FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Simulation Framework for Task-based Crowdsourcing with Auctions

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering/Internet Computing

eingereicht von

Tobias Hammerer
Matrikelnummer 0526269

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ. Prof. Mag. Dr. Schahram Dustdar
Mitwirkung: Univ. Ass. Dr. Benjamin Satzger

Wien, 07.05.2012

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

A Simulation Framework for Task-based Crowdsourcing with Auctions

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering/Internet Computing

by

Tobias Hammerer

Registration Number 0526269

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ. Prof. Mag. Dr. Schahram Dustdar
Assistance: Univ. Ass. Dr. Benjamin Satzger

Vienna, 07.05.2012

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Tobias Hammerer
Ramperstorffergasse 27/25, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Danksagung

Diese Masterarbeit bildet den Abschluss der bisher wohl spannendsten, arbeitsintensivsten und doch schönsten Zeit meines Lebens, meiner Studienzeit. Viele wichtige Menschen haben mich durch diese ereignisreichen und wunderbaren Jahre begleitet, welchen ich an dieser Stelle danken möchte.

Der größte Dank gebührt meinen Eltern die mir diese universitäre Ausbildung in erster Linie ermöglicht haben. Ihr habt mich stets in jeglicher Art und Weise, sei es finanziell oder moralisch, unterstützt und mich jederzeit bekräftigt in dem was ich tue. Ohne euch wäre all dies nicht möglich gewesen und dafür möchte ich euch danken.

Des Weiteren möchte ich all meinen Freunden, vor allem dem "Sparverein", und meinen Studienkollegen danken die mir den benötigten Ausgleich zum Studium boten und mein Leben dadurch sicherlich um einiges lustiger und schöner gestaltet haben.

Nicht zuletzt möchte ich mich bei meinen Betreuern Dr. Benjamin Satzger und Prof. Dr. Schahram Dustdar bedanken die mich bei der inhaltlichen Gestaltung unterstützt haben. Die Besprechungen waren stets konstruktiv und auf Feedback musste ich nie lange warten, was nicht als selbstverständlich angesehen werden darf.

Abstract

Crowdsourcing describes a model for distributed online problem solving harnessing the creative solutions as well as the scalability and availability of the distributed network of individuals. A popular way of crowdsourcing where one can outsource a task to a crowd and receives a result in reasonable time is referred to as *task-based crowdsourcing* (TBCS). A multitude of different TBCS platforms is currently available. However, they have to cope with certain challenges. Firstly, they have to address the issue of the varying quality of user-processed tasks and secondly, they have to find solutions for distributing tasks in a way that all stakeholders get the highest attainable benefit. For the latter, the use of auctions as a means of task distribution seems to be an interesting approach, which has not been discussed in related literature yet. Another challenge in TBCS is the difficulty of testing and evaluating new approaches due to the scarcity of "real" data for evaluations.

The contribution of this thesis is twofold. Firstly, we address the lack of an appropriate simulation framework for task-based crowdsourcing and come up with a solution. We introduce a highly configurable, modular and extensible framework which is grounded on an agent-based modeling approach. The framework supports state of the art quality management methods based on a skill profiling module and various auction mechanisms to simulate task distribution methods. Secondly, we discuss the suitability of different auction types for various crowdsourcing scenarios. By means of the simulation framework, we simulate five different scenarios and compare three reversed standard auction types (sealed-bid, Dutch, English) and one double auction type (continuous double auction) with each other. Based on the analysis of quality, task distribution and payoff we find that among the standard auction types, the sealed-bid auction provides the most balanced results, whereas in comparison to the double auction market, the latter provides slightly better results in most categories. We have further analyzed three different quality management methods and come to the conclusion that the best quality can be achieved by using an auction mechanism that matches the task to the best suited worker. The most balanced approach, however, is to use a qualification policy.

Kurzfassung

Unter *Task-Based Crowdsourcing* versteht man die Auslagerung einer Aufgabe (*Task*) an eine Menge von unbekannten Akteuren (*Crowd*) unter Zuhilfenahme von Webtechnologien. Die Crowd bearbeitet die Aufgabe und stellt ein Ergebnis bereit. Plattformen, die dieses Service anbieten sehen sich mit verschiedenen Herausforderungen konfrontiert, insbesondere die Berücksichtigung von Qualitätsunterschieden der bearbeiteten Tasks sowie die Erzielung des für alle Beteiltigten höchst möglichen Nutzens durch eine angemessene Aufgabenverteilung. Auktionen stellen hierbei einen interessanten Ansatz dar, der in der Fachliteratur in diesem Zusammenhang bisher noch nicht eingehend analysiert wurde. Das Testen und Evaluieren neuer Ansätze kann als eine weitere Herausforderung identifiziert werden, zumal der Zugang zu "echten" Daten oftmals mit Schwierigkeiten verbunden ist.

In dieser Masterarbeit wird zunächst ein hochgradig konfigurierbares, modulares und erweiterbares Simulations-Framework vorgestellt, welches dem Ansatz der Agenten basierten Modellierung folgt. Das Framework unterstützt dabei aktuelle Qualitätsmanagement-Methoden (basierend auf einem *"Skill Profiling"* Modul) und verschiedene Auktionsmechanismen (zur Verteilung der Aufgaben). Anschließend wird die Eignung verschiedener Auktionsmechanismen für diverse Crowdsourcing-Szenarien untersucht. Darüber hinaus werden drei Standard- Auftragsauktionen (*Reverse Auctions*) und eine *Double Auction* einem Vergleich unterzogen. In Hinblick auf die Auswertung von Qualität, Taskverteilung und Profit erzielt die *Reversed Sealed-Bid* Auktion in Summe die besten Resultate. Im Vergleich zu den Standard Auktionsmechanismen, liefert die *Continous Double Auction* in den meisten Kategorien etwas bessere Ergebnisse. Zudem wurden drei verschiedene Qualitätsmanagement-Methoden einer Analyse unterzogen. Dabei kann festgehalten werden, dass die Verwendung einer Auktion, die die Aufgaben den qualifiziertesten Usern zuteilt, die höchsten Qualitätswerte erzielt.

Contents

List of Figures

List of Tables

List of Listings

CHAPTER 1

Introduction

Within the last few decades the World Wide Web has rapidly evolved from a network offering basic communication services to a platform offering a vast number of different services. In the wake of the dotcom bust, *Web 2.0* arose and changed the common understanding of the Internet. *"Network as a Platform"* is one central idea of Web 2.0, meaning that web applications should go further than just providing web accessible "desktop applications". They should be built especially for the web, tapping the potential of this new platform, namely its users. Simply put, a web application should get better and more interesting, the more people use and contribute to it [50]. Using this insight, the World Wide Web moved into a new phase where various systems – providing this new functionality – emerged. Users, for example, were now able to connect with each other and share personal information using services like *Facebook*, *MySpace* or *Twitter* (to name only a few) [21, 45, 64]. They were able to collaborate and contribute their knowledge to create new contents whose total value exceeds the sum of values provided by the single users (as it is the case on *Wikipedia* [69] for example) [40, 50].

Along with these developments, a new model for distributed on-line problem solving harnessing the creative solutions as well as the scalability and availability of the distributed network of individuals was born [11, 67]. This model called *crowdsourcing* was first mentioned in [28] by Jeff Howe. It describes a web-based outsourcing strategy consisting of the idea that tasks being traditionally processed by employees, are now offered to a so called *crowd* – an unknown but large group of people on the Internet. By offering various incentives the crowd is encouraged to process the offered tasks. Incentives may be monetary, intrinsic (e.g. enhancing reputation in a specific community, acquiring new skills, . . .) or both [5, 29].

There is a multitude of different crowdsourcing platforms available which can be categorized, according to Vukovic [67], based on the *Crowdsourcing Function* and the *Crowdsourcing Mode*. The crowdsourcing function refers to the type of environment the task or product is being crowdsourced in. More information and examples regarding this point will be provided in Chapter 2. The crowdsourcing mode may be either *Contest* or *Marketplace*. Contest-based platforms refer to crowdsourcing systems where users submit their solutions but only the best solution will win and be rewarded. *Threadless.com* [61] constitutes a representative example of this category. Users are encouraged to design their own T-shirt and submit their design to the website. Then the online-community is invited to vote for their best design. The design with the most votes will win the contest, the winner will get a monetary reward and T-shirts with the winning design will be sold on the website [11].

Contrary to contest-based platforms, tasks submitted to marketplace-based platforms are advertised as bidding items and therefore can be assigned to single crowd members who will be rewarded (assuming the tasks are processed successfully). This thesis mainly focuses on the latter type of crowdsourcing systems which is often referred to as *Task-Based Crowdsourcing*. Figure 1.1 (based on [67]) shows the different agents and their possible actions in a task-based crowdsourcing system which will be explained using the following example.

The *Amazon Mechanical Turk* (AMT) [3] is a very prominent representative for task-based crowdsourcing. Employers on the AMT, so called *requesters*, are encouraged to submit tasks, so called *human intelligence tasks* (HITs), to the system. Those HITs mostly require minor effort – most of them can be processed in a few seconds up to a few minutes - but still need human intelligence. As a recent study [32] has shown: transcription, classification and categorization tasks are the most occurring HITs offered

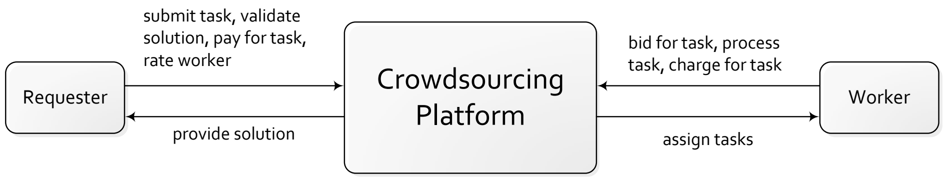

submit task, validate solution, pay for task, rate worker
Requester
provide solution
Crowdsourcing Platform
bid for task, process task, charge for task
Worker
assign tasks

Figure 1.1: Agents and actions in a task-based crowdsourcing process

on AMT. The crowd of employees, so called *workers*, pick up these HITs, complete them and get a monetary reward (normally a few cents per HIT) if the quality of their solution is accepted by the requester [32].

Task-based crowdsourcing obviously depends on the crowd, but the crowd also embodies a big challenge for those systems. On the one hand, workers worldwide from different cultures and time zones can register to those platforms and compete for tasks, providing highly available workforce. This in turn is an incentive for companies to outsource tasks to the system. On the other hand, this mix implies varying skills and performances among workers leading to the possibility that workers might not be qualified for processing a given task. As a consequence, the variation in quality of provided results is high [2, 54]. Furthermore, studies based on the AMT have shown that, not only missing skills or low performances have impact on the quality, but also the behavior of *malicious workers*. They exploit the fact that manually verifying the quality of the submitted results is hard, and therefore submit low quality solutions [32, 33].

Literature shows various approaches to cope with the issue of varying quality. Agichtein et al. [2] describe a method for automatically identifying high quality content in social media based on content, link analysis and user rating. They state that *Yahoo! Answers* [74], a famous question & answer crowdsourcing platform, is one example where their algorithm could be applied on.

As a representative example for task-based crowdsourcing, the AMT provides two strategies to cope with quality issues. Firstly, requesters may require workers to pass a *qualification test* in case special skills are needed for processing a task. Secondly, requesters have to pay for quality work only (e.g. correct solutions) [4]. As mentioned earlier, it is hard to manually verify the quality of results. Therefore, many requesters rely on redundancy by submitting the same task to several workers in order to identify correct solutions. Ipeirotis et al. [33] state that this method is *1) expensive* for workers and *2) biased* by malicious workers. In their paper, they describe an algorithm based on the work of [14], which is capable of analyzing the quality of workers with respect to error and bias.

The necessity of creating qualification tests to ensure worker skills is another restriction of this AMT quality assurance plan. On the one hand, every time a requester is submitting a new sort of HIT that needs special skills, a new test has to be designed. That needs valuable time and might be tedious. On the other hand, workers have to attend and pass various qualification tests before being allowed to work. That in turn may cause unnecessary overhead for workers and could come in conflict with the aforementioned principle of highly available workforce on demand. Satzger et al. [54] discuss an automated crowdsourcing marketplace capable of estimating the skills of its workers based on a confidence management model. One interesting point in their work is the *matching* of tasks to workers. Matching on common systems, like the AMT, is dependent on the worker who simply selects a task of interest available on the platform. In the approach provided by [54], task matching is business of the marketplace itself. By means of an auctioning system, workers are invited to bid for tasks they are interested in. The winner of the auction is determined based on the worker's quality rating and the bid price.

1.1 Motivation

The aim of each task-based crowdsourcing system is to satisfy its clients (i.e. requesters and workers). By means of a large number of active clients over a long period of time, profit can be gained. Requesters need to obtain high quality results in reasonable time at a reasonable price to be satisfied. On the other hand, workers are interested in gaining high rewards for completing tasks and they will only stick to the platform as long as tasks are available [54].

The challenges of task-based crowdsourcing platforms can be summed up as follows:

- *Varying Quality:* The quality of processed tasks is dependent on the worker's skill level and attitude (e.g. malicious workers). Strategies to cope with parts of this issue are available but prominent platforms like the Amazon Mechanical Turk only provide limited possibilities to ensure high quality results.

- *Task Distribution:* A common way of task distribution is that a worker selects its preferred task among a list of many, knowing the (predefined) price he will get in case of successful completion. This may have restrictions: Firstly, there is no guarantee that a task is processed by a well or even a perfectly-suited worker. Secondly, since prices are predefined both parties might not get the maximum benefit according to their own task valuation.

Regarding the task distribution, the above mentioned method is justifiable, even preferable for platforms like the AMT where micro-tasks are traded. In case one wants to trade larger, costlier tasks (e.g. small software projects) other distribution systems may be interesting. In [54], a new way of task distribution is introduced, using an auction mechanism that matches tasks to workers based on the workers qualification and its bid price.

Auctions are a very popular way of trading goods on the Internet, with *eBay* [18] as its most prominent representative. However they are neither used in current task-based crowdsourcing systems, nor have been paid attention to in relevant literature.

As mentioned above, using auctions for task distribution and price negotiation is an interesting new approach in this area. Thus the suitability of different auction mechanisms in a task-based crowdsourcing environment should be analyzed.

One problem we are facing is the difficulty of testing and evaluating new approaches in crowdsourcing due to the scarcity of “real” data for evaluation. Obtaining data from an existing crowdsourcing platform might be applicable if one wants to analyze general facts (e.g. user studies). When it comes to testing new paradigms the most reasonable way is to create a model of the new approach and run simulations on it.

To the best of our knowledge there is currently no simulation environment focusing on task-based crowdsourcing available. By means of a modular, highly configurable and extensible framework, various different crowdsourcing scenarios could be tested and analyzed.

1.2 Contribution

In this thesis we address the issues mentioned in Section 1.1. Thereby, providing an analysis of the suitability of different auction mechanisms in task-based crowdsourcing platforms through a highly configurable, modular and extensible agent-based modeling framework, especially designed for such platforms. In particular the contribution of this thesis is twofold:

- A simulation framework suited to the needs of a task-based crowdsourcing environment based on the *Java Agent-Based Modeling (JABM)* toolkit [35] is built. Key features of this new framework can be identified as follows:
  - Highly configurable due to the use of dependency-injection
  - Highly extendable due to clear defined interfaces and an open design (e.g. possibility to implement workers organized in teams processing nested tasks)
  - Supportive of skill recognition strategies to cope with quality issues
  - Supportive of various auction types for task distribution
  - Supportive of various reports for analysis
  - Agent-based approach, i.e. each agent follows its own strategies and rules (e.g. trading, pricing, etc.)

- A suitability analysis of four commonly known auction mechanisms[1]: *Sealed Bid Second Price Auction, English Auction, Dutch Auction, Continuous Double Auction* is provided. The auction mechanisms are used for task distribution in a task-based crowdsourcing system. Based upon data obtained from several simulation runs utilizing the aforementioned simulation framework, we discuss advantages and disadvantages of each auction type in a given context. We analyze the data based on the following criteria: quality of processed tasks, payoff of the agents and throughput based on assigned, unassigned and completed tasks as well as tasks with deadline violations.

1.3 Organization

The remainder of this thesis is structured as follows:

- Chapter 2 details the current state of the art and provides basic knowledge. At first an basic introduction in auction theory is given. Then, the term *crowdsourcing* is defined and a basic overview of current crowdsourcing platforms is provided. Finally, essentials of agent-based modeling are discussed.

- Chapter 3 provides an overview of related work in the area of task-based crowdsourcing (TBCS) and simulation. Different quality management approaches in TBCS are discussed, task matching methods are detailed and information on basic strategies to recruit and retain users is given. Lastly, two different agent-based modeling frameworks are introduced.

- In Chapter 4 the architecture, the theoretical models and the approaches used to implement the simulation framework are detailed. After information on the simulation model is provided, models used to define the behavior of agents are discussed. This chapter further covers information on the design of the marketplace and simulation specific components.

[1]Detailed information regarding those auction types will be provided in Section 2.1

- Chapter 5 covers the implementation aspects of the simulation framework prototype. At first, the underlying agent-based simulation framework (JABM) is outlined and the integration in our framework is described. Then, selected components are discussed and the implementation of the simulation model is detailed.
- Chapter 6 contains an evaluation of the suitability of different auction types for a given crowdsourcing scenario. In detail, five different scenarios are simulated and four different auction types are compared. For each scenario a discussion based on the simulation results is provided.
- Finally, Chapter 7 provides a conclusion of the thesis and suggestions for further improvements.

CHAPTER 2

State of the Art Review

This chapter reviews the current state of the art in crowdsourcing and discusses basic concepts of auction theory and agent-based modeling techniques.

2.1 Auction Theory

In this section, an overview of commonly known and widely used auction types is given. Jap defines an auction as

> *"[. . .] a market institution with an explicit set of rules determining resource allocation and prices on the basis of bids from market participants."* [36]

With the set of rules defining the behavior of an auction, various different auction types can be identified. These auction types may differ in several factors such as the way the price is determined, the number of items traded in an auction or the number of participants trading in an auction [39]. On basis of the latter, we can divide auctions into three categories:

1. *Standard (or Demand) auctions:* A single seller v offers a good and multiple buyers c compete for it pushing the price up (see Figure 2.1(a)).
2. *Reverse (or Supply) auctions:* The roles of sellers and buyers are reversed, as a single buyer c offers a business to multiple sellers v who compete with each other

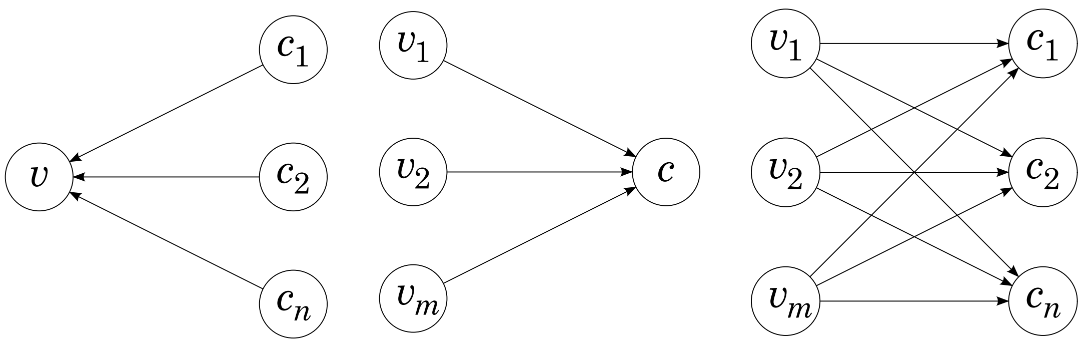


Figure 2.1: Auction types categorized by the number of participants, taken from [70]

by pushing the price down, in order to sell their workforce and get the business assigned (see Figure 2.1(b)).

3. *Double Auctions:* Multiple buyers c submit their bids and multiple sellers v simultaneously submit their asks. If a bid exceeds an ask, a transaction is consummated between the two participants (see Figure 2.1(c)).

2.1.1 Valuations

One key feature of auctions is that bidders and sellers are uncertain about the exact value of an object being sold. In auction theory this is called asymmetric information. Two basic valuation models can be distinguished [38, 39]:

1. *Private-Value Model:* In this model each bidder knows his value of the object being sold. This information is private to the bidder himself which implies that no bidder knows the value other bidders attach to the object. Further, knowledge about the private value of others would not effect the own private value. According to [39], using the private value model is most plausible in case a bidder derives the value of an object from its consumption or use. For example, a painting may have different values to different bidders.

2. *Pure Common-Value Model:* In this model the value of the goods is the same for everyone (once revealed), but at the time of the auction, this actual value is

unknown to the bidders. Each bidder has different information about the actual value and the information of one bidder would be informative to another bidder thereby affecting his valuation. A typical example would be the sale of a piece of land with an unknown amount of oil underground. At auction time, the amount of oil is unknown and bidders may have different estimates based on geological tests. The real value of the land, however, is derived from the future sales of the oil.

2.1.2 Standard Auction Types

In standard auctions, which are also referred to as demand auctions, the bidder submitting the highest price wins the auction. The methodology to obtain bids, however, may vary. As stated in [39] and [38], four basic auction types that are widely used can be distinguished:

- *English Auction:* The English auction is also called ascending-bid or open auction. In this auction type, a seller determines a *reserve price* which equals the start price of this auction. The price is then successively raised either by an auctioneer (in small increments) or by the buyers themselves (by submitting bids). The auction runs until only one bidder remains who wins the traded good at the final price. In this auction type, each bidder is at all times aware of the current highest bid. One prominent example of an online marketplace using this type of auction is *eBay.com* [18].

- *Dutch Auction:* The Dutch auction is also called descending-bid auction. In this auction, an auctioneer basically starts at a very high price where presumably no bidder is interested in buying the traded good. By successively lowering the price, bidders are encouraged to participate in the auction. The first bidder accepting the current price wins the good. As in the English auction, each bidder is at all times aware of the current highest bid.

