Creative software crowdsourcing: from components and algorithm development to project concept formations

Wenjun Wu

State Key Laboratory of Software Development Environment,
Beihang University,
Beijing, 100191, China
E-mail: wwj@nlsde.buaa.edu.cn

Wei-Tek Tsai*

School of Computing, Informatics, and Decision Systems Engineering,
Arizona State University,
Tempe, AZ 85281, USA
and
Department of Computer Science and Technology,
INLIST, Tsinghua University,
Beijing, 100084, China
Fax: (480) 965-2751
E-mail: wtsai36@gmail.com
*Corresponding author

Wei Li

State Key Laboratory of Software Development Environment,
Beihang University,
Beijing, 100191, China
E-mail: liwei@nlsde.buaa.edu.cn

Abstract: Software development is complex and creative as it involves requirement analysis, design, architecture, coding and testing. Recently, software crowdsourcing has been popular with numerous software coders participated in various software competitions. This paper first analyses the data collected on software crowdsourcing and summarises major lessons learned. This paper then examines two software crowdsourcing processes including TopCoder and AppStori processes. Lastly, this paper identifies the min-max nature among participants as an important design element in software crowdsourcing for software quality and creativity. Although in a min-max game, one party tries to maximise the finding of bugs in a set of artefacts, and the other parties try to minimise the potential bugs in the same artefact, software crowdsourcing can still be a collaborative and win-win process for all parties. By using this approach, lots of aspects of software development can be crowdsourced with the crowd can contribute their creativity to each aspect.

Keywords: crowdsourcing; software development; creativity; TopCoder; AppStori; game theory.

Reference to this paper should be made as follows: Wu, W., Tsai, W-T. and Li, W. (2013) 'Creative software crowdsourcing: from components and algorithm development to project concept formations', *Int. J. Creative Computing*, Vol. 1, No. 1, pp.57–91.

Biographical notes: Wenjun Wu is a Professor in the School of Computer Science and Engineering at the Beihang University. He was previously a Research Scientist from 2006 to 2010, at the Computation Institute (CI) at the University of Chicago and Argonne National Laboratory. He was a Technical Staff and Postdoctoral Research Associate from 2002 to 2006, at the Community Grids Lab at the Indiana University. He received his BS, Master and PhD in Computer Science from Beihang University in 1994, 1997 and 2001, respectively. He has published over 50 peer-review papers on journals and conferences. His research interests include: crowdsourcing, green computing, cloud computing, e-science and cyber infrastructure, and multimedia collaboration.

Wei-Tek Tsai is currently a Professor in the School of Computing, Informatics, and Decision Systems Engineering at Arizona State University, USA. He received his PhD and MS in Computer Science from University of California at Berkeley, and SB in Computer Science and Engineering from MIT, Cambridge. He has produced over 300 papers in various journals and conferences, two best paper awards, and was awarded several guest professorships. His work has been supported by US Department of Defense, Department of Education, National Science Foundation, EU, and industrial companies such as Intel, Fujitsu, and Guidant. In the last ten years, he focused his energy on service-oriented computing and SaaS, and worked on various aspects of software engineering including requirements, architecture, testing, and maintenance.

Wei Li is a member of Chinese Science Academy. He received his PhD in Computer Science from University of Edinburgh and BS in Mathematics from Peking University. He is the Director of State Key Lab of Software Environment Development and Vice-chair of Chinese Institute of Electronic. He was President of Beihang University from 2002 to 2009. Currently, he is serving as the Editor-in-Chief for *Science China – Information Sciences*, Editor for *Journal of Computer Science and Technology* and *International Journal of Advanced Software Technology*, Science in China Publisher. He has published over 100 papers and one book.

1 Introduction

Crowdsourcing has captured the attention of the world recently (Doan et al., 2011). Numerous tasks or designs conventionally carried out by professionals are now being crowdsourced to the general public who may not know each other to perform in a collaborative manner. Specifically, crowdsourcing has been used for identifying chemical structure, designing mining infrastructure, estimating mining resources, medical drug development, logo design, and even software design and development.

The proliferation of crowdsourcing practices indicates a new peer-production paradigm, where large numbers of regular end-users are empowered as co-creators or co-designers, and their creative energy is coordinated to participant in large projects without a traditional organisation. Such a new model of socio-economic production is called *commons-based peer production*, a term coined by Benkler (2012). Several well-known software systems and services have been produced and available for public access on a daily basis using this approach. One of the most popular website is Wikipedia where hundreds of thousands of authors participated in creating an online encyclopaedia without a central organisation managing all the contents. But is this approach suitable for software crowdsourcing?

Software development is considered one of the most challenging and creative activities. As one software problem is solved by a new solution, another new software problem is subsequently created by the solution. Thus, the software engineering history has a long list of techniques, processes, and tools in the last 50 years, yet, the field is still seeking for new solutions and new technologies each year as it encountered new problems. The term 'software engineering' was invented to address the importance of the engineering aspects of software development where many traditional engineering techniques such as modelling, simulation, prototyping, testing and inspection are used in software development. Furthermore, many new techniques such as model checking, automated code generation, design techniques, have been developed for software.

Many authors have argued that crowdsourcing encourages creativity and problem solving (Kittur, 2010), but software crowdsourcing has many unique features and issues different from general crowdsourcing. Specifically, software crowdsourcing need to support

- The rigorous engineering discipline of software development, such as rigid syntax and semantics of programming languages, modelling languages, and documentation or process standards such as UML, CMMI (2012), and 2167A (DoD, 2012).
- The creativity aspects of software requirement analysis, design, testing, and evolution. The issue is to stimulate creativity in these software development tasks through collective intelligence?
- The big data aspects as numerous data will be generated and need to be analysed for ranking and evaluation of both clients and developers including their products.
- The psychology issues of crowdsourcing such as competition, incentive, recognition, open, sharing, collaboration, and learning. The psychology must be competitive while at the same time friendly, socialable, learning, and personal fulfilment for participants, requesters and administrators.
- The financial aspects of all parties including requesters, crowdsourcing platforms, and participants.
- Quality aspects including objective qualities such as functional correctness, performance, security, reliability, maintainability, safety, and subjective qualities such as usability.
- The liability issues in case of failure of software that caused the harm. For example, who is responsible for the software faults? Developers, administrators, requesters, or users?
- Reputation of all parties including requesters, administrators, and participants.

Thus, software crowdsourcing is different from general crowdsourcing. Furthermore, most code developed by the crowd carries no liability in case of damage that may be incurred by running the application. A mobile app is a typical example. It can be downloaded and provided useful services for clients, but the app carries no liability. For the software that has liability issues, the organisation that crowdsourced the software will perform additional validation, manual or automated (Li and Li, 2012), before the software can be used.

This paper examines two software crowdsourcing development processes, and identifies key issues in these processes. A key feature of software crowdsourcing is that competitions are held to select the best software and to identify the best software coder. While competitions promote creativity and support quality software development, but stiff competitions may also restrict massive participation. Specifically, only limited number of people can play the Olympic game as only the best can participate while leaving the massive to passively watch. Thus, stiff software competitions may restrict the activities of the crowd. Another key feature of software crowdsourcing process is the min-max nature of game playing by different people in different roles. This kind of min-max phenomenon has observed often in general crowdsourcing (Tibbetts, 2012), and this is the first time this kind of min-max nature is identified in software crowdsourcing.

This paper is organised as follows: Section 2 reviews the general crowdsourcing work; Section 3 provides some software crowdsourcing sites and original data collected from the TopCoder website; Section 4 analyses the TopCoder process including the competition rules, compares it with the IBM Cleanroom methodology known to produce zero-defect software, identifies the min-max nature in collaborative software development, and produces an updated process with more participants to contribute to software development; Section 5 analyses the AppStori process with respect to the min-max nature of collaborative competitions; Section 6 provides game theory interpretation of games used in software crowdsourcing; Section 7 concludes this paper.

2 Related work

Many researchers have analysed the economics of crowdsourcing contests. Archak and Sundarrarajan (2009) used game theory in analysing crowdsourcing contests particularly related to optimal price structure for designing contests. Then Archak (2010) extended the approach to studying the impact of TopCoder's reputation system on the TopCoder community members, and analysed the principal factors such as project payment and requirements on the quality of the outcome in the competition. The author presents an in-depth analysis of the reputation system and the registration strategy utilised by contestants. Similarly, other researchers (DiPalantino and Vojnović, 2009; Horton and Chilton, 2010) attempted to model crowdsourcing as business auction and leverage the research of auction theory to build models for reward system and effective strategies for crowdsourcing participants. DiPalantino proposed an all-auction model to describe the contest process of crowdsourcing, and capture the essential relationship between rewards and participation. All these efforts have different focus from this paper. Their goal is to study the mechanism of crowdsourcing systems, such as pricing and bidding strategies as well as rewarding rules. Their discussions are mostly in the scope of classic auction methodology and are not related to issues in software development, i.e., maximising the software quality and creativity via crowdsourcing.

Bacon et al. (2009) introduced a new paradigm of software evaluation through a market-driven mechanism that presents rewards for developers, testers, and bug reporters to bidding for the tasks of bug fixing. The author also defined the notion of 'sufficient correctness' in the context of crowdsourcing and designed the components for the market design.

Bullinger and Moeslein (2010) listed several factors in designing crowdsourcing contests such as media (online, offline, or mixed), organiser (companies, government agencies, non-profit, and individuals), participants (individuals, teams, or both), contest periods (from short term to long term), reward/motivation (monetary rewards, reputation rewards, or both), evaluation (such as jury evaluation, peer evaluation, self evaluation), types of deliverables (such as concepts, prototypes, and solutions). They also suggested several research topics in crowdsourcing.

Leimeister et al. (2009) proposed the concept of 'activation-enabling' as the basis for using competitions in software crowdsourcing and open innovation, and presented many factors related to IT software crowdsourcing. For example, motivation for participation can be learning, self-marketing, social motives, and direct compensation.

Relationship between creativity and software development has been extensively discussed in Resnick et al. (2005), Obrenovic et al. (2008) and Shneiderman (2007). They summarised a dozen principles for supporting creativity in design tools, such as embracing exploration, collaboration, low threshold, high ceiling, and wide walls as well as iterative thinking processes. Opportunistic software development framework (Obrenovic et al., 2008) was proposed to encourage professional developers and students in computer science to develop innovative ideas and solutions by thinking out-of-box and exploring unconventional combination of different technologies. The focus of these efforts is on individual or group level creativity not in crowdsourcing.

