Software Economies

David F. Bacon[1], Eric Bokelberg[2], Yiling Chen[3], Ian A. Kash[3]
David C. Parkes[3], Malvika Rao[3], Manu Sridharan[1]

[1]IBM T.J. Watson Research Center, Yorktown Heights, NY
[2]IBM Global Business Services, Essex Junction, VT
[3]School of Engineering and Applied Sciences, Harvard University, Cambridge, MA
{dfb,ebokelb,msridhar}@us.ibm.com
{yiling,kash,parkes,malvika}@eecs.harvard.edu

ABSTRACT

Software construction has typically drawn on engineering metaphors like building bridges or cathedrals, which emphasize architecture, specification, central planning, and determinism. Approaches to correctness have drawn on metaphors from mathematics, like formal proofs. However, these approaches have failed to scale to modern software systems, and the problem keeps getting worse.

We believe that the time has come to completely re-imagine the creation of complex software, drawing on systems in which behavior is decentralized, self-regulating, non-deterministic, and emergent—like economies.

In this paper we describe our vision for, and prelimary work on, the creation of software economies for both open systems and internal corporate development, and our plans to deploy these ideas within one of the largest developer communities at IBM.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Reliability; D.2.10 [**Design**]: Methodologies; J.4 [**Social and Behavioral Sciences**]: Economics; K.4.4 [**Electronic Commerce**]: Distributed commercial transactions; K.6.3 [**Software Management**]: Software development

General Terms

Design, Economics, Reliability, Verification

1. INTRODUCTION

Software construction has often been described using metaphors like building bridges or cathedrals, which emphasize architecture, specification, central planning, and determinism. These engineering-based metaphors have gone hand-in-hand with a notion that a specification has a "correct" implementation. This absolute notion of correctness provides the underpinnings of both pragmatic and theoretical approaches to reaching this ideal: via testing disciplines or code review for the former, and manual or automated proof techniques for the latter.

Despite significant progress in both engineering- and mathematics-based approaches to correctness, these techniques have failed to scale to large systems. This was already true in 1968 with the identification of the "software crisis" [2], and 40 years of exponential growth in computing has greatly exacerbated the problem. Furthermore, as systems have moved from a Turing machine "input tape to output tape" model of computation to one of continuous interaction with other large systems and with the physical world, the creation of a complete specification of potential behaviors is often impossible. Coming ultra-large-scale systems [5] will likely present further correctness challenges.

In addition to quality issues, a major problem that bedevils the software industry is the creation of systems on time and on budget – going back to Brooks' "mythical man-month" [4]. In a traditional software organization with staffed resources, the labor market and the customer requirements are fairly fixed (developers are salaried, and requirements are often agreed upon in detail before development commences).

One extremely important aspect of making efficient use of developers and keeping costs low is to provide *accurate budgeting*. Note that inaccurate time and cost estimates are problematic whether they are too low or too high: if too low, the result is lost revenue and missed deadlines; if too high, the result is idle employees and contracts lost due to excessive pricing. For example, if work is overestimated by 10% on average, an organization with 100,000 software professionals would have the equivalent of 10,000 people idle – more than the size of Adobe and Facebook combined.

Another critical component of cost control is to promote continual and accurate sharing of information about the skills of workers, the needs of different projects, and progress being made or difficulties encountered. By sharing such information, appropriate prioritization can be made in assigning the right workers to the right tasks.

1.1 Software Economies

We argue that the time has come to completely re-imagine the creation of complex software, drawing on metaphors like biology, economics, and others in which behavior is decentralized, self-regulating, non-deterministic, and emergent.

In this paper we describe our vision for, and preliminary work on, a software development process based around the creation of economies in which supply and demand for fine- or coarse-grained work products drive the allocation of work and the evolution of the system. The methods of game theory and mechanism design are used to create a self-regulating system that drives the development process towards a dynamic equilibrium that maximizes overall utility. Also, by eliciting dollar values for software properties like quality, time to market, and performance, a market system can enable quantitative evaluation of classic tradeoffs (e.g., quality vs. time to market) and provide a useful basis for comparing different software metrics.