- *Sealed-Bid First-Price:* In contrast to the former two auction types, bidders are not aware of the bids of other participants. Therefore, in this auction each bidder submits a single bid independently of the bids of the other competitors. The bid with the highest price eventually wins the traded good.

- *Sealed-Bid Second-Price:* This auction is also referred to as Vickery auction. The procedure is basically the same as in the sealed-bid first-price auction with the difference that the winner has to pay the price of the second-highest bid.

Two out of those four auction types are *open* auctions, namely the English and Dutch auction, and two are *sealed-bid auctions*, the first-price and second-price formats. In game theory, the Dutch auction and the sealed-bid first-price auction are considered *strategically equivalent*. This means that the set of strategies available in the Dutch auction is the same as in the sealed-bid first-price auction [38]. More specifically, to win a Dutch auction a bidder has to submit the first bid. He must therefore determine a price (i.e. the private value) at which he places a bid in the auction, provided that no other bid has been submitted yet. The same strategy of choosing a price can be applied in the Dutch auction as in both auctions the highest bid wins and the price of the winning bid constitutes the price the agent has to pay. The fact that the Dutch auction offers information about current prices is not useful for bidders, as once this information is available, the Dutch auction has already determined a winner.

Bidders trading in the English or the Ssealed-bid second-price auction also have to choose the price (i.e. *private value*) they are willing to pay to obtain goods. In contrast to the former two auction types, this private value need not be equal to the price that winning bidders finally have to pay. In the English auction, it is sufficient to stay in an auction until one is the last remaining bidder. The winning price therefore equals the price of the second highest bidder, i.e. the price of the last bidder leaving the auction. Again, this is the same as in the sealed-bid second-price auction and hence these both auction types are considered strategically equivalent. However, as stated in [39], this equivalence is weaker than the one between the sealed-bid first-price auction and the English auction since the English auction offers information of other bidders dropping out and thus the remaining bidders may be able to infer something about their privately known information.

2.1.3 Reverse Standard Auction Types

As mentioned above, in reverse auction types, the roles of buyers and sellers are reversed. Buyers aim to buy workforce in the auction at a low price and sellers try to sell their workforce by pushing the price down. The same four standard auction types described in the previous section can be distinguished. However, slight alterations in the auction process have to be made:

- For the two *sealed-bid auction* types, the only alteration to the auction process is that the lowest bid, in contrast to the highest bid, wins the auction.
- The *English auction* starts at a high price which is equal to the buyer's reservation price. This price is then successively decreased until only one seller remains.
- The *Dutch auction* starts at a very low price which is equal to the buyers reservation price. This price is then successively raised until the first bid is submitted.

2.1.4 Double Auction Types

In standard and reverse auctions, one seller (or buyer) accepts bids from multiple buyers (or sellers) which is why these auction types are also called *one-sided*. In contrast to that, double auctions can be categorized as *two-sided* auctions as multiple buyers and sellers place their bids in an auction to trade homogeneous goods. Friedman [24] provides a survey on double auction theory and discusses multiple variants of the *Continuous Double Auction* (CDA). In its basic version, the CDA immediately matches bids of buyers and sellers on detection of compatible bids, i.e. when the price of a given bid exceeds the price of a given ask. In contrast to that, the *Clearinghouse Auction* (CH) type should also be mentioned as it collects bids over a certain period of time and matches buyers and sellers at the end of this period.

Further work on double auctions for electronic commerce has been done by Wurman et al. [73]. Besides analyzing economic incentives, they introduce an efficient *4-heap algorithm* to process incoming bids and calculate allocations. The algorithm uses four heap structures: two for buy offers (bids), and two for sell offers (asks). Each of them can be further divided into two heaps: one heap containing matched shouts and one heap containing unmatched shouts.

2.2 Crowdsourcing

The term *crowdsourcing* was first mentioned by Jeff Howe in 2006 in an issue of the *Wired magazine* [28] where he described it as a new web-based outsourcing strategy. Particularly, he defines crowdsourcing as

> *"[...] the act of taking a task traditionally performed by a designated agent (such as an employee or a contractor) and outsourcing it by making an open call to an undefined but large group of people. Crowdsourcing allows the power of the crowd to accomplish tasks that were once the province of just a specialized few."* [29]

The main idea of this open call is that there are no restrictions regarding the people addressed. Everybody may work on a task provided by a crowdsourcing platform. As a consequence, the *crowd* consists of people with different interests, skills and cultures. However, Howe further notes that this crowd primarily consists of *professional amateurs* who are knowledgeable and educated, committed and networked. Their motivation of contributing their knowledge and skill set mostly does not come from monetary, but rather from intrinsic incentives. They contribute as they want to enhance their reputation in a specific community or as they get the possibility to acquire new skills. The monetary reward is mostly of secondary importance [29].

Basically, two groups of crowdsourcing platforms can be distinguished:

- *Contest:* In contest-based platforms, a task is assigned to multiple users. Many users may process this task and submit their solutions to the platform but only the best solution will win and be rewarded. Even though rewards range from a million dollars for improving the performance of a video recommender system [46] and a thousands of dollars for solving a research problem on *innocentive.com* [31] to a few hundred dollars for developing a software component [63], “real” professionals typically are not represented in the crowd [5]. Firstly, they are normally not able to make a living by solely working for such contest-based platforms and secondly they are simply not willing to put resources and work in something they probably will not be rewarded for. Hence, the crowd on such platforms mainly consists of the aforementioned professional amateurs. A game-theoretic approach to model such contests is provided by DiPalatino and Vojnovic in [15], as they represent contest-based platforms as all-pay auctions. In an all-pay auction, every participant has to pay the price he submits to the auction, but only the highest price wins the auction. In crowdsourcing contests, this price is represented by the effort the participants put into creating their solutions. A typical representative for a contest-based crowdsourcing platform is *threadless.com* [61]. Users are encouraged to create and submit T-shirt designs. Among all submissions, the online community can vote for their favorite and the design obtaining the most votes is rewarded. T-shirts with the winning design can be bought on the website.

- *Marketplace:* In marketplace-based platforms, tasks are advertised as bidding items. Platform users may bid on tasks and start working once they get a task assigned. In contrast to contest-based platforms, each user processing a task gets paid (assuming the tasks are processed successfully) and not just the user providing the best solution. Marketplace-based platforms are often referred to as *Task-based Crowdsourcing* (TBCS) platforms. One prominent example for TBCS is the *Amazon Mechanical Turk* (AMT). On the AMT, users submit tasks, so called *human intelligence tasks* (HIT) to be processed by other users. These HITs require human intelligence and most of them can be processed in a few seconds up to a few minutes. As a consequence, the rewards for those HITs range between a few cents up to a few dollars [32]. A detailed description of a typical task-based crowdsourcing process is provided in Section 4.1.

Vukovic [67] states that crowdsourcing platforms can be further classified by, what he calls, a *crowdsourcing function*. This refers to the part of the product that is being crowdsourced. Returning to *threadless.com*, for example, the part of the product life cycle being outsourced is the design process. Therefore, the crowdsourcing function for this platform is design. In Table 2.1, a selection of current crowdsourcing platforms categorized by their crowdsourcing function is listed.

Design and Marketing	
Threadless	sales T-shirts designed by the crowd.
99designs	sales designs for logos, websites, mobile apps, etc.
IStockPhoto	trades royalty free pictures made by professional amateurs.
Research and Innovation	
InnoCentive	uses open innovation for problem solving
One Billion Minds	solves problems in science, technology, design, business or social innovation.
Collective Intelligence & Prediction	
Inkling Markets	uses wisdom of the crowd for forecasting.
We Are Hunted	lists and predicts music charts.
Yahoo!Answers	provides answers to questions.
Human Resources	
Amazon Mechanical Turk	offers low-cost crowdsourcing of micro tasks.
TopCoder	offers competition-based software crowdsourcing.

Table 2.1: A selection of current crowdsourcing platforms

99designs [1] is a crowdsourcing platform selling customized design packages for websites, brochures or flyers to name a few. *IStockPhoto* [34] sells royalty free pictures, animations and video clips that may be used for example on websites or in presentations. Sample platforms that offer outsourcing of the research and innovation process are *InnoCentive* [31] and *One Billion Minds* [49]. InnoCentive is one of the more prominent platforms where scientific problem solving is accomplished in a crowdsourced way using the *open innovation* paradigm. According to [13], open innovation means that companies should use internal and external ideas to advance their technology, as they cannot afford to rely entirely on their internal research in a world of widely distributed knowledge. *One Billion Minds* [49] takes a similar approach and offers solutions to problems in science, technology or design to name a few. *Inkling Markets* [30] uses the *wisdom of the crowd* for forecasts. It tries to help organizations to decrease operational and strategic risk by the use of prediction markets and opinion polls. *We Are Hunted* [68] uses the crowd (and their social networks) to list songs and artists that are popular and enables forecasts on emerging tracks. *Yahoo!Answers* [74] allows users to submit questions which are then answered by the crowd. Finally, *TopCoder* [63] uses a contest-based approach to develop software products.

2.3 Agent-based Modeling

Agent-based modeling (ABM) is used to simulate complex systems of interacting individuals. Individuals in such simulations are called *agents*, and each agent has its own behavior responsible for the actions and interactions an agent takes at any given time. Behavioral rules may range from simple "if-then" rules to complex artificial intelligence techniques that adapt and alter the agent's behavior over time. To introduce randomness, ABM often uses *Monte Carlo Methods* [9], which rely on repeated random experiments, to compute results [52]. Bonabeau [10] states that ABM has three benefits compared to other modeling techniques. Firstly, it captures *emerging phenomena*. Secondly, it provides a *natural description* of a system and lastly, it is *flexible*. Emerging phenomena, like behavioral patterns or structures, arise through interactions between agents. *"[. . .] the whole is more than the sum of its parts [. . .]"* [10] is how Bonabeau describes these emerging phenomena. This means that they cannot be reduced to the single parts of a system as they only occur as a result of interactions. As a consequence, these phenomena may have properties decoupled from the properties that are explicitly programmed into the model. One may consider a traffic jam as an example. Vehicles are moving forward, but through their interactions, a resulting traffic jam moves in the opposite direction. ABM describes the behavior of a single agent in a system instead of describing the behavior of the whole system (for example by the use of differential equations). Thus it provides a very natural way of describing and simulating complex systems.

The area of applications of agent-based modeling is widely spread. Models range from simulations of stock markets [6] to traffic flows [10] and the prediction of spread of epidemics [7] and many other phenomenas in social science [19].

2.3.1 Components of an Agent-based Model

According to Macal [42], an agent-based model in its simplest form consists of:

1. A set of *agents* with defined attributes and behavior.
2. A set of agent *relationships* for defining possible interactions.
3. The agents' *environment* they may interact with in addition to other agents.

He further states that, from a practical modeling standpoint, agents may be described by the following characteristics:

- *Autonomous*: The most important characteristic of agents may be the one that they are considered *autonomous*. They act and react to situations they observe on their own and make decisions based on their implemented behavioral rules. Their behavior may be specified by anything that transforms the agent's input to an according output. This can range from simple rules to abstract models (e.g. neuronal networks, artificial intelligence heuristics, etc.).
- *Self-contained*: Each agent is self-contained, modular and uniquely identifiable which allows agents in a system to be recognized by others.
- *Stateful*: Each agent has a state, described by various attributes. The agent's state may determine his behavior and through interactions with other agents this state may change over time.
- *Social*: Agents dynamically interact with other agents and their environment. To enable these interactions agents possess implemented protocols. Examples for interactions may comprise communication with other agents or movement in the environment.

- *Adaptive:* As mentioned earlier, agents implement rules defining their behavior. Some agents may additionally have the ability to learn based on their experiences. Thus, they may implement rules to change their behavior over time. In addition to this individual adaption, populations may adapt to their environment through the process of selection as the number of better suited agents may increase while the number of worse suited agents may decrease over time.

- *Goal-directed*: Agents may have goals they pursue, and thus implement functionality to compare their results relative to their goals and adapt their behavior, for further interactions.

- *Heterogeneous*: In the real world, individuals are different as each individual has its own behavior and attributes. In ABM, some models may also consider this diversity and simulate a population consisting of various agents each with its own behavior.

The environment may be used to define the agent's relative location to other agents, or to change the agents state as they interact with the environment.

CHAPTER 3

Related Work

This chapter provides a basic overview on related work in the area of task-based crowdsourcing (TBCS). At first, methods to cope with varying quality in TBCS environments are detailed and different approaches for task matching are introduced. Further, advanced aspects of crowdsourcing are discussed including user management strategies and game-theoretical approaches. Then, related work in the area of social computing is introduced. At last, a selection of current agent-based modeling toolkits is discussed.

3.1 Quality in Crowdsourcing Platforms

Although task-based crowdsourcing platforms provide a very popular way for companies to outsource certain tasks, these platforms have their limitations. One challenge they have to face is the varying quality of user processed tasks. As recent studies [32,33] on the *Amazon Mechanical Turk* (AMT) [3] have shown, the variation in quality is high. According to [33], this variation may have two reasons. Firstly, due to the open call nature of crowdsourcing platforms, workers worldwide from different cultures and time zones may provide their workforce on task-based crowdsourcing platforms. This diversity implies varying skills and performances among workers which in turn may cause a task assignment to an unqualified worker. Secondly, the attitude of workers may influence the quality of tasks. For example, on the AMT various transcription tasks are

Term	*Description*
Requester	Embodies the employer on the AMT as he submits tasks workers have to process.
Worker	Embodies the employee on the AMT as he processes tasks submitted by requesters.
HIT	Abbreviation for human intelligence tasks. The tasks submitted by requesters and processed by workers

Table 3.1: Terminology of the Amazon Mechanical Turk

offered. Such tasks mostly consist of a picture containing some text (that cannot be recognized using computational methods) which workers have to type in a submission form. Verifying the results of submitted tasks is expensive since one would have to check each result manually. *Malicious workers* exploit this fact and may submit the same result for multiple HITs.

Various methods to cope with quality issues are known in literature. A common method to ensure quality in a task-based crowdsourcing environment is to estimate the qualification of a worker for a certain task. This may be accomplished by different techniques:

The AMT[1], for example, provides qualification tests to check certain skills which each *worker* has to pass before being allowed to work on a *HIT*. These tests however comprise certain disadvantages. Firstly, *requesters* have to create the tests themselves which takes valuable time and might be tedious. Secondly, workers have to attend these tests before being allowed to work which, again, is tedious and might conflict with the crowdsourcing principle of highly available workforce on demand. Lastly, passing a test does not guarantee high quality as malicious workers may provide bad quality despite passing a test [33].

Another technique to estimate the qualification of workers for tasks is to estimate the workers skills and then infer their qualification. Based on the work of Dawid and Skene [14], Ipeiros et al. [33] provide an algorithm designed for the use on the AMT.

[1]Table 3.1 provides an overview of the terminology used by the AMT

Based on the results of multiple workers, they infer a correct solution and by comparing the inferred solution to the workers results they are able to draw a conclusion on the quality of each worker.

Khazankin et al. [37] provide a quality management approach based on Service Level Agreements (SLA) negotiated between the marketplace and the requesters. In their approach, the crowdsourcing platform possesses a profile of each registered worker, containing a description of the worker's skills and his current availability. The skills of workers are continuously updated by a profiling component based on the feedback (of requesters) on processed tasks. On task submission, requesters have to specify skill requirements and a deadline. In compliance with this provided information and the negotiated SLAs the platform then assigns a suited worker to the task. The suitability is calculated based on the worker's skill profile and his current availability by means of a scheduling component which has the objective to maximize quality while fulfilling deadlines.

Satzger et al. [54] tackle the issue of varying quality by introducing an auction mechanism to match tasks to workers based on a confidence management model. As Khazankin et al., they use a profiling component that tracks the skills of workers, but their profiling mechanism provides additional functionality. Firstly, for each skill in the worker's profile, an additional confidence value is provided defining the confidence the platform has in a given skill. Secondly, the profiling component provides functionality to bootstrap new skills. Skill values are, again, derived from the feedback a requester provides for a processed task; each processed task increases the confidence. As in the previous approach, requesters have to provide information on skill requirements and deadlines. Once submitted, a task is advertised by means of a modified sealed-bid auction mechanism. Workers suiting the quality requirements are invited to join the auction and place bids. Among all placed bids a winner is determined on the basis of the observed skills, the confidence and the bid price. In order to bootstrap new or low skills "test-tasks" are assigned to workers. These test-tasks are copies of tasks that have already been assigned to high-skilled workers, which enables the system to infer new skills by comparing the results of the test-task to the real task.

Finally, an automatized method to find high quality content in social media is described in [2]. Agichtein et al. classify the quality of the content based on the analysis of the content itself, the user rating and a link analysis between users.

3.2 Task Matching on Task-based Crowdsourcing Platforms

The task matching process describes the procedure by which a given task is assigned to a given worker. The simplest form of task matching can be observed on the Amazon Mechanical Turk. Tasks submitted by requesters are advertised in a list which is visible to registered workers. The task matching process is initialized and executed by the worker as he simply selects a task of interest.

An altered and more sophisticated task matching approach is presented by Yuen et al. [76]. Again, a worker can select a task of interest from a list. The list, however, is sorted providing best matching tasks first. The matching is calculated on the basis of the worker's selection and performance history. According to [76], the advantages of this approach are twofold. Firstly, the quality of results on the platform can be improved as workers perform better if they are familiar with a task. Secondly, using this matching method encourages users to work on the platform in the long run, as they remain motivated if they find suited tasks easily.

In the aforementioned two approaches it is the worker initiating and executing the task matching process. In contrast to that, the task matching process described in [54] is initiated by the worker but the execution is accomplished by the crowdsourcing platform. In the approach of Satzger et al., tasks are advertised to workers as bidding items. In case a worker is interested in a task he may place a bid in the corresponding auction. After a defined period of time the auction matches the task to a worker based on the worker's bid price, the monitored performance and the confidence in the worker.

In [37] the task matching process is initiated and executed by the crowdsourcing platform. Workers solely inform the platform about their availability and in case the platform has a task to process, it assigns this task to the best matching worker. The matching depends on the monitored performance of a worker and its availability.

The approaches described above are mainly built to match a single task to a single worker. However, large-scale tasks (consisting of subtasks and workflows) may need to be assigned to multiple workers. Skopik et al. [57] describe such a scenario where workers collaborate in context of joint task. In their approach, they monitor the social interactions in the crowd and assign the tasks based on these interactions.

3.3 Advanced Aspects of Crowdsourcing

This section focuses on two further aspects of crowdsourcing. At first, information on how to categorize and analyze crowd members is provided. Further, an overview on recruitment strategies as well as on encouragement and retention schemes is provided. Lastly, two papers discussing a game-theoretic approach to crowdsourcing are introduced.

3.3.1 User Management

Crowdsourcing platforms are dependent on the crowd and it is therefore important for platform providers to understand which types of users they attract in order to be able to recruit new and retain existing users. Doan et al. [16] have analyzed multiple current crowdsourcing platforms and find that users can be divided in four different categories based on their roles:

- *Slaves*: Users providing their workforce to solve simple and small problems are called slaves. These types of users may be found on the AMT or in the ESP game [65] where users implicitly collaborate labeling images while playing the game.
- *Perspective providers*: These users may be found an platforms utilizing the *wisdom of the crowd* [58], i.e. users contribute different perspectives to the platform

which in turn combines these perspectives to provide a better solution. One example for such a platform is *InklingMarkes* [30] providing predictions based on user bets.

- *Content providers*: Users contributing self-generated content are called content providers. These users may be found on *IStockPhoto* [34] or on social media sites like *YouTube* [75] or *Flickr* [22].
- *Component providers*: On some platforms, such as social networks for example, a user may additionally take the role of a component. Platform providers can then utilize these components and, for instance, show them ads they get paid for.

It has to be noted that within a single crowdsourcing system one user may play multiple roles. As mentioned above, knowing these roles is important as they may influence the recruiting strategy. Take *InklingMarkets* for example where the users in the crowd take the role of perspective providers. For such platforms, it is important to recruit diverse users, each able to make independent decisions in order to avoid "group thinking" [16, 58]. For the recruiting process, Doan et al. list five major solutions. The first solution presumes that one has the authority to *require users* to make contributions (e.g. a company requires employees to contribute to an internal system). Another solution is to *pay users*, as it is done on the AMT. Thirdly, a platform can build on *volunteers*, which is the most popular approach as it is free and easy to execute. The drawback of this solution is the unpredictability of the number of users that can be recruited for a given platform. The forth solution is demonstrated by the *reCAPTCHA project* [66] which uses the crowd to digitize text that could not be recognized by an OCR program. reCAPTCHA is used to protect websites from malicious bots. In order to leave a comment on a given website, for example, one has to retype two words provided by the reCAPTCHA form. The system already knows the correct answer of one word; the second word has to be transcribed by the crowd. Therefore the solution to recruit users is to let them *pay for a service*, i.e. their payment for leaving a comment on a website is a transcription of a word. The last solution listed in [16] is to *piggyback on the user-traces* of well-established platforms. One example would be using the user-traces of the *Google* [27] search engine to build a spelling correction system.

Once users are recruited it is the objective of each platform to retain them and encourage them to steadily contribute to the platform. According to [16], many *encouragement and retention (E&R) schemes* exist, but the most popular are either built on

- *instant gratification*, providing immediate feedback to the user showing him that his contribution makes the difference, or
- providing an *enjoyable experience* like playing a game while contributing (e.g. ESP game [65]), or
- establishing and showing *reputation, fame or trust* to other users on the system, or
- providing *competition mechanisms* and show, for example, the users with the most contributions to the community, or
- providing *ownership situations* and give users the feeling that they own parts of the system.

3.3.2 Applying Game Theory

Although a game-theoretic discussion of crowdsourcing is not part of this thesis, it should be mentioned that there are some approaches dealing with this issue. DiPalatino et al. [15] for example analyze contest-based crowdsourcing platforms using a game theoretic model. They model contests as all-pay auctions with incomplete information and analyze the relationship between incentives and user participation levels.

Archak and Sundararajan [5] analyze the optimal design of crowdsourcing contests. They also use an all-pay auction model with incomplete information to describe the crowdsourcing contest and provide a rule of thumb to determine the optimal price structure for a given contest.