Recent papers on social-level creativity show the new insight into the dynamic of competition and collaboration in the crowdsourcing process. Fischer (2004) and Shneiderman (2007) discussed the social nature of creativity and the necessity of social-technical environments to sustain social creativity. The authors believe that the distribution among location, time, and background from contributing individuals can enhance creativity by enabling people to be aware of others' work and learn from each other. Hutter et al. (2011) and Shneiderman (2011) studied the synergy between competition and collaboration in crowdsourcing based design contests. The authors suggested to introduce the firm-level concept co-opetition in the research of crowdsourcing and highlighted that the vital factor in the creativity of the contests is the balance between competition and cooperation. Competitive participation should be employed to stimulate crowd's motivation of making contributions and performance without disabling the climate for knowledge sharing and collaboration. Most of the work today is related to crowdsourcing in general, and those specifically related to software crowdsourcing will be discussed in the next section. Specifically, this paper analyses two software crowdsourcing development processes, and identifies the unique nature of these processes such as collaborative and competitive games played by people of different roles in these processes with the end goal of either identify top coders and/or delivering quality software.

3 Software crowdsourcing websites, data, and major lessons learned

Software development is distinct from other human creative activates as it is highly iterative, and traditional software development processes often emphasise on delivery of intermediate documents such as requirement documents, design documents, test plans, test case documents. Furthermore, software development processes have evolved from the traditional Waterfall model, Spiral model, and model-driven process to recent Agile methods (Agile Software Development, 2012), component-based methods, open-source approach, and service-oriented computing (Chen and Tsai, 2010). These processes differ significantly with respect to the steps used as well as the intermediate deliverables made. For example, the Waterfall model requires significant documentation efforts during the process, and each document is cross validated with other documents during the process. But those modern Agile processes are light on specifications, but heavy on code development.

Current practices of software crowdsourcing suggest the feasibility of every phase in software development:

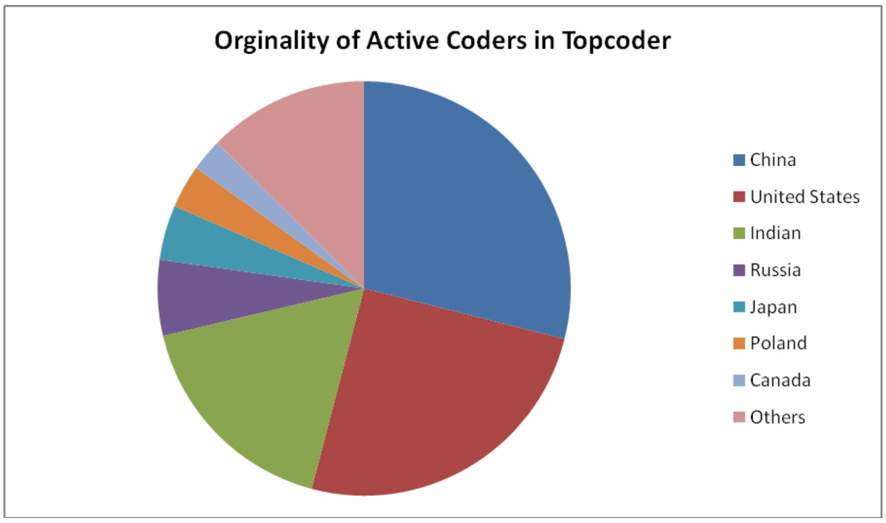
- Source code can be crowdsourced and this is evidenced by TopCoder.com and AppStori.com.
- Testing (and code review) can be crowdsourced and this is evidenced by uTest.com, mob4hire.com, and TopCoder.com.
- Security testing can be crowdsourced, and this is evidence by uTest.com (Fink et al., 2011).
- Requirement specification can be crowdsourced and this can be evidenced by TopCoder.com and AppStori.com.
- Architecture can be crowdsourced, and this can be evidenced by TopCoder.com.
- Usability and performance evaluation can be crowdsourced and this can be evidenced by AppStori.com, TopCoder.com, uTest.com, and Mob4hire.com
- Algorithms can be crowdsourced and this is evidenced by TopCoder.com.

3.1 Popular software crowdsourcing sites and their applications

In fact, software crowdsourcing has been popular with numerous software coders participated in software crowdsourcing. Specifically, TopCoder reported that it has 450,000 of software coder registered at their site with 50,000 of active coders from 204 countries. Figure 1 shows the originality of the active coders. Given such a global workforce, some large research institutes are taking advantage of the TopCoder platform for their software crowdsourcing projects. For instance, NASA, teamed up with Harvard University, has established NASA Tournament Lab (NTL) (2010), to encourage competitions among talented young programmers for the most creative algorithms needed by NASA researchers.

While software crowdsourcing has been popular, the planned applications of software crowdsourcing are still growing at a rapid speed. Specifically, Harvard University and TopCoder will provide an environment where MBA students will develop new online software (Harvard University, 2012), from concept generation to final product deployment. Harvard also puts this practice as a part of its Business School curriculum Field Immersion Experiences for Leadership Development (FIELD) as a new trend of IT software development.

Figure 1 TopCoder has 48,850 active contestants from 204 countries (see online version for colours)



Another example is that US DARPA initiated a software crowdsourcing platform to attract middle and high-school students to perform software development ('DARPA and TopCoder also seek a platform middle and high school students on software development in October 2011', 2011) in October 2011. As many high-school students find computer science a difficult subject to learn, but the competition nature of software crowdsourcing may be interesting enough to attract youth to computing.

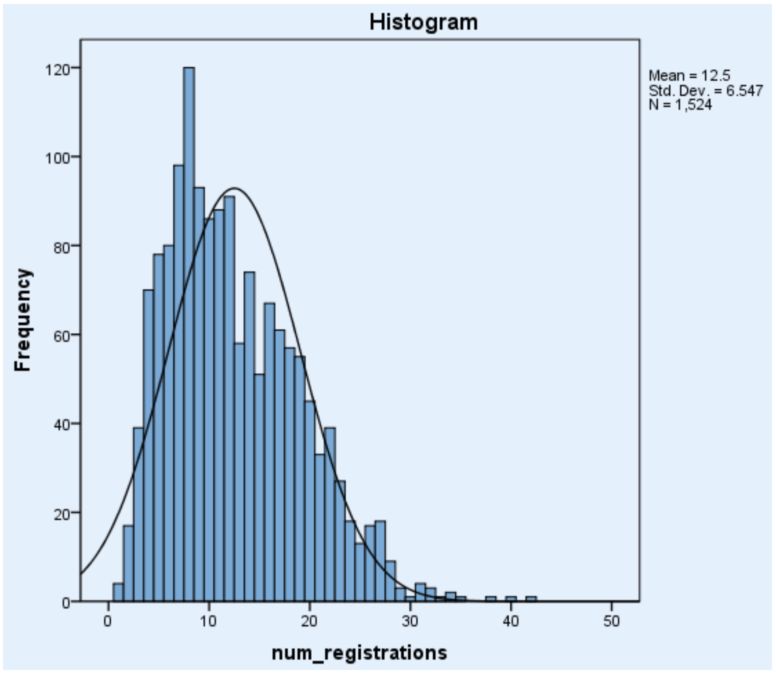
3.2 Interesting software crowdsourcing data

3.2.1 Number of people participating is smaller than expected

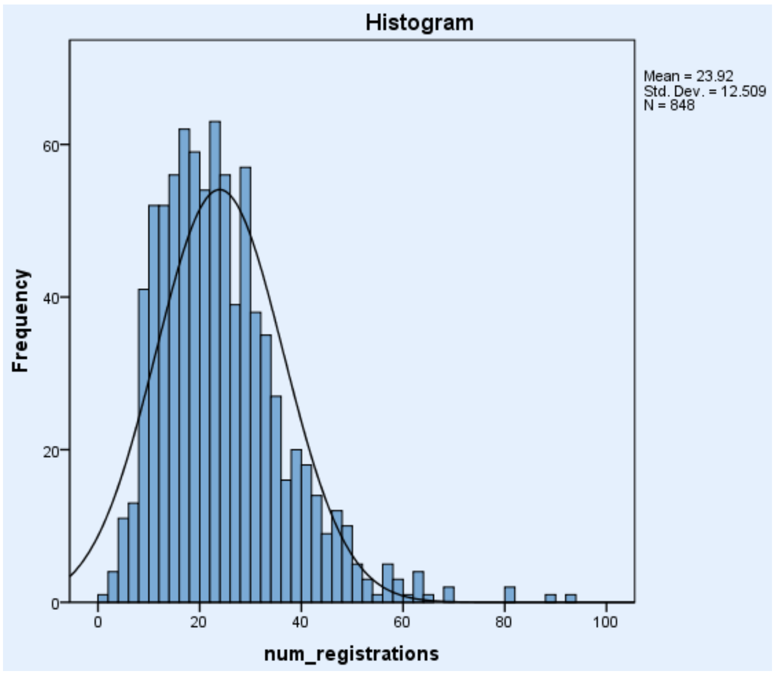
One of the premise of crowdsourcing is that a large number of people will be attracted to work on problems posted on the web, and this is consistent with TopCoder data where it has shown that many people from a large number of countries (Figure 1 showed) are willing to participate in TopCoder competitions.

However, close examination of data tells a different story. According to Tibbetts (2012), for a common TopCoder competition, the optimal number of submissions turns out to be just *two*, and this is surprisingly low comparing to the number of active coders that are available. Figures 2 to 3 show that *the average registration number for each TopCoder design and development competition is about 13 and 25 respectively. And out of these registrations, only two design submissions and five development submissions can be generated*. This is a far cry from a large number of software coders available. One possible reason is that as often only the top two competitors will win the price, once a potential participant sees that two strong contenders already signed up for a competition, the participant will opt out of the competition.

