Wallis test revealed that all allocation systems are similar among themselves with respect to approximate precision and approximate recall. Both the average approximate precision and the average approximate recall (among the allocation systems) varied from 62.01% to 64.83%. By analysing the boxplots, all the allocation systems demonstrated to be superior to the random algorithm. For unassignment (percentage of test cases the algorithm could not allocate), three of our six allocation systems presented a better performance by achieving zero unassignment rate for all the allocation inputs, namely Exp-Manager, Exp-Blind and MO-Blind. The average unassignment varied from 0% to 2.34% (recall that for unassignment, the lower the better). These figures are not different from the precision and the recall of other recommender systems applied to Software Engineering. In the related work presented in Chapter 6 precision varied from 6.00% to 64.60%, while recall varied from 3.00% to 73.20%.

As we previously mentioned, the Exp-manager had the better performance with respect to the strict precision and the strict recall metrics. This possibly happened because this allocation system combines the profile that aggregates more information about the testers plus the workload information from the manager: Expertise + Manager-based.

Concerning the metrics approximate precision and approximate recall there was no winner: the allocation systems showed to be similar among themselves. It is important to emphasise that the results from approximate precision and approximate recall are more meaningful as they better capture the reality of the context in which this work was developed: the test managers do not allocate test cases in a strict way but in an approximate way (in the real world, a given tester can always be replaced by another tester with similar experience).

As all allocation systems have similar performance, we developed the following guideline for helping the managers to decide which allocation system to choose. The guideline is a set of context dependent heuristics. For choosing the profile, “time” and “quality” are key issues to be considered. If the manager needs to speed up a particular test execution, then the “Expertise” profile is the best choice as experienced testers run the test cases faster. On the other hand, if quality is important, then the “Effectiveness” profile would fit better. If a combination of both (time and quality) is needed, then the manager can use the “Multi-objective” profile and assign a different weight for “Expertise” and “Effectiveness”. Choosing between Blind or Manager-based depends on whether the manager already knows the workload of the testers. In summary:

- **Expertise:** if execution speed is crucial.
- **Effectiveness:** if defect finding is crucial.
- **Multi-objective:** if the manager wants a combination of both speed and defect finding.
- **Blind:** if there is no workload restriction.
- **Manager-based:** if the manager already knows the workload of the testers.

There are some concerns that need to be raised with respect to our evaluation considering the manager’s choices as golden standard for comparing the results obtained by our allocation systems. Works related to recommender systems, or other techniques from the Artificial Intelligence domain, frequently compare the performance of their algorithms to a human being as those systems typically attempt to mimic the human being choices. This is a very well-established practise. In our case, however, we can question ourselves if the human being (the manager) is, in fact, a good parameter to be imitated, or if the managers are in fact *a standard to be overcome*.

Ideally, we should had ran the 100 test plans recommended by our algorithms manually, and collect the total execution time and the amount of defects found for each test plan for further comparison. This would have given us enough inputs to conclude whether or not the algorithms assign test cases better than the human being (the manager). Unfortunately, this experiment would consume way too much resources (time, testers, managers, and wages) than we had. To perform such experiment, we would need to execute the results from the 6 allocation systems (600 test plans), plus the 100 test plans that were manually allocated by the manager (so we could collect the execution times and defects found in all test plans). This would result

in 71,134 test cases (if we consider the same test plans that were used in our experiment). Of course we could reduce this effort by taking advantage of real test plans being performed (thus we would not count the test plans allocated by the manager as an extra effort as they would have to be carried out anyway). Besides that, we could analyse the intersection between the test plan manually allocated by the test manager and the one suggested by the allocation system, and rerun only the difference. Nevertheless, this would still require too much of our resources. Section 7.1 describes a possible experiment that could capture the quality of the manager's allocation in a cheaper way.

Independently from having carried out the previously described experiment, our results have already shown that our tool, if it is not as good as the manager, can achieve similar results in most of the cases. Moreover, it automates part of the job that is currently being manually performed. At least good suggestions can be given to the managers to be used as a starting point of their allocations; new managers can benefit even more from our tool as it can provide them with some knowledge of test cases, testers and previous allocations.

Although in this work we chose to apply recommender systems to automatically assign test cases to testers, the problem reported here may be tackled by different approaches/techniques. In the context of TTC, for example, if we knew the amount of time each tester takes to run a given test case (TTC's database does not store this information), we could have modelled the allocation problem as a **Linear Programming Minimization problem**. Linear programming [DT97], or linear optimization, is a mathematical method for determining a way to achieve the best outcome (such as maximum profit or lowest time) in a given mathematical model for some list of requirements represented as linear relationships. This is just one example of an alternative approach and, of course, other alternatives may be applied depending on how a given test company handles the test allocation activity.