3.4 Social Computing

Current task-based crowdsourcing platforms mostly support micro-tasks which can be processed by a single user in little time. However, many “real world” tasks are more comprehensive, as the consist of sub tasks and dependencies between these subtasks which can be described by workflows. These large-scale tasks can be processed by multiple users which may be organized as teams. Current crowdsourcing platforms have only limited support for automated processing of large-scale tasks and most of them do not support automated and organized user collaboration.

Schall et al. [55, 56] introduce *human provided services* (HPS) which describe a service oriented architecture (SOA) based approach for human interactions in business processes. In their framework they follow a typical SOA approach which can be described by the three steps: (i) *publish*, (ii) *search*, (iii) *interact*. At first a service provider publishes its service to a registry. In the HPS framework this may be either a software-based service (SBS) or a service provided by an individual (HPS). Then, a requester performs a search to find a desired service which may again be either a HPS or a SBS. Lastly, the requester and the service provider interact with each other.

Further work focusing on automated socio-technical interaction models utilizing SOA approaches is provided by Skopik et al. [57], who discuss *crowdcomputing*, a social network based crowdsourcing approach. Dustdar and Bhattacharya [17] introduce a concept called *Social Compute Unit* (SCU) which also focuses on unifying human- and software-based computing.

3.5 Agent-based Modeling Frameworks

This section provides a selection of two prominent agent-based modeling frameworks which are introduced in the following.

3.5.1 NetLogo

NetLogo [47] is an agent-based modeling toolkit designed for both education and research purposes. The framework itself is written in Java, the language to define simulations, however, is based on the functional programming language *Logo*. The advantage of this scripting approach is that Logo is easy to learn for novices with little programming knowledge, but still meets the needs of experienced, high powered users. NetLogo is freeware such that everybody can download it for free and build models without any restriction. It comes with a vast number of pre-defined models pooled in a model library, an extensive documentation and many tutorials [62].

In NetLogo, three different types of agents can be distinguished:

- *Turtles*: Mobile agents are called turtles as they move around in a world interacting with the world or other turtles.
- *Patches*: The world in which turtles move around consists of a grid of patches. Patches are also programmable and may possess specified behavior.
- *Observer*: In each simulation only one observer exists. The observer is basically responsible for running the simulation and providing reports.

One goal of NetLogo is to generate scientifically reproducible results. In order to so, the simulation models have to operate deterministically, i.e. a simulation, run multiple times with the same "seed", has to provide identical results. NetLogo supports *parameter sweeping* to systematically test the behavior of a system across a range of parameter settings. One further advantage is that various tools to visualize the results are provided by the framework [62].

3.5.2 MASON

The *Multi-Agent Simulator Of Neighborhoods* (MASON) [43] is a discrete-event simulation framework developed for large-scale multi-agent simulations. The design philosophy of Mason is to provide a small, fast and easy understandable and modifiable core which is written in Java. In addition to this core, a separate extensible visualization mechanism, which is capable of 2D and 3D visualizations, is provided. The framework can be divided into three layers. The *utility layer* containing "helper" classes like for example a random number generator, the *model layer* containing classes for event scheduling and agent representation, and the *visualization layer* containing the aforementioned visualization functionality. The model layer and visualization layer are fully decoupled so that the model can be treated as a self-contained entity. To create a customized model, the classes of the model layer may be extended by the use of inheritance. Mason provides the possibility to "checkpointing" a simulation at any given point (by saving its state to disk) and resume it on any other platform. As in NetLogo, scientifically reproducible results can be created by defining a "seed" [41].

CHAPTER 4

Methodology

This chapter details the architecture, the models and the approaches used to implement the simulation framework. After an introductory example scenario and a definition of requirements, the basic architecture is discussed. Section 4.4 then provides information on the design of the simulation model. Detailed information on the behavior of agents in the population as well as underlying models is provided in Section 4.5. The design of the marketplace is presented in Section 4.6 and finally, Section 4.7 discusses simulation specific components such as the reporting mechanism. This chapter mainly focuses on conceptual ideas, whereas detailed information concerning the prototypical implementation is provided in Chapter 5.

4.1 Task-based Crowdsourcing – A Motivating Example

As a motivating example for our framework, we consider an imaginary crowdsourcing platform (ICP) specialized in software projects. Two different types of clients can be identified: employers providing tasks for the system, hereinafter referred to as *requesters*, and employees processing the provided tasks, hereinafter referred to as *workers*. Since the quality of code is important for software development, the marketplace seeks to guarantee a predefined level of quality to requesters. Hence, the marketplace

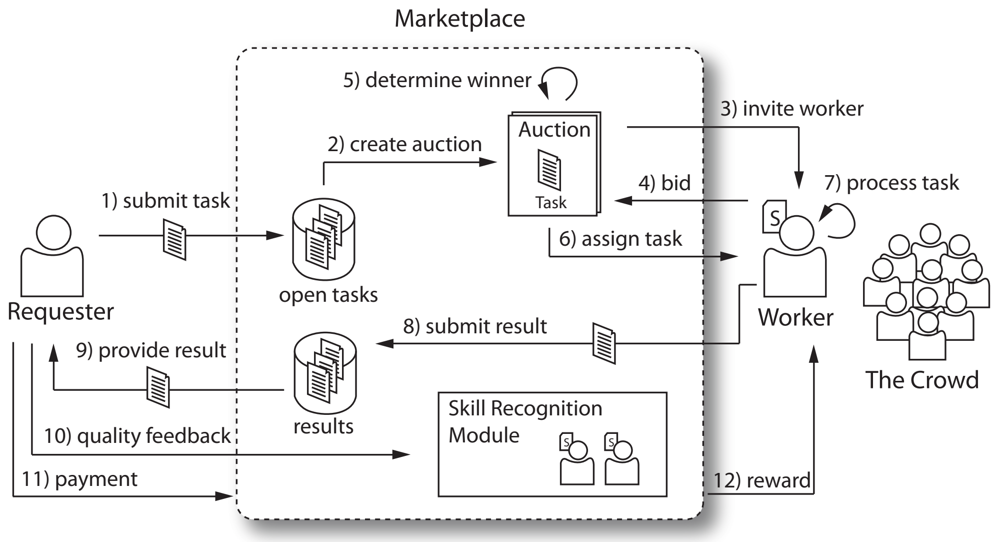


Figure 4.1: The task-based crowdsourcing process

has implemented methods to estimate the skills of each registered worker. Aware of the skills of all workers, the marketplace allows only qualified workers to compete for tasks. The marketplace provides an auction mechanism for task distribution and price negotiation, namely the reversed English auction. This kind of auction type has a given starting price. Bidders can try to win the auction by undercutting the current lowest bid. After a certain period of time the auction closes and determines a winner which is the bidder submitting the lowest bid. Figure 4.1 shows a typical process of crowdsourcing for the platform described above. At first, a requester submits a new software project (referred to as *task*), to the platform. The requester specifies the skills needed (e.g. Java programming, database engineering, etc.), the importance of each skill, the amount of money he is willing to pay and the deadline. The marketplace then creates an auction for this task, inviting only workers to the auction matching the skills specified by the requester. Since this is an introductory example we assume that the marketplace at this stage is to some degree aware of the skills of its members and we will not go into detail regarding skill recognition techniques (see Chapter 3). Workers interested in processing the task can place their bids in the auction. After a predefined amount of time, the auction will close, determine a winner and inform the worker about winning the auction.

It is now the job of the worker to process the task in time and to submit the result to the platform. Assuming the task has been processed in time, the marketplace informs the requester, who in turn provides a rating for the worker based on the quality of the processed task. In case the quality conforms to the predefined agreement, the requester transfers a defined amount of money, based on the outcome of the auction and on a possible fee for using the crowdsourcing platform, to the marketplace. Considering the requester's feedback, the marketplace updates the skill profile of the worker and finally pays the worker for his effort.

Suppose that the ICP operator now wants to test a new form of task distribution, where for example the outcome of the auction is no longer solely dependent on the price but on a combination of observed performance, confidence in the worker and the price. Since this new approach is still in a development phase and has not been tested yet, the operator is not willing to take any risks by implementing it to the "life system". Running tests on a "test system" is mostly not an option as one is dependent on the crowd, and the crowd might not want to work on test systems. Therefore, one way for the operator is to run a simulation of the platform. By creating a model of the crowdsourcing platform and defining the population which uses the platform, the operator is able to run various simulations to analyze the suitability of new approaches.

4.2 Requirements

Creating a simulation framework for task-based crowdsourcing implies dealing with certain requirements of such a system. In this thesis, two groups of requirements are distinguished: *basic requirements* primarily encompassing non-functional requirements which effect the architecture of the framework and *specific requirements* comprising functional and non-functional requirements [53] that define the features needed for a task-based crowdsourcing framework. In the following, an overview of the main requirements will be provided.

Basic requirements include *configurability* and *extendibility*. In order for a framework to be versatile, it needs to be configurable. Since configurability is a comprehensive term, we cut it down to the following requirement: simulating various different crowdsourcing *scenarios* should be possible. A scenario is a particular situation on a particular crowdsourcing system. For example, one scenario would be testing a new way of task distribution (e.g. a sealed-bid auction) in a task-based crowdsourcing environment using different populations (e.g. high-skilled vs. low-skilled workers). As one can see, a scenario contains all the information needed for a simulation; different scenarios may however contain completely different information. Therefore, we have to find a way to create and configure different scenarios to be run on the simulation framework.

Extendibility is the second basic requirement. Crowdsourcing platforms and their used techniques regarding quality management or task distribution evolve quickly. New methods arise and patterns change. One trend, for example, is that crowdsourced tasks become more complex and workers join teams to process these tasks [40, 57]. To keep up with this rapid development, the design of the framework has to be open and extendable. By the use of clear defined interfaces, it should be possible to implement new paradigms to the framework or increase the predictive power in the time to come.

Based on the example provided in Section 4.1, specific requirements can be listed as follows:

- *Model-based:* To be able to simulate various scenarios, it should be possible to create a model of any common task-based crowdsourcing system and run simulations on it.

- *Population:* It should be possible to define a *population of agents*. Each agent should possess a set of skills (drawn randomly from a given distribution) and a specific behavior defined by rules and strategies. Agents should be able to interact with the marketplace at any given time. Two types of agents can be distinguished: workers and requesters. Requesters submit tasks to the marketplace and workers select tasks published on the marketplace.

- *Skill recognition:* The issue of varying quality in task-based crowdsourcing and solutions to it has been discussed in Chapter 3. Since common platforms support quality management approaches, it should be possible to define strategies for skill recognition and quality management for the marketplace in the simulation model.

- *Auction mechanisms:* A new way for task distribution is to use auctions. It should therefore be possible to define basic auction mechanisms for task distribution in the model.

- *Reporting system:* To be able to analyze data gained in the simulation, a reporting system is needed. It should be possible to use reports based on the situation one wants to analyze by simply defining them in the model.

4.3 Architecture

The core functionality of this framework is simulating autonomous agents interacting with the marketplace. Each agent is driven by its own behavior, described by simple rules, policies or strategies, which defines the agent's actions at any given time. A common approach to model such systems is *agent-based modeling* (ABM). As stated in [42], there is no precise definition on what an agent has to fulfill to be considered autonomous. Some authors argue that simple reactive behavior realized by "if-then" rules is sufficient, others insist that agents have to implement adaptive artificial intelligence techniques in order to be considered autonomous [10, 12]. Henceforth, the former definition of autonomous agents will be used in this thesis, as agents will implement only simple rules for decision making. According to [42], an agent-based model consist of three components:

1. A set of *agents* each containing defined behavior

2. A set of *relationships* defining the interactions among agents

3. An *environment* the agents interact with

In contrast to this, the approach taken in this thesis differs therein, that agents have no relations with other agents but are allowed to interact with the environment exclusively, i.e. the crowdsourcing platform.

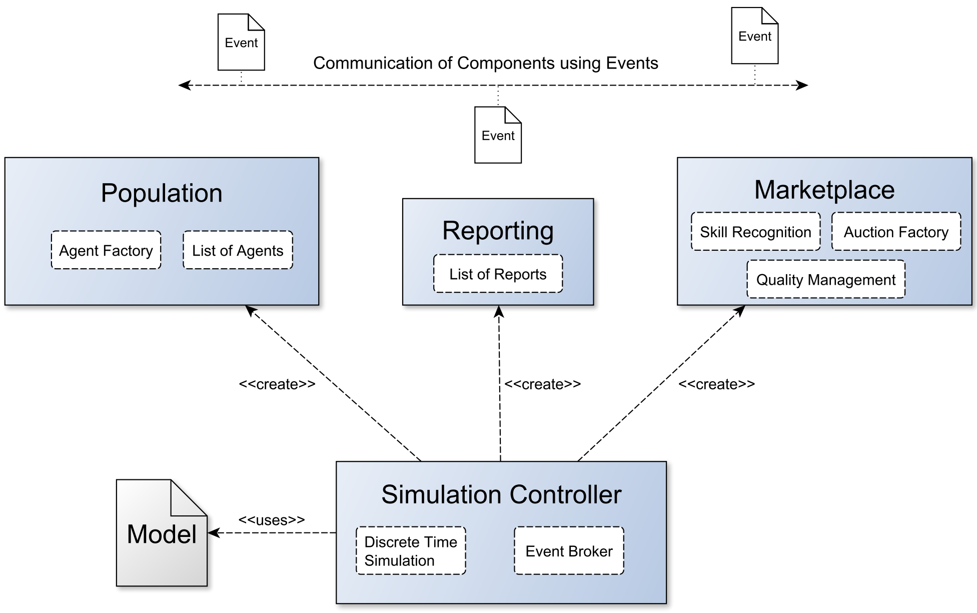


Figure 4.2: Basic architecture

Figure 4.2 gives an abstract overview of the architecture which is based on the ABM approach described above. The `Simulation Controller`, representing the main component of this figure, receives a model as input. On the basis of this model, the simulation is created and run by the controller. Besides that, it is also responsible for generating simulation time and broker events between the model's components for interaction. The `Population` component contains all the agents participating in the auction as well as methods for generation and initialization of agents. As the population is dependent on the simulation type, an exact specification of this component has to be provided in the model. As stated in Section 4.2, a flexible reporting mechanism is another requirement for the framework. The `Reporting` component is responsible to address this matter. Existing reports may be reused in several simulations by simply defining them in the simulation model. Finally, the `Marketplace` component contains the modeled behavior of a task-based crowdsourcing platform. Specific behavior such as quality management or task distribution has to be defined in the simulation model. A detailed description of each component and its sub components is provided in the reminder of this chapter.

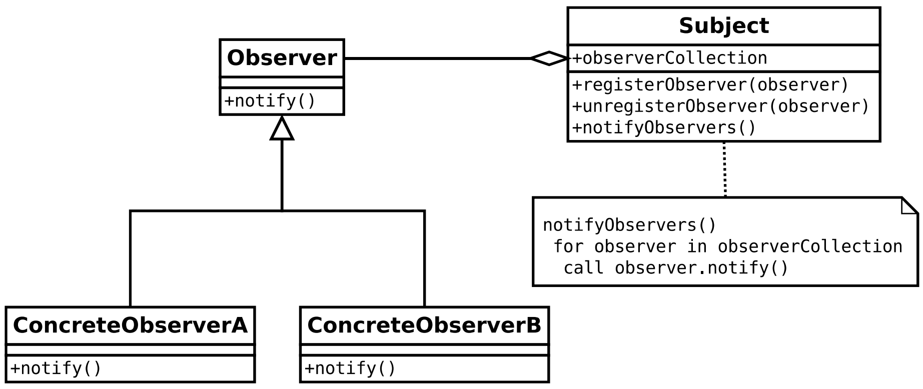


Figure 4.3: Observer design pattern in UML syntax, taken from [72]

4.3.1 Simulating Time using Discrete-Event Simulation

As mentioned above, agents may interact with the system at any given period of time. Therefore, *time* has to be defined in the simulation. According to [26], two simulation models for time–based simulations can be distinguished:

1. Continuous Models
2. Discrete-Event Models

Whereas in the first model, the state continuously changes within time, it only changes at discrete points in time in the second (e.g. if a specific event occurs). We use the second approach for our framework as the event based character perfectly suites the design of our framework which uses events for communication between components as described in the following section.

Components of the simulation, as for instance agents or marketplace, have to be aware of the current discrete time period we call *round*. Therefore, they somehow have to be informed of the current round. A common approach to solve this issue is to use a design pattern called `Observer` or `Event Listener` [25]. Figure 4.3 illustrates this pattern in *Unified Modeling Language* (UML) [48] notation. Applying this pattern to our framework results in the following: every component defined in the model has to register itself with the `Simulation Controller`. Whenever a new round starts, the registered components are notified and can thus react to this new state.

4.3.2 Event-based Communication

Configurability is a very basic requirement for a simulation framework. Returning to the motivating example of the imaginary crowdsourcing platform provided in Section 4.1, we assume that the ICP provider has created a simulation model of its platform and wants to set up an experiment in which different populations are compared to each other. In order to do this, one has to modify the model or create a new one for each simulation. After having analyzed the results of the simulation, the ICP provider comes to the conclusion that the current task distribution mechanism is not suited for the population currently using the platform. Therefore, he creates a new series of experiments using a different task distribution mechanism. Once again, the provider either has to change the current model or create a new one for each simulation. In object-oriented ABM frameworks, such as *Repast* [59], models and agents are implemented as objects and thus written in source code. Every time the model has to be altered, changes in the code have to be made. This involves the risk of introducing bugs into the code every time a new simulation is created and likewise impairing the configurability of the framework.

By means of the *Inversion of Control* (IoC) principle, which is naturally implemented through the *Dependency Injection* (DI) pattern, a separation of source code and model can be achieved [23]. The basic idea of IoC and DI is that components are only aware of dependency interfaces at compile time. The concrete implementation is *"injected"* to the interface at runtime by a component often called *container*. This allows high flexibility, as the *wiring* of components can be defined in an external file for example. In order to fully benefit from the DI pattern, a loose coupling between components has to be ensured. This can be achieved by the use of event-based communication, which is implemented through the Observer pattern described in the previous section.

Applying these approaches affects the design of our framework as follows:

- The *components* needed for a simulation, such as agents, marketplace, reports, auction mechanisms and various strategies representing behavior of agents or the marketplace are implemented as objects.
- The *simulation model* is represented by an external file containing a declarative definition of components, their properties and dependencies needed for a specific simulation.
- The *communication* between components is based on events. Each component can register for events of interest and gets informed whenever one of these events occurs.

To summarize the advantages of this approach, one can state that decoupling the configuration of the model from its implementation in combination with event-based communication allows high configurability. Since components are decoupled and dependencies are defined in an external file the underlying objects can be *"rewired"* easily, allowing us to create new simulation scenarios by reusing already existing components. Furthermore, functionality such as reporting can be clearly separated from the agent model itself, as reports simply have to listen to events to calculate statistics [52].

4.4 Defining the Simulation Model

In the previous section, we have discussed the framework architecture and stated that for each simulation a model has to be created. This section will provide detailed information regarding the design of the simulation model.

As mentioned earlier, a typical agent-based model consists of a set of agents, their properties and behavior, a set of relations defining interactions between agents and the environment agents interact with. Our model, however, is restricted to a set of agents and an environment they interact with, but additionally provides possibilities to include reporting functionality directly into the model. Basically, the model consists of two parts.

Firstly, the components and sub components provided by the framework, which define various kinds of behavior. Since the main components are described by interfaces the set of components can easily be extended. Secondly, the model file, containing a declarative definition of all components and sub components. This file *"wires"* the components and sub components together and therefore defines the explicit behavior of the components in the simulation. For example, returning to the imaginary crowdsourcing platform ICP, the framework provides the component `Marketplace` and the sub components `EnglishAuction` and `SealedBidAuction` which describe the behavior of two different task distribution mechanisms. To simulate a different behavior of the marketplace, i.e. a different task distribution mechanism, the provider simply has to change the configuration ("wiring") in the external file without writing or modifying any source code. Technically the simulation model can be divided into three parts:

1. The *Population* defining the agents in the simulation. Various groups of agents with different behavioral rules and properties can be defined in this part.

2. The *Marketplace* defining the crowdsourcing system one wants to simulate. Task distribution mechanisms such as various auction mechanisms, quality management methods such as qualification policies and skill recognition strategies can be defined in this section of the model.

3. The *Simulation* part defining simulation specific properties such as reporting mechanisms, mixing strategies to simulate the arrival of agents on the market and algorithms to generate probability distributions.

4.5 Population

In our model, two different kinds of agents in the population are distinguished. Requesters submitting tasks to the market and workers providing workforce to solve these tasks offered on the market. As we follow an agent-based approach in our framework, these agents are considered autonomous and they therefore implement their own properties and behavior which is described by `Strategies` and `Policies`. By interchanging the various available strategies and policies in the model, a diverse population can be created.

Both types of agents are trading in a market. Requesters act as buyers as they buy workforce from the market and workers act as sellers since they provide and sell workforce on the market. Either way, both agent types have to know the value of the task they are trading in (i.e. how much they are willing to pay or how much they want to be paid). In literature, various ways to model this valuation behavior are known (see Section 2.1.1). We chose to implement the *private-value* model, where each agent knows its own value for a specific task, but this value is private information to the agents themselves [38]. Again, this behavior is implemented by policies, so called `ValuationPolicies`.

Agents are autonomous and as in the real world, one of their goals is to gain profit. In order to do this, they have to place bids in the market. If an agent places a bid in the market or not depends on various factors such as private value, current market value, interest in a task, qualification for a task or workload. We model that behavior using so called `TradingStrategies`. One further property that both agent types have in common is an `Account` in order to pay or get paid for finished tasks. At this point, one has to mention that many of the assumptions in the following sections regarding the population and their behavior are based on the work provided in [37] and [54].