Figure 2 (a) Average registration numbers for design and (b) development tasks in TopCoder (see online version for colours)

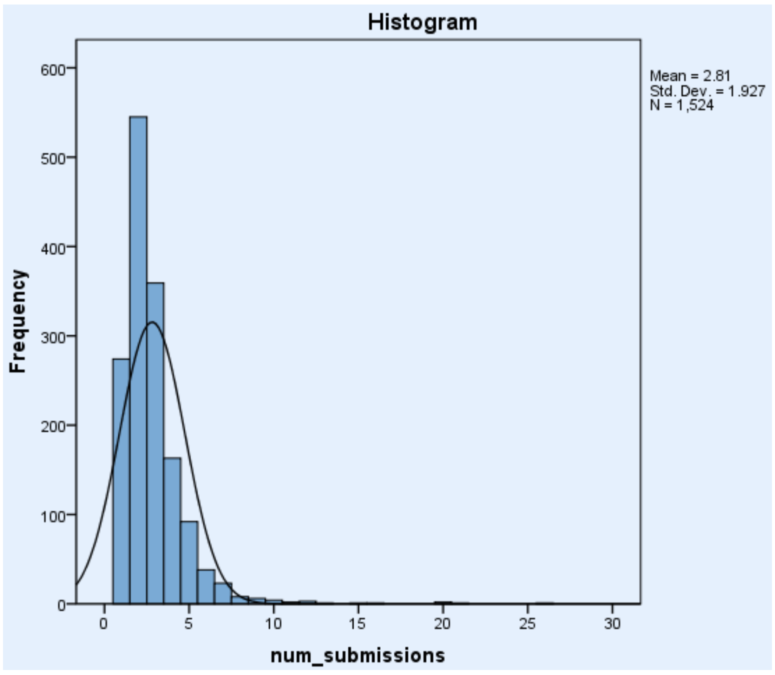


(a)

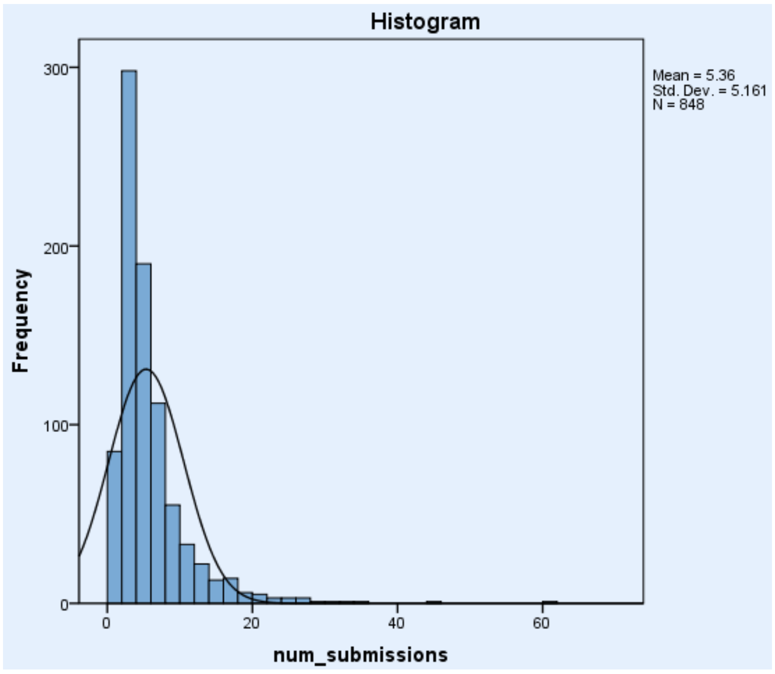


(b)

Figure 3 (a) Average submission numbers for design and (b) development tasks in TopCoder (see online version for colours)



(a)



(b)

3.2.2 Number of hours spent on competition is smaller than anticipated

Most of the competition was done within two or three weeks. If each competition lasted for such short period of time, the sophistication of software constructed cannot be significant.

3.3 Lessons learned in current software crowdsourcing

Many reports are now available for major lessons learned from software crowdsourcing, and as software crowdsourcing is another form of software engineering, some of major lessons learned have been shown to be consistent with traditional software engineering.

1. Problem definition must be clear: This is reported as the most important factor in software crowdsourcing, just like in traditional software engineering, clear requirement definition is the most critical factor. If the problem is not well defined, regardless whether the traditional software development process or crowdsourcing development process is used, requirement bugs will be incorporated into the design and the code. This may mean
   - Thus, problems must be well written in natural languages (such as English), formal or semiformal modelling languages such as UML.
   - The overall architecture of the application should be shown to all contestants so that they know the role of specific components to be developed during the competition.
   - The tools that must be used should be explicitly stated.
   - The tasks should be well specified and decomposed into components to be used in competitions.
2. Transparency is critical. It is important that potential participants know who they will be competing with before deciding to join a competition, and they must be able to understand the problem, and know exactly how their solutions will be evaluated in an objective manner. Also, the participants need to understand that they retain the ownership of their work unless they release the ownership. This transparency can be reflected in the following ways:
   - All questions posed are answered in a timely manner and the all the responses will be shown to every contestant.
   - The ranking of contestants of the current competitions will be known to all the contestants.
   - The reputation of the client should be informed to all contestants and potential participants to know the value of the current competition.
   - The evaluation criteria and processes including evaluation of products and evaluation of contestants must be objective and open. The contestants should have the right to appeal the evaluation results in an open and objective manner. For example, Apple App Store published a comprehensive list of review criteria (Apple App Store Review Guidelines, 2010) in 2010, and thus all participants know clearly how the evaluation will be performed.

3 Diversity is important for creativity in a community.

- Crowdsourcing needs a diverse community of crowd workers in different locations with different background to ensure a diversity of opinions are expressed to encourage creativity, and to avoid any biased of individual leaderships within a specific community. This reflects *one of creativity principles 'low threshold, high ceiling and wide walls'. Crowdsourcing often sets up a 'low threshold' to encourage more people to take a variety of software development tasks to broaden the community.*
- Software crowdsourcing also adopts 'high ceiling' strategy to screen qualified programmers for complex and challenging projects. Moreover, it also enables access to traditional IT personnel to propose, interact, and evaluate the submissions from the community workers. Thus, at the current stage, it is not reasonable to completely depend on the community workers to produce the quality products needed.
- Idea exploration is the important aspect in the processing of software crowdsourcing. 'Wide walls' in software projects suggest that designers can propose a wide range of concepts and models for a public solicitation.

- Software crowdsourcing also adopts 'high ceiling' strategy to screen qualified programmers for complex and challenging projects. Moreover, it also enables access to traditional IT personnel to propose, interact, and evaluate the submissions from the community workers. Thus, at the current stage, it is not reasonable to completely depend on the community workers to produce the quality products needed.
- Idea exploration is the important aspect in the processing of software crowdsourcing. 'Wide walls' in software projects suggest that designers can propose a wide range of concepts and models for a public solicitation.

As one can easily see that while software crowdsourcing has been popular and even Harvard University teaches future business managers to develop software completely using the software crowdsourcing on the TopCoder platform, but software crowdsourcing is at an early stage of development.

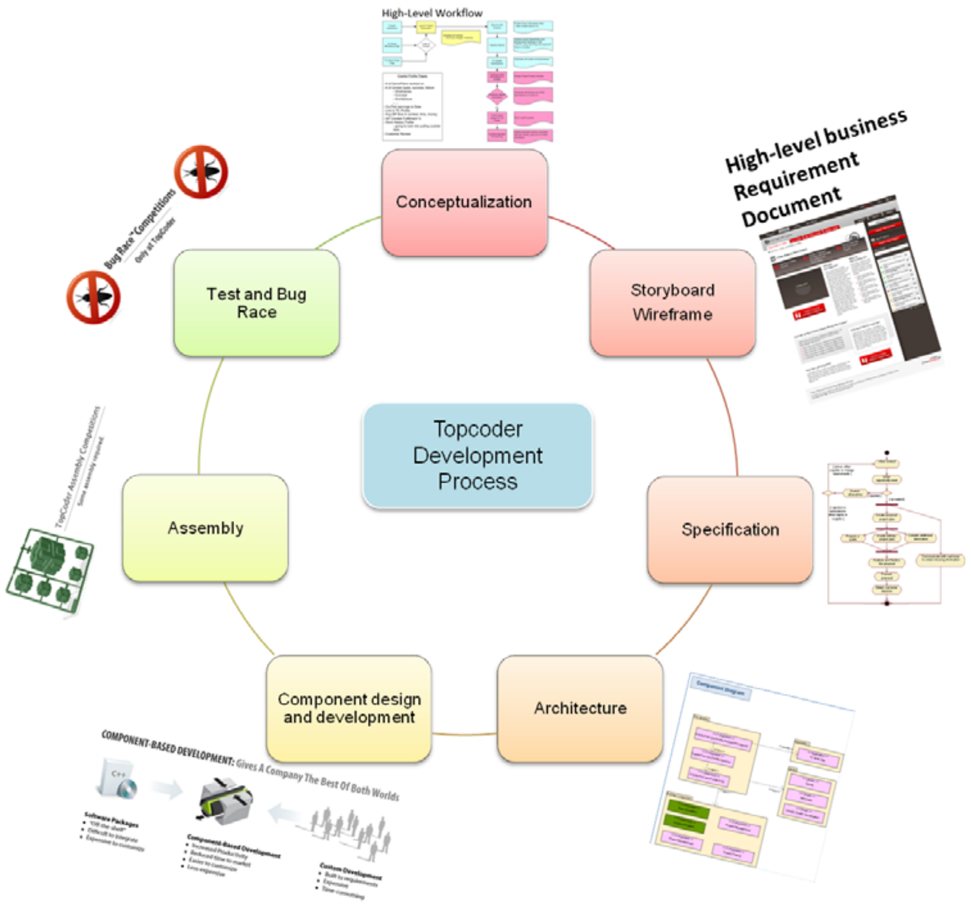
This paper evaluates two well-known software crowdsourcing processes: the TopCoder process and the AppStori process. Both processes show distinct approaches to software crowdsourcing, one follows a rigorous engineering process with strong documentation requirements, and the other follows more like an Agile process of software development.

The goal is to identify factors that may extend from the current crowdsourcing projects with single person versus teams of multiple people ranging from the small scale of 2 to 5 to the medium scale of 100 to 200, even to the large scale of 1,000 to 2,000, from mainly components to large and complex systems, from developing well-defined modules to ill-defined complex and even evolving systems, and from individual applications to a cloud-based system with lots of infrastructure.

4 TopCoder development processes

Section 4.1 analyses the overall development process including activities performed at each step, and deliverables produced during the process; Section 4.2 examine the competition rules used in software crowdsourcing and their implications to activities that will be performed, and quality of software produced; Section 4.3 identifies the people involved, the time they are involved, and the kind of activities involved including evaluation; and Section 4.4 compares the TopCoder processes with a traditional software development process, IBM Cleanroom methodology, that may lead to a potential update in the current software crowdsourcing process.

