

A Game-Theoretic Approach to Software Process Improvement

Carlos Gavidia-Calderon



A dissertation submitted in fulfillment
of the requirements for the degree of

Doctor of Philosophy
of
University College London

Department of Computer Science
University College London

15 April 2020

Declarations

I, Carlos Gavidia-Calderon, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis. Parts of this document are based on the following publications:

1. Chapters 2 and 3 are based on: Carlos Gavidia-Calderon, Emmanuel Letier and Earl T. Barr. A Survey of Game Theory Applied to Software Engineering. In progress.
2. Chapter 4 is based on: Carlos Gavidia-Calderon, Federica Sarro, Mark Harman and Earl T. Barr. Game-Theoretic Analysis of Development Practices: Challenges and Opportunities. Journal of Systems and Software. 2019.
3. Chapter 5 and 6 are based on: Carlos Gavidia-Calderon, Federica Sarro, Mark Harman and Earl T. Barr. The Assessor's Dilemma: Improving Bug Repair via Empirical Game Theory. IEEE Transactions on Software Engineering. 2019.

This research project is a collaborative enterprise. While I led this work, I did it in close collaboration with my supervisors. I use "I" throughout this thesis in this sense.

Date

Signature

Abstract

There is a plethora of software development practices. Practice adoption by a development team is a challenge by itself. This makes software process improvement very hard for organisations. I believe a key factor in successful practice adoption is proper incentives. Wrong incentives can lead a process improvement effort to failure.

I propose to address this problem using game-theory. Game theory studies cooperation and conflict. I believe its use can speed the development of effective software processes. I surveyed game-theory applications to software engineering problems, showing the potential of this technique. By using game-theoretic models of software development practices, we can verify if the behaviour at equilibrium converges towards team cooperation.

Modern software development is performed by large teams, working multiple iterations over long periods of time. Classic game representations do not scale well to model such scenarios, so abstraction is needed. In this thesis, I propose GTPI (game-theoretic process improvement), a software process improvement approach based on empirical game-theoretic analysis (EGTA) abstractions. EGTA enables the production of software process models of manageable size.

I use GTPI to address technical debt, modelling developers that prefer quick and cheap solutions instead of high-quality time-consuming fixes. I have also approached bug prioritisation with GTPI, proposing the *assessor-throttling* prioritisation process, and developing a tool to support its adoption.

Impact Statement

Software is part of many aspects of modern life: from trivial things like ordering takeaway to vital tasks like aeroplane navigation. Software quality does not only depend on the technical aspects of engineering, but also on the process that drives how people build it. In the words of Linus Torvalds, "All the really stressful times for me have been about process: they haven't been about code".

In this thesis, I propose game theory as a formal approach for the analysis and design of software processes. Game theory is widely adopted in fields like economics, biology and cybersecurity. I bring game theory to software process design, using it to identify and remove incentive problems.

I used game theory to resolve real problems affecting software teams in industry. Both technical debt and priority inflation are still relevant problems in software development. I recommended actionable process interventions for their removal. I used game-theoretic models to confirm the effectiveness of established processes — like code reviews — and to challenge the adequacy of others — like bug triage teams.

Finally, I have established the foundations of a new exciting research field. I believe that technical debt and priority inflation are two of the many incentive problems that affect software processes. In this thesis, I developed a framework to address these problems with game-theoretic models. I expect that both practitioners and researchers adopt this novel approach of software process improvement.

Acknowledgements

First, I would like to thank my supervisory team at UCL: Dr Earl Barr, Dr Federica Sarro and Prof. Mark Harman. And Paul Baker, my mentor during my time at Visa Europe. The path from practitioner to academic is not an easy one, and it was your guidance that made it possible.

I would also like to thank my fellow PhD students: DongGyun Han, Matheus Paixao, Chaiyong Ragkhitwetsagul and Bobby Bruce. The PhD was definitely more enjoyable by sharing it with you. And also my friends at Ifor Evans Hall: Hillary Ingram, Silvia Schmidt, Lauren Coyne, Vibhav Mishra, Razvan Marinescu, Robbie Wilson and Mariflor Vega. Ifor Evans was my very first home-away-from-home, thanks to your friendship.

I also like to thank Dr Angel Gavidia and Lilia Calderon, my parents, and Dr Jorge Gavidia and Mario Gavidia, my brothers. Leaving Peru to pursue my academic dream was only possible with your help. It is an honour to become the third "Doctor Gavidia".

Finally, I would like to especially thank Giovanni Guizzo. In the good times, you were always up for a pint; and in the bad ones, I could always count on your support. And to Ireen Islam. I came to this country for a Doctorate, and I ended up finding you. For me, that is even more valuable.

Contents

1	Introduction	14
1.1	Research Problem	15
1.2	Objectives	17
1.3	Contributions	17
1.4	Thesis Outline	18
2	Background	20
2.1	Concepts and Terminology	20
2.2	Normal-form Games	22
2.3	Extensive-form Games	23
2.4	Multistage Games	26
2.5	Bayesian Games	27
2.6	Mechanism Design	28
2.7	Cooperative Game-Theory	30
3	Literature Review	32
3.1	Methodology	33
3.2	Game Models in Software Engineering	35
3.2.1	Software Requirements	36
3.2.2	Software Design	37
3.2.3	Software Construction	37
3.2.4	Software Testing	38
3.2.5	Software Maintenance	39
3.2.6	Software Engineering Management	41
3.2.7	Software Engineering Professional Practice	43
3.3	Challenges and Opportunities	44
3.3.1	The Need for Game Abstractions	44
3.3.2	Beyond the Rationality Assumptions	48

Contents

4	GTPI: A Game-Theoretic Approach to Process Improvement	50
4.1	Motivating Example	52
4.2	An Introduction to GTPI	54
4.2.1	Software Process Modelling	54
4.2.2	Software Process Improvement	59
4.3	Practical considerations	61
4.3.1	Data Gathering	62
4.3.2	Technical Validation	63
4.3.3	Securing Acceptance	64
4.4	Related Work	65
5	TaskAssessor: A Game-Theoretic Model of Priority Inflation	67
5.1	The Assessor's Dilemma	69
5.2	Identifying the Process Anomaly	71
5.2.1	Shared Prioritisation Tooling Adoption	72
5.2.2	The Cost of Priority Inflation	73
5.3	Empirical Game Design	77
5.3.1	Bug Repair and Issue Resolution Corpus	78
5.3.2	Game Models with <i>TaskAssessor</i>	80
5.3.3	<i>TaskAssessor</i> under Twins and EGTA	82
5.3.4	Validating <i>TaskAssessor</i>	85
5.3.5	Threats to Validity	88
5.3.6	Using <i>TaskAssessor</i>	89
6	Assessor-Throttling: A Novel Task Prioritisation Process	91
6.1	Empirical Game Improvement	93
6.1.1	Distributed Bug Prioritisation	94
6.1.2	Do Gatekeepers Prevent Priority Inflation?	95
6.1.3	The Assessor-Throttling Process	98
6.2	Process Deployment	106
7	Conclusion and Future Work	111
7.1	Summary of Contributions	111
7.2	Future Work	112
7.3	Final Remarks	113
	Bibliography	114

List of Figures

2.1	Normal-form representation of a two-player game (DEV1 and DEV2), where each player has two actions (cooperate and oppose). At equilibrium, both players adopt the opposing action with a probability of 100%.	21
3.1	Scope of this review with respect to the SWEBOK guide [1]. For this literature review, we only consider papers focussing on <i>practice core knowledge areas</i>	34
3.2	EGTA-abstracted game for a 2-player-3-round rock-paper-scissors game: each round victory is rewarded with 1 point and draws give no points to either player.	46
4.1	Extensive form game for a multi-project freelancer’s dilemma: this figure contains the part of the tree corresponding to the first two projects. Nodes in the tree correspond to players (Freelancer <i>A</i> and Freelance <i>B</i>) and edges to actions (<i>C</i> for cooperation, and <i>NC</i> for no cooperation) The size of this representation grows with the number of projects, freelancers and actions available to them. EGTA is crucial for managing this explosive growth (section 4.2).	53
4.2	The GTPI approach starts by detecting a candidate process that exhibits an aberrant behaviour. Such a process is then modelled using EGTA and the Nash equilibrium is obtained. In case the equilibrium is not the one desired, the EGTA model is updated iteratively until a desirable one is obtained. Finally, the improved process is adopted by the team.	55

List of Figures

4.3	Empirical game design: the process engineer builds a heuristic payoff table using heuristic strategies as actions: The entries of this table are expected payoff values obtained via simulation. They can later use any algorithm to calculate the NE of the empirical game, once the payoff matrix has been built.	56
4.4	Developers in the technical debt simulation: the implementation of work items can be fixes or kludges. Fixes demand more time but are less likely to require rework. Kludges are quick but are more likely than fixes to be reworked. Also, kludges negatively impact codebase health.	58
4.5	The tragedy of the test suite: job insecurity makes software engineers deliver more features without removing potentially useless tests. Over time, this behaviour can cause the collapse of the test infrastructure.	62
5.1	GTPI's software process modelling phase for priority inflation: to identify the process anomaly, we conducted a developer survey and a study on prioritisation in GitHub labels. <i>TaskAssessor</i> , the empirical game model, was built based on a bug resolution corpus from the Apache Software Foundation.	68
5.2	Role in the development process: the horizontal axis represents the number of participants per role. The survey allowed the selection of multiple roles per participant. This figure represents the answers of 152 software professionals.	74
5.3	Empirical game design stage for <i>TaskAssessor</i>	77
5.4	Calculating values for a Twins Player Reduction payoff matrix: one agent of the population is assigned strategy A, while the others are assigned strategy B in the simulation. The value to include in the matrix for strategy A against strategy B, is the average payoff of the single agent over multiple simulation iterations.	86
5.5	Validating <i>PlayGame's</i> simulation model. We split the process dataset in three parts: 1) the training dataset is used to obtain simulation parameters, 2) the validation dataset is used for model calibration, 3) and the testing dataset is used to assess the simulation output.	86

List of Figures

6.1	GTPI’s software process improvement phase for priority inflation: during software process improvement, we use <i>TaskAssessor</i> to model current task prioritisation practices. Finding them susceptible to priority inflation, we propose the assessor-throttling process as a solution. To facilitate assessor-throttling deployment, we developed TheFed. It is a Chrome plugin that connects to JIRA tracking systems.	92
6.2	Bug prioritisation processes: the blue components are shared between the two processes (including distributed prioritisation), the red ones are exclusive to the gatekeeper process . The solid lines represent the input/output of an activity, and the dashed lines link an activity with its performing role.	93
6.3	<i>PlayGame</i> ’s input-output transformation: QA engineers place reports in the development queue. Developers in the team then build patches to address each report. In a full-bandwidth scenario D , all developers are available for bug fixing duties. In a reduced bandwidth scenario $\frac{D}{2}$, only half of them are active	94
6.4	The assessor-throttling prioritisation process: after fixing a bug, the fixer assess the priority made originally by the reporter. This assessment impacts the reporter’s position in the reputation’s ranking, that determines which bugs are fixed first.	100
6.5	Equilibrium profiles for prioritisation processes at the reduced bandwidth scenario $\frac{D}{2}$. From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).	103
6.6	Equilibrium profiles for prioritisation processes at the full bandwidth scenario D . From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).	104
6.7	Performance comparison of task prioritisation processes in the reduced bandwidth scenario $\frac{D}{2}$. From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).	106

List of Figures

6.8	Performance comparison of task prioritisation processes in the full bandwidth scenario D . From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).	107
6.9	<i>TheFed's</i> prioritised inbox: JIRA issues are sorted by QA engineer's reputation.	108
6.10	<i>TheFed's</i> QA engineer's ranking by reputation score R^r	109

List of Tables

3.1	Publications using game-theoretic models to address software engineering problems	36
4.1	Pay-off matrix for the freelancer’s dilemma: each cell contains the payoff in dollars obtained by Freelancer A and Freelancer B given their choice to cooperate or not.	52
4.2	Input and output variables of the simulation model of technical debt: the output variable F_i corresponds to the payoff function of developers. The process engineer needs to work with the customer to identify the relevant variables of the process under analysis.	57
4.3	Payoff matrix after the empirical game design stage in Figure 4.2: it has a single Nash equilibrium where developer A and developer B adopt only the kludge-intensive heuristic strategy.	60
4.4	Payoff matrix after the empirical game improvement stage in Figure 4.2: it has a single Nash equilibrium where developer A and developer B adopt only the fix-intensive heuristic strategy.	60
5.1	Pay-off matrix for the assessor’s dilemma: each cell is the payoff Alice (A) and Bob (B) obtain, under the combination of actions each takes.	70
5.2	The questionnaire presented to 152 software engineers.	75
5.3	Survey responses to questions about the frequency of priority inflation and deflation in the respondent’s current software project.	76
5.4	The <i>TaskAssessor</i> Corpus of Issues extracted from JIRA and GitHub.	78

List of Tables

5.5	Strategy catalogue S for the task prioritisation game. P_I and P_D represent the conditional probabilities that a QA engineer inflates or deflates an issue.	80
5.6	$PlayGame$'s input variables. Random variables are sampled during a run until the number of bugs fixed equals N_f . The nonrandom variables are constant during a simulation run. . .	83
5.7	Pay-off matrix $TaskAssessor$ builds: since it is symmetric, the game has only two players ($Twin_1$ and $Twin_2$) and both player has $ S $ actions. Algorithm 2, $Twins$, computes the payoff for each pair of actions for each cell.	84
6.1	Pay-off matrix for the assessor's dilemma using assessor throttling.	101

1 Introduction

Software development practices are constantly evolving. From the spiral model in the 80s to the current dominance of modern agile software development, practitioners are looking for the best ways to organise their work. Methodologists nowadays are advocating practices, like pair programming and test-driven development, with the aim of improving productivity and boosting quality. These practices improve software development once adopted; however adoption itself can be challenging [2]. Pair programming is pointless if developers refuse to share a workstation, and test-driven development benefits evaporate if developers write tests after implementation. For successful practice adoption, *cooperation* among process performers is essential.

Game theory is “the study of mathematical models of conflict and cooperation” [3], and I believe it should be part of every process engineer toolkit. Game theory studies interactions, called *games*, between rational and self-interested agents called *players*. Among the multiples potential outcomes of a game, the *Nash equilibrium* is arguably the most studied. Nash equilibrium is a game state where every player is adopting the best response to the other player’s actions. This gives this outcome stability, since any deviation from it ends in a lower pay-off for the deviating player. Nash equilibrium has been proven to exist for any finite game [4] and it has been observed in real-world scenarios, especially in situations where players are extremely familiar with the game (like professional sports [5; 6]).

Researchers have used game theory to analyse the behaviour of software professionals and to suggest improvements to existing practices. A literature review of the publications on the subject is available at [chapter 3](#). Due to the limitations of classic game theory, the proposed models are highly idealised. Instead of modelling hundreds of software engineers working on multiple releases, they only consider a few players with a limited number of interactions.

1 Introduction

Also, they are rarely validated against actual process data. Although these models are informative and can provide insights about potential conflicts, they are far from real software development scenarios.

Game representations for multi-stage games — like extended-form games — grow exponentially with the number of players and actions [7], so we need abstractions to keep the game size manageable. In this thesis, I propose Empirical Game-Theoretical Analysis (EGTA) as a suitable game abstraction approach for software development scenarios [8; 9]. The reduced games produced by EGTA adopt the normal-form representation — used for single-shot games — where the actions are the behaviours under study, and the pay-off values are obtained via simulation. I rely heavily on the existing literature on software process simulation to ensure that the models effectively reflect the practices under analysis [10; 11].

In this thesis, I propose GTPI: an EGTA-based approach for game-theoretic modelling of software development practices (chapter 4). Using GTPI, we can obtain the behaviour of its players — process performers — at equilibrium. Ideally, the behaviour at equilibrium is cooperating players. If that is not the case, GTPI proposes to iteratively modify the game-theoretic model of the practice until this behaviour is obtained.

1.1 Research Problem

Software processes are instrumental to the quality of a software product. Nowadays, software process design is driven by hard-won experience instead of formal theory. Game theory is a suitable tool for the formal analysis of software development processes. However, classic game theory faces limitations when modelling software development at scale. The research problem is enabling game theory for software teams, so they can use it to improve their development processes. In this thesis, I propose a game-theoretic approach to identify incentive problems in software development processes. The approach, called GTPI, uses game-theoretic models to diagnose process issues and resolve them with improved software processes.

1 Introduction

Game theory can be used to solve process problems affecting engineering teams in the field. *Priority inflation* is one of such problems: it affects task prioritisation processes ([chapter 5](#)). Practitioners have reported that although their bug tracking systems support a multi-level prioritisation scheme, they tend to be overflowed with top-priority bug reports [12]. They believe it is caused by lack of team bandwidth. A software team overflowed with bugs can only focus on the top-priority ones. When bug reporters notice this, they realise that to obtain fixes reporting them as top-priority is the only viable option. Priority inflation makes the prioritisation effort useless — since suddenly everything is important — which can lead to a waste of development time. I confirmed the extent of this problem with a survey to 152 software professionals, where 25% of them reported that priority inflation is frequent in their work environment and 31% of them believe it has a significant impact in their activities.

For the identification of incentive problems, I rely in the analysis of the game-theoretic model of the process under scrutiny. This model needs to support multiple stakeholders interacting during the development project. Since classic game theory struggles with scale, I adopt EGTA abstractions. *TaskAssessor* is the EGTA-based model I designed for task prioritisation. The data-driven modelling approach used 42,620 bug reports obtained from Apache’s JIRA repository. From this dataset, I extracted the bug reporting behaviours that would become strategies in the model. The players of the game are bug reporters, who place reports with a priority label assigned according to their strategy. The development team fix the bugs in priority order until they reach a target number of fixes. The number of fixes reported as high-priority represent the reporter’s pay-off value.

Once ready, the game-theoretic model can then be tailored to diagnose alternative designs of the software process to improve. First, I use *TaskAssessor* to analyse two task prioritisation processes adopted by practitioners. The first is the one used by open-source projects, where the end-user assigns a priority when reporting a bug. I found a single equilibrium, where every bug reporter adopts a dishonest strategy with 100% probability. The next practice analysed is adopting bug triage, where the priorities assigned by bug reporters were overridden by the triage team. The analysis shows, that as long as the bug triage team is imperfect in their assessment, the equilibrium still has a positive probability for dishonest strategies.

1 Introduction

Game-theoretic equilibrium analysis can drive the design of new software processes that do not expose the original incentive issues. These new processes need to be deployed, including tool support to ease their adoption. To address the limitation of existing prioritisation processes, in [chapter 6](#), I propose a novel process called *assessor-throttling*. In assessor-throttling, bug reporters keep assigning bug priorities, but developers evaluate their assessment after delivering a fix. If the developer believes the assessment of the bug reporter is incorrect, the reporter's *reputation score* is diminished. The order of fixes in assessor-throttling does not only depend on the reported priority: it is also a function of the reporter's reputation score. In that way, honest reporters are more likely to obtain fixes than dishonest ones. After parameter tuning, assessor-throttling has a single equilibrium where all the reporters adopt an honest strategy, the desired outcome. To support assessor-throttling adoption, I developed a free open-source Chrome-extension that can connect to a JIRA repository and prioritise the developer inbox based on the bug reporters reputation.

1.2 Objectives

The objectives of this thesis are the following:

- Develop an approach for building game-theoretic models of software development practices based on software process data.
- Analyse existing software development practices to determine if the incentives in place effectively encourage cooperation.
- Improve software practices that show anomalies at equilibrium by proposing deployable interventions.

1.3 Contributions

The contributions follow:

1 Introduction

- This thesis is the first application of empirical game-theoretic analysis to model software processes: technical debt ([chapter 4](#)) and task prioritisation ([chapter 5](#)).
- I surveyed game-theoretic models in software engineering ([chapter 3](#)).
- I performed a complete, end-to-end analysis and fix of budget-driven technical debt: I diagnose it using game theory, then apply mechanism design to suggest a simple and inexpensive change, and validate the new process via its Nash Equilibrium ([chapter 4](#)).
- I showed that task prioritisation processes in which quality assurance (QA) engineers prioritise issues suffer from priority inflation, and that the common solution of interposing a gatekeeper does not prevent inflation and, in fact, reduces productivity ([subsection 6.1.1](#) and [subsection 6.1.2](#)).
- I propose assessor-throttling, a novel and lightweight task prioritisation process that is immune to priority inflation ([subsection 6.1.3](#)).

1.4 Thesis Outline

The rest of the thesis is organised as follows. In [chapter 2](#), I present key concepts from game theory and introduce the most important game representations. Per game representation, I discuss their capabilities, limitations and mention some representative game models.

In [chapter 3](#), I survey the applications of game-theoretic models to software engineering problems. I grouped the papers by knowledge area from the Software Engineering Book of Knowledge (SWEBOK) guide. I also discuss what I believe are the two main challenges for a massive adoption of game theory by software engineers: the need for game abstractions and the rationality assumptions behind game models.

In [chapter 4](#), I introduce GTPI: the proposed software improvement approach based on EGTA abstractions. I discuss each stage of GTPI's two phases: 1) software process modelling and 2) software process improvement. I illustrate its usage with an application to budget-driven technical debt and discuss practical aspects for GTPI's adoption.

1 Introduction

I approach priority inflation using GTPI in [chapter 5](#). In this chapter, I focus on the software process modelling phase and its two stages: 1) identifying process anomaly ([section 5.2](#)) and 2) empirical game design ([section 5.3](#)). To identify the process anomaly, I perform a developer survey regarding the extent and impact of inflated priorities on their work. I also perform an empirical study of GitHub labels, to analyse their use as priority markers. Next, I describe *TaskAssessor*, the game-theoretic model produced from the empirical game design phase. I elaborate on the whole game design process, starting from data gathering to the final model validation.

Continuing the GTPI approach on priority inflation, in [chapter 6](#) I describe the software process improvement phase. This phase has two stages: 1) empirical game improvement ([section 6.1](#)) and 2) process deployment ([section 6.2](#)). For empirical game improvement, I use *TaskAssessor* to model two prioritisation processes adopted in industry. Both of them — distributed prioritisation and gatekeeper — are susceptible to priority inflation according to the equilibrium analysis. In contrast, the proposed prioritisation process, *assessor-throttling*, has no priority inflation at equilibrium. Regarding process deployment, I describe the Chrome plugin I developed to enable an easy adoption of assessor-throttling, targeted at developers using the JIRA tracking system.

Finally, [chapter 7](#) concludes this thesis. I summarise the contributions and directions for future work.

2 Background

This chapter is a concise, minimal reference of game theory tailored to the papers I survey in [chapter 3](#). I describe the main game representations and how software engineering researchers have used them, emphasising each representation potential and limitations.

I envision game theory as an essential tool for software engineers. This chapter can provide such professionals with the best practices for game-theoretic modelling according to their discipline and environment. Game theory offers a plethora of representation models. Each representation comes with a set of assumptions. For example, player's knowledge of the game ranges from perfect information, through imperfect information, to incomplete information. Each of these levels have serious implications for the expressiveness of the model. A key contribution of this chapter is to elucidate how these assumptions impact the game-theoretic models in a software engineering context.

2.1 Concepts and Terminology

In game-theoretic context, a *game* is a scenario where self-interested agents, or *players*, interact and their actions impact the payoff of all of them. Games can be represented in multiple ways, but the *normal-form* ([section 2.2](#)) is arguably the most fundamental. Normal-form games are represented by *payoff matrices*, like the one shown in [Figure 2.1](#). It models a game between 2 players, named DEV1 and DEV2, where both of them have 2 possible actions in the context of the game: to cooperate or to oppose. Each cell in the payoff matrix contains the *payoff values* obtained by each player after they perform their corresponding actions. For example, the top-right cell

2 Background

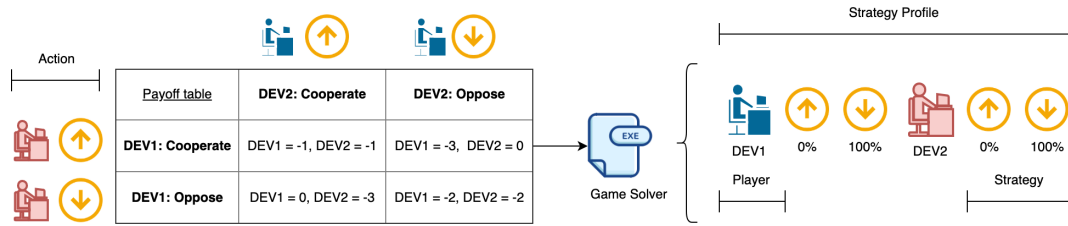


Figure 2.1: Normal-form representation of a two-player game (DEV1 and DEV2), where each player has two actions (cooperate and oppose). At equilibrium, both players adopt the opposing action with a probability of 100%.

of the payoff matrix states that when DEV1 adopts the cooperate action and DEV2 adopts the oppose action, DEV1 obtains a payoff value of -3 and DEV2 obtains a payoff value of 0. A *payoff function* assigns payoff values to each player according to a game outcome. These are numerical values representing the player's valuation of a game outcome. A *strategy* defines the behaviour of the player in a game. Strategies fall into two categories. In *pure strategies*, players select a single action; in *mixed strategies* players randomise over actions according to a probability distribution [13]. A *strategy profile* is a mapping of *strategies* to each player.

In a *Nash Equilibrium (NE)* strategy profile, all the players adopt a strategy that is the best response to their opponents' strategies: any deviation lowers a player's payoff. Nash proved that every finite game has at least one NE [4]. The game described in Figure 2.1 has a single NE where both players adopt the same strategy: To oppose with a probability of 100% and to cooperate with probability 0%. That strategy profile corresponds to the lower-right cell of the pay-off matrix, where both players get a pay-off of -2. NE are stable since any deviation from them produces payoff penalties. This is visible in the pay-off matrix in Figure 2.1: when the game is at equilibrium and both players oppose, the player who moves to cooperate would see their pay-off diminished to -3, while their opponent would see their pay-off increased to 0. If players are rational and understand the game, we expect that repeated play will converge to NE. Empirical evidence in professional sports suggests this happens in practice [5; 6].

Game theory has multiple applications in Computer Science. For example, in distributed systems game theory has been used to analyse peer-to-peer

2 Background

systems, like BitTorrent. We can model these systems as games between system users, with player actions expressed as seeding attitudes and their payoff as a function of file availability. Researchers in distributed systems use game-theoretic models to design system protocols that, at equilibrium, discourage unwanted behaviours (i.e. free riders) or make these systems resilient if such behaviours arise [14].

2.2 Normal-form Games

Normal-form games represent scenarios where players act without interaction or coordination [15]. In this section, I am addressing *complete information normal form games*, where the game structure is common knowledge among the players. I discuss incomplete information games in [section 2.5](#). A single-round rock-paper-scissors match is a good example of a complete information normal-form game: both players are well aware of the rules of the game and, due to the need of simultaneous play, no coordination is possible. Leyton-Brown and Shoham [13] define normal-form game as

Definition 1 (Normal-form game) A normal-form game is a tuple (N, A, u) where:

1. N is a set of n players.
2. $A = A_1 \times \dots \times A_n$, where A_i is a set of actions available to player i .
3. $u = (u_1, \dots, u_n)$ where $u_i : A \mapsto \mathbb{R}$ is the payoff function of player i .

The Prisoner's Dilemma: This game is arguably the most famous game-theoretic model. Researchers have used it extensively in domains like economics and politics [15]. In software engineering, it has been used to model cooperation among developers ([subsection 3.2.6](#)) and between testers and developers ([subsection 3.2.4](#)). In a popular variant, the police interrogates two felons in separate rooms. They have evidence to prosecute them individually for a minor offence, but need testimony to go after a more serious crime. During interrogation, each prisoner has two options: 1) To defect and incriminate their fellow criminal in exchange for a reduced sentence or 2)

2 Background

to cooperate with their fellow criminal and remain silent, which means the police will still charge them for the minor offence. The dilemma of the game lies in the fact that, while it is convenient for both prisoners to cooperate, the reduced prison time of defection moves equilibrium towards mutual betrayal. Since both prisoners act rationally and try to minimise their prison time, defecting is more attractive, regardless of whether the other inmate defects or cooperates.

Hawk-Dove: This is a two-player game where each player has two actions: 1) an aggressive hawk-like behaviour or 2) a friendly dove-like behaviour. The canonical version of the game models two animals fighting for prey [13]. If they both behave like hawks, they will be severely injured. If they both behave like doves, they would need to share the prey, which is good but not ideal. A player obtains the maximum payoff in this game when their opponent is friendly and they respond aggressively. Hawk-dove models have been used to explain lack of cooperation in software teams and to justify the implementation of stand-up meetings (subsection 3.2.6).

Stag Hunt: Initially described by Jean-Jacques Rousseau, the game is played by two hunters [15]. Each hunter has two possible targets: 1) a stag with a significant payoff or 2) a hare with a modest one. Each hunter by themselves is capable of capturing a hare. To be able to capture a stag, both hunters must cooperate. A stag hunt game has two pure-equilibrium profiles. In one, the hunters choose to hunt the stag together; in the other, they individually go for a hare. This model has been used to model cooperation between developers (subsection 3.2.7). Authors used stag hunt to show the potential of informal talk in moving equilibrium towards total cooperation.

