# Characteristics of Sustainable OSS Projects: A Theoretical and Empirical Study

Hideaki Hata*, Taiki Todo†, Saya Onoue*, Kenichi Matsumoto*
*Graduate School of Information Science
Nara Institute of Science and Technology, Japan
Email: {hata, onoue.saya.og0, matumoto}@is.naist.jp
†Graduate School and Faculty of Information Science and Electrical Engineering
Kyushu University, Japan
Email: todo@agent.inf.kyushu-u.ac.jp

***Abstract*—How can we attract developers? What can we do to incentivize developers to write code? We started the study by introducing the population pyramid visualization to software development communities, called *software population pyramids*, and found a typical pattern in shapes. This pattern comes from the differences in attracting coding contributors and discussion contributors. To understand the causes of the differences, we then build game-theoretical models of the contribution situation. Based on these results, we again analyzed the projects empirically to support the outcome of the models, and found empirical evidence. The answers to the initial questions are clear. To incentivize developers to code, the projects should prepare documents, or the projects or third parties should hire developers, and these are what sustainable projects in GitHub did in reality. In addition, making innovations to reduce the writing costs can also have an impact in attracting coding contributors.**

## I. Introduction

Maintaining and increasing the populations in software development communities are challenges of OSS projects for sustainability. Yamashita et al. proposed a pair of population metrics, namely, magnetism and stickiness [1]. Based on their measurements, they empirically studied OSS project histories, and found at-risk projects in sustainability. To increase the population, attracting newcomers is important. However, newcomers tend to face difficulties when joining OSS projects [2]–[4].

For understanding the characteristics of succeeded OSS projects in human aspects, we need to analyze human resources. To capture the distribution of developers' variations in experiences, we have proposed software population pyramid, which consists of coding contributors and discussion (non-coding) contributors [5]. Figure 1 shows an example of a software population pyramid.

From the study with GitHub dataset, we found that there is a typical pattern in shapes. This pattern comes from the differences of attracting coding contributors and discussion contributors. To understand the causes of the differences, we then built game-theoretical models of the contribution situation based on leader-follower games. The project is the leader and a developer is the follower, and the developer can choose coding or discussion after the project select keep the environment or setup or prepare something that can reduce the cost of writing

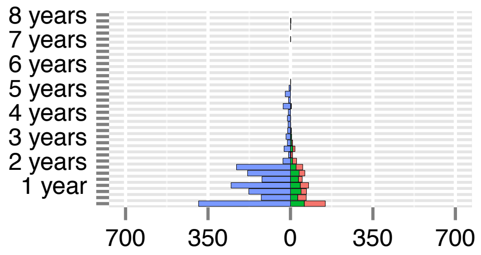


Fig. 1. An example of a software population pyramid. Coding contributors (green represents contributors moved from discussion contributions) on right and discussion (non-coding) contributors on left

code. Based on the models, we obtained the possible options to incentivizing developers to write code, that is, setup the development environment (for example, preparing documents) to decrease the cost of writing code, hire developers, and make innovations. From the empirical study, we found that more projects with a higher amount of code contributors prepared more documents and/or hired developers. In addition, the impact of GitHub can be regarded as the innovation to incentivize developers to code.

## II. Types of Software Population Pyramids

In this paper we targeted the GitHub data to analyze the OSS populations. We obtained the dataset provided by Gousios [6]. This dataset includes developers' activity history of 90 OSS projects. The projects were selected from the top-10 starred software projects for the top programming languages on GitHub[1].

Now we can create all the 90 software population pyramids from the dataset. In the entire projects, we remove projects that have short histories in GitHub because they are not mature to be analyzed nor have enough data compared with the rest. Newly started projects or projects moved to GitHub recently were ignored. We used 22 projects, which have more than 3-year activities in GitHub on December, 2012, to create those software population pyramids.

[1]MSR 2014 Mining Challenge Dataset: http://ghtorrent.org/msr14.html.