Figure 4 TopCoder development process (see online version for colours)


High-Level Workflow
Conceptualization
High-level business Requirement Document
Storyboard Wireframe
Specification
Architecture
Component design and development
COMPONENT-BASED DEVELOPMENT: Gives A Company The Best Of Both Worlds
Assembly
TopCoder Assembly Competitions
Test and Bug Race
Bug Race Competitions
Only at TopCoder
Topcoder Development Process

4.1 TopCoder processes

TopCoder has the following overall process as shown in Figure 4:

1 *Conceptualisation*: This defines the initial requirement analysis including features and use cases. The inputs to this step are goals, high-level workflow, and questionnaire, and the outputs are high-level business requirements captured in text documents with use-case diagrams and workflow diagrams. After the high-level requirement for a project is described, the next step called 'studio ideation', where brainstorming tools such as studio storyboard and wireframe are utilised to quickly create new product ideas and express them in form of mockup. In studio ideation, contestants can engage in two types of competitions: wireframe and storyboard.

2 *Wireframes*: This step defines the screen blueprint and informational process used in the application. The input to this step is the conceptualisation document developed in the first step, and the output results are the wireframes. During wireframe competitions, contestants define web page flows, user inputs, and content of each page of an application, without regard to the specific UI styles of the application.

3 *Storyboards*: This step develops a storyboard that consists of GUI or the high-level application view. The inputs to this step are wireframes and conceptualisation document, and the outputs are storyboards.

4 *UI prototype*: This step develops a GUI interface of the application based on HTML based on the storyboards developed earlier.

The above four steps are designed to stimulate crowd creativity through opportunistic requirement analysis, idea formation and fast prototyping. Popular tools such as wireframes and storyboards are effective for enabling people to capture the requirement essences and express their design ideas in an intuitive and graphical way. These tools also facilitate the open interchange between other design tools and allow designer to collaborate and explore together with alternative ideas and UI schemes. Meanwhile, experienced designers can give guidelines and suggestions for other contestants to improve their conceptual designs. By leveraging all the creativity and contribution from community designers, the project manager can synthesise novel and attractive elements from various designs.

5 *Specification*: This step produces detailed requirement specifications. In this phase, provided with a business requirements document and a set of wireframes resulted from previous phases, contestants make the detailed requirements specification and UML diagrams such as use case and activity diagrams.

6 *Architecture*: This step develops the application architecture including all the modules based on the functional specification developed in the previous step. In this phase, contestants are required to complete a group of design documents and UML diagrams including sequence diagrams, class interface diagrams, component specifications and diagrams, an architecture design specification document and even a small prototype to demonstrate the architecture.

7 *Component design and development*: This step develops the component code by participants in contests. At the design step of the phase, TopCoder contestants are asked to convert a set of architecture documents into a set of diagrams and a component specification document that define design patterns, algorithms, standard technologies, class list, exception handling, running environment as well as configuration. Following the component design documents and diagrams, contestants develop the specified components.

8 *Assembly*: Once the components are developed, together with the architecture, the fully functional application is developed by linking all the components together with the application flow. Contestants follow the architecture and specification documentations from the previous phases and reuse TopCoder component templates if necessary.

9 *Test scenarios and test suites*: This step develops the software assurance plan to validate that all the requirements are fully implemented in the code. Contestants are required to define testing scenarios, test code, and automated scripts to ensure that the software under test (SUT) meets the specified requirements and functionalities. In test scenario competitions, they need to figure out a QA plan for the application with both high-level application test cases and detailed test scenarios. Test scenarios are derived from requirements documentation and in many cases a prototype. In automated script competitions, they are expected to deliver automated test scripts derived from the QA plan.

10 *Bug hunt and race*: TopCoder provides competitions for finding and fixing bugs. In the competition of Bug Hunt, all registrants access the application in trial and file bug reports to a JIRA issue tracker for the application. In the competition of Bug Race, contestants make changes, resolve issues, and perform test validations.

Table 1 summarises the deliverables in all the development phases of TopCoder. As one can see, the key elements of the TopCoder development processes are component-based development, peer review, component customisation, and application integration.

Table 1 TopCoder software development phases and deliverables

Phases	*Deliverables*
Specification	Application requirement specification; use cases, activity diagrams, architecture diagram, site map and site definition, prototype, quality assurance plan, and logical ER model.
Application architecture	Design specification, component deployment diagrams, component sequence diagrams, component interface diagrams, persistence schemas.
Component production	Component specification, use case diagram, class diagram, sequence diagrams, configuration data, working solutions, and test cases.
Application assembly	Complete and tested application.
Certification	Certified solution and certified performance.
Deployment	Fully functioning solution.

4.2 Competition rules analysis

TopCoder has many kinds of competitions, and the following descriptions show typical competition rules:

- '*The coding phase* is a timed event where all contestants are presented with the same three questions representing three levels of complexity and, accordingly three levels of point earnings potential. Points for a problem are awarded upon submission of any solution that successfully compiles and are calculated on the total time elapsed from the time when the problem was opened until the time it was submitted. The coding phase lasts 75 minutes.
- *The challenge phase* is a timed event wherein each competitor has a chance to challenge the functionality of other competitors' code. A successful challenge will result in a loss of the original problem submission points by the defendant, and a 50-point reward for the challenger. Unsuccessful challengers will incur a point reduction of 25 points as a penalty, applied against their total score in that round of competition. The challenge phase lasts 15 minutes.
- *The system testing phase* is applied to all submitted code that has not already been successfully challenged. If the TopCoder System Test finds code that is flawed, the author of that code submission will lose all of the points that were originally earned for that code submission. The automated tester will apply a set of inputs, expecting the output from the code submission to be correct. If the output from a coder's submission does not match the expected output, the submission is considered flawed. The same set of input/output test cases will be applied to all code submissions for a given problem. All successful challenges from the challenge phase will be added to the sets of inputs for the system testing phase.

Note the following key elements in competition rules:

a limited time

b cross testing among contestants to win competition

c the software is further validated by automated tools to ensure minimum quality.

These rules show that the major objective of the TopCoder is to identify top software engineers first, and the quality product is a by-product of competition. As often only the top two teams will win any award or price, and thus one of the main goals is to identify the bugs in competitor's code to eliminate the competitor. However, at the same time, as the competitor will do the same, the contestant will spend significant effort to ensure the code is correct. Thus, the contestant may spend significant effort in verifying his/her code as many times as possible with the time limit to ensure that the competitor will not be able to identify a bug. These activities can be seen in Table 2.

Table 2 TopCoder process

First phase (defence)	*Second phase (offence)*
Inspect/test the code over and over again to ensure it has no bug. For example, use the list of features in the specification documents provided to see if the code satisfies the requirements. Rank those features that competitors may fail.	Inspect/test the code of competitor using the ranked features. As own code has been inspected already, inspection of competitor's code can be easier.
Provide a list of test cases, and rank them according to the possibility of faults. Use the ranked test cases to go through own code to ensure no fault.	Use the ranked test cases developed to test run the competitor's code.
Identify the most difficult aspect of components, such as the starting and ending point of complex loop statements. Make a list of such difficult spots.	Use this list to check the competitor's code.

Note that this competition rule is not a *zero-sum game* in game theory. The zero-sum is where the gain of one party will be matched by the loss of the other party. In a TopCoder's competition, everyone has a chance of lose as each party can find bugs in counterpart's code. If coder A finds a bug in coder B's software, and B also finds a bug in A's software, both A and B lose the game, and no one will advance. Similarly, A cannot find a bug in B's code, and B cannot either in A' code, both can advance, but only the top two coders will win. Thus, as reported in Tibbetts (2012), most often only two teams will join a competition. To become winners in such a mutual destructive contest, a coder must do his/her best to avoid losing to other coders and secure a high ranking score in the system to scare off other potential competitors. That is the nature of this kind of competition, often called Chicken Game or Hawk-Dove Game (Rapoport and Chammah, 1966) in the game theory because less aggressive players (chicken or dove) will yield to aggressive players.

If software quality is most important, and it is not necessary to rank top engineers in the current competition, the following rules may be used instead:

- Limit development time by *software quality constraints* rather than *competition time*. For example, unless the fault arrival rate of the target code has been approaching zero for an extended period time, the competition will be continued. This criterion has been used in conventional software development for test completion, and a rule of thumb is that the testing should continue until no new bugs are discovered for an extended period of time.
- Encourage cross testing and validation such as paying money for any bugs identified rather than using the bug counts to eliminate competitors.
- Supply automated tools for developers and testers so that everyone has the same starting point.
- Reward unique contribution such as a new design framework that others can build on to develop the software.

This is a *collaboration-based* software crowdsourcing rather than competition-based crowdsourcing where limited number of contestants will win a price. In the collaboration-based approach, more people will win either a monetary reward or a reputation. The jury is out whether the competition-based rules or the collaboration-based rule will be better in term of software quality. The competition-based rules may be better in identifying top software designers, coders, and testers, but collaboration-based rules may be better for massive participation and education. The selection of rule styles also impacts the creativity of the community. Collaboration-based contests can trigger intense interactions between participants so that the community members with less experience can improve their skills from expert members. And competitive coding can stimulate them to do their best to seek innovative solutions.

