AMERICAN UNIVERSITY OF BEIRUT

COMPONENT-BASED CROWDSOURCING IN SOFTWARE ENGINEERING

by
BILAL YEHYA ABI FARRAJ

A thesis
submitted in partial fulfillment of the requirements
for the degree of Master of Science
to the Department of Computer Science
of the Faculty of Arts and Sciences
at the American University of Beirut

Beirut, Lebanon

AMERICAN UNIVERSITY OF BEIRUT

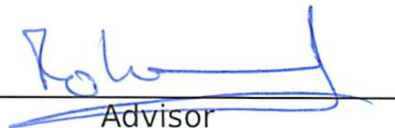
COMPONENT-BASED CROWDSOURCING IN

SOFTWARE ENGINEERING

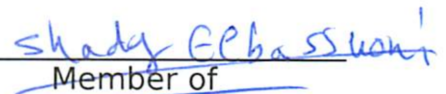
by

BILAL YEHYA ABI FARRAJ

Approved by:



Dr. Mohamad Jaber, Assistant Professor — Advisor
Computer Science



Dr. Shady Elbassuoni, Assistant Professor — Member of Committee
Computer Science

Dr. Mohamad El Baker Nassar, Assistant Professor — Member of Committee
Computer Science

Date of thesis defense: February 5, 2018

AMERICAN UNIVERSITY OF BEIRUT

THESIS, DISSERTATION, PROJECT RELEASE FORM

Student Name: Abi Farraj (Last) Bilal (First) Yehya (Middle)

☑ Master's Thesis ○ Master's Project ○ Doctoral Dissertation

☑ I authorize the American University of Beirut to: (a) reproduce hard or electronic copies of my thesis, dissertation, or project; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes.

☐ I authorize the American University of Beirut, to: (a) reproduce hard or electronic copies of it; (b) include such copies in the archives and digital repositories of the University; and (c) make freely available such copies to third parties for research or educational purposes
after : **One ---- year from the date of submission of my thesis, dissertation, or project.**
Two ---- years from the date of submission of my thesis, dissertation, or project.
Three ---- years from the date of submission of my thesis, dissertation, or project.

Signature Date

This form is signed when submitting the thesis, dissertation, or project to the University Libraries

ACKNOWLEDGEMENTS

I would first like to thank my thesis advisor Dr. Mohamad Jaber of the Computer Science Department at the American University of Beirut. Before being my advisor, Dr. Jaber gave me my first computer science course ever, and introduced me to this new world I live in today. Throughout both undergrad and grad schools, Dr. Jaber guided me to opportunities I would not have taken without him. He has always been a great mentor motivating me to successfully fulfill my graduation requirements by writing this paper.

I would also like to thank my co-advisor Dr. Shady Elbassuoni who opened the door for this research by introducing me to the topic and guided me on the way up to this stage. Moreover, I would like to express my gratitude to Dr. Mohamad El Baker Nassar for reviewing my thesis and for the input and comments he gave me, acting as a member of the thesis committee.

Finally, I must express my gratitude to my family for always being by my side and supporting me, through the process of researching and writing this thesis. Thank you.

AN ABSTRACT OF THE THESIS OF

Bilal Yehya Abi Farraj for Master of Science
Major: Computer Science

Title: Component-Based Crowdsourcing in Software Engineering

Crowdsourcing in software engineering is a fast-growing and promising area, yet it still lacks a rigorous platform that circumvents the well-known issues such as task decomposition and composition, scheduling, coordination, quality, payment and time-to-delivery. This thesis introduces novel methods and tools for automating the crowdsourcing process of a software project, following the component-based paradigm. We mainly exploit the abilities of a professional crowd on a user-friendly and easy-to-use environment, to design, verify, code, test and then assemble various high-quality software components. The procedure starts with a request consisting of a project idea or a high-level specification on our platform. Then, the request is hierarchically developed with the help of the crowd. At each phase, the crowd competes to decompose a specification into (1) components with well-defined interfaces and (2) a glue that defines the corresponding composition operator between components. This phase is repeated until we reach atomic components that are ready for development. Additionally, the crowd competes to develop atomic components and to compose/integrate components. Our platform provides a fair and rigorous payment that can be adapted according to the requester needs as well as an intelligent rating system that mimics the crowds performance. We evaluate our approach on non-trivial case studies and compare it to our main competitors: (1) software companies; (2) freelancers; (3) state-of-the-art software crowdsourcing platforms. Experimental results show the effectiveness of our platform with respect to cost, time-to-deliver, fairness, reusability, and quality.

Contents

ACKNOWLEDGEMENTS v

ABSTRACT vi

LIST OF FIGURES ix

LIST OF TABLES x

1 INTRODUCTION 1

2 PROBLEM STATEMENT DESCRIPTION AND FORMALIZATION . . . 5

3 CROWDSOURCING PLATFORM DESIGN 7
3.1 The Crowd 7
3.2 The Projects 10
3.3 The Jobs 11

4 COMPONENTS AND CBSE 14
4.1 Component-Based Software Engineering- (CBSE) 14
4.2 CBSE on the Platform 16
4.3 Challenges and Solutions 20
4.3.1 Challenge 1: Design Refinement 20
4.3.2 Challenge 2: Code Composition 24
4.3.3 Challenge 3: Component Verification/Testing 25
4.3.3.1 User Rating 25
4.3.3.2 Submissions Rating 27
4.3.4 Challenges 3 and 4: Component Pricing and Winners' Payments 27
4.4 On-Platform Challenges 30

5 CROWDSOURCED PROJECT BUILDING WORKFLOW 32
5.1 Design Phase 32
5.2 Development Phase 34
5.3 Integration Phase 35
5.4 Miscellaneous 35
5.4.1 The importance of user ratings 35
5.4.2 Payment fairness 36
5.4.3 Maintenance and bug hunts 37
5.4.4 Automation and Copilot's Absence 38

6 IMPLEMENTATION 39
6.1 Design and Architecture 39
6.2 Requester's Perspective 41
6.3 Worker's Perspective 44
6.4 Project building process 45

7 CASE STUDIES AND BENCHMARKS 49
7.1 Measurables 49
7.2 The Sample Projects 50
7.3 Experimental Observations And Results 52
7.3.1 Observations 52
7.3.2 Experimental Results 53

8 RELATED WORK 59
8.1 Full-task Crowdsourcing 60
8.2 Stage-specific Crowdsourcing 60
8.2.1 Crowdsourcing software requirements extraction 60
8.2.2 Crowdsourcing for Software Design 62
8.2.3 Crowdsourcing for Software Coding 63
8.2.4 Crowdsourcing for Software Testing and Verification 63
8.2.5 Crowdsourcing for Software Evolution and Maintenance 64
8.3 Research work in crowdsourcing softwares 65
8.4 Comparing existing platforms to ours 65

9 CONCLUSION AND FUTURE WORK 68
9.1 Conclusion 68
9.2 Future Work 69

List of Figures

1.1 Project Decomposition . . . 3

4.1 Black Box Code . . . 16
4.2 Components Pool . . . 17
4.3 Component Decomposition . . . 22

5.1 Pre-development graph . . . 34
5.2 Post-development graph . . . 35

6.1 MVC design pattern [1] . . . 40
6.2 Options Tabs . . . 41
6.3 Project Creation View . . . 42
6.4 Project Details . . . 43
6.5 Design Job Details . . . 43
6.6 My Projects . . . 44
6.7 Design Search . . . 45
6.8 Development Job Search . . . 45
6.9 Assembly Job Search . . . 46
6.10 Testing Job Search . . . 46
6.11 A Simple Hierarchy . . . 47
6.12 A Simple Hierarchy- Full . . . 48

7.1 Number of submissions versus job hardship . . . 56
7.2 Quality of submissions versus job hardship . . . 57
7.3 Number of submissions versus job level . . . 57
7.4 Quality of submissions versus job level . . . 58

List of Tables

3.1 Duties of the Crowd 10

7.1 CPISE versus software company versus freelancer 54
7.2 Component-based versus single-component 55

Chapter 1

INTRODUCTION

Contents

"Crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of open call" [2]

Crowdsourced jobs are usually relatively small, requiring little time and effort, in a way the worker can finish as many jobs as possible and make as much money as possible. Examples of crowdsourced jobs are data annotation, essay editing/writing, answering questionnaires, review services. People seek posting jobs on crowdsourcing platforms due to: the diversity of solutions, readily available human workforce, cost reduction, faster time-to-market, higher quality through broad participation, creativity and open innovation [3]. Job owners assign an amount of money, either for whoever participates in the job, or only for those who do it correctly. Payments on crowdsourced jobs are usually in terms of cents up to a few dollars and might require some or no expertise. Before working on tough tasks, some platforms have the worker undergo training or a simple quiz to make sure of his/her ability to do the job correctly. Generally, the harder the task given is, the bigger the payment gets, and the tougher joining it becomes. Example of crowdsourcing platforms used in software-related fields [2] are:

- Amazon Mechanical Turk [4]: Mainly used for program synthesis, GUI Testing, oracle problem mitigation, program verification
- uTest [5]: Mainly used for functional testing, usability testing, localization testing and load testing.

- AppStori [6]: A mobile crowdsourcing platform that is based on a crowdfunding model
- TopCoder [7]:A contest-based software development

Software engineering and development in practice are different from the“micro-jobs” done on crowdsourcing platforms. This is mostly because the tasks are harder, requiring more time and a higher level of expertise. But despite this fact, crowdsourcing in software engineering do exist, starting with *TopCoder* since 2001. In general, crowdsourcing in SE is divided into the following main sections [2]:

- Study of practice
- Theories and models
- Evaluations ion SE research
- Applications to SE:
    1. Requirements
    2. Design
    3. Coding
    4. Testing
    5. Verification
    6. Other

Existing platforms are usually specialized in one or a few of the mentioned sections, and each does its job differently. The main objective of the thesis is to introduce a new platform that overcomes the well-known issues of the aforementioned platforms, such as task decomposition and composition.

Our platform tweaks two software engineering cycle models: the waterfall model and the agile model, in a way that would fit the crowdsourcing paradigm. From

the waterfall model, we take the basic work-flow: requirement analysis, design, implementation, testing, acceptance testing, production and finally modification and maintenance. As for the agile model, we adapt its ability to scale well and the idea of iterations. Both models are needed to create a well structured framework that is able to exploit the abilities of the crowd to a maximum. The main functionalities are decomposed in order to parallelize all of the cycle stages. Hence, projects get delivered in the shortest time possible.

Moreover, we offer an automized system, with a set of specific rules moving away from the human "copilot"[2]. In a standard crowdsourcing platform, a copilot acts as a manager. For instance, a copilot in TopCoder is responsible running challenges, communicating with the crowd, and posting periodic status reports. In order to automatically manage projects of a large dimension, we use the component-based approach to fine-grain their requirements and deal with each one individually. We treat each set of sub-requirements as a project by itself, and perform the same procedure for an undefined number of iterations until the project is accomplished.

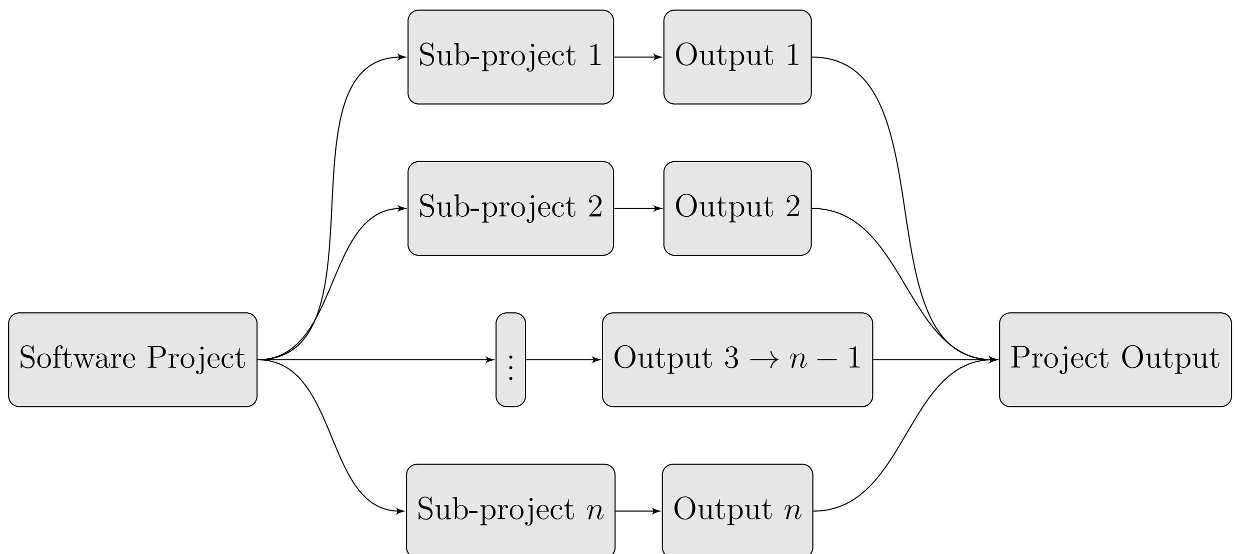


Figure 1.1: Project Decomposition

The proposed platform does not use crowdsourcing in software engineering as a mass freelancing technique, but it uses the power of the crowd to properly disinte-

grate the once huge and expensive software project into several, much smaller fine-grained software particles, allowing the developers to implement reasonably sized pieces of code per job. This approach makes testing the software particles easier due to their small sizes, and increases the parallelism on the platform, thus maintaining high quality and vastly improving the time-to-delivery *(TTD)*. Figure 1.1 shows one level of refinement, at which a project is decomposed into n sub-projects. Each sub-project is targeted separately and is the crowd is expected to produce a solution for it. Outputs are then reduced into one final output representing the solution.

