The Journal of Systems and Software 159 (2020) 110424

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Game-theoretic analysis of development practices: Challenges and opportunities

Carlos Gavidia-Calderon*, Federica Sarro, Mark Harman, Earl T. Barr

Department of Computer Science, University College London, London, UK

ARTICLE INFO

Article history:
Received 15 March 2019
Revised 29 May 2019
Accepted 17 September 2019
Available online 18 September 2019

Keywords:
Game theory
Empirical analysis
Technical debt
Software engineering practices

ABSTRACT

Developers continuously invent new practices, usually grounded in hard-won experience, not theory. Game theory studies cooperation and conflict; its use will speed the development of effective processes. A survey of game theory in software engineering finds highly idealised models that are rarely based on process data. This is because software processes are hard to analyse using traditional game theory since they generate huge game models. We are the first to show how to use game abstractions, developed in artificial intelligence, to produce tractable game-theoretic models of software practices. We present Game-Theoretic Process Improvement (GTPI), built on top of empirical game-theoretic analysis. Some teams fall into the habit of preferring "quick-and-dirty" code to slow-to-write, careful code, incurring technical debt. We showcase GTPI's ability to diagnose and improve such a development process. Using GTPI, we discover a lightweight intervention that incentivises developers to write careful code: add a *single* code reviewer who needs to catch *only* 25% of kludges. This 25% accuracy is key; it means that a reviewer does not need to examine each commit in depth, making this process intervention cost-effective.

1. Introduction

What's important is not just to develop the technology;
it's to develop the processes.

—Hal Abelson

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" (Talk of Tech Innovation, 2018). 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, course-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 (McConnell, 2006; Sarro et al., 2016), 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 (Butcher, 2009).

These software processes can be viewed as games. Game theory studies mathematical models of conflict and cooperation (Myerson, 1991), and has already been used in the analysis of complex scenarios like arms races between nations or duopolies in markets (Gibbons, 1992). One of its more important findings is the *Nash equilibrium* (NE): a game outcome where players have no incentive to deviate by choosing different actions. Real-world scenarios where the players have mastered the game — like in professional sports— converge on their NE (Palacios-Huerta, 2003; Walker and Wooders, 2001). When we model software processes as games, we can compute their NE to diagnose their problems and prescribe fixes when needed.

We believe that most software processes are emergent phenomena that arise to solve problems. Since they 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. *Mechanism design* identifies and fixes games whose NE is undesirable. Players use strategies to decide their

* Corresponding author.
E-mail addresses: carlos.gavidia.15@ucl.ac.uk (C. Gavidia-Calderon), f.sarro@ucl.ac.uk (F. Sarro), mark.harman@ucl.ac.uk (M. Harman), e.barr@ucl.ac.uk (E.T. Barr).

https://doi.org/10.1016/j.jss.2019.110424

actions. *Game designers* apply mechanism design to formulate games whose NE favours desirable strategies. We are proposing that process engineers (Münch et al., 2012) be game designers, who craft software processes considering cooperation incentives