4.5.1 Tasks and Transactions

As in every marketplace, traders have to know the objects they are trading, which in our context are `Tasks`. A task consists of a `duration` representing the period of time a perfectly suited worker will take to process a given task. The duration is drawn randomly from the set $\{1, 2, \ldots, n\}$ where by default n is set to 24. Furthermore, a task has a predefined `expectedResult`, representing the result of a perfectly executed task achieving maximum quality. This result is randomly drawn from a uniform distribution $\mathcal{U}(0, 1)$ in the range $[0, 1]$.

As mentioned in the motivating example in Section 4.1, requesters may define the skills needed to process a task, the importance of each skill and a minimum quality level. To stick to the motivating example, we assume that one wants to submit a task which requires writing a program in Java that stores its data to a database by obtaining at least medium quality. Writing the Java code will constitute about ninety percent of the effort, designing the database about ten. To model this behavior, two sets are introduced as follows:

$$skillQuality = \begin{pmatrix} s_{q1} \\ \vdots \\ s_{qn} \end{pmatrix} \qquad skillWeight = \begin{pmatrix} s_{w1} \\ \vdots \\ s_{wn} \end{pmatrix} \tag{4.1}$$

Each element s_i in those sets represents a skill. In the $skillQuality$ set the value of each element s_{qi} stands for the required minimum quality of a skill s_i and in the $skillWeight$ set s_{wi} stands for the importance of a skill s_i. For the latter it holds that

$$\sum_{i=1}^{|skillWeight|} s_{wi} = 1 \tag{4.2}$$

whereby the partitions are generated randomly. In case $s_{wi} \in skillWeight = 0$, skill s_i is not needed to process the task. The number of needed skills is drawn randomly from a uniform distribution $\mathcal{U}(0, n)$ where n is the number of skills supported by the marketplace, which is by default set to 5. The value of the minimum quality of each skill is drawn from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$ where only values in a predefined interval $[lowerBound, upperBound]$ are allowed. The default values are: $\mu = 0.5, \sigma = 0.25$, $lowerBound = 0.1$, $upperBound = 0.9$.

As soon as a requester submits a task to the platform, a `Transaction` is generated containing the task and additional information provided by the requester. This information involves the price the requester is willing to pay and the deadline. For the deadline it is assured that it will be after the task's expected duration.

4.5.2 Requester Model

In our model, requesters are responsible for generating tasks and offering them to the marketplace. Every round in the simulation, each requester registered to the platform is asked to submit tasks. This behavior is modeled using so called *Task Supply Policies*. The default `SimpleTaskSupplyPolicy` provided by the framework forces the requester to submit one task every round. Once a task is finished requesters have to validate the result and provide a rating based on the observed quality. In case the task is not finished in time or the quality is below a defined threshold (by default 0.3), the requester will stop supplying tasks to the market for a certain amount of time (specified in the simulation model) and rate the quality of the received result with the value zero, due to the negative experience. The rating behavior is once again implemented by `RatingPolicies`. The default behavior provided by the framework is described by the `SimpleRatingPolicy` where the rating is calculated as described in Equation 4.3.

$$rating = 1 - |expectedResult - workerResult| \tag{4.3}$$

$ExpectedResult$ and $workerResult$ are both within the range $[0, 1]$. The former represents the best result possible; the latter the result processed by the worker. Therefore, the $rating$ represents the quality achieved by the worker in percent, since the more the results are equal the more the rating converges to one and the more the results differ the more the rating converges to zero.

Component	Description
`TaskSupplyPolicy`	Specifies the task supply behavior and defines the number of tasks submitted by a requester each round and the rounds to suspend in case the requester is not satisfied with the quality.
`RatingPolicy`	Specifies the requester's rating behavior.
`ValuationPolicy`	Calculates the requester's *private value* depending on various factors.
`TradingStrategy`	Determines if a task should be submitted and provides a price.
`Account`	Represents an account for monetary transactions with the marketplace.
`initialFunds`	Represents the initial amount of money available to requester at the beginning of the simulation.

Table 4.1: Components of the requester model

When submitting a task to the system, a requester has to specify a deadline and a price. By default, the deadline is set to a random value within the range $[1.5t, 2t]$ where t represents the expected duration of the task as described in the previous section. The price is within $[0, 1]$ and depends on the `ValuationPolicy` and the `TradingStrategy` defined for the requester in the simulation model. Table 4.1 summarizes the components used in the requester model.

4.5.3 Worker Model

Workers provide the workforce on the marketplace. By bidding for tasks they are willing to process and by processing tasks they won in an auction they make profit. Like in the real world, each worker has a set of skills, denoted by the set

$$skill = \begin{pmatrix} s_{p1} \\ \vdots \\ s_{pn} \end{pmatrix} \tag{4.4}$$

Each element $s_{pi} \in skill$ represents the performance p of a skill s_i where s_{pi} is within a range $[0, 1]$. In case s_{pi} is zero, the worker does not possess the skill s_i. The number of skills as well as the type of skills s_i a worker possesses are uniformly distributed in $\mathcal{U}(0, n)$, where n is the number of skills supported by the marketplace. The values of s_{pi} are drawn randomly from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$ where μ and σ have to be defined in the simulation model.

When invited to join an auction, workers first evaluate if they are interested in processing the task. This behavior is implemented by using trading strategies which is discussed separately in Section 4.5.5. At this point, one can state that trading strategies determine whether a bid is placed or not. In case it is the trading strategy sets the price level. Both actions are depending on various factors such as workload of the worker, current market price of the task and the worker's private value.

In order to support multiple task processing strategies and to determine the completion time, the worker model contains the `TransactionBook` component. By means of this component, one can specify the worker's task processing ability, e.g. the ability to process tasks in a row or in parallel or the ability to process time-sensitive tasks first and postpone others. The default `SimpleTransactionBook` which is provided by the framework provides serial task processing only. The time needed to finish a task depends on the expected duration t (see Section 4.5.1) and the worker's performance. The better the worker's skills are, the more the processing time converges to t, as we argue that a higher performance enables one to be faster in processing a given task. With Equation 4.1 and Equation 4.4 the `SimpleTransactionBook` calculates the processing time as follows:

$$processingTime = t + (1 - \sum_{i=1}^{|skillWeight|} s_{pi} * s_{wi}) * t \tag{4.5}$$

As one can see, the processing time is within $[t, 2t]$ and only those performances needed to process a task are taken into account for calculation. The processing time is encapsulated in this component so that the workers are not exactly aware of the time they will need to process a task in advance. Another responsibility of the `TransactionBook` is to calculate the worker's result of the task, and thus determine the quality a worker provides. As illustrated with the processing time, the result also depends on the worker's performance. Equation 4.6 depicts the calculation of the worker's result used in the `SimpleTransactionBook` applying Equation 4.1 and Equation 4.4.

$$workerResult = \sum_{i=1}^{|skillWeight|} x_i * s_{wi} \quad \text{with } x_i \in X \sim \mathcal{N}(expResult, 1 - s_{pi}) \quad (4.6)$$

For each skill needed to process the task, we draw a value of a Gaussian distribution with the expected result as mean and the worker's reverse performance for this skill as variation and multiply it with the skills weight. The sum of these values equals the worker's result for the whole task. Therefore, the better a worker's performance the more likely the result will be close to the expected result; the more important a skill is for processing a task, the more it impacts the result. Table 4.2 summarizes the components used in the worker model.

Component	**Description**
`Skills`	Represents the skills a worker possesses
`TransactionBook`	Manages the workers task processing. Calculates processing time and task result.
`ValuationPolicy`	Calculates the workers *private value* depending on various factors.
`TradingStrategy`	Determines if a bid should be placed and provides a price depending on various factors.
`Account`	Represents an account for monetary transactions with the marketplace.

Table 4.2: Components of the worker model

4.5.4 Models for Task Valuation

As mentioned earlier, the agent's behavior regarding task valuation is based upon a *private-value* model. By evaluating a task, based on several factors, an agent determines a price reflecting the personal value, i.e. the amount of money the worker wants to receive at least or the requester is willing to pay. This behavior is implemented using `ValuationPolicies` and the factors influencing the `privateValue` depend on the policy used. The private value ranges within an interval $[0, 1]$. By default, the framework implements four different valuation policies; two for requesters and two for workers. This is necessary since requesters, for example, will only consider factors such as effort and required quality, but workers might as well consider their suitability for processing a given task. Howsoever, only worker valuation policies will be explained in the following, since they are equal to requester policies, except for considering an additional factor, namely the suitability for processing a task.

Basically, two different types of valuation polices are provided by the framework:

- Policies determining the private value based on the expected duration, the required skill quality and the workers suitability for a task.
- Policies determining the private value based on expected duration, skill quality, suitability and additionally a market price representing the current trading price of similar tasks in the market.

The rationale is that the higher the effort and the required minimum quality, the higher is the worker's private value. Furthermore, workers are aware of their capabilities and want to make profit by processing as many tasks as possible. Being high skilled means that one is fast in processing a task and can thus process more tasks in given time than a low skilled worker. Therefore, high skilled workers calculate lower private value levels than low skilled workers. Using Equation 4.4 and Equation 4.1 we define:

$$suitability(w) = \sum_{i=1}^{|skillQuality(w)|} \frac{s_{qi}}{s_{pi}} * \frac{1}{|skillQuality(w)|} \tag{4.7}$$

and calculate the private value as follows:

$$privateValue(t, w) = effort(t) * suitability(w) \tag{4.8}$$

The $effort$ is a random value drawn from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$ based on the required quality of a task and the normalized expected duration, where $\mu = Max(reqQuality, expDuration)$ and $\sigma = Min(reqQuality, expDuration)$. We can draw the following conclusion: the higher a worker's qualification, the lower the suitability factor and the lower the suitability the lower the private value.

The market price based valuation policy calculates the private value similar to Equation 4.8, with the difference that the current market price is taken into account.

4.5.5 Trading Strategies

In the previous section, we discussed a way to model the agent's private value which determines the agent's valuation of a task. However, by only bidding the private value one cannot make profit. `Trading Strategies` are implemented to determine whether a bid should be placed in the market and which price the bid should contain. Requesters, in contrast to workers, will always submit a task since they can choose the price freely without restrictions regarding deadlines or load rate. For workers to determine whether a bid should be placed in an auction the trading strategy may take several factors into account. Each factor has to be set in the simulation model:

- *Deadline:* Workers may only bid for a task if they have a realistic chance of processing it in time. Some workers might show cautious behavior and place bids only if a certain amount of time is left. Others might show a risky behavior and place bids despite the fact that there is only little time left. We model this as follows: for a worker to place a bid there must be at least $x * t$ time left, where x is typically within the range $[1, 2]$ and t is the expected duration of a task.

- *Workload:* Workers might place bids based on their load rate. Again, there might be greedy workers trying to get as many tasks as possible without considering the current workload, and modest workers placing bids only if their workload is not too high.

- *Current Price:* Several auctions allow workers to see the current price. For such auction types, workers will only place a bid in case the current price is higher than their private value.

- *Open Bids:* It might take a few rounds for auctions to clear and determine a winner. Workers can place bids in many auctions without knowing if they will finally win the specific auction. Bids placed in auctions that are still in progress are referred to as *open bids*. The number of allowed open bids determines if a worker acts aggressively or conservatively. A high number of allowed open bids may let the worker win several auctions at a time which could result in deadline violations due to high workload. In contrast, a low number of open bids means that the worker behaves more conservatively, winning only few auctions.

The level of the bid price depends on the trading strategy used. By default, there are five different trading strategies provided by the framework:

- The `TruthtellingStrategy` sets the bid price according to the agent's private value. Using this strategy in an auction based platform might be appropriate for requesters. Since for them the private value represents the price they at most want to pay in order to get a task processed, they may use this value as starting point for an auction as it can be expected that workers will push the price downward. For workers in contrast, using this strategy will not be appropriate, as they will not be able to gain profit using the private value as bid price.

- The `FixedDirectionStrategy` bids a specified markup x on the agent's current valuation. The bid price is modeled as follows:

$$bidPrice = \begin{cases} privateValue + x, & \text{if worker} \\ privateValue - x, & \text{if requester} \end{cases} \tag{4.9}$$

  The markup x is drawn from a uniform distribution $\mathcal{U}(0, maxMarkup)$, where $maxMarkup$ has to be specified in the simulation model. By altering the maximum markup, the greed of agents can be influenced.

- The `BeatTheQuoteStrategy` determines the bid price based on the auction's current price, the so called *quote*. Most auction mechanisms provide a quote for bidders. A very common behavior of bidders is to bid slightly over the quote trying to maximize their profit. This behavior can be observed on eBay [18] for example. In contrast to auctions on eBay, we provide reverse auctions only which means that workers using this strategy will bid slightly under the current quote and requesters slightly over the quote. In case the auction does not provide a quote, agents will bid slightly over or under their own private value. The bid price is calculated as follows:

$$bidPrice = currentQuote * (1 \pm (x * maxMarkupRate)) \tag{4.10}$$

The variable x is randomly drawn from a uniform distribution $\mathcal{U}(0, 1)$. The $maxMarkupRate$ defines the maximum percentage by which the quote is altered and has to be set in the simulation model.

- The `LoadBasedStrategy` may be used for workers only, since the price is dependent on the load rate. The rational is that workers with a high current workload charge higher prices as they only want to work on further tasks in case they make high profit. The bid price will be determined as shown in Equation 4.11.

$$bidPrice = privateValue * (1 + (load * maxMarkupRate)) \tag{4.11}$$

  The $load$ variable is within $[0, 1]$ and represents the workload of a given worker. The $maxMarkupRate$ represents the maximum percentage by which the private value is altered and has to be set in the model.

- The `LoadBasedBeatTheQuoteStrategy` which is a combination of the two aforementioned strategies, determines the bid price as follows. At first two prices are calculated: a *loadBasedPrice* as shown in Equation 4.11 and a *qouteBasedPrice* as shown in Equation 4.10. The bid price is set to the latter but the bid will only be submitted if $loadBasedPrice \leq qouteBasedPrice$ holds. In case there is no quote available, the $loadBasedPrice$ will be set as bid price.

4.6 Marketplace

In our model, the marketplace component is representing a task-based crowdsourcing platform. Requesters submit tasks to this platform and it is the job of the marketplace to find suited workers and assign tasks to them. As mentioned earlier, keeping its members satisfied by providing high quality results to requesters and at the same time trying to assign tasks to most of the workers is a big challenge for the system. The model therefore has to provide methods for simulating different task distribution and quality management approaches. Once a task is assigned, processed and returned to the requester, a rating is provided by the requester. Based on this rating, an update on the skills and confidence levels of the involved worker is conducted by means of a *Skill Recognition Module*. Furthermore, the marketplace has to take care of the payment.

4.6.1 Task Distribution by Means of Auction Mechanisms

As the main focus of this thesis is to evaluate the use of auctions in a TBCS, task distribution mechanisms supported by the framework are all based on auctions. Since the design of the framework is very open, implementing other task distribution mechanisms is possible but out of scope in this thesis. All the auction mechanisms we consider are *reversed auctions*, as the roles of buyers and sellers are reversed. In contrast to commonly known *forward auctions* where buyers compete to obtain goods by increasing their bid prices, in a reverse auction sellers compete to obtain business from the buyer by decreasing their bid prices [36,71]. By buying workforce from the market, requesters take on the role of buyers and by selling workforce to the market workers take on the role of sellers.

Basically, two different auction mechanisms are distinguished in our model: *reverse standard auctions* where a single buyer, the requester, submits an *ask* (i.e. the task) while many workers compete by placing *bids* in order to get the task assigned, and *reverse double auctions* where workers and requesters are treated symmetrically with workers placing bids to sell their workforce and requesters placing asks to buy workforce.

Reverse Standard Auctions

For reverse standard auction types, the marketplace creates a new auction for every task submitted by a requester. By means of *selection strategies*, which will be discussed in

Section 4.6.2, those workers matching the task's quality requirements are selected and invited to participate in the auction. Each auction has an assigned `duration`, defining the number of rounds during which workers are allowed to place their bids until the auction closes and determines a winner. For many auction types, it is essential that agents are aware of the current auction price to be beat. Hence, each auction implements a `Quote`.

The most important component for an auction is the `OrderBook`, implementing behavior to cope with incoming bids and asks. In order to determine a winner the order book has to match bids and asks depending on the auction type. Usually, the matching is dependent on the price, i.e. in reverse auctions the lowest bid will win. The `LowestPriceOrderBook` implements this behavior. For some task-based crowdsourcing platforms it might not be sufficient that a winner is solely determined by the price, since this might provide low quality results. By altering the order book, additional factors such as observed worker performance and confidence may be taken into account, facilitating quality dependent task matching. This behavior is implemented by the `WeightedScoreOrderBook` which is based on an approach described in [54]. Utilizing the skill recognition mechanism provided by the marketplace, this order book calculates a score for each bid based on the worker's observed performance s_{oi} of a skill s_i, the confidence s_{ci} the marketplace has in the worker's skill s_i and the worker's bid price $price$. Applying Equation 4.1 and Equation 4.15 we define

$$wObPfmc = \sum_{i=1}^{|oPfmc|} (s_{pi} * s_{wi}) \tag{4.12}$$

and

$$wCnfd = \sum_{i=1}^{|cnfd|} (s_{ci} * s_{wi}) \tag{4.13}$$

the score is calculated as

$$score = x * wObPfmc + y * wCnfd + z * (1 - price). \tag{4.14}$$

Only those skills needed to process a task are taken into account for the calculation of weighted performance and weighted confidence, since $s_{wi} = 0$ in case a skill is not needed. Variables x, y, z defining the influence each factor has on the score have to be set in the simulation model, such that $x + y + z = 1$. The bidder obtaining the highest score represents the best match and therefore wins the auction. Submitting the lowest bid will in many cases result in not winning the auction, thus providing a solid quote to agents is not possible. This is why this order book should be used for sealed-bid auctions only.

By default the framework supports three different standard auction types [1]:

- `SealedBidSecondPriceAuction`: On auction creation the requester's ask a is submitted to the auction. Then, each worker may submit one bid b to the auction, whereas only those bids are accepted where $b \leq a$ holds. Since workers must not know the bids of other workers, this auction type does not provide a quote. The winner of the auction is the agent who submitted the bid with the lowest price. The price the requester has to pay, so called *clearing price*, amounts to the value of the second lowest bid.

- `DutchAuction`: For a Dutch auction, a start price q_0 and a increment rate r have to be defined in the model. On auction creation, the requester's ask a is submitted to the auction and the auctions current quote q is set to q_0. Each round, q is incremented by r until $q = a$. Workers may submit bids b, whereas only those bids are accepted where $b \leq q$ holds. The worker submitting the first accepted bid wins the auction.

- `EnglishAuction`: On auction creation the requester's ask a is submitted to the auction providing the auctions start price. Each round, workers may submit bids b whereas only those bids are accepted, where $b \leq a$ holds. The quote q amounts the price of the lowest accepted bid. After a predefined amount of rounds t, the auction closes and the worker submitting the bid with the lowest price wins the auction.

[1]A detailed discussion is provided in Chapter 2

Double Auctions

In double auctions, multiple buyers and sellers may place their *shouts* (i.e. bids and asks) simultaneously. For matching shouts, the auction determines a price and informs buyer and seller about the successful transaction. In order to conduct these kinds of auctions, a uniform object or good (e.g. one pound of coffee) traded in the auction market has to be defined.

Tasks created and traded in our framework are based on the assumption that they are all different. Each task has its own duration and quality requirement and is thus unique. As a consequence, it is not possible to trade multiple tasks in a double auction. In contrast to standard auctions where agents are trading tasks, in double auctions we are following the idea that agents are trading workforce as uniform good on the market in terms of *person hours*. To ensure quality, several auction markets, each trading a specific skill s_i at a specific *quality level* ($qLevel(s_i)$), are installed on the marketplace. In order to be allowed to trade in a specific market, workers have to match the given minimum quality requirements. We define that a worker is allowed to trade in the market if it holds that $s_{oi} \geq qLevel(s_i)$, where $s_{oi} \in oPfmc$ represents the observed performance of a given skill s_i provided by the marketplace. In our model, we install two markets for each skill: one for low quality requirements and one for high quality requirements. These levels can be set in the simulation model. The framework implements a *Continuous Double Auction*, in which the auction never closes and every time a shout arrives, the market *clears* the auction by trying to find matching shouts and informing the involved agents of a successful transaction. Only shouts beating the current quote are accepted. In double auctions two different quotes can be identified:

- *Bid Quote:* The bid quote represents the quote that requesters have to beat (i.e. offer a higher price) in order to buy workforce from the market. Its value is set to the price of the *lowest unmatched bid.*
- *Ask Quote:* The ask quote represents the quote workers have to beat (i.e. offer a lower price) to sell workforce to the market. Its value is set to the price of the *highest unmatched ask.*

Since in double auctions, requesters cannot submit a whole task, they have to calculate the effort in terms of person hours per skill and buy hours in the specific markets. The requesters trading behavior for such auction types is modeled very simple. We assume that the amount of hours needed to finish a task equals the `expectedDuration`. To calculate the amount of hours for each skill we multiply the expected duration with each skill's associated weight. For each hour the requester places an ask in a matching auction, whereas an auction matches if $qLevel(s_i) \geq s_{qi}$ holds. Whereby $qLevel(s_i)$ represents the minimum quality of a skill s_i workers trading in this auction obtain and $s_{qi} \in skill$ represents the minimum quality of a skill s_i demanded by the requester. A placed ask stays in the auction for a specified amount of time. In case a matching bid is found within this period, the ask is removed and a sub task is assigned to the worker, otherwise the requester is informed that no matching bid was submitted. A sub task represents a proportioned amount of work, offered by the requester. Each auction market contains a predefined deadline which defines the period of time a worker has left to finish a sub task. This deadline is the same for all tasks traded in the auction and starts once a sub task is assigned to a worker. Two shouts in an auction match in case the bid price is lower than the ask price. As there may be a lot of shouts in the market, the current highest ask is matched with the current lowest bid where the price the requester has to pay equals the price of the worker's bid. To determine the quality of the processed task we use the average quality of all sub tasks. It has to be mentioned that this is a very naive approach, as in the real world quality might also depend on the number of different workers processing a task and their ability to cooperate. Since implementing more sophisticated behavior, like defining cooperation levels between agents, would have gone beyond the scope of this thesis, we chose to implement this very model.