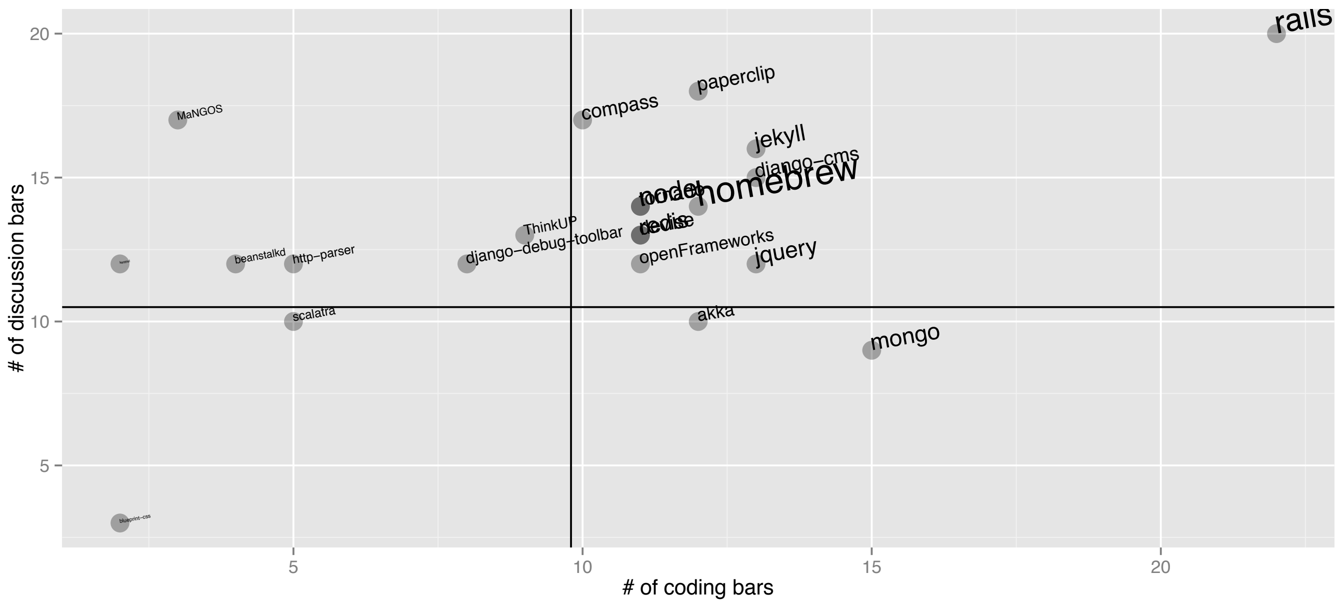


Fig. 2. Distribution of the software population pyramids. The size of the project name represents the number of coding contributors

The shapes of software population pyramids are different from each other, depending on the number of bars, the number of entire contributors, and so on. Since the studied 22 projects have more than three year histories in GitHub, their software population pyramids can have more than 10 bars for coding and discussion (there can be four bars in a year). The size of the project name represents the number of coding contributors. As seen in the figure, `rails` and `homebrew` have many coding contributors.

From Figure 2, we found that the projects have big variations, which means the shapes of the software population pyramids varies largely. We roughly classified those software population pyramids into four types with two lines shown in the figure.

(a) The software population pyramids at the upper left area have many discussion bars but not many coding bars. Although the projects have attracted many discussion contributors, they seem to fail to incentivize developers to code.
(b) The software population pyramids at the upper right area have both discussion and coding contributors. These projects can be considered to successfully attract many contributors including coding contributors. The `rails` project is separated from the others because this project has relatively long history in GitHub, and have many bars in the pyramid.
(c) The software population pyramids at the lower left area do not have many coding contributors nor many discussion contributors. The projects had failed to retain coding contributors in various generations.
(d) The software population pyramids at the lower right area do not have many discussion contributors but many coding contributors. Although the projects do not attract many casual developers who only contribute to discussion activities, they seem to have many coding contributors.

Since we are interested in incentivizing developers to code, the right part of the Figure 2, that is, (b) and (d), are considered to be successful in attracting and retaining coding contributors. From the following sections, we try to understand the key to this success, and explain why the software population pyramids form those.

## III. Game Theoretic Representation

We model a software development situation where there are two stakeholders, namely a project (or its core member community) and a software developer (or not core members), as an *extensive form game*. We first calculate the *subgame perfect Nash equilibrium* of the game, for both binary/continuous action spaces (in Sections III-B and III-C). We then introduce a third party that has an ability to bring an innovation, e.g., the invention of GitHub, to the situation, and discuss how such innovations could affect in the real software development situation (in Section III-D).

### A. Terms and Definitions

We begin with an (informal) introduction of several game-theoretic terms and definitions used in this section. We basically follow the explanations in Chapter 5 of [7].