7.1 Future Work

The following activities are suggested topics for future work:

- Integrate the *Test Case Recommender* with TTC's test management tool.
- Integrate our algorithms in a test management tool such as Testlink [Tes09]. Just like the proprietary test management tool used on TTC, Testlink also enables the allocation of test cases to testers.
- Improve the algorithms to allow the managers to allocate whole test suites instead of single test cases. Not all testing companies adopt the model in which the test manager allocates single test cases to testers. In some of them, the test manager assigns the tasks in a higher level, say by allocating whole test suites (or even a whole software product) to testers. This enhancement will make our tool more flexible to be used in testing companies that adopt different allocation approaches.
- To overcome the concerns related to the experiment reported in Chapter 5, we plan to perform a new experiment in which the managers are not the golden standard anymore

and even their allocations will be evaluated in terms of quality. We intend to use the equivalence groups that were defined by the managers (the same groups described in Section 5.2.4 that were used to calculate the approximate metrics) to associate a different amount of *points* to the testers based in the group in which they are classified. For example, a tester classified in the group of the most talented testers receives *3 points*, a tester classified in the group of the second most talented testers receives *2 points*, and a tester classified in the group of the third most talented testers receives *1 point*. Having done that, we would be able to calculate the *score* (the total amount of points) achieved by a given allocation. With the experiment reported on Chapter 5 we are able to evaluate how similar our allocation systems are compared to the manager. However, we can say nothing about the portion in which there are disagreements (different pairs allocated). The adoption of the score can help us to achieve conclusions such as “*Algorithm X is 65% similar to the manager; and in the 35% it is different, it allocates 50% more testers from the group of the most talented testers than the manager does*”. Notice that only the Blind systems and the random algorithm could be used to perform this experiment. As the Manager-based algorithm receives the amount of test cases to be allocated to each tester as input, its score would be always less than or equal to the manager, never greater. Of course this experiment also has some threats to its validity. For example, *does a high score really characterise a good allocation?* Once again we are back to the same situation: to answer this question we would need to run all the test plans and collect the execution time and amount of defects raised to evaluate whether or not the algorithms assign test cases better than the manager. Nevertheless, relying on groups of equivalent testers carefully created by managers seems to be more reasonable than relying on past allocations done (possibly) in a rush.

- Perform an experiment to measure the allocation time with and without the use of our tool. Although we believe that the introduction of a tool like ours will reduce the allocation time, we need to measure the gains in a controlled experiment. We reported that our tool takes no more than 30 seconds to recommend a test plan (considering an allocation input with 300 test cases and 10 testers) while the manager takes about 15 minutes (in the better case) to create the same test plan. The initial suggestion provided by our tool, however, can encourage the manager to think more about the allocation and, consequently, take more time on this task: it will probably be a better allocation, but may not be faster. To evaluate the gain (or loss) of manager’s time, we need to measure.

Bibliography

- [AHM06] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *Proceedings of the International Conference on Software Engineering*, pages 361–370, New York, NY, USA, 2006. ACM.
- [Bas07] Victor R. Basili. The role of controlled experiments in software engineering research. In *Proceedings of the 2006 international conference on Empirical software engineering issues: critical assessment and future directions*, pages 33–37, Berlin, Heidelberg, 2007. Springer-Verlag.
- [BCR94] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. Goal Question Metric Paradigm. In John J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 528–532. John Wiley & Sons, 1994.
- [BD00] H.H. Bock and E. Diday. *Analysis of symbolic data: exploratory methods for extracting statistical information from complex data*. Studies in classification, data analysis, and knowledge organization. Springer, 2000.
- [BdC10] Byron Leite Dantas Bezerra and Francisco Tenório de Carvalho. Symbolic data analysis tools for recommendation systems. *Knowledge and Information Systems*, pages 1–34, 2010. 10.1007/s10115-009-0282-3.
- [Bei90] Boris Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.
- [Bla02] Rex Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.
- [CBLW01] M. Claypool, D. Brown, P. Le, and M. Waseda. Inferring user interest. *Internet Computing, IEEE*, 5(6):32–39, nov/dec 2001.
- [CWHC06] Marcelo Cataldo, Patrick A. Wagstrom, James D. Herbsleb, and Kathleen M. Carley. Identification of coordination requirements: implications for the design of collaboration and awareness tools. In *Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, CSCW '06*, pages 353–362, New York, NY, USA, 2006. ACM.
- [Did03] Edwin Diday. An introduction to symbolic data analysis and the sodas software. *Journal of Symbolic Data Analysis*, 7:1723–5081, 2003.