Table 4.3 summarizes the main properties of the auction component.

4.6.2 Quality Management and Skill Recognition

Providing high quality results to requesters is essential for most TBCS. This framework follows an approach described in [54] and [37], which is based on skill recognition and

Component	Description
`OrderBook`	Specifies the auction behavior. Matches shouts and determines the winner of an auction.
`Quote`	Provides the current auction price to agents. Might not be used.
`duration`	Defines a period of time in which the auction accepts bids.

Table 4.3: Components of the auction model

confidence management techniques. For each registered worker a profile is created, containing the worker's observed performance and the confidence in this performance.

$$oPfmc = \begin{pmatrix} s_{o1} \\ \vdots \\ s_{on} \end{pmatrix} \qquad cnfd = \begin{pmatrix} s_{c1} \\ \vdots \\ s_{cn} \end{pmatrix} \tag{4.15}$$

Each value $s_{oi} \in oPfmc$ describes the worker's observed performance of a skill s_i and is within an interval $[0, 1]$. $s_{oi} = 0$ means that a worker does not possess the skill s_i and $s_{oi} = 1$ means that a worker is perfect in using skill s_i. The same holds for each value $s_{ci} \in cnfd$ which describes the confidence the marketplace has in a worker's specific skill s_i. Every time a worker processes a task, the confidence and performance levels are updated, whereas the confidence level will always increase and the alteration of the performance level will depend on the rating provided by the requester. This behavior is implemented using `SkillUpdatePolicies`. By default, two different policies are implemented. The `UniformSkillUpdatePolicy` updating each of the involved skills uniformly, and the `WeightedSkillUpdatePolicy` updating the worker's skills based on the skill weight defined in the task definition. With Equation 4.3 and Equation 4.15 we define the performance update as

$$oPfmc(s_{oi}) = s_{oi} + pfmcUpdateRate * (rating - s_{oi}) \tag{4.16}$$

and the confidence update as

$$cnfd(s_{ci}) = s_{ci} + cnfdUpdateRate * (1 - s_{ci}) \tag{4.17}$$

where $pfmcUpdateRate$ and $cnfdUpdateRate$ have to be set in the simulation. By default these values are set to $pfmcUpdateRate = 0.2$ and $cnfdUpdateRate = 0.1$.

The initial values of $cnfd(s_{ci})$ are drawn from a Gaussian distribution $\mathcal{N}(\mu, \sigma)$ where μ and σ have to be set in the simulation model. The initial values of $oPfmc(s_{oi})$ are drawn from a Gaussian distribution $\mathcal{N}(s_{pi}, 1 - s_{ci})$ where s_{pi} refers to the worker's real performance defined in Equation 4.4.

Based on the values provided by the skill recognition component, only those workers matching the skill requirements for a specific task are invited to join an auction. To realize this matching behavior, `QualificationPolicies` are implemented by the framework. The framework has by default two different qualification policies implemented: one allowing all registered workers join an auction, the so called `AllQualificationPolicy`, and the `MinPerformanceQualificationPolicy`, allowing only those workers meeting the minimum requirements of a task to join the auction, i.e. $\forall s_{qi} \in skillQuality : s_{qi} \leq s_{oi}$

Table 4.4 summarizes the main properties of the marketplace component.

Component	**Description**
`AuctionType`	Specifies the auction type used for task distribution.
`WorkerProfiles`	Contains skill and confidence profiles of workers.
`SkillUpdatePolicy`	Specifies the requesters skill recognition behavior and performs updates on the workers skill profiles.
`QualificationPolicy`	Specifies the qualification requirements for workers to be invited to a specific auction. Is only supported in combination with skill recognition strategies.

Table 4.4: Components of the marketplace model

4.7 Simulation

This part of the model contains components needed to configure simulation specific behavior. Firstly, by means of the *reporting component*, one can define reports to calculate statistics of certain variables of interest. Following the design principle of our framework, reports themselves are implemented as objects, listen to events brokered by the `SimulationController` and have to be defined in the model file in case they should be used in the simulation. By default the simulation framework has implemented a variety of reports listed in Table 4.5.

Report	**Description**
Transaction Report	Lists various information for each transaction conducted in the simulation, such as quality requirements, price statistics, deadline and completion rounds, etc.
Requester Report	Lists for every requester trading in the simulation various information such as offered tasks, assigned tasks, completed tasks, deadline violations, revenue, payoff, etc.
Worker Report	Lists for every worker trading in the simulation various information such as skill levels, obtained ratings, assigned tasks, revenue, payoff, etc.
Payoff Report	Provides a statistical summary on payoffs of requesters and workers.
Task Report	Provides a statistical summary on the number of tasks submitted, assigned, unassigned, completed and the occurred deadline violations.
Quote Report	Lists the quotes of double auctions.

Table 4.5: A list of available reports

Secondly, the `AgentMixer` is responsible for simulating the agents arrival at the market. By default there is one mixing strategy implemented, namely the `RandomRobinAgentMixer` which lets all agents arrive in each round, shuffled in a random order. By extending or altering the agent mixer, one can simulate different arrival strategies. To simulate a certain fluctuation of activity for example, one could create randomly generated sub groups of agents arriving on the market in each round. Finally, implementation specific details such as the definition of a *random generator* to generate random distributions used in the model have to be defined in this part of the model.

CHAPTER 5

Implementation

The previous chapter introduced the architecture, the models and the approaches needed to build a simulation framework for task-based crowdsourcing. This chapter details the implementation aspects of our prototype implementation. At first, an overview on the JABM framework, on which our simulation framework is built upon, is given. Then, the key structure of the framework is discussed. Finally, the implementation of the simulation model is explained.

This simulation framework has been developed in the *Java programming language* (version 6). Designing the framework, we decided to build it upon an already existing agent-based modeling (ABM) framework as there are already various available. *NetLogo* [47], *MASON* [43] or *Repast* [59] are among the more prominent ABM frameworks written in Java, each with their own advantages and disadvantages. We, however, chose to use the *Java Agent-Based Modelling Toolkit* (JABM) [35] due to the following reasons. Firstly, it is an open-source framework and fully written in Java which allows us to use it in our framework. Secondly, its light-weight character enables us to easily extend it to our needs. Thirdly, it uses the Spring framework to cleanly separate the implemented components from the simulation model itself, which conforms to the design of our framework. Lastly, it is the underlying framework of the open-source *Java Auction Simulator API* (JASA) which implements useful components for auction processing that can be reused and extended in our framework.

5.1 The Underlying JABM Framework

The open-source Java Agent-Based Modeling toolkit (JABM) is a light-weight agent based modeling framework fully written in Java. It uses discrete-event simulation [8] to model temporal aspects, and by the use of the *dependency injection design pattern* a separation of the simulation model and its implementation is achieved [52]. JABM was originally used in combination with the JASA toolkit to run trading simulations using various auction mechanisms and is therefore appropriate for market simulations.

According to the basic design principle of JABM, the entities used in the agent-based model are represented as Plain Old Java Objects (POJOs)[1]. These POJOs are then configured by means of dependency injection using the *Spring framework* [60], whereby the configuration information is contained in a file. This file is, according to the Spring standard, written in *Extensible Markup Language* (XML) [20] , and represents the simulation model. Figure 5.1 shows the key components of the JABM framework.

The `SimulationController` is the main component of a JABM simulation, containing the `Simulation` and the `Population` object. Using the `EventScheduler` interface, communication between components is realized as all `Agents` in the population listen to this scheduler and react to events they receive. The `Population` provides functionality to create various agents at run-time, utilizing `ObjectFactories` provided by the Spring framework. To ensure that all agents are initialized using values drawn from the same random distribution, the JABM framework provides a `Random Generator`, which is declared using *singleton scope*. By means of the singleton design pattern [25], the random generator object exists only once in the simulation and this is why each component using the generator obtains a random value drawn from the same distribution. The `Simulation` object in combination with the `AgentMixer` object defines how and how often the agents in the population interact with each other. For example, by using a `RepeatedInteractionSimulation` in combination with a `RandomRobinAgentMixer`, all agents would interact with the simulation a given

[1]A Plain Old Java Object is basically an object containing a zero argument constructor and getter and setter methods for each attribute [51]

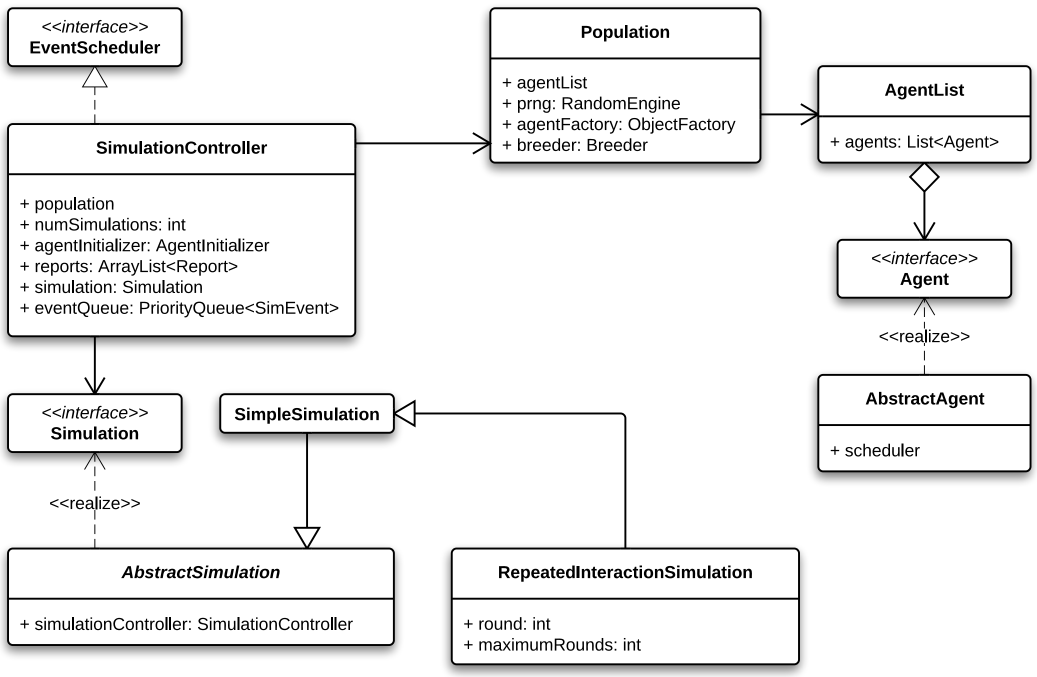


Figure 5.1: UML diagram showing the key components of JABM, taken from [52]

number of times. By declaring `Reports`, which are already implemented by the JABM framework, in the model, the modeler may obtain a variety of different statistics or charts. Furthermore, custom reports can be created by implementing objects which extend the `ReportVariables` interface.

5.2 Design of the Framework

The implementation of the simulation framework is based on the architecture provided in Chapter 4 and influenced by the design of the JABM framework. Further, a few classes and design principles of the JASA framework have been adapted and reused in the implementation of our framework. This comprises, for example, the reuse of `Shout` objects to represent asks and bids or the use of the `FourHeapOrderBook` (described in [73]) to manage shouts in auctions. Figure 5.2 presents the abstract structure of the

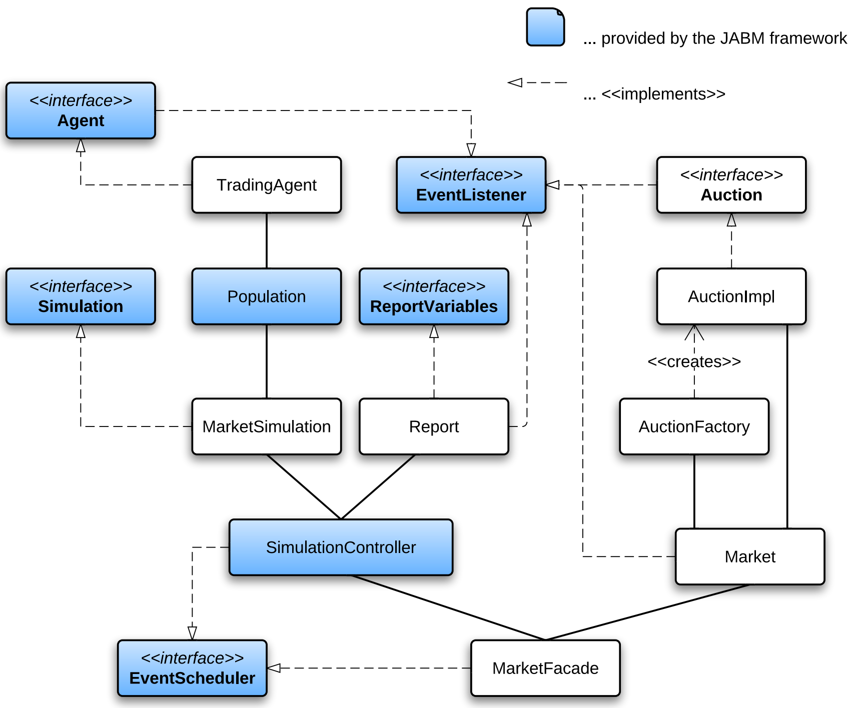


Figure 5.2: Framework structure in UML syntax

implementation of our simulation framework in UML notation. Objects modeled in blue are provided by the JABM framework. Since most of the components and their behavior shown in Figure 5.2 have already been discussed in Chapter 4, they will not be detailed again in the following.

The base class used to run a simulation is represented by the `MarketFacade`. Using the *Facade pattern* [25], agent-based modeling functionality provided by the JABM framework is combined with task-based crowdsourcing functionality created by us. By implementing the `EventScheduler` interface, the `MarketFacade` is able to send events to registered `EventListeners`. Functionality to broker those events, however, is implemented by the `SimulationController`, and the `MarketFacade` simply forwards all invocations of methods implemented by the `EventScheduler` interface to this controller. Since many objects, such as `TradingAgents` or `Market`, obtain references to the `MarketFacade` on creation, they can utilize the facade to communicate with other components by using events. For a component to receive events, it has to implement the `EventListener` interface. The behavior of `Trading Agents` is defined by `Policies` and `Strategies` as described in the previous chapter.

The `MarketSimulation` class defines the interactions between agents and the marketplace. As mentioned in Chapter 4, our simulation is based on discrete time simulation using events to inform components of the current *round*. Each round simulates a certain period of time. In our implementation a round can basically be divided into three phases:

1. *Arrival Phase*: Whenever a round starts agents are informed and may decide whether they want to join the market at the current round or not, by registering or unregistering to the market. Requesters at this phase submit tasks to the system, provided they want to trade in the market at this round. Once this phase is finished, the actual trading phase starts.

2. *Trading Phase*: At first, the market creates an `Auction` for each task submitted by requesters in the previous phase. Then, for each auction in the market, all agents trading in the respective auction are informed about the start of the trading phase and are from now on allowed to place their bids.

3. *Clearing Phase*: Before a round closes agents stop trading and each auction checks if the auction period is expired. In case it is, a winner is determined and the marketplace assigns the requester's task to a worker and informs both agents. Likewise, each worker in the simulation checks if he has finished a task in this round and informs the marketplace which in place informs the requester, demands a rating and manages the payment.

The `Market` uses an `AuctionFactory` to dynamically create auction instances of a certain auction type which is defined in the simulation model. Each object defining an auction type has to implement the `Auction` interface. Properties and behavior of the market and auctions have already been discussed in Chapter 4.

5.3 Creating the Simulation Model

The simulation model connects the implemented components together and therefore defines the simulation. All properties and references between components are defined in a declarative form in the simulation model. As in the JABM framework, our model file is denoted by an XML file whose type description is provided by the Spring standard XML Schema Definition (XSD) file, `spring-beans.xsd`. Using *Dependency Injection (DI)*, the Spring framework initializes and configures the denoted objects. There are two ways of dependency injection made possible by the Spring framework: *constructor-based DI* and *setter-based DI*. The former is accomplished by the Spring container invoking a bean's constructor and providing the dependencies as arguments, and the latter is accomplished by the Spring container calling the POJOs setter methods, after invoking a no argument constructor to instantiate the bean.

In the remainder of this section, examples showing the configuration of the main components in the simulation model are provided. References in these examples are set using either the aforementioned constructor-based DI denoted by `<constructor-arg ref="reference"/>` or the setter-based DI denoted by `<property name= "beanName" ref="beanRef"/>`.

Listing 5.1 shows how the basic structure of a simulation experiment is defined.

```
<!-- Configure the marketFacade to connect the controller with the Market-->
<bean id="marketFacade"
      class="at.ac.tuwien.dsg.crowdsim.market.MarketFacade">
      <property name="controller" ref="simulationController"/>
   <property name="market" ref="market"/>
</bean>

<!--  Configure a simulation which we will run just the once -->
<bean id="simulationController"
      class="net.sourceforge.jabm.SpringSimulationController">
```

```
  <property name="numSimulations" value="1"/>
  <!-- This is the name of the bean representing the simulation. -->
  <property name="simulationBeanName">
    <idref local="marketSimulation"/>
  </property>
  <!-- Report objects collect data on the simulation runs. -->
  <property name="reports">
    <list>
      <ref bean="workerCSVReport"/>
    </list>
  </property>
</bean>
```

Listing 5.1: Configuring the simulation environment

The `MarketFacade` is the top–level bean wiring the `SimulationController` and the crowdsourcing platform represented by the `Market` together. In the `reports` section, various reporting beans, implementing the `ReportVariable` interface, can be defined. The `simulationBeanName` property defines the underlying simulation type to be used, which in our case is the `MarketSimulation`. An example configuration of the latter is provided in Listing 5.2. This simulation bean defines how agents interact and contains the `population` specifying the agents. Moreover, the `maximumRounds` property specifies the number of rounds to be run in the simulation, whereas the `agentMixer` bean defines which agents interact with the market in a given round.

```
<bean id="marketSimulation" scope="prototype"
     class="at.ac.tuwien.dsg.crowdsim.market.MarketSimulation">
  <property name="maximumRounds" value="500"/>
  <property name="population" ref="population"/>
  <property name="agentInitialiser" ref="agentInitialiser"/>
  <property name="agentMixer" ref="randomRobinAgentMixer"/>
  <property name="simulationController" ref="simulationController"/>
</bean>
```

Listing 5.2: Configuring the market simulation

The `population` is represented by a list of agents, where the list may consist of sub lists containing different sets of agents as shown in Listing 5.3. Here, one list containing a set of workers and one list containing a list of requesters is created.

```xml
<!-- The population consists of workers and requesters -->
<bean id="population" scope="prototype"
    class="net.sourceforge.jabm.Population">
  <property name="agentList">
    <bean class="net.sourceforge.jabm.agent.AgentList">
      <constructor-arg>
        <list>
          <ref bean="requesterAgentList"/>
          <ref bean="workerAgentList"/>
        </list>
      </constructor-arg>
    </bean>
  </property>
  <property name="prng" ref="prng"/>
</bean>
```

Listing 5.3: Configuring the population

The creation of workers is shown in Listing 5.4. The first constructor argument specifies the number of agents to be created for a given agent list. In this example, thousand workers are constructed on simulation start. Each worker is constructed from the specified prototype bean `workerAgent` using a `ObjectFactory` which is provided by the Spring framework. The prototype bean specifies the class used to represent the agent and defines its behavior. Besides the already in Chapter 4 discussed properties representing behavior (i.e. `tradingStrategy`, `valuationPolicy` and `transactionBook`) the `skillConfig` property has to be set too. On agent creation, the skills are initialized using randomly drawn values from a Gaussian distribution; the configuration for this distribution is provided by the aforementioned `skill Config` property.

```xml
<!-- An agent list comprising 1000 workers -->
<bean id="workerAgentList" class="net.sourceforge.jabm.agent.AgentList">
  <constructor-arg ref="1000"/>
  <constructor-arg ref="workerAgentFactory"/>
</bean>

<!-- The factory used for manufacturing the workers -->
<bean id="workerAgentFactory"
     class="org.springframework.beans.factory.config.
ObjectFactoryCreatingFactoryBean">
  <property name="targetBeanName">
    <idref local="workerAgent"/>
  </property>
</bean>

<!-- The prototype used to manufacture workers -->
<bean id="workerAgent" scope="prototype"
     class="at.ac.tuwien.dsg.crowdsim.agent.worker.SimpleWorker">
  <property name="marketFacade" ref="marketFacade"/>
  <property name="scheduler" ref="simulationController"/>
  <property name="valuationPolicy" ref="marketOrientedValuation"/>
  <property name="transactionBook" ref="simpleTransactionBook"/>
  <property name="tradingStrategy" ref="truthTellingStrategy"/>
  <property name="skillConfig" ref="experiencedWorkerConfig"/>
  <property name="randomData" ref="randomData"/>
</bean>

<!-- The configuration for experienced Workers-->
<bean id="experiencedWorkerConfig"
     class="at.ac.tuwien.dsg.crowdsim.agent.worker.SkillConfig">
  <property name="meanPerformance" value="0.70"/>
  <property name="deviancePerformance" value="0.25"/>
  <property name="meanConfidence" value="0.8"/>
  <property name="devianceConfidence" value="0.25"/>
</bean>
```

Listing 5.4: Creating agent lists

Listing 5.5 shows the configuration of the `Market` component. Besides beans defining skill recognition and quality management behavior, an `auctionFactory` bean has to be specified. The auction factory creates and configures an `Auction` for each task submitted by the requester using a prototype bean to set `auctionType` and `orderBook`.