4.3 Who-what-when analysis

While the focus of TopCoder is for competition-based component development, but its process has been extended to develop almost all kinds of software deliverables including specification, design, architecture, and testing. To simplify the analysis, we will discuss component and algorithm development only first, specifically we will use the simplified model as shown in Harvard Catalyst (Harvard University, 2012). This process has essentially four parties with their responsibilities below:

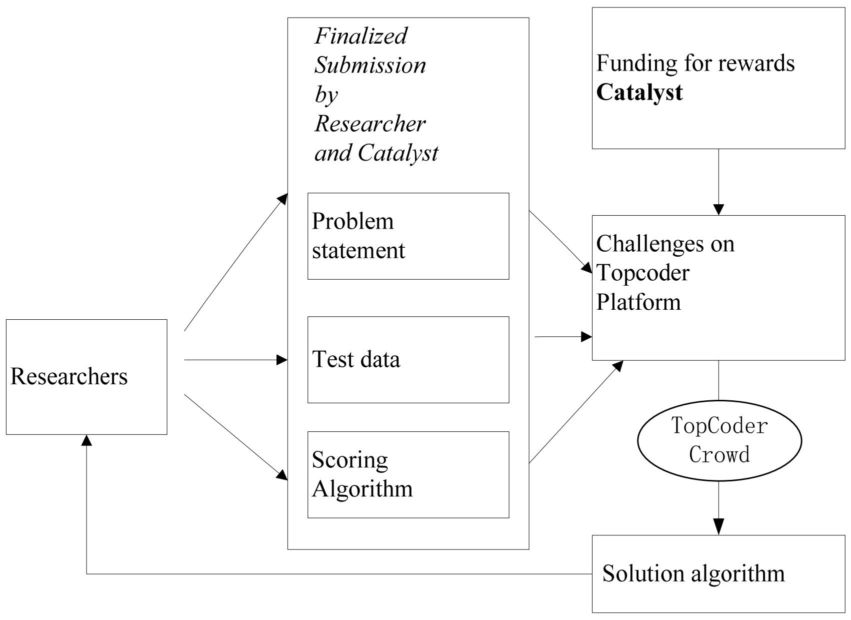
- *Researchers:* They work with catalysts to finalise the submission to the crowd including problem statement, they also prepare test data, and finally score algorithms submitted by the crowd.
- *Catalysts:* They prepare the needs and funding for researchers and the crowd.
- *Crowd:* They look at the problems issued by researchers, participate in the competition by coming up with new algorithms, and if they win, they will get rewards from catalysts.
- *TopCoder people:* They will work with researchers in preparing the platform for the competition.

The operational sequence indicates that researchers and catalysts are earliest participants of this process, and they need to finalise the problem statements together. However, researchers have additional tasks of preparing the test data, and scoring the algorithms submitted.

Using software engineering terms, the catalysts are responsible for problem statements and they approve the requirement statements prepared by researchers, and finally accept the solutions done by the crowd; the researchers responsible for requirement analysis, high-level design, test case generation, and test case execution; the crowds responsible for algorithm design within the context of a problem.

Software engineering textbooks often indicated that the most critical element in software development is that the requirement analysis must be properly done. This lesson is re-learned in crowdsourcing as stated in Section 2 (Tibbetts, 2012). However, if the characterisation of the Harvard-TopCoder process is correct, majority of the requirement work will be done by researchers, and the crowd works on a limited scope in software development after requirements are well understood and high-level architecture is known. Recall that the researchers need to come up with the problem statements (requirement analysis and specification and also high-level design), prepare test data, and actually score algorithms submitted. Furthermore, they work from the beginning to the end of the competition. Thus, a great majority of work for the project is done by researchers, rather than the crowd. While, the crowd may contribute a critical piece of the puzzle, i.e., the algorithm, but they will be active only in the middle step as they are not involved either in problem statements or in scoring algorithms.

Figure 5 Harvard-TopCoder algorithm development process


Finalized Submission by Researcher and Catalyst
Funding for rewards Catalyst
Problem statement
Challenges on Topcoder Platform
Test data
Researchers
TopCoder Crowd
Scoring Algorithm
Solution algorithm

The TopCoder process also allows other aspects of software development to be crowdsourced such as design and testing. But the overall processes are similar, except the funding party can crowdsource the design first, and after the design is fixed, they can crowdsource the code, and finally they can crowdsource the testing of the code to complete their project.

4.4 Comparing with a traditional development process

IBM Cleanroom development methodology has been popular in late 1980s to 1990s. It focuses on formal methods on specification and testing using statistical control. An important element of this methodology is the separation of responsibility. Specifically, in a three-party Cleanroom, one team will be responsible for specification, the second team responsible for coding without compilers, and the third team is responsible to verify the code developed by the second team with respect to the specification developed by the first team. Software development by three independent parties in the IBM Cleanroom methodology allows cross-validation of the deliverables by other two teams. The cross validation activities can be characterised as offence (finding bugs in other people's work) and defence (reducing bug in people's own work).

- *Offence*: To carry out the tasks, each team needs to understand its requirements, and examine the validity of the *inputs* to determine if they are feasible, correct, consistent and complete. This can be done by inspecting, reviewing, simulating, model checking, verifying the contents of the inputs. This kind of process often reveals mistakes, inconsistency, incompleteness, complex user interaction, invalid assumptions, and other issues, which can be useful feedbacks to those who prepared the input documents. Because any mistakes in the input document may cause significant problems in the current tasks. Thus, the goal is to *maximise* the fault detection rate of the input documents.
- *Defence*: Once requirements are understood, the team needs to prepare its output. However, the team realises that their *outputs* will be cross examined by other teams carefully, and the team may lose its creditability if its outputs are of low quality. Thus, the team needs to spend significant time to check and verify its deliverables to *minimise* the probability of bugs and to minimise the damage of potential bugs.

Table 3 summarises the offence-defence relationship.

Table 3 Offence and defence definition

Offence activities	*Defence activities*
Evaluate the *inputs* including any input documents, prototypes, interviews and relevant materials that will be used in performing the tasks.	Evaluate the *outputs* including any deliverables such as documents and software.
Goal: *Maximise* the number of faults in the input documents, and provide feedback to those who prepared the inputs.	Goal: *Minimise* the number of bugs that will be found by other teams or people (crowd).

Table 4 shows the cross-relationships among these three teams with their tasks, and their offence and defence activities.

Table 4 Offence and defence of IBM Cleanroom teams

	Main tasks	*Offence*	*Defence*
Specification team	Develop a specification for the problem at hand. May need to do requirement analysis and high-level design before developing a specification detailed enough for the coding team to develop the code.	The team must understand the problem well, identify those bugs in the requirement documents to *maximise* the probability for identifying bugs in requirements.	As the specification will be reviewed carefully by the coding team, the specification team will inspect the specification and possibly even simulate the specification multiple times before delivering it to the coding team to *minimise* the probability of bugs in the specification.

Table 4 Offence and defence of IBM Cleanroom teams (continued)

	Main tasks	*Offence*	*Defence*
Coding team	Develop the code based on the specification delivered without using compilers. As no compiler will be available, thus code will be reviewed and inspected multiple times before delivering to the testing team.	The team needs to review, inspect, and simulate the specification to *maximise* the probability to identify bugs in the specification. The specification bugs will be reported back to the specification team.	The coding team will inspect and review the code over and over again with no compilation to *minimise* any potential bugs in the code.
Testing team	Compile the code, develop test cases, and run test cases according to statistical quality control models.	As the team needs to review and inspect the specification and the code, the code will also be compiled and tested to *maximise* the probability of finding bugs in the specification and or the code. If there is any bug in the specification and/or code, and any bugs found will be reported back to the specification or coding team	The code will be reviewed by customers, and thus the testing team must verify and test the code over and over again according to statistical models to *minimise* the probability of bugs in the final code.

One reason that IBM Cleanroom methodology has received significant attention in 1980s and 1990s is that its ability to develop zero-defect software. While one may argue various features in the IBM Cleanroom methodology[1] for its capability to develop zero-defect software, one reason is the division of responsibility where all three parties must cooperate with each other intensively but independently:

- The specification team will be evaluated with respect to the quality of the specification delivered (such as the number of bugs in the specification detected by other teams), and thus they must work hard to minimise any potential specification faults.
- The coding team will be evaluated with respect to the quality of the code (such as the number of coding bug delivered by the testing team during the process), even worse they must develop the code without compilers, and thus they must inspect and re-inspect the code carefully.
- The testing team will be evaluated with respect to the quality of the final code using statistical models, and thus they must work on all the documents including the specification and code carefully, and they must compile and test the software according to statistical models.

By classifying the activities of each team into offence and defence activities, one can see that the IBM Cleanroom methodology is an application of game theory in software development as there are min-max (or defence-offence) activities among these independent teams. The idea is that once the developers are divided into different teams with different tasks, objectives, and evaluation criteria, these teams will compete according to their evaluation criteria, and the end result is the quality of software produced due to the competition among the team members.

Table 5 Comparing IBM Cleanroom and Harvard-TopCoder process

	IBM Cleanroom	*Harvard-TopCoder*
Number of parties	Three.	Three.
Specification	Formal.	UML.
Party one responsibility	Specification. They develop the specification of the software to be developed by the second team.	Catalysts. They provide: 1 funding 2 original problem statements 3 review of the final solutions submitted.
Party two responsibility	Coding. They review the specification done by the first team, develop the code according to the specification developed by the first team and they inspect the code over and over again to minimise the bug without a compiler.	Researchers. They decompose the problem, develop the specification, develop the test cases, and evaluate the code developed by the crowds.
Party three responsibility	Testing team. They verify the code developed by the second team with respect to the specification developed by first team.	Crowds. They develop algorithms according to the specification delivered by the party one.
Allow incremental development	Yes.	Yes as this process can be modified so that the whole algorithm can be decomposed into sub-algorithms.

One can now compare the IBM Cleanroom with Harvard-TopCoder process to see their similarities and differences.

One can see that in the Harvard-TopCoder process, the task for the researchers is much heavier than the IBM Specification team as the team carries the load of both the specification team and the testing team of the IBM Cleanroom methodology. Now the offence-defence analysis can be done for the Harvard-TopCoder process as shown in Table 6.

Table 6 Offence and defence analysis of Harvard-TopCoder teams

	Main tasks	*Offence*	*Defence*
Catalysts	Develop problem statements with Researchers for the project, need to specify final acceptance criteria, secure funding for crowdsourcing.	Need to ensure the problem is feasible, thus go over with the Research team to *maximise* the probability of identifying problems in requirements.	The team needs to ensure that the Researcher understand the problem and well decompose the problem to *minimise* the probability of bugs in problem statements.
Researchers	Understand the problem, decompose the problem, develop the high-level design for the crowd to develop components or algorithms, develop test cases for acceptance testing, and evaluate the algorithm/code submitted by the test cases developed.	Working with the catalysts to develop quality specification, make sure that the problem is feasible to *maximise* the probability of identifying problems in requirements.	The specification/test cases developed must be of high quality, and thus they need to review and inspect the specification carefully, and answer any inquiries from the crowd. These *minimise* the probability of finding bugs in specifications and test cases.
Crowd	Develop algorithms or components based on specification supplied.	The crowd will review the specification carefully to ensure complete understanding, and identify any bugs in the specification, and develop test cases based on specifications to test other algorithms or components submitted by other contestants. These will *maximise* the probability of finding any bugs in specification provided by the research team.	The crowd will evaluate its algorithm or components carefully using test cases developed to *minimise* the possibility of bugs in the code.