2.3 Extensive-form Games

While normal-form games (section 2.2) are well suited for games with simultaneous moves, this definition falls short for games where plays in sequence are a key feature of the game, like poker. Besides the sequencing of actions,

2 Background

a model of poker needs to consider the information available to a player when taking an action. In poker, the player has limited information about the state of the game, since not all cards are visible. Also, the state of the game depends on probabilistic events, like the chance of obtaining an ace from the deck. In these scenarios, extensive-form games are a better representation. By using game trees, they support modelling players and their game knowledge, their actions, and action sequences.

Game theory classifies games like chess and tic-tac-toe as *perfect information* games. In this game representation; players, at every time step, know the game state and all the previous actions that led to that state, without interference from probabilistic events. Perfect-information games have been used to model software evolution. In this game, players are end-users and developers. End-users make several change requests, and developers respond to them by adapting the software's design (subsection 3.2.5).

Leyton-Brown and Shoham [13] define perfect-information games in extensive form as:

Definition 2 (Perfect-information game in extensive form) A perfect-information game in extensive form is a tuple $(N, A, H, Z, \chi, \rho, \sigma, u)$ where:

1. N is a set of n players.
2. A , is a set of actions.
3. H , is a set of non-terminal nodes.
4. Z , is a set of terminal nodes, where $H \cap Z = \emptyset$.
5. $\rho : H \mapsto N$ is the player function.
6. $\chi : H \mapsto \mathcal{P}(A)$ is the action function.
7. $\sigma : H \times A \mapsto H \cup Z$ is the successor function, such as $\forall h_i \in H, a_i \in A$, if $\sigma(h_1, a_1) = \sigma(h_2, a_2)$ then $h_1 = h_2 \wedge a_1 = a_2$.
8. $u = (u_1, \dots, u_n)$ where $u_i : Z \mapsto \mathbb{R}$ is the payoff function of player i .

The player function ρ determines which player is able to perform an action at an specific non-terminal node, while the action function χ defines the actions available at this node. The successor function σ defines the game flow, by producing the next node after a player performs an action. Note that each payoff function in u produce a value for each terminal node.

2 Background

In *imperfect information* games like poker, the players must act on partial information. Poker players know their hand and the visible cards at the table but have no access to the deck or the opponent hand. Leyton-Brown and Shoham [13] define them as:

Definition 3 (Imperfect-information game in extensive form) An imperfect-information game in extensive form is a tuple $(N, A, H, Z, \chi, \rho, \sigma, u, I)$ where:

1. $(N, A, H, Z, \chi, \rho, \sigma, u)$ is a perfect information game in extensive form.
2. $I = (I_1, \dots, I_n)$, where $I_i = (I_{i,1}, \dots, I_{i,k})$ is a partition of $P \subset H$ such as $\forall p \in P, \rho(p) = i$ and $\forall p_i, p_j \in I_{i,j}, \chi(p_i) = \chi(p_j) \wedge \rho(p_i) = \rho(p_j)$

$I_{i,j}$ represents *information set* j of player i . All the nodes in information set j are equivalent from player's i perspective, so every node in $I_{i,j}$ have the same actions available.

Stackelberg Competition: The inclusion of time and sequence in a game-theoretic model has important consequences in the equilibrium results. For example, let us imagine a market with only two producers for a specific good. Both competitors have the same production costs, and since they operate in the same market they face the same demand. The *Cournot competition* game-theoretic model of this market requires both companies to define the quantity to produce simultaneously and without coordination, configuring a complete information normal-form game (section 2.2). In contrast, a *Stackelberg competition* has a market leader who announces their production quantity first, and the market follower defines their production later taking the leader production into account. A Stackelberg competition model is better suited for a perfect-information extensive-form game representation [15]. A Cournot competition and a Stackelberg competition differ on the quantities produced at equilibrium: in a Stackelberg competition, the market leader advantage allows them to take a bigger share of the market.

In software engineering, interactions between developers and testers have been modelled using Stackelberg competitions: The tester as leader defines a testing strategy and the developer as follower implements the features to test (subsection 3.2.4). Also, contributions in open-source projects were

2 Background

approached with this model, having the open-source organisation as leader setting up the coding environment followed by developers contributing either by submitting code or by engaging in project discussions ([subsection 3.2.5](#)).

2.4 Multistage Games

A multistage game consists of a finite sequence of stage games, where each stage game is expressed in the normal-form ([section 2.2](#)). In a multistage game setting, the same players interact over multiple stage games, collecting the corresponding payoffs after each game ends [[15](#)]. Due to the time dimension inherent to multistage games, imperfect information extensive-form games ([section 2.3](#)) are a suitable representation for them.

A *repeated game* is a multistage game where the *same stage game* is played at every interaction. Tadelis [[15](#)] define them as:

Definition 4 (Finitely repeated game) A finitely repeated game is a tuple (N, A, u, T, β) where:

1. (N, A, u) is a normal-form game, called stage game.
2. T is the number of times the stage game is played.
3. β is the discount factor.

The total payoff player i after T stage games conclude is $\sum_{t=1}^{T-1} \beta^{t-1} r_i^t$, where r_i^t is the reward obtained by player i and stage game t . We use the discount factor β to represent that players value earlier payoff increases than increments that will occur later. This is a common practice in economic modelling [[15](#)].

Infinitely repeated games can be used to represent uncertainty regarding the end of game. In this model, $\beta < 1$ also represents the probability of engaging on another stage game. Having the players interacting an indefinite number of times enables the emergence of cooperation, even when the stage game in isolation has a single equilibrium towards defection [[15](#)]. This model was used to analyse the role of stand-up meetings in fostering collaboration inside software teams ([subsection 3.2.6](#)).

2.5 Bayesian Games

Bayesian games are also called games of *incomplete information* [13]. As mentioned in [section 2.3](#), in perfect information games — like chess — players know exactly the game state at every time step, while in imperfect information games — like poker — players understand the game structure but they have partial access to the game state. In contrast, games of incomplete information are games in which the players are uncertain about their opponents preferences. To support modelling these scenarios, Bayesian games incorporate the notion of *epistemic types*, that represent each player’s private information. Leyton-Brown and Shoham [13] define Bayesian games as:

Definition 5 (Bayesian game) A Bayesian game is a tuple (N, A, Θ, p, u) where:

1. N is a set of n players.
2. $A = A_1 \times \dots \times A_n$, where A_i is the set of actions available to player i .
3. $\Theta = \Theta_1 \times \dots \times \Theta_n$, where Θ_i is the epistemic type space of player i .
4. $p : \Theta \mapsto [0, 1]$ is a common prior over epistemic types.
5. $u := (u_1, \dots, u_n)$, where $u_i : A \times \Theta \mapsto \mathbb{R}$ is the payoff function of player i .

To enable equilibrium analysis with uncertainty over payoffs, Bayesian games require that the probability distribution p is common knowledge [15]. This is also known as the *common prior assumption*. In a software engineering context, Bayesian games have been proposed to model the relationship between software vendors and their clients ([subsection 3.2.6](#)).

Auctions: Auctions are the canonical Bayesian game: although bidders know their own valuation of the auctioned good, they do not know how their opponents value it. Governments use auctions extensively to assign public goods — like portions of the electromagnetic spectrum — to private companies. In software engineering, auctions have been proposed in middleware design as mechanisms for dealing with conflicting resource requests ([subsection 3.2.2](#)). A well-designed auction discourages collusion and, at equilibrium, assigns the auctioned good to the bidder with the highest valuation [15].

2 Background

In an auction game model, the players in N are the bidders. A bidder's i epistemic type $t \in \Theta_i$ corresponds to their valuation of the auctioned good. They can bid $b \in A_i$ during an auction, and their payoff u_i will depend on their valuation t , their bid b and if they obtain the good at the end of the auction, according to the auction rules. Auctions can be grouped in two categories: In 1) *open auctions*, players have visibility of the bidding process until a winner emerges, and in 2) *sealed-bid auctions*, players privately submit their bids and are not aware of the bids of the rest of the players.

2.6 Mechanism Design

Instead of representing scenarios with Bayesian games, *mechanism design* studies what Bayesian game a designer can implement; in order to make players accomplish a desired outcome independent of their type [15]. In software engineering, this outcome can be high-quality code commits (subsection 3.2.4) or optimal team productivity (subsection 3.2.6). Shoham and Leyton-Brown [16] define mechanism as:

Definition 6 (Deterministic mechanism) A deterministic mechanism for a Bayesian setting (N, O, Θ, p, u) is a pair (A, M) where:

1. N is a set of n players.
2. O is a set of outcomes.
3. $\Theta = \Theta_1 \times \dots \times \Theta_n$ is a set of joint type vectors.
4. p is probability distribution over Θ .
5. $u := (u_1, \dots, u_n)$, where $u_i : O \times \Theta \mapsto \mathbb{R}$ is the payoff function of player i .
6. $A = A_1 \times \dots \times A_n$, where A_i is the set of actions available to player i .
7. $M : A \mapsto O$ maps each action profile to an outcome.

Mechanism design goal is to, given a Bayesian setting, produce a mechanism whose equilibrium exhibits specific properties. One of these properties is *incentive compatibility* or *truthfulness*: In such mechanisms, rational players prefer to reveal their private information instead of lying. For example, let us imagine a government as a mechanism designer, designing an auction for an infrastructure project. Project bidders are the players of this game, and is

2 Background

in the interest of the government for them to bid a fair price for the project. In this case, the fair bid is the epistemic type the mechanism designer wants to extract from the players with an incentive compatible auction.

Stable Marriage Problem: Also called the *stable matching problem*, it has been used to model assigning students to colleges. In its canonical form, we have a group of men and a group of woman, with equal number of members. Each man has an ordered preference over the women in the opposing group. The same happens to each woman, with respect to men in the other group. The designer's goal is to produce a game that generates a *stable matching* at equilibrium, in which there is no man desiring to match a different woman, having this woman reciprocating his feelings. In software project management, task assignment was approached as a stable matching problem (subsection 3.2.6). The *deferred acceptance algorithm*, proposed by David Gale and Lloyd Shapley, produces stable matchings. It requires a series of iterations where one of the groups propose to the other group. This algorithm is incentive compatible with respect to the proposing group, meaning that none of its members can obtain a better matching by lying about their preferences.

Principal-Agent Problem: In a principal-agent game, the players are a principal and its agent, where the agent acts in behalf of the principal. For example, in an industrial setting the principal can be the owner and the agent the manager of the company. Or in a political context, the principal can be the constituents and their agent a member of parliament. In software estimation, the project manager has been proposed as principal, requesting estimates from developers who act as agents (subsection 3.2.6). The principal has only partial visibility of the agent's actions, so the agent may have an incentive to act in their own interest instead of focusing on the principal's goals. From a mechanism design perspective, the principal acts as designer with the goal of proposing a compensation scheme — the mechanism — where the agent maximises the principal's payoff at equilibrium [17].

2 Background

Prediction markets: In a prediction market problem, the mechanism designer's goal is to elicit player's beliefs with respect to an event, like an election result. The resulting mechanism defines contracts that assign payments to specific outcomes. For example, a contract can pay its owner \$Y in case candidate Z wins an election. Participating players, according to their beliefs, can buy or sell these contracts. The price of the contract at equilibrium is an aggregate measure of the beliefs of the players [16]. A *scoring rule* is a prediction mechanism to obtain the beliefs of a single player. Given a set of outcomes, a scoring rule assigns a payoff to each outcome in case they were both realised and reported. A *proper scoring rule* is incentive compatible, which requires that truthful reporting maximises the player's payoff [18]. In software effort estimation, proper scoring rules were proposed to elicit accurate estimates from developers (subsection 3.2.6).

2.7 Cooperative Game-Theory

Non-cooperative game-theory focusses on individual players and the actions they have available in a game [19]. In contrast, cooperative game-theory focusses on groups of players, and the coalitions they can form [13]. Lets picture, for example, a budget bill that needs a minimum number of votes to be approved by parliament. Lets also assume that no individual party has the required votes, and each party wants control over a part of the budget. Cooperative game-theoretic models can provide insights on which coalitions can arise, and how parties should distribute the budget if parliament approves the bill.

Bargaining problem: In this scenario, players need to select among several collaboration settings [19]. Each collaboration setting distributes payoff among the collaborating players. If the players do not reach an agreement, the players receive the *disagreement outcome* as payoff. Myerson [3] defines the bargaining problem as:

Definition 7 (N-person bargaining problem) A bargaining problem is a tuple (N, F, d) where:

2 Background

1. N is a set of n players.
2. $F \subset \mathbb{R}_n$ is the set of feasible payoff allocations if players collaborate.
3. $d = (r_1, \dots, r_n)$ is the disagreement outcome.

The bargaining problem can model salary negotiations in the workplace or even trade deals among countries. It is also used to find a viable software configuration when end-users have conflicting requirements ([subsection 3.2.1](#)).

3 Literature Review

In this chapter, I present a detailed discussion of the application of game-theoretic models to several software engineering disciplines. I also discuss the two main challenges for game theoretic modelling in software engineering: the need of abstraction for making tractable games and the validity of game theory's rationality assumptions.

Given the ubiquitous presence of software in modern life, ensuring software quality is of utmost importance. Software engineering is the technical discipline whose goal is to deliver high-quality software. Every human endeavour is subject to conflict, the process of engineering software is no exception. Testers want to maximise defect detection while developers want to minimise it. Project managers try to minimise cost and schedule while technical leads want project plans with enough buffer time. Software architects want a modular and healthy codebase while clients want to focus most resources on feature delivery.

As [chapter 2](#) details, game theory studies scenarios where participants have conflicting interests. Game theory studies mathematical models of conflict and cooperation [3], with multiple applications in economics, biology and business. There are also many applications in computer science [20]. Game theory allows to include motivations and interests in their models. Researchers and practitioners can then use these models to obtain insights into agent behaviour when agent interactions impact their perceived utility.

In a software engineering context, these agents — or *players* in game-theoretic terms — can be end-users, software engineers or even software systems. For example, Grechanik and Perry identified that many software project quality attributes — like cost, delivery time and number of features — are valued differently by each stakeholder community [21]. While project managers and

3 Literature Review

customers are interested in minimising delivery time, a tight development schedule can translate to an overworked development team. Grechanik and Perry propose game theory as an appropriate tool for handling these conflicts. Like them, I believe in game theory's potential for software engineering. In this chapter, I review the pioneering work in this area, mentioning the main achievements and the challenges it faces.

3.1 Methodology

In this section, I describe in detail the paper collection process I followed for this review.

Inclusion Criteria: To explore the application of game-theoretic models to software engineering, let's start by defining software engineering. Among its multiple definitions [22], the one from the ISO/IEC/IEEE Systems and Software Engineering Vocabulary [23] seems adequate: “[software engineering is the] application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software”.

Software engineering relies on a wide spectrum of disciplines [22]. Computer science, project management and mathematics, among others, are part of the modern software engineer toolbox. Reviewing the game-theoretic models on software engineering including all its related disciplines requires huge effort. Instead, I rely in the Guide to the Software Engineering Book of Knowledge (SWEBOK) [1] to establish a boundary between what is in scope and what to exclude from this review. The SWEBOK guide makes a distinction between core knowledge areas for software engineers and its related disciplines [24]. The last version of the SWEBOK guide [1] contains 15 core knowledge areas, as seen in Figure 3.1. The 15 knowledge areas are divided in two categories: 1) practice of software engineering and 2) educational requirements of software engineering [25]. I consider a publication to be in scope if it uses game-theoretic models to approach a problem in a *practice* SWEBOK knowledge area. This excludes the knowledge areas of

3 Literature Review

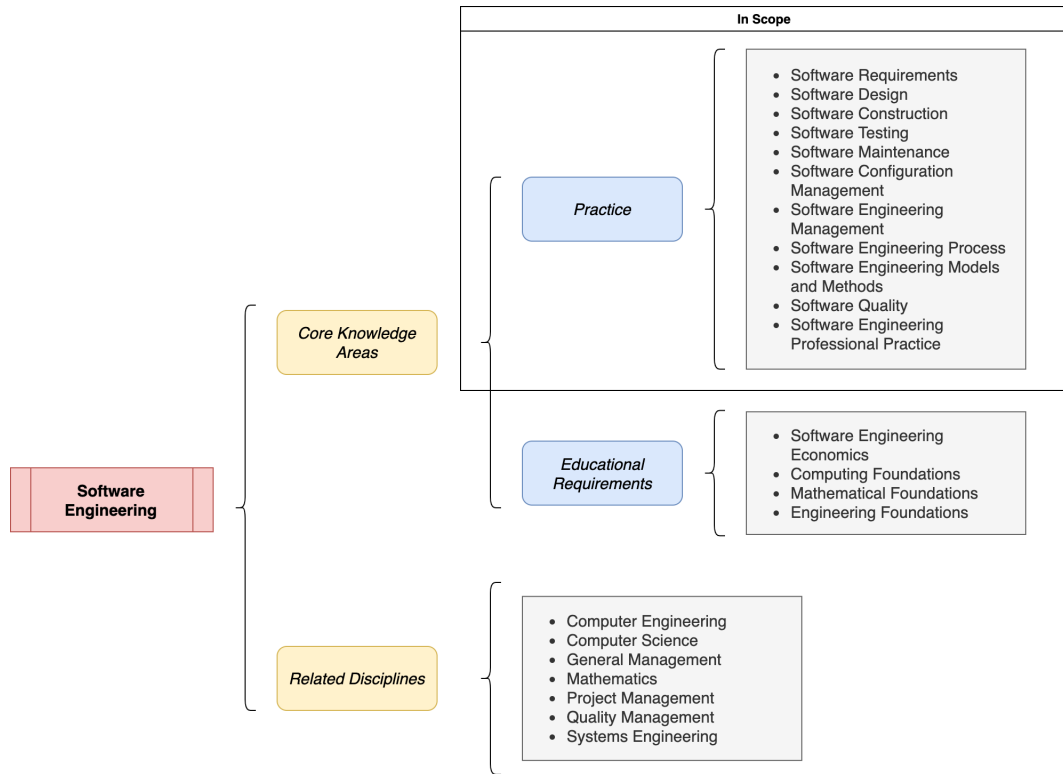


Figure 3.1: Scope of this review with respect to the SWEBOK guide [1]. For this literature review, we only consider papers focussing on *practice core knowledge areas*.

Software Engineering Economics, Computing Foundations, Mathematical Foundations, and Engineering Foundations.

For example, publications focussed on Computer Security or Networking are not considered in scope; since they fell into the Computing Foundations knowledge area and/or the Computer Science related discipline. While game-theoretic models have been widely applied in both disciplines [26; 27], this literature review’s focus is on the approach to software development and not on the properties of the final software product. We can apply a similar reasoning to Algorithmic Game Theory, that studies algorithm development for game-theoretic problems [19]. We can place this discipline in the intersection of the related disciplines of Computer Science and Mathematics.

Research that approach game-theoretic problems — with conflicting objec-

3 Literature Review

tives of utility maximising agents — without explicitly using game theory is considered out of scope. For example, Yu *et al.* [28] propose modelling users and their conflicting needs as part of the requirements engineering effort. While this publication belongs to a core software engineering discipline and can certainly be approached with game-theoretic models, the authors adopt social modelling instead. I additionally excluded publications from regional venues, like the work of Kumar and TV on game theory for software design [29]. This filter was necessary to keep a manageable number of papers to review.

Search Procedure: I used the Google Scholar search engine¹ to look for relevant publications, performing the following queries:

1. Papers that contain the phrases “game theory” and “software engineering”.
2. Papers that contain the phrases “game theory” and “software development”.
3. Papers that contain the phrase “game theory” and where published in a venue whose name contains “software engineering”

All these queries look for papers that contain the phrase “game theory”, since I am looking for publications aware of game theory as a discipline and that use its techniques and terminology. Per query, I manually inspected the abstract of the first 50 matches to verify if they fit the inclusion criteria. I deemed this number high enough to explore the most relevant publications. For each publication in scope, I reviewed its bibliography and looked for relevant papers. I initially performed the search in September 2015, and then repeated it in January 2019 to include the latest publications. The papers I found are listed in [Table 3.1](#).

3.2 Game Models in Software Engineering

In this section, I describe previous work in game-theoretic models for software development. Each of the following subsections corresponds to a SWE-

¹<https://scholar.google.co.uk/>

3 Literature Review

Table 3.1: Publications using game-theoretic models to address software engineering problems

Reference	Knowledge Area	Game Representation	Reference Model
García-Galán <i>et al</i> [30]	Software requirements	Cooperative game theory	Bargaining problem
Capra <i>et al.</i> [31]	Software design	Mechanism design	Auctions
Kitagawa <i>et al.</i> [32]	Software construction	Normal-form game	Hawk-dove
Feijs [33]	Software testing	Normal-form game	Prisoner's dilemma
Kukreja <i>et al.</i> [34]	Software testing	Extensive-form game	Stackelberg competition
Rao <i>et al.</i> [35]	Software maintenance	Mechanism design	–
Sazawal and Sudan [36]	Software maintenance	Extensive-form game	–
Bavota <i>et al.</i> [37]	Software maintenance	Normal-form game	–
Hata <i>et al.</i> [38]	Software maintenance	Extensive-form game	Stackelberg competition
Oza [39]	Software maintenance	Bayesian games	–
Bacon <i>et al.</i> [40]	Soft. Eng. Management	Mechanism design	Scoring rules
Bacon <i>et al.</i> [41]	Soft. Eng. Management	Mechanism design	Principal-agent problem
Lagesse [42]	Soft. Eng. Management	Mechanism design	Stable marriage problem
Yilmaz <i>et al.</i> [43]	Soft. Eng. Management	Mechanism design	–
Yilmaz and O'Connor [44]	Soft. Eng. Management	Mechanism design	–
Hassan and Dubinsky [45]	Soft. Eng. Management	Normal-form game	Prisoner's dilemma
Wang and Redmiles [46]	Soft. Eng. Professional Practice	Normal-form game	Stag hunt
Hasnain <i>et al.</i> [47]	Soft. Eng. Professional Practice	Normal-form game	–

BOK *core* knowledge area (see Figure 3.1).

3.2.1 Software Requirements

This knowledge area deals with the elicitation, analysis, specification and management of software requirements.

Requirement negotiation is an important part of requirement analysis [1]. It requires to resolve conflicts between incompatible features, demanded by different end users. In that area, García-Galán *et al* [30] propose modelling multi-user system configuration as an instance of the bargaining problem (section 2.7). Given a system with multiple configuration options — like a smart home — and a set of users with conflicting configuration needs, they use a cooperative game model to produce an adequate configuration. This configuration corresponds with the *Nash bargaining solution*, proven to be unique for every bargaining problem game [3].

3.2.2 Software Design

Software design is the process of defining the architecture, elements, and interfaces of a software system [23].

A key issue in software design is the management of heterogeneous system components [1]. A common approach is to delegate system integration to middleware software. Capra *et al.* [31] propose a middleware layer to enable context-aware applications on mobile devices. These applications could, for example, display images in lower quality to save battery when this resource is low. Such middleware would require application developers to include policies as part of their software, to define how it should behave under given environmental conditions. The authors define environment as any system resource under middleware monitoring; including memory, battery or bandwidth. The proposed middleware requires a conflict resolution mechanism in case of incompatible policies. For example, while one application would like to reduce screen brightness to save battery, another would prefer to maximise brightness to enhance user experience. The authors propose a conflict resolution mechanism based on sealed-bid auctions (section 2.5). In their model, the middleware plays the role of the auctioneer, the applications running on the middleware are the bidders, and the auctioned goods are viable policies. The author's goal is to design a resolution mechanism that benefited the largest number of applications, instead of classic auction models where only one bidder obtains the good.

3.2.3 Software Construction

Software construction is the activity of building software via coding, testing and debugging. Due to its nature, software construction is related to *all* SWEBOK knowledge areas.

Constructing for verification is a fundamental topic in software construction [1]. It includes techniques that enable fault detection during the software development lifecycle, like code reviews. Kitagawa *et al.* [32] posit that reviewer participation configures a hawk-dove game (section 2.2). Their model

3 Literature Review

has two reviewers as players, with the actions of review and not-review. Review corresponds to the dove behaviour and not-review to a hawk attitude. In correspondence with the hawk-dove game, their proposed game has two pure equilibrium profiles, where one reviewer behaves like hawk and its opponent like dove. These results do not match their empirical observations, where single-reviewer code reviews were a minority. The authors explain this in terms of the benefit of the code review, a model parameter that can move equilibrium towards cooperation given a sufficiently high value. I believe their model can be improved by changing the game representation. Modern code review happens *in sequence* — one review after the other — and not simultaneously, which violates a key assumption of normal-form games. Extensive-form games (section 2.3) are a more suitable representation for this scenario.

3.2.4 Software Testing

According to the SWEBOK guide, software testing consists on dynamically executing a finite set of test cases against a system, to verify if its behaving according to specification.

A software testing effort is constrained by the impossibility of executing every test case possible and the resources available for testing, given a project schedule and budget [1]. Hence, the time and effort engineers devote to software testing should be optimal. Feijs [33] models testing resource utilisation using normal-form games (section 2.2). He considers developers and testers as players and that their actions had a quality degree attached. For developers, this translates to “poor quality” or “high quality” implementations, and for testers to “poor testing” or “high quality” testing. While testers prefer to detect and report bugs, developers want implementations to not have bugs reported. Both players want to minimise the invested effort. Feijs argues that due to time constraints, software projects normally have programming and test design activities running in parallel. So, players perform their actions without coordination, as required for normal-form models. Feijs’ game model — called the *Idealized Testing Game (ITG)* — constitutes a Prisoner’s Dilemma instance (section 2.2), where both testers and developers perform their tasks with high quality at equilibrium. He briefly explores

3 Literature Review

an scenario where the developer performs an action before the tester does. However, Feijs did not use the extensive form representation (section 2.3), as is usual in games where actions have a sequence. Equilibrium analysis of the sequential version of ITG found the same result, where both players perform high quality activities.

The inclusion criteria for test case execution has a big impact on the effectiveness of software testing [1]. Kukreja *et al.* [34] propose a randomisation strategy for selecting test cases over an existing suite. Their approach is based on modelling software testing as a Stackelberg competition (section 2.3), having the testers as leaders and the developers as followers. Unlike Feijs' proposal [33], testers start by selecting a test randomisation strategy, and later developers select a quality level for requirement implementation. The authors define payoff so testers obtain a bigger payoff by testing requirements with high business value, while developers want to minimise time invested by submitting low quality implementations. The authors show that the randomisation strategies at equilibrium in their game-theoretic model produce higher payoff values than a uniformly at random approach. By adopting a Stackelberg competition, their model requires that developers know the testing strategy *before* they submit the implementations. Although this might not be a common use case, it can be a suitable model for some development contexts, like outsourced testing teams.

3.2.5 Software Maintenance

Software maintenance is related to the support activities required for software operation. They include bug fixing, adapt the software to new requirements, and update the underlying software platform; among others.

Bug fixing is commonly perceived as the most common software maintenance activity [1]. Rao *et al.* [35] approach bug fixing from a mechanism design perspective (section 2.6). Their goal is to define a process that incentivise addressing root causes, called *deep fixes*, instead of superficial workarounds, called *shallow fixes*. The proposed mechanism relies on a payment scheme that favours subsuming fixes, meaning fixes that also address bugs resolved previously with shallow fixes. Due to the scaling limitations of classical game

3 Literature Review

theory, the authors adopted mean field games as their abstraction approach (subsection 3.3.1).

Software evolution refers to the changes a software system experiences over time. It can produce increased complexity, unless the maintenance team adopts corrective measures [1]. Sazawal and Sudan [36] use extensive-form games (section 2.3) to model software evolution. Their proposed *Basic Software Evolution Game* has the software team and the user community as players. The software team makes the first move by selecting an initial design, to later respond to multiple change requests from the user community. After each request, the software team can: 1) keep the design and address the change request, 2) improve the design and then address the change request or 3) simply ignore the change request. Each of these design decisions have an impact on the final payoff. For example, an early design improvement can make subsequent change requests cheaper for the software team, or ignoring a change request can force the user community to migrate to a different provider at a higher cost. The authors propose to use the equilibrium predictions of the model to plan software maintenance activities.

Refactoring is a well-known software maintenance technique [1]. It modifies a software system to improve maintainability while preserving its original behaviour. Bavota *et al.* [37] designed a class refactoring method based on game-theoretic models. Their goal is to split a class in two, where the classes obtained have higher cohesion and lower coupling than the original class. Their approach has two software agents as players, each one in charge of extracting a class. Their actions in the game are to choose a method from the original class. Payoffs measure the impact of the player's chosen method in the cohesion/coupling of their extracted class. The authors represent this scenario using normal-form games (section 2.2). Software agents engage in multiple rounds, until there are no methods left on the original class. On each iteration, the methods assigned to each player's class correspond to the equilibrium profile of the normal-form game. This approach naturally fits the multistage game representation (section 2.4), but it is not adopted by the authors. Unlike Bavota *et al.*'s proposal, a multistage game *do not* necessarily adopt the equilibrium profile of the stage game at each round.