In the following chapters we define the building blocks of our system, identify the methods used, describe the designed workflow, demonstrate the platform built, and finally analyze the test results we obtained. In the process, we compare our approach to others whilst illustrating where and how our method is superior.

Chapter 2

PROBLEM STATEMENT DESCRIPTION AND FORMALIZATION

Contents

Creating a software from a non-formal idea is generally a very challenging task, especially for people who are less experienced in software development. One can choose between either hiring a software company, or a freelancer to do the job, which may introduce the following trade-offs:

1. High cost but guaranteed quality when handing the project to a well-reputed software company.

2. Low cost but delayed delivery and untrusted quality of a freelancer's work.

Residing to a third untraditional option, leads to posting the project on a crowd-sourcing platform in the hope of receiving a diversity of solutions. Looking into existing platforms showed promising features that could be utilized while designing a new platform, but also showed downsides that must be overcame:

1. Software jobs are usually cumbersome, and workers do not like to take the risk of not getting paid after investing too much time in them. Since a submission to such jobs might not win, resulting in the worker's demotivation.

2. Unfair competition for workers often exists. More experienced workers are allowed to register for a job earlier than others, and thus giving them a higher chance of gaining money. Less experienced workers thus suffer from unfair treatment and competition.

3. Dependency on human "copilots" to manage the development process. Those copilots might unexpectedly leave the project, or maybe mismanage it, leading to the project's failure or delivery delay.

4. Platforms are usually specialized in a certain domain, resulting in limited job options, such as UI design, testing, mobile app development.

5. The work on a software job is done as a whole by one person, defying the idea of crowdsourcing concerning the "wisdom of the crowd".

In order to overcome the stated drawbacks and benefit from the added value of crowdsourcing, we introduce a novel method that combines the quality of a program made by a software company, with a cost close to that of a freelancer, and a fast *TTD*. It allows the requester to initiate his/her project and view the progress made on it in a very simple way, without having any background in software. The workers, on the other hand, benefit from a wide variety of job options, due to the language-independent and platform-independent projects to be posted. Workers should also find fair payments for the "correct" jobs they submitted, even if their submissions were not chosen as winners. Dealing with the copilot issue, we suggest a software-driven automated copilot that operates based on a predefined workflow.

Chapter 3

CROWDSOURCING PLATFORM DESIGN

Contents

3.1 The Crowd 7

3.2 The Projects 10

3.3 The Jobs 11

In the broad concept, our platform shares its main components with other crowd-sourcing platforms, but in details, it is divided into three main sections:

1. The crowd
2. The projects
3. The jobs

3.1 The Crowd

The crowd, is the group of users signed up to the platform and they are classified into two parties:

1. **Requesters**: Are people with no necessary technical background. Their job is to post a project request and details about it, to facilitate the software building process. The requester can supervise and manage the projects created by him/her, and view their progress.

2. **Workers**: Those are the users with experience, who are willing to work on as many jobs as possible in order to get paid per job. A worker is defined by the type of job he/she would perform on the platform, and thus can be a: designer, developer, assembler, tester. Refer to table 3.1 to see the different workers and their duties.

Worker	Duties
Designer	(a) Submits refined designs/requirements based on the requester's needs (b) The new requirements are supposed to be categorized, such that developers can work on each category or group of sub-requirements independently from the other (c) The designer should also attach tips for integration, if needed (d) If the requirements are simple enough for development, the designer sets the job to be *"final"*. A final job can be developed in parallel to the design process. (e) Create and attach the necessary test cases for the development jobs. The test cases should be split into two parts: one for the developers and one for the testers. (f) Set the following for each job: due date, budget, and programming language (g) Generally, gets paid the most due to the difficulty of the job

Developer	(a) Implements the final output specifications of the design stage i.e. the reads the specifications of the *"final"* jobs and code implement them. (b) Run the test cases attached by the designers on the codes to be submitted and make sure they all work (c) Stick to the programming language(s) set by the designer
Assemblers	(a) Read the related development job specifications (b) Integrate the related implementations based on the interfaces and the tips attached by the designer (c) Make sure that the integrated code runs properly on the test cases attached by the designer
Testers	(a) In case of a design job: study the designs submitted and the budget decomposition per design (b) In case of a development job: run a different set of test cases attached by the designer and make sure the codes match the requirements (c) Rate the submissions (0-5) (d) Leave a comment in case the submissions need editing

All	(a) Have access to all the jobs on the platform as long as the worker has a rating higher than the required to join (b) Have no limit on the number of jobs to participate in (c) Are free to request a project or work on any job type (d) Get to know the payment for the job before joining it, and will get paid if their submission gets highly rated (e) Their rating will adjust based on every submission made, whether correct or wrong

Table 3.1: Duties of the Crowd

3.2 The Projects

A project can be a general non-formally defined idea posted by an inexperienced person, leaving the full design process for the crowd; or it can be a set of well-defined requirements posted by an experienced person or organization aiming for a better design for their proposed software, added to the development part. Moreover, the project has the following details, all set by the requester:

- The budget planned to be spent on the project
- A due-date
- Programming language(s) desired
- Number of winning submissions to be paid
- Minimum rating a worker should have to join the project building process

- Any document(s) that might help the designers and developers

Projects might vary in requirements, scale, programming language(s), platform, technology, and needed crowd expertise. The design of the platform allows the creation of any software, under the assumption that the crowd is large enough and have experienced members in all the software domains. Another assumption made, is that the number of submissions by the crowd will always be enough to satisfy the needs of the requester, within the time limits set.

The flexibility of the platform and the diversity of the potential projects to be posted on it, makes it different from other platforms as follows:

1. Attracts a wide range of requesters in very different domains such as software development, GUI design, phone apps development, web development, computation, code optimization, software maintenance.

2. The categorization of requirements done by the designers can give the workers a chance to work on several parts of the same project at a time and thus make more money. Workers have the complete freedom to participate in as many project as they want.

3. The diversity of projects to be posted boosts the workers' interest in the platform.

4. There is no limit on the project sizes.

5. Workers' interest in the project is highly dependent on the budget set by the requester. Thus the requester is responsible for the feasibility of his/her project idea.

3.3 The Jobs

One of the main downsides of some other software crowdsourcing platforms is the big and often uninteresting jobs posted. Generally, the bigger the job is, the higher the

risk of a worker not getting paid is, since a lot of time is invested to solve large-scale problems, and the worker might end up not getting paid even if the work submitted is correct. Therefore, we decided to treat each project on the platform as a *job*, where this job can be decomposed into several smaller jobs and each gets handled independently. The new jobs can also be decomposed an unrestricted number of times based on the need and the consent of the crowd. If the job requirements are simple enough to be handled by one person, we call it a *final* job.

Therefore, a project P is decomposed recursively into sub-jobs. Then umber of sub-jobs is unknown on project creation and might vary between one project and another.

Jobs are related in a parent-child relationship, a child cannot exist without the parent, and only gets added to the project if needed. Dealing with a child-job is of course simpler than dealing with the parent, due to the smaller scale, improving by that the manageability of software entities.

Similar to the waterfall model, our process goes as follows:

1. Design
2. Verification
3. Development
4. Unit testing
5. Integration
6. Integration testing
7. Delivery
8. Maintenance

Based on the aforementioned, our platform has four types of jobs: *design*, *development*, *assembly* and *testing/ verification*. Our workflow is not linear though, since

we mimic the agile model by parellizing the work and running in several iterations. Any combination of jobs can co-exist in the project building process, due to the variable difficulty levels among jobs. As an example: two sibling jobs might have very different requirements, so that one is *final* and the other is not. The *final* job is ready to be developed and tested, whereas the other is still in the design phase. This decomposition of work exploits the crowd power, allows a very fast *TTD*, and encourages the workers to join more jobs due to the little time they need, compared to projects on other platforms.

Each of the job types on the platform suits a worker type and thus offering a wide range of job opportunities. The same worker can participate in the design, implementation, integration and testing of the same project with only one restriction: a worker cannot test and rate his/her own submissions.

When a job is available, workers can join it and submit their work to it. All the submissions made to the platform get tested for correctness and rated by other workers of the crowd. After the due date of a job is reached, the submissions with the highest ratings win and get paid. The same rule is followed in case of a design jobs, development jobs and assembly jobs.

Chapter 4

COMPONENTS AND CBSE

Contents

4.1 Component-Based Software Engineering- (CBSE) 14
4.2 CBSE on the Platform 16
4.3 Challenges and Solutions 20
4.3.1 Challenge 1: Design Refinement 20
4.3.2 Challenge 2: Code Composition 24
4.3.3 Challenge 3: Component Verification/Testing 25
4.3.3.1 User Rating 25
4.3.3.2 Submissions Rating 27
4.3.4 Challenges 3 and 4: Component Pricing and Winners' Payments 27
4.4 On-Platform Challenges 30

4.1 Component-Based Software Engineering- (CBSE)

Moving from less formal general requirements to well defined formal ones, require refinement and categorization of requirements. The refined requirements could each define components, in a component-based system. Components sharing the same *"parent"* or same requirements origin must be connected to each other by the means of interfaces. An interface links any two related components or glues them in several ways to avoid any mismatch [8]:

- *Parameterized Interface:* Makes it possible to change the component properties by specifying parameters that are the parts of the component interface. Such parameters can be: memory allocation, number of input data· · ·

- *Wrapper:* A special type of glue-code that encapsulates a component and provides a new interface that either restrict or extend the original interface, or to add or ensure particular properties.

- *Adapter*: An adapter is a glue code that modifies ("adapts") the component interface to make it compatible with the interface of another component. The intention of an adapter is not to hide or modify the component properties, but to adjust the interfaces.

As explained in [9], CBSE gives developers the chance to reuse existing pieces of code and not redo the whole job from scratch. It requires focus on system specs and development added to additional consideration for the overall system context. Individual components necessitates consideration on the level of: properties, acquisition, and integration.

An individual software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be deployed independently and is a subject to third parties.

Component-based development (CBD) is a section of CBSE, and it is defined by the following:

- Searching and identifying components based on preliminary assessment

- Selecting components based on stakeholder requirements

- Integrating and assembling the selected components

- Updating the system as components evolve over time with newer versions

Using CBD comes with the following advantages:

- Reduce development time

- Increased flexibility

- Reduced process risk
- Enhanced quality
- Low maintenance
- Standardization

4.2 CBSE on the Platform

Starting from the fact that several components can be integrated to form a fully functional software, we established the idea of using component-based structures as the basis for our designed platform. The implementation of such components is done by the crowd, thus each component needs to be available for development through some job. Therefore, each job J on the platform is associated with a component C, so a project P having n jobs, also has n parallel components. Every component posted for development would receive a number of implementations; all the *correct* implementations would accept the same input and are expected to produce the same output. Assuming that all the submitted codes are *correct*, we treat each one as a *black boxes*: codes will differ just by their time and space complexities if they have no visual aspect.

Figure 4.1: Black Box Code

For example a certain component might require a custom complex algorithm:

1. Worker 1 submits implementation I_1 having time complexity $= O(n^2)$
2. Worker 2 submits implementation I_2 having time complexity $= O(nlog(n))$

Both I_1 and I_2 accept the same input and produce the same outputs, but I_2 does it efficiently, thus we select it to be the implementation of our component. If at some

point in time, an implementation I_3 gets implemented and is more optimized than I_2, the component's implementation could be simply be replaced by I_3.

Adapting the component-based approach on a crowdsourcing platform also makes sense in the following ways:

- Components can be very small in size, thus require little time to implement and test, and also favors the micro-job concept
- Many implementations will be received for the same component, giving us the chance to choose the best and thus boost the quality
- The number of components that would be needed to build one project should be big and thus offering a bigger number of job opportunities
- Facilitates code maintenance and optimization
- Allows future code reuse
- Integrating two or more components will result in a new component, which could be configured based on different permutations of the sub-components it is made of. For example in figure 4.2, we have three implementations for the first component and two for the second, therefore, we get six possible permutations and thus six different compositions.

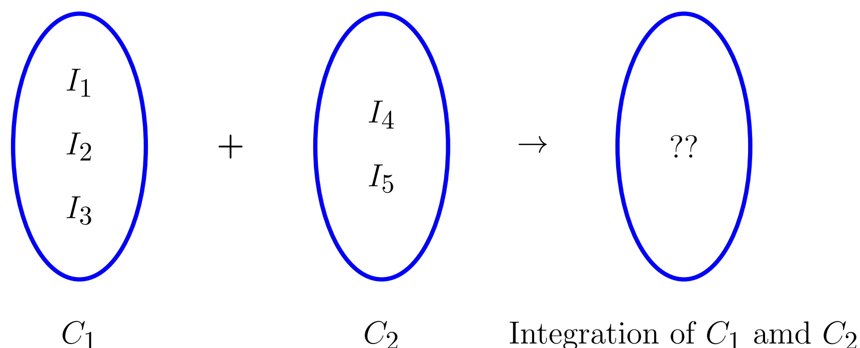


Figure 4.2: Components Pool

Figure 4.2 shows the post-development stage at which integration happens, but prior to this stage, requirements must be defined for both C_1 and C_2. The two sets of requirements are the output of the design stage, where the designer decided on a specific refinement that led to the creation of the two components. Reaching a component-based design is not as straight forward, especially when being handed to a crowd. Therefore, using this approach in building the platform introduced some challenges over the challenges faced in a traditional crowdsourcing platform:

- *Design refinement without losing integrity:*

  Each complex component will have its requirement decomposed into several sub-requirements to facilitate the development process and improve the gains of using the CBSE. This decomposition, if done wrong, might ruin the integrity of the whole system or might cause the loss of requirements due to human error. To be a designer, the worker should have experience in the domain and thus be careful while creating the sub-components.