4.5 Updated software crowdsourcing process

Based on the IBM Cleanroom methodology, the Harvard-TopCoder process may be updated with the following processes with two or more kinds of crowds:

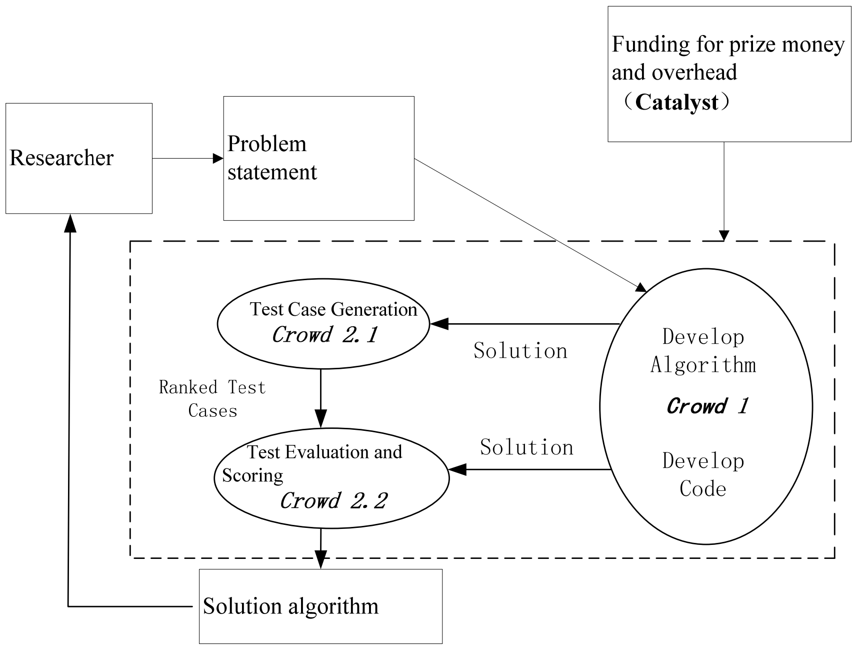
- *Catalysts*: they have the usual role as before.
- *Researchers*: Their role is now reduced as test cases preparation is crowdsourced, and evaluation may be also crowdsourced.
- *Crowd 1*: Their role is the same, i.e., to develop algorithm or components.
- *Crowd 2*: They prepare test cases and perform evaluation. Crowd 2 can be further divided into two or more sub-groups.

a *Crowd 2.1*: This sub-group is responsible for test case development, and cross validation of test cases by each other. They may be responsible for ranking test cases according to various criteria such as criticality. Note that multiple crowds can serve here, for example, one group can develop test cases for GUI navigation, and the other for generating cases based on control flow.

b *Crowd 2.2*: This sub-group is responsible for execution of test cases on the submitted solutions provided by Crowd 1, and they can also cross validate the evaluation performed by other fellow participants. They can rank test cases according to potency of test cases (Tsai et al., 2008).

The updated process is shown in Figure 6.

Figure 6 Updated software crowdsourcing process



The evaluation of each party will also be updated. The Crowd 2 will be evaluated with respect to the number of test cases developed, as well as the test results. For example, one ranking criteria is test *potency*, and the potency is defined as the capability of detecting bugs in the code. The more bugs it can detect, the higher the potency is. Test cases can also be ranked according to *criticality*. A critical test case means that if the software fails it, the system does not pass the acceptance test. A test case can be critical but not potent because the software passes the test cases. Similarly, a test case can be potent but not critical because the software fails the test case, but as it touches only the minor feature of the software. For example, the logos in the presentation may be displayed at a place that is not most appropriate while the software works fine. A test case may be both critical and potent if the software fails this feature and is rejected by the client. Table 7 summarises the discussions.

While coders and software engineering will be rewarded for developing algorithms or components, testers will be rewarded for developing test cases and more importantly finding bugs in algorithms and components. Highest awards should be given to test cases that are both critical and potent, and lowest awards should be given to those test cases that are not critical and not potent at the same time.

Table 7 Test case criticality and potency

	Criticality	*Potency*	*Cases*	*Ranking*
1	High	High	A critical test case and it failed the software often.	Highest ranking
2	High	Low	A critical test case but it failed to identify bugs in the software.	High
3	Low	High	A non-essential test case, but it failed the software often.	High
4	Low	Low	A non-essential test case and it does not catch a bug in the software.	Low

Note that the updated process follows the Webstrar (Tsai et al., 2004, 2005) principles:

- software can be tested, evaluated and ranked by running test cases on the software
- test cases can be verified, evaluated, and ranked according to the criticality and potency criteria by various methodologies and models of test case evaluation
- methodologies and models of test case evaluation can be evaluated and ranked as well by cross evaluation and validation
- different policies can be established to set the testing guidelines for specific domain or specific design techniques.

Thus, the entire test and evaluation items can be evaluated and ranked. The updated process allows testers to conduct cross evaluation of each other's test cases, and rank these test cases based on both criticality and potency.

This new crowdsourcing process, while similar in certain ways with the current TopCoder process, such as shown in Section 4.2, has different psychology effect. In the TopCoder process, contestants who developed a component will be competing in discovering bugs in competitor's code directly, i.e., the competition is cut throat. Specifically, if there K contestants, we have the following competition:

- [K × K] where K parties look over K components to eliminate each other.

But in the updated processes, we have instead:

- [M × N × L] where M components will be developed by M teams, and these components will be evaluated using test cases prepared by N parties, and the actual test runs done by L parties.

The discussion can be summarised in Table 8.

Table 8 Competition parties and rules in Harvard-TopCoder and revised processes

	Competition Parties	*Competition rules*
Harvard-TopCoder process	K × K among contestants, 1 × K between researchers and crowd, 1 × 1 between catalysts and researchers	Live and let live between catalysts and researchers; live and let live between researchers and crowd; live and let die among contestants.
Updated process	M × N × L among algorithm crowd, test case crowd, and test execution crowd; 1 × M between researchers and crowd, 1 × 1 between catalysts and researchers	Live and let live among and between all parties.

Furthermore, while individual persons can participate in all the parties, but in general, this is a *live-and-let-live* competition, where each party will be rewarded by their respective criteria stated before. Thus, the psychology in this kind of competition will be different from the [K × K] competition.

While traditional independent verification and validation (IV&V) and IBM Cleanroom methodology do not allow any party to overlap, the updated crowdsourcing process can allow overlap. As the person who developed the initial component may be a good candidate to develop critical test cases as the person just went over the mental process of developing the algorithm. The issue of whether allowing the people to be involved in both development and testing and evaluation (T&E) in a crowdsourcing environment is a research topic.

Table 9 Offence and defence analysis of the updated process

	Main tasks	*Offence*	*Defence*
Catalysts	Develop a problem statement with Researchers; specify final acceptance criteria; secure funding.	Need to ensure the problem is feasible, thus go over with the Researcher team to *maximise* the probability of identifying problems in requirements.	The team must ensure the Researchers understand the problem and well decompose the problem to *minimise* the probability of bugs in problem statements.
Researchers	Understand the problem; develop a specification for the problem at hand; May also develop the high-level design for the crowd to develop algorithms and test cases.	Working with the catalysts to develop quality specifications, make sure that the problem is feasible to *maximise* the probability of identifying problems in requirements.	The specification developed must be of high quality, and thus they need to review and inspect the specification carefully, and answer any inquiries from algorithm or testing crowds timely. These *minimise* the probability of finding bugs in specifications.

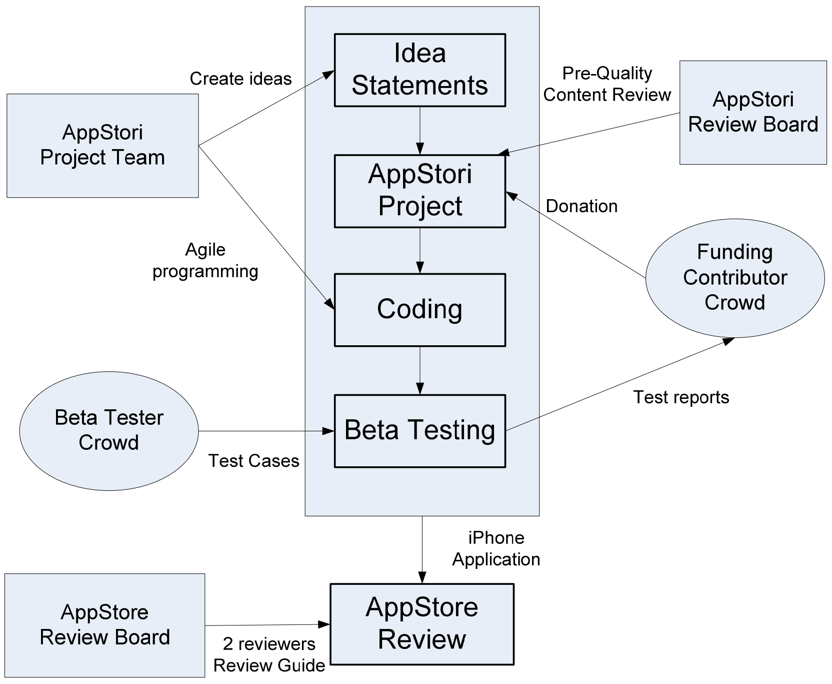
Table 9 Offence and defence analysis of the updated process (continued)

	Main tasks	*Offence*	*Defence*
Algorithm Crowd	Develop the algorithm for the specification.	The crowd must review the specification carefully to identify any bugs in the specification, and must develop algorithm or code to be tested and evaluated by all other teams. These will *maximise* the probability of finding any bugs in specifications.	As the algorithm or code delivered by the crowd may be reviewed by all other teams, and thus the deliverables will be self-tested and inspected over and over again to *minimise* the possibility of bugs.
Testing Case Crowd	Based on the specification, develop black-box or functional test cases, and develop white-box test cases on selected output from the Algorithm crowd; rank test cases based on criticality.	Inspect the specifications delivered carefully to find any bug to develop test cases; inspect the algorithm/code delivered by the Algorithm crowd to identify key features to test. These will *maximise* the probability of finding bugs in specifications or algorithm/code, and report any bugs found to the Researcher team and Algorithm crowd immediately.	All the test cases and ranking can be cross validated by fellow test case participants can be cross validated. These will *minimise* the probability of bugs in test cases, report any report any bug in test cases developed by fellow participants.
Test Evaluation Team	Based on the specification, algorithm/code developed by the Algorithm crowd, and test cases developed by testing case crowd, execute the test cases to see if the algorithm/code pass the evaluation. Record and rank the potency of test case dynamically based on test results.	The team can cross validate specifications, algorithm/code, and test cases for possible inconsistency to *maximise* the probability of identifying bugs in these documents, report any bugs in the specification, algorithm/code, and test cases to appropriate teams.	Test results can be cross validated by fellow test evaluation team, this will *minimise* the probability of mistakes in test evaluation.