A (perfect information) extensive form game is represented as a tree (in the sense of graph theory), which is also known as a *game tree*. Each non-terminal node in a game tree corresponds to the moment of the choice of one player, and each edge from a node corresponds to an action possible for the player at the moment. Each terminal (or leaf) node represents a final outcome, under which each player receives a utility. Finally we assume that every player knows all the information, including how much utility an adversary receives at a node. That is why we call this game *perfect information*.

In this section we focus on extensive form games in which there are only two players, namely the project and the developer, and whose game trees are of depth 2. Moreover, the project takes an action at the root node, and then the developer takes an action after seeing the project's choice. This class of games are known as *leader-follower* games, which have been attracting much attention for real-world security scheduling [8].

A *strategy* of a player is a complete specification of which deterministic action to take at every node corresponding to the player in the game tree. In our model, only the root node is corresponding to the project, so a strategy of the project just specifies which action it chooses at the root node. On the other hand, the number of nodes corresponding to the developer is the same as the number of actions possible by the project at the root node. Therefore, a strategy of the developer specifies which action it chooses after seeing each action by the project, regardless of whether or not the project chooses the action by a given strategy.

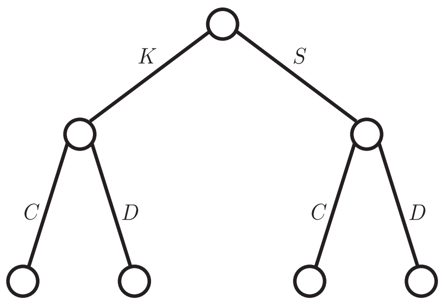
The concept of *subgame perfect Nash equilibrium* (SPNE) can be formally described as follows: for a given extensive form game, a profile of strategies is an SPNE if it induces a Nash equilibrium in every subgame of the original extensive form game. Intuitively, for our model, a pair of strategies of the project and the developer is an SPNE if (i) the developer chooses, at every corresponding node, the action that maximizes its utility and (ii) the project chooses the action that maximizes its utility at the root node under the choice of the above strategy by the developer.

From the intuition, now we can see that for a given extensive form game, an SPNE is computed by *backward induction*, that is, moving nodes from bottom to top with sequentially choosing the best action for the corresponding player at the current node. For a given SPNE, the *equilibrium path* is a sequence of actions from the root node to one of the terminal node specified by the SPNE. An equilibrium path is, in a sense, an expected outcome in the game.

## *B. A Simple Case with Binary Actions*

We first demonstrate how the software development situation could be analyzed based on the concept of SPNE. As an illustrative example, here we consider an extensive form game in which the project just chooses between making a new setup or keeping its current features. After observing the project's action, the developer chooses between writing code for the project and just making a discussion/comment on some codes. The game tree is given in Figure 3.

*1) Game Description:* At the root node (displayed on the top of the figure), the project chooses either Setup (S) or Keep (K). Then, the developer chooses either Code (C) or Discussion (D) at both of corresponding nodes (displayed on the middle). Each player's utility consists of a *benefit* (or



|  |  |  |  |  |
|---|---|---|---|---|
| developer | $\beta_{C,K} - \pi_{C,K}$ | $\beta_{D,K} - \pi_{D,K}$ | $\beta_{C,S} - \pi_{C,S}$ | $\beta_{D,S} - \pi_{D,S}$ |
| project | $b_C - p_K$ | $b_D - p_K$ | $b_C - p_S$ | $b_D - p_S$ |

Fig. 3. The extensive form game with binary actions and the players' utilities

reward) and a *price* (or cost). For the project, its benefit only depends on whether or not the developer chooses the action C. Specifically, the project receives $b_C$ if the developer chooses C and $b_D$ if the developer chooses D. On the other hand, the project's price only depends on its action. Choosing S and K have costs $p_S$ and $p_K$ for the project, respectively. It seems natural to assume that $p_S > p_k$.