Staffing is hard for software maintenance teams. It is not considered as attractive as developing new software [1]. This is a critical problem for open-source

3 Literature Review

teams, since they mostly rely on volunteers. Hata *et al.* [38] use Stackelberg competitions (section 2.3) to investigate how to attract code contributions in open source projects. In their model, the leader is the organisation of an ongoing open source project, who start the game by selecting among two actions: 1) To keep the current project setup or 2) To improve the setup to make code contributions easier, for example, by improving developer documentation. The follower in their game is the open source project contributor. Their possible actions are to: 1) contribute with code or to 2) only engage in online discussion, without dealing with the codebase. Open source project organisers find improving the setup more costly, but it can have the benefit of promoting code contributions by making them cheaper for developers. The authors performed an empirical analysis of open source projects in GitHub, finding that projects with better setups have more code contributions.

Outsourcing software maintenance teams is becoming popular among IT departments [1]. Oza [39] explores player preferences when modelling offshore software outsourcing as a game between client and vendor. He proposes that the most common outsourcing scenario is an incomplete information game (section 2.5). Although the relationship is explicitly defined by a contract, there are expectations on both sides that remain hidden so payoffs are uncertain. Oza also posits that the client-vendor relationship develops over time, so expectations can become common knowledge and the game evolves to a perfect-information model.

3.2.6 Software Engineering Management

Software engineering management refers to the management activities required to deliver high-quality software systems. This activities include planning, monitoring, and reporting, among others.

Software project planning is part of this knowledge area. Estimates of effort, schedule and cost are necessary for project planning [1]. Bacon *et al.* [40] designed a mechanism for software effort estimation. In their effort estimation mechanism, project managers estimate the tasks developers perform, and developers invest time in completing these tasks. This mechanism is based on scoring rules (section 2.6) for both the developers and the project manager.

3 Literature Review

At equilibrium, project managers make the most accurate estimate possible and developers finish their tasks as soon as possible. Bacon *et al.* [41] refined their prediction mechanism in a later work, where they frame software effort estimation as a principal-agent problem (section 2.6). In their model, the project manager is the principal and the development team is the agent. The project manager, as a principal, relies on the developers to make accurate estimations and maximise the number of tasks finished. Their improved mechanism has three steps: 1) The developer shares information with the manager regarding the time required to finish a task, 2) Taking the developer input into account, the manager makes an effort estimation and 3) Finally, the developer completes the task. Their mechanism is based on proper scoring rules (section 2.6), to assign payoffs according to the manager's estimation accuracy.

Resource allocation is another relevant topic of software project planning [1]. Arguably, software engineer time is the most critical resource to allocate. Lagesse [42] propose to model task assignment in a software development team as a stable marriage problem (section 2.6). He models task assignment as a Bayesian setting, having team member preferences over tasks as their epistemic type. The author posits that the deferred acceptance algorithm (section 2.6) would need an extension to support project-specific concerns, like manager preferences, time and budget constraints, and team member skills. He provides a high-level description of the algorithm, mentioning its implementation and real-world evaluation as future work. Yilmaz *et al.* [43] also approach task assignment in a software project using mechanism design. Like Lagesse, they also consider that task preferences constitutes the team member's type: although they know their own task preferences they are unaware of the preferences of the rest of the team. Yilmaz *et al.* define team member payoff in terms of productivity, that is a function of their type and the tasks they have assigned.

Yilmaz and O'Connor [44] also approach resource allocation in software projects, but from a team structure perspective. They propose to use mechanism design (section 2.6) to optimise team composition and maximise its productivity. They define a *game-theoretic personality type (GTPT)* taxonomy, according to team member behaviour in a software project game. They also propose to extract the GTPT's from an organisation via interviews and surveys.

3 Literature Review

Contract management is an important part of software project execution. Service providers — including software engineers — need a contract where scope, penalties and incentives are properly defined [1]. Hassan and Dubinsky [45] propose that certain incentive policies can produce unwanted Prisoner’s Dilemma instances (section 2.2). They use as an example a software company with a bonus policy for project completion. If a two-developer team cooperate and work together, developers share the bonus equally between them. If only one cooperates, the defecting developer obtains a bigger cut of the bonus. In case of mutual defection, project is not finished in time and the company do not pay a bonus. The payoff matrix Hassan and Dubinsky present does not have mutual defection at equilibrium, as expected on Prisoner’s Dilemma instances.

3.2.7 Software Engineering Professional Practice

According to the SWEBOK guide, this knowledge area addresses the characteristics required for a “professional, responsible and ethical” practice of software engineering.

Group dynamics and psychology are key topics of this knowledge area [1]. Organisations expect software engineers to act cooperatively and constructively inside their teams. Wang and Redmiles [46] view cooperation in software development as a stag hunt game (section 2.2). Two developers in a team can cooperatively work on their tasks, obtaining a high-quality output (the stag payoff), or they can approach their tasks individually, but without the benefits of cooperation (the hare payoff). As explained in section 2.2, stag hunt games have two pure strategy Nash equilibria: both players cooperate or both defect. To move equilibria towards cooperation, the authors propose to include *e-cheap talk* as an extra action in the model. They define *e-cheap talk* as informal conversations over the internet, like instant messaging. Their analysis shows that *e-cheap talk* moves equilibrium towards full cooperation. Instead of adopting a classic game-theoretic approach, the authors opted for an evolutionary game-theoretic model. In this approach, they assumed a population of agents where each agent adopt single specific strategy [13]. The number of agents per strategy varies according to a fitness function that depends on the other strategies adopted in the population. An evolutionary

3 Literature Review

game-theoretic model outputs the proportion of agents per strategy after convergence. While most evolutionary game-theoretic models assume very large populations, Wang and Redmiles adopted Nowak's method for handling finite populations, a more suitable model for software teams [48].

Hasnain *et al.* [47] also explore collaboration in software teams, but using infinitely repeated games (section 2.4). The stage game of their model has two players and two actions for each of them: to work and contribute to the project or to shirk and benefit from the project as a free rider. Although the authors did not mention it, the payoff structure of their stage game corresponds to a hawk-dove game (section 2.2), where working corresponds to a dove-like behaviour and shirking is a hawk-like action. A hawk-dove game does not have the two players behaving like doves (*i.e.* cooperating) at equilibrium. The authors posit that stand-up meetings between stages can foster cooperation. They perform simulated software projects with real participants, separating them in two groups: one group had stand-up meetings while in the other they forbid communication. Their results show that cooperation is more frequent in the group performing stand-up meetings.

3.3 Challenges and Opportunities

In this section, I describe two issues I believe are impeding a massive adoption of game-theory in the software engineering domain, along with some suggestions for their solution.

3.3.1 The Need for Game Abstractions

Software processes are inherently temporal. In game theory, extensive form games (EFGs, section 2.3) represent players interacting over time. At their core, EFGs model sequential games as trees: some nodes inject non-determinism into the game, and the rest represent player's actions.

A game tree grows exponentially in the out-degree of each node with the number of interactions as the base. Naïve use of EFGs requires reasoning about astronomically huge trees. Sophisticated use of EFGs has an extensive

3 Literature Review

literature that details various ways to employ abstraction to reduce game size [7]. Among these approaches, I describe two: empirical game-theoretic analysis and the twins reduction.

Empirical Game-Theoretic Analysis: Empirical game-theoretic analysis (EGTA), proposed by Wellman [8], is a game theoretic framework that employs two techniques to reduce game size: 1) sampling action sequences and 2) simulation to reduce an EFG to a normal form game, in which all players move only once, simultaneously. It samples each player’s action sequences to reduce the out-degree of player nodes and restricts the tree’s height to the number of players, as shown in Figure 3.2. In the abstracted game, each player’s “action” is to choose an action sequence from among that player’s possible action sequences in the original game. This restriction of a player’s actions to action sequences restricts the height of the tree.

EGTA compresses complex games into smaller representations, and simulation is key to accomplishing this. EGTA simulates each terminal history of the reduced representation to compute the corresponding pay-off values per player.

Figure 3.2 models a two-player three-round rock-paper-scissors game under EGTA: from a full game tree of height 6 and 364 nodes we obtain an abstract game of height 2 and 7 nodes. Instead of having actions at the round level, now a player’s action is to select an *action sequence*. Player 1 has two available sequences: rock, paper, and scissors (RPS) or a stochastic sequence where playing scissors, then paper and finally rock has a probability of 0.3 and the probability of playing paper, rock, and scissors is 0.7 (30% SPR / 70% PRS). Player 2 can choose between rock-rock-paper (RRP) and paper-paper-scissors (PPS). The terminal nodes contain the pay-off values per player. For example, when player 1 selects RPS and player 2 selects RRP, player 1 wins twice, producing a 2-0 score. However, in the terminal nodes that involve a stochastic sequence (like 30% SPR / 70% PRS) the expected pay-off values must be obtained by averaging simulation results.

The first realization of EGTA, due to Walsh *et al.*, predates the definition of EGTA itself [9]. Walsh *et al.* use *heuristic strategies*, defined as “policies that govern the choice of individual actions” [9]. This definition is very general:

3 Literature Review

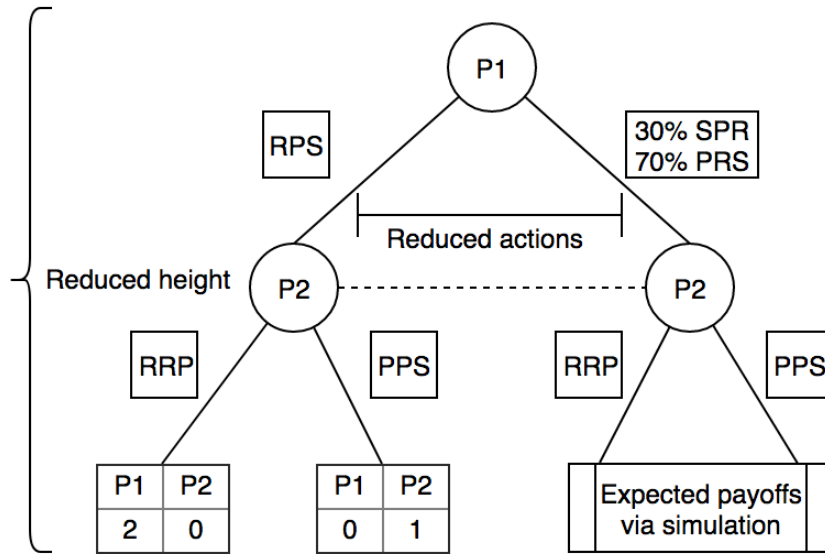


Figure 3.2: EGTA-abstracted game for a 2-player-3-round rock-paper-scissors game: each round victory is rewarded with 1 point and draws give no points to either player.

if we are using a game tree representation like the one in Figure 3.2, any program that traverses it qualifies as a heuristic strategy. I do not need this degree of generality, so, in this thesis, heuristic strategies are probability distributions over the actions at a decision node, guarded by a condition over the game state. A game analyst defines heuristic strategies based on their understanding of the game, to test hypotheses about player behaviour, or by interviewing experts or participants. I call the set of heuristic strategies for an empirical game its *strategy catalogue*.

Normal form games (section 2.2) are those games in which all players move only once, simultaneously, like a single round of rock-paper-scissors. Many game representations can be reduced to a normal-form, so it is considered “arguably the most fundamental in game theory” [13]. Walsh *et al.* represent empirical games using the normal-form, whose heuristic pay-off table is obtained via simulation [9]. Once the heuristic payoff table is ready, several algorithms are available for obtaining its Nash Equilibria [19].

When building the pay-off table, Walsh *et al.* assume a *symmetric game* [9]. In symmetric games, pay-off values are independent of player identity and

3 Literature Review

depend instead only on player actions. Consider the scenario in [Figure 3.2](#): Both players have the same action set and their pay-offs depend only on the action played. A symmetric game with a player set N and an action set S needs to compute $\binom{|N|+|S|-1}{|N|}$ entries for its pay-off table, instead of the $|S|^{|N|}$ entries required for an asymmetric game.

The Twins Player Reduction: EGTA is not enough to bring many interesting software processes into computational reach. Game representation size using the normal form and a pay-off matrix grows exponentially with the number of players and strategies [49]. Game-theory researchers already address this issue with multiple *player reduction* techniques [50; 51; 52].

An intuitive player reduction approach is to cluster players by their payoffs and strategies. Modelling all the players in a cluster as a single decision maker, however, ignores the fact that players within a cluster may act differently because of the lockstep actions of the other players within the cluster. Thus, the Nash equilibria computed for games using naïve clustering can be inaccurate.

To solve this problem, Ficici *et al.* propose the *Twins Player Reduction* approach [52]. Given a set of pure (*i. e.* deterministic) strategies, Ficici *et al.* compute a feature vector for each player whose components are the average payoff for each strategy over a set of sample game instances, then cluster them through k-means. In Ficici *et al.*'s nomenclature, players in the same cluster have the same *strategic view*. To support a reduced game that permits a player to deviate from their cluster's strategy, Ficici *et al.* represent each cluster with two players, the eponymous *twins*, in the reduced game. In a twins game, assume Player 1 selects Strategy A and his twin, Player 2, selects Strategy B: player 1's payoff corresponds to an agent who plays Strategy A in the full game while all other agents in their cluster play Strategy B and player 2's payoff corresponds to an agent who plays Strategy B in the full game while all other agents in their cluster play Strategy A.

Twins Player Reduction applies to both symmetric and asymmetric games, but is especially powerful when applied to symmetric games. When the game is symmetric, all players have the same expected payoffs for all strategies, hence they all have the same strategic view and fall into the same cluster. As

3 Literature Review

noted above, Walsh’s EGTA (section 3.3.1) assumes symmetry, so combining it with the Twins Player Reduction reduces the number of players to two and improves the scalability of the analysis. Although a twins game allows a twin to defect, Ficici *et al.* choose to restrict their analysis to Twin Symmetric Nash Equilibria (TSNE), a subset of Nash equilibria in which both twins adopt the same strategy. Ficici *et al.* prove that all twin games have a TSNE. Ficici *et al.* obtained pay-offs from a linear regression model trained with actual game data or simulation outputs. However, Wiedenbeck and Wellman [51] obtained better results via direct simulation, so this is the approach I adopt in this thesis.

3.3.2 Beyond the Rationality Assumptions

Game-theoretic models rely on strong assumptions regarding player knowledge. For NE convergence, game structure, player rationality and player beliefs must be common knowledge. These assumptions do not always align with human behaviour [15]. In contrast, *bounded rationality* approaches model rationality within the limits of attention, information, and mental capabilities of the decision maker [53]. *Prospect theory* is a bounded rationality model that describes decision-making under uncertainty. According to it, agents make decisions based on utility relative to current wealth — gains or losses— rather than the absolute wealth obtained. Prospect theory has been successfully applied to the design of drone delivery systems [54] and to the study of smart-grid adoption [55]. *Cognitive hierarchy* also relaxes game-theory’s rationality assumptions. Each player in a cognitive hierarchy model has a level, which determines the number of strategic reasoning iterations the player can perform. Level-0 players choose actions uniformly at random; players of superior levels are “smarter” in their decision-making. *Psychological games* extend the utility function definition to also depend on the players beliefs and their beliefs regarding the other players. This includes social norms and emotions in the game-theoretic model. In this thesis, I adopt the conventional game-theory approach that predicts convergence to NE under its rationality assumptions. I believe teams of experienced developers working over well-know codebases, like in open-source projects or in mature software maintenance teams, meet these assumptions. Results can improve

3 Literature Review

by adopting a bounded rationality approach closer to team behaviour. This is left as future work.

4 GTPI: A Game-Theoretic Approach to Process Improvement

Modern society revolves around software. We use it to communicate, understand global warming, operate machines, and decide what to buy. Like the human beings who write it, software is fallible. Software engineers rely heavily on tools to write and maintain robust software: tools that find bugs, repair bugs, or help developers avoid introducing them in the first place. Development tools are important, but so are the processes that use them. In the words of Linus Torvalds, “All the really stressful times for me have been about process: they haven’t been about code” [56]. Despite their importance, researchers and practitioners have lacked the tooling to build bespoke formal, yet still intuitive and explainable, models for software processes. Instead, they had to rely on generic, coarse-grained models or models difficult to elicit and often hard to understand.

Developers, customers, and managers cooperate and compete to write software. These interactions define a software process. Examples include effort estimation, where managers underestimate to win a project bid, while technical leaders overestimate to minimise risk [57; 58], and defect prioritisation, where end users exaggerate the importance of their bugs to obtain their fix quickly, while the engineering team needs accurate priorities (business value, not technical severity) to maximise the value delivered per release [12].

These software processes can be viewed as games. When we model software processes as games, we can compute their Nash Equilibrium (NE) to diagnose their problems and prescribe fixes when needed. I believe that most software processes are emergent phenomena that arise to solve problems. Since they

4 GTPI: A Game-Theoretic Approach to Process Improvement

are not designed, many are suboptimal: they misalign a player's payoffs with the overall goals of a process, permitting, even encouraging, players to act in ways that undermine collective goals. These games have an undesirable NE. I am proposing that process engineers [11] be game designers, who craft software processes considering cooperation incentives and conflict avoidance mechanisms.

Despite game theory's obvious applicability to software processes, software engineering researchers have not fully exploited game theory, limiting their analysis to idealised models. The reason is that software processes are inherently temporal: they are not one-off events; they comprise interactions over time. Game theory models these scenarios with extensive form games. This formalism is intricate and its analysis must cope with exponentially large game trees. Even constrained versions of poker have trees with 10^{18} leaves [59].

In this chapter, I use abstractions to build tractable game-theoretic models of real-world software development scenarios. *Empirical game-theoretic analysis* (EGTA) is an instance of a large set of abstractions for dealing with large games [7]. EGTA relies heavily on data for simulation input analysis and output validation. Fortunately, software engineers can now easily access a wealth of data from building and using software products [60]. The proposed solution — called Game-Theoretic Process Improvement (GTPI) — relies on this data and EGTA models to improve software development processes (section 4.2).

Many tasks can be partially completed. At one end of the scale, one can use a quick and dirty short term fix; at the other, one takes the time and care to find and implement a complete and lasting fix. Neither approach is always better. Quick code can help a development team beat a competitor to market, but, taken too far, turn a codebase into an unmaintainable mess. When developers are not racing, we prefer more deeply considered code. Some development workflows settle into a suboptimal point along this continuum. Lavallée and Robillard observed that some software teams had a tendency to prefer quick-cheap solutions over time-consuming ones, incurring technical debt due to fear of exceeding their budget [61]. In this chapter, I use this problem to showcase GTPI.

4 GTPI: A Game-Theoretic Approach to Process Improvement

Table 4.1: Pay-off matrix for the freelancer’s dilemma: each cell contains the payoff in dollars obtained by Freelancer A and Freelancer B given their choice to cooperate or not.

	<i>B: cooperate</i>	<i>B: not cooperate</i>
<i>A: cooperate</i>	$A = \$150, B = \150	$A = \$50, B = \200
<i>A: not cooperate</i>	$A = \$200, B = \50	$A = \$100, B = \100

4.1 Motivating Example

Consider a software project that hires two freelance engineers: a frontend engineer and a backend engineer. Their contract stipulates \$50 upfront and an additional \$50 upon completion. If the project goes live before the deadline, they each receive an additional \$50. They have two actions: to cooperate and work together or to work individually, not cooperate and ignore each other. If they cooperate, they will certainly finish before the deadline. Otherwise, they will certainly finish their individual tasks, but lack of integration means the project will not go live. If only one cooperates, the isolationist will finish their task quickly and free themselves to take another contract, while the cooperating freelancer will not complete their task. I represent this game with a *payoff table* — shown in Table 4.1 — that contains payoff information per player given the actions they performed.

When one freelancer cooperates and the other does not, the outcome is unstable since only the uncooperative player has adopted the best response, while the cooperative freelancer would be better off not cooperating. The Nash equilibrium for both players is to adopt the same strategy: ignore each other. Any deviation from this strategy allows their colleague to take advantage. Hence, Nash equilibrium analysis predicts the freelancers work independently and obtain \$100 each without completing the project. However, in the unstable scenario where they work together, they obtain \$150 and a satisfied client with a completed project. This dilemma faced by the freelancers is an instance of the prisoner’s dilemma.

To model the freelancers working on multiple projects over time, a classic game-theoretic approach is to use an extensive form representation (section 2.3). Figure 4.1 shows a game tree for a multi-project freelancer’s dilemma. The size of an extensive form game tree can become inconveniently

4 GTPI: A Game-Theoretic Approach to Process Improvement

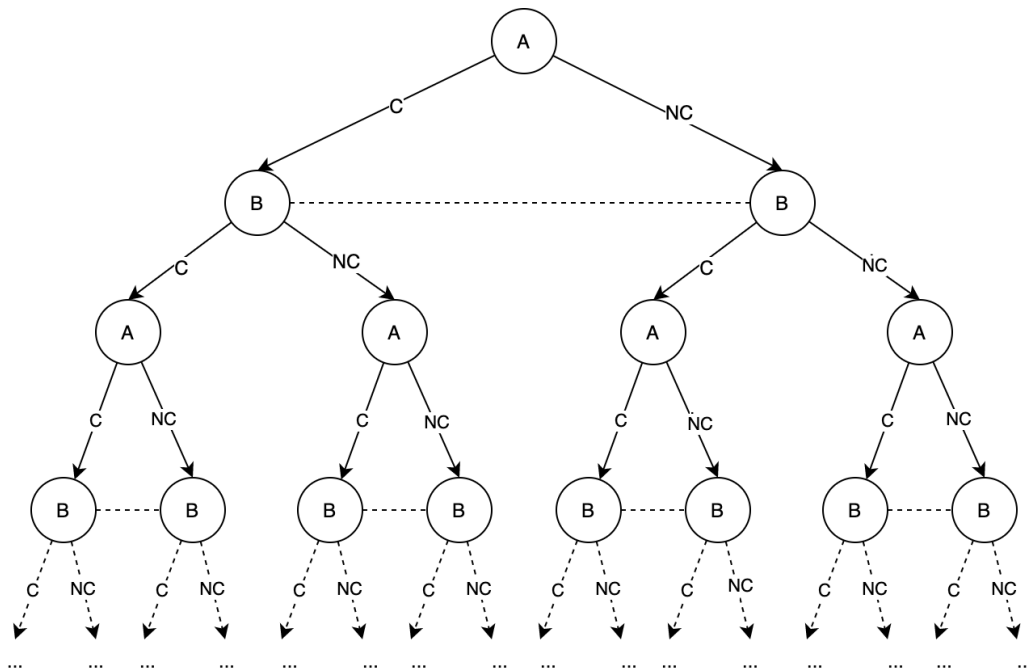


Figure 4.1: Extensive form game for a multi-project freelancer's dilemma: this figure contains the part of the tree corresponding to the first two projects. Nodes in the tree correspond to players (Freelancer *A* and Freelance *B*) and edges to actions (*C* for cooperation, and *NC* for no cooperation) The size of this representation grows with the number of projects, freelancers and actions available to them. EGTA is crucial for managing this explosive growth (section 4.2).

large when dealing with real-world scenarios. These sizes quickly become unmanageable for Nash equilibrium calculation algorithms.

Software development is inherently temporal: time is a critical dimension of most software processes. For example, the Scrum process framework divides development in time boxes, called *sprints*, where development tasks are assigned and prioritised. At the end of each sprint, the status of the project is assessed and the project plan is refined accordingly. Other process frameworks, like the Unified Process, also propose guidelines for sequencing tasks and organising them over time. The extensive form is more suitable for representing software processes than the normal form, given static games inability to handle time [15]. However, real-world software development — with large teams working over multiple releases — generate immense game

4 GTPI: A Game-Theoretic Approach to Process Improvement

trees. Hence, abstraction is needed. *Empirical game-theoretic analysis* (EGTA) is one of these abstraction approaches [8] [9]. EGTA reduces the action space by restricting it to a set of *heuristic strategies*, which are a subset of strategies that are of interest to the game designer. In the freelancer scenario, let us assume that the freelancers work together for an undetermined number of projects under the contractual conditions of [Table 4.1](#). Strategies can go from “not cooperate on any project” to “imitate my colleague’s last action”. In this strategy space, the game designer needs to select a subset of these strategies. Heuristic strategies can be obtained from data or hand-crafted following first principles, based on the properties of the system under study [9].

EGTA uses a simulation-based *heuristic payoff table* instead of the game tree of the extended form representation. This table has an entry for each action combination, where the actions available for each player are heuristic strategies. Each entry also contains the expected payoff for each player given the actions they perform, which are obtained through simulation. The heuristic payoff table can then be processed by a game solver to obtain its NE.

4.2 An Introduction to GTPI

Game-theoretic Process Improvement (GTPI) is the proposed process improvement framework. It relies on EGTA abstractions to model software processes, diagnose process anomalies and prescribe solutions to remove them. GTPI has two phases — software process modelling and software process improvement — and each phase has two stages. The output of the software process modelling phase is an empirical game model of the process under analysis, while the output of the software process improvement phase is a deployed new process. GTPI is summarised in [Figure 4.2](#).

4.2.1 Software Process Modelling

Anomalous processes are often itchy: something about them mystifies or annoys their participants. Some behaviours that are apparently irrational

4 GTPI: A Game-Theoretic Approach to Process Improvement

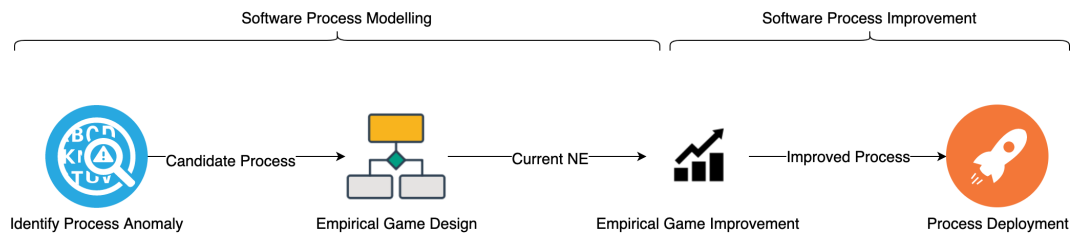


Figure 4.2: The GTPI approach starts by detecting a candidate process that exhibits an aberrant behaviour. Such a process is then modelled using EGTA and the Nash equilibrium is obtained. In case the equilibrium is not the one desired, the EGTA model is updated iteratively until a desirable one is obtained. Finally, the improved process is adopted by the team.

appear due to incorrect incentives. *Identifying process anomalies* is the first step of the approach.

Workers of every discipline, when approaching a task, need to choose between “cutting corners” or not. Technical debt is a software development instance of this seminal problem [61]. Practitioners [62; 63] and researchers [61; 35] have identified that software teams have an incentive to deliver sub-optimal but quick solutions — like kludges — instead of optimal but time-consuming ones, like proper fixes. One of the causes is the pressure put on teams to deliver on time and under budget, which triggers them to maximise the number of features delivered regardless of quality. Individual developers too balance getting things done quickly versus getting them done right. Developers face this dilemma repeatedly and, of course, they seek a Goldilocks solution¹.

The next step is the *empirical game design*, where we model the process to improve. Underlying Figure 4.3 are two models. One is a simulation model designed to capture real world behaviour. To the simulation model, we apply heuristic strategies to produce the other, an empirical game-theoretic model in the form of pay-off table. Simulation of software processes is a well-developed area, with plenty of options and paradigms [11]. I adopted discrete-event simulation because it is easy to reproduce and evaluate [64].

The parameters for the simulation model of technical debt are described in

¹“Goldilocks and the Three Bears” is a 19th century fairy tale, in which a girl, given a series of three-way choices, consistently chooses the one between the extremes.

4 GTPI: A Game-Theoretic Approach to Process Improvement

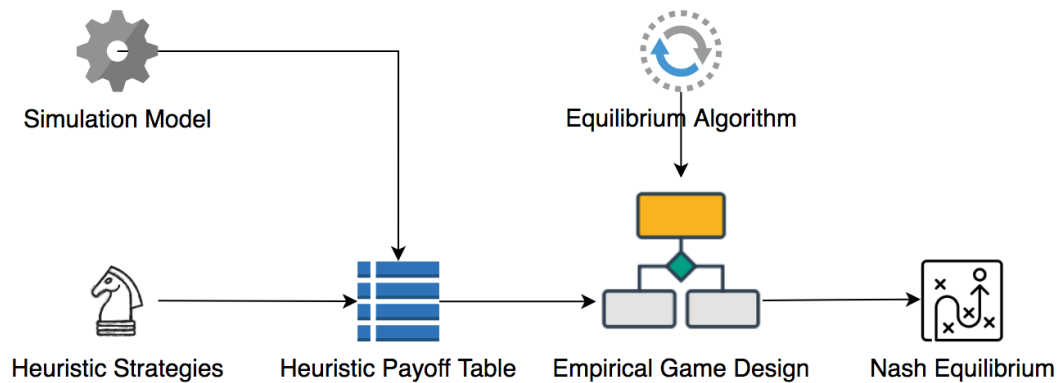


Figure 4.3: Empirical game design: the process engineer builds a heuristic payoff table using heuristic strategies as actions: The entries of this table are expected payoff values obtained via simulation. They can later use any algorithm to calculate the NE of the empirical game, once the payoff matrix has been built.