5 Creativity in AppStori development processes

Another interesting case of software crowdsourcing is the mobile application development. One well-known place is Apple App Store where it is essentially an online IOS application market where developers can directly delivery their products with creative design to smartphone users. These developers are motivated to contribute innovative design and attract more user downloads by the micro-payment mechanism of the App Store. Within less than four years, it becomes a huge mobile application ecosystem with over 150,000 active developers and over 700,000 IOS applications. It is also the largest online software distribution channel for IOS applications.

Figure 7 AppStori Crowdsourcing Development Processes (see online version for colours)



Apple App Store publishes the review guidelines and sets up an executive review board with 40 staffs that check every application submitted and determine whether to accept these submissions based on the guideline. Apple keeps all the review jobs completely within its own organisation to make sure that all the submissions strictly follow the guideline and achieve reasonable level of quality. The delay between submission and review is not guaranteed. According to App Store Metrics (http://148apps.biz/app-store-metrics/), this delay currently averages at 5.91 business days with a maximum delay of 34 days.

Recent studies on millions of IOS developers suggest that the majority of iPhone apps are actually developed by small teams or individuals. To create a successful and profit iPhone application, they often need external funding and beta testers to support their projects and improve the quality of the applications. Many online platforms such as AppStori (http://www.appstori.com) and iBetaTest (http://www.iBetaTest.com), adopt software crowdsourcing model to help these IOS developers to make their creative ideas and dreams into reality. AppStori is a community-based, collaborative platform that encourages mobile creativity and foster innovations by bringing developers, beta-testers, donors and supportive customers together. AppStori provides a 'preview' window for iPhone application enthusiasts to choose their favourite ideas, support and actively engage in the promising projects. The process is shown in Figure 7.

- *Crowdfunding and stakeholders*: Similar to Kickstarter.com in the financial market, any AppStori member can post a novel idea of IOS application onto the AppStori, specifying the project goals and deadlines, and ask for funding. After the AppStori Review Board approves the project, its content will be visible online. Any people who may be interested in the project can become a stakeholder by funding a project. Although a stakeholder cannot withdraw its fund before the end of the project, the person can check the progress of the project and request the project leader to finish the project within a specified time frame. Otherwise, the funding will not be transferred to the developer team at the end. In addition, the developers may setup a tiered rewarding system to allow shareholders to benefit from the project profit.
- *Transparency and Agile development*: AppStori encourages 100% transparency between the developer team and the community. Every team member must provide detailed personal profile to present their background and role in the project. And the project status must be updated in time to display the progress to the community. Anyone who is willing to be part of the team can join the project as either a developer or a beta tester. The beta testers can evaluate early versions of the application under test and give direct feedbacks and bug reports to the developers.
- *Web 2.0 collaboration and communication*: AppStori provides Web 2.0 tools that can support collaboration among all parties with Blog, Twitter and Facebook. These Web 2.0 tools greatly facilitate users to know about the activities and status of the project

5.1 Stimulating creativity in mobile application development

AppStori encourages exploration diverse paths and embracing rich styles to create innovative mobile applications. It supports transparent and agile development to create an open platform for all the stakeholders and allow iterative refinement for innovative ideas. Web 2.0 communications make it easier for donors, developers, and testers to exchange ideas, post comments and figure out brilliant ideas. AppStori's open and creative environment enables numerous cycles of 'trial-and-error' towards the successful design until it can be ready for publication on AppStore.

Comparing to the TopCoder development process, AppStori is much flexible with significant freedom that TopCoder contestants do not enjoy. Table 10 compares with these processes.

One can see that AppStori is much close to the agile methods while TopCoder is closer to the traditional processes such as Waterfall, Spiral and documentation-based processes. The most striking idea of AppStori is that from the beginning, almost everything can be crowdsourced including initial project concept formation (where developers come up with an idea to put on AppStori.com), to development (developers can join the developer team), to T&E (developers and users can join the testing), and funding (users and testers can see the software to see if the product will be downloaded by people for profit).While TopCoder catalysts and researchers control much of the development processes, and show only those things that they want to expose to the crowd, AppStori is a go-anywhere style of project development.

Table 10 Comparing TopCoder and AppStori

	TopCoder	*AppStori*
Project ideas	The clients (catalysts) come up with the idea.	Developers (and they can be co-users) come up with the idea.
Developing specification	Clients with researchers or in-house IT professionals to understand the problem, decompose the problem, and deliver the specification written in UML.	Developers come up with the design, and specification languages can be informal or light weighted.
Time for algorithm or component development	Limited so that contestants need to be focused during the competition time	Freestyle, yet developers have project deadlines to meet, they can work asynchronously any time in a day.
Test case preparation	Can be done by researchers or crowdsourced.	Crowdsourced to beta testers and other users.
Test run and evaluation	Can be done by researchers or crowdsourced.	Crowdsourced to best testers and other users.
Crowd participants	Mainly compete against each other to win competitions	Mainly help each other to develop the best products for customers to download.
Development processes	While the code or algorithm may be important, documents generated are important.	Software developed is important with minor concerns with documentation.
Competition Parties	People can first choose to participate in certain competitions only, and then the best of two among all those participate will win. Those not participated are not engaged in any competition.	As the project ideas are open to AppStori, and thus the competitions are really among the crowd, or the crowds form their own parties to compete with another crowd parties. In certain ways, everyone is competing with every other for most monetary/enjoyment rewards, either they are developers, beta testers, users, or shareholders.

Can AppStori develop quality software? If they cannot develop software, no one will download their software from App Store, and thus no profit potential, and that is the main driving force for quality. Table 11 shows the offence and defence among AppStori parties to come up with quality products.

Crowdsourcing sites such as AppStori enable the app developers to build and maintain relationship with their user base through a variety of communication. While crowdfunding is an important part of the solution that AppStori provides, it is only one aspect of the platform. There are other ways that AppStori community members can participate. For example: users can contribute ideas and feedback that guide an app’s development process, provide social support for projects via social media promotion, become beta testers to test out early versions of apps, as well as discover the next generation of mobile apps.

Table 11 Offence-defence analysis of AppStori process

	Main tasks	*Offence*	*Defence*
AppStori Review Board	Review the proposal of a AppStori project creator and decide whether to allow the creator to publish his project	Screen the creative ideas from project team in the pre-quality step and make decision for admission based on the innovation of the idea and the qualification of the team leader. This will *maximise* the quality of accepted proposal and the probability of its success.	Ensure the accepted project submission is a legitimate proposal for mobile application. This will *minimise* the possibility of web spam and low quality proposals.
Project team	Propose an innovative mobile application to the AppStori; raise funding for the project; attract skilled programmers to join the team; develop the mobile app in an agile manner; design a rewarding structure for people to fund the project; develop the quality code quickly to meet the project milestone.	Interact with potential users about the project concepts to know their preference and choices. Submit the proposal to the AppStori and convince the AppStori Review Board about the promising prospect of the proposal. This will *maximise* the probability of success as user preferences are known.	Ensure the software is good enough to be tested by Beta testers by performing in-house testing and/or inspection rigorously. This will *minimise* the probability of bugs in the code.
Beta-test volunteers	Following the project requirement, develop mobile app test cases and perform functional tests, locality tests and usability tests of the mobile app. And post test reports to the development team for further improvement.	Understand the project goal, attempt multiple user interface approach to ensure the software is good for as many users as possible. This will *maximise* the probability of finding bugs in the project concepts and user preference early to provide feedbacks and suggest alternative design.	Make sure that tests are done correctly so that they can maintain their reputation in Beta testing. This will *minimise* the probability of bugs in test cases.
Funding contributors	Evaluate the project to see the financial future of the product, and provide fund for the project team.	Select the best projects for funding; check the status of the project often to remind the developer team. These will *maximise* the chance for the project to be successful and future profit brought from their contributions.	The best defence is to identify the best project idea that people wanted, and identify the best available team to carry out the project. These will *minimise* the possibility of project failure.

Table 11 Offence-defence analysis of AppStori process (continued)

	Main tasks	*Offence*	*Defence*
AppStore review team	Perform the conformity test on Mobile apps according to the IOS APP Guideline and decide whether to put the app into the AppStore.	Rigorously review and test all aspects of the app under test, and two reviewers can jointly decide whether to admit the app or not. These *maximise* the quality level of apps in the AppStore platform.	The team needs to test the app and verify the conformity of the app. These will *minimise* the probability of non-compliance.

AppStori is a great place for people to learn new applications, technologies, and trend in mobile applications, and mobile consumers and enthusiasts can connect directly with mobile developers and entrepreneurs for direct feedbacks. These feedbacks are expensive to obtain in the past. All these features of AppStori are essential to foster crowd creativity for better mobile applications.

6 Game theory interpretation

6.1 IBM Cleanroom methodology

If IBM Cleanroom methodology is a game, it is a *cooperative* game. A cooperative game is a game where groups of players will work together to achieve common goals, and individuals within groups do not make their own moves independently. The group will decide its strategies to achieve its own goals. Specifically, each of three teams in the IBM Cleanroom works as a group to reach its own goals and objectives.

It is also an *asymmetric* game. A symmetric game is a game that will reach the same results if the same strategy is used regardless who are involved in the strategy. Software development is unfortunately highly human dependent, two persons working on the same problem, using the same methodology and even the same design patterns, are likely to end up with two distinct even though similar designs. Similarly, two testing evaluations, using the same testing techniques and processes, will identify different set of bugs even though the two sets overlap.

It is *not* a *zero-sum game*, but it is a *simultaneous game* where participants can make their moves simultaneously even though activities may need to be coordinated. It is also an imperfect game as not all the information will be available to all the parties.

6.2 Harvard-TopCoder process and revised process

Both have elements of a cooperative game as the catalyst and researcher teams work as groups with a team goal and objective, but the individual people in the crowd work alone, not cooperative. It is *asymmetric* as different people will produce different results in all three teams with significant performance variation. It is *not a zero-sum game*, but a *simultaneous game*, and an *imperfect game* as no one knows the complete status of the project.