In our conceptualization of a *public software economy*, coalitions of users bid for features and fixes. Developers, testers, bug reporters, and analysts share in the rewards for responding to those bids. The traditional (unachievable) notion of absolute correctness is replaced by quantifiable notions of *correctness demand* (the sum of bids for bugs) and *correctness potential* (the sum of the available profit for fixing those bugs). The research problem then becomes one of game theory and mechanism design: creating a market in which the rewards constantly drive the system towards a *correctness equilibrium* in which all bugs or features for which there is enough value and low enough development cost are fixed or implemented [1].[1]

In a *private software economy*, managers estimate the resources required for development tasks and partition those tasks among an essentially fixed set of developers. For these cases, we can also apply ideas from *prediction markets*, to drive project managers to more accurately estimate project costs and schedules. Participants in a prediction market can trade in contracts that payout contingent on observable outcomes (e.g., “$1 if the release date is before April 10, $0 otherwise.") The trading price on such a contract can provide a useful estimate of the probability with which this event will occur [3, 13].

An internal market ecosystem can be created in which employees earn rewards based on the speed and quality of their work and the re-use and generation of reusable components, while project managers are rewarded for accurate prediction of work required for tasks. In addition, programmers can earn extra rewards when they collaborate or compete to perform excess work from other projects, therey reducing idle time between assigned tasks.

IBM has implemented an outcomes-based model for quantifying the delivery performance of developers working within the CIO-led organization where IBM’s internal IT initiatives are supported. This model provides game-based incentives for meeting and exceeding delivery targets. We are working to extend and improve this system based around market economy principles. Our initial goal is to design a system with an equilibrium in which the market incentivizes developers to work efficiently and managers to give accurate predictions of development time.

2. NASCENT SOFTWARE ECONOMIES

There are already several partial software markets in place. These systems serve both as illustrations of how market principles can be applied to software systems and as early indicators that market-based systems are already arising organically. Here we briefly review a few such systems.

Vulnerability Markets. One of the earliest software markets to emerge was for the potentially most expensive bugs: security vulnerabilities. Several market-based vulnerability reporting systems have been introduced with the goal of incentivizing users to report bugs and vulnerabilities. The *Mozilla Foundation* offers a cash award of $500 to anyone who reports a valid, critical security bug [8]. Schechter [11] proposes to use a vulnerability market in which the reward for reporting a security bug grows over time—the current reward places a valuation on the assurance that the software is free of vulnerabilities. Ozment [9] has demonstrated that vulnerability markets are essentially auctions for bug reports (in particular, an *open first-price ascending auction*).

Freelance Marketplaces. Online freelance marketplaces are platforms that connect individuals, small-business owners, and even Fortune 500 companies with freelance technology specialists to satisfy their technological needs. The sites provide vivid details about workers’ histories and qualifications, and some even feature tools that let the businesses monitor the work they are paying for [6]. *TopCoder* uses programming competitions to build professional-grade software outsourced by clients [12]. The rewards for a competition can vary based on the difficulty of the task, and only the top two contestants receive awards. On *Rent-a-Coder* [10], coders place bids on projects posted by buyers, and buyers chooses a suitable coder based on the bids and detailed coder profiles. Once the work is completed the buyers and sellers may rate each other, in contrast to TopCoder, where most of the reputation information is aggregated from directly measurable performance metrics.

App Stores. App stores also provide an an interesting example of software economies. For example, one can view the *iPhone* as a “system” and the apps as system components. When viewed in this manner, the *iTunes App Store* [7] allows different developers to compete to produce a winning module for the system. The existence of an effective micropayment system (most apps are 99 cents) is a substantial enabler—some applications have aggregated a significant profit via micropayments from a large user base. The store has a rating and review system—popular apps may have hundreds of reviews, including comments from users about outstanding bugs and desired features. The system serves as an important form of disintermediation which connects users directly with the developers of their portions of the overall system.

3. PUBLIC SOFTWARE ECONOMIES

In this section we discuss the application of market mechanisms to increasing software correctness and functionality in “public software economies” (as described in detail in our previous work [1]). Public economies involve projects with a direct connection to a large user base and would typically be organized around a single large-scale piece of software.[2] Our proposed system unifies many of the partial market-based mechanisms described in Section 2, incorporating bug reporters (as in vulnerability markets), feedback providers (as in the App Store) and developers and validators (as in freelance marketplaces). The system also *aggregates user demand* for fine-grained tasks like fixing a particular bug. We believe these properties can provide the necessary scale to bring about a fundamental change in software development.