Table 4.2. Technical debt can arise because developers write sloppy code under pressure. Game theory suggests the answer is incorrect incentives, so the first modelling question to ask is “How are developers rewarded?”. The answer is not directly money or reputation, so I look to the output of the process; F_i features per release. I use F_i as the game’s payoff function below. F_i is governed by D , the number of developers and the release periods of N days. Kanban is popular among agile teams, so the development team in the model works a Kanban board with three columns: To-Do, In-Progress and Done². Tasks appear as To-Do’s with probability I . On average, a task takes T days to reach “Done” and increase F_i .

Technical debt is a *tragedy of the commons*: the codebase quality, the shared resource or *common*, progressively degrades as the team pushes kludges. So, I define the model to make kludges faster to code than fixes at the expense of codebase quality and with a higher risk of rework. To this end, I model the probability of reworking a “Done” task as due to a bug. Fixes take T_f time and are R_f likely to require rework, while kludges take T_k time and are R_k likely to require rework. Kludges reduce the quality of the codebase and increase the average resolution time by Q_k . Of the myriad

²Kanban boards visualise workflows. They have several columns and work items flow between them [63].

4 GTPI: A Game-Theoretic Approach to Process Improvement

Table 4.2: Input and output variables of the simulation model of technical debt: the output variable F_i corresponds to the payoff function of developers. The process engineer needs to work with the customer to identify the relevant variables of the process under analysis.

Input	Description
T	Resolution time for work items in days.
T_a	Resolution time for work items coded with action $a \in \{f, k\}$.
I	Work item arrival probability at the “To-Do” column per day.
S_i	Heuristic strategy adopted by developer i .
D	Number of developers available.
R	Rework a work item after it is placed in “Done”.
R_a	Rework a work item coded with action $a \in \{f, k\}$ after it is placed in “Done”.
N	Iteration duration in days.
Output	Description
F_i	Work items finished by developer i .

actions developers can take, the simulation model gives a developer only two: write a kludge or code a time-consuming fix, as [Figure 4.4](#) shows. The simulation model must be validated against data with respect to the targeted behaviour, which is defined in cooperation with the customer. In essence, one statistically compares the simulation outputs with actual process data [65]. Consolidating data across sources is non-trivial and time-consuming: I discuss how one might approach this problem in [section 4.3](#).

The EGTA approach also requires a set of heuristic strategies. These strategies model player behaviour, and the game designer employs mechanism design to control their adoption in the empirical game improvement step. Ideal candidates are behaviours we want to encourage or discourage. To obtain the strategies players actually adopt, we extract data from bug tracking systems, source code repositories, eliciting them from the customer or domain experts, or other relevant data sources. When the available data is insufficient, we turn to experts to expand the strategy catalogue, keeping in mind that the number of strategies determines the size of the game.

The EGTA model of technical debt uses two strategies that I believe are worth

4 GTPI: A Game-Theoretic Approach to Process Improvement

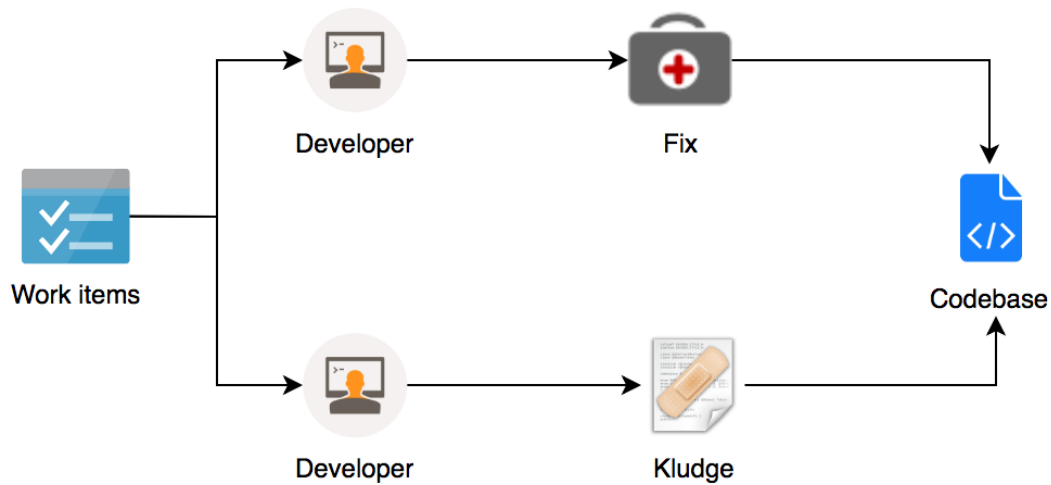


Figure 4.4: Developers in the technical debt simulation: the implementation of work items can be fixes or kludges. Fixes demand more time but are less likely to require rework. Kludges are quick but are more likely than fixes to be reworked. Also, kludges negatively impact codebase health.

exploring. The first commits proper fixes up to a week before the release, when, due to pressure, it shifts to pushing kludges. This behaviour favours fixes over kludges, most of the time. The second is sensitive to the work items accumulating in the “To-Do” column: when the “To-Do” backlog exceeds 2 items, it starts committing kludges. In a heavily loaded project, this strategy favours kludges. While actual player strategies will be some more subtle mix of fix and kludge, I have picked two extremal strategies to study how to reduce kludging. In focusing on extremal strategies, I am simply seeking to make the process resilient to unwanted behaviour, following a long tradition in mechanism design [66].

This model of technical debt is simple by design in order to capture fundamental behaviour. The key assumption is that the aberrant behaviour that I seek to capture has such strong signal that a simple model *can* capture it. Also, the explanatory power of simple models greatly favours process adoption. I believe customers and practitioners will be more positively inclined towards process changes justified using concepts they can understand. Further, capturing the essence of the phenomena under study in a simple

4 GTPI: A Game-Theoretic Approach to Process Improvement

model generates more flexible models. Indeed, cutting corners is not unique to software. With just a few modifications, we could easily port the simulation model to non-software domains. Below, I show how the model suggests a simple, inexpensive solution to budget-driven technical debt: “All models are wrong but some are useful”, G. Box.

From the simulation model and the heuristic strategies, I obtain the heuristic payoff matrix by simulating each of the table entries and including the expected values over a number of iterations. This payoff matrix represents the game-theoretic model. Once I have the payoff matrix, I obtain its equilibria. Nash equilibrium calculation is a vast and diverse area with many options [19]. I rely on the Gambit game solver: a software tool that contains ready-to-use implementations of equilibrium calculation algorithms, facilitating quick experimentation [67].

For demonstration purposes, let us model a small team working under heavy load, with $D = 2$ and $I = 1.0$. For realistic parameter values, I rely on information extracted from the Eclipse Platform’s bug repository. An annual release has $T = 29.79$, $N = 360$, and a raw rework probability of $R = 0.069$ [68]. I want to emphasise that I do *not* assume that the Eclipse platform is kludge-prone. Kludges are more prone to rework than fixes, so I adopt $R_k = 1.05 \times R$ and $R_f = 0.9 \times R$. To reflect the negative impact of kludges on bug resolution time, I set $Q_k = 0.05 \times T$. Finally, kludges take less time to code than fixes, hence $T_k = 0.75 \times T$ and $T_f = 1.1 \times T$.

I simulated each heuristic payoff table entry for 100 releases and recorded the average F_i per developer. Table 4.3 shows the resulting heuristic payoff matrix. Gambit finds a single equilibrium where both developers adopt the kludge-intensive strategy. This outcome matches the itchy behaviour identified at the first stage of GTPI, and the empirical game model offers an explanation: developers have an incentive to kludge instead of generating proper fixes.

4.2.2 Software Process Improvement

Following design, we proceed with the *empirical game improvement* step. The goal is a process whose corresponding empirical game has a single

4 GTPI: A Game-Theoretic Approach to Process Improvement

Table 4.3: Payoff matrix after the empirical game design stage in Figure 4.2: it has a single Nash equilibrium where developer A and developer B adopt only the kludge-intensive heuristic strategy.

	<i>B: fix-intensive</i>	<i>B: kludge-intensive</i>
<i>A: fix-intensive</i>	$A = 9.80, B = 9.80$	$A = 8.34, B = 10.77$
<i>A: kludge-intensive</i>	$A = 10.77, B = 8.34$	$A = 9.06, B = 9.06$

Table 4.4: Payoff matrix after the empirical game improvement stage in Figure 4.2: it has a single Nash equilibrium where developer A and developer B adopt only the fix-intensive heuristic strategy.

	<i>B: fix-intensive</i>	<i>B: kludge-intensive</i>
<i>A: fix-intensive</i>	$A = 9.79, B = 9.79$	$A = 8.40, B = 7.77$
<i>A: kludge-intensive</i>	$A = 7.77, B = 8.40$	$A = 6.83, B = 6.83$

equilibrium where the heuristic strategies we believe beneficial have high probability. To remove technical debt, we seek an equilibrium where the fix-intensive strategy has a significantly higher probability than the kludge-intensive one.

The analysis of [Table 4.3](#) shows that the advantage of kludges — a reduced coding time — is worth their cost — a higher rework probability — so it becomes dominant at equilibrium. The absence of a code quality control mechanism before committing makes kludges very cheap, which translates in a progressive deterioration of codebase quality. Thus, I knew any solution should make kludges more expensive. Adopting pair programming can accomplish this but, due to its cost, I did not consider it. I posited that adding a part-time experienced code reviewer who can detect 10% of the kludges would be sufficient and also cost effective. So, I updated the simulation model with $R_k = 10\%$ and use it to produce payoff values for a new heuristic payoff table. In this configuration, Gambit found three equilibrium profiles and in one of them both agents perform the kludge-intensive strategy, which was far from ideal. However, when I set $R_k = 25\%$ and rebuild the payoff matrix, shown in [Table 4.4](#), its equilibrium analysis produced the desired outcome: a single equilibrium where both players adopt the fix-intensive strategy.

The last step is the *process deployment*. The results of this analysis would be

pointless without a feasible deployment strategy. The improved process is deployable: most software teams already review commits. Since we only require a 25% kludge detection accuracy, the reviewer needs only to perform a lightweight quick-pass.

4.3 Practical considerations

Adopting GTPI is challenging; data gathering, technical validation, and securing customer and performer acceptance is hard. I will use the following scenario to illustrate these challenges.

Imagine a large organisation whose flagship software product has been under constant development for years. Even though software products tend to grow by adding features, over time they also lose some of them. These lost features obsolete some tests. The problem is that this subset is largely unknown and, in practice, rarely removed. The main reason is that the engineering time needed to identify these useless tests with certainty is significant. Also, mistakenly removing a useful test can cost a software engineer their job. There are some — perhaps folkloric — stories of engineers losing their jobs because they removed a test that would have detected a security vulnerability. These factors explain why potentially useless tests are rarely removed, as illustrated in [Figure 4.5](#). This wastes testing resources.

I call this scenario the *tragedy of the test suite* since it constitutes a tragedy of the commons: the test infrastructure is a shared resource that self-interested software engineers spoil by constantly adding tests without removing unneeded ones. Over time, maintaining and running the test infrastructure becomes more expensive, and test execution takes more time, which is harmful for the engineering team and the organisation as a whole. Adopting a test prioritisation technique can mitigate the problem but it does not in general remove it. Ideally, when software engineers modify, or remove, functionality that makes some tests redundant, they should proceed to remove them from the test suite. Right after performing the change, engineers are in good position to identify the obsolete tests, when compared to a spring cleaning approach done later. I believe it is more efficient, and therefore cheaper, to incentivise software engineers to keep the test suite clean and correct. GTPI

4 GTPI: A Game-Theoretic Approach to Process Improvement

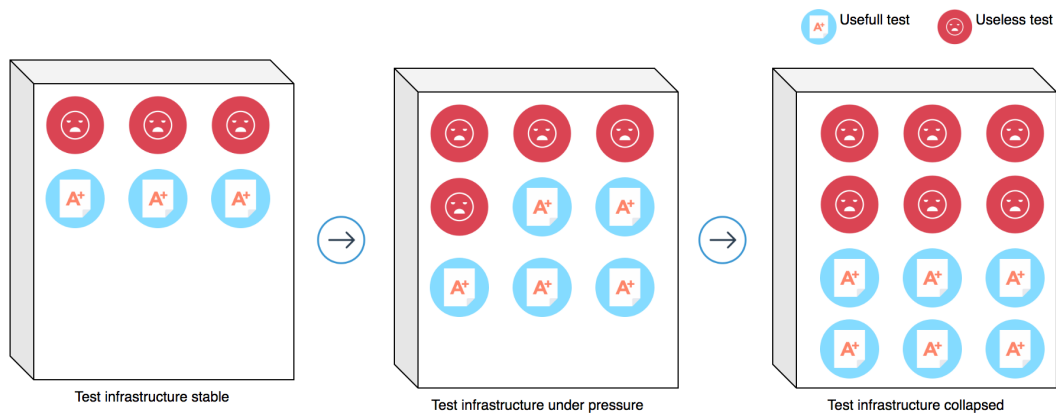


Figure 4.5: The tragedy of the test suite: job insecurity makes software engineers deliver more features without removing potentially useless tests. Over time, this behaviour can cause the collapse of the test infrastructure.

is a good fit for approaching this problem from a strategic perspective, given the large number of engineers in the company and the diversity of their behaviour.

4.3.1 Data Gathering

Data gathering is hard in general. Depending on the complexity of the proposed simulation model, the process engineer will need to mine from the source code repository data regarding active tests, potentially useless tests and their evolution. Also, the pay-off calculation requires data regarding how often a deleted test fails to discover a future bug and the consequences for the developer in case this happens. Obtaining this kind of data can be time-consuming or even infeasible. It might be the case that the model requires data that no one thought about collecting before the process improvement effort. The process engineer needs to start its analysis by designing a plan on how to build the dataset required for empirical game modelling, and backstop plans — like proxies — in case the organisation has not collected the required data.

In the tragedy of the test suit scenario, data identifying obsolete tests is unavailable because the whole problem evaporates if the obsolete tests are

known. Test prioritisation can be an effective proxy for test utility, assuming it has been empirically validated [69]. The intuition is that the test prioritisation outcome includes more useful tests than obsolete ones. Although is an imperfect proxy, it can serve as an initial baseline until more elaborate approaches are put in place.

4.3.2 Technical Validation

As discussed in subsection 4.2.1, after acquiring an adequate dataset the process engineer must build a simulation model of the process under study. Software process simulation is a well-known technique in the process engineering domain [11; 10], and there is a rich literature on the topic. It is critical in GTPI that the simulation model reflect the underlying process with fidelity. The empirical game improvement stage in Figure 4.2 requires perturbing a *validated* simulation model to obtain a process intervention that removes the undesired behaviour. Improving an imprecise model is a waste of time, so the validation of the simulation model is of the utmost importance. The validation of the model has two dimensions: technical and social. I address the *technical validation* dimension first.

In the tragedy of the test suite scenario, the simulation model needs the test execution time distribution as an input. A goodness-of-fit test can be applied to the data to see if, for example, it behaves according to an exponential distribution [65, Chapter 9]. Given that the pay-off function depends on the number of successful test executions, a statistical test can be applied to samples of the simulation output to verify if they reflect what is observed in the data [65, Chapter 10]. Simulation model validation is a hard problem, and several iterations might be needed in order to obtain a model with the required accuracy. Once the empirical game is ready, the process engineer can use any software package to calculate its NE. The obtained equilibrium also needs to be technically validated against the process data. The NE obtained from the model of the tragedy of the test suite needs to match the useless test accumulation observed in the organisation.

4.3.3 Securing Acceptance

The empirical game model also needs to be *accepted by stakeholders*: they must agree that the model accurately captures the process. This is essential since GTPI's recommendations are learned from and justified by interventions in the model, as part of the mechanism design process. The key challenge here is convincing the customer that the proposed model does not oversimplify.

Once the game-theoretic model is validated and accepted, we can proceed to the empirical game improvement stage. This stage requires exploring the space of potential process improvement interventions with a desirable NE in their corresponding game, as discussed in [subsection 4.2.2](#). Building an empirical game can be computationally costly — each pay-off matrix cell needs to be simulated by several iterations — so the search space traversal needs to be done carefully.

A suggested process improvement intervention can fail if it is not accepted. Going back to the tragedy of the test suite, if the improved process requires a no-penalty policy for removing obsolete tests, companies can resist not punishing the unlucky developer who removed the wrong test. Cost is also an important factor: if the cost of the proposed intervention is low, convincing the stakeholders about its implementation should be easier. Given that game theory has been sparsely used in process engineering, I believe stakeholder acceptance is the biggest challenge to GTPI adoption.

The process deployment stage also proposes acceptance challenges regarding *process performers*, the people doing the actual work. The process engineer needs to ensure that the adoption of the proposed practice goes as smoothly as possible. Tool support can be crucial to this end: if the tragedy of the test suite is improved by minimising the cost of test removal, code analysis tools would help identify obsolete tests and the impact of its removal. The process deployment step needs to be monitored constantly, and perform corrective measures if needed. Without adoption, process improvement fails.

4.4 Related Work

GTPI has two phases: 1) in software process modelling, we assess the process in place by analysing the Nash Equilibrium of its game-theoretic model (subsection 4.2.1) 2) then in software process improvement, we use mechanism design to propose and validate an improved process (subsection 4.2.2). In the software process literature, a *descriptive* process model is used to describe the current practice and a *prescriptive* model describes the improvements required for the process to meet its requirements [11]. Hence, GTPI includes both a *descriptive* model of the current process and a *prescriptive* model as an output of mechanism design.

The *Elicit* method is an example of a descriptive process model: it is a process elicitation method that includes guidelines, a modelling notation and tool support [70]. *Multi-view process modelling* is another descriptive process method, that proposes building a descriptive process model by integrating the process views of individual process performers [71]. Both approaches can not represent conflicts among process performers, while GTPI relies on the equilibrium analysis of the game-theoretic model for this purpose.

Prescriptive process models fall in two categories [11]: the scope of a *life-cycle* process model is the complete development process of a software product — like in the Unified Process [72] and in Cleanroom Software Engineering [73]— while the scope of an *engineering* process model is a specific practice within this process. Statistical testing [74] and hybrid cost estimation [75] are examples of engineering process models. In this chapter, I use GTPI to produce an *engineering* process model to address budget-driven technical debt. As an outcome, I suggest implementing a code review process to minimise the the number of kludge commits. Bringing strategic reasoning into process analysis, to later produce a new actionable process is a key contribution of this thesis.

Both the development of a prescriptive and a descriptive process model rely on notation to characterise the software process under study. The Software Process Engineering Metamodel (SPEM) notation [76] can represent roles, tasks and work-products. In GTPI, we use game-theoretic notation — like payoff matrices— for strategic interactions. Yu *et al.* [28] had already proposed modelling constructs for representing strategic reasoning in software

4 GTPI: A Game-Theoretic Approach to Process Improvement

teams, through a strategic dependency model and a strategic rationale one. These models do not provide the numerical information needed to guide mechanism design. For this purpose, the equilibrium analysis of GTPI's payoff matrices produce probabilities for strategies at equilibrium.

The software process literature references benchmark-based approaches to software process improvement. In them, the current software process is compared to a reference model to later assign a maturity level. Two of the most important reference models are CMMI [77] and ISO/IEC 15504 (SPICE) [78]. Both contain assessment guidelines for a software process. GTPI is not benchmark-based: it guides process improvement without a reference, but through mechanism design principles and equilibrium analysis. Benchmark-based approaches, like CMMI, assign maturity levels to process areas that group organisational processes. In GTPI, we analyse process performers at the software practice level. This allows us to make an assessment at a finer granularity than the levels supported by CMMI and SPICE.

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

People often break projects into tasks, then prioritise them. More important tasks, because they produce something of value or because other tasks depend on them, have higher priority. In an issue tracking system with shared prioritisation tooling, QA engineers, testers, or project managers assign a priority label to tasks; this label informs a project team about the fixes or features the next release should incorporate. *Priority inflation* occurs when an assessor increases the priority of an issue above their true assessment, so that tasks they care about are delivered more quickly [12; 79]. By undermining priority labels, priority inflation can misallocate developer time.

I contend that priority inflation hampers software development for three reasons: 1) Despite the fact that most teams would prefer to work on important, unclaimed tasks first, I found that teams using GitHub tend not to use its shared prioritisation tooling when its use is optional (subsection 5.2.1); 2) I surveyed software development professionals who reported that priority inflation is frequent and significantly misallocates resources (subsection 5.2.2); and 3) Industry leaders have deployed processes to triage bug reports and correct inflated priorities [80; 81].

Standard game-theoretic models of real-world scenarios become intractable when dealing with many players or numerous strategies [8]. Having a dataset with hundreds of bug reporters and multiple strategies represents a challenge for classic game-theoretic approaches. To overcome this, I used GTPI (chapter 4) to understand priority inflation, of both new features and bug repair

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

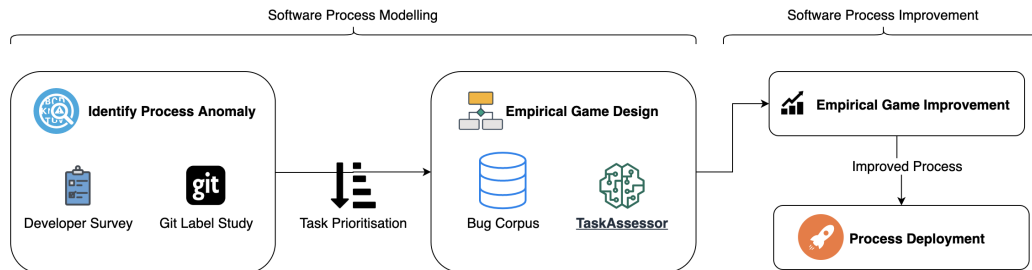


Figure 5.1: GTPI’s software process modelling phase for priority inflation: to identify the process anomaly, we conducted a developer survey and a study on prioritisation in GitHub labels. *TaskAssessor*, the empirical game model, was built based on a bug resolution corpus from the Apache Software Foundation.

tasks. As shown in Figure 5.1, in this chapter I elaborate on the software process modelling phase of GTPI for addressing priority inflation. *TaskAssessor*, the empirical game model of priority inflation, is the output of this phase. It is the first tool to apply empirical game-theoretic analysis (EGTA) to software engineering.

TaskAssessor simulates a process with prioritised tasks. Given an EGTA model of that process, it computes the model’s Nash equilibrium to diagnose problems. A task prioritisation process immune to priority inflation would produce a model with a single equilibrium where all the players — people filing tasks or reporting bugs — adopt a strategy under which players honestly prioritise tasks.

To build the EGTA model, I used 42,620 issues collected from the JIRA issue tracker of the Apache Software Foundation. From them, I extracted prioritisation strategies. JIRA priority labels combine technical severity and business value or risk of an issue. End users often confuse and conflate the two [82]. Focussing on priority inflation means that I am concerned with the proportion of resolved tasks that are, in fact, high priority. This is relevant to software development teams that seek to maximise the number of tasks resolved. So, I validated that the simulation component of *TaskAssessor* is sufficiently accurate with respect to the proportion of high priority tasks completed (subsection 5.3.4).

5.1 The Assessor's Dilemma

I now showcase game-theoretic modelling in a software development context and use it to illustrate priority inflation at Foo Inc, a small or medium-sized enterprise (SME).

Economic models represent behaviours where human motivation can be expressed as a function of price [83]. Neoclassical economists like Alfred Marshall believed that money is also a suitable measure for intangibles like desires and aspirations. The magnitude of a person's preference towards a product or service can be approximated as the amount of money this person is willing to pay for it: this applies to both smartphones and to political platforms. The ability to approximate motivations — although imperfectly — with real numbers is what makes economics “the most exact of social sciences” [83]. In particular, game-theoretic models require expressing a player's payoffs as real numbers [13]. In the example below, I meet this requirement by assuming Foo Inc has a bonus policy tied to bug fixing measures, like the companies studied by Laplante and Ahmad [84].

Foo Inc uses an Enterprise Resource Planning (ERP) system for its daily activities. Alice and Bob work in Foo Inc's quality assurance team. They report bugs to a development team that cannot fix all known bugs, so Alice and Bob are competing for development time. Foo Inc wants its developers to fix more important bugs first, so it rewards QA engineers who report higher priority bugs that the developer team fixes with a higher bonus [84].

Foo Inc's finance manager tells Alice that the ERP system has two problems — the cash management module produces incorrect figures and the financial consolidation module is too slow. The first problem is severe and costs the company \$10,000/day; the second is inconvenient, costing only \$1,000/day. These figures are arbitrary, but consistent with the cost these bugs might impose on an SME, like Foo Inc. I picked them to separate a severe bug from a trivial one by an order-of-magnitude. At the same time, Foo Inc's Human Resources Manager informs Bob that the payroll module crashes every day and that the learning module misplaces images when accessed from mobile devices. The payroll bug is high priority, costing \$10,000/day; while the learning module bug is minor, costing only \$1,000/day since Foo Inc does not yet widely use mobiles.

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

Table 5.1: Pay-off matrix for the assessor’s dilemma: each cell is the payoff Alice (A) and Bob (B) obtain, under the combination of actions each takes.

	<i>Bob: accurate</i>	<i>Bob: inflate</i>
<i>Alice: accurate</i>	A = 125, B = 125	A = 100, B = 150
<i>Alice: inflate</i>	A = 150, B = 100	A = 112.5, B = 112.5

At Foo Inc, a fixed, high priority bug increases the reporting QA engineer’s bonus by \$100, while the resolution of a trivial bug increases it by \$50. I assume that Foo Inc has found that these values are sufficient incentive and within the bonus it is willing to pay. For the next release, the development team can only fix three of the four bugs that Alice and Bob report. Thus, they have both the means and the motivation to inflate their bugs’ priorities. [Table 5.1](#) shows Alice’s and Bob’s expected bonuses, assuming that developers resolve higher priority bugs first and that bugs with the same priority have the same probability of being fixed. For example, when both Alice and Bob inflate their priorities, all four bugs are labelled high priority and have the same probability of getting fixed. In this case, Alice’s and Bob’s expected pay-off is

$$0.5(0.5 \times \$100 + 0.5 \times \$50) + 0.5(\$100 + \$50) = \$112.5.$$

Let us analyse [Table 5.1](#) from Alice’s perspective. If Bob accurately prioritise his bugs, Alice’s best option is to inflate hers since she would obtain \$150 instead of the \$125 she would receive if she too were honest. If, instead, Bob inflates his bugs’ priorities, Alice’s best option remains inflating, since her bonus would be \$112.5 *vs.* \$100. Thus, Alice is better off inflating no matter what Bob does. This same analysis symmetrically holds for Bob. In game theory, this outcome is the *Nash equilibrium* of the game: no player has an incentive to change their actions in response to any other player’s actions.

This Nash equilibrium is bad for Foo Inc: it represents a bug repair process that encourages testers to inflate priorities and misallocate developer time. Specifically, the Nash equilibrium entails 0.5 probability that one of the high priority bugs is not fixed in the next release. In monetary terms, the equilibrium scenario reduces costs only by \$16,500/day, not the \$21,000/day that could have been achieved.

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

Also, Table 5.1 shows that, if Bob and Alice honestly reported priorities, they would be better off than if they took the actions leading to the equilibrium — \$125 *vs.* \$112.5. Rational play, however, dictates priority inflation; hence, Alice and Bob face a *dilemma*. I call this game the *Assessor's Dilemma*, since it is an instance of the *Prisoner's Dilemma* [85].

5.2 Identifying the Process Anomaly

Prioritisation is challenging and important, since it drives how time, money and energy are spent [86]. Thus, I take for granted that prioritisation is essential for efficient and effective bug repair. Without prioritisation, one tends to fall into the trap of neglecting important tasks for the merely urgent¹. Task prioritisation is also at the core of agile software development [63]. For example, Scrum development starts with a prioritised list of tasks created by the project sponsor, called the product backlog. In Extreme Programming, low-priority tasks — called slack — are included in each iteration to be discarded first in case of unexpected delays.

Shared prioritisation tooling (SPT) is a means for a team to share their assessments of task priority. The *shared* dimension of SPT requires the assessment to be public among team members, so they can both avoid overlapping and prioritise their work. By *tooling* I want to include only software solutions in this category. Mental prioritisation and pen-and-paper mechanisms do not constitute SPT solutions. Bug tracking systems — like Bugzilla and JIRA — have an SPT as part of their functionality. The SPT on bug tracking systems requires the inclusion of a measure of the importance of the bug filed. Bug importance has two dimensions: impact on system functionality — called *severity* — and impact on system value — called *priority* [88]. For example, a web application that crashes on Internet Explorer 5.0 has a high severity since functionality is lost, but low priority if the user base of that browser is minimal.

¹In a speech in 1954, Dwight D. Eisenhower said “I have two kinds of problems, the urgent and the important. The urgent are not important, and the important are never urgent.”. This quote is the basis of the Eisenhower matrix, to which I refer here [87].