- [Dow97] Mark Dowson. The ariane 5 software failure. *SIGSOFT Softw. Eng. Notes*, 22:84–, March 1997.
- [DT97] George B. Dantzig and Mukund N. Thapa. *Linear programming 1: introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1997.
- [FP09] Anna Formica and Elaheh Pourabbas. Content based similarity of geographic classes organized as partition hierarchies. *Knowledge and Information Systems*, 20:221–241, 2009. 10.1007/s10115-008-0177-8.
- [GPG10] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, pages 36–40, Cape Town, South Africa, 2010.
- [GVEB08] Dorothy Graham, Erik Van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Intl Thomson Business Pr, 2008.
- [HKTR04] Jonathan L. Herlocker, Joseph A. Konstan, Loren G. Terveen, and John T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Trans. Inf. Syst.*, 22:5–53, January 2004.
- [Hol04] Steve Holzner. *Eclipse*. O’Reilly Media, Inc, 2004.
- [IEE90] IEEE. IEEE Standard Glossary of Software Engineering Terminology, September 1990.
- [IEE98] IEEE. Ieee standard for software test documentation. *IEEE Std 829-1998*, page i, 1998.
- [JKZ09] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. Improving bug triage with bug tossing graphs. In Hans van Vliet and Valérie Issarny, editors, *Proceedings of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, pages 111–120. ACM, 2009.
- [JZFF10] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich. *Recommender Systems: An Introduction*. Cambridge University Press, 2010.
- [KFN99] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software, Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 1999.
- [KMM00] Mark Kantrowitz, Behrang Mohit, and Vibhu Mittal. Stemming and its effects on tfidf ranking (poster session). In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval, SIGIR ’00*, pages 357–359, New York, NY, USA, 2000. ACM.
- [Kon04] Joseph A. Konstan. Introduction to recommender systems: Algorithms and evaluation. *ACM Transactions on Information Systems*, 22(1):1–4, January 2004.

- [KRGA08] Sègla Kpodjedo, Filippo Ricca, Philippe Galinier, and Giuliano Antoniol. Not all classes are created equal: Toward a recommendation system for focusing testing. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, Atlanta, Georgia, USA, 2008.
- [LKB04] Mark Last, Abraham Kandel, and Horst Bunke. *Artificial Intelligence Methods In Software Testing*. World Scientific Publishing Company, 2004.
- [LMS10] Susan F. Henssonow Lambert M. Surhone, Mariam T. Tennoe. *Pydev*. Betascript Publishing, 2010.
- [LT93] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26:18–41, July 1993.
- [Lut06] Mark Lutz. *Programming Python*. O’Reilly Media, 2006.
- [LZ05a] Zhenmin Li and Yuanyuan Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*, pages 306–315. ACM, 2005.
- [LZ05b] V. Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 296–305. ACM, September 2005.
- [MA00] David W. McDonald and Mark S. Ackerman. Expertise recommender: a flexible recommendation system and architecture. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work, CSCW ’00*, pages 231–240, New York, NY, USA, 2000. ACM.
- [MH02] Audris Mockus and James D. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering*, pages 503–512, New York, NY, USA, 2002. ACM.
- [MIM10] Breno Miranda, Juliano Iyoda, and S. R. L. Meira. Test case recommender: um sistema de recomendação para alocação automática de testes baseada no perfil do testador. In *Proceedings of the IV Brazilian Workshop on Systematic and Automated Software Testing, SAST 2010*, pages 57–66, Natal, Rio Grande do Norte, Brazil, 2010.
- [MM07] Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *MSR*, page 5. IEEE Computer Society, 2007.
- [MRS08] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

- [MS04] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.
- [MS10] Prem Melville and Vikas Sindhvani. Recommender systems. In *Encyclopedia of Machine Learning*, pages 829–838. 2010.
- [Nat10] Mary Natrella. *NIST/SEMATECH e-Handbook of Statistical Methods*. NIST/SEMATECH, 2010.
- [PCSF08] C Pilato, Ben Collins-Sussman, and Brian Fitzpatrick. *Version Control with Subversion*. O’Reilly Media, Inc., 2 edition, 2008.
- [PdSRE10] Thais A. B. Pereira, Vinicius S. dos Santos, Bruno L. Ribeiro, and Glêdson Elias. A recommendation framework for allocating global software teams in software product line projects. In *Proceedings of the International Workshop on Recommendation Systems for Software Engineering*, pages 36–40, Cape Town, South Africa, 2010.
- [RP06] P. Ralph and J. Parsons. A framework for automatic online personalization. In *System Sciences, 2006. HICSS ’06. Proceedings of the 39th Annual Hawaii International Conference on*, volume 6, page 137b, jan. 2006.
- [SG09] Guy Shani and Asela Gunawardana. Evaluating recommendation systems. *Recommender Systems Handbook*, pages 1–41, 2009.
- [Som06] Ian Sommerville. *Software Engineering*. Addison Wesley, 8th edition, 2006.
- [Ste86] Michael A Stephens. *Tests based on edf statistics*, volume 68, pages 97–193. Marcel Dekker, Inc., 1986.
- [Tes09] TestLink Community. *Testlink User Manual (v 1.8)*, 2009. <http://www.teamst.org>.
- [Tic85] Walter F. Tichy. RCS: a system for version control. *Softw. Pract. Exper.*, 15:637–654, July 1985.
- [Tri09] Mario F. Triola. *Elementary Statistics*. Addison Wesley, 11th edition, 2009.
- [TWFL98] C. Reid Turner, Alexander L. Wolf, Alfonso Fuggetta, and Luigi Lavazza. Feature engineering. *Proceedings of the 9th International Workshop on Software Specification and Design*, page 192, 1998.
- [Ves03] Jennifer Vesperman. *Essential CVS*. O’Reilly Media, Inc., 2003.