3.1 Market Function

The key distinguishing aspect of the market that we envision is that users can bid for a bug fix or feature.[3] Bids can be as little as a penny, but the market aggregates demand from users (using either manual or automatic classification – itself a component of the market). Even very small bids may generate sufficient aggregate demand to make it worthwhile for a developer to satisfy them. Existing bug reporting systems do not elicit valuations from users for bug fixes; by doing so, the market provides users with a direct means to influence developer actions.

Briefly, the market comprises the following entities: a set of **Users** (individuals, corporations, etc.), a set of **Jobs**, a set of **Work-**

[1]We use the phrase ‘correctness’ in ‘correctness equilibrium’ not as a refinement on an equilibrium concept, as might be expected from game theory, but as a descriptive phrase to indicate the application to which equilibrium is applied.

[2]Both open-source systems like Mozilla Firefox and closed-source, single-supplier systems like Adobe Photoshop would be candidates, albeit with some differences in mechanism.

[3]Bug bids may be solicited for program crashes, as an additional alternative to the already common “Report to [Vendor]” dialog.

ers (which may overlap with the user set), and a set of **Kinds** which are used to categorize jobs (e.g. `correctness`, `feature`, `security`, `mac`, etc.). A user may offer a reward for a particular job at some time. A worker able to perform a job has an associated cost, namely their time and materials and including the opportunity cost.

The function of this market incorporates both the aggregation of user bids (like bug voting systems) and multiple competing workers (like TopCoder). This exchange structure, with information and preferences (e.g., costs for different kinds of work, values for different kinds of fixes) on both sides, is designed to provide for a more efficient market place. Of course, there are challenges in enabling such a market to function; e.g., demand for jobs must be accurately aggregated, successful completion of a task must be identified, and so on.

However, we can already consider how the system's performance can be characterized in some interesting quantitative ways. For a particular job, the demand is the sum of the individual user rewards bid for that job. For the complete software system, the *market demand* is the sum of of the demand for all of the jobs.

The **Kinds** can be used to evaluate the nature of the demand. For instance, the sum of all bids for jobs of kind `correctness` is the *correctness demand* for the system. Note that a correctness demand of 0 does not imply that there are no bugs in the software – just that there are no bugs to which any users attach value for fixing. This could mean that the software is flawless, or it could mean that nobody cares about using it. Analogously, we can quantify the demand for security, new features, support for a particular platform, and so on.

Intuitively, the jobs that are "worth doing" for workers are those where the cost of performing the work is less than the expected reward. We define the *potential value* of a job as the net reward that can be obtained by the worker who can perform it for the lowest cost (or 0 if the reward is less than their cost). Ideally, whenever the potential value of a job is greater than 0, the market should drive the work to happen.

The *market potential* is the sum of the potential values of all jobs, and the *correctness potential* is the sum of the potential values of all rewards whose kind is `correctness`. In *correctness equilibrium*, the correctness potential of a system is zero and all bugs that are "worth fixing" have been fixed. There may still be plenty of latent bugs, or even significant correctness demand, but there are no longer any bugs that a worker can fix without losing money.

3.2 Market Principles

The open question of course is how to design a market that operates in the manner we have just described. There are four fundamental principles that we have idenitified for the market design:

- **Autonomy**. All of the actions necessary to bring jobs to completion should be driven by market forces; the process is never gated by an entity outside of the market.
- **Inclusiveness**. Everyone who provides information or performs work that leads to improvements should share in the rewards.
- **Transparency**. The system should be transparent with respect to both the flow of money in the market and the tasks performed by workers in the market.
- **Reliability**. The system should be immune to manipulation, robust against attack (e.g., via insertion of untrusted code), and prevent "shallow" work which would have to be re-done later.

For there to be a market, there has to be a source of funding. Most obviously, as we have described, users can directly bid with their own funds. However, other funding models are possible: for example, a portion of the sale price paid by the user could be placed in escrow, with the user able to allocate the funds as she sees fit.

In all of these cases, the proposed market is funded with real money, which can be earned by those contributing to the software. The only differences are the degree to which the money in the market is fungible to the bidders. In particular, the sale price escrow model is applicable to completely closed source systems – and allows users to "tunnel through" the organizational barriers that separate them from developers.