5.2.1 Shared Prioritisation Tooling Adoption

SPT exploits collective intelligence to assess and focus work. To find out how widely SPT is used, we ask:

RQ: Do developers adopt shared prioritisation tooling?

The GitHub platform offers issue tracking functionality for software projects, but unlike other issue trackers like JIRA, it does not assign priority labels to issues by default. Instead, GitHub offers a generic labelling system, that developers can use to “signify priority, category, or any other information you find useful” [89].

Thus, to answer this research question, I performed an exploratory study over GitHub repositories. I collected GitHub projects and counted how many use GitHub’s labelling system as SPT. To determine whether a project is using labels as SPT, I applied two heuristics to its label’s text and colour: a project uses SPT 1) if the tokenisation and stemming of label text snippets intersects a bag of priority related words or 2) if its colour scheme suggests a priority ranking. For #1, I took the list of priority-related words from the field names and default priority rankings used by JIRA v6.3 (subsection 5.3.1), JIRA v6.4 [90] and Bugzilla [91]. For #2, I used the semaphore colours (red, yellow and green) to identify repositories that colour-encode priorities, as suggested by industry practitioners [92] [93]. I evaluated the heuristics by applying them to 60 GitHub repositories, sampled uniformly. I manually assessed these projects’ use of SPT and found that the heuristics have an F_1 score of 0.8 for repositories using labels as SPT.

I applied these heuristics to the labels I extracted from the 600 most forked repositories created between January 2017 and April 2018. The GitHub development model requires contributors to first create a copy of the repository via a fork, and then submit code contributions using pull requests [94]. The number of forks is a good indicator of project activity [95], as it is highly correlated with the number of contributors, number of commits, and number of branches [96]. I conservatively considered that the most forked repositories are more likely to use SPT, as they are more likely to have active teams that would benefit from the coordination that SPT provides. The finding is that

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

developers on GitHub, a pre-eminent developer collaboration site, rarely use its prioritisation facilities:

Finding: Only 6.3% of 600 uniformly sampled GitHub projects adopt shared prioritisation tooling.

To the extent to which GitHub generalises, development teams rarely use SPT when its use is optional. I argue that this is *not* evidence that shared prioritisation is unneeded, but rather evidence that existing SPT is not fit for purpose. In the next section, I describe a survey of developers who use SPT. The key finding is that priority inflation is, indeed, a problem. This may explain the initial finding that developers do not adopt SPT when given the option.

Threats to Validity: The study faces the standard external validity threat: it generalises only to the extent GitHub does. I uniformly sampled the most active GitHub repositories to mitigate this threat. Since I rely on the number of forks as a proxy measure for projects more likely to use prioritisation, the study faces a construct validity threat. This threat is mitigated by empirical studies that show the number of forks is correlated with project activity [95; 94].

5.2.2 The Cost of Priority Inflation

Previous finding shows developers generally tend not to use SPT when its adoption is optional. Here, I investigate whether priority inflation is the reason, via an online survey.

Participant Selection: I solicited survey participants from Apache Software Foundation (ASF) contributors and using social networks. I obtained ASF developer's contact information from the dataset built mining their JIRA and Git repositories (subsection 5.3.1). After contacting them by email, 39 contributors took the survey. My supervisors and I also shared the survey on

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

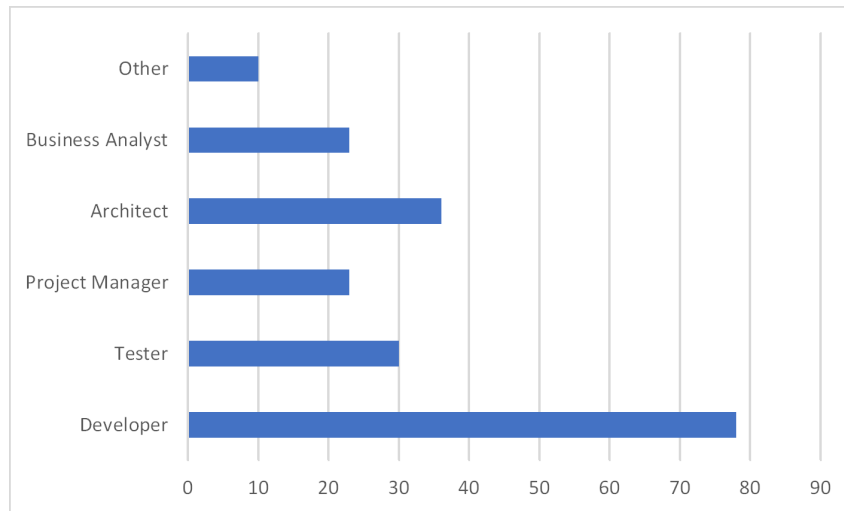


Figure 5.2: Role in the development process: the horizontal axis represents the number of participants per role. The survey allowed the selection of multiple roles per participant. This figure represents the answers of 152 software professionals.

social networks (Facebook and Twitter), obtaining 113 responses from software engineers. Convenience sampling² is appropriate given the exploratory nature of this study [97].

Figure 5.2 reports the roles covered in the software development process by the survey respondents in their organisations. The distribution of data in Figure 5.2 indicates that the sample is diverse, with an emphasis on the developer role. Since I did not know how many roles a respondent might perform, I allowed them to select more than one role. Only-developers are the largest group with 34 participants, followed by developer-architects with 13 participants. 3 participants reported performing all 5 roles included in the survey. Participants are also diverse with respect to expertise: the most experienced reported 10 years of industrial experience, while the most novice only 1. The average participant experience is 7.14 years. I posted complete survey responses at the project page [98].

²In convenience sampling, the main selection criteria is ease of collection.

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

Table 5.2: The questionnaire presented to 152 software engineers.

Bug Reporting Duties
R1: When you create a bug report, which fields do you usually fill out?
R2: If a bug report changes, who changes it?
R3: When does your bug report need updating?
R4: In what percentage of your bug reports does the priority field change?
R5: When writing bug reports, how often do you overstate the priority to speed resolution?
Bug Fixing Duties
F1: How useful are bug reports for fixing bugs?
F2: In what percentage of your bug reports are the priority fields useful?
Bug Prioritisation
P1: How many priority levels are typically supported by the bug reporting system(s) you use?
P2: How many priority levels do you think are needed for your current project(s)?
P3: Considering your current software project(s): How often is priority understated (or deflated) in bug reports?
P4: Considering your current software project(s): How often is priority overstated (or inflated) in bug reports?
P5: Is priority inflation/deflation affecting your work?
P6: If priority inflation/deflation is affecting your work, please detail how and what steps are being taken to address it.

Questionnaire: I surveyed using the questionnaire in Table 5.2. While question P6 is an open-ended question, the rest of the questionnaire is multiple-choice. The questions fall into three groups by role: bug reporting, fixing, and prioritisation. I instructed participants to only answer questions pertaining to roles they actually perform.

An indicator that priority inflation may have occurred is that the priority value filled by the original reporter was later corrected by another member of the team, such as a software developer or a business analyst. I formulated the survey questions about bug reporting to investigate this behaviour. In the limit, as priority inflation becomes the rule, the priority field of issues becomes irrelevant, since developers will learn to ignore it. The survey's bug-fixing questions seek to elucidate the relevance of the priority field information for bug fixers, when compared with other fields included in the bug report. I asked survey participants about the usefulness of a list of bug report fields, including "steps to reproduce", "attached screenshot" and, of course, "priority". Survey participants can then indicate, for each field, if they normally find useful information, or if they find blank, incomplete, or incorrect information. Finally, the bug prioritisation questions ask developers directly how prevalent priority inflation (or deflation) is, how it impacts their work, and what measures are taken to alleviate it.

RQ: How does priority inflation impact software development teams?

The bug prioritisation questions aim to discover whether the bug reporters

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

Table 5.3: Survey responses to questions about the frequency of priority inflation and deflation in the respondent's current software project.

Anomaly	Question	Never	Occasionally	Frequently
Priority Inflation	P3	11%	64%	25%
Priority Deflation	P4	20%	65%	15%

assign priorities that differ from their true assessment. Reporters can dishonestly over or understate bug priority. As seen in Table 5.3, 25% of the participants reported working on projects where priority inflation is frequent while another 64% reported that priority inflation occurs occasionally. Regarding priority deflation, 15% work on projects where the bug report priorities are frequently understated, while 64.63% report that deflation occurs occasionally.

31% of those who answered P5 affirm that understated/overstated priorities have a significant impact on their daily duties, while 50% of them believe the impact is minimal. P5 is inadvertently ambiguous: I contend that most readers would interpret it to be one-sided and only about negative impact, but I recognise that some may interpret it as two-sided. To address this, I analysed P6 in depth and found that from the participants that include an impact description, 82% reported a negative impact, resource misallocation being the most common response with 37%. These numbers show that many participants found that priority inflation has a negative impact on their daily activities.

Question P6 of the survey is optional. Among 65 responses, the most popular measures were the following: 1) 34% reported a *gatekeeping* procedure, where a third-party verifies the priority included by the original reporter. This implies that the reported priority might become irrelevant if the gatekeeper unilaterally overwrites it; 2) 12% mentioned *user training*, indicating the requisites and characteristics required by each level on the priority hierarchy. In summary:

Finding: In a survey of 152 developers, 31% of respondents reported that inaccurate priorities misallocated development effort, 25% stated that priority inflation occurs frequently in their projects, and 15% reported working on projects where priority deflation is frequent.

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

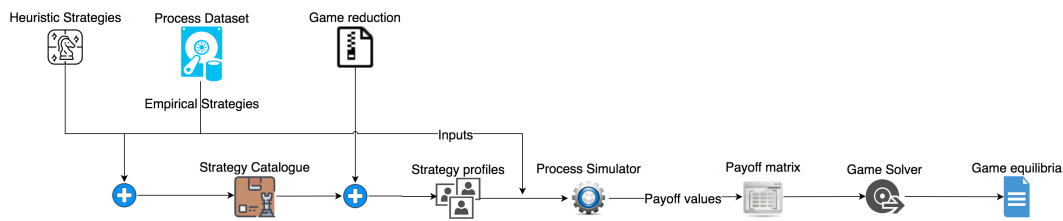


Figure 5.3: Empirical game design stage for *TaskAssessor*.

This finding suggests that priority inflation is a common problem in software teams adopting SPTs and that it has a negative impact on their daily activities. Using convenience sampling to recruit participants for the survey is a strong threat to this finding’s external validity. It is, however, standard practice in an exploratory study such as this [97].

5.3 Empirical Game Design

Figure 5.3 describes the empirical game design stage (subsection 4.2.1) of *TaskAssessor*, the approach for modelling priority inflation. It has three inputs. Two involve strategies, which players use to decide which actions to take. The game analyst must find sufficient process data from which to extract empirical strategies, strategies they observe players following in the data, and inputs for the process simulator. Heuristic strategies are obtained from domain experts. The game analyst can also propose heuristic strategies to test hypotheses about how participants interact in the game. The analyst must generate a reduced game to make the analysis tractable. They merge the empirical and heuristic strategies, then feed them, along with the relevant process data and the reduced game, to the simulator to compute the payoff matrix. Finally, they compute the Nash equilibria.

With Nash equilibria in hand, the game analyst compares them with the goals of the process they are analysing. When they apply *TaskAssessor* to ineffective processes, they expect to find mismatches between the desired equilibrium and a player’s equilibrium strategies. They diagnose how a player’s action set and incentives cause these mismatches, then use mechanism design to

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

Table 5.4: The *TaskAssessor* Corpus of Issues extracted from JIRA and GitHub.

Project Name	Drive-by R.	Engaged R.	Issues	Non-default
OFBIZ	151	95	5120	51.5%
CASSANDRA	281	116	7417	53.7%
CLOUDSTACK	115	99	7463	47.9%
MAHOUT	54	25	1044	36.8%
ISIS	15	5	1125	67.3%
SPARK	35	14	1330	44.4%

consider changes to the action set and rewards to reduce or eliminate these mismatches.

This section introduces and validates *TaskAssessor*, the game-theoretic modelling approach for bug repair and issue resolution. In [subsection 5.3.1](#), I describe the bug repair and issue resolution corpus with which I built and validated *TaskAssessor*. Modelling task prioritisation requires deciding what is relevant and important to capture and what is not. In [subsection 5.3.2](#), I detail those decisions and [subsection 5.3.3](#) describes how I built *TaskAssessor*. In [subsection 5.3.4](#), I validate *TaskAssessor*, then [subsection 5.3.5](#) discusses the threats to *TaskAssessor*'s validity. In [subsection 5.3.6](#), I describe how to use *TaskAssessor*.

5.3.1 Bug Repair and Issue Resolution Corpus

I collected bug repair and issue resolution data from open source projects in the Apache Software Foundation JIRA Repository (version 6.3.4) [99], using its public REST API [100]. JIRA manages *issues*, which represent software artefacts, such as bugs, feature requests, or tasks. From these data sources, I built a corpus of issue lifecycle data. In the corpus, 53% of the issues are bug reports. I dropped JIRA projects that I could not match with a Git repository [101], because I used Git commits to determine whether or not an issue was resolved. After this, 15 projects remain.

The game-theoretic model of bug prioritisation is suitable for scenarios where 1) teams resolve bugs according to their assigned priority and 2) QA engi-

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

neers are interested in obtaining fixes for their reported bugs and therefore compete for developer time and attention. Such projects are subject to the assessor's dilemma. Despite the cost of developer time, many projects do not use the priority field, so I exclude them. I consider a project is not actively using priorities if the proportion of issues with non-default priorities is less than 30.0%, the rounded median of non-default priority usage over the dataset. This project-using-priority filter left 6 projects and their issues, as shown in [Table 5.4](#). When building and evaluating *TaskAssessor*, I consolidated the issues across these projects to maximise the total data available.

Reporters that participate sporadically in bug repair and issue resolution are not really involved in the task assessment game and will not learn from or respond to changing rewards: they are not acting as QA engineers. I define an engaged reporter as one who files at least 10 different issues on 10 different days. Under this definition, 53.3% of reporters are engaged. I deem the rest to be drive-by, unengaged reporters, and discard their issues. Under this engaged-reporter, *i.e.* QA engineer, filter, I extracted 23,499 issues from these 6 projects, reported between May 2006 and November 2015 and involving 354 reporters. The code to extract the corpus from JIRA is available at the project page [98].

I applied the using-priorities and engaged-reporter filters to focus on people and projects actively using JIRA's shared task prioritisation tooling. They can, of course, also introduce bias. Ablation showed that removing these filters just slows experimentation without changing the results. Game generation for distributed prioritisation under reduced bandwidth ([subsection 6.1.1](#)) takes 96% more time without the filters. When only the engaged-reporter filter is active, game generation takes 44% more time. This figure is in 28% when the only filter active is using-priorities. All three of these scenarios produce the same Nash equilibria.

Out-of-the-box, JIRA supports five priority labels: Blocker, Critical, Major, Minor, Trivial [102]. JIRA defines these labels, but few developers know JIRA's definitions and rely instead on their meanings in ordinary language. These meanings naturally split these five labels in two: {Blocker, Critical, Major} and {Minor, Trivial}. Within each subset, distinctions can be hard to make: is Blocker worse than Critical? Further, different definitions and rankings will emerge in different projects, especially in a corpus that is not

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

Table 5.5: Strategy catalogue S for the task prioritisation game. P_I and P_D represent the conditional probabilities that a QA engineer inflates or deflates an issue.

Strategy	P_I	P_D	Cluster Size	Origin	Description
Honest	0.00	0.00	–	Heuristic	Players adopting this strategy always report their priority assessment
Always Inflate	1.00	0.00	–	Heuristic	Players adopting this strategy report every bug discovered as high priority
Empirically Honest	0.05	0.01	50.39%	Apache data	Empirical strategy with the lowest probability for dishonesty
Empirically Inflater	0.19	0.02	9.06%	Apache data	Empirical strategy with the highest probability for priority inflation
Persistent Deflater	0.08	1.00	7.87%	Apache data	Empirical strategy with the highest probability for priority deflation
Regular Deflater	0.04	0.58	16.54%	Apache data	Empirical strategy with a significant probability for priority deflation
Occasional Deflater	0.06	0.26	16.14%	Apache data	Empirical strategy with a high probability for priority deflation

Google-scale. For these reasons, I reduced these labels to two, mapping Blocker, Critical, Major to High and Minor and Trivial to Low. This boosts signal and allows focussing on harmful mislabelling of priority (whether inflation or deflation).

5.3.2 Game Models with *TaskAssessor*

Developers are expensive; their attention is a scarce resource for which new features and bug fixes compete [103]. Some software processes rely on “Quality Assurance” (QA) engineers to report issues, monitor their progress, and verify their resolution. I modelled such processes as a tragedy of the commons in which QA engineers — the players — compete with each other for the shared commons of developer time.

The focus is priority inflation in shared prioritisation tooling, so, in the game, QA engineers can inflate, deflate, or honestly report an issue’s priority. In line with the competent programmer hypothesis [104], I assume QA engineers are competent and usually know the ground truth priority of an issue. In this work, I am using classic game theory, in which players behave rationally. Thus, QA engineers seek to maximise the number of their issues that developers resolve. Later sections show that this simple model is sufficient to capture actual issue prioritisation behaviour and to provide a solid foundation for a mechanism design solution to the problem of priority inflation. Considering behavioural game theory is future work [53].

In a game model produced by *TaskAssessor*, a QA engineer’s strategy is their propensity to change an issue’s ground truth priority. Let P_g be a random

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

variable that denotes the ground truth priority of an issue, H_q be a random variable for the QA engineer q reporting an issue as high and L_q denote reporting an issue as low priority. A QA engineer's *inflation probability* is $P_I = P(H_q | P_g = L)$; it is a QA engineer's conditional probability to inflate a low priority task. A QA engineer's *deflation probability* is $P_D = P(L_q | P_g = H)$; it is a QA engineer's conditional probability to deflate a high priority task. Hence, the probability for a QA engineer to honestly assess an issue is $1 - P_I$ and $1 - P_D$. A QA engineer is "honest" if s/he never knowingly misprioritises an issue, *i.e.* $P_I = P_D = 0$ for her/him, and as "dishonest" if s/he always inflates or deflates, *i.e.* $P_I = P_D = 1$ for her/him.

To determine the players' empirical strategies, I looked to the data. To learn an engineer's strategy is to learn his P_I and P_D . To do this from data, I needed the ground truth. Using labelled data is a possibility, but I adopted a different strategy: the expedient of 3rd party assessment. In the data set, 3rd party assessment manifests itself as a report whose priority label was changed by a 3rd party; 254 bug reporters filed such a report. In [subsection 5.3.5](#), I discuss the construct threat this proxy for misprioritisation poses.

To extract empirical strategies from the corpus, I clustered observed strategies. I used the k-means algorithm implementation from *scikit-learn* [105] to cluster the players and infer these strategies. In [Table 5.5](#), the rows whose Origin is "Apache data" show the empirical strategies obtained. I am also very interested in assessing how the honest and "Always Inflate" ($P_I = 1, P_D = 0$) strategies perform in the assessor's dilemma because I want to encourage honest prioritisation and discourage inflation. Thus, I added these two heuristic strategies ([subsection 3.3.1](#)) to the empirical strategies I mined. [Table 5.5](#), as a whole, is *TaskAssessor*'s strategy catalogue.

Surprisingly, deflation dominates inflation in three of the five empirical strategies in [Table 5.5](#); indeed, persistent deflators deflate *all* high priority issues that pass through their hands. Just over 40.0% of all reporters are deflators. Clearly, a large portion of reporters are focusing on reducing the number of high priority issues that developers see, rather than merely maximising the number of their issues that developers fix. A payoff function that counts all of a QA engineer's issues, would implicitly penalise deflators and fail to explain their behaviour. Thus, the payoff function is simply the count of issues that a QA engineer files as high priority that the developer team

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

fixes:

$$\text{payoff}(r) = \sum_{f \in F} h(r, f), \quad (5.1)$$

where F is the set of all fixes or features the development team implements and h returns 1 if r files f with high priority.

A classical game consists of players, actions, strategies, and payoff functions (chapter 2). Here, I have described such a game. Unfortunately, classical game theory does not scale.

5.3.3 TaskAssessor under Twins and EGTA

Now, let us discuss how to reduce the game to make its analysis of priority inflation tractable. It requires two major changes. Working top down, I reduced the number of players by clustering them following the Twins Player Reduction, then implemented the payoff function as a simulation model to handle temporality, as required by EGTA (subsection 3.3.1).

PlayGame is the implementation of *TaskAssessor*'s process simulator component (Figure 5.3). Table 5.6 describes its parameters. N_D is the size of the development team. *TaskAssessor* uses the queueing discipline of the development queue to model whether developers consider the reported priority. $M_{dev} = \text{Priority}$ specifies total trust in priority labels; $M_{dev} = \text{FIFO}$ specifies total mistrust. When $M_{dev} = \text{Priority}$, tie breaking is FIFO. A simulation run stops when the development team fixes N_f bugs.

I also assume that bug fixes and new features are independent from each other and can be resolved with a single commit, since evidence suggests this happens in the majority of cases [106]. QA engineers file reports in the tracking system in batches after executing a group of test cases. Hence, I modelled report arrival with two random variables: the time between batches T_{IA} and the number of reports per batch N_b .

Each report has a ground truth JIRA priority P_g and is assigned to QA engineer R . The JIRA project observed that its users tended to confuse severity, the technical difficulty of a task, with its priority, its value to an enduser [82]. Thus, they decided to only keep one field — priority — whose purpose is to

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

Table 5.6: *PlayGame*'s input variables. Random variables are sampled during a run until the number of bugs fixed equals N_f . The nonrandom variables are constant during a simulation run.

Nonrandom Parameters	
R	The set of QA engineers (reporters).
N_D	Size of the developer team.
M_{dev}	Queuing discipline of the development queue (FIFO or Priority).
N_r	Number of simulation runs.
N_f	Number of bugs to fix in a simulation run.
Random Parameters	
T_{IA}	Interarrival time of report batches.
N_b	Count of bug reports contained in a report batch.
P_g	Ground-truth priority of a bug (High or Low).
R	QA engineer (reporter) who filed a report or issue.
$\{S_r\}$	Set of mixed strategies over inflate/deflate/honest each reporter r adopts.
T_{rp}	Resolution time of issues/bug reports with JIRA priority p .
Q_p	Probability the developer team ignores a bug with priority p .

define “the order in which engineers should work on issues” [107]. Since I used JIRA data, I used JIRA priorities. A QA engineer’s assessment strategy S_r governs the priority they assign to a task. Tasks require different time to be fixed, which depends on their JIRA priority; T_{rp} determines, for each priority p , the amount of time to resolve a task. Developers ignore some reports; Q_p captures this probability. To set these parameters to *TaskAssessor*, I built empirical probability distributions based on a linear interpolation between sample quantiles [108]. I define \bar{I} to be a tuple of settings bound to all the parameters in Table 5.6. I treat \bar{I} as an associative array and use $\bar{I}[name]$ to access its components.

Under the Twins Player Reduction, ours is a symmetric, two player game. From $|S|$ and \bar{I} , *TaskAssessor*, as defined in Algorithm 1, forms the payoff matrix in Table 5.7. The coordinates of each cell are a pair of actions, *i.e.* an action profile. Each cell contains $Twins(s_1, s_2, \bar{I})$, the payoff for each player under that action profile.

Algorithm 2, which defines *Twins*, manifests *TaskAssessor*’s use of the Twins

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

Algorithm 1 [*TaskAssessor*] This algorithm uses *Twins*, [Algorithm 2](#), to construct a payoff matrix for the priority inflation game.

Input: S , The strategy catalogue, defined [Table 5.5](#).

\bar{I} , tuple of *PlayGame*'s simulation parameters in [Table 5.6](#).

Output: *payoffMatrix*, *TaskAssessor*'s payoff matrix.

- 1: **for all** $(s_i, s_j) \in S \times S$ **do**
 - 2: *payoffMatrix*[i, j] := *Twins*(s_i, s_j, \bar{I})
 - 3: **return** *payoffMatrix*
-

Table 5.7: Pay-off matrix *TaskAssessor* builds: since it is symmetric, the game has only two players (*Twin*₁ and *Twin*₂) and both player has $|S|$ actions. [Algorithm 2](#), *Twins*, computes the payoff for each pair of actions for each cell.

	<i>Twin</i> ₂ : s_1	\dots	<i>Twin</i> ₂ : s_n
<i>Twin</i> ₁ : s_1	$u_1, u_2 = \text{Twins}(s_1, s_1, \bar{I})$	\dots	$u_1, u_2 = \text{Twins}(s_1, s_n, \bar{I})$
\dots	\dots	\dots	\dots
<i>Twin</i> ₁ : s_n	$u_1, u_2 = \text{Twins}(s_n, s_1, \bar{I})$	\dots	$u_1, u_2 = \text{Twins}(s_n, s_n, \bar{I})$

players reduction, in the context of a symmetric game: it binds one action to a distinguished player and binds the other action to all the other players on line 4, as described in [Figure 5.4](#). Then swaps those bindings on line 6.

PlayGame, defined in [Algorithm 3](#), lies at [Algorithm 2](#)'s core. [Algorithm 2](#) calls *PlayGame* N_r times and returns the average of the results of the payoffs of each run. When $\bigcup A_i = A$, the set of agents, *PlayGame*($\{(A_i, s_i)\}, \bar{I}$) runs the issue resolution and bug repair game among the players in A , using the action s_i for the agents in A_i and the simulation parameters in \bar{I} . It returns the payoff function defined in [Equation 5.1](#).

Adapting *TaskAssessor* to a new task prioritisation process requires only redefining *PlayGame*, a simple but extremely general task prioritisation simulation. In this thesis, I used three different definitions to model the three different processes I discuss in the next chapter. For each of these prioritisation processes, I was able to reuse large parts of the *PlayGame* algorithm.

Finally, *TaskAssessor* passes the resulting payoff matrix to a game solver. Although I am using Gambit [67] in this work, *TaskAssessor* is solver agnostic.

Algorithm 2 [TWINS] This algorithm uses symmetric twins player reduction to estimate the payoff of an action profile in a symmetric twins game (subsection 3.3.1) via the *PlayGame* simulation.

Input: s_1 , *Twin*₁'s action.

s_2 , *Twin*₂'s action.

\bar{I} , tuple of *PlayGame*'s simulation parameters in Table 5.6.

Output: Average payoffs for *Twin*₁ and *Twin*₂.

1: $r := \text{choose } \bar{I}[R]$

2: $U_1, U_2 := \{\}, \{\}$

3: **for** $i = 1$ to $\bar{I}[N_r]$ **do**

4: $\text{payoffs} := \text{PlayGame}(\{(\{r\}, s_1), (\bar{I}[R] \setminus \{r\}, s_2)\}, \bar{I})$

5: $U_1 := U_1 \cup \{\text{payoffs}(r)\}$

6: $\text{payoffs} := \text{PlayGame}(\{(\{r\}, s_2), (\bar{I}[R] \setminus \{r\}, s_1)\}, \bar{I})$

7: $U_2 := U_2 \cup \{\text{payoffs}(r)\}$

8: **return** $\frac{1}{\bar{I}[N_r]} \sum_{u \in U_1} u, \frac{1}{\bar{I}[N_r]} \sum_{u \in U_2} u$

TaskAssessor produces payoff values for each cell of the payoff matrix, that can then be organised in the format required by a specific solver. The solver computes one or more probability distributions over each player actions (the rows or columns in Table 5.7, corresponding to heuristic strategies), or a *mixed strategy* per player in game-theoretic terms. This map of players to strategies is called a *strategy profile*, and each strategy profile produced by the solver corresponds to a Nash equilibrium. According to the TSNE definition (subsection 3.3.1), *TaskAssessor* only considers strategy profiles in which both twin players perform the same mixed strategy. There are various ways to interpret the probability distributions *TaskAssessor* returns. I adopt the learning interpretation: the probability associated to each action is the fraction of the time this action is adopted in the limit, when the game is played multiple times [13].

5.3.4 Validating *TaskAssessor*

There is no point in diagnosing or fixing a process using an inaccurate model; useful models capture a phenomenon under study with sufficient accuracy

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

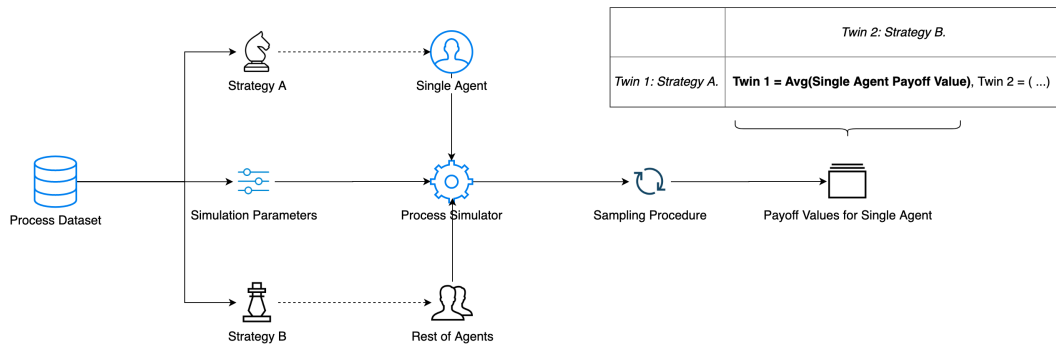


Figure 5.4: Calculating values for a Twins Player Reduction payoff matrix: one agent of the population is assigned strategy A, while the others are assigned strategy B in the simulation. The value to include in the matrix for strategy A against strategy B, is the average payoff of the single agent over multiple simulation iterations.