6.3 AppStori process

This process is a cooperative game even though the group decision may be made within the crowd, and thus may be difficult to reach consensus. It is *asymmetric* as the results depend on the people who participate in the game; it is not a zero-sum game, but a simultaneous game, and an imperfect game.

As both processes are not zero-sum game, they can encourage co-creativity by allowing each one to contribute and gain rewards through participation. Cooperative games often tend to have positive influence on the player creativity as they are willing to help each other to improve their performance. Non-cooperative game such as Harvard-TopCoder has fewer stimuli for people to learn from each other. Furthermore, TopCoder is also like a *game of Chicken* (Chicken Game, 2012) where each participant will try their best to beat their competitors, and thus each party may be mutually destroyed by each other as the worst outcome for all participants. The Chicken game came from a game where two drivers facing each other for a head-on collision, if any party changes the direction, the other party will win. However, the worst possible outcome will happen if both will not yield, and both are killed in the collision.

AppStori is like a *Coordination game* (Coordination Game, 2012) where all parties can gain from the game mutually, if they trust each other and are willing to make mutually satisfying group decisions. In some scenarios where people have not formed a well-functioning community yet, they may also take tit-for-tat strategy to cooperate with others on the condition that other players also make reciprocal contributions.

While many papers are available today on software crowdsourcing, however, few have studied the detailed process to identify the min-max issues. This paper starts from software crowdsourcing processes, and identified key ideas particularly related to the min-max issues and the impact on creativity in software development.

7 Conclusions

Software development has been deemed as both technical and creative activities. It demands disciplined engineering approach to achieve project goals in an efficient and cost-effective manner. And it also needs creativity from developers to think out-of-box for innovative concepts, elegant design, and clean code. The emerging crowdsourcing practices inspire a new dimension to stimulate creativity through online collective intelligence while ensuring high quality of software products.

One can see that, TopCoder, being one of the pioneers in software crowdsourcing, starts a process based on competition and rigorous documentation. Furthermore, while many software development artefacts can be crowdsourced, TopCoder is known to produce quality software components and algorithms. AppStori takes a rather distinct path, and it has even crowdsourced project concepts, development, funding, and evaluation all to the crowd. In fact, in AppStori, crowds interact with crowds with the organisers play a less role, while in TopCoder, the crowd interact with the client mostly with the organisers play a heavy role. One can view that AppStori has pushed the capability to crowdsourcing further as the crowds can play multiple roles during the process, and stimulate collective creativity during the process.

Another important identification is that it is the min-max nature of competition within software crowdsourcing that produce the creativity and quality of software. The min-max nature can be practiced in a live-and-let-die competitions (such as in TopCoder process) or in a collaborative manner where all participants can win, including the crowds who fund the project, create the initial project concept, develop the project, test and evaluate the project, download and use the product delivered the product can all win this by participating in this highly complex and iterative process. Therefore, the min-max is the key component in the crowdsourcing process to affect the collective creativity in the software development.

TopCoder and AppStori showed that two dramatically different software crowdsourcing approaches with different techniques and processes can both lead to delivering quality software.

Acknowledgements

This project is sponsored US National Science Foundation project DUE0942453 and National Science Foundation China (No. 61073003), National Basic Research Programme of China (No. 2011CB302505), the Open Fund of the State Key Laboratory of Software Development Environment (No. SKLSDE-2009KF-2-0X and SKLSDE-2012ZX-19), and Fujitsu Laboratory.

References

'DARPA and TopCoder also seek a platform middle and high school students on software development in October 2011' (2011) [online] http://www.topcoder.com/aboutus/archive/2010/10/darpa-and-topcoder-seek-to-change-how-students-spend-time-online-2/ (accessed 10 July 2011).

Agile Software Development (2012) *Wikipedia* [online] http://en.wikipedia.org/wiki/Agile_software_development (accessed 29 June 2012).

Apple App Store Review Guidelines (2010) [online] https://developer.apple.com/appstore/guidelines.html (accessed 9 September 2010; 10 June 2012).

Apple Store Metrics [online] http://148apps.biz/app-store-metrics/ (accessed 25 June 2012).

Archak, N. (2010) 'Money, 'glory and cheap talk: analyzing strategic behavior of contestants in simultaneous crowdsourcing contests on TopCoder.com'', *Proceedings of the 19th International Conference on World Wide Web*.

Archak, N. and Sundararajan, A. (2009) 'Optimal design of crowdsourcing contests', *Proc. of International Conference on Information Systems (ICIS), Association for Information Systems*, pp.1–16.

Bacon, D.F., Chen, Y., Parkes, D. and Rao, M. (2009) 'A market-based approach to software evolution', *24th ACM SIGPLAN Conference Companion on Object oriented Programming Systems Languages and Applications*, 25–29 October, Orlando, Florida, USA.

Benkler, Y. (2012) *Wikipedia* [online] http://en.wikipedia.org/wiki/Commons-based_peer_production (accessed retrieved 29 June 2012).

Bullinger, A. and Moeslein, K. (2010) 'Innovation contests – where are we?', *AMCIS (Americas Conference on Information Systems) 2010 Proceedings*, Paper 28, http://aisel.aisnet.org/amcis2010/28 (accessed 28 June 2012).

Capability Maturity Model Integration (CMMI) *Wikipedia* [online] http://en.wikipedia.org/wiki/Capability_Maturity_Model_Integration (accessed 28 June 2012).

Chen, Y. and Tsai, W.T. (2011) *Service-Oriented Computing and Web Software Integration*, 3rd ed., Kendall Hunt Publishing, ISBN: 978-1-4652-0558-2

Chicken Game (2012) Wikipedia entry [online] http://en.wikipedia.org/wiki/Chicken_(game) (accessed 25 June 2012).

Coordination Game (2012) *Wikipedia* [online] http://en.wikipedia.org/wiki/Coordination_game (accessed 25 June 2012).

Department of Defense-STD-2167A (DoD) (2012) *Wikipedia* [online] http://en.wikipedia.org/wiki/DOD-STD-2167A (accessed 29 June 2012).

DiPalantino, D. and Vojnović, M. (2009) 'Crowdsourcing and all-pay auctions', *EC '09 Proceedings of the 10th ACM conference on Electronic Commerce*.

Doan, A., Ramakrishnan, R. and Halevy, A.Y. (2011) 'Crowdsourcing systems on the World-Wide Web', *Communications of ACM*, April, Vol. 54, No. 4, pp.86–96.

Fink, E., Sharifi, M. and Carbonell, J.G. (2011) 'Application of machine learning and crowdsourcing to detection of cybersecurity threats', *Proceedings of the DHS Science Conference, Fifth Annual University Network Summit*.

Fischer, G. (2004) 'Social creativity: turning barriers into opportunities for collaborative design', *Proceedings of the Eighth Conference on Participatory Design: Artful Integration: Interweaving Media, Materials and Practices*, Vol. 1, pp.152–161.

Harvard University (2012) Harvard University Clinical and Translational Science Center, 'Algorithm Development through Crowdsourcing', http://catalyst.harvard.edu/services/crowdsourcing/ (accessed 10 June 2012).

Horton, J.J. and Chilton, L.B. (2010) 'The labor economics of paid crowdsourcing', *EC '10 Proceedings of the 11th ACM conference on Electronic Commerce*, pp.209–218.

Hutter, K., Hautz, J., Füller, J., Mueller, J. and Matzler, K. (2011) 'Communitition: the tension between competition and collaboration in community-based design contests', *Creativity and Innovation Management*, March, Vol. 20, No. 1, pp.3–21

Kittur, A. (2010) 'Crowdsourcing, collaboration and creativity', *XRDS*, Winter, Vol. 17, No. 2, pp.22–26.

Lakhani, K.R., Garvin, D.A. and Logstein, E. (2010) 'TopCoder: developing software through crowdsourcing', Harvard Business School Case 610-032.

Leimesister, J.M., Huber, M., Bretschneider, U. and Krcmar, H. (2009) 'Leveraging crowdsourcing: activation-supporting components for it-based ideas competition', *Journal of Management Information Systems (JMIS)*, Vol. 26, No. 1, pp.197–224.

Li, W. and Li, N. (2012) 'A formal semantics for program debugging', *Science China – Information Sciences*, Vol. 55, No. 1, pp.133–148.

NASA Tournament Lab (2010) [online] http://www.topcoder.com/aboutus/archive/2010/10/nasa-to-crowdsource-software-development/ (accessed 10 June 2012).

Obrenovic, E., Gasevic, D. and Eliens, A. (2008) 'Stimulating creativity through opportunistic software development', *IEEE Software 2008*, Vol. 25, No. 6, pp.64–70.

Rapoport, A. and Chammah, A.M. (1966) 'The game of chicken', *American Behavioral Scientist*, Vol. 10.

Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B. and Pausch, R. (2005) *Design Principles for Tools to Support Creative Thinking*, Institute for Software Research, Paper 816.

Schenk, E. and Guittard, C. (2009) 'Crowdsourcing: what can be outsourced to the crowd, and why?', 7 December [online] http://halshs.archives-ouvertes.fr/docs/00/43/92/56/PDF/Crowdsourcing_eng.pdf (accessed 20 June 2012).

Shneiderman, B. (2007) 'Creativity support tools: accelerating discovery and innovation', *Communications of the ACM*, December, Vol. 50, No. 12, pp.20–32.

Shneiderman, B. (2011) 'Social discovery framework: building capacity and seeking solutions', *C&C '11 Proceedings of the 8th ACM Conference on Creativity and Cognition*, pp.307–308.

Tibbetts, H. (2012) 'Maximizing crowdsourcing success: strategies for minimizing risk and maximizing success', 13 September [online] http://hollistibbetts.sys-con.com/node/1975837 (accessed 12 June 2012).

Tsai, W.T., Cao, Z., Chen, Y. and Paul, R.A. (2005) 'Web services-based collaborative and cooperative computing', *ISADS 2005*, pp.552–556.

Tsai, W.T., Chen, Y., Paul, R.A., Liao, N. and Huang, H. (2004) 'Cooperative and group testing in verification of dynamic composite web services', *Proc. of IEEE COMPSAC Workshops*, pp.170–173.

Tsai, W.T., Zhou, X., Chen, Y. and Bai, X. (2008) 'On testing and evaluating service-oriented software', *IEEE Computer*, Vol. 41, No. 8, pp.40–46.

Notes

1 IBM Cleanroom methodology has evolved, and this paper focused a specific version only.