Further, the `auctionDuration` property defines how many rounds an auction runs before it closes and determines a winner.

```xml
<!-- Configures the crowdsourcing platform -->
<bean id="market"
    class="at.ac.tuwien.dsg.crowdsim.market.Market">
  <property name="marketFacade" ref="marketFacade"/>
  <property name="auctionFactory" ref="auctionFactoryBean"/>
  <property name="skillUpdatePolicy" ref="weightedSkillUpdate"/>
  <property name="auctionQualificationPolicy"
    ref="minPerformanceQualification"/>
</bean>

<!-- Configures and creates the auction type -->
<bean id="auctionFactoryBean"
    class="at.ac.tuwien.dsg.crowdsim.market.auction.AuctionFactory">
  <property name="auctionPrototype" ref="sealedBid"/>
  <property name="orderBookProtoType" ref="lowestPriceOrderBook"/>
  <property name="auctionDuration" value="1"/>
</bean>
```

Listing 5.5: Configuring the market

Finally, Listing 5.6 shows how the random generator is configured. We use the Mersenne Twister algorithm [44] as it is more powerful than the standard Java `Random` implementation.

```xml
<!--  The random data object used to generate distributions -->
<bean id="randomData" scope="singleton"
    class="org.apache.commons.math.random.RandomDataImpl">
  <constructor-arg ref="randomGenerator"/>
</bean>
<!-- The pseudo-random number generator algorithm -->
<bean id="randomGenerator" scope="prototype"
    class="org.apache.commons.math.random.MersenneTwister">
  <constructor-arg ref="seed"/>
</bean>
<!-- The seed to generate reproducable results-->
<bean id="seed" scope="singleton" class="java.lang.Long">
  <constructor-arg value="123"/>
</bean>
```

Listing 5.6: Configuring the random generator

CHAPTER 6

Evaluation

In this chapter, the suitability of different auction types for five different scenarios is evaluated using the simulation framework described in Chapter 4 and in Chapter 5. At first, the evaluation design is described. Then, in the first three scenarios (Section 6.2.1 to Section 6.2.3) three standard auction types are evaluated using different populations in each scenario. In Scenario 4 (Section 6.2.4), a standard auction type is compared to a double auction type. Lastly, in Scenario 5 (Section 6.2.5), three different quality management methods are compared by means of a reversed sealed-bid second price auction.

6.1 Evaluation Design

To evaluate the suitability of different auction mechanisms in a given environment several series of experiments are set up. Each experiment series describes a *scenario* consisting of a certain population of workers and requesters and a crowdsourcing marketplace with defined behavior regarding skill recognition and quality management strategies. In each series the same initial population is used, the auction type is however altered. Then, the results of the experiments in a series are compared with each other, to discuss advantages and disadvantages of each auction type used in the series. Figure 6.1 shows an illustration of the experiment design.

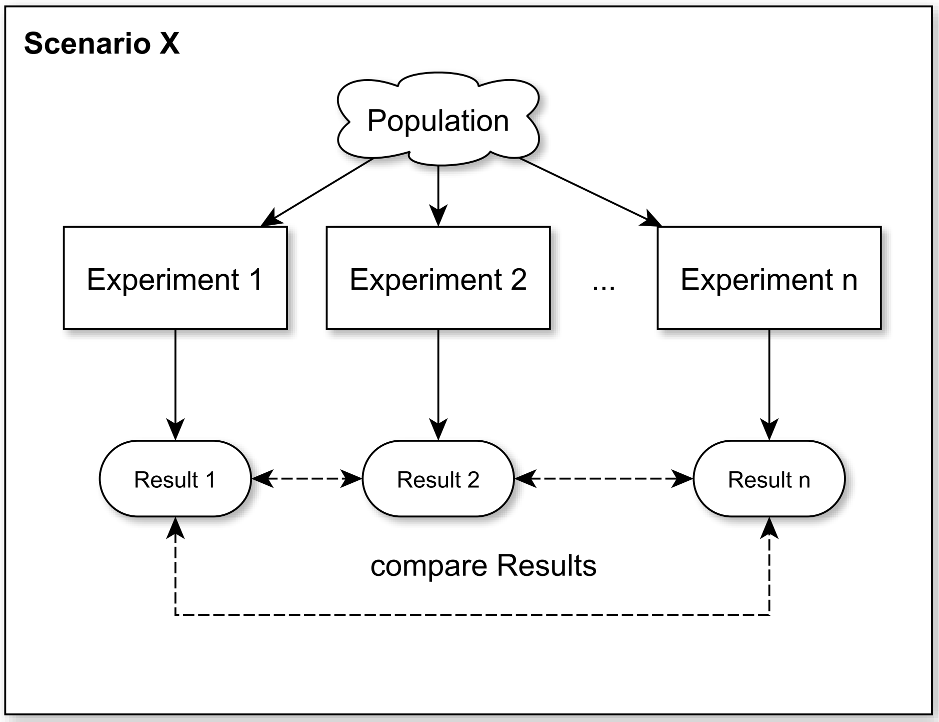


Figure 6.1: Evaluation design of a scenario X containing a series of experiments

For each experiment the following values are analyzed:

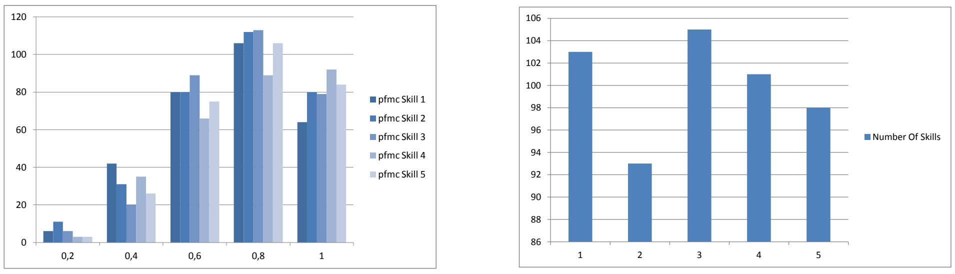
- *Submitted Tasks:* The number of submitted tasks represents the *loading* of the system and therefore indicates the satisfaction of requesters as they only submit tasks to the system if they are satisfied. Therefore, the more tasks submitted the better for the platform.
- *Assigned Tasks:* The number of tasks assigned to workers indicates the qualification of workers for submitted tasks on the one hand (as only qualified workers are allowed to join an auction in our experiments), and the interest of workers to process the submitted tasks on the other hand (as workers won't submit bids for tasks with a too low price). The more tasks assigned the better for the platform.
- *Unassigned Tasks:* The lower the number of unassigned tasks the better for the

crowdsourcing platform, as requesters, ideally, want all their submitted tasks processed.

- *Completed Tasks:* The number of tasks processed by requesters in time. The more tasks processed the better for the platform.
- *Deadline Violations:* The number of tasks processed by workers but not submitted in time. The lower the number of deadline violations the better for the platform.
- *Quality:* This value represents the average quality of tasks processed by workers. The higher the quality, the better it is for the system.
- *Payoff:* This value represents the average profit made by an agent. The payoff of workers and requesters are analyzed separately. The payoff of workers is calculated as $p_w = price - privateValue$ as they only make profit if they earn more money than they actually need to process a task. The payoff of requesters is calculated as $p_r = privateValue - price$ as requesters only make profit if they can buy workforce at a price level lower than their own valuation. In an ideal system, requesters and workers will make the same profit.

6.2 Evaluation Scenarios

This section provides evaluations of different scenarios using different populations. The experiments are set up using the simulation framework described in Chapter 4 and Chapter 5. Each of the following simulations will run for 500 rounds. The default marketplace behavior is described by the `MinPerformanceQualificationPolicy` and the `WeightedSkillUpdatePolicy`. The marketplace therefore only invites those workers to auctions who meet the defined quality requirements of a task as described in Chapter 4. The skill update rates are set to $pfmcUpdateRate = 0.2$ and $cnfdUpdateRate = 0.1$. To provide an initial configuration of the skill recognition module, we draw confidence values from the Gaussian distribution $\mathcal{N}(0.8, 0.25)$ and calculate the observed performance values as described in Chapter 4 to simulate proper knowledge of the workers' skills. Unless stated otherwise, the marketplace supports five



(a) Worker Performance (b) Skill Distribution

Figure 6.2: Generated worker population according to Scenario 1

different skills. A sample simulation model containing the configuration of an experiment in Scenario 1 is provided in Listing B.1 in Appendix B.

6.2.1 Scenario 1 - A Large, High-Skilled Crowd

In this scenario we simulate a large, high-skilled crowd and compare three standard reverse auction types with each other: *Reversed Sealed-Bid Second-Price Auction*, *Reversed English Auction* and *Reversed Dutch Auction*[1].

The population consists of 20 requesters and 500 workers which simulates a high worker/requester ratio, i.e. a relatively large crowd of workers. Requesters implement a `marketOrientedValuationPolicy` for private value generation and a `BeatTheQuoteTradingStrategy` defining the requesters trading behavior. They submit one task per round containing up to five randomly generated required skills within range $[0.1, 0.9]$. The Gaussian distribution to set the workers skills is set to $\mathcal{N}(0.7, 0.25)$, with each worker possessing up to five skills. The generated worker population is shown in Figure 6.2. The workers private values are generated using the `marketOrientedValuationPolicy` and their trading strategy is set to the `LoadBasedBeatTheQuoteTradingStrategy`. The configuration of the latter is as follows: $quoteMaxMarkupRate = 0.05$, $loadMaxMarkupRate = 0.5$, $maxLoadRate = 1$. Each of the three auctions

[1]Theory of these auction types is discussed in Chapter 2; Implementation aspects in Chapter 4

implements the `LowestPriceOrderBook`, letting the worker with the lowest bid win the auction.

Table 6.1 shows the results of the simulation. In terms of *task distribution*, the results

	Sealed-Bid	**English**	**Dutch**
Submitted Tasks	3561	2611	3695
Assigned Tasks (%)	3399 95.45%	2505 95.94%	3552 96.13%
Unassigned Tasks (%)	162 4.55%	106 4.06%	143 3.87%
Completed Tasks (%)	3253 91.35%	2237 85.68%	3374 91.31%
Deadline Violations (%)	146 4.10%	268 10.26%	196 4.82%
Average Quality	0.8442	0.7907	0.8266
Payoff Worker *Average*	122.98 0.0362	66.78 0.0267	98.12 0.0278
Payoff Requester *Average*	644.64 0.1981	402.93 0.1801	658.74 0.1952

Table 6.1: Evaluation results for Scenario 1

of the reversed sealed-bid second-price auction show no significant difference. In both auction types a total number of about 3600 tasks is submitted and about 95% of the submitted tasks are assigned to workers; 91% of the submitted tasks are completed. Using the Dutch auction, there are slightly more assigned tasks than by using the sealed-bid auction, although slightly more deadline violations occur using the former. Proportionally, significantly more deadline violations occur within the English auction compared to the other auction types. This impacts the total number of submitted task which is significantly lower ($\sim$ 40%) compared to the sealed-bid auction and the Dutch auction. The high number of deadline violations may be explained by the workers' trading behavior and the auction design: sealed-bid auctions clear after one round, and Dutch auctions are cleared as soon as a valid bid is placed, thus tasks in those auction types are assigned to workers within a relatively short period of time. English auctions, in contrast, accept bids for a defined (longer) period of time and as a consequence the pe-

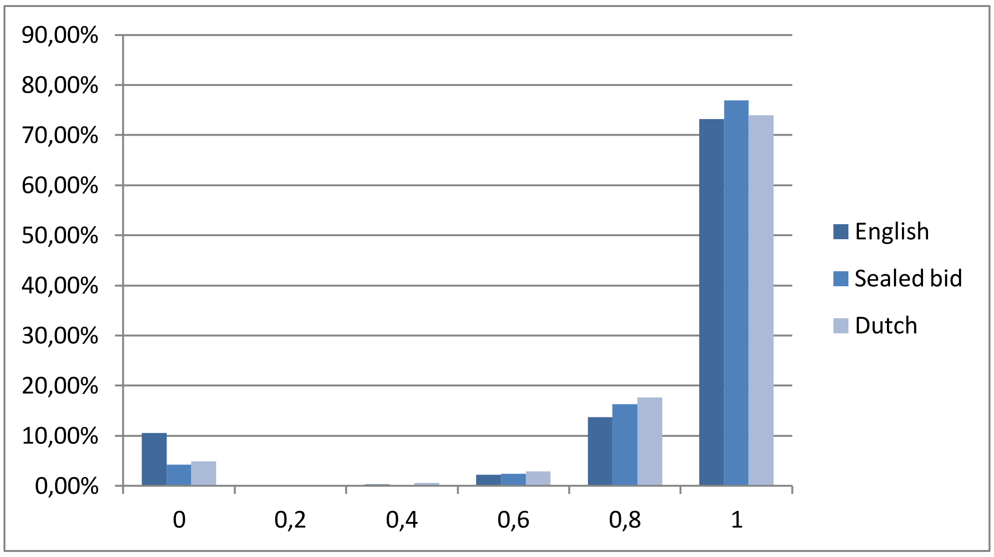


Figure 6.3: Task quality distribution for Scenario 1

riod of time between bid placement and task assignment is longer than in the other two auction types. As workers with greedy behavior place bids in many different auctions each round (provided that their workload is not too high), they might win more tasks than they could process, which in turn leads to deadline violations.

The *quality* of assigned tasks processed in the sealed-bid auction is slightly higher than in the Dutch auction, which in turn is only slightly higher than in the English auction. The English auction obtains the lowest quality results which may be attributed to the higher number of deadline violations (which are rated with zero). A histogram of the quality distribution is provided in Figure 6.3.

The average *payoff* of workers is significantly higher ($\sim$ 40%) for the sealed-bid auction than for the English and the Dutch auction (the latter both achieving nearly equal values). This may be explained by the fact that for the English and the Dutch auction the current market prices are visible to agents, leading to massive undercutting due to the high number of workers and thus to relatively little profit. Furthermore, in the sealed-bid auction, the clearing price is set to the price of the secend lowest bid, which also lets workers make higher profits. The average payoff of requesters is, compared to the average payoff of workers, significantly higher (about six times as much). This may be explained by the fact that the high worker/requester ratio forces workers to bid closer to their own private values to win a task, therby enabling requesters to make a higher profit. What has to be considered is that on certain task-based crowdsourcing platforms requesters have to pay for the services provided by the system which in turn would lower the total profit. Based on the results we can state, that the sealed-bid auction provides the highest average payoff for requesters followed by the Dutch auction providing nearly equal values. The English auction provides about 10% less payoff for requesters compared to the sealed-bid auction.

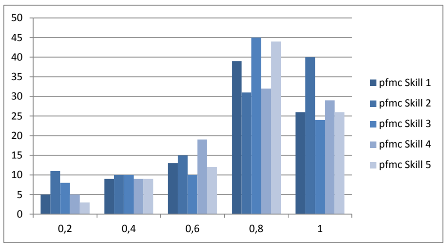
In summary it can be stated that for a large, high-skilled crowd, the results provided by the reversed sealed-bid second price auction are almost similar compared to the results provided by the reversed Dutch auction, with the former tending to provide better results. The reversed English auction provides the worst results of the three examined auction types.

6.2.2 Scenario 2 - A Small High-Skilled Crowd

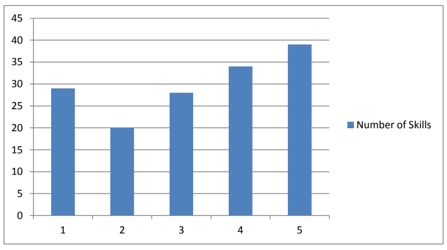
In this scenario we simulate a small, high skilled crowd and compare three standard reverse auction types with each other: *Reversed Sealed-Bid Second-Price Auction*, *Reversed English Auction* and *Reversed Dutch Auction*.

The population consists of 20 requesters and 150 workers which simulates a low worker/requester ratio, i.e. a relatively small crowd of workers. The agent configuration is equal to the configuration described in Scenario 1 and each of the three auctions used for the simulation implements the `LowestPriceOrderbook` as well. The population used for the experiments is shown in Figure 6.4.

In Table 6.2, the results of each experiment are listed. As in Scenario 1, the reversed sealed-bid auction and the Dutch auction achieve similar results in terms of *task distribution* with the latter obtaining a slightly higher number of submitted tasks, and proportionally a slightly higher number of assigned and completed tasks. The English



(a) Worker Performance



(b) Skill Distribution

Figure 6.4: Generated worker population according to Scenario 2

	Sealed-Bid	**English**	**Dutch**
Submitted Tasks	1881	1495	1991
Assigned Tasks (%)	1572 83.57%	1247 83.41%	1686 84.68%
Unassigned Tasks (%)	309 16.43%	248 16.59%	305 15.32%
Completed Tasks (%)	1486 79.00%	1071 71.64%	1579 79.31%
Deadline Violations (%)	86 4.57%	176 11.77%	107 5.37%
Average Quality	0.8241	0.7565	0.8038
Payoff Worker Average	63.97 0.0406	42.83 0.0383	53.86 0.0319
Payoff Requester Average	255.40 0.1718	171.13 0.1597	288.94 0.1829

Table 6.2: Evaluation results for Scenario 2

auction, again, achieves the lowest number of submitted tasks ($\sim$ 25% less compared to the sealed-bid auction) and proportionally the highest number of deadline violations; 11.77% compared to 5.42% (Dutch) and 4.63% (sealed-bid). The reason for the significantly lower number of submitted tasks in the English auction can be explained by the higher number of deadline violations as requesters submit fewer tasks in case their already submitted tasks are not finished in time. The higher number of deadline violations may be explained by the workers' trading behavior and the auction design, as already described in Scenario 1 (see Section 6.2.1). The relatively high number of unassigned tasks, compared to Scenario 1, may be explained by the small crowd as workers do not

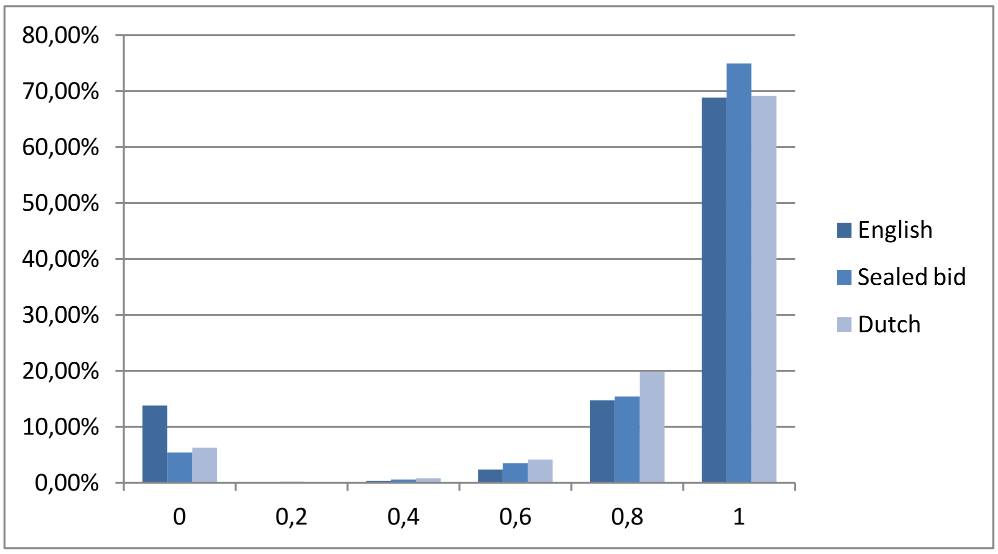
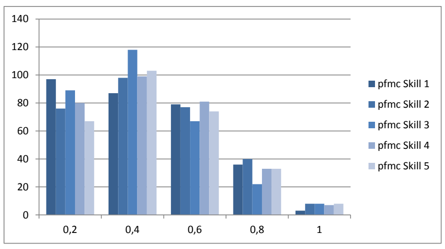


Figure 6.5: Task quality distribution for Scenario 2

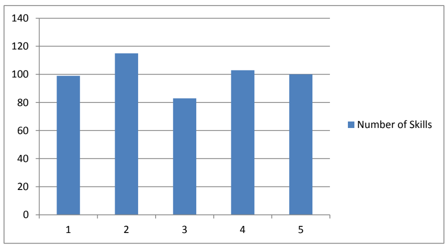
compete for certain tasks due to their high workload.

The achieved average *quality* per assigned task is the highest in the sealed-bid auction (0.82) followed by the Dutch auction (0.80) and the English auction (0.75). The low average quality in the English auction can, again, be attributed to the high number of deadline violations as shown in Figure 6.5.

Regarding the average worker *payoff*, one can state that, compared to Scenario 1, higher values are achieved in each experiment. This can be explained by the smaller crowd, which leads to less competition, and therefore to higher profits for workers. Workers in the reversed sealed-bid auction obtain the highest average payoff (0.0406). Interestingly, workers in the English auction obtain almost equal average payoff values (0.0383), compared to workers in the sealed-bid auction which may also be attributed to the small crowd. Workers competing in the Dutch auction make the lowest average payoff (0.0319). In contrast to workers, requesters trading in Dutch auctions make the highest payoff (0.1829) followed by requesters trading in sealed-bid auctions (0.1718) and English auctions (0.1597).



(a) Worker Performance



(b) Skill Distribution

Figure 6.6: Generated worker population according to Scenario 3

In summary, it can be stated that for small high-skilled crowds the results achieved by the reversed sealed-bid second price auction are similar to the results achieved by the reversed Dutch auction, with the former tending to provide slightly better results. Results of the English auction, except for the average worker payoff, provide the worst results. Compared to Scenario 1, there are fewer assigned tasks and lower average requester payoffs, but higher average worker payoffs.

6.2.3 Scenario 3 - A Large, Low-Skilled Crowd

In this scenario, simulate a large, low-skilled crowd and compare three standard reverse auction types with each other: *Reversed Sealed-Bid Second-Price Auction*, *Reversed English Auction* and *Reversed Dutch Auction*.

The population consists of 20 requesters and 500 workers which simulates a high worker/requester ratio, i.e. a relatively large crowd of workers. The behavior of requesters and workers is configured as described in Scenario 1. The Gaussian distribution to set the workers skills is set to $\mathcal{N}(0.3, 0.25)$, with each worker possessing up to five skills. The generated worker population is shown in Figure 6.6. As in the last two scenarios, all three auction types implement the `LowestPriceOrderbook`, letting the worker submitting the lowest price win the auction.