4. PRIVATE SOFTWARE ECONOMIES

Here we describe the potential benefits of market mechanisms in "private software economies," in which the size and user base of the software is potentially smaller than in a public economy, but the number of software systems under development is much larger. In a private economy, market incentives have the potential to reduce costs, increase predictability, and provide insights on which processes and tools yield the greatest benefits in the development process.

Our main discussion is in regard to *scoring systems*, designed to incentivize desired behaviors at a fine-grained level (e.g., for each development task). But we also note that competition platforms can find a role in private economies, where otherwise idle developers can compete for some subset of tasks and additional compensation. A side benefit of this system is the information it yields on which developers are the top performers. A preliminary competition system named *Liquid* is already being piloted within IBM GBS.

4.1 Scoring Systems

A scoring system could directly reward faster task completion, greater use and development of reusable components, higher quality code, or more accurate predictions of the required resources for tasks, yielding better overall outcomes than the more coarse-grained evaluations typically employed today. Additionally, game theory can be used to identify potentially undesirable outcomes of a scoring system before it is put into use. Here, we describe some initial work on such a scoring system and show the benefits of bringing game theory to bear on the problem.

Outcomes Model. Our goal is to tune the outcomes-based incentive system currently applied to IBM's internal IT initiatives. Within the project delivery environment, software professionals (developers[4]) execute tasks assigned by their project managers (leaders) to produce project deliverables. In the outcomes model, each completed assignment has an associated "Blue Sheet" that records the original cycle time and effort planned for the task (as established by the leader), the actual completion time and effort that it required, a self-assessment of the deliverable quality against specified standards, and the extent that reuse of pre-existing assets was leveraged to complete the deliverable. The Blue Sheet parameters are used as inputs to compute scores for both practitioners and leaders, and top scorers are recognized for their achievement.

A Simplified Problem. In trying to create a scoring rule for the outcomes model with provable properties, we will start by making some simplifications. Initially, the only inputs to our system are estimated and actual effort – the amount of work time that the

[4]We use the term "developer" for brevity; incentivized scoring applies equally well to software designers, testers, etc.

developer spends on the task. Cycle time, quality, and reuse are temporarily ignored. We note that studying a system with this simplification complements the more typical approach of focusing only on software quality. As an additional simplification, we focus on devising a scoring system that need only satisfy the following:

- Developers should be incentivized to finish tasks as quickly as possible.
- Leaders should be incentivized to predict the time required for tasks as accurately as possible.

We can initially formalize the problem as a two-player sequential game between the leader and the developer. The leader "plays" first by estimating the effort ($\hat{e}$) that some development task will take. The developer "plays" second by completing the task, which results in an actual amount of effort applied (e). We denote the scoring function for leaders as $L(e, \hat{e})$ and the scoring function for developers as $D(e, \hat{e})$ (a higher score is better).

The following scoring functions satisfy the desired properties for the case of a single task:

$$D(e, \hat{e}) = 2\hat{e} - e \qquad\qquad L(e, \hat{e}) = e - (\hat{e} - e)^2$$

Independent of the leader's strategy, it is always optimal for the developer to work as quickly as possible since this minimizes effort e and maximizes $D(e, \hat{e})$. Let e^* denote this optimal effort. The exact value of e^* is private to the developer, but the leader has information on how e^* is distributed. Given the optimal strategy of the developer, it is optimal for the leader to predict $\hat{e} = \mathbb{E}[e^*]$ (where $\mathbb{E}[\cdot]$ is expectation) based on her information, as this estimate minimizes $\mathbb{E}[(\hat{e} - e^*)^2]$ and thus maximizes the leader's expected score. These strategies form a *Nash equilibrium*. Fixing the developer's strategy, the leader cannot benefit by deviating from her own strategy. Furthermore, because the developer's strategy strictly dominates all other strategies and the leader has a unique best-response, this is a unique Nash equilibrium.

Note the above analysis does not immediately hold when the game is *repeated* (where the developer and leader participate across multiple tasks). While the above strategies remain an equilibrium for the repeated game, care is needed to show that other undesirable equilibria don't arise.