- *Code composition based on requirements and interfaces:*

  Just like the design refinement stage, requirements might end up not being implemented due to human error. Its the designer's job to attach the necessary requirements, interfaces and tips for the designer to base the implementation on. The developer should also pay attention to all the posted details, and test the code using the cases posted by the designer. Testers should not only run the codes submitted, but also examine them for any missing points.

- *Unit testing and integration testing:*

  A component might work fine on its own but when integrated it might fail and thus causing a problem, that could be resolved by choosing a different implementation. Another problem might be arbitrary rating by the testers; we attempt to solve this problem by using weighted ratings.

- *Unfair component pricing:*

  Since the sub-components are priced by the designer, based on the budget set by the requester, some components might be either underpriced or overpriced. This could be due to:

  1. The poor judgment of the designer
  2. The low budget set by the requester
  3. A possible run out of budget due to excessive design refinement

- *Submissions ranking:*

  Submissions need to be rated by testers and given a $(0-5)$ grade. The crowd of testers might highly rank a certain submission, although its not the best. The rules of the platform dictate that the submission with the highest rate gets chosen as a winner, regardless of how it compares to other submissions. Weighted ratings act as a good safety net, to avoid random ratings by spam users and maintain a good quality of submissions.

- *Winner's payments:*

  Paying a winner as soon as the job is closed seems to have some risk in it due to the problem mentioned above: the component might be functional by itself, but does not fit the project properly. This problem might be introduced after the winner is paid, and thus losing the chance of a refund. To fix that we proposed a certain percentage as a security fee per job, as a portion of the prize. This security fee will not be paid for the winners until the whole project is over. In this way, contacting the worker with a faulty component would be easier.

- *Cost estimation:*

  At this stage, a requester estimates the budget needed for the project to be done, and sets it for the crowd. If we were to estimate the budget needed

based on the number of components to be used, we would have to estimate the number of components as well. Going in this direction would suggest the need for machine learning techniques, which we will leave for a future work.

4.3 Challenges and Solutions

4.3.1 Challenge 1: Design Refinement

Starting with the first and most important challenge, we introduce our first goal: retrieving the general system specification from the requester and refining it into simpler sub-specifications, where each sub-specification gets dealt with independently. Each component C on the platform thus has a certain specification $\mathcal{S}$, such that:

Definition 1. *Specification: A (sub)specification $\mathcal{S}$ is defined by a formal or narrative description of the desired system with optional interfaces used to compose specifications. Give a specification $\mathcal{S}$, we define the following two methods*

- *Design method:* $\texttt{design}(\mathcal{S}) = \bigoplus\{S_i\}_{i\in I}$*, which defines the set of sub specifications* $\{S_i\}_{i\in I}$ *and their corresponding composition operator* $\bigoplus$ *(w.r.t. interfaces of sub specifications). If* $\texttt{design}(\mathcal{S}) = \emptyset$*, we consider that* $\mathcal{S}$ *is an atomic specification that cannot be decomposed further.*

- *Implementation method: the implementation of a specification is defined by the* `code` *method as follows.*

$$\texttt{code}(\mathcal{S}) = \begin{cases} \textit{Implementation of } \mathcal{S} & \textit{If } \mathcal{S} \textit{ is atomic} \\ \bigoplus\{\texttt{code}(S_i)\}_{i\in I} & \textit{Otherwise} \end{cases}$$

```
design(Specification s) {
   if(s is not atomic) {
      lunch a design CrowdSource job of s;
      s.design = winningDesign(s);
```

```
    for(Specification si: s.design.specs) {
      CrowdSource(si);
    }
  }
}
```

Listing 4.1: Design Method

```
code(Specification s) {
  if(s is atomic) {
    lunch an implementation job of s;
    s.code = winningCode(s);
  } else { // s is not atomic
    if(s.design ≠ ∅ ∧ ∀si ∈ s.design.specs : si.code ≠ ∅) {
      lunch ⊕ compose implementation job of s;
      s.code = winningCode(s);
    }
  }
}
```

Listing 4.2: Code Method

Based on the above, when a design job is available on the platform, the crowd members compete against each other to submit the best component-based design for the component associated with the job:

1. *Analyze* the job requirements i.e. the component's specification
2. *Study* the possible ways to categorize the specifications by: topic, programming language, difficulty level, design logic, and so on.
3. *Create* as many sub-components as needed, one for each categorization. Other users must be able to read the requirements of the sub-components and work independently on them, without any need to refer to other components.

4. *Ensure* that the compositions of the sub-components will result in the component itself. The designer should make sure that no requirements go missing during the refinement process.

5. *Specify* the budget needed for each new component.

6. *Assign* the necessary programming language(s) needed to code each component.

Each refinement process creates a hierarchy where the root node is the parent component, and the leaf nodes are the sub-components:

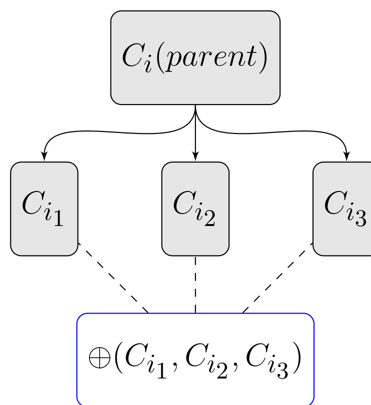


Figure 4.3: Component Decomposition

In figure 4.3 we notice the following: $\texttt{design}(\mathcal{S}_i) = \oplus(\mathcal{S}_{i_1}, \mathcal{S}_{i_2}, \mathcal{S}_{i_3})$. This illustrates the output of one design phase, but the whole refinement process should be applied on all non-atomic components *top-down*, and is described by the following method:

- *Refinement method: Recursively refine components in the hierarchy. A single refinement phase is taking any component C having a non-atomic specification S, and calling on the* `design` *method on S. Then calling on the method* `refine` *on all the sub-components of C. The base case is reached when C has an atomic S.*

$$
\texttt{refine}(\mathcal{C}) = \begin{cases} \textit{return} & \text{If } \mathcal{S} \text{ corresponding} \\ \text{(Do not further refine)} & \text{to C is atomic} \\ & \\ \texttt{design}(\mathcal{C}.\mathcal{S}) & \text{Otherwise} \\ \texttt{refine}(\mathcal{C}_i)_{i \in I} & \end{cases}
$$

```
refine(Component c) {
   if(c.S is not atomic) {
      design(c.S); // the output is a list of sub-components
      for(Component ci: c.subcomponents) {
         refine(ci);
      }
   } else { // c.S is atomic
      return; //no refinement needed
   }
}
```

Listing 4.3: Refine Method

The previous method necessitates the two following definitions on the type of components:

Definition 2. *Compound Components: Each C is considered a compound component if its specification can be further decomposed ($C.\mathcal{S}$ is non-atomic)*

Definition 3. *Base Components: A component C is called a base component when its corresponding specification $\mathcal{S}$ is atomic. In this case C requires no further decomposition and is ready for development $\Rightarrow$* $\texttt{code}(C.\mathcal{S})$ *can be applied on it.*

4.3.2 Challenge 2: Code Composition

Now that we have implementations for all base components, we address the second challenge: code composition. Referring to figure 4.3, and assuming that the sub-components $C_{1,\ 2,\ 3}$ are all base components, having $C_{1,\ 2,\ 3}.code \neq \phi$, but $C.code = \phi$. Thus we introduce a new method, similar to the `refine` method, but goes bottom-up instead of top-down.

- *Integrate method: Recursively go over each compound component C and integrate the codes of its children, based on the specification of C and its interfaces. If at least one of the components does not have its code ready, wait for it.*

$$\texttt{integrate}(\mathcal{C}) = \begin{cases} \text{Merge the codes of C's children and assign the output code to C} & \text{If all } C_i's \text{ have } \texttt{code}(C_i) \neq \phi \\ \texttt{integrate}(Parent(\mathcal{C})) & \text{If } C \neq root \end{cases}$$

```
integrate(Component c) {
  if(c is compound and c.code = φ) {

    for(Component ci: c.subcomponents) {
      if(ci.code = φ){
        return;
      }
    }

    lunch an implementation job of c;

    if(c ≠ root){ //root has no parent
```

```
        integrate(c.Parent);
      }
   }
}
```

Listing 4.4: Integrate Method

When `integrate`(*root*) = *final code*, the project finishes and is ready for delivery.

4.3.3 Challenge 3: Component Verification/Testing

Every submission to the platform gets tested by the crowd, starting by designs, then codes, then integrations. For this reason our testing method depends on the type of the job at which a submission is made: design testing is done by designers, code testing is done by developers and integration testing is done by assemblers. The added value given by the repetitive testing is of course quality assurance: each subcomponent is tested independently, then the composition is tested again to eliminate possible error. All submissions get rated and thus ranked, allowing the replacement of any faulty component implementation and thus improving maintainablity.

A submission's rating is done by the crowd, and the crowd might contain spammers or fakes who would set random ratings for submissions and thus negatively effecting the overall quality and effectiveness of the system. To conquer this challenge we define two interrelated rating systems: user rating and submission rating.

4.3.3.1 User Rating

A user's rating can increase or decrease based on several factors, and this rating effects the payments he/she gets, added to the jobs he/she can join on the platform:

- User rating has three independent values: R_{des}, R_{dev}, R_i standing for design, development, and integration ratings.

- All ratings are set to 2.5/5 upon sign up
- R_{des}, R_{dev}, and R_i vary based on the user's submissions (to be discussed below)
- The rating also vary based on the user's testing accuracy per job field. For example, the design rating of a worker will be adjusted, in case of a correct or wrong rating of a submission done to a design job.

Taking R to be any of R_{des}, R_{dev}, and R_i. Let $R = \alpha R_s + \beta R_t$ such that $(\alpha + \beta = 1)$ where R_s depends on every submission made to the platform by the worker, and R_t depends on the testing jobs performed by the worker, on a certain job type.

Calculating R_s:

User U has a set of n submissions $\{S_1, S_2, ..., S_n\}$, where each S_i has a rating $0 \leq r_i \leq 5$ and a binary value $a_i = 1$ if S_i was accepted and $a_i = 0$ otherwise. A user's rating is directly proportional to the average of his submission's ratings $r_{avg} = \frac{\sum_{i=1}^{n} r_i}{n}$.

If $a_i = 1$, rating r_i must be boosted by a certain percentage p_1 set by the platform, now: $r_{i_{new}} = (1 - p_1) r_{i_{old}} + 5(p_1)$

Else $r_{i_{new}} = r_{i_{old}}$

Therefor, $r_{avg} = \frac{\sum_{i=1}^{n} r_{i_{new}}}{n}$ can be used to calculate the new rating: $R_{s_{new}} = \frac{R_{s_{old}} + r_{avg}}{2}$. The ratio of accepted submissions over total submissions should also be included in the calculation of user rating based on a certain percentage p_2, therefore:

$$R_s = (1 - p_2) \frac{R_{s_{old}} + r_{avg}}{2} + p_2 \frac{S_{accepted}}{S_{total}}$$

Calculating R_t:

User U has a set of m verifications on m different submissions, where those submissions were made by other workers on a certain job type. A verification has four characteristics: (1) the user who made it, (2) the job at which it was made on, (3) the submission verified, and (4) the rating given. In order to calculate R_t we need to examine the validations of U, to see how accurate the ratings are: After a job J is closed and a winner is chosen, each submission to J has a final rating $r_{s_{i \in I}}$. We should now check if the ratings given by U for each s_i is within a certain threshold: $accuracy = |r_{s_{i-final}} - r_{s_{i-given}}| <= threshold$. R_t is thus simply an average of accurate ratings over the total number of verifications m: $R_t = \frac{\text{correct ratings}}{\text{all ratings}}$

4.3.3.2 Submissions Rating

Every submission made to the platform should be rated, as part of its verification/testing procedure. This rating is tightly connecting to that of the worker who is doing the verification. Submission ratings are calculated based on the weighted averaging of the workers' ratings

4.3.4 Challenges 3 and 4: Component Pricing and Winners' Payments

Given a specification $\mathcal{S}$, a user can specify a budget $\mathcal{B}$ to crowd-source its development. The budget is split into three phases: (1) design; (2) integration; and (3) development. Let $\mathtt{p_{des}}$, $\mathtt{p_{int}}$, $\mathtt{p_{dev}}$ denotes the percentage of the design, integration, and development of specification $\mathcal{S}$, respectively, where $\mathtt{p_{des}} + \mathtt{p_{int}} + \mathtt{p_{dev}} = 1$. The value of these percentages depends on the level and the hardship of the task (i.e., $\mathtt{p_{des}} = \mathtt{p_{des}}(\ell, h)$[1]). Clearly, when $\mathcal{S}$ is atomic then $\mathtt{p_{des}}$, $\mathtt{p_{int}}$, $\mathtt{p_{dev}}$ are equal to 0, 0, 1, respectively.