In the real software development situation, the developer's benefit and price are a bit more intricately determined. So here we assume they depend on actions of both of the players, by introducing the following eight constants, $\beta_{C,S}$, $\beta_{D,S}$, $\beta_{C,K}$, $\beta_{D,K}$ for its benefit, and $\pi_{C,S}$, $\pi_{D,S}$, $\pi_{C,K}$, $\pi_{D,K}$ for its price. We assume $\pi_{C,S} > \pi_{D,S}$ and $\pi_{C,K} > \pi_{D,K}$, which reflects the natural trend in software development that writing code is more costly than just making a discussion for simplicity. Now we do not have clear evidence to support this assumption. All the utilities are summarized on the bottom of the figure. Notice that due to our motivation, we assume $\beta_{D,K} - \pi_{D,K} > \beta_{C,K} - \pi_{C,K}$, meaning that when there is no setup by the project, the developer chooses not to write code (at the middle-left node).

*2) Equilibrium Analysis:* We now calculate an SPNE of the game via backward induction. As we already see, the developer chooses D at the middle-left node. On the other hand, at the middle-right node, the developer chooses C if E1 holds

$$\beta_{C,S} - \pi_{C,S} > \beta_{D,S} - \pi_{D,S}, \tag{E1}$$

and D if E1 does not hold.

Assume E1 holds so that the developer chooses C at the middle-right node. Then, the project would receive utility of $b_D - p_K$ when it chooses D, while it would receive utility of $b_C - p_S$ when it chooses S. So the project chooses S in an SPNE if E2 holds

$$b_D - p_K < b_C - p_S, \tag{E2}$$

and K in an SPNE if E2 does not hold.

On the other hand, if E1 does not hold and thus the developer chooses D at the middle-right node, the project would choose K in an SPNE at the root node. This is because, now that it knows that the developer always chooses D regardless of its action, it compares $b_D - p_K$ and $b_D - p_S$. Since we already assumed $p_K > p_S$, the project prefers choosing K.

As a result, we can summarize the SPNEs by dividing them into three independent cases.

- E1 and E2: (S, (D,C))
- E1 but not E2: (K, (D,C))
- not E1: (K, (D,D))

Note that the first component indicates the strategy of the project, while the second component indicates the strategy of the developer. Also, in the second component, the former and the latter component corresponds to the action the developer chooses at the middle-left node and the middle-right node, respectively.

*3) Discussion:* Our main question is when the developer chooses C (i.e., write code) in an equilibrium path. From the equilibrium analysis presented in the previous subsection, the first is the only case in which the developer chooses C *on the equilibrium path*. On contrary, although the action C by the developer is also included in an SPNE for the second case, it is never realized on the equilibrium path.

*Observation 1:* For the game represented in Figure 3, the project chooses Setup and the developer chooses Code on the equilibrium path if and only if both E1 and E2 hold.

What is a natural interpretation of having both E1 and E2 in real software development situations? In many realistic situations it may be natural to assume $p_K = 0$. Therefore, E2 is equally described as

$$p_S < b_C - b_D,$$

meaning that the cost of making a new setup for the developer is smaller than the benefit from the immigration of the developer to writing code.

Also, the other condition E1 can be rewritten as follows:

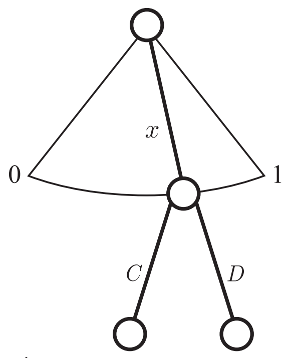
$$\beta_{C,S} - \beta_{D,S} > \pi_{C,S} - \pi_{D,S},$$

meaning that the gain by moving to write code is bigger than the loss by the move. In the following subsection we will conduct another case study for a more elaborated setting.

## *C. Considering Continuous Actions*

In realistic software development situations, it seems quite natural to think that the project has more than two actions, e.g., making a half-baked setup. Such an option is actually very realistic when the cost depends on the level of the action. In this subsection we consider continuous actions of the project, as a generalization of the discussion in the previous subsection.