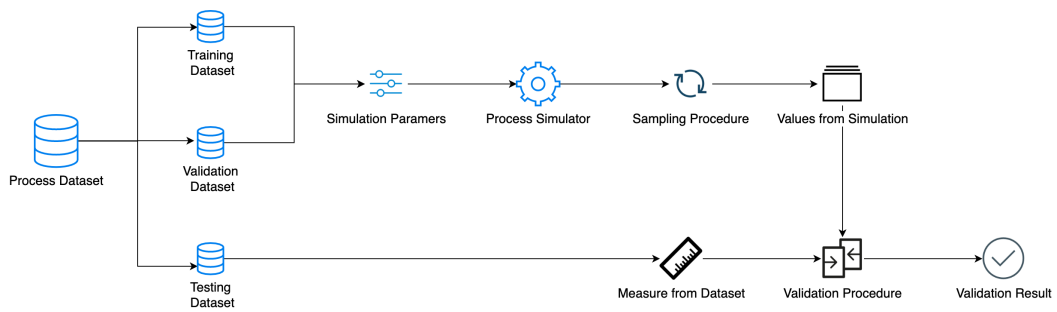


Figure 5.5: Validating *PlayGame's* simulation model. We split the process dataset in three parts: 1) the training dataset is used to obtain simulation parameters, 2) the validation dataset is used for model calibration, 3) and the testing dataset is used to assess the simulation output.

5 TaskAssessor: A Game-Theoretic Model of Priority Inflation

Algorithm 3 [PlayGame] This algorithm is a discrete-event queueing simulator for generating the number of resolved issues per QA engineer.

Input: $\bar{I} = \langle M_{dev}, N_D, N_f, T_{IA}, N_B, P_g, P_R, T_{rp}, \{S_r\}, Q_p \rangle$,
the simulation parameters defined in Table 5.6.

Output: *Fixes*, a map from reporters to their fixes.

```

1: time := 0
2: IssueQueueDev := createQueue( $M_{dev}$ )
3: Devs := initDeveloperTeam( $N_D$ )
4: Fixes := {} # An empty map
5: while notFinished(Fixes,  $N_f$ ) do
6:   if newBatch( $T_{IA}$ , time) then
7:     Batch := generateBatch( $N_B, P_g, P_R, T_{rp}$ )
8:     for all issue  $\in$  Batch do
9:       reportedPriority := assignPriority(issue,  $\{S_r\}$ )
10:      enqueue(issue, reportedPriority, IssueQueueDev)
11:     for all dev  $\in$  Devs do
12:       devIssue := dev.currentIssue
13:       if done(devIssue, time) then
14:         Fixes[devIssue.reporter] += payoff(devIssue)
15:         if notIgnore(IssueQueueDev,  $Q_p$ ) then
16:           dev.currentIssue := dequeue(IssueQueueDev)
17:       time += 1
18: return Fixes

```

to support decisions. Thus, validation is key to assuring stakeholders that the model effectively reflects their software development process and, therefore, is a solid test bed for evaluating the effects of the mechanism design decisions.

To validate *TaskAssessor*, I assessed the ability of its core simulator, *PlayGame*, to produce an output indistinguishable from the process it is modelling. Specifically, *PlayGame* outputs f_l , the percentage of low, and f_h , high priority issues resolved. As seen in Figure 5.5, I started by splitting the dataset into training, validation and testing. I obtained *PlayGame*'s parameters from the training dataset (Table 5.6). I calibrated *PlayGame* on the validation dataset. As usual, I reserved the test data to measure the quality of *PlayGame*'s sim-

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

ulation. I selected 60% of the data for training-validation purposes and the other 40% for testing.

In discrete-event simulation, validation techniques range from hypothesis testing to human assessment. Hypothesis testing can be too strict and rule out simulation models that are sufficiently precise for decision making [65], which in the context of *TaskAssessor* is process diagnosis. The adopted approach evaluates if the simulation output and the real system are close enough to ensure stakeholder trust via confidence intervals. To this end, I built a confidence interval from the simulation output, obtain the best-case and worst-case error of the interval with respect to the measure in the testing dataset, and accept or reject the simulation model by comparing the errors obtained with a threshold ε . The value of ε should be “small enough to allow valid decisions” [65]. I set $\varepsilon = 20\%$ to ensure *PlayGame* is *at most 20%* wrong when predicting the percentage of reported bugs that were fixed. Despite this imprecision, the subsequent results show that *PlayGame* captures the influence of priorities in bug fixing while keeping *PlayGame*’s model simple, easy to understand, and quick to execute.

Results: In the testing dataset, 16.1% of low priority bugs were fixed on average and 33.8% of high priority ones. When $M_{dev} = Priority$, the 95.0% confidence interval for f_l , obtained from 1,000 simulation runs, is [18.1%, 20.4%]. When the tested value falls outside the confidence interval as ours does, the best-case error is $18.1\% - 16.1\% = 2.0\%$ and the worst-case error is $20.4\% - 16.1\% = 4.0\%$. Under the validation procedure [65, Chapter 10, p.326], the validation of *PlayGame* for f_l succeeds because its worst-case error is $4.0\% \leq 20.0\% = \varepsilon$. By similar reasoning, validation succeeds for f_h as well, since the worst-case error is $17.0\% \leq 20\% = \varepsilon$, despite the fact that $f_h = 33.8\% \notin [45.6\%, 50.1\%]$.

5.3.5 Threats to Validity

TaskAssessor faces the standard threat to its external validity: its results generalise only to the extent to which its corpus is representative. It is drawn from JIRA projects, filtered for use of Git, use of prioritisation, and engaged

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

reporters. These filters can introduce bias not already present in the JIRA projects. In [subsection 5.3.1](#), however, the last two filters do not change the computed Nash equilibria. To extract empirical strategies from the corpus, I used the number of third-party corrections to indicate dishonest reporting. Their use to proxy inflation or deflation rates represent a construct validity threat. Of course, a third-party may reprioritise a report for reasons other than a dishonest initial assessment, including honest mistakes and new information. However, under this assumptions, a dishonest QA engineer benefits from an inflated report while a third-party assessor does not. Thus, I think it is reasonable to assume that third-party assessment is more likely to be accurate. Mistakes or logic errors in *TaskAssessor*'s design or implementation are the main internal validity threat to this work. I mitigated this threat in two ways. First, I have detailed *TaskAssessor*'s construction so that readers can themselves assess its logical validity. Second, I validated *TaskAssessor* output using state of practice techniques from the simulation community as described in [subsection 5.3.4](#).

5.3.6 Using *TaskAssessor*

TaskAssessor is a diagnosis tool for task prioritisation processes, tailored to a specific process by redefining its *PlayGame* process simulator. When modelling a process that is immune to priority inflation, *TaskAssessor* outputs a single equilibrium with a probability of 1.0 for the Honest Strategy. Such output means that at equilibrium every task has a reliable priority. In contrast, processes susceptible to priority inflation have a positive probability for inflationary strategies — where $P_I > 0$ like “Always Inflate” or “Empirically Inflater” in [Table 5.5](#) — at one of its equilibria. A non-zero probability for these strategies means inflated reports at equilibrium. A worst-case scenario is a single equilibrium where “Always Inflate” has a probability of 1.0. It is also possible that *TaskAssessor* finds multiple, opposing equilibria for a task prioritisation process: like “Always Inflate” with a probability of 1.0 in one equilibrium and Honest with a probability of 1.0 in another. As stated in [chapter 2](#), a Nash equilibrium is *stable*: once reached, players have no incentive to deviate. When a game-theoretic model has multiple equilibria, the analyst can then adopt a learning model that explains how players

5 *TaskAssessor*: A Game-Theoretic Model of Priority Inflation

interact and reach an specific equilibrium profile [9].

One needs to discuss *TaskAssessor*'s equilibrium results with stakeholders to validate whether they explain a prioritisation process. Process modelling is hard: usually one needs to discuss several models before stakeholder acceptance. If stakeholders reject *TaskAssessor*'s results, revise the simulation model: over-simplification or over-engineering can distort pay-off calculations. Also, ensure that the strategy catalogue does not obviate common or impactful prioritisation behaviour. Once stakeholders agree with *TaskAssessor*'s results, it can also evaluate process interventions to improve a prioritisation process, as shown in [chapter 6](#).

I implemented *TaskAssessor* in Python³. I implemented *TaskAssessor*'s [Algorithm 2](#) component in SimPy, a discrete-event simulation library [109]. For example, you issue

```
taskAssessor.py -r 100 -d 50 -n 50 -o equilibria.csv
```

to generate a payoff matrix for 100 QA engineers and 50 developers, running until $N_t = 50$ bugs are fixed and storing the equilibria in `equilibria.csv`.

³*TaskAssessor* is available on the project page [98].

6 Assessor-Throttling: A Novel Task Prioritisation Process

In this chapter, I elaborate on the software process improvement phase of GTPI's approach to priority inflation, as shown in [Figure 6.1](#). I start by using *TaskAssessor* to model *distributed prioritisation*, which distributes prioritisation to the person filing a task or bug report [110]. After building the model, I computed its Nash equilibrium and found that the equilibrium shows that the "Always Inflate" strategy is optimal ([subsection 6.1.1](#)). In this way, I have used game theory to corroborate the conventional wisdom that distributed prioritisation is prone to priority inflation.

To combat priority inflation, development teams have incorporated *bug triage* into their prioritisation processes [12]. In this process, a team of *gatekeepers*, typically distinct from those who file or report issues, checks (and may reprioritise) each issue. Gatekeepers can be technical or business-focused employees. In [subsection 6.1.2](#), I show that gatekeeper processes reprise distributed prioritisation, which implies that they are also susceptible to priority inflation. I confirmed this using *TaskAssessor* and found that, at equilibrium, task filers and QA engineers still have an incentive to inflate priorities in a gatekeeper process. I used a Jackson network, from queueing theory [111], to show that gatekeeping slows development, even in the presence of duplicate tasks or bug reports. Thus, while gatekeepers can improve the prioritisation of issues in terms of better matching priorities with their business value, I have used game theory to contradict the conventional wisdom; showing that the gatekeeper process does not mitigate priority inflation and slows development.

The game theoretic analysis, in short, shows that the current state of practice does *not* fix priority inflation which may explain the extraordinarily low use

6 Assessor-Throttling: A Novel Task Prioritisation Process

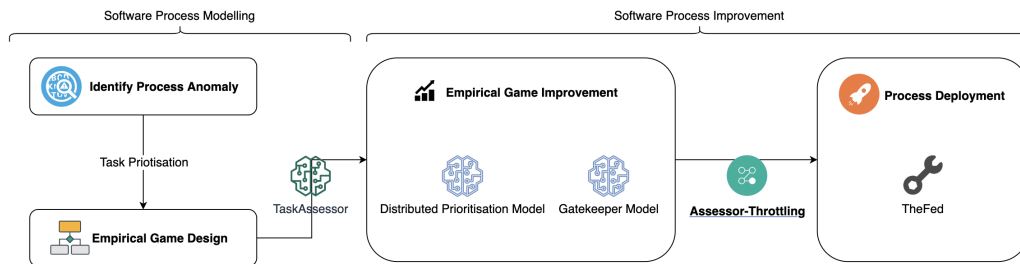


Figure 6.1: GTPI’s software process improvement phase for priority inflation: during software process improvement, we use *TaskAssessor* to model current task prioritisation practices. Finding them susceptible to priority inflation, we propose the assessor-throttling process as a solution. To facilitate assessor-throttling deployment, we developed TheFed. It is a Chrome plugin that connects to JIRA tracking systems.

of priority mechanisms in GitHub (subsection 5.2.1). To fix priority inflation, I turned to mechanism design, the branch of game theory concerned with designing games whose equilibrium strategies constrain players to behave in desirable ways. Using *TaskAssessor* to evaluate process interventions (or mechanisms), I devised a novel, lightweight prioritisation mechanism, *assessor-throttling*, to tackle priority inflation (subsection 6.1.3). When they have completed a task or fixed a bug, developers have also assessed its JIRA priority along the way: they know its technical severity and often have acquired the expertise needed to evaluate its business value [112]. When they finish a task, assessor-throttling merely requires developers to record their assessment of the task’s priority label. If the developer’s assessment differs from the task’s priority label, the offending reporter’s reputation drops, which restricts the reporter’s ability to submit tasks or bugs. Via simulation, I show that assessor-throttling matches an ideal gatekeeper in the completion of high priority tasks (section 6.1.3). To help developers transition to assessor-throttling and thereby combat priority inflation, I realised it in TheFed, a browser plugin for Chrome (section 6.2) that individual developers can easily download and install from the project page [98].

6 Assessor-Throttling: A Novel Task Prioritisation Process

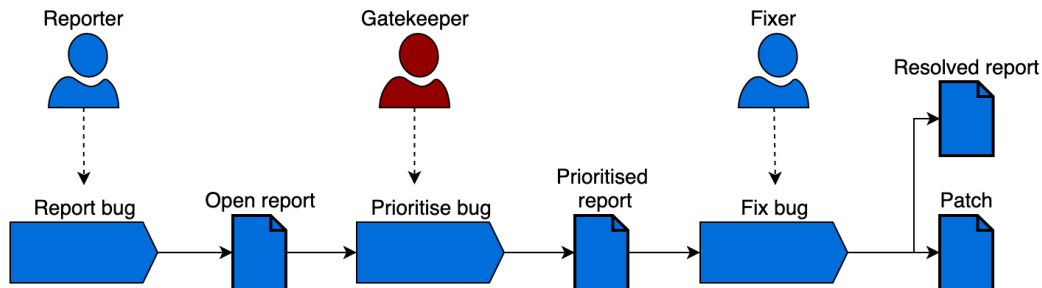


Figure 6.2: Bug prioritisation processes: the blue components are shared between the two processes (including distributed prioritisation), the red ones are exclusive to the gatekeeper process. The solid lines represent the input/output of an activity, and the dashed lines link an activity with its performing role.

6.1 Empirical Game Improvement

Figure 6.2 describes task prioritisation using the Software Process Engineering Metamodel (SPEM) notation to represent its roles, tasks and work products [76]. Three tasks, coloured blue, are common to all the task prioritisation processes under analysis: reporting, prioritising, and resolving tasks.

Two different task prioritisation processes superimpose in Figure 6.2. In *distributed prioritisation*, the reporter role both files and prioritises tasks. In [subsection 6.1.1](#), I show that distributed prioritisation is susceptible to priority inflation. To correct distributed prioritisation’s tendency to priority inflation, development teams have taken prioritisation away from reporters and given it to a *gatekeeper* (light red in Figure 6.2). In [subsection 6.1.2](#), I present two findings. First, I make an argument from queueing theory that gatekeeping *only slows* task resolution. Second, I show that even a perfect gatekeeper that correctly prioritises all reports does not remove the incentive for inflating priorities. In short, I first confirm the conventional wisdom about distributed bug prioritisation, then I *contradict* the conventional wisdom that gatekeepers improve bug repair.

I am especially interested in the impact of developer bandwidth on priority inflation: intuitively, scarce development time magnifies the reward for inflating priorities. As shown in [Figure 6.3](#), I computed Nash equilibria for

6 Assessor-Throttling: A Novel Task Prioritisation Process

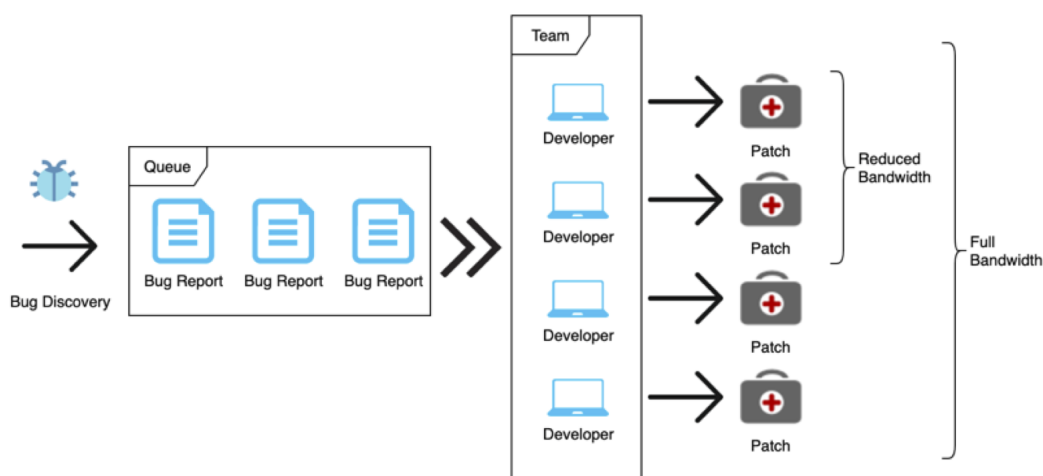


Figure 6.3: *PlayGame*'s input-output transformation: QA engineers place reports in the development queue. Developers in the team then build patches to address each report. In a full-bandwidth scenario D , all developers are available for bug fixing duties. In a reduced bandwidth scenario $\frac{D}{2}$, only half of them are active

the following two scenarios: in *full bandwidth* D , all the developers actively remove reports from the development queue; in *reduced bandwidth* $\frac{D}{2}$, only half of them are active. In all scenarios, I assume the developers consider a task's priority label when deciding whether to work on the task. Without this assumption, I cannot distinguish the inflationary propensity of the two processes analysed below.

6.1.1 Distributed Bug Prioritisation

A bug prioritisation process can assign the reporting and prioritisation of bugs to the QA engineer role. Such processes are common in both FOSS and industry projects. A company, for instance, adopts such a process when they decide to outsource the development services to an external IT provider. In this scenario, the outsourcing company fixes the bug that the contracted company reports and prioritises [110]. A QA role appears in some agile settings, where the team member that discovers a bug is in charge of logging a bug and assigning its priority [113].

6 Assessor-Throttling: A Novel Task Prioritisation Process

I call bug prioritisation process involving a QA engineer *distributed bug prioritization*, because it distributes bug prioritisation to QA engineers, or bug reporters. Common knowledge suggests that this process encourages priority inflation as is already reported by practitioners [79] [12]. To verify this, we ask

RQ: Is distributed bug prioritisation susceptible to priority inflation?

I built game-theoretic models by applying *TaskAssessor* (chapter 5) to each bandwidth scenario D and $\frac{D}{2}$, finding a single equilibrium where the probability of “Always Inflate” is 1.0. This differs from the desired outcome of an equilibrium with a probability of 1.0 for Honest. This finding corroborates conventional wisdom and validates the accuracy of the approach. In summary:

Finding: Distributed bug prioritisation *is* susceptible to priority inflation.

6.1.2 Do Gatekeepers Prevent Priority Inflation?

A standard approach to address priority inflation is to appoint a *bug triage* team, that inspects and corrects bug reports, including the reported priority [12]. I call such a process a *gatekeeper process*, because these teams act as gatekeepers who control access to developer time. In this section, I show that a gatekeeper process does *not* prevent priority inflation and, in fact, can only slow development.

Not all software organisations use gatekeepers, but several have reported on their use. For example, Microsoft reports that their gatekeeper is a cross-discipline team [81], while Google reports that some teams delegate this task to a tester-developer pair [80]. In open-source projects, developers establish a gatekeeping rota, or they rely on volunteers from the community to perform gatekeeping duties [114; 115]. Teams adopting an agile process can also include bug triage tasks, where the product owner, a developer, or a customer representative performs the gatekeeper role [116; 117]. The survey also shows that a gatekeeper is a common way to tackle priority inflation: 34% of responses (subsection 5.2.2). Although developers can be part of the gatekeeping process, in my experience, gatekeepers are usually not software

6 Assessor-Throttling: A Novel Task Prioritisation Process

developers. Given the high salaries of developers, I believe teams prefer to invest their time in building features rather than gatekeeping.

In a gatekeeper process, QA engineers place their issues into the gatekeeper's queue, not the development team's queue. This process reprises distributed prioritisation in one of two ways. In the first one, gatekeepers face priority distortion instead of developers; while in the second one gatekeepers are the ones distorting priorities for the development team.

Two justifications are usually advanced for adopting a gatekeeper process. One is to involve business expertise in task prioritisation to ensure that priorities correctly reflect business value. The other is economic: gatekeepers are usually cheaper than developers [103] and, since they focus on issue/bug report quality, become more efficient at that task than developers. These two justifications are often in tension because of the cost of business expertise.

When an organisation opts for business expertise, it assigns product managers, business analysts, or even clients to the gatekeeper role. These stakeholders can potentially be scarce, expensive, and busy even before taking on a gatekeeper role. As gatekeepers, they tend to make development slower and more costly. The economic justification of gatekeeping breaks down when gatekeeping is a role that developers or product managers play. Recall that JIRA priorities, which I use in this work, combine technical severity, *i.e.* difficulty of resolution, and value, including risk, to end-users. Regardless of who fills the gatekeeper role, in general gatekeepers do not learn their technical severity, since they do not actually resolve issues. When gatekeepers are drawn from technical employees, like testers, they are no better or worse at learning to assess business value than developers.

The gatekeeping process includes a bug triage queue that protects developers from poor issue descriptions, but can be a bottleneck, especially if the gatekeeping team lacks resources. To represent this process, I added two elements to the process simulation (*PlayGame* in subsection 5.3.3): a bug triage queue, with a queuing discipline of $M_{gk} = FIFO$, and a gatekeeping team G , whose members take R time to triage a task with an error rate of A_{gk} .

RQ: What is the impact of a gatekeeper on issue resolution?

Researchers have successfully used queuing systems with Poisson arrivals and exponential service times to model real-world bug repair processes [118];

119]; I followed their lead. I assume QA engineers file issues under a Poisson distribution, a gatekeeper takes exponentially distributed time to review them, and the development team take exponentially distributed time to fix them. When $\mathbf{E}(W_G)$ is the mean sojourn time of an issue in the gatekeeper (or triage) queue, $\mathbf{E}(W_D)$ is the mean sojourn time of an issue in the development team queue, and $\mathbf{E}(W_{GP})$ is the mean, end-to-end, sojourn time of an issue/bug report in a gatekeeping process, I have

Finding: Over a sequence of issues, a gatekeeper can only slow issue resolution: $\mathbf{E}(W_{GP}) = \mathbf{E}(W_G) + \mathbf{E}(W_D)$.

Given the assumptions of Poisson arrivals and exponential service times, the gatekeeper process is a Jackson network using tandem queues (a triage queue for gatekeepers and a developer queue) [111]. In Jackson networks, the mean sojourn time of the whole system at steady state — that is, the time an issue/bug spends in the system from reporting to fixing — is the sum of the mean sojourn times of each individual queue of the system [120].

This finding matters even when an organisation adopts gatekeeping to detect and remove duplicate issues before they reach developers. In general, a gatekeeper must observe and consider several issues before 1) determining that they are duplicates and 2) learning to quickly identify and drop them. Let k denote the expected number of issues one needs to view before realising that they are duplicate and that each gatekeeper filters reports for a team of n developers. A developer is at least as good as a gatekeeper at detecting duplicate bug reports, but each developer learns independently, so, collectively, they will need to see nk duplicates before they all can quickly drop them. Both gatekeepers and developers can learn in parallel; so, given enough duplicates, the expected time needed to learn recognise duplicates for both a gatekeeper and a developer team is the same. For m duplicate reports, there are three cases. If $m < k$, gatekeeping does not remove duplicates before they reach developers. If $m > kn$, then all the developers will have learned to identify and remove them. In this case, the gatekeeper provides no advantage in the limit. It is only when $k \leq m \leq kn$ that gatekeepers remove duplicates at less cost than simply asking developers to do it.

Under this finding, whenever an issue's mean sojourn on the triage queue exceeds zero, a gatekeeper reporting process slows the delivery of bug fixes.

6 Assessor-Throttling: A Novel Task Prioritisation Process

Despite the overhead the gatekeeper process imposes, if it reduces or eliminates priority inflation, it might be worth adopting, so we ask

RQ: Is gatekeeper prioritisation susceptible to priority inflation?

For the equilibrium analysis, I considered a team of $G = 2$ gatekeepers that spend $R = 20$ minutes assessing the priority of an issue. I set $G = 2$ because Ayewah reported that Google used this number [121]; I set $R = 20$ because Page reported 20 minutes as the approximate triage effort per bug at Microsoft [81]. I also explored three performance profiles: a fallible gatekeeper with an error rate of $A_{gk} = 50\%$, an expert gatekeeper with an error rate of $A_{gk} = 10\%$ and an ideal gatekeeper with an error rate of $A_{gk} = 0\%$.

In both bandwidth scenarios D and $\frac{D}{2}$, the fallible gatekeeper ($A_{gk} = 50\%$) has a single equilibrium with probability 1.0 for inflating priorities, the “Always Inflate” strategy. This is the expected result: a fallible gatekeeper leaves the door open to QA engineers profiting from inflating their reports. The expert gatekeeper ($A_{gk} = 10\%$) does no better: in both D and $\frac{D}{2}$, the expert gatekeeper also has a single equilibrium profile with probability 1.0 for the “Always Inflate” strategy: even 90% prioritisation accuracy is insufficient. What about perfect accuracy? Under the ideal gatekeeper ($A_{gk} = 0\%$), each scenario produces multiple equilibria: 3 under $\frac{D}{2}$ bandwidth and 4 under D , not the desired result: a single equilibrium with probability 1.0 for honest prioritisation¹. These equilibria do not rule out priority inflation. The reason is, under the ideal gatekeeper, each QA engineer’s pay-off is the same regardless of their prioritisation decisions. In summary,

Finding: Gatekeeper prioritisation <i>is</i> susceptible to priority inflation.
--

6.1.3 The Assessor-Throttling Process

The gatekeeper process slows issue resolution and bug repair and does not prevent priority inflation. In this section, I present assessor-throttling: a

¹These equilibria are available at the project page [98].

6 Assessor-Throttling: A Novel Task Prioritisation Process

novel task prioritisation process that, unlike gatekeeping, is lightweight and removes priority inflation.

I first apply mechanism design (section 2.6) to priority inflation to define and model assessor throttling, the proposed task prioritisation process. Later, I validate that assessor-throttling *prevents* priority inflation. Finally, I compare distributed prioritisation, gatekeeping, and assessor-throttling with respect to the expected proportion of high priority issues resolved.

Modelling Assessor-Throttling: Assessor-throttling (AT) rests on the insight that developers naturally assess tasks while resolving them. Currently, this developer assessment is wasted. We can use it to assess the priority assigned to a task by the bug reporter or QA engineer who filed it. To construct an incentive compatible mechanism from this developer assessment of task priority, AT rates each QA engineer's assessment accuracy; this rating becomes a QA engineer's reputation. A QA engineer's reputation then controls their access to the developer team in two ways: 1) AT uses reputation to control how many issues a QA engineer can add to the development queue when it is under contention and 2) AT displays the QA engineer's reputation when developers are considering whether to take up a task from the work queue. Honest QA engineers tend to get more developer time; dishonest (or incompetent) ones will get less, and eventually, no access.

The assumption that underlies this reputation mechanism is that developers can accurately assess a task's or bug's priority. JIRA priorities combine technical severity and business value. Developers necessarily overcome a task's technical severity when they resolve it. While developers vary in expertise and some might find a bug more difficult, and thus more severe, than other developers, they are better placed to assess severity than testers or triage teams. Assessing business value is harder. The ground-truth assessment of business value relies on the role that generates software requirements, like the customer, a business analyst, or the business owner. Nonetheless, software engineers can estimate the business value when they work in the same domain long enough, they can eventually qualify as domain experts [112].

Like distributed prioritisation, assessor-throttling decentralises prioritisation: AT does not introduce a second queue, in contrast to gatekeeping, and re-

6 Assessor-Throttling: A Novel Task Prioritisation Process

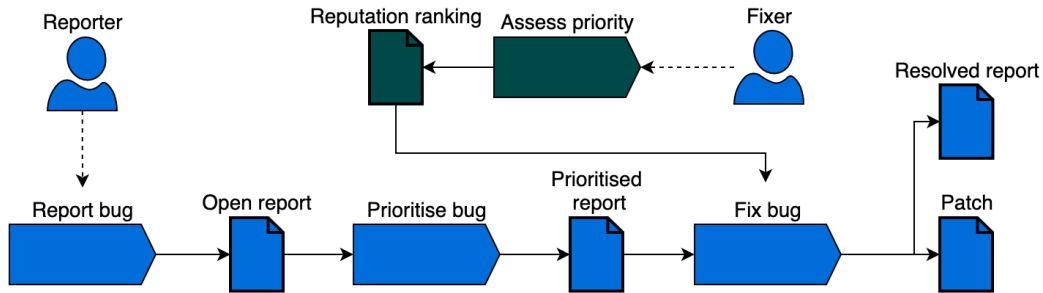


Figure 6.4: The assessor-throttling prioritisation process: after fixing a bug, the fixer assesses the priority made originally by the reporter. This assessment impacts the reporter's position in the reputation's ranking, that determines which bugs are fixed first.

quires *both* QA engineers and developers to assess task priority. AT models developer assessment mistakes with A_{dev} . The developer's assessment is the green task in Figure 6.4. QA engineers and task assessors have a reputation R^r . When a developer finishes a task, they consider its assigned priority. If the developer agrees with that priority, they reward the assessor by increasing R^r by T_+ ; if they disagree, they penalise the assessor by decreasing R^r by T_- . To allow an assessor to recover from a mistake, AT has $T_+ > 0$; to discourage priority distortion, $T_+ \ll T_-$. Assuming that developer assessment approximates the ground-truth, this behaviour penalises both dishonest and incompetent QA engineers. AT does not rely on QA engineer's intentions to improve bug prioritisation: both dishonest and incompetent prioritisation are indistinguishable and discouraged under the model, under the weak assumption that QA engineers learn over time.