- *Design Budget*: $\mathtt{b_{des}}$ is the budget allocated to the wining(s) design, ratings of designs and a security fees, which is equal to $\mathtt{p_{des}} \times \mathcal{B}$. Note, a user can

1. This function can be embedded by the users, which can also return fixed percentages.

Algorithm 1 userRating

```
1: function getUserSubmissionsRating(U, S, p1, p2)
2:     n ← number of submissions
3:     Rs ← rating of user U's submissions
4:     if n = 0 then
5:         return 2.5
6:     else
7:         Sacc ← numberofacceptedsubmissions
8:         ravg ← getSubmissionsRatingAvg(S, p1)
9:         Rs ← (1 − p2)(R+ravg)/2 + p2 Sacc/n
10:        return Rs
11:
12: function getUserValidationsRating(U, V)
13:    n ← number of validations in V
14:    c ← correct ratings
15:    i ← 0
16:    for i < n do
17:        sub ← V(i).submission
18:        ri.given ← V(i).rating
19:        acc ← |ri.given − sub.rating|
20:        if acc < threshold then
21:            c ← c + 1
22:    Rt = c/n
23:    return Rt
24:
25: function getUserRating(U, α, β)
26:    Rs ← getUserSubmissionsRating
27:    Rt ← getUserValidationsRating
28:    R = αRs + βRt
29:    return R
30:
31: function getSubmissionsRatingAvg(S, p1)
32:    sum ← 0
33:    n ← number of submissions
34:    i ← 0
35:    for i < n do
36:        ri ← S(i).rating
37:        ai ← ri.accepted
38:        if ai = True then
39:            ri ← (1 − p1)ri + 5(p1)
40:        sum ← sum + ri
41:    savg = sum/n
42:    return savg
```

Algorithm 2 Update Submission Rating

```
1: function UPDATESUBMISSIONRATING(s, V)
2:     n ← number of verifications
3:     sum_1 ← 0
4:     sum_2 ← 0
5:     i ← 0
6:     for i < n do
7:         R_i ← V(i).User.rating
8:         r_i ← V(i).ratingGiven
9:         sum_1 ← sum_1 + (R_i × r_i)
10:        sum_2 ← sum_2 + r_i
11:    rating ← sum_1/sum_2
12:    return rating
13:
```

specify the number of winning designers and the corresponding percentage distribution between them (e.g., 90% for the first and 10% for the second). Rating and security fees take a fixed percentages, $\mathtt{p}_{\mathtt{des}}^{\mathtt{sec}}$, $\mathtt{p}_{\mathtt{int}}^{\mathtt{rat}}$, respectively, of the budget $\mathtt{b}_{\mathtt{des}}$ (e.g., between 5% and 10%, which can be also given as input or dynamically set by the designers). The security fees is released to the designer when the integration of its design is accepted. The remaining of $\mathtt{b}_{\mathtt{des}}$, i.e., $\mathtt{b}_{\mathtt{des}} \times 1 - \mathtt{p}_{\mathtt{des}}^{\mathtt{sec}} - \mathtt{p}_{\mathtt{int}}^{\mathtt{rat}}$, is to be paid for the wining(s) design. Let the winning design, $\mathtt{design}(\mathcal{S}) = \bigoplus\{S_i\}_{i \in I}$. The designer must submit (1) hardship of the integration; (2) hardship of development; and (3) a percentage for each sub specification, $\mathtt{p}_{\mathtt{sub}}$, where, $\sum_{i \in I} \mathtt{p}_{\mathtt{sub}}(S_i) = 1$.

- *Integration Budget*: $\mathtt{b}_{\mathtt{int}}$ is the budget allocated to the wining(s) integration and ratings of integrations, which is equal to $\mathtt{p}_{\mathtt{int}} \times \mathcal{B}$. As for the design, a user can specify the number of winning integrations and the corresponding percentage distribution between them. Rating takes a fixed percentages, $\mathtt{p}_{\mathtt{int}}^{\mathtt{rat}}$, of the budget $\mathtt{b}_{\mathtt{int}}$.

- *Development Budget*: $\mathtt{b}_{\mathtt{dev}}$ is the budget to be allocated for sub specifications (design, development and integration), which is equal $\mathtt{p}_{\mathtt{dev}} \times \mathcal{B}$. Given the wining design, $\mathtt{design}(\mathcal{S}) = \bigoplus\{S_i\}_{i \in I}$, the budget allocated to sub specifica-

tions S_i is equal to $\mathtt{b_{dev}} \times \mathtt{p_{sub}}(S_i)$. The winning designer of $\mathcal{S}$ is in charge of setting the budgets for the sub-components.

4.4 On-Platform Challenges

Despite all the added value of using the component-based system, facing some platform-related hurdles was inevitable. Putting our platform under test showed us the following challenges, that are solved as explained:

- *Lack of user knowledge:* Some workers might not be familiar with the component-based design, and would prefer on a rather traditional workflow. To solve this issue documentations and tutorials are available for the users for reference and guidance.

- *Designer's duties:* A designer has far too many duties just like in any traditional approach, in terms of: *proper* requirements refinement, *fair* budget decomposition, *well defined* interfaces and sub-specifications. Thus a designer gets rewarded more than other workers in return for the challenging job they do.

- *The dead component scenario:* We are working under the assumptions that we have a crowd big enough and experienced enough to commit to all the jobs posted. But in the real world, a component could be intentionally or unintentionally left without development, causing the delay of the process. The issue could be solved by raising the payment on such a job and giving it priority over others.

- *An unsolved component:* A similar scenario to the "dead component", is having a component with several submissions to it, but the submissions are wrong or not good enough. A similar case would be having a wrong judgment by the testers, and having a submission highly rated, although its not the best.

This would also result in a delay of the development process, but back to our assumption: there will always be a correct submission.

Chapter 5

CROWDSOURCED PROJECT BUILDING WORKFLOW

Contents

5.1 Design Phase 32
5.2 Development Phase 34
5.3 Integration Phase 35
5.4 Miscellaneous 35
5.4.1 The importance of user ratings 35
5.4.2 Payment fairness 36
5.4.3 Maintenance and bug hunts 37
5.4.4 Automation and Copilot's Absence 38

Now that the basic elements of the platform are defined, they should all be put together in a well-defined workflow, in order to enforce automation. The rules stated in the workflow, are meant to replace the human copilot that manages projects on other platforms. The steps to be explained start from a requester's post on the platform initiating a project P, and finish at the delivery of P.

5.1 Design Phase

As mentioned before, upon posting a project request on the platform, a design job gets automatically created. This root job J_0 is associated with the root component C_0. The `refine` method should then be applied on C_0: the output of the first stage of refinement is several independent sub-components, smaller in size than C_0 by definition, such that the composition of those sub-components is equal to C_0. An example of three sub-components of a certain random project:

1. Front-end: this component can be further decomposed into several pages, and each page will then be designed by a different worker. The designer must state the proposed pages and a general description about each, leaving the detailed description for the designers that will follow. The designer should state the necessary interfaces to communicate with the back-end and the database, for example: the database name and the languages to be used.

2. Back-end: can be of large-scale and needs several rounds of refinement. The designer should give an overview of the requirements needed, and some details as mentioned in the front-end component.

3. *Database: might be simple and can be directly ready for development (an example of a base component).* Therefore, a detailed description of all the needed tables, relations, keys, and data should be stated by the designer.

Working on the three sub-components from the example can be done in parallel, noticing that the development stage can start independently before the design stage is done. The flexibility offered by using the component-based approach allows us to replace any defected or less optimized component at any stage of time, as long as the inputs and the outputs of the two components are compatible.

Each job available on the platform, allows workers to sign up to it if their *design rating* is higher than the minimum rating required for this job. Limiting the minimum required rating improves the quality of work, and emphasizes weaker workers to improve their ratings by working on simpler jobs requiring lower ratings. Designers then read the specification $\mathcal{S}_i$ of the component $\mathcal{C}_i$ they're joined, and submit a component-based design for it. All the submissions to the job get verified and rated by verifiers, whose job is to make sure that the submitted design fits the specification, then rated accordingly (0-5) and leave a comment if needed. The comments give the designers a chance to resubmit their work edited and not lose their chance of winning. The validation process runs in parallel with the design submission pro-

cess, so the earlier workers submit the better. When the job reaches its due date, a pool of top-rated submissions get select and their submitters get paid (the size of the pool is predetermined by the project requester).

Finally, the submission having the highest rating gets selected as a design for the new sub-components, and the winning designer must create those sub-components on the platform based on his design. Following the same recursive refinement approach, the output of the design phase is a tree hierarchy where the root is C_0 and all the leafs are base components, ready for development.

5.2 Development Phase

Development can start at any point in time when a base component design is ready, so no need to wait for the whole design phase to finish. For the sake of simplicity, we assume that the development phase at a component starts when all the sibling components have their designs ready, and no code has yet been submitted to the platform:

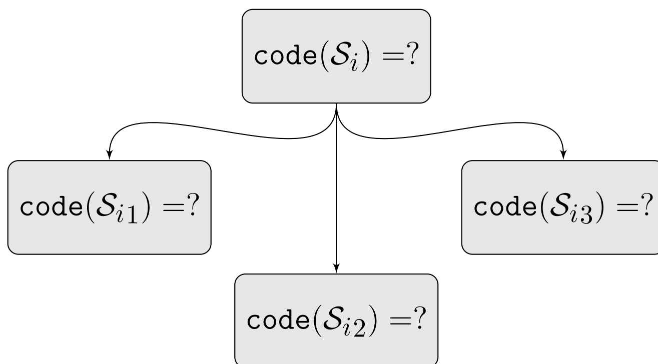


Figure 5.1: Pre-development graph

Based on the definition of the method `code`, the code of $\mathcal{S}_i$ is the composition of the codes of its children. And since the children are base components, their code is the implementation submitted to the platform by the crowd.

Each component C having no implementation is part of an open job J on the platform. Similar to the design phase, a competition gets carried out where workers

read the specification of C and submit a code that fits it. The code submitted should also work properly on the test cases attached to the job, by its designer. Testers from the crowd run a different test suite, rate the quality of the code submitted, and give comments for the submitters. Just like the design phase, a certain pool of winners gets paid for having the highest rated submissions. Finally, the highest rated code gets assigned to the component such that $\texttt{code}(\mathcal{C}) = \text{Implementation of } \mathcal{S}$.

5.3 Integration Phase

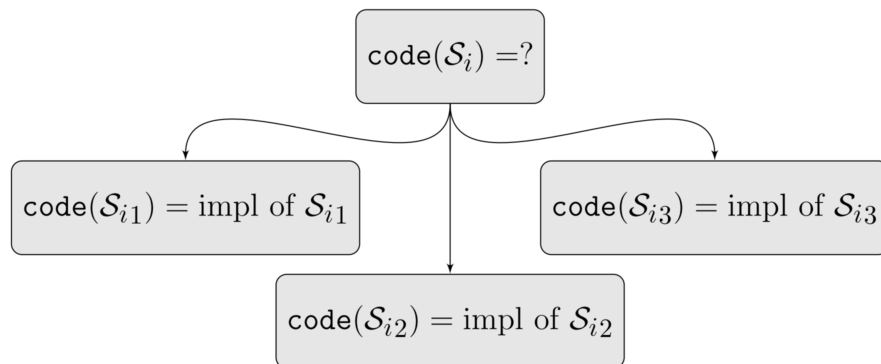


Figure 5.2: Post-development graph

Whenever all the children of a compound component have their $\texttt{code} \neq \phi$, an integration competition starts on the platform, where the workers' job is:

1. Examine the codes of sibling components
2. Examine the interfaces describes in the design (if they exists)
3. Create a functioning code out of the codes of the different sibling components
4. Submit the new code to the platform

5.4 Miscellaneous

5.4.1 The importance of user ratings

As discussed in the previous chapter, the rating of a user changes by the end of each job he/she participated in. A rating plays a huge role in allowing the user

to join more jobs and gain more money, it is the only ticket in. Dividing the user rating into three different ratings is to allow him/her to participate in a jobs type even if their rating is zero in a different type: a user with zero design rating can still participate freely in any development and integration jobs. Similarly a worker might choose to participate in only one of the job types, and this is perfectly fine. Ratings in different fields have no effect on one another.

One's rating might allow him/her to join a certain job with a low minimum required rating, but in the case of testing, the weighted submissions' rating system followed limit the user's effectiveness on the submission's final rating. This strict rule is necessary on our automated platform to limit/restrict the participation of workers with little or no experience. Moreover, it improves fairness towards experienced workers, by giving their opinion a stronger effect on the project building process.

5.4.2 Payment fairness

Adopting the component-based approach allowed us to hugely improve the fairness towards the workers. Since their is no way all the participants could be paid we tried to reach a middle ground between the requesters and the workers. Fairness was accomplished for both parties by the following:

- *Budget set in advance:* The platform forces the user to take responsibility of the budget set. If the budget was little, prizes on jobs will be low and workers will not be interested. But if the budget is set properly, the project will attract more job seekers.

- *Smaller jobs:* Working for a little time and receiving no payment is much easier than spending a long while and ending up losing the challenge and not getting paid. Components are simplified as much as possible and thus giving the chance for the worker to participate in several competitions, highly improving the chance of a win.

- *Submissions ranking:* A requester can set any number of winners to be paid based on rank. If the number is n, the workers having the top n submission ratings all get paid. The worker can see the prize he/she would win in advance, without knowing what n is, and thus freely deciding whether to participate or not.

- *Tester payments:* A tester gets paid for each *correct rating* given. Each tester gets an equal share of the testing budget set for the component.

- *Security budget:* Some percentage of the prize money is saved by the platform and not given to the worker, until the project is done. This percentage is kept as security, in case of a faulty component. Requesters would feel safer if this is done, and would encourage them to post more projects. Workers on the other hand will be more cautious with their submissions, knowing that they will be paid the full amount after a certain period of time.

5.4.3 Maintenance and bug hunts

Maintenance is a crucial part of every software project, and using components in building the software facilitates maintenance greatly. At any point, a component can be replaced by another, having the same specifications, input and output. For example, if a website owner wants to change the looks of a certain page, the component responsible for that page can be replaced by another, either from the pool of submissions made earlier, or a new job gets created for its sake.

Some existing platforms conduct a bug hunt during the user's trial period. The same exercise is done on our platform during the testing of the last component's composition i.e. the root component. Having all the codes assembled within it, the code of the root component represents the whole project with all its functionalities. Therefore, testing it, rating it, and commenting on its problems, is actually a bug hunt.