*1) Game Description:* Let $x \in [0, 1]$ be the *level* (or quality) of setup being chosen at the root node by the project, in which a bigger $x$ means a better setup. For instance, choosing $x = 0$ corresponds to choosing K and choosing $x = 1$ corresponds to choosing S in the previous binary case. After observing the level $x$, the developer chooses between C and D, which is still a binary decision. The game tree can be drawn as Figure 4.



| | | |
|---|---|---|
| developer | $\beta_C(x) - \pi_C(x)$ | $\beta_D(x) - \pi_D(x)$ |
| project | $b_C - p(x)$ | $b_D - p(x)$ |

Fig. 4. The extensive form game with continuous actions

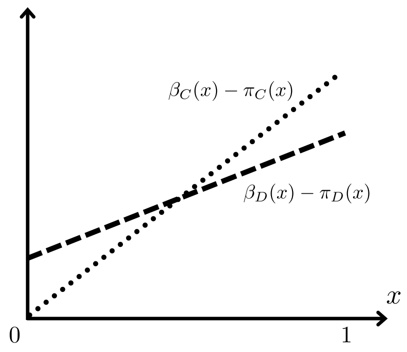


Fig. 5. Example of two curves satisfying condition F1

All the constants that were affected from the choice between K and S in the previous binary case are now going to be re-defined as functions on $x$. The price of the project is given as a function $p$ that is increasing on $x$, and the benefit/price of the developer are given as $\beta_C(x)$, $\beta_D(x)$, $\pi_C(x)$ and $\pi_D(x)$. We still focus on the case that $\beta_D(0) - \pi_D(0) > \beta_C(0) - \pi_C(0)$, meaning that under no setup, the developer never chooses C. For technical reasons, we also assume that $\beta_C(x)$, $\beta_D(x)$, $\pi_C(x)$, and $\pi_D(x)$ are continuous.

*2) Equilibrium Analysis:* Even though there are infinitely many nodes in this continuous model, we can still apply the idea of backward induction to calculate an SPNE. For any given level $x$ chosen by the project, the developer chooses C if $\beta_C(x) - \pi_C(x) > \beta_D(x) - \pi_D(x)$, and chooses D otherwise. Let us refer this best response strategy of the developer as BR.

Since all $\beta_C$, $\beta_D$, $\pi_C$, and $\pi_D$ are continuous and we assume $\beta_D(0) - \pi_D(0) > \beta_C(0) - \pi_C(0)$, either of the followings must be the case:

| | |
|---|---|
| F1: | The two curves $\beta_C(x) - \pi_C(x)$ and $\beta_D(x) - \pi_D(x)$ have at least one cross point during the domain $x \in [0, 1]$ as shown in Figure 5. |
| not F1: | For any $x \in [0, 1]$, $\beta_D(x) - \pi_D(x) > \beta_C(x) - \pi_C(x)$. |

If above F1 is the case, then we can find a smallest $x$ such that $\beta_C(x) - \pi_C(x) = \beta_D(x) - \pi_D(x)$ holds. Let us refer such $x$ as $x^*$. Intuitively, $x^*$ is the smallest level of setup under which the developer chooses C. Notice that $x^* > 0$. Then, the project compares $b_C - p(x^*)$ and $b_D - p(0)$. If

$$b_C - p(x^*) > b_D - p(0) \tag{F2}$$

Then, the project chooses $x = x^*$ if above F1 and F2 simultaneously occur, and chooses $x = 0$ if either (i) F1 but not F2 or (ii) not F1 occurs.

As a result, SPNEs are described as follows:

- F1 and F2: ($x^*$, BR)
- F1 but not F2: (0, BR)
- not F1: (0, D)

*3) Discussion:* As in the case of binary actions, we consider when the developer chooses C on the equilibrium path. We have just seen that when F1 is not the case, the developer never chooses C. On the other hand, when F1 is the case, the developer chooses C on the equilibrium path if F2 also holds.

*Observation 2:* For the game represented in Figure 4, the project chooses the level $x^*$ of setup and the developer chooses C on the equilibrium path if and only if F1 and F2 holds.

For instance, consider the following linear (or constant) benefit/price functions:

$$b_C = 1, b_D = 1/4$$

$$p(x) = x, \beta_C(x) = x, \beta_D(x) = \frac{1}{2}x + \frac{1}{4}$$

$$\pi_C(x) = -\frac{1}{4}x, \pi_D(x) = \frac{1}{8}$$

In this case, F1 holds since the two curves cross at $x = 1/2$. Then $x^* = 1/2$. Furthermore, at $x^*$, the project's utility is 0.5, which is strictly greater than $1/4$, the utility when it chooses $x = 0$. Thus the condition F2 also holds. An SPNE is (0.5, BR), and on the equilibrium path the project chooses $x^* = 0.5$ and then the developer chooses to write code.