	Sealed-Bid	**English**	**Dutch**
Submitted Tasks	1286	1126	1325
Assigned Tasks *(%)*	900 69.98%	806 71.58%	961 72.53%
Unassigned Tasks *(%)*	386 30.02%	320 28.42%	364 27.47%
Completed Tasks *(%)*	864 67.17%	699 62.08%	900 67.92%
Deadline Violations *(%)*	36 2.79%	107 9.50%	61 4.60%
Average Quality	0.7662	0.7016	0.7482
Payoff Worker *Average*	28.51 0.0317	24.75 0.0307	25.63 0.0266
Payoff Requester *Average*	114.71 0.1328	79.83 0.1142	109.08 0.1212

Table 6.3: Evaluation results for Scenario 3

Table 6.3 shows the simulation results for Scenario 3. The results of this scenario conform with the observations made in Scenario 1 and Scenario 2 since, again, all three auctions obtain similar results with the reversed sealed-bid auction tending to achieve the best results and the English auction tending to achieve the worst. In detail, one can see that compared to the previous two scenarios, the percentage of unassigned tasks is significantly higher as there are much more low-skilled and thus unqualified workers. As a result, the total number of submitted tasks is lower compared to Scenario 1 and Scenario 2. Proportionally, in the Dutch auction the most tasks are assigned to workers (72.53%) followed by the English auction (71.57%) and the sealed-bid auction (69.98%). In terms of deadline violations, the sealed-bid auction provides the best results followed by the Dutch auction (4.60%) and the English auction (9.50%). The high number of deadline violations in the English auction can, once again, be attributed to the workers' behavior and the auction design (see Scenario 1).

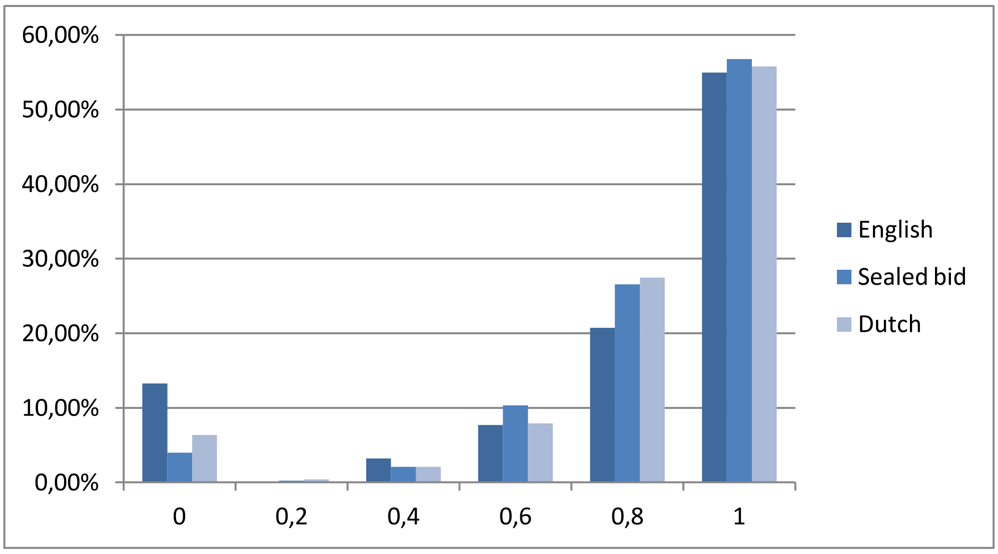


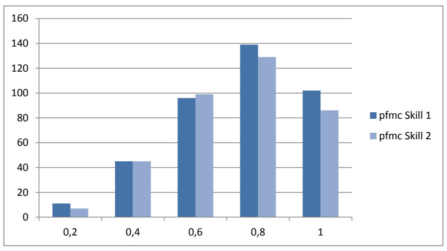
Figure 6.7: Task quality distribution for Scenario 3

The average *quality* of assigned tasks, is also slightly lower compared to Scenario 1 and Scenario 2, with the sealed-bid auction obtaining the best results and the English auction obtaining the worst. As Figure 6.7 shows, this can be explained due to the higher number of deadline violations, as they are rated with zero.

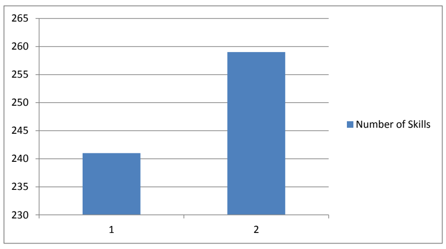
In terms of average *payoff*, workers using the sealed-bid auction (0.0316) and the English auction (0.0307), once again, make the most profit obtaining similar values and workers using the Dutch auction obtain the worst results (0.0266). The payoff of requesters is the highest in the sealed-bid auction, followed by the Dutch auction and the English auction.

6.2.4 Scenario 4 - Standard Auction vs. Double Auction

In this scenario, we simulate a large, high-skilled crowd and compare the reverse sealed-bid auction to a continuous double auction.



(a) Worker Performance



(b) Skill Distribution

Figure 6.8: Generated worker population according to Scenario 4

For the double auction experiment, we create two double auction marketplaces for each skill supported by the marketplace; one for medium quality (0.6) and one for high quality (0.8). Since objects traded in double auctions have to be homogeneous (see Chapter 2), and we basically trade skills, double auctions are not suited to support many skills. Thus, we restrict the marketplace to support two skills only in this scenario. Theory and implementation details regarding double auctions are described in Section 4.6.1. The population for these experiments consists of 20 requesters and 500 workers. Requesters implement the `SkillEffortAwareValuationPolicy` for private value generation and a `BeatTheQuoteTradingStrategy` to define the trading behavior. Each round, a requester generates a task (with skills within range $[0.1, 0.8]$), splits it into sub tasks and buys workforce in respective auction markets (according to the quality requirement of the sub task). The detailed behavior for requesters trading in double auction markets is described in Section 4.6.1. Workers possess up to two skills with values randomly drawn from the Gaussian distribution $\mathcal{N}(0.7, 0.25)$. The used worker population is shown in Figure 6.8. The private values of workers are generated using the `SkillEffortAwareValuationPolicy` and their trading behavior is defined by the `LoadBasedBeatTheQuoteTradingStrategy` (Configuration: $quoteMaxMarkupRate = 0.05, loadMaxMarkupRate = 0.5, maxLoadRate = 1$).

Table 6.4 shows the results of both experiments.

	Sealed-Bid	**Continuous Double**
Submitted Tasks	5085	3137
Assigned Tasks (%)	5083 99.96%	2733 87.12%
Unassigned Tasks (%)	2 0.04%	404 12.88%
Completed Tasks (%)	4867 95.71%	2941 87.12%
Deadline Violations (%)	216 4.25%	0 0.00%
Average Quality	0.8317	0.8455
Payoff Worker *Average*	141.16 0.0278	446.71 0.0615
Payoff Requester *Average*	1063.44 0.2185	446 0.1634

Table 6.4: Evaluation results for Scenario 4

In terms of *task distribution*, requesters in the sealed-bid auction submit a significantly higher number of tasks than the requesters in the double auction. This may be inferred from the fact that in the double auction about 14% of submitted tasks are not assigned to a worker, letting the requesters submit fewer tasks. The high number of unassigned tasks may be attributed to the requesters trading behavior, as they place asks in auctions that are based on the current quote. This behavior may lead to asks with too low prices for which no matching bid can be found, i.e. no worker wants to work for that little money. If trading in double auctions, requesters split a task and assign it to multiple workers which entails that, in our simulation, all tasks are processed in time.

The *quality* of processed tasks traded in double auctions is slightly better than the quality of tasks traded in sealed-bid auctions (0.8455 compared to 0.8316), which may be inferred from the slightly different qualification methods used in both auctions. Suppose a requester wants a task with 0.7 minimum quality processed. In the sealed-bid auction, a worker providing 0.7 quality is allowed to process the task whereas in the double auction the requester has to trade workforce in the auction market trading at least 0.8 minimum quality. This "quality gap" may provide slightly better results for the double

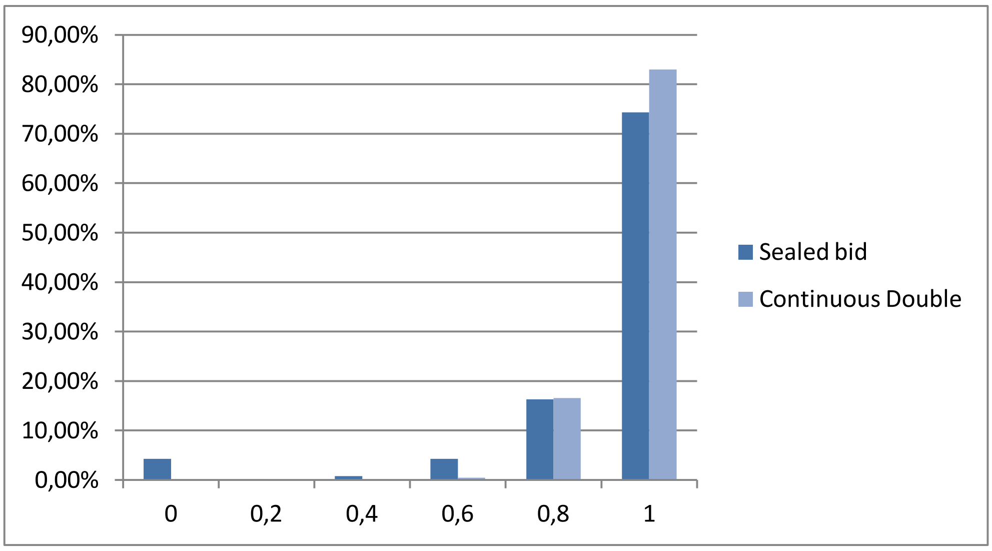


Figure 6.9: Task quality distribution for Scenario 4

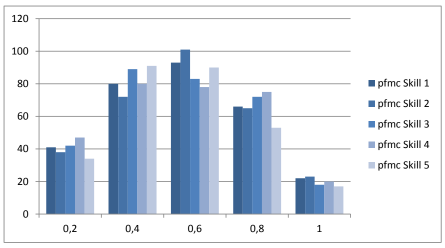
auction. The distribution of quality is shown in Figure 6.9.

In terms of *payoff*, one can state that workers trading in double auctions make an significantly higher average payoff than workers trading in sealed-bid auctions; 0.0278 in sealed-bid auctions to 0.0615 in double auctions. Requesters, in contrast, make about 25% less payoff when trading in double auctions. This may be explained by the partitioning of the marketplace, as tasks have to be submitted in markets that provide higher quality than required. In such markets, the prices are typically higher and hence workers make higher profits.

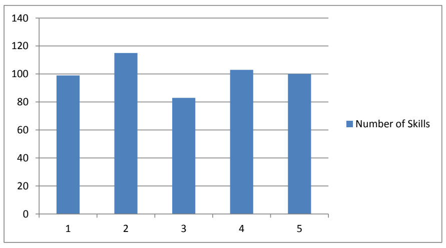
6.2.5 Scenario 5 - Quality Management Methods

In this scenario, we simulate a large, low-skilled crowd and compare three different quality management methods with each other, using the *reversed sealed-bid second-price auction* for task distribution. The three different quality management methods are:

1. *No Quality Management:* In the first experiment, we use no quality management at all.



(a) Worker Performance



(b) Skill Distribution

Figure 6.10: Generated worker population according to Scenario 5

2. *Qualification Policy:* In the second experiment, we use the platform's skill recognition functionality in combination with a `MinPerformanceQualificationPolicy`. Using this quality management method, only those workers are invited to join auctions, whose system profile information matches the skills required for the task.

3. *Order Book:* In the third experiment, we use the platform's skill recognition functionality in combination with the `WeightedScoreOrderBook` and the qualification policy mentioned in point 2. Using this strategy, only qualified workers are invited to join an auction and the worker with the best *suitability* wins the auction. Detailed information regarding the `WeightedScoreOrderBook` is provided in Section 4.6.1. The `OrderBook` component is configured as follows: $x = 0.6; y = 0.3; z = 0.1$.

The population consists of 20 requesters and 500 workers which simulates a high worker/requester ratio, i.e. a relatively large crowd of workers. The behavior of requesters and workers is configured as described in Scenario 1. The Gaussian distribution used to generate the workers' skills is set to $\mathcal{N}(0.3, 0.25)$, with each worker possessing up to five skills. The generated worker population is shown in Figure 6.10.

Table 6.5 contains the results of the three experiments. In the first experiment (No QM), each submitted task is assigned to a worker due to the large crowd and the absence of a pre-selection of workers. In contrast to this, in the "Policy experiment" only 70% and in

	No QM	**Policy**	**Order Book**
Submitted Tasks	3080	1302	1166
Assigned Tasks	3080	912	777
(%)	100.00%	70.05%	66.64%
Unassigned Tasks	0	390	389
(%)	0.00%	29.95%	33.36%
Completed Tasks	2760	878	735
(%)	89.61%	67.43%	63.04%
Deadline Violations	320	34	42
(%)	10.39%	2.61%	3.60%
Average Quality	0.6588	0.7789	0.7732
Payoff Worker	48.09	28.91	28.61
Average	0.0156	0.0317	0.0368
Payoff Requester	684.82	116.51	56.14
Average	0.2481	0.1227	0.0763

Table 6.5: Evaluation results for Scenario 5

the “Order Book experiment” only 67% of the submitted tasks are assigned to workers. The pre-selection, however, has a positive impact on the number of tasks with deadline violations, as there occur significantly fewer violations in the Policy experiment (2.6%) and the Order Book experiment (3.6 %) compared to the No QM experiment(10.4%). The slightly higher percentage of deadline violations in the Order Book approach, compared to the Policy approach, may be explained by the fact that the “best” workers win many auctions which in turn leads to a (too) high workload and therefore to deadline violations.

In terms of quality, the Policy approach and the Order Book approach achieve similar results, which are significantly better than the results achieved in the No QM approach; 0.7789 and 0.7732 compared to 0.6588. As Figure 6.11 shows, the Order Book approach provides slightly better results than the Policy approach, in case a task is completed, but the aforementioned higher number of deadline violations lowers the average quality level.

Because of the high number of workers and the resulting competition the experiment without quality management achieves the highest average payoff for requesters (0.2481)

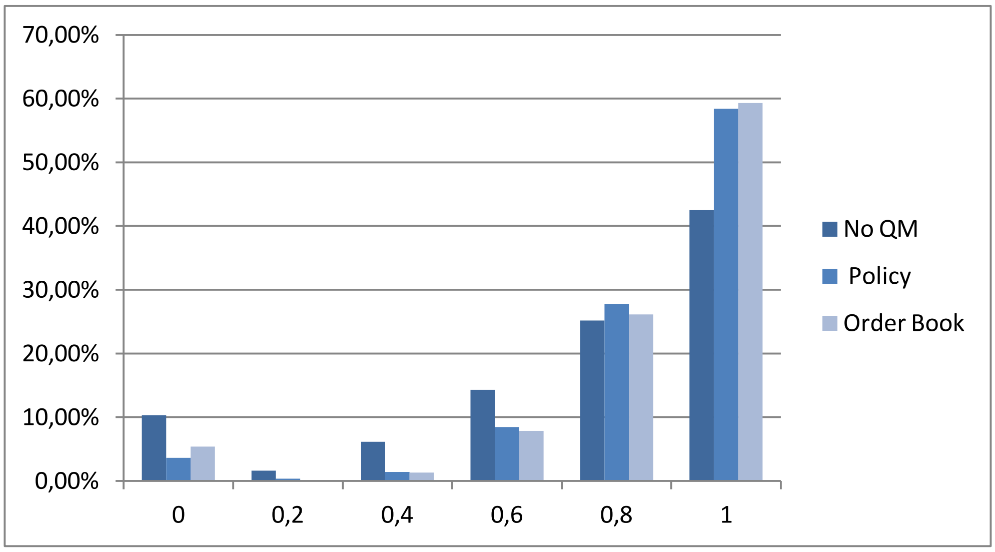


Figure 6.11: Task quality distribution for Scenario 5

and the lowest payoff for workers (0.0156). Using the Order Book approach, workers make the highest payoff (0.0368) and as a consequence requesters make the lowest (0.076). This can be attributed to the fact that the winner is determined by the observed performance, the confidence in this performance and the price, thus not the cheapest worker but the best suited wins an auction. Requesters in the Policy approach make about twice as much profit (0.1326) as in the Order Book approach, and workers make about 15% less (0.0317).

In summary it can be stated that using no quality management results in the use of cheap workforce with the quality of results being correspondingly low. In contrast, a quality management approach that determines an auction winner based on the suitability provides the best quality results and the highest payoff for workers. The requesters, however, make the lowest profit in this approach. Using worker pre-selection provides quality results similar to the suitability approach, with requesters making about twice as much profit.

CHAPTER 7

Summary and Future Work

Crowdsourcing has become a popular outsourcing strategy for companies lately. Especially task-based crowdsourcing has become the focus of attention, as highly available workforce can be flexibly allocated at a generally low cost. Currently available task-based crowdsourcing platforms, however, have to face primarily two different challenges. Firstly, they have to cope with the varying quality of processed tasks, and secondly, they have to provide adequate methods for assigning tasks to workers. For the latter, one approach is to use auction mechanisms as means of task distribution. The suitability of different auction mechanisms in task based crowdsourcing, however, has not been discussed in related literature yet. Another challenge in task-based crowdsourcing is the difficulty of testing and evaluating new approaches due to the scarcity of "real" data for evaluations.

In this thesis we have addressed the lack of an appropriate simulation framework for task-based crowdsourcing platforms and have come up with a solution. We have developed a highly configurable, modular and extensible framework which is based on an agent-based modeling approach. This framework supports different quality management and task distribution mechanisms to simulate the behavior of common crowdsourcing platforms. More precisely, a skill recognition strategy to cope with quality requirements and various auction types for task distribution has already been implemented by the framework. Further, by means of a population component the user to can define a certain crowd which interacts with the marketplace.

By means of this simulation framework, several experiments have been run, to analyze the suitability of different auction types for given scenarios. We have compared three reversed standard auction types (sealed-bid second-price, Dutch, English) and one double auction type (continuous double auction) with each other and have discussed the results. The evaluation presented in Chapter 6 has shown that, independent of the crowd, among the standard auction types the reversed sealed-bid second-price auction provides the most balanced and, in our opinion, the best results in terms of quality, task distribution and payoff. However, the Dutch auction provides similar results which are just slightly worse compared to the sealed-bid auction. Based on the evaluation of the continuous double auction compared to the reversed sealed-bid second-price auction, it can be concluded that a double auction market may have its advantages in terms of worker payoff, number of deadline violations and quality. In terms of requester payoff and assigned tasks, the sealed-bid auction, however, achieves better results. Furthermore, three different quality management methods have been analyzed leading to the conclusion that the use of an auction mechanism that matches tasks to the best suited worker achieves the best quality results and the highest payoff for workers but as a consequence, requesters make the lowest payoff. The most balanced approach is to use a qualification policy (i.e. inviting only suited workers to an auction), as the quality results are similar and the results regarding requester payoff and assigned tasks are better, compared to the auction-based matching approach.

7.1 Future Work

The prototype implementation of the simulation framework has already implemented various components defining marketplace and agent behavior. Even though it is fully functioning in its current state the following improvements may be made in the future:

- The implemented strategies defining the agents trading behavior are based on a relatively simple model in which agents react on a current state, limiting the predictive power of the framework. By implementing more powerful models using artificial intelligence approaches, the predictive power may be increased.

- Visualizations of simulated results are a common feature of agent-based modeling frameworks. Currently there is, however, no support of visualizations in our framework but as the JABM framework provides basic visualization functionality it should be no problem to implement this feature in the future.
- As approaches to process large-scale tasks (consisting of work flows and subtasks) by means of social network-based collaboration are discussed in recent literature, it might be useful to implement functionality to simulate these new approaches.

Appendices

APPENDIX A

List of Abbreviations

ABM	Agent-based Modeling
AMT	Amazon Mechanical Turk
API	Application Programming Interface
CDA	Continuous Double Auction
CH	Clearinghouse Auction
DI	Dependency Injection
E&R	Encouragement and Retention
HIT	Human Intelligence Task
HPS	Human Provided Services
ICP	Imaginary Crowdsourcing Platform
IoC	Inversion of Control
JABM	Java Agent-Based Modeling
JASA	Java Auction Simulator API
OCR	Optical Character Recognition
POJO	Plain Old Java Object
QM	Quality Management
SBS	Software-Based Services

Continued on Next Page . . .