5.4.4 Automation and Copilot's Absence

With all the rules of the system set and well defined for both the requesters and the workers, the need for a human copilot diminishes. The approach we followed takes the managerial position from the copilot and gives it to all the winners in a project. It lets the winners create the jobs, of designs they see best fit, putting them in the position of responsibility and thus increasing their interest in the project. Not having a fixed manager, minimizes the risk of that manager abandoning the project or not being worthy of the position.

Chapter 6

IMPLEMENTATION

Contents

6.1 Design and Architecture 39

6.2 Requester's Perspective 41

6.3 Worker's Perspective 44

6.4 Project building process 45

Based on the design our research yielded, and for the sake of testing our approach, we developed a prototype of a platform that implements our model. The platform is a website created using ASP.net MVC5 [10], so-called CPISE: Crowdsourcing Projects in Software Engineering. CPISE implements the crowdsourcing platform described in the previous chapters, in all its aspects. The implementation process improved the previously described design into its current form because it offered a hands-on experience with our defined approach and a closer look at its details. In the following sections, we describe the implemented platform and present a parallel version of the workflow on CPISE, both from a requester and a worker's perspective.

6.1 Design and Architecture

By default, using the MVC pattern divides the implementation into three interconnected parts: model, view and controller.

Starting with the *model* part, objects created allow the interaction with the MS SQL database [11] through the object relational mapper "Entity Framework (EF)" [12]. Models also allow the secure transfer of data between the different views of the website, passing through controllers. Several models were created to favor the different data types needed: user, project, job, validation, submission and more. A variety of model configurations were implemented, since the platform

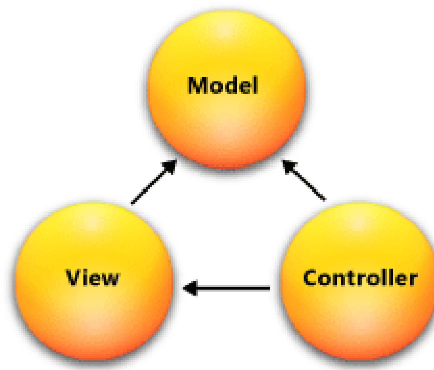


Figure 6.1: MVC design pattern [1]

contains several forms and different operations require different pieces of data to be communicated.

Views in MVC represent webpages and their contents, written mainly in HTML5, styled with CSS, Java Script and Bootstrap. ASP.net allows the usage of "razor syntax" [13], which allows embedding server-based code into webpages: the language used to write in *razor* was *C#*. Partial-views were also implemented to improve code reuse, since they can be embedded within other views and rendered by HTML. Views and partial views were manipulated using razor syntax in order to display different information, based on the user of the website and his/her status, for example:

- Project requester/owner: can view more details about his project, have the ability to edit some of its parts, have access to detailed statistics, and more.
- Regular worker: has the ability to join a job, make a submission, rate a submission, all which the project owner cannot do. Has limited access to jobs based on his/her rating. Moreover, jobs based on preferred programming languages or job types are recommended for the worker on the home page.
- Winner of a competition: gets congratulated and informed about the money won. A payment is issued for him/her, and a rating adjustment occurs.
- Loser of a competition: a regret message for not winning and the rating his/her submission received.

Finally, controllers are the classes that handle user inputs, models, and query strings. Controllers contain all the methods needed to select which views to open *(based on URL and parameters)*, send data to them, or receive data from them through the different implemented models. Most of the operations done on the database are written in the controllers, whether updating or selecting data. Back end operations are mostly written in the controllers in the form of static and non-static methods.

6.2 Requester's Perspective

CPISE was built in a way that would serve its targeted audience, in the simplest way possible. The audience can be either requesters or workers, and requesters could be either experienced or inexperienced in the software domain. As mentioned before, a user can be both a requester and a worker, and can choose what to do from a set of options like in figure 6.2.

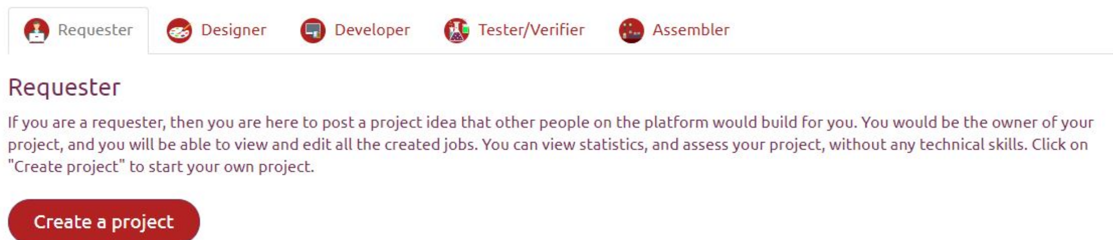


Figure 6.2: Options Tabs

A project post is the very first step of the workflow on CPISE, and it cannot happen without a requester. Our main concern was always the inexperienced requesters who would like to get their software project built easily and fast, without having to deal with companies or freelancers. So after a regular sign up, the user will have to click a "Create Project" button that would lead to the new project form found in figure 6.3.

An inexperienced requester can only fill the project description text area before clicking *"Create Project"*, and then the crowd will be fully responsible for the

Figure 6.3: Project Creation View

requirement extraction and design refinement process. On the other hand, more experienced requesters are encouraged to attach any document that could be of help for the designers. Note that having clear requirements from the start would drastically facilitate the project building process, and mainly the design phase.

Upon filling the form a project gets created on the platform, and along with it an *open* design job having the same description and details: budget, hardship level, minimum worker rating, maximum accepted submissions, due date, and attached files. An example is illustrated in figures 6.4 and 6.5.

In figure 6.4, we are assuming that the requirements of the requested project are in the attached document *"Cpise Specs"*, since this project is created by an experienced requester.

Figure 6.5 shows the respective design job automatically created by the platform, so that the design phase gets initiated. The design job is represented as the root

Name:	CPISE
Project Description:	Please read the attached document for details.
Requester:	Bilal
Approximate due date:	1/1/2019 12:00:00 AM
Attachment:	Cpise Specs
Redundancies allowed:	False
Number of jobs available:	› Design: 1 › Development: 0 › Merging: 0

Workflow | Project Design

Figure 6.4: Project Details

component in the component hierarchy and called *"Parent Design"*, like in figure 6.11.

Project:	CPISE
Job description:	Please read the attached document for details.
Status:	Open
Job level in hierarchy:	1
Hardship level:	10
Reward:	$30
Minimum user rating to joing:	2
Job attachements:	Cpise Specs
Due Date:	1/1/2019 12:00:00 AM

Figure 6.5: Design Job Details

Now the job shown in figure 6.5 is available for all workers who have their design rating > 2. Unlimited submissions can be made by any of the designers to the job, but only a certain pool will be selected for payment, based on the number of accepted submissions entered by the requester earlier.

Now the requester owns the project and can act as a supervisor to track the progress of the process, with detailed statistics and overview of the project hierarchy.

The same user might own several projects and can visit them at any time (figure 6.6). The platform would indicate if the projects is done, or if its still under construction.

Figure 6.6: My Projects

6.3 Worker's Perspective

In order to help the worker find a suitable job on the platform, he/she should ideally set a preferred job type upon sign up: design, development or integration. Moreover, the worker can select a set of preferred programming languages, improving the job recommendations on the home page and boosting the chance of joining relevant and interesting job. Upon sign up, the user gets an initial rating of 2.5 on design, development and integration, giving him the chance to join a wide range of available jobs. Those ratings get adjusted based on the submissions in each field, or the correct submissions' rating in case of verification or testing.

Regardless of the preferences set, the worker can still join any type of job he/she likes, as long as the rating in the particular field is higher than that required for the job. Figures (6.7, 6.8, 6.9, 6.10) show the different tabs at which a user can find suitable jobs.

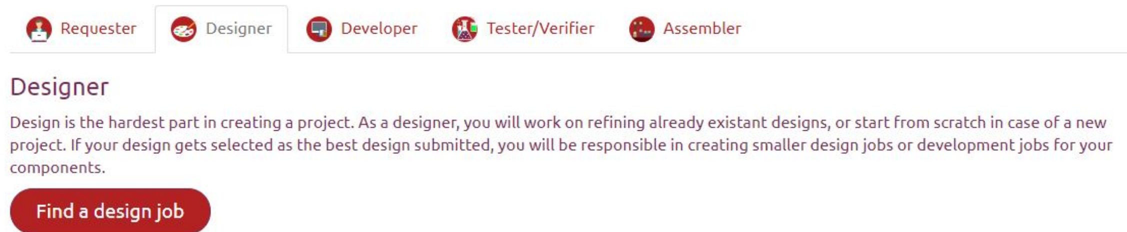


Figure 6.7: Design Search

Figure 6.8: Development Job Search

6.4 Project building process

Naturally, requesters and workers come together in the project building process, that starts as soon as a project gets created. As the aforementioned sections state, the first job in the project is a design job that will start accepting submissions from workers eligible to join it. The designer whose submission gets selected as a winner for the first components must then create new components, based on the design he/she submitted. Each new component created will have budget set by the winning designer, and limited by the platform below the budget of the parent component. The new components must also have a due date earlier than that of the parent, to stay within time limits. A design competition will be held on every component created, in a parallel fashion. Designers will make submissions that will then be verified by other crowd members who should also have an above-the-limit rating.

The same rules of winning and sub-jobs creation are followed recursively, until a base component is reached. As mentioned before, a base component has the simplest form of a specification, that can no longer be refined. In this case, the winning designer will have to create a development job, and thus a development

Figure 6.9: Assembly Job Search

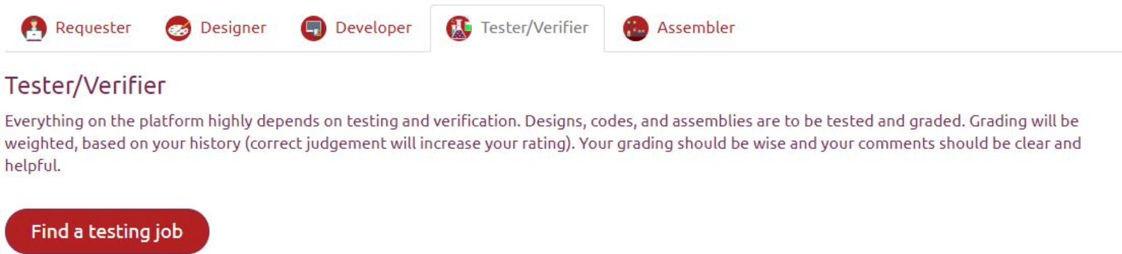


Figure 6.10: Testing Job Search

competition will start.

After all the design jobs are done *(components are implemented)*, their respective design jobs retrieve the winning code submissions. Based on the example hierarchy reported in figure 6.11, if development jobs numbered: 3031, 3032 and 3033 all have their codes ready, those codes will be handed respectively to the design jobs numbered: 2030, 3029 and 3030. The only job with no code ready would now be the *"Parent design"* numbered 2029. In order to give the root its code, the codes of its child-nodes should be integrated by the crowd through a *"merging job"*. Such a job will be automatically created by the platform as soon as all the sub-components of any parent component in the hierarchy have implementations assigned for their specifications. The full hierarchy is presented in figure 6.12, showing the new merging job numbered 3034. The integrated codes from the three jobs submitted to the merging job will be tested and rated as usual, where the winning code will be assigned to the *"Parent design"*.

When *"Parent design"* has an implementation assigned to its specification, the projects closes and a final code is available for the requester to download.