### D. Innovation vs. Mandatory

We now consider a third-party who has an ability to bring an innovation to the software development situation. The purpose of this subsection is to search for a way to incentivize both the project and the developer to work coordinately, even when on the equilibrium path the project and the developer are choosing K and D, respectively. An innovation brought by the third party reduces costs of both the project $p$ and the developer $\pi$. At the same time, the developer's gross benefit may be increased by the innovation, but as the first step we ignore the effect for the benefit functions.

When F1 does not hold, reducing the cost of the developer for writing code could shift up so that there is at least one point in which the two curves are crossing. Also, when F1 holds but the additional condition F2 does not hold, reducing the cost of the project incentivize it to choose $x = x^*$ instead of $x = 0$, which will lead a SPNE in which the project brings (some level of) setup and then the developer commit on writing code on the equilibrium path.

On the other hand, there might also be the situation where writing code is *mandatory* for the developer. For instance, developers in a company may be in charge of an open source software due to their contract. Such situation can be represented in our game theoretic model by defining $\pi_D(x) = \alpha$ with sufficiently large constant $\alpha \in \mathbb{R}$. Then, $\beta_C(x) - \pi_C(x) > \beta_D(x) - \pi_D(x)$ always holds, meaning that regardless of the level of setup being chosen by the project, the developer writes code. Thus an SPNE is represented as follows:

- The project always chooses $x = 0$, i.e., no effort on Setup
- The developer chooses C for any $x \in [0, 1]$

Actually, from the viewpoint of the project, there is no incentive to make any new setup into the project, because it knows that the developer always writes code even if there would be no new setup, and so choosing less setup gives better utility for itself. Then, on the equilibrium path, no setup will be implemented, which tends to result in a bad quality of software development environments.

### E. Summary

From the analysis of game-theoretical models, the followings are the options to incentivize developers to write code:

**Setup**: To increase the utility of writing code compared to the utility of just discussing, projects need to setup the development environment, which can decrease the cost of writing code.

**Mandatory**: Employment is a big incentive to write code. The project itself or other third-parties can select this option.

**Innovation**: Although it is not easy to make innovations, innovations can decrease the cost and may increase the reward. For example, developing new tools like `Git`, a version control system, and deploying new services like `GitHub`, a web-base hosting service and social networking system for developers, can be regarded as such innovations.

## IV. Empirical Analysis

We analyzed the studied projects to find empirical evidence that supports the results of game-theoretical analysis. We conducted empirical analysis to see the impact of setup, mandatory, and innovation.

### A. Setup Coverage

In software development, preparing document is helpful setup [9]. We list the following five setup candidates, and survey the coverage (there can be other helpful documents like `contribute.md`).

- *Wiki*: The project have prepared wiki in the GitHub page.
- *Website*: The project have its own webpage outside the GitHub service.
- *How to contribute*: There is a document for contributing code, how to contribute, in the wiki or other official website.
- *Coding guideline*: There is a document for writing code, including style guides and patch guides.

TABLE I
COVERAGE OF SETUP

| Project | Wiki | Web | How to | Guide | Multi | # of y |
|---|---|---|---|---|---|---|
| (b) | | | | | | |
| rails | n | **y** | **y** | n | n | 2 |
| jekyll | **y** | **y** | **y** | n | n | 3 |
| django-cms | n | **y** | n | n | n | 1 |
| jquery | **y** | **y** | **y** | **y** | n | 4 |
| paperclip | **y** | **y** | **y** | n | **y** | 4 |
| homebrew | **y** | **y** | **y** | **y** | **y** | 5 |
| nde | **y** | **y** | **y** | n | n | 3 |
| tornade | **y** | **y** | n | n | n | 2 |
| devise | **y** | **y** | **y** | n | n | 3 |
| redis | **y** | **y** | n | n | n | 2 |
| openFrameworks | **y** | **y** | n | n | n | 2 |
| compass | **y** | **y** | **y** | **y** | n | 4 |
| (d) | | | | | | |
| mongo | n | **y** | **y** | **y** | n | 3 |
| akka | n | **y** | n | n | n | 1 |
| (a) | | | | | | |
| ThinkUP | **y** | **y** | n | n | n | 2 |
| django-debug | | | | | | |
| -toolbar | **y** | **y** | n | n | n | 2 |
| http-parser | n | n | n | n | n | 0 |
| beanstalkd | **y** | **y** | n | n | n | 2 |
| MaNGOS | n | **y** | n | n | n | 1 |
| kestrel | **y** | **y** | n | n | n | 2 |
| (c) | | | | | | |
| scalatra | n | **y** | n | n | n | 1 |
| blueprint-css | **y** | **y** | n | **y** | n | 3 |

- *Multi-language document*: There are other language documents in official, which should be beneficial.