Under AT, developers take action: they reward or penalise QA engineers. This action implies that developers should be strategic agents in the model, or players, rather than a commons, as TaskAssessor currently does. Indeed, developers might abuse their power to rank QA engineers who prioritise tasks they like. As explained in subsection 5.3.3, however, adding developers as players to the game-theoretic model would produce an immense game tree and break the symmetry assumption on which a number of the game reduction techniques depend (subsection 3.3.1). Treating developers as players is future work.

6 Assessor-Throttling: A Novel Task Prioritisation Process

Table 6.1: Pay-off matrix for the assessor’s dilemma using assessor throttling.

	<i>Bob: accurate</i>	<i>Bob: inflate</i>
<i>Alice: accurate</i>	$A = 125, B = 125$	$A = 125, B = 100$
<i>Alice: inflate</i>	$A = 100, B = 125$	$A = 112.5, B = 112.5$

The simulation demonstrates the effectiveness of the mechanism. This is unsurprising, because it aligns with previous work: restricted to bug repair, Guo *et al.* found that bug reporter reputation is a key factor in the likelihood of a bug being fixed [122]. They define reputation as the proportion of reported bugs that are fixed, following Hooimeijer and Weimer [123]. Neither of these prior works propose a new process that exploits reputation. AT is a novel task prioritisation process that exploits a reporter’s reputation and permits developers to change a QA engineer’s reputation. Further, AT defines reputation as an agreement with a task resolver, not the proportion of bugs reported and fixed. Integrated into AT’s reward system, this operationally changes the definition of reputation into a measure of honest reporting.

Assessor-Throttling Prevents Priority Inflation: Let us see how throttling works in the situation described in section 5.1. If Alice reports the accurate priorities of her bugs and Bob inflates his reports, we have two possible outcomes: Alice gets only her high priority report fixed — Bob’s inflation was detected after all his patches were delivered — or she gets two fixes, because Bob’s inflated trivial bug was fixed first so he was marked as an offender. Hence, Alice’s expected pay-off is $0.5 \times 100 + 0.5 \times (100 + 50) = 125$ and offender Bob obtains $0.5 \times (100 + 50) + 0.5 \times 50 = 100$.

If we update the original pay-off matrix with the throttling pay-off values, we obtain the matrix in Table 6.1. The Nash Equilibrium of the new matrix has both Alice and Bob reporting the accurate priorities, which is in *Foo Inc.*’s best interest. Assessor-throttling appears to prevent priority inflation. To confirm, we ask

RQ: *Is assessor-throttling susceptible to priority inflation?*

Assessor-throttling depends on two parameters: the error rate of the development team (A_{dev}) for detecting dishonesty and the penalty they apply

6 Assessor-Throttling: A Novel Task Prioritisation Process

when they detect such behaviour (T_-). In the experiments, the development team error rate was fixed to $A_{dev} = 5\%$: it is a more palatable value than $A_{dev} = 50\%$, which makes the priority field irrelevant, or $A_{dev} = 100\%$, which would necessarily lead to an equilibrium with dishonest prioritisation. Regarding inflation penalty T_- , it needs to have a value big enough so that the expected benefits from inflation are less than the expected penalty due to reputation loss. For the sake of deployability, we want T_- to be small since big penalties can face resistance. I started with $T_- = 3\%$ and progressively augment it until obtaining an equilibrium with honest prioritisation. Each value of T_- was analysed under the reduced bandwidth $\frac{D}{2}$ (Figure 6.5) and full bandwidth D (Figure 6.6) scenarios used in subsection 6.1.1 and subsection 6.1.2.

When applying *TaskAssessor* to assessor-throttling with $T_- = 3\%$, the D bandwidth scenario produced a single equilibrium where the probability of the “Always Inflate” strategy was 1.0. As explained in subsection 5.3.6, that result identifies a process susceptible to priority inflation. This suggests that the value of the penalty with respect to the inflation reward is too low to discourage this behaviour.

If the dishonesty penalty is set to $T_- = 10\%$, *TaskAssessor* produced one TSNE equilibrium for each bandwidth scenario. In the $\frac{D}{2}$ bandwidth scenario, *TaskAssessor* outputs the desired equilibrium where the honest strategy has a probability of 1.0, while in the D bandwidth scenario this probability is 0.97. Although close, for the D bandwidth scenario I still did not obtain priority inflation immunity. By increasing the penalty value, equilibria with high probability for honest behaviour start to appear.

When applying *TaskAssessor* to an assessor-throttling process with $T_- = 20\%$, I obtained the same result in both scenarios: a single equilibrium where the honest strategy has a probability of 1.0. I obtained the same result with a dishonesty penalty of $T_- = 22\%$. Having the expected equilibrium with such a low penalty value is an indicator of the actionability of the assessor-throttling process. This also reflects that penalty calibration is a key factor for its effectiveness. In summary:

Finding: Assessor-throttling prevents priority inflation.
--

6 Assessor-Throttling: A Novel Task Prioritisation Process

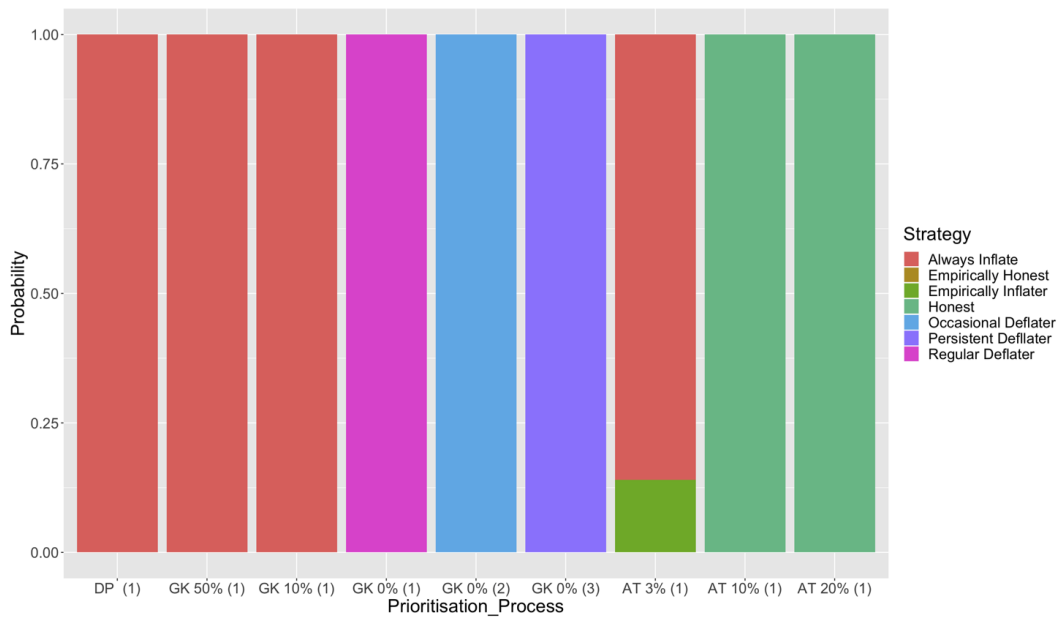


Figure 6.5: Equilibrium profiles for prioritisation processes at the reduced bandwidth scenario $\frac{D}{2}$. From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).

After calibrating the penalty-value parameter, assessor-throttling produces an equilibrium where the honest strategy has a probability of 1.0, which implies honest bug prioritisation.

Racing to the Fixes: I now compare each of the task prioritisation processes presented under the mean percentage of high priority bugs that were fixed $E(g_h)$. I show that assessor-throttling always performs at least as well as the other two prioritisation processes in the bandwidth scenarios used in subsection 6.1.1 and subsection 6.1.2. In fact, assessor-throttling is statistically indistinguishable from an ideal gatekeeper. The comparison is made in terms of impact on software quality, so the last research question is

RQ: What is the impact of the adopted task prioritisation process on the quality of the software product?

6 Assessor-Throttling: A Novel Task Prioritisation Process

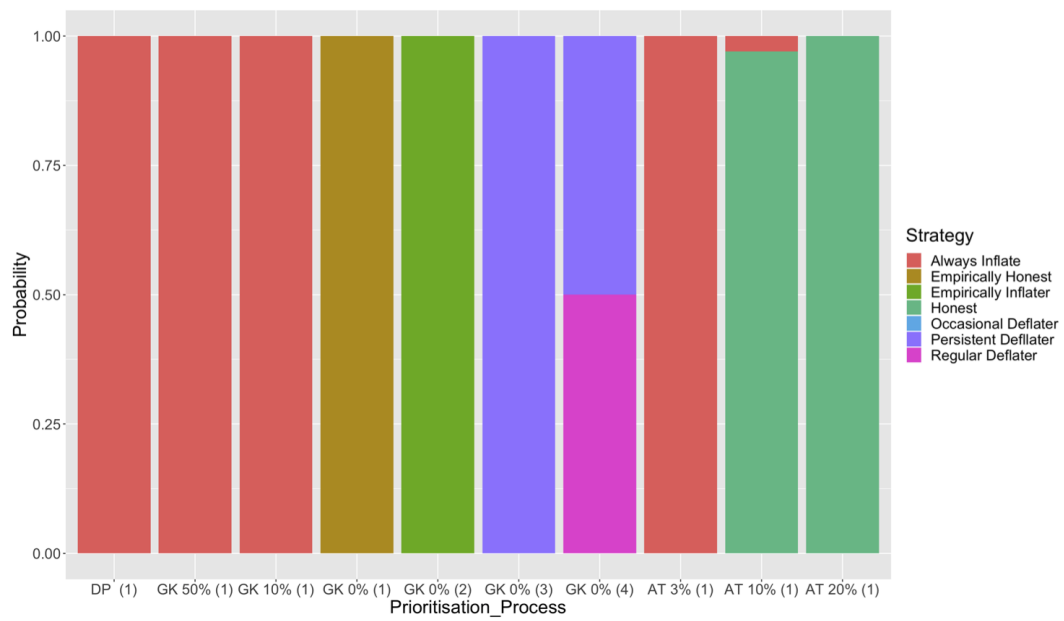


Figure 6.6: Equilibrium profiles for prioritisation processes at the full bandwidth scenario D . From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).

6 Assessor-Throttling: A Novel Task Prioritisation Process

Several techniques are available in the simulation and operations research communities for finding the best simulated system design according to an expected performance measure [124]. The two-stage Bonferroni procedure proposed by Nelson and Matejcik [125] is one of the many techniques that rely on an *indifference zone*. Indifference zone techniques are known to be statistically conservative: they guarantee a lower bound to the probability of selecting the best system $1 - \alpha$ given that this system is at least ϵ better than the rest of the systems [126]. In this section, the system designs under comparison are the prioritisation processes at their equilibria (Figure 6.7) and the performance measure is $\mathbf{E}(g_h)$. The two-stage Bonferroni procedure takes three parameters: ϵ , $1 - \alpha$, and the first-stage sample size R_0 . I executed R_0 iterations on each task prioritisation process at equilibrium. The output is then used as an input to obtain the second-stage sample size R . When $R > R_0$, two-stage Bonferroni requires executing additional $R - R_0$ iterations. From the simulation iterations, I obtained $\mathbf{E}(g_h)$ per task prioritisation process at equilibrium. From the simulation results, I built confidence intervals. From them, we can conclude that a task prioritisation process is either inferior to the best performer or statistically indistinguishable from it.

Due to computational costs, I simulated each scenario for $R_0 = 120$ iterations. I would also like a 95% confidence of obtaining the best process, given that the best differs from the second best by at least $g_h = 5\%$. That translates to $1 - \alpha = 0.95$ and $\epsilon = 0.05$. Figure 6.7 presents the $\mathbf{E}(g_h)$ for each task prioritisation process in a $\frac{D}{2}$ bandwidth scenario. The statistically indistinguishable best repair processes are gatekeeper with $A_{gk} = 10\%$ error rate, gatekeeper with a $A_{gk} = 0\%$ error rate, assessor-throttling with $T_- = 20\%$ dishonesty penalty and assessor-throttling with $T_- = 10\%$ dishonesty penalty. The best performer has a performance value of $\mathbf{E}(g_h) = 0.97$, which is significantly better than the ones marked as inferior.

Under D , when all developers are available, the best performer is one of the equilibria of the gatekeeper with $A_{gk} = 0\%$ error rate with a performance value of $\mathbf{E}(g_h) = 0.96$ (see Figure 6.8). Distributed prioritisation with $\mathbf{E}(g_h) = 0.57$ and assessor-throttling with a $T_- = 3\%$ dishonesty penalty and $\mathbf{E}(g_h) = 0.16$ are inferior, while the rest are statistically indistinguishable from the best performing equilibrium of a gatekeeper with $A_{gk} = 0\%$. Given a sufficiently strong penalty value, we find

6 Assessor-Throttling: A Novel Task Prioritisation Process

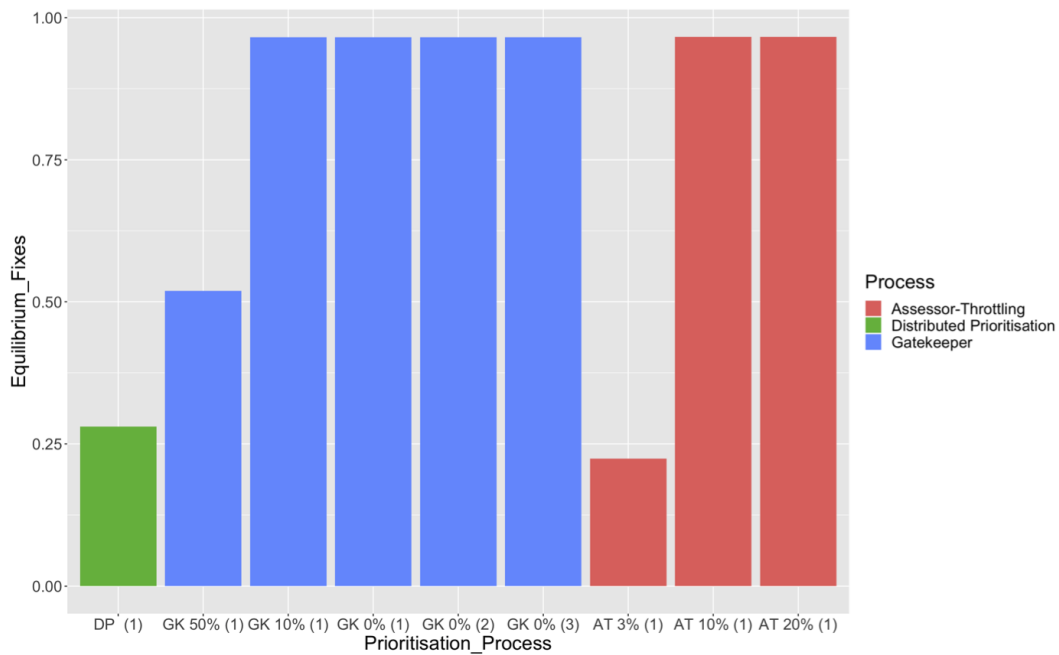


Figure 6.7: Performance comparison of task prioritisation processes in the reduced bandwidth scenario $\frac{D}{2}$. From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).

Finding: Developers fix as many high priority bugs under assessor-throttling as under an ideal, error-free gatekeeper.

6.2 Process Deployment

Assessor-throttling relies on a reputation system for task assessors, so I built a software tool to track assessor reputation and support the teams who want to adopt assessor-throttling. The tool, *TheFed*², is a Chrome extension for JIRA. *TheFed* is open source [127] and has these key features: *TheFed*

1. tracks each QA engineer's reputation.

²*TheFed* stands for Federal Reserve: the central banking system of USA. As such, it defines target inflation rates.

6 Assessor-Throttling: A Novel Task Prioritisation Process

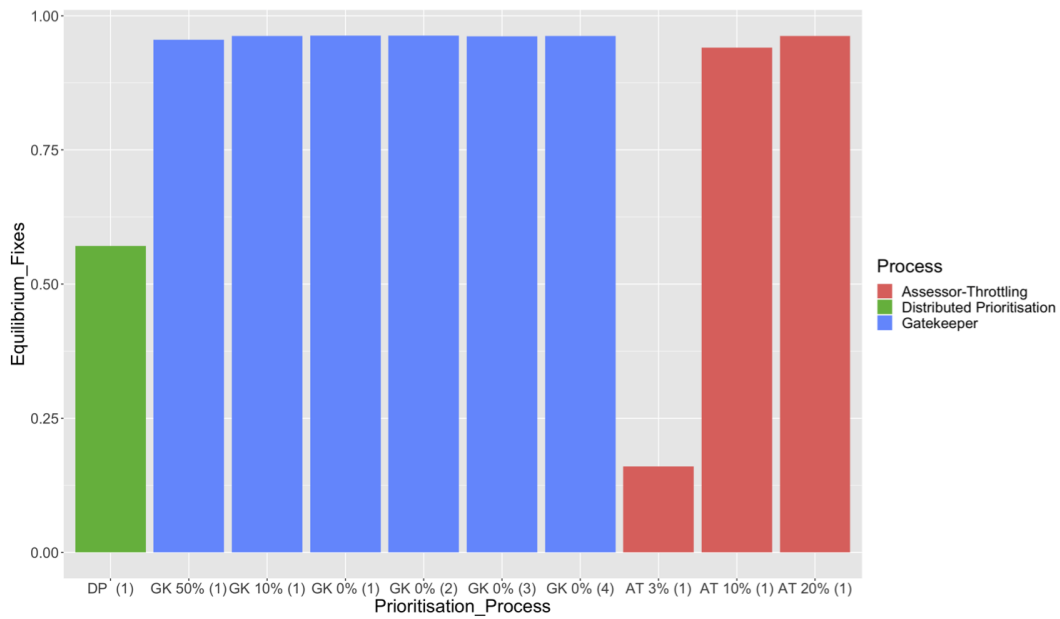
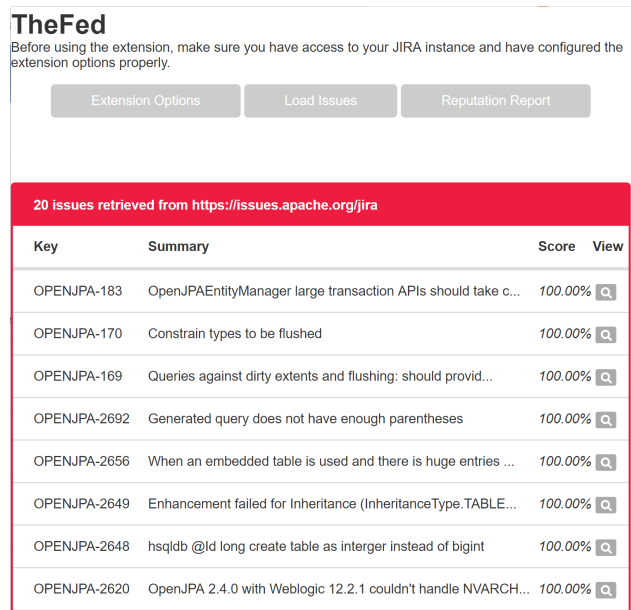


Figure 6.8: Performance comparison of task prioritisation processes in the full bandwidth scenario D . From left to right, we see distributed prioritisation (DP), gatekeeper (GK, with corresponding error rate A_{gk}) and assessor-throttling (AT, with corresponding dishonesty penalty T_-).

6 Assessor-Throttling: A Novel Task Prioritisation Process



The screenshot shows a web interface titled "TheFed". At the top, there is a warning message: "Before using the extension, make sure you have access to your JIRA instance and have configured the extension options properly." Below this are three buttons: "Extension Options", "Load Issues", and "Reputation Report". A red banner indicates "20 issues retrieved from https://issues.apache.org/jira". Below the banner is a table with columns: Key, Summary, Score, and View. The table lists 10 issues, all with a score of 100.00%.

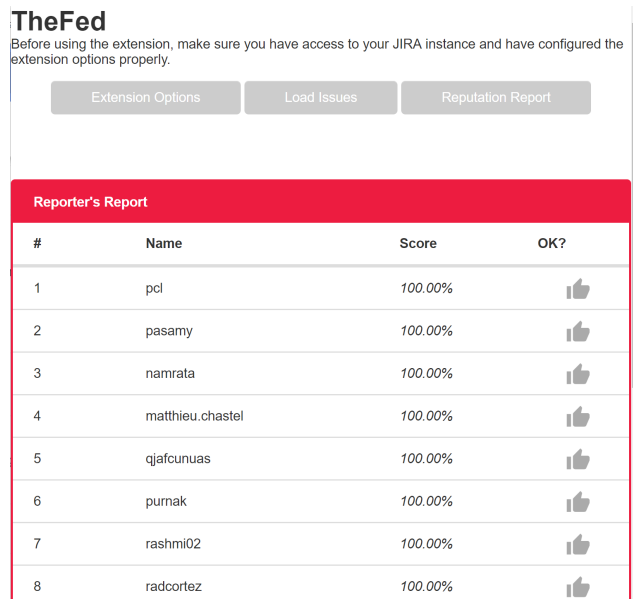
Key	Summary	Score	View
OPENJPA-183	OpenJPAEntityManager large transaction APIs should take c...	100.00%	Q
OPENJPA-170	Constrain types to be flushed	100.00%	Q
OPENJPA-169	Queries against dirty extents and flushing: should provid...	100.00%	Q
OPENJPA-2692	Generated query does not have enough parentheses	100.00%	Q
OPENJPA-2656	When an embedded table is used and there is huge entries ...	100.00%	Q
OPENJPA-2649	Enhancement failed for Inheritance (InheritanceType.TABLE...	100.00%	Q
OPENJPA-2648	hsqldb @ld long create table as interger instead of bigint	100.00%	Q
OPENJPA-2620	OpenJPA 2.4.0 with Weblogic 12.2.1 couldn't handle NVARCH...	100.00%	Q

Figure 6.9: *TheFed*'s prioritised inbox: JIRA issues are sorted by QA engineer's reputation.

2. allows developers to penalise inflation.
3. rewards honest QA engineers.

In assessor-throttling, developers and task assessors make decisions based on the reputation score R^r , so *TheFed* must calculate and store these values. To this end, *TheFed* uses JIRA's REST API [128] to obtain how many times a QA engineer incorrectly prioritised an issue, relative to the priority assessment of the developer who resolved the issue. R^r is a function of the number of these "infractions", so developers need to update it when a priority assignment is inaccurate. To support this feature, *TheFed* relies on JIRA's existing functionality for updating priority; issue resolvers use this functionality to state that they disagree with a QA engineer's priority assessment. Assessor-throttling aims to give high-reputation task assessors a higher probability that their bugs will be fixed than low-reputation assessors. To this end, *TheFed* provides a prioritised inbox, as shown in Figure 6.9. This inbox shows open and unassigned bugs sorted by R^r : honest QA engineers will have their bugs listed on top. As shown in Figure 6.10, *TheFed* can also displays QA engineers ranked by R^r .

6 Assessor-Throttling: A Novel Task Prioritisation Process



The screenshot shows the 'TheFed' interface. At the top, there is a header 'TheFed' and a warning message: 'Before using the extension, make sure you have access to your JIRA instance and have configured the extension options properly.' Below this are three buttons: 'Extension Options', 'Load Issues', and 'Reputation Report'. The main content is a table titled 'Reporter's Report' with the following data:

#	Name	Score	OK?
1	pcl	100.00%	👍
2	pasamy	100.00%	👍
3	namrata	100.00%	👍
4	matthieu.chastel	100.00%	👍
5	qjafcunuas	100.00%	👍
6	purnak	100.00%	👍
7	rashmi02	100.00%	👍
8	radcortez	100.00%	👍

Figure 6.10: *TheFed*'s QA engineer's ranking by reputation score R' .

I hope that development teams adopt assessor-throttling. This cannot happen if its adoption disrupts existing practices and tools. To maximise its deployability, *TheFed*

1. is compatible with existing toolkits;
2. tackles priority inflation immediately after installation; and
3. deploys to clients, not servers.

The penalty value per infraction is a key parameter. As is shown in [section 6.1.3](#), a penalty value that is too low might not produce the desired output of a single equilibrium with the honest strategy probability of 1.0. *TheFed* allows this parameter to be configurable. To this end, users of *TheFed* can customise its behaviour via the options page of Chrome extensions. Some of the parameters available are `penalty_per_infraction`, `issues_in_inbox` and `JIRA_project`.

Tools drive software development: text editors, IDE's and bug tracking systems are among them. *TheFed* integrates easily with existing tools. The current version of *TheFed* is a Chrome extension that relies on JIRA's REST API to

6 Assessor-Throttling: A Novel Task Prioritisation Process

access issue data. Chrome extensions are also compatible with the Opera browser; *TheFed* can be extended to other bug tracking systems that expose a RESTful API, like Bugzilla [129]. From the moment it is installed, *TheFed* provides value: it immediately tackles priority inflation, calculating R' using priority updates accessed through JIRA's API. From this useful starting point, *TheFed* only improves through network effects. Developers directly and immediately benefit from *TheFed* since it helps them better prioritise their work. Crucially, developers can install *TheFed* locally. This allows individual team developers to adopt it without requiring support from a JIRA system administrator, unlike a JIRA plugin.

7 Conclusion and Future Work

In this chapter, I summarise the contributions of this thesis and elaborate on research directions for future work.

7.1 Summary of Contributions

Researchers have already approached software development problems using game-theoretic models. In [chapter 3](#), I surveyed the work published on each software engineering discipline. While I found a wide range of scenarios and techniques, most of the proposed models suffer from scalability issues. Classic game representations ([chapter 2](#)) only support a limited number of players, actions and interactions while keeping a manageable model size for analysis. This limits the application of game theory to modern software development, where large distributed teams release software continuously at a high pace.

In this thesis, I enabled the application of game theory to large scale software development using empirical-game theoretic analysis (EGTA). EGTA is a simulation-based game abstraction technique, and this thesis is its first application to the software engineering domain. In [chapter 5](#), I described *TaskAssessor*: an EGTA-based model of task prioritisation for software teams. EGTA is at the core of GTPI (Game-Theoretic Process Improvement), the end-to-end approach to software process improvement approach I proposed in [chapter 4](#).

GTPI uses EGTA models for the diagnosis of software process problems. In [chapter 4](#), I used GTPI to describe how technical debt can accumulate as a consequence of software teams protecting their budget. In [chapter 6](#), I

7 Conclusion and Future Work

showed that bug triage teams — an industry best-practice — do not address priority inflation and, in fact, slow the delivery of bug fixes.

GTPI is not only a diagnosis tool: it can also prescribe solutions. The game-theoretic model of an inefficient software process can be used to test interventions for their improvement. Using GTPI, I found that software teams can address technical debt by including a lightweight code review process in their workflow ([chapter 4](#)). Also, GTPI enabled the design of a new prioritisation process — assessor-throttling — immune to priority inflation ([chapter 6](#)).

7.2 Future Work

EGTA allows the modelling of more complex scenarios than with classic game-theoretic techniques. However, the scale of modern software development can surpass EGTA capabilities. For example, the number of users in the bug reporting dataset ([chapter 5](#)) made it impossible to model bug fixing using only EGTA, so *TaskAssessor* relies on an additional player reduction technique — the twins' reduction — to produce models of tractable size. The design decision of *not* modelling developers as strategic agents ([chapter 6](#)) was also a consequence of EGTA limitations. Including developers as additional players with their corresponding actions would produce an intractable game. In future work, I would like to develop game abstraction techniques tailored to the software development context, that produce models of tractable size while preserving the equilibria of the original game.

Classic game theory's predictions on player behaviour rely on hard assumptions regarding their knowledge, rationality, and motivations. In the field of behavioural economics, researchers are exploring models where these assumptions are relaxed. This research area is intensive on human studies, and model predictions are usually verified empirically via controlled experiments with human participants. For the game-theoretic models in this thesis, I assumed rational players with perfect knowledge of the game, as in classic game theory. In future work, I would like to explore with real developers how often these assumptions hold and, if they not, apply modelling paradigms with less strict rationality assumptions.