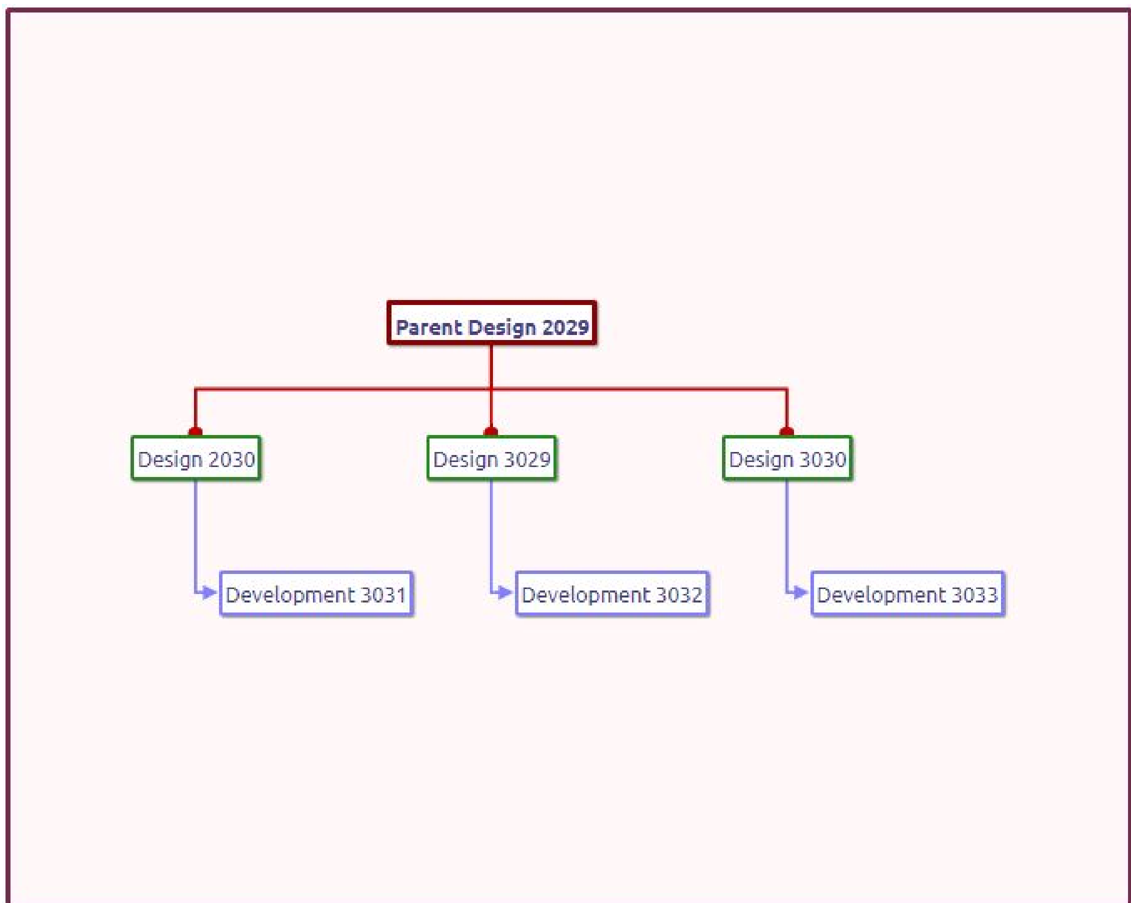


Figure 6.11: A Simple Hierarchy

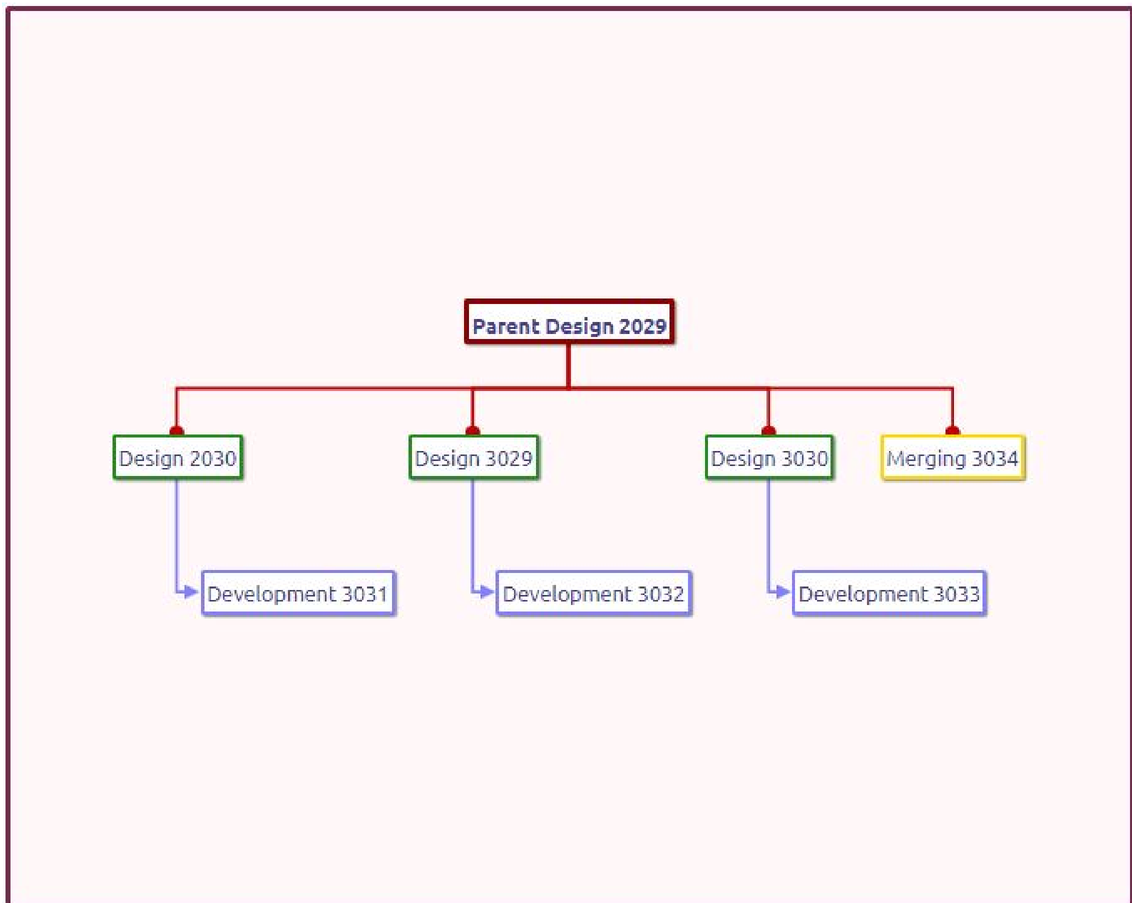


Figure 6.12: A Simple Hierarchy- Full

Chapter 7

CASE STUDIES AND BENCHMARKS

Contents

7.1 Measurables 49
7.2 The Sample Projects 50
7.3 Experimental Observations And Results 52
    7.3.1 Observations 52
    7.3.2 Experimental Results 53

In order to study the validity of our approach in the first place, and its effectiveness in the second place, a real life case study has been conducted over a group of computer scientists acting as a crowd. The crowd members have different expertise in different fields and collaboratively built the project to be described, without having any direct contact between them.

7.1 Measurables

Our experiments test for the following in comparison with a freelancer or a software company:

- TTD
- Price
- Quality
- Number of paid workers
- Worker's availability
- Number of reusable codes

We also study was how our platform could be compared to other platforms. More specifically how the component-based approach performs compared to the single-component approach. We thus measure the following:

- Time to receive the first submission
- Time to delivery (TTD)
- Average submissions' rating
- Overall quality

One more thing to analyze is the effect of the different jobs posted, on the workers and their submissions in terms of:

- Number of submissions versus hardship level
- Number of submissions versus component level
- Number of submissions versus prize money
- Quality of submissions versus hardship level
- Quality of submissions versus component level
- Quality of submissions versus prize money
- Average submissions rating per job type
- Average workers' rating at the end of the case study

7.2 The Sample Projects

Three projects were used for this case study, and each targets a certain set of measurables. The first two projects were used to compare our platform to some other existing crowdsourcing platforms. The projects have similar hardship levels to make fair comparisons.

In order to perform the test we divided the crowd into two groups:

- *First Group:* Work on a single job project to simulate a project competition on different platforms. That is, the worker reads the requirements and does the whole project on his/her own and submit it to the platform, fully functional.
- *Second Group*: Build a component-based design, develop it and integrate it as explained in the previous chapter
- *Both Groups*: Each group tests and rates the other group's submissions

For the third project we used a real life project done by both a freelancer and a software company, supplying us with the necessary data for the study: price, TTD and the source code. The group of workers now acted as independent workers and contributed to the building process in all its stages.

The projects were:

- Project 1: Daily expenses tracker. Given a monthly income and allowing the addition of any daily income as well, add or remove expenses, monitor savings and warn user if daily limit exceeded.
- Project 2: Supermarket items reminder that allows the creation of a list of items from a pool of frequently added items. If the items do not exist in the pool, create and add them. Each item should have: type, size, quantity and prices.
- Project 3: A website that allows collecting information about local companies (mostly SMEs) related to labor market trends, required skills and existing job vacancies. This Labor Market Observatories will become resource hubs on labor market data to job seekers, as well as private public and international stakeholders. SMEs must be able to register and fill in their basic profile information (name, corporation type, sectors) and another detailed profile (governance and structure, employment status). SMEs must also fill Skills

Based Needs Information, composed of: Required technical skills based on sector, required soft skills, and training history. Public visitors must be able to register to access the platform, in order to see the profiles of the SMEs and find job vacancies.

7.3 Experimental Observations And Results

The two following subsections are based on the case study we conducted and monitored. Before working on the platform directly, we introduced the project to the *crowd* in general and the rules of component-based design, development and integration. We then showed some design examples (component specification decomposition) and possible implementations. The tutorial was thirty minutes long and after that the workers were ready to work on the platform independently. Since the projects were already posted by us, we acted as requesters.

7.3.1 Observations

When the crowd started working, we examined each individual separately and had the following observations:

- *Attention grabbed:* As we explained our approach and our platform rules, the crowd was positively reacting. The idea appealed to them and they were excited to work on it.
- *Fast start:* What we directly noticed was the very fast adaptation to the system, all the crowd members knew exactly what to do and competed against each other in a real life scenario, despite the different skills and levels.
- *Satisfactory results:* Submissions were generally satisfactory with a relatively high average submissions rating (≈ 4.14). We made sure that the winning submissions are actually the best, by manually comparing the competing submissions.

- *Positive feedback:* Despite some remarks about issues that could be fixed on our implemented prototype of a platform, the feedback was generally positive and the workers were satisfied and comfortable with the work-flow.

- *Crowdsourcing effectiveness:* Seeing the crowd work on the project collaboratively, yet each worker doing the job independently, showed us the power of crowdsourcing. We retrieved several solutions for the same problems, got them tested, rated and ranked. The best submissions were used in the following stages.

- *CBS effectiveness:* The huge added value to our platform was the utilization of the component-based system, and this showed clearly when we compared the regular crowdsourced project to a component-based crowdsourced project, especially in terms of *TTD* (experimental results below).

7.3.2 Experimental Results

The results obtained are divided into three main parts:

1. CPISE versus software company versus freelancer

2. Component-based versus single-component

3. Jobs versus workers

Based on project 3 described earlier, we illustrate in table 7.1 the foundations of comparing our platform to a software company and a freelancer, based on a common project. Therefore, if the same project is to be distributed on the three parties we can clearly notice that CPISE offers:

1. A much faster TTD, which could have been even faster if the crowd was bigger.

2. A much cheaper approach, half the price asked by a freelancer and one seventh of that asked by a software company.

3. Good quality, that could be improved with more submissions from a bigger crowd.

4. More shared resources, since more workers get paid in a component-based project: a minimum of three workers per component assuming we are paying only one designer, one developer and one tester. And thus the number of workers can grow exponentially as the number of workers participating increases.

5. Each coded component serves as a potentially reusable code. Companies and freelancers eventually reuse their codes, but on the platform, each component has a pool of implementations that could be reused at will.

Criteria	CPISE	Software Company	Freelancer
TTD (Days)	7	30	21
Price ($)	2000	14,000	4,000
Quality	Good	Very Good	Good
Number of paid workers	8	4	1
Number of reusable codes	Equal to the number of base components	~	~

Table 7.1: CPISE versus software company versus freelancer

The natural thing to look at while benchmarking a crowdsourcing platform, is how it compares with other already existing platform. In order to do that we created two projects equal in hardship, and crowdsourced them. The first has to be worked on individually by the crowd members, while the second has to be done collaboratively through the component-based approach. The differences were obvious, and they are illustrated in table 7.2.

Knowing that the two crowds are randomly selected, the number of participants is considered very little compared to an actual crowdsourcing practice, we notice the following:

Criteria	Component-based	Single-component
Time to receive first submission (hours)	0.5	3
TTD (hours)	3	4
Average Submissions' ratings	4.14	4.09
Overall Quality	Very Good	Good

Table 7.2: Component-based versus single-component

1. The component-based project felt very alive compared to the other one. Consecutive submissions started reaching the platform after hitting the half hour mark, and kept flowing for three hours until the project was finished. On the other hand, the single-component project's first submission was made after 3 hours, that is when the component-based project was done. The second project needed one extra hour for the other workers to submit their projects and for testing to take place, and thus showing the difference in performance. Note that the projects are small in size, and the marine would grow bigger as the project scale increases.

2. Submissions' ratings are highly dependent on the crowd members individually and not necessarily state that one approach is better than the other. But what is worth mentioning is that the small size of components made it much easier for the workers to achieve high quality. What pulled the rating down a little was the early stages of the project, where the workers were new to the platform and had no experience in component-based systems. The quality directly improved after we analyzed together the problems they faced and the errors they made.

3. The variety of submissions made on the platform's various jobs gave us a very good output quality, due to the selection process that was carried on based on ranking.

The final test we carried on was to analyze the effect of different jobs on the workers submissions. The first test results shown in figures 7.1 and 7.2, provided us with a general notion of how things would work on the platform in terms of job hardship versus worker submissions:

1. The number of submissions is maximum at an average hardship level equals to 5, being the simplest of the jobs. The simpler the job is, the more workers it will attract, due to the easier win and the little time it would cost.

2. As the hardship of the job increases, the quality of the submissions decreases. This is obvious due to the error-prone tougher jobs. But still, the lowest average rating at hardship level 10 is about 3.6, which is still very acceptable.

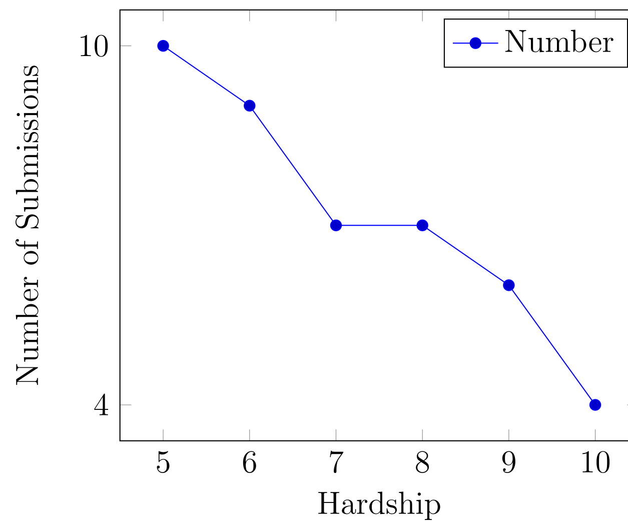


Figure 7.1: Number of submissions versus job hardship

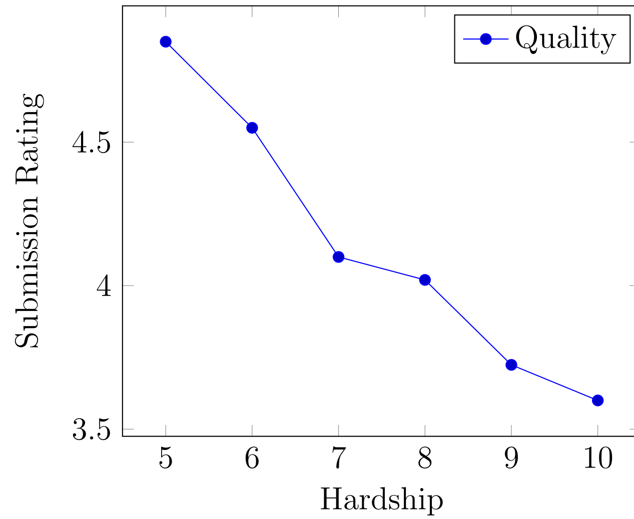


Figure 7.2: Quality of submissions versus job hardship

The plot in figures 7.3 and 7.4 illustrates how the quality and interest of workers in the posted jobs varied, as we went down the components' hierarchy. Level 1 which is the very first component received the lowest number of submissions due to its hardship. The number of submissions kept increasing with the level due to the increasing number of components. Quality levels were variable due to the almost unrelated level-hardship figures, but maintained a fairly good average.