Table I summarizes the result of the setup coverage. Note that the software population pyramids are created with data in 2012, but this survey is conducted in 2014. So the coverages may be different from 2012. The projects are classified into 4 as discussed in Section II. The result is relatively clear. The projects in (b), which have balanced software population pyramids with many coding and discussion contributors, have many y's (yes), that is, there are various helpful documents in such popular projects. The projects in (a) and (c), which do not have many coding contributors, tend to provide only a few documents. Especially, there is no "How to Contribute" documents. This result indicates that setting up useful documents can be the key of attracting coding contributors. And this may be the reason of what makes the differences between the software population pyramids of (a) and (b), only (b) have many coding contributors although both attract developers.

### B. Employment

The projects in (d) seems strange because there are not so many discussion contributors but many coding contributors. In addition, the project `akka` does not prepare enough documents to reduce the writing cost as shown in Table I. However, the reason of this is clear: both projects are owned by companies. When we see developers as paid if their GitHub account belong to any organization, there are 12 paid developers out of 34 on 2012/12 for `akka`, and 25 paid developers out of 116 on the same period for `mongo`. Since we counted paid developers roughly, the correct number may be different. For example, some developers belonging to organizations may contribute to the projects voluntarily. However, the existence of paid developers in this two projects is clear, and this may be the reason not to attract many developers for discussion and coding.

### C. The Impact of Innovation

The impact of *Social Coding* introduced by GitHub has attracted researchers [10], [11]. Starting the service of GitHub can be regarded as innovation in the meaning of our models.

Figure 6 presents the transition of software population pyramids of the `rails` project from December 2007 to December 2010. Note that the scales of the x-axis are not the same. At the initial snapshot (2007/12), there are only eight coding contributors although the project had more than three year histories. Since the project had no data in GitHub at that time, we cannot show the discussion contributors. The project moved to GitHub on August 2008[2]. After that first the project attracted discussion contributors, and lately it seems that it successfully attracted coding contributors. Although it is difficult to distinguish other factors, the project attracted coding contributors incomparably after it moved to GitHub.

## V. DISCUSSION

### A. Limitations of Theoretical Models

**Bounded rationality**. Some readers may claim about the applicability of game theory into the real human environment, because humans are known not to be rational in general. Nevertheless, we would like to emphasize that our approach based on game theory is quite important for analyzing human behaviors in real life. One of the main reasons is that we need such mathematical models for several scientific situations, such as estimating market dynamics or simulating crowd behavior in emergency situations. Furthermore, having a theoretical foundation enables us to easily extend the result of the analysis to a bit different situations, such as a new market with a slight modified pricing rules and/or social laws.

**Too much simplification**. Another possible criticism for our model is its immoderate simplification. We absolutely agree with that point, but at the same time we also believe that simplicity is a pro, rather than a con. Actually, even in our simple model, some new theoretical findings are revealed, as we already discussed in the previous section. Introducing an elaborate model is an obviously meaningful future direction.

**Future direction**. Although we obtained a bunch of new findings by focusing on SPNE, it must be very interesting to take into account different equilibrium concepts, such as Nash equilibrium and/or sequential equilibrium. We should carefully

[2]Rails Moving to Git, https://github.com/blog/32-rails-moving-to-git.

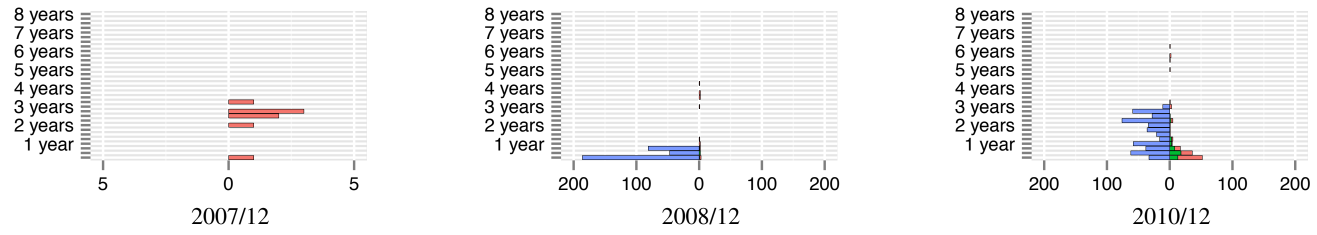


Fig. 6. The transition of software population pyramids of the `rails` project. This project moved to GitHub on 2008/4