Open Challenges. Our simplified game captures a vastly simpler problem than real-world development. We hope to extend the game to capture the following properties:

- **Optimizing assignment size**. As formulated, both the leaders and developers may be tempted to split reasonably-sized assignments into many smaller assignments: it may be easier to predict the time required for the small tasks, thereby increasing the leader scores but reducing the overall effectiveness of the development plan. The scoring rules should be enhanced to discourage this behavior.
- **Rewarding challenging assignments**. In our current game, a developer who completes a complex task in time t gets no additional reward over someone else who only completes an easy task in time t (assuming accurate estimated times). We'd like to add a "complexity" measure for tasks to suitably reward stronger developers.
- **Component reuse**. The overall cost of development may be reduced significantly if, rather than just being incentivized to complete tasks as quickly as possible, developers are rewarded for creating reusable components (to accelerate the work of others) and learning to reuse existing components (to accelerate their own future work).
- **Quality**. We'd also like to integrate a notion of quality in the game to promote an optimal time-quality tradeoff; for example, it would be desirable to reduce the overall reward to a developer who completes a task very quickly but introduces bugs that are only discovered later.

Additional issues include those of eliciting information from developers, who may have information that can help the leader to generate a more accurate estimate. We are concerned also about preventing collusion between developers and leaders, whereby coordinated deviations lead to higher total scores across repeated interactions but without improving development efficiency. We also seek to ensure that there are no new biases introduced where some tasks are *ex ante* more likely to lead to higher scores over a given time frame than other tasks.

Analyzing the above in a rigorous game-theoretic setting should provide new insights into fundamental tradeoffs and ways to improve the development process.

4.2 Assets and Meta-Assets

Another application of markets is to assign quantitative scores to various *assets* in a development organization, such as reusable components. Given detailed data on tasks and scoring systems tying the data to desirable global outcomes, one could devise statistical analyses to measure the historical value of assets, e.g., how the reuse of a component contributed to a good outcome and a good score.

Similarly, market mechanisms could enable quantative comparisons of tools aimed at *predicting* the future value of assets; we deem such tools *meta-assets*. Meta-assets could include code-level metrics like cyclomatic complexity along with the various other metrics tested for correlation with post-release defect density by the empirical software engineering community (test coverage, developer turnover, etc.). As the market's scoring system comes closer to capturing the actual monetary value of an asset, meta-assets that can accurately predict impact on scores become more valuable. Also note that given historical data, one can easily test new meta-assets and compare their predictive power to others.

5. CONCLUSIONS

We have proposed a re-imagining of the software development process in terms of decentralized, self-regulating economies. Such economies would drive market participants to game-theoretic equilibria that maximize the overall utility of the system, encompassing software quality, time to market, and other desirable features. The economies would also provide a quantitative basis for evaluating tradeoffs in the development process and the utility of various meta-assets like software metrics. We have sketched preliminary instantiations of these principles for public and private software economies and described some challenges that need to be addressed in order to make such economies a reality.

6. REFERENCES

[1] BACON, D., CHEN, Y., PARKES, D., AND RAO, M. A market-based approach to software evolution. *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications* (2009).

[2] BAUER, F. The software crisis. In *The 1968/69 NATO Software Engineering Reports* (1968).

[3] BERG, J. E., AND RIETZ, T. A. Prediction markets as decision support systems. *Information Systems Frontier 5* (2003), 79–93.

[4] BROOKS, F. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[5] FEILER, P., GABRIEL, R. P., GOODENOUGH, J., LINGER, R., LONGSTAFF, T., KAZMAN, R., KLEIN, M., NORTHROP, L., SCHMIDT, D., SULLIVAN, K., AND WALLNAU, K. *Ultra-Large-Scale Systems: The Software Challenge of the Future*. Carnegie Mellon Software Engineering Institute, June 2006.

[6] FLANDEZ, R. Help wanted – and found. *The Wall Street Journal* (October 2008).

[7] iTunes App Store website. *http://www.apple.com/iphone/apps-for-iphone/*.

[8] Mozilla website. *https://www.mozilla.org/security/bug-bounty.html*.

[9] OZMENT, A. Bug auctions: Vulnerability markets reconsidered. In *In Third Workshop on the Economics of Information Security* (2004).

[10] Rentacoder Inc. website. *http://www.rentacoder.com*.

[11] SCHECHTER, S. E. How to buy better testing: using competition to get the most security and robustness for your dollar. In *In Infrastructure Security Conference* (2002).

[12] Topcoder Inc. website. *http://www.topcoder.com*.

[13] WOLFERS, J., AND ZITZEWITZ, E. Prediction markets. *Journal of Economic Perspective 18*, 2 (2004), 107–126.