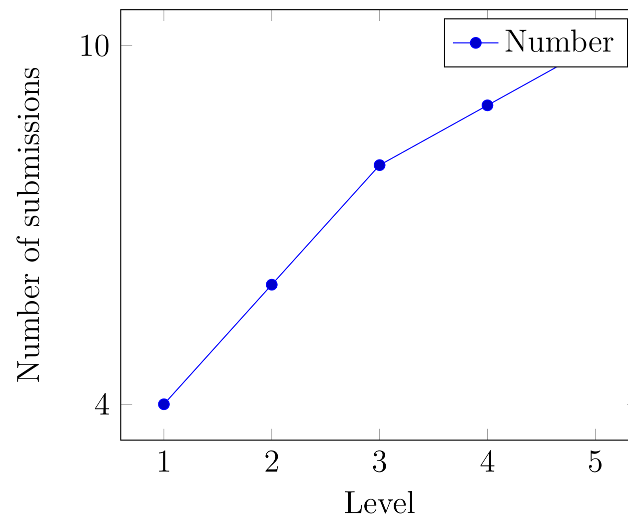


Figure 7.3: Number of submissions versus job level

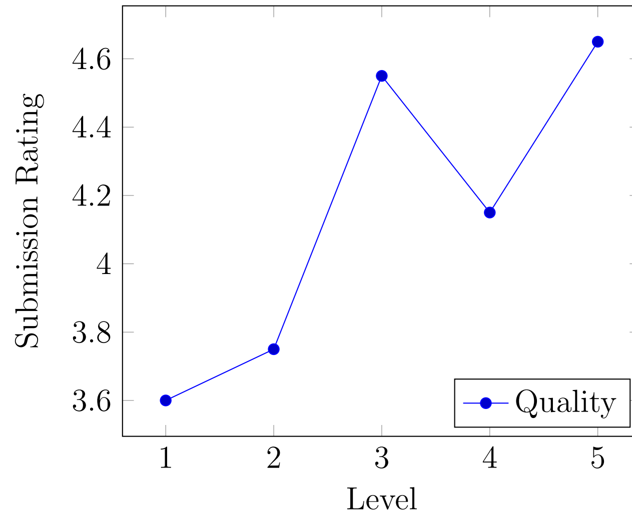


Figure 7.4: Quality of submissions versus job level

As for the number and quality of submissions versus prize money we could not retrieve the needed results, because the crowd operated on jobs with imaginary prizes and participated in most of the jobs regardless of the prize. But it would be safe to say that there should be a relationship between prize money and hardship level, such that an easy job with little money would probably get more submissions and better quality than a hard job with little money. We should also mention that the hardship levels are roughly estimated, and set based on the objective opinion of the designer, making the hardship-prize comparison harder to attain.

Average submissions rating per job type were all around 4.1 showing the consistency of the submissions and the ratings. As for the workers' ratings, they were very similar to the submissions with the lowest rated at ≈ 3.5 and the highest at about ≈ 4.6.

Chapter 8

RELATED WORK

Contents

8.1 Full-task Crowdsourcing 60
8.2 Stage-specific Crowdsourcing 60
8.2.1 Crowdsourcing software requirements extraction 60
8.2.2 Crowdsourcing for Software Design 62
8.2.3 Crowdsourcing for Software Coding 63
8.2.4 Crowdsourcing for Software Testing and Verification 63
8.2.5 Crowdsourcing for Software Evolution and Maintenance 64
8.3 Research work in crowdsourcing softwares 65
8.4 Comparing existing platforms to ours 65

Crowdsourcing [8] proved its effectiveness with tasks that require a big number of people, and platforms such as Amazon Mechanical Turk [4] gave a space for those willing to make a small amount money for every “hit” they make. “Hit” is the term used on crowdsourcing platforms while referencing a micro-job. Mechanical Turk is not software oriented, but some users still post their software-related jobs on it. Despite the size of the crowd on Mechanical Turk, it does not have the means to support software crowdsourcing, mainly due to its advertisement of micro-tasks and not the macro-task of building a software.

As for pure software orientation, TopCoder [7] is the most known. It is an online crowdsourcing platform where challenges get carried on the levels of development, design and data science. The winning challenger gets rewarded by money and/or gifts [2].

8.1 Full-task Crowdsourcing

TopCoder [14] is an example of full-task crowdsourcing, whether it is UI design, software development, or algorithm optimization. By full-task we mean that one person does the job from A to Z, and wins the money prize in case of winning the competition. GetACoder [15] is another platform that serves the same goal using yet a different technique: online bidding. GetACoder is more like online freelancer searching, where bidders comment on the post and give the price they want to get paid in return of their services. The chosen bidder gets to develop the job and wins the money.

8.2 Stage-specific Crowdsourcing

Authors specifies the different fields software engineering, and how they get dealt on different crowdsourcing platforms [2].

8.2.1 Crowdsourcing software requirements extraction

In the early stage of our research we considered the idea of extracting formal requirements from natural language. An example of that is a requester, not experienced in the domain of software engineering, asking for a project on a software crowdsourcing platform. The request should then be processed by the platform, and the output would then be a set of requirements understandable by the experienced crowd.

A similar work was done by [16] when they introduced an Extended Functional Requirement Framework (EFRF) that takes a natural language process (NLP) as input and gives a software requirements specification (SRS) as output. An EFRF is made up of the following elements:

- *Agentive*: the agent who's activities will occur in EFRF's affairs
- *Action*: defines the main action of the activity

- *Objective*: defines the object affected by the activity
- *Agentmod*: defines the feature of agent
- *Objmod*: defines the feature of object
- *Location*: defines the location where the EFRF affairs occur
- *Temporal*: defines the duration or frequency of the EFRF's activity
- *Manner*:defines the way or tool by which the EFRF's activity is performed
- *Goal*: defines the goal of the EFRF's activity
- *Constraint*: defines the requirement and constraint that make the activity occur

A different approach was followed by [17] to refine specification from spoken English to formal specs using: system goals. *Goal orientation* is an increasingly recognized paradigm for eliciting, modeling, specifying and analyzing software requirements. Goals are statements of intent organized in AND/OR refinement structures; they range from high-level, strategic concerns to low level, technical requirements on the software-to-be and assumptions on its environment.

Operational software specifications are built incrementally from higher-level goal formulations in a way that guarantees their correctness by construction. The *operationalization* process is based on formal derivation rules that map goal specifications to specifications of software operations; more specifically, these rules map real-time temporal logic specifications to sets of pre-, post- and trigger conditions.

How modeling works: An application model is composed of four sub-models:

1. a **goal model** in which the goals to be achieved by the system are described together with their alternative refinement links and their conflict links
2. an **object model** in which the application objects involved are described together with their relationships and attributes

3. an **agent model** in which the agents in the system are described together with their interfaces and responsibilities with respect to the goals
4. an **operation model** in which the services operationalizating the goals assigned to software agents are described.

The paper concentrates on the derivation of the operation model from the goal model; some features of the operation model will therefore be presented in greater detail as they provide the basis for our operationalization process. Where **operationalization** refers to: the process of prescribing additional pre-, trigger-, and post conditions on operations in order to achieve goal specifications.

- a required precondition captures a permission to perform the operation when the condition is true;
- a required trigger condition captures an obligation to perform the operation when the condition becomes true provided the domain precondition is true;
- a required post condition captures an additional condition that must hold after any application of the operation [18] [19] [20]

8.2.2 Crowdsourcing for Software Design

Software design has several fields of action: user interface design, architecture design and design revision. Some of the platforms specialized in crowdsourced design are [2]: 99designs [21], DesignCrowd [22], crowdSPRING [23]. Those platforms are mainly into the graphical aspect of design, and leave the architectural design to a proposed platform named Apparition [24]. Apparition helps designers prototype interactive systems in real-time based on sketching and function description [2]. Those platforms have their limitations in terms of evolving designs for multiple designers' solutions, whereas [25] proposed a system to let designers produce initial designs and evolve their solutions based on others' solutions. The study is still a theory and not yet implemented and cannot be considered while comparing platforms.

8.2.3 Crowdsourcing for Software Coding

1. IDE enhancement: crowd knowledge can help improve multiple functions such as API documentation, code completion, code utilization, bug detection, and code search. Some of the supporting platforms are: HelpMeOut [26], BlueFix [27], Calcite [28], SeaHawk [29].

2. Crowd programming environment: There exists a design for a Cloud-based Integrated Development and Runtime Environment (CIDRE) that consists of: a crowd developer community, online IDE, and an app store [30]. These components link the designers, developers and users together and promote the mutual feedback among them. Examples of other platforms are: Code Hunt [31], Jabberwocky [32], Automan [33]. Those platforms offer different ways to crowdsource development practices either by retrieving solutions via different submissions, or sharing an online API where coding is done in a shared fashion.

3. Program optimization: Crowdsourcing has been used to support compilation optimization [34] and program synthesis [35] placing flags to guide the compiler to perform optimization [2].

8.2.4 Crowdsourcing for Software Testing and Verification

Testing platforms recruit not only professional testers, but also end users to support the testing task. They can be in both, black box or white box testing, in test case generation, and the oracle problem.

1. Usability testing: Online users showed capability for detecting usability problem as good as done by experts [36]. Platforms supporting this kind of testing found it challenging to detect cheating among users.

2. Performance testing: There's a way to collect performance data from users like Google Chrome's and Firfox's built in telemetries [37, 38, 39], and is considered to be a kind of crowdsourcing although workers are mostly unaware about it.

3. GUI testing: Done using A/B tests of UI via remote virtual machines or by offering remote virtual machines to testers. Such a job could be crowdsourced on platforms like Amazon Mechanical Turk due to its simplicity.

4. Test Case generation: Puzzle-based Automatic Testing (PAT) environment was presented for decomposing and translating the object mutation and constraint solving problems into human-solvable games (gamification) by [40]. Existing test case generation tools are jCUTE [41], Randoop [42], and Pex [43].

5. Oracle problem: Oracles rely on human input, making it hard to fully automate software testing. It requires skilled workers provided with well-designed and documented tasks. Frameworks to support such crowdsourced testing are: CrowdBlaze [44], iTest [45], Caiipa [46], Xie [47].

8.2.5 Crowdsourcing for Software Evolution and Maintenance

1. Crowdsourced Software Evolution: A proposed solution named Social Sensing [48] suggests to leverage the wisdom of the end users and use them as monitors for software runtime adaptation. This helps software designers to capture adaptation drivers and define new requirement and contextual attributes through users' feedback.

2. Crowdsourced Software Documentation: Most of the studies were mainly conducted on StackOverflow [49] and found that frequently asked questions can be maintained for generating expanded API documentation automatically.

3. Crowdsourced Software Localization: Software localization is relevant to "software internationalization" or "globalization", such as tailoring the nature language output for system for each country in which they are deployed. Localization may be an important factor for the adoption and success of international products [50].

4. Crowdsourcing for Other Software Engineering Activities: Security, privacy [51, 52, 53], software end user support and software ideation [54, 55, 56], and malware analysis can all be crowdsourced [57, 58].

8.3 Research work in crowdsourcing softwares

This section can be divided into two part: (1) using crowdsourcing to favor research and (2) research done on software crowdsourcing. Starting with the first part, any research involving human subjects can be crowdsourced [2]. An example can be Social Sensing mentioned earlier or performance testing methods. The results harvested can be used in research, but there is still no crowdsourcing platform solemnly created for research purposes. As for the second part, research done on software crowdsourcing covers several topics, listed in decreasing frequency [2] application, theoretical, practical, evaluations.

8.4 Comparing existing platforms to ours

Existing platforms have the following issues, that were all fixed on our platform as follows:

1. Using the waterfall model like in TopCoder brought coordination issues between clients and coders. To solve that, we added the iterative characteristic from the agile mode. Moreover, adapting the component-based system highly improved the ability to maintain and replace any unwanted part of the built softwares, thus making any edit possible.

2. Unfair competition for workers often exists: more experienced workers are allowed to register for a job earlier than others, and thus giving them a higher chance of gaining money. Less experienced workers then suffer from unfair treatment and competition. Our platform on the other hand gives equal

chances to all workers, despite their level. The granularity of software components further reduces the risk of not getting paid, offering workers a more friendly working environment.

3. Managing: the crowd, the process, the techniques is not always satisfactory. Thus we moved to the automated design with well defined rules, that would outline the workflow for the workers and give them all the details they have to be familiar with before working.

4. Dependency on human "copilots" to manage the development process. Those copilots might unexpectedly leave the project, or maybe mismanage it, leading to the project's failure. Automation solved this issue too, by giving the managerial power to competition winners, decentralizing the copilot's position and reducing the risk.