SCU	Social Compute Unit
SLA	Service Level Agreement
SOA	Service Oriented Architecture
TBCS	Task-based Crowdsourcing
UML	Unified Modeling Language
XML	Extensible Markup Language
XSD	XML Schema Definition

Table A.1: List of Abbreviations

APPENDIX B

A Sample Configuration File

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="marketFacade" class="at.ac.tuwien.dsg.crowdsim.market.MarketFacade">
        <property name="controller" ref="simulationController"/>
        <property name="market" ref="market"/>
    </bean>

    <bean id="nrOfRequestersTyp1"  class="java.lang.Integer">
        <constructor-arg value="20"/>
    </bean>
    <bean id="nrOfWorkersTyp1"  class="java.lang.Integer">
        <constructor-arg value="500"/>
    </bean>
    <bean id="simulationController"
          class="net.sourceforge.jabm.SpringSimulationController">
        <property name="numSimulations" value="1"/>
        <property name="simulationBeanName">
            <idref local="marketSimulation"/>
        </property>
        <property name="reports">
            <list>
                <ref bean="workerCSVReport"/>
                <ref bean="workerSimulationFinishedCSVReport"/>
                <ref bean="workerReportVariables"/>
                <ref bean="requesterSimulationFinishedCSVReport"/>
                <ref bean="requesterReportVariables"/>
                <ref bean="transactionCSVReport"/>
                <ref bean="transactionReportVariables"/>
            </list>
        </property>
    </bean>

    <bean id="marketSimulation" scope="prototype"
          class="at.ac.tuwien.dsg.crowdsim.market.MarketSimulation">
        <property name="maximumRounds" value="500"/>
        <property name="population" ref="population"/>
        <property name="agentInitialiser" ref="agentInitialiser"/>
        <property name="agentMixer" ref="randomRobinAgentMixer"/>
        <property name="simulationController" ref="simulationController"/>
    </bean>

    <bean id="randomRobinAgentMixer"
  class="net.sourceforge.jabm.mixing.RandomRobinAgentMixer">
        <property name="prng" ref="prng"/>
    </bean>

    <bean id="population" scope="prototype"
          class="net.sourceforge.jabm.Population">
        <property name="agentList">
            <bean class="net.sourceforge.jabm.agent.AgentList">
                <constructor-arg>
                    <list>
                        <ref bean="requesterAgentList1"/>
                        <ref bean="workerAgentList1"/>
                    </list>
                </constructor-arg>
            </bean>
        </property>
        <property name="prng" ref="prng"/>
    </bean>

    <bean id="requesterAgentList1" class="net.sourceforge.jabm.agent.AgentList">
        <constructor-arg ref="nrOfRequestersTyp1"/>
        <constructor-arg ref="requesterAgentFactory1"/>
    </bean>

    <bean id="requesterAgentFactory1"
          class="org.springframework.beans.factory.config.
      ObjectFactoryCreatingFactoryBean">
        <property name="targetBeanName">
            <idref local="simpleRequesterAgent1"/>
        </property>
    </bean>

    <bean id="abstractRequester" abstract="true" scope="prototype"
  class="at.ac.tuwien.dsg.crowdsim.agent.requester.AbstractRequester">
        <property name="marketFacade" ref="marketFacade"/>
        <property name="scheduler" ref="simulationController"/>
        <property name="valuationPolicy" ref="marketOrientedValuationPolicyRQ"/>
        <property name="supplyPolicy" ref="simpleTaskSupplyPolicy"/>
        <property name="ratingPolicy" ref="simpleRatingPolicy"/>
        <property name="taskFactory" ref="simpleTaskFactory" />
    </bean>

    <bean id="simpleRequesterAgent1" scope="prototype" parent="abstractRequester"
          class="at.ac.tuwien.dsg.crowdsim.agent.requester.SimpleRequester">
        <property name="tradingStrategy" ref="beatTheQouteRequesterTS"/>
    </bean>

    <bean id="marketOrientedValuationPolicyRQ"
  class="at.ac.tuwien.dsg.crowdsim.agent.pricing.MarketOrientedValuationRQ">
        <property name="randomData" ref="randomData"/>
    </bean>

    <bean id="simpleTaskSupplyPolicy" scope="prototype"
          class="at.ac.tuwien.dsg.crowdsim.agent.policy.SimpleTaskSupply">
    </bean>

    <bean id="beatTheQouteRequesterTS" scope="prototype"
  parent="strategy"
  class="at.ac.tuwien.dsg.crowdsim.agent.strategy.trading.requester.
  BeatTheQuoteRequesterTS">
         <property name="randomData" ref="randomData"/>
         <property name="maxBetPercentageOverPrivateValue" value="0"/>
        <property name="maxQuoteIncrease" value="0.05"/>
    </bean>

  <bean id="simpleRatingPolicy"
  class="at.ac.tuwien.dsg.crowdsim.agent.policy.SimpleRatingPolicy"/>

    <bean id="simpleTaskFactory"
  class="at.ac.tuwien.dsg.crowdsim.agent.task.factory.SimpleTaskFactory">
        <property name="randomData" ref="randomData"/>
    </bean>

    <bean id="workerAgentList1" class="net.sourceforge.jabm.agent.AgentList">
        <constructor-arg ref="nrOfWorkersTyp1"/>
        <constructor-arg ref="workerAgentFactory1"/>
    </bean>
```

```

    <bean id="workerAgentFactory1"
            class="org.springframework.beans.factory.config.
      ObjectFactoryCreatingFactoryBean">
        <property name="targetBeanName">
            <idref local="workerAgent1"/>
        </property>
    </bean>

    <bean id="abstractWorker" scope="prototype" abstract="true"
            class="at.ac.tuwien.dsg.crowdsim.agent.worker.AbstractWorker">
        <property name="marketFacade" ref="marketFacade"/>
        <property name="scheduler" ref="simulationController"/>
        <property name="valuationPolicy" ref="marketOrientedValuation"/>
        <property name="randomData" ref="randomData"/>
        <property name="transactionBook" ref="simpleTransactionBook"/>
    </bean>

    <bean id="workerAgentList1" class="net.sourceforge.jabm.agent.AgentList">
        <constructor-arg ref="nrOfWorkersTyp1"/>
        <constructor-arg ref="workerAgentFactory1"/>
    </bean>

    <bean id="workerAgentFactory1"
            class="org.springframework.beans.factory.config.
      ObjectFactoryCreatingFactoryBean">
        <property name="targetBeanName">
            <idref local="workerAgent1"/>
        </property>
    </bean>

    <bean id="workerAgent1" scope="prototype" parent="abstractWorker"
            class="at.ac.tuwien.dsg.crowdsim.agent.worker.SimpleWorker">
        <property name="tradingStrategy" ref="loadBasedBeatTheQuoteTradingStrategy"/>
        <property name="skillConfig" ref="experiencedWorkerConfig"/>
    </bean>

    <bean id="experiencedWorkerConfig" class="at.ac.tuwien.dsg.crowdsim.agent.worker.
  SkillConfig">
        <property name="meanPerformance" value="0.70"/>
        <property name="deviancePerformance" value="0.25"/>
        <property name="meanConfidence" value="0.8"/>
        <property name="devianceConfidence" value="0.25"/>
    </bean>

    <bean id="simpleTransactionBook" scope="prototype"
            class="at.ac.tuwien.dsg.crowdsim.agent.worker.transaction.
      SimpleTransactionBook">
        <property name="randomData" ref="randomData"/>
    </bean>

    <bean id="marketOrientedValuation"
    class="at.ac.tuwien.dsg.crowdsim.agent.policy.MarketPriceValuationPolicy"
          scope="prototype">
        <property name="randomData" ref="randomData"/>
    </bean>

    <bean id="market"
          class="at.ac.tuwien.dsg.crowdsim.market.Market">
        <property name="marketFacade" ref="marketFacade"/>
        <property name="auctionFactory" ref="auctionFactoryBean"/>
        <property name="skillUpdatePolicy" ref="weightedSkillUpdate"/>
        <property name="auctionQualificationPolicy"
          ref="minPerformanceQualification"/>
    </bean>

    <bean id="minPerformanceQualification"
          class="at.ac.tuwien.dsg.crowdsim.agent.policy.
      MinPerformanceQualificationPolicy"/>

    <bean id="allQualificationPolicy"
          class="at.ac.tuwien.dsg.crowdsim.agent.policy.
      AllQualifyAuctionQualificationPolicy"/>

    <bean id="weightedSkillUpdate"
  class="at.ac.tuwien.dsg.crowdsim.agent.policy.WeightedSkillUpdatePolicy">
        <property name="OBS_PERF_UPDATE_RATE" value="0.2"/>
        <property name="CONF_UPDATE_RATE" value="0.1"/>
    </bean>

    <bean id="uniformSkillUpdate"
  class="at.ac.tuwien.dsg.crowdsim.agent.policy.UniformSkillUpdatePolicy">
        <property name="OBS_PERF_UPDATE_RATE" value="0.2"/>
        <property name="CONF_UPDATE_RATE" value="0.1"/>
    </bean>

    <bean id="auctionFactoryBean"
  class="at.ac.tuwien.dsg.crowdsim.market.auction.AuctionFactory">
        <property name="auctionPrototype" ref="sealedBid"/>
        <property name="orderBookProtoType" ref="lowestPriceOrderBook"/>
        <property name="auctionDuration" value="1"/>
        <property name="seed" ref="seed"/>
    </bean>

    <bean id="lowestPriceOrderBook" scope="prototype"
          class="at.ac.tuwien.dsg.crowdsim.market.auction.orderbook.
      LowestPriceOrderBook"/>

    <bean id="sealedBid" scope="prototype"
  class="at.ac.tuwien.dsg.crowdsim.market.auction.SealedBidSecondPriceAuction"/>


    <bean id="agentInitialiser"
          class="net.sourceforge.jabm.init.BasicAgentInitialiser">
    </bean>

    <bean id="prng" scope="singleton"
    class="cern.jet.random.engine.MersenneTwister64">
        <constructor-arg ref="seed"/>
    </bean>

    <bean id="randomData" scope="singleton"
          class="org.apache.commons.math.random.RandomDataImpl">
        <constructor-arg ref="randomGenerator"/>
    </bean>
    <bean id="randomGenerator" scope="prototype"
  class="org.apache.commons.math.random.MersenneTwister">
        <constructor-arg ref="seed"/>
    </bean>

    <bean id="seed" scope="singleton" class="java.lang.Long">
        <constructor-arg value="123"/>
    </bean>

    <bean id="strategy" scope="prototype"
          class="net.sourceforge.jabm.strategy.AbstractStrategy" abstract="true">
        <property name="scheduler" ref="simulationController"/>
        <property name="deadlineMultiplier" value="1.3"/>
    </bean>

    <bean id="loadBasedBeatTheQuoteTradingStrategy"
          class="at.ac.tuwien.dsg.crowdsim.agent.strategy.
      trading.worker.LoadBasedBeatTheQuoteTradingStrategy"
          parent="strategy">
        <property name="maxQuoteDecrease" value="0.05"/>
        <property name="maxMarginInPercent" value="0.5"/>
        <property name="randomData" ref="randomData"/>
        <property name="nrOfMaxOpenBids" value="100"/>
        <property name="maxLoadRate" value="1"/>
    </bean>

```

```
    <bean id="workerCSVReport" scope="singleton"
  class="at.ac.tuwien.dsg.crowdsim.report.CSVMasterReport">
        <property name="reportVariables">
            <bean class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.
      CSVReportVariables">
                <property name="numColumns" value="30"/>
                <property name="fileNamePrefix" value="data/sealedBid1_worker"/>
                <property name="reportVariables" ref="workerReportVariables"/>
            </bean>
        </property>
    </bean>

    <bean id="workerSimulationFinishedCSVReport" scope="singleton"
          class="net.sourceforge.jabm.report.SimulationFinishedReport">
        <property name="reportVariables">
            <bean class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.
      CSVReportVariables">
                <property name="numColumns" value="30"/>
                <property name="fileNamePrefix"
          value="data/sealedBid1_workerSimulationFinishedReport"/>
                <property name="reportVariables" ref="workerReportVariables"/>
            </bean>
        </property>
    </bean>

    <bean id="workerReportVariables"
  class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.WorkerReportVariable">
        <constructor-arg ref="marketFacade"/>
    </bean>

    <bean id="requesterSimulationFinishedCSVReport" scope="singleton"
          class="net.sourceforge.jabm.report.SimulationFinishedReport">
        <property name="reportVariables">
            <bean class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.
      CSVReportVariables">
                <property name="numColumns" value="13"/>
                <property name="fileNamePrefix"
        value="data/sealedBid1_requesterSimulationFinishedReport"/>
                <property name="reportVariables" ref="requesterReportVariables"/>
            </bean>
        </property>
    </bean>

    <bean id="requesterReportVariables"
          class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.
      RequesterReportVariables">
        <constructor-arg ref="marketFacade"/>
    </bean>

    <bean id="transactionCSVReport" scope="singleton"
  class="net.sourceforge.jabm.report.SimulationFinishedReport">
        <property name="reportVariables">
            <bean class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.
      CSVReportVariables">
                <property name="numColumns" value="33"/>
                <property name="fileNamePrefix"
        value="data/sealedBid1_transactionReport"/>
                <property name="reportVariables" ref="transactionReportVariables"/>
            </bean>
        </property>
    </bean>

    <bean id="transactionReportVariables" scope="singleton"
          class="at.ac.tuwien.dsg.crowdsim.report.taskReportVariables.
      TransactionReportVariables"/>
</beans>
```

Listing B.1: A sample configuration file of a simulation model

Bibliography

[1] 99designs. http://99designs.com/. last accessed: April 2012.

[2] Eugene Agichtein, Carlos Castillo, Debora Donato, Aristides Gionis, and Gilad Mishne. Finding high-quality content in social media. In *Proceedings of the international conference on Web search and web data mining*, pages 183–194. ACM, 2008.

[3] Amazon Mechanical Turk. www.mturk.com. last accessed: March 2012.

[4] Amazon Mechanical Turk FAQ. http://aws.amazon.com/mturk/. last accessed: March 2012.

[5] Nikolay Archak and Arun Sundararajan. Optimal design of crowdsourcing contests. In *ICIS 2009 Proceedings*, volume 200, pages 0–16, Phoenix, 2009.

[6] Brian W Arthur, Steven N Durlauf, and David A Lane. The economy as an evolving complex system II. In *SFI Studies in the Sciences of Complexity*. Addison-Wesley: Reading, 1997.

[7] Raul Bagni and Roberto Berchi. A comparison of simulation models applied to epidemics. *Journal of Artificial Societies and Social Simulation*, 5(3), 2002.

[8] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete event system simulation*. Prentice Hall, 1984.

[9] Kurt Binder. Monte-Carlo Methods. In George L Trigg, editor, *Mathematical Tools for Physicists*, pages 249–281. John Wiley & Sons, 2006.

[10] Eric Bonabeau. Agent-based modeling: methods and techniques for simulating human systems. *Proceedings of the National Academy of Sciences of the United States of America*, 99 Suppl 3:7280–7, May 2002.

[11] Daren C. Brabham. Crowdsourcing as a Model for Problem Solving: An Introduction and Cases. *Convergence: The International Journal of Research into New Media Technologies*, 14(1):75–90, February 2008.

[12] John L. Casti. *Would-be worlds: How simulation is changing the frontiers of science*. John Wiley & Sons, 1st edition, 1998.

[13] Henry W. Chesbrough. *Open innovation: The new imperative for creating and profiting from technology*. Harward Business School Press, 2003.

[14] A. P. Dawid and A. M. Skene. Maximum likelihood estimation of observer error-rates using the EM algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):20–28, 1979.

[15] Dominic DiPalantino and Milan Vojnovic. Crowdsourcing and all-pay auctions. In *Proceedings of the 10th ACM conference*, pages 119–128. ACM, 2009.

[16] Anhai Doan, Raghu Ramakrishnan, and Alon Y. Halevy. Crowdsourcing systems on the World-Wide Web. *Communications of the ACM*, 54(4):86, April 2011.

[17] Schahram Dustdar and Kamal Bhattacharya. The Social Compute Unit. *IEEE Internet Computing*, 15(3):64–69, May 2011.

[18] EBay. http://www.ebay.com/. last accessed: March 2012.

[19] Joshua M. Epstein. Agent-based computational models and generative social science. *Complexity*, 4(5):41–60, May 1999.

[20] Extensible Markup Language (XML). http://www.w3.org/XML/. last accessed: April 2012.

[21] Facebook. www.facebook.com. last accessed: March 2011.

[22] Flickr. http://www.flickr.com/. last accessed: April 2012.

[23] Martin Fowler. Inversion of control containers and the dependency injection pattern. *http://www.martinfowler.com/articles/injection.html (last accessed: April 2012)*, 2004.

[24] Daniel Friedman and John Rust. The double auction market institution: A survey. *The Double Auction Market: Institutions, Theories, and Evidence*, pages 3–25, 1993.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[26] Jose M. Garrido. *Object-oriented discrete-event simulation with java: A practical introduction*. Kluwer Academic Publishers, 2001.

[27] Google. https://www.google.at/. last accessed: April 2012.

[28] Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(14):1–5, 2006.

[29] Jeff Howe. *Crowdsourcing: How the power of the crowd is driving the future of business*. Century, 2008.

[30] InklingMarkets. http://inklingmarkets.com/. last accessed: April 2012.

[31] InnoCentive. http://www.innocentive.com/. last accessed: April 2012.

[32] Panagiotis G. Ipeirotis. Analyzing the Amazon Mechanical Turk marketplace. *XRDS:Crossroads, The ACM Magazine for Students, Forthcoming*, 17(2):16–21, 2010.

[33] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on Amazon Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation - HCOMP '10*, HCOMP '10, pages 64–67, New York, New York, USA, 2010. ACM Press.

[34] IStockPhoto. www.istockphoto.com/. last accessed: April 2012.

[35] JABM Framework. http://sourceforge.net/apps/phpwebsite/jabm/index.php. last accessed: March 2011.

[36] S. D. Jap. Online Reverse Auctions: Issues, Themes, and Prospects for the Future. *Journal of the Academy of Marketing Science*, 30(4):506–525, October 2002.

[37] Roman Khazankin, Harald Psaier, Daniel Schall, and Schahram Dustdar. QoS-based Task Scheduling in Crowdsourcing Environments. In G. Kappl, Z. Maamar, and H. Motahari Nezad, editors, *Service-Oriented Computing 9th International Conference, ICSOC 2011*, pages 297–311. Springer-Verlag Berlin Heidelberg, 2011.

[38] Paul Klemperer. Auction theory: A guide to the literature. *Journal of economic surveys*, 13(3):227–286, 1999.

[39] Vijay Krishna. *Auction theory*. Academic Press, 1st edition, 2002.

[40] Gioacchino La Vecchia and Antonio Cisternino. Collaborative workforce, business process crowdsourcing as an alternative of BPO. In *ICWE'10 Proceedings of the 10th international conference on Current trends in web engineering*, pages 425–430. Springer-Verlag Berlin Heidelberg, 2010.

[41] Sean Luke, Claudio Cioffi-Revilla, Liviu Panait, Keith Sullivan, and Gabriel Balan. MASON: A Multiagent Simulation Environment. *SIMULATION*, 81(7):517–527, July 2005.

[42] Charles M Macal and Michael J. North. Tutorial on agent-based modeling and simulation. In *Proceedings of the 37th conference on Winter simulation, WSC '05*, pages 2–15. Winter Simulation Conference, September 2005.

[43] MASON - Multi Agent Simulator. http://cs.gmu.edu/~eclab/projects/mason/. last accessed: April 2012.

[44] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[45] MySpace. www.myspace.com. last accessed: March 2011.

[46] Netflix Price. http://www.netflixprize.com/. last accessed: April 2012.

[47] NetLogo. http://ccl.northwestern.edu/netlogo/. last accessed: April 2012.

[48] Object Management Group. Unifed Modeling Language (UML). http://www.uml.org/. last accessed: March 2012.

[49] One Billion Minds. http://www.onebillionminds.com/. last accessed: April 2012.

[50] Tim O'Reilly and John Battelle. Web squared: Web 2.0 five years on. *Web 2.0 Summit*, 2009.

[51] R Parsons, MacKenzie J, and Fowler M. Plain Old Java Object. URL http://martinfowler.com/bliki/POJO.html. last accessed: April 2012.

[52] Steve Phelps. Applying dependency injection to agent-based modeling: the JABM toolkit. *ACM Transactions on Computer Simulation and Modeling*, pages 1–20, 2011.

[53] Suzanne Robertson and James Robertson. *Mastering the requirements process*. Addison-Wesley, 2nd edition, 2006.

[54] Benjamin Satzger, Harald Psaier, Daniel Schall, and Schahram Dustdar. Stimulating skill evolution in market-based crowdsourcing. In S Rinderle-Ma, F Toumani, and K Wolf, editors, *Business Process Management Proceedings of the 9th International Conference, BPM 2011*, pages 66–82, Clermont-Ferrand, France, 2011. Springer.

[55] Daniel Schall, Schahram Dustdar, and M. Brian Blake. Programming Human and Software-Based Web Services. *Computer*, 43(7):82–85, July 2010.

[56] Daniel Schall, Hong-Linh Truong, and Schahram Dustdar. Unifying Human and Software Services in Web-Scale Collaborations. *IEEE Internet Computing*, 12(3):62–68, May 2008.

[57] Florian Skopik, Daniel Schall, Harald Psaier, Martin Treiber, and Schahram Dustdar. Towards Social Crowd Environments Using Service-Oriented Architectures. *it - Information Technology*, 53(3):108–116, May 2011.

[58] James Surowiecki. *The Wisdom of Crowds*. Anchor, 2005.

[59] The Repast Framework. http://repast.sourceforge.net/. last accessed: March 2012.

[60] The Spring Framework. http://www.springsource.org. last accessed: April 2012.

[61] Threadless. http://www.threadless.com. last accessed: March 2012.

[62] Seth Tisue and Uri Wilensky. NetLogo : A Simple Environment for Modeling Complexity. In *International Conference on Complex Systems*, pages 1–10, 2004.

[63] Top Coder. www.topcoder.com/. last accessed: April 2012.

[64] Twitter. www.twitter.com. last accessed: March 2011.

[65] Luis von Ahn and Laura Dabbish. Labeling images with a computer game. In *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*, pages 319–326, New York, New York, USA, 2004. ACM Press.

[66] Luis von Ahn, Benjamin Maurer, Colin McMillen, David Abraham, and Manuel Blum. reCAPTCHA: human-based character recognition via Web security measures. *Science (New York, N.Y.)*, 321(5895):1465–8, September 2008.

[67] Maja Vukovic. Crowdsourcing for Enterprises. In *2009 Congress on Services - I*, pages 686–692. IEEE, July 2009.

[68] WeAreHunted. http://wearehunted.com/. last accessed: April 2012.

[69] Wikipedia. http://www.wikipedia.org/. last accessed: March 2011.

[70] Wikipedia - Auctions. http://en.wikipedia.org/wiki/Auction. last accessed: April 2012.

[71] Wikipedia Definition of Reverse Auctions. http://en.wikipedia.org/wiki/Reverse_auction. last accessed: March 2012.

[72] Wikipedia Observer Pattern. http://en.wikipedia.org/wiki/File:Observer.svg. last accessed: March 2012.

[73] Peter R Wurman, William E Walsh, and Michael P Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems*, 24(1):17–27, 1998.

[74] Yahoo! Answers. http://answers.yahoo.com/. last accessed: March 2012.

[75] YouTube. http://www.youtube.com/. last accessed: April 2012.

[76] Man-Ching Yuen, Irwin King, and Kwong-Sak Leung. Task Matching in Crowdsourcing. In *2011 International Conference on Internet of Things and 4th International Conference on Cyber, Physical and Social Computing*, pages 409–412. IEEE, October 2011.