choose one of them depending on the level of players rationality. Another possible direction is presuming that the third party considered in the paper is a *mechanism designer*, which could determine the market rule. The discussion with mandatory coding is one of such example with a bad mechanism. By doing so, we can naturally handle with the participation of more than one player, which enables us to consider several features in real market, such as competition between players.

Beside these two directions, there are several possibilities on modeling software development situations via game theory and microeconomics theory. For instance, we could investigate the way to achieve sustainable cooperation between developers based on repeated games, incentivize developers to participate based on online/dynamic mechanism design, analyze the population dynamics based on segregation model, and make a decision for a new release of software based on voting theory.

### *B. Threats to Validity*

**Limited dataset**. In our study, we studied 22 OSS projects in GitHub. Although there seem to be varieties of projects in the dataset, the dataset can be biased to popular projects. Therefore there can be a threat of generalization of results. Since the history of GitHub is not long, collecting a lot of project data enough for population analysis is not easy now. If we have more data, we can mitigate this threat by analyzing various project data.

**Analysis results may have error**. Although we analyzed documentation manually, the results may have errors. We might think that there are no documents although there are. In addition, there may be other important setup that can affect developers to write code. We have not checked whether the documents were up-to-date or not. Outdated documentation can harm the newcomers. Identifying paid developers may have errors too.

## VI. Conclusion

This paper studied the characteristics of sustainable OSS projects by a theoretical and an empirical analysis. We consider projects that can attract and retain coding contributors as sustainable projects. Based on game-theoretical models, we obtained the possible options to incentivize developers to write code, that is, setup the development environment like documentation to decrease the cost of writing code, employ developers, and make innovations. From the empirical study, we found that the projects that can successfully attract coding contributors have prepared documents, and/or employed developers. In addition, the impact of GitHub can be regarded as the innovation to incentivize developers to code.

## Acknowledgment

This study has been supported by JSPS KAKENHI Grant Number 26540029, and has been conducted as a part of Program for Advancing Strategic International Networks to Accelerate the Circulation of Talented Researchers.

## References

[1] K. Yamashita, S. McIntosh, Y. Kamei, and N. Ubayashi, “Magnet or sticky? an oss project-by-project typology,” in Proc. of MSR ’14, 2014, pp. 344–347.

[2] I. Steinmacher, I. Wiese, A. Chaves, and M. Gerosa, “Why do newcomers abandon open source software projects?” in Proc. of CHASE ’13, 2013, pp. 25–32.

[3] C. Hannebauer, M. Book, and V. Gruhn, “An exploratory study of contribution barriers experienced by newcomers to open source software projects,” in Proc. of CSI-SE ’14, 2014, pp. 11–14.

[4] I. Steinmacher, I. S. Wiese, T. Conte, M. A. Gerosa, and D. Redmiles, “The hard life of open source software project newcomers,” in Proc. of CHASE ’14, 2014, pp. 72–78.

[5] S. Onoue, H. Hata, and K. Matsumoto, “Software population pyramids: The current and the future of oss development communities,” in Proc. of ESEM ’14, 2014, pp. 34:1–34:4.

[6] G. Gousios, “The ghtorent dataset and tool suite,” in Proc. of MSR ’13, 2013, pp. 233–236.

[7] Y. Shoham and K. Leyton-Brown, *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, Cambridge University Press, 2008.

[8] M. Tambe, *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*, 1st ed, Cambridge University Press, 2011.

[9] I. Steinmacher, A. Chaves, T. Conte, and M. Gerosa, “Preliminary empirical identification of barriers faced by newcomers to open source software projects,” in Proc. of SBES ’14, Sept 2014, pp. 51–60.

[10] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, “Social coding in github: Transparency and collaboration in an open software repository,” in Proc. of CSCW ’12, 2012, pp. 1277–1286.

[11] J. Tsay, L. Dabbish, and J. Herbsleb, “Influence of social and technical factors for evaluating contribution in github,” in Proc. of ICSE ’14,