5. Platforms are mostly specialized in a certain domain, resulting in limited job options. Our platform on the other hand, supports any kind of software development in any language available. It also combines the different fields of software engineering: design, development, integration and testing all in one place.

6. On the existing platforms, each project phase is done as a whole by one person, that might end up not getting paid and causing a demotivation for this person. Our platform splits each phase to multiple jobs and broadcasts them to the crowd, giving the chance to as many workers as possible to participate.

7. Quality issues arise on some other platforms resulting in client distrust. To overcome this problem, we introduced the rating system described earlier. Ratings vary based on tester's performance in a certain domain, and their contribution is always weighted. Less skilled testers cannot affect the overall rating of submissions and that is how we make sure that quality is guaranteed.

8. Cheating testers have been reported on testing platforms, and it was hard to stop the cheaters activity. The rating system also solves this issue, since any wrong or malicious act aiming for fast gains can be rewarded negatively, reducing the risk of having the cheater acting again on the platform.

9. Requirement extraction techniques relying on NLP can be adapted, but might cause problems rather than solving them. Therefore, having an old-fashioned communication between the requester and the crowd remains the better way. On our platform the requester is the project owner, and any component requirement he does not approve can be edited and reworked. If the requester lacks experience, the crowd designers and testers have to propose requirements and validate those requirements among themselves, in order to give the requester what he/she needs.

10. Its true that some specialized platforms have gained popularity over the years, but up till now, there is still no platform that fully combines all the aspects of the software engineering cycle, in a true crowdsourced fashion. Theoretical platforms do exist, but an implementation is still not yet available, unlike our platform which implements and tests the theory.

Chapter 9

CONCLUSION AND FUTURE WORK

Contents

9.1 Conclusion 68
9.2 Future Work 69

9.1 Conclusion

Crowdsourcing is a very strong potential rival to both software companies and freelancers in the future, due to the power and knowledge the crowds have to offer. The crowd being software developers, engineers, designers and more, can gather to make a difference on the web, like they do in their workspaces. Working on a crowdsourcing platform would give those people a chance to make extra money from the comfort of their homes, build their experience further more, and help requesters to satisfy their software needs.

The platform we proposed overcomes the downsides of some of the existing platform and proposes a new way of problem approaching and solving. If managed properly, any problem could be solved after being broken down into simpler, more manageable problems. And this is the case with software projects, they are large in scale and hard to be dealt with, but our platform facilitates the breaking down process through utilizing a component-based design. This design gave us the needed tools to simplify projects, increase the number of jobs and offered a simple architecture, allowing us to set the rules of automated crowdsourcing.

After thoroughly testing our platform, then using it to perform a real life case study, we proved the validity and effectiveness of our approach. The results we retrieved were both satisfactory and promising, perhaps due to the easy workflow and the friendly environment our platform provides. The reaction of our test crowd

towards the platform gave us the needed assurance that what is correct in theory, is also as correct in practice.

9.2 Future Work

The possible expenditures of the described platform are very broad. For instance, the project requester on the existing platform, sets the budget he/she wants to place for building his/her project in advance. This can be a tricky business, especially for people who have no expertise in the software industry. Applying machine learning algorithms to help predict the approximate budget would both help the requester with the confusion, and unify the payment criteria over the platform projects. This addition would require extra parameters to be set by the requester *and/or* by the crowd, in order to improve the accuracy of the algorithm. Machine learning could also be used to predict an extra factor: approximate *TTD*. Knowing the estimate *TTD* would improve job scheduling and give the requester a clear view of how the project building process will proceed. Different budget and TTD methods could be tested as well. Pay-as-you-go could be implemented on a unified job price, that is: jobs of a certain type, level, and hardship have the same price all over the platform. The more jobs needed to complete the job, the more money is paid by the requester. As for TTD, there would be no time limit and delivery would depend on the number of participants and their submissions.

As mentioned before, the design process results in a tree hierarchy where each node is a component. The next step would be redundancy elimination: finding redundant component specifications and removing their respective jobs. Such an action will reduce the design and development efforts needed, and thus testing and assembly. But it will also lead to payment and scheduling reassignments, development phase delay and hierarchy alteration, thus it requires further study.

A natural thing to do on such a platform is allowing group submissions or edits, such that a group of people make a unified submission and have the freedom to edit

it in a shared environment. In order to do that, a simple IDE extension could be added to the platform, so the group members can perform the edits on the spot. An extension of this idea would be a Wikipedia-like approach, where the posted code could be open for edits to all participants. To overcome the quality issues that might rise, the edited code could be part of a crowdsourced testing job to make sure of its correctness, before the edits are applied to the original code.

After running the platform for some time and having enough coded components, a components' pool can be constructed to fully utilize the component-based concept. By doing that, designers can refer to the pool and select a suitable code, facilitating the project building processes and saving the requester time and money.

References

[1] "Mvc design pattern," https://i-msdn.sec.s-msft.com/dynimg/IC263184.png, accessed: 2018-01-08.

[2] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *Journal of Systems and Software*, 2016.

[3] B. Fitzgerald and K.-J. Stol, "The dos and dont's of crowdsourcing software development," in *International Conference on Current Trends in Theory and Practice of Informatics.* Springer, 2015, pp. 58–64.

[4] A. M. Turk, "Amazon mechanical turk," *Retrieved August*, vol. 17, p. 2012, 2012.

[5] "utest website," https://www.utest.com/, accessed: 2018-01-08.

[6] "Appstori website," https://www.appstori.com/, accessed: 2018-01-08.

[7] K. R. Lakhani, D. A. Garvin, and E. Lonstein, "Topcoder (a): Developing software through crowdsourcing," 2010.

[8] I. Crnkovic, M. Chaudron, and S. Larsson, "Component-based development process and component lifecycle," in *Software Engineering Advances, International Conference on.* IEEE, 2006, pp. 44–44.

[9] S. Mahmood, R. Lai, and Y. S. Kim, "Survey of component-based software development," *IET software*, vol. 1, no. 2, pp. 57–66, 2007.

[10] A. Freeman, "Pro asp. net mvc 5 platform," in *Pro ASP. NET MVC 5 Platform.* Springer, 2014, pp. 3–8.

[11] "Ms sql server," https://www.microsoft.com/en-us/sql-server, accessed: 2018-01-08.

[12] "Entity framework," https://docs.microsoft.com/en-us/aspnet/entity-framework, accessed: 2018-01-08.

[13] "Razor syntax for asp.net," https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor, accessed: 2018-01-08.

[14] "Topcoder website," https://www.topcoder.com/, accessed: 2018-01-08.

[15] "Getacoder website," https://www.getacoder.com/, accessed: 2018-01-08.

[16] Y. Mu, Y. Wang, and J. Guo, "Extracting software functional requirements from free text documents," in *Information and Multimedia Technology, 2009. ICIMT'09. International Conference on.* IEEE, 2009, pp. 194–198.

[17] E. Letier and A. Van Lamsweerde, "Deriving operational software specifications from system goals," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering.* ACM, 2002, pp. 119–128.

[18] D. Sannella and M. Wirsing, "Specification languages," in *Algebraic Foundations of Systems Specification.* Springer, 1999, pp. 243–272.

[19] S. Apel, C. Kästner, and C. Lengauer, "Language-independent and automated software composition: The featurehouse experience," *IEEE Transactions on Software Engineering*, vol. 39, no. 1, pp. 63–79, 2013.

[20] A. Guerra-Hernández, J. M. Castro-Manzano, and A. E. F. Seghrouchni, "Ctl agentspeak (l): A specification language for agent programs," *Journal of Algorithms*, vol. 64, no. 1, pp. 31–40, 2009.

[21] "99designs website," https://www.99designs.com/, accessed: 2018-01-08.

[22] "Designcrowd website," https://www.DesignCrowd.com/, accessed: 2018-01-08.

[23] "crowdspring website," https://www.crowdspring.com/, accessed: 2018-01-08.

[24] W. S. Lasecki, J. Kim, N. Rafter, O. Sen, J. P. Bigham, and M. S. Bernstein, "Apparition: Crowdsourced user interfaces that come to life as you sketch them," in *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems.* ACM, 2015, pp. 1925–1934.

[25] T. D. LaToza, M. Chen, L. Jiang, M. Zhao, and A. Van Der Hoek, "Borrowing from the crowd: A study of recombination in software design competitions," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 2015, pp. 551–562.

[26] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: suggesting solutions to error messages," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2010, pp. 1019–1028.

[27] C. Watson, F. W. Li, and J. L. Godwin, "Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair," in *International Conference on Web-Based Learning.* Springer, 2012, pp. 228–239.

[28] M. Mooty, A. Faulring, J. Stylos, and B. A. Myers, "Calcite: Completing code completion for constructors using crowds," in *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on.* IEEE, 2010, pp. 15–22.

[29] T. D. LaToza, W. B. Towne, A. Van Der Hoek, and J. D. Herbsleb, "Crowd development," in *Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on.* IEEE, 2013, pp. 85–88.

[30] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich, "Touchdevelop: programming cloud-connected mobile devices via touchscreen," in *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software.* ACM, 2011, pp. 49–60.

[31] J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux, "Code hunt: Experience with coding contests at scale," in *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 2015, pp. 398–407.

[32] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar, "The jabberwocky programming environment for structured social computing," in *Proceedings of the 24th annual ACM symposium on User interface software and technology*. ACM, 2011, pp. 53–64.

[33] D. W. Barowy, C. Curtsinger, E. D. Berger, and A. McGregor, "Automan: A platform for integrating human-based and digital computation," *Acm Sigplan Notices*, vol. 47, no. 10, pp. 639–654, 2012.

[34] R. Auler, E. Borin, P. de Halleux, M. Moskal, and N. Tillmann, "Addressing javascript jit engines performance quirks: A crowdsourced adaptive compiler," in *International Conference on Compiler Construction*. Springer, 2014, pp. 218–237.

[35] R. A. Cochran, L. D'Antoni, B. Livshits, D. Molnar, and M. Veanes, "Program boosting: Program synthesis via crowd-sourcing," in *ACM SIGPLAN Notices*, vol. 50, no. 1. ACM, 2015, pp. 677–688.

[36] C. Schneider and T. Cheung, "The power of the crowd: Performing usability testing using an on-demand workforce," in *Information Systems Development*. Springer, 2013, pp. 551–560.

[37] "Microsoft lync," https://http://ofce.microsoft.com/lync/, accessed: 2018-01-08.

[38] "Chrome telemetry," https://http://www.chromium.org/developers/telemetry, accessed: 2018-01-08.

[39] "Firefox telemetry," https://https://telemetry.mozilla.org/, accessed: 2018-01-08.

[40] N. Chen and S. Kim, "Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2012, pp. 140–149.

[41] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification.* Springer, 2006, pp. 419–423.

[42] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th international conference on Software Engineering.* IEEE Computer Society, 2007, pp. 75–84.

[43] N. Tillmann and J. De Halleux, "Pex–white box test generation for. net," in *International conference on tests and proofs.* Springer, 2008, pp. 134–153.

[44] H. Xue, *Using redundancy to improve security and testing.* University of Illinois at Urbana-Champaign, 2013.

[45] M. Yan, H. Sun, and X. Liu, "itest: testing software with mobile crowdsourcing," in *Proceedings of the 1st International Workshop on Crowd-based Software Development Methods and Technologies.* ACM, 2014, pp. 19–24.

[46] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra *et al.*, "Caiipa: Automated large-scale mobile app testing through contextual fuzzing," in *Proceedings of the 20th annual international conference on Mobile computing and networking.* ACM, 2014, pp. 519–530.

[47] T. Xie, "Cooperative testing and analysis: Human-tool, tool-tool and human-human cooperations to get work done," in *Source Code Analysis and Manipula-*

tion (SCAM), 2012 IEEE 12th International Working Conference on. IEEE, 2012, pp. 1–3.

[48] R. Ali, C. Solis, M. Salehie, I. Omoronyia, B. Nuseibeh, and W. Maalej, "Social sensing: when users become monitors," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering.* ACM, 2011, pp. 476–479.

[49] "Stackoverflow website," https://www.stackoverflow.com/, accessed: 2018-01-08.

[50] B. Esselink, *A practical guide to localization.* John Benjamins Publishing, 2000, vol. 4.

[51] J. Lin, "Understanding and capturing people's mobile app privacy preferences," Ph.D. dissertation, Carnegie Mellon University, 2013.

[52] C. Arellano, O. Díaz, and J. Iturrioz, "Crowdsourced web augmentation: a security model," in *International Conference on Web Information Systems Engineering.* Springer, 2010, pp. 294–307.

[53] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices.* ACM, 2011, pp. 15–26.

[54] P. K. Chilana, "Supporting users after software deployment through selection-based crowdsourced contextual help," Ph.D. dissertation, 2013.

[55] P. K. Chilana, A. J. Ko, and J. O. Wobbrock, "Lemonaid: selection-based crowdsourced contextual help for web applications," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2012, pp. 1549–1558.

[56] P. K. Chilana, A. J. Ko, J. O. Wobbrock, and T. Grossman, "A multi-site field study of crowdsourced contextual help: usage and perspectives of end users and software teams," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2013, pp. 217–226.

[57] W. Ebner, M. Leimeister, U. Bretschneider, and H. Krcmar, "Leveraging the wisdom of crowds: Designing an it-supported ideas competition for an erp software company," in *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*. IEEE, 2008, pp. 417–417.

[58] R. Jayakanthan and D. Sundararajan, "Enterprise crowdsourcing solutions for software development and ideation," in *Proceedings of the 2nd international workshop on Ubiquitous crowdsouring*. ACM, 2011, pp. 25–28.