7 Conclusion and Future Work

Besides priority inflation and technical debt, I believe software estimation can benefit from game-theoretic analysis. This scenario naturally fits the definition of a game. Considering developers, managers and clients as players; the utility of each player is tied to the actions of their "opponents". Clients want a speedy release, managers need a competitive bid, and developers rely on a buffer to avoid overwork. The main challenge for a GTPI approach to software estimation is data gathering. Given the commercial context, it is not possible to rely on open-source data like in this thesis. Assembling a dataset of project bids with estimate's accuracy is left as future work, as it highly depends on an industrial partner.

7.3 Final Remarks

Inefficiencies in software processes can translate to quality problems in the final software product. Conflicting incentives built inside software practices can produce process issues. I believe that game-theoretic models can be used to identify these problems and remove them. I proposed GTPI as an end-to-end software process improvement framework, using game-theoretic models from anomaly identification to process deployment. In this thesis, I used GTPI to diagnose and prevent budget-driven technical debt and priority inflation, both relevant problems in modern software development. I posit that GTPI, and game-theoretic models in general, can be used to diagnose and fix many other common software process problems.

Bibliography

- [1] IEEE Computer Society, *Guide to the Software Engineering Body of Knowledge (SWEBOK): Version 3.0*, 3rd ed., P. Bourque and R. Fairley, Eds. IEEE Computer Society Press, 2014. [Online]. Available: <https://www.computer.org/education/bodies-of-knowledge/software-engineering>
- [2] B. W. Boehm and R. Turner, "Management challenges to implementing agile processes in traditional development organizations," *IEEE Software*, vol. 22, no. 5, pp. 30–39, 2005. [Online]. Available: <https://doi.org/10.1109/MS.2005.129>
- [3] R. B. Myerson, *Game theory - Analysis of Conflict*. Harvard University Press, 1997. [Online]. Available: <http://www.hup.harvard.edu/catalog/MYEGAM.html>
- [4] J. Nash, "Non-cooperative games," *Annals of mathematics*, pp. 286–295, 1951.
- [5] I. Palacios-Huerta, "Professionals play minimax," *The Review of Economic Studies*, vol. 70, no. 2, pp. 395–415, 2003.
- [6] M. Walker and J. Wooders, "Minimax play at wimbledon," *American Economic Review*, vol. 91, no. 5, pp. 1521–1538, 2001.
- [7] T. Sandholm, "Abstraction for solving large incomplete-information games," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, B. Bonet and S. Koenig, Eds. AAAI Press, 2015, pp. 4127–4131. [Online]. Available: <http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/10039>

Bibliography

- [8] M. P. Wellman, "Methods for empirical game-theoretic analysis," in *Proceedings, The Twenty-First National Conference on Artificial Intelligence and the Eighteenth Innovative Applications of Artificial Intelligence Conference, July 16-20, 2006, Boston, Massachusetts, USA*. AAAI Press, 2006, pp. 1552–1556. [Online]. Available: <http://www.aaai.org/Library/AAAI/2006/aaai06-248.php>
- [9] W. E. Walsh, R. Das, G. Tesauro, and J. O. Kephart, "Analyzing complex strategic interactions in multi-agent systems," in *AAAI-02 Workshop on Game-Theoretic and Decision-Theoretic Agents*, 2002, pp. 109–118.
- [10] R. Madachy, *Software Process Dynamics*. Wiley, 2007.
- [11] J. Münch, O. Armbrust, M. Kowalczyk, and M. Soto, *Software Process Definition and Management*, ser. The Fraunhofer IESE Series on Software and Systems Engineering. Springer Berlin Heidelberg, 2012.
- [12] P. Butcher, *Debug It!: Find, Repair, and Prevent Bugs in Your Code*, ser. Pragmatic Bookshelf Series. Pragmatic Bookshelf, 2009.
- [13] K. Leyton-Brown and Y. Shoham, *Essentials of Game Theory: A Concise Multidisciplinary Introduction*, ser. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2008. [Online]. Available: <https://doi.org/10.2200/S00108ED1V01Y200802AIM003>
- [14] I. Abraham, L. Alvisi, and J. Y. Halpern, "Distributed computing meets game theory: combining insights from two fields," *SIGACT News*, vol. 42, no. 2, pp. 69–76, 2011. [Online]. Available: <https://doi.org/10.1145/1998037.1998055>
- [15] S. Tadelis, *Game Theory: An Introduction*. Princeton University Press, 2013.
- [16] Y. Shoham and K. Leyton-Brown, *Multiagent Systems - Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press, 2009.

Bibliography

- [17] R. Aumann, *Handbook of game theory with economic applications*. Amsterdam New York New York, N.Y., USA: North-Holland Distributors for the U.S. and Canada, Elsevier Science Pub. Co, 1992, vol. 2.
- [18] Y. Chen and D. M. Pennock, "Designing markets for prediction," *AI Magazine*, vol. 31, no. 4, pp. 42–52, 2010. [Online]. Available: <http://www.aaai.org/ojs/index.php/aimagazine/article/view/2313>
- [19] N. Nisan, T. Roughgarden, É. Tardos, and V. V. Vazirani, Eds., *Algorithmic Game Theory*. Cambridge University Press, 2007. [Online]. Available: <https://doi.org/10.1017/CBO9780511800481>
- [20] Y. Shoham, "Computer science and game theory," *Commun. ACM*, vol. 51, no. 8, pp. 74–79, 2008. [Online]. Available: <https://doi.org/10.1145/1378704.1378721>
- [21] M. Grechanik and D. E. Perry, "Analyzing software development as a noncooperative game," in "*Sixth International Workshop on Economics-Driven Software Engineering Research (EDSER-6)*" W9L Workshop - 26th International Conference on Software Engineering. IEE, 2004. [Online]. Available: <https://doi.org/10.1049/ic:20040282>
- [22] The Joint Task Force on Computing Curricula, "Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering," Association for Computing Machinery, New York, NY, USA, Tech. Rep., 2004.
- [23] International Organization for Standardization, International Electrotechnical Commission, and Institute of Electrical and Electronics Engineers, *24765-2017 - ISO/IEC/IEEE International Standard - Systems and software engineering—Vocabulary*, 2nd ed. IEEE, 2017.
- [24] P. Bourque, R. Dupuis, A. Abran, J. W. Moore, and L. L. Tripp, "The guide to the software engineering body of knowledge," *IEEE Software*, vol. 16, no. 6, pp. 35–44, 1999. [Online]. Available: <https://doi.org/10.1109/52.805471>
- [25] SEBoK authors, "The guide to the systems engineering body of knowledge (sebok), v. 2.1," 2019, [Online; accessed 02 Dec 2019]. [Online]. Available: www.sebokwiki.org

Bibliography

- [26] X. Liang and Y. Xiao, "Game theory for network security," *IEEE Communications Surveys and Tutorials*, vol. 15, no. 1, pp. 472–486, 2013. [Online]. Available: <https://doi.org/10.1109/SURV.2012.062612.00056>
- [27] C. T. Do, N. H. Tran, C. S. Hong, C. A. Kamhoua, K. A. Kwiat, E. Blasch, S. Ren, N. Pissinou, and S. S. Iyengar, "Game theory for cyber security and privacy," *ACM Comput. Surv.*, vol. 50, no. 2, pp. 30:1–30:37, 2017. [Online]. Available: <https://doi.org/10.1145/3057268>
- [28] E. Yu, P. Giorgini, N. Maiden, and J. Mylopoulos, *Social Modeling for Requirements Engineering*, ser. Cooperative information systems. MIT Press, 2011.
- [29] K. K. Vajja and P. TV, "Quality attribute game: a game theory based technique for software architecture design," in *Proceeding of the 2nd Annual India Software Engineering Conference, ISEC 2009, Pune, India, February 23-26, 2009*, K. Deshpande, P. Jalote, and S. K. Rajamani, Eds. ACM, 2009, pp. 133–134. [Online]. Available: <https://doi.org/10.1145/1506216.1506244>
- [30] J. García-Galán, P. Trinidad, and A. R. Cortés, "Multi-user variability configuration: A game theoretic approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 574–579. [Online]. Available: <https://doi.org/10.1109/ASE.2013.6693115>
- [31] L. Capra, W. Emmerich, and C. Mascolo, "A micro-economic approach to conflict resolution in mobile computing," in *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*. ACM, 2002, pp. 31–40. [Online]. Available: <https://doi.org/10.1145/587051.587057>
- [32] N. Kitagawa, H. Hata, A. Ihara, K. Kogiso, and K. Matsumoto, "Code review participation: game theoretical modeling of reviewers in gerrit datasets," in *Proceedings of the 9th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE@ICSE 2016, Austin, Texas, USA, May 16, 2016*. ACM, 2016, pp. 64–67. [Online]. Available: <https://doi.org/10.1145/2897586.2897605>

Bibliography

- [33] L. Feijs, "Prisoner's dilemma in software testing," in *Proceedings 7e Nederlandse Testdag (Eindhoven, The Netherlands, November 8, 2001)*, ser. Computer Science Reports, L. Feijs, N. Goga, S. Mauw, and T. Willemse, Eds. Technische Universiteit Eindhoven, 2001, pp. 65–80.
- [34] N. Kukreja, W. G. J. Halfond, and M. Tambe, "Randomizing regression tests using game theory," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, E. Denney, T. Bultan, and A. Zeller, Eds. IEEE, 2013, pp. 616–621. [Online]. Available: <https://doi.org/10.1109/ASE.2013.6693122>
- [35] M. Rao, D. C. Parkes, M. I. Seltzer, and D. F. Bacon, "A framework for incentivizing deep fixes," in *Proceedings of the AAI Workshop on Incentives and Trust in E-Communitates*, 2015.
- [36] V. Sazawal and N. Sudan, "Modeling software evolution with game theory," in *Trustworthy Software Development Processes, International Conference on Software Process, ICSP 2009 Vancouver, Canada, May 16-17, 2009 Proceedings*, ser. Lecture Notes in Computer Science, Q. Wang, V. Garousi, R. J. Madachy, and D. Pfahl, Eds., vol. 5543. Springer, 2009, pp. 354–365. [Online]. Available: https://doi.org/10.1007/978-3-642-01680-6_32
- [37] G. Bavota, R. Oliveto, A. D. Lucia, G. Antoniol, and Y. Guéhéneuc, "Playing with refactoring: Identifying extract class opportunities through game theory," in *26th IEEE International Conference on Software Maintenance (ICSM 2010), September 12-18, 2010, Timisoara, Romania*. IEEE Computer Society, 2010, pp. 1–5. [Online]. Available: <https://doi.org/10.1109/ICSM.2010.5609739>
- [38] H. Hata, T. Todo, S. Onoue, and K. Matsumoto, "Characteristics of sustainable OSS projects: A theoretical and empirical study," in *8th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2015, Florence, Italy, May 18, 2015*, A. Begel, R. Prikladnicki, Y. Dittrich, C. R. B. de Souza, A. Sarma, and S. Athavale, Eds. IEEE Computer Society, 2015, pp. 15–21. [Online]. Available: <https://doi.org/10.1109/CHASE.2015.9>

Bibliography

- [39] N. V. Oza, "Game theory perspectives on client: Vendor relationships in offshore software outsourcing," in *Proceedings of the 2006 International Workshop on Economics Driven Software Engineering Research*, ser. EDSE '06. New York, NY, USA: ACM, 2006, pp. 49–54. [Online]. Available: <http://doi.acm.org/10.1145/1139113.1139125>
- [40] D. F. Bacon, E. Bokelberg, Y. Chen, I. A. Kash, D. C. Parkes, M. Rao, and M. Sridharan, "Software economies," in *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, G. Roman and K. J. Sullivan, Eds. ACM, 2010, pp. 7–12. [Online]. Available: <https://doi.org/10.1145/1882362.1882365>
- [41] D. F. Bacon, D. C. Parkes, Y. Chen, M. Rao, I. A. Kash, and M. Sridharan, "Predicting your own effort," in *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes)*, W. van der Hoek, L. Padgham, V. Conitzer, and M. Winikoff, Eds. IFAAMAS, 2012, pp. 695–702. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2343796>
- [42] B. Lagesse, "A game-theoretical model for task assignment in project management," in *2006 IEEE International Conference on Management of Innovation and Technology*, vol. 2, June 2006, pp. 678–680.
- [43] M. Yilmaz, R. V. O'Connor, and J. Collins, "Improving software development process through economic mechanism design," in *Systems, Software and Services Process Improvement - 17th European Conference, EuroSPI 2010, Grenoble, France, September 1-3, 2010. Proceedings*, ser. Communications in Computer and Information Science, A. Riel, R. O'Connor, S. Tichkiewitch, and R. Messnarz, Eds., vol. 99. Springer, 2010, pp. 177–188. [Online]. Available: https://doi.org/10.1007/978-3-642-15666-3_16
- [44] M. Yilmaz and R. V. O'Connor, "A software process engineering approach to improving software team productivity using socioeconomic mechanism design," *ACM SIGSOFT Software Engineering Notes*, vol. 36, no. 5, pp. 1–5, 2011. [Online]. Available: <https://doi.org/10.1145/2020976.2020998>

Bibliography

- [45] O. Hazzan and Y. Dubinsky, "Social perspective of software development methods: The case of the prisoner dilemma and extreme programming," in *Extreme Programming and Agile Processes in Software Engineering, 6th International Conference, XP 2005, Sheffield, UK, June 18-23, 2005, Proceedings*, ser. Lecture Notes in Computer Science, H. Baumeister, M. Marchesi, and M. Holcombe, Eds., vol. 3556. Springer, 2005, pp. 74–81. [Online]. Available: https://doi.org/10.1007/11499053_9
- [46] Y. Wang and D. F. Redmiles, "Cheap talk, cooperation, and trust in global software engineering - an evolutionary game theory model with empirical support," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2233–2267, 2016. [Online]. Available: <https://doi.org/10.1007/s10664-015-9407-3>
- [47] E. Hasnain, T. Hall, and M. J. Shepperd, "Using experimental games to understand communication and trust in agile software teams," in *6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013, San Francisco, CA, USA, May 25, 2013*. IEEE Computer Society, 2013, pp. 117–120. [Online]. Available: <https://doi.org/10.1109/CHASE.2013.6614745>
- [48] M. A. Nowak, *Evolutionary dynamics : exploring the equations of life*. Cambridge, Massachusetts: The Belknap Press of Harvard University Press, 2006.
- [49] É. Tardos and V. V. Vazirani, "Basic solution concepts and computational issues," in *Algorithmic game theory*, N. Nisan, T. Roughgarden, É. Tardos, and V. V. Vazirani, Eds. Cambridge University Press, 2007, ch. 1, pp. 3–28.
- [50] M. P. Wellman, D. M. Reeves, K. M. Lochner, S. Cheng, and R. Suri, "Approximate strategic reasoning through hierarchical reduction of large symmetric games," in *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*, M. M. Veloso and S. Kambhampati, Eds. AAAI Press / The MIT Press, 2005, pp. 502–508. [Online]. Available: <http://www.aaai.org/Library/AAAI/2005/aaai05-079.php>

Bibliography

- [51] B. Wiedenbeck and M. P. Wellman, "Scaling simulation-based game analysis through deviation-preserving reduction," in *International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2012, Valencia, Spain, June 4-8, 2012 (3 Volumes)*, W. van der Hoek, L. Padgham, V. Conitzer, and M. Winikoff, Eds. IFAAMAS, 2012, pp. 931–938. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2343830>
- [52] S. G. Ficici, D. C. Parkes, and A. Pfeffer, "Learning and solving many-player games through a cluster-based representation," *arXiv preprint arXiv:1206.3253*, 2012.
- [53] K. Holyoak and R. Morrison, *The Oxford Handbook of Thinking and Reasoning*, ser. Oxford Library of Psychology. OUP USA, 2012.
- [54] A. Sanjab, W. Saad, and T. Başar, "Prospect theory for enhanced cyber-physical security of drone delivery systems: A network interdiction game," in *2017 IEEE International Conference on Communications (ICC)*. IEEE, 2017, pp. 1–6.
- [55] W. Saad, A. L. Glass, N. B. Mandayam, and H. V. Poor, "Toward a consumer-centric grid: A behavioral perspective," *Proceedings of the IEEE*, vol. 104, no. 4, pp. 865–882, 2016.
- [56] The Register, "Talk of tech innovation is bullsh*t. shut up and get the work done – says linus torvalds," 2017, accessed: 17-09-2018. [Online]. Available: https://www.theregister.co.uk/2017/02/15/think_different_shut_up_and_work_harder_says_linus_torvalds/
- [57] S. McConnell, *Software Estimation: Demystifying the Black Art*, ser. Developer Best Practices. Pearson Education, 2006.
- [58] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, May 2016, pp. 619–630.
- [59] T. Sandholm, "The state of solving large incomplete-information games, and application to poker," *AI Magazine*, vol. 31, no. 4, pp. 13–32, 2010. [Online]. Available: <http://www.aaai.org/ojs/index.php/aimagazine/article/view/2311>

Bibliography

- [60] F. Sarro, "Predictive analytics for software testing: Keynote paper," in *Proceedings of the 11th International Workshop on Search-Based Software Testing*, ser. SBST '18. New York, NY, USA: ACM, 2018, pp. 1–1. [Online]. Available: <http://doi.acm.org/10.1145/3194718.3194730>
- [61] M. Lavallée and P. N. Robillard, "Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 677–687.
- [62] P. Goodliffe, *Becoming a Better Programmer: A Handbook for People Who Care About Code*. O'Reilly Media, 2014.
- [63] A. Stellman and J. Greene, *Learning agile: Understanding scrum, XP, lean, and kanban*. "O'Reilly Media, Inc.", 2014.
- [64] A. Greasley, "A comparison of system dynamics and discrete event simulation," in *Proceedings of the 2009 Summer Computer Simulation Conference*. Society for Modeling & Simulation International, 2009, pp. 83–87.
- [65] J. Banks, J. Carson, and B. Nelson, *Discrete-event System Simulation*. Prentice Hall, 2010.
- [66] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, ser. Always learning. Pearson, 2016.
- [67] R. D. McKelvey, A. M. McLennan, and T. L. Turocy, "Gambit: Software Tools for Game Theory, Version 15," 2014. [Online]. Available: <http://www.gambit-project.org>
- [68] Q. Mi and J. Keung, "An empirical analysis of reopened bugs based on open source projects," in *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 2016, p. 37.
- [69] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67–120, 2012.

Bibliography

- [70] N. H. Madhavji, D. Holtje, Won Kook Hong, and T. Bruckhaus, "Elicit: a method for eliciting process models," in *Proceedings of the Third International Conference on the Software Process. Applying the Software Process*, 1994, pp. 111–122.
- [71] M. Verlage, "Multi-view modelling of software processes," in *Software Process Technology, Third European Workshop, EWSPT '94, Villard de Lans, France, February 7-9, 1994, Proceedings*, ser. Lecture Notes in Computer Science, B. Warboys, Ed., vol. 772. Springer, 1994, pp. 123–126. [Online]. Available: https://doi.org/10.1007/3-540-57739-4_17
- [72] I. Jacobson, G. Booch, and J. E. Rumbaugh, *The unified software development process - the complete guide to the unified process from the original designers*, ser. Addison-Wesley object technology series. Addison-Wesley, 1999.
- [73] H. D. Mills, M. G. Dyer, and R. C. Linger, "Cleanroom software engineering," *IEEE Software*, vol. 4, no. 5, pp. 19–25, 1987. [Online]. Available: <https://doi.org/10.1109/MS.1987.231413>
- [74] T. Bauer, F. Bohr, D. Landmann, T. Beletski, R. Eschbach, and J. Poore, "From requirements to statistical testing of embedded systems," in *Fourth International Workshop on Software Engineering for Automotive Systems (SEAS '07)*, 2007, pp. 3–3.
- [75] A. Trendowicz, J. Heidrich, J. Münch, Y. Ishigai, K. Yokoyama, and N. Kikuchi, "Development of a hybrid cost estimation model in an iterative manner," in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds. ACM, 2006, pp. 331–340. [Online]. Available: <https://doi.org/10.1145/1134285.1134332>
- [76] Task Force for SPEM, "Software & systems process engineering meta-model specification," Object Management Group (OMG), Standard, Apr. 2008. [Online]. Available: <https://www.omg.org/spec/SPEM/2.0/PDF>
- [77] C. P. Team, "Capability maturity model® integration (cmmi), version 1.1–continuous representation," 2002.

Bibliography

- [78] K. E. Emam, W. Melo, and J.-N. Drouin, *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*, 1st ed. Washington, DC, USA: IEEE Computer Society Press, 1997.
- [79] M. B. Doar, *Practical Development Environments*. "O'Reilly Media, Inc.", 2005.
- [80] J. A. Whittaker, J. Arbon, and J. Carollo, *How Google tests software*. Addison-Wesley, 2012.
- [81] A. Page, K. Johnston, and B. Rollison, *How we test software at Microsoft*. Microsoft Press, 2008.
- [82] Atlassian, "Bug tracking for JIRA server," 2002, accessed: 2019-03-22. [Online]. Available: <https://jira.atlassian.com/browse/JRASERVER-886>
- [83] S. Brue and R. Grant, *The Evolution of Economic Thought*. Cengage Learning, 2012.
- [84] P. A. Laplante and N. B. Ahmad, "Pavlov's bugs: Matching repair policies with rewards," *IT Professional*, vol. 11, no. 4, pp. 45–51, 2009. [Online]. Available: <https://doi.org/10.1109/MITP.2009.80>
- [85] R. Axelrod, *The Evolution of Cooperation: Revised Edition*. Basic Books, 2009.
- [86] R. Banfield, *Product leadership : how top product managers Launch awesome products and build successful teams*. Beijing: O'Reilly, 2017.
- [87] M. Krogerus and R. Tschäppeler, *The Decision Book: Fifty models for strategic thinking (New Edition)*. Profile, 2017.
- [88] R. Black, *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. Wiley, 2013.
- [89] GitHub, "About labels - user documentation," 2018, accessed: 2018-10-19. [Online]. Available: <https://help.github.com/articles/about-labels/>

Bibliography

- [90] Atlassian, “What is an issue (v6.4) - atlassian documentation,” 2018, accessed: 2018-10-19. [Online]. Available: <https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html>
- [91] MediaWiki, “Bugzilla/fields - mediawiki,” 2018, accessed: 2018-10-19. [Online]. Available: <https://www.mediawiki.org/wiki/Bugzilla/Fields>
- [92] A. Savchenko, “Github labels for better workflows - yoast,” 2015, accessed: 2018-10-19. [Online]. Available: <https://yoast.com/dev-blog/github-labels/>
- [93] Mediocre Laboratories, “How we use labels on github issues at mediocre laboratories,” 2014, accessed: 2018-10-19. [Online]. Available: <https://mediocre.com/forum/topics/how-we-use-labels-on-github-issues-at-mediocre-laboratories>
- [94] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian, “The promises and perils of mining github,” in *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, P. T. Devanbu, S. Kim, and M. Pinzger, Eds. ACM, 2014, pp. 92–101. [Online]. Available: <https://doi.org/10.1145/2597073.2597074>
- [95] V. Cosentino, J. L. C. Izquierdo, and J. Cabot, “A systematic mapping study of software development with github,” *IEEE Access*, vol. 5, pp. 7173–7192, 2017. [Online]. Available: <https://doi.org/10.1109/ACCESS.2017.2682323>
- [96] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki, “Github projects. quality analysis of open-source software,” in *Social Informatics - 6th International Conference, SocInfo 2014, Barcelona, Spain, November 11-13, 2014. Proceedings*, ser. Lecture Notes in Computer Science, L. M. Aiello and D. A. McFarland, Eds., vol. 8851. Springer, 2014, pp. 80–94. [Online]. Available: https://doi.org/10.1007/978-3-319-13734-6_6
- [97] P. Lavrakas, *Encyclopedia of Survey Research Methods: A-M.*, ser. A SAGE reference publication. SAGE Publications, 2008.

Bibliography

- [98] Carlos Gavidia-Calderon, Federica Sarro, Mark Harman, and Earl T. Barr, "Improving software processes via empirical game theory," 2015, accessed: 2019-02-14. [Online]. Available: <https://cptanalatriste.github.io/priority-inflation-site/>
- [99] Atlassian, "JIRA- issue and project tracking software," 2018, accessed: 2018-10-21. [Online]. Available: <https://www.atlassian.com/software/jira>
- [100] Apache Software Foundation, "System dashboard - ASF JIRA," 2018, accessed: 2018-10-21. [Online]. Available: <https://issues.apache.org/jira/>
- [101] —, "Github - the apache software foundation," 2018, accessed: 2018-10-21. [Online]. Available: <https://github.com/apache>
- [102] Atlassian, "What is an issue (v6.3) - atlassian documentation," 2015, accessed: 2018-10-21. [Online]. Available: <https://confluence.atlassian.com/jira063/what-is-an-issue-683542485.html>
- [103] J. Spolsky, *Joel on software : and on diverse and occasionally related matters that will prove of interest to software developers, designers, and managers, and to those who, whether by good fortune or ill luck, work with them in some capacity.* Berkeley, CA: Apress, 2004.
- [104] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [105] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.
- [106] M. Rath, J. Rendall, J. L. C. Guo, J. Cleland-Huang, and P. Mäder, "Traceability in the wild: automatically augmenting incomplete trace links," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27*

Bibliography

- June 03, 2018, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 834–845. [Online]. Available: <https://doi.org/10.1145/3180155.3180207>
- [107] D. Radigan, “Organizing issues with priority to optimize delivery,” 2014, accessed: 2019-03-24. [Online]. Available: <https://www.atlassian.com/blog/jira-software/organizing-issues-priority-optimize-delivery>
- [108] G. Fishman, *Discrete-Event Simulation : Modeling, Programming, and Analysis*. New York, NY: Springer New York, 2001.
- [109] K. G. Müller and T. Vignaux. (2011) Simulation with simpy - in depth manual. [Online]. Available: <https://pythonhosted.org/SimPy/Manuals/Manual.html>
- [110] J. Watkins and S. Mills, *Testing IT: an off-the-shelf software testing process*. Cambridge University Press, 2010.
- [111] D. Gross, *Fundamentals of queueing theory*. John Wiley & Sons, 2008.
- [112] S. Kelly, *Domain-specific modeling : enabling full code generation*. Hoboken, N.J: Wiley-Interscience IEEE Computer Society, 2008.
- [113] M. Lacey, *The Scrum field guide : practical advice for your first year*. Addison-Wesley Professional, 2012.
- [114] H. Naguib, N. Narayan, B. Brügge, and D. Helal, “Bug report assignee recommendation using activity profiles,” in *Proceedings of the 10th Working Conference on Mining Software Repositories, MSR '13, San Francisco, CA, USA, May 18-19, 2013*, T. Zimmermann, M. D. Penta, and S. Kim, Eds. IEEE Computer Society, 2013, pp. 22–30. [Online]. Available: <https://doi.org/10.1109/MSR.2013.6623999>
- [115] J. Anvik and G. C. Murphy, “Reducing the effort of bug report triage: Recommenders for development-oriented decisions,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 10:1–10:35, 2011. [Online]. Available: <https://doi.org/10.1145/2000791.2000794>

Bibliography

- [116] L. Crispin, *Agile testing : a practical guide for testers and agile teams*. Upper Saddle River, NJ: Addison-Wesley, 2009.
- [117] E. Brechner, *Agile project management with Kanban*. Redmond, WA: Microsoft Press, 2015.
- [118] S. S. Gokhale and R. E. Mullen, "Queuing models for field defect resolution process," in *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA, 2006*, pp. 353–362.
- [119] B. Luong and D.-B. Liu, "Resource allocation model in software development," in *Annual Reliability and Maintainability Symposium. 2001 Proceedings. International Symposium on Product Quality and Integrity (Cat. No.01CH37179), 2001*, pp. 213–218.
- [120] S. Bose, *An Introduction to Queueing Systems*. Boston, MA: Springer US Imprint Springer, 2002.
- [121] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 2007, pp. 1–8.
- [122] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010, 2010*, pp. 495–504.
- [123] P. Hooimeijer and W. Weimer, "Modeling bug report quality," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 34–43.
- [124] S.-H. Kim and B. L. Nelson, "Selecting the best system," *Handbooks in operations research and management science*, vol. 13, pp. 501–534, 2006.
- [125] B. L. Nelson and F. J. Matejczik, "Using common random numbers for indifference-zone selection and multiple comparisons in simulation," *Management Science*, vol. 41, no. 12, pp. 1935–1945, 1995.

Bibliography

- [126] K. Inoue, S. E. Chick, and C.-H. Chen, "An empirical evaluation of several methods to select the best system," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 9, no. 4, pp. 381–407, 1999.
- [127] C. Gavidia-Calderon, "TheFed at GitHub," 2017, accessed: 2019-03-10. [Online]. Available: <https://github.com/cptanalatriste/inflation-tracker-extension>
- [128] Atlassian, "The JIRA Cloud Platform REST API," 2019, accessed: 2019-03-10. [Online]. Available: <https://docs.atlassian.com/jira/REST/cloud/>
- [129] Bugzilla, "WebService API Reference," 2019, accessed: 2019-03-10. [Online]. Available: <http://bugzilla.readthedocs.io/en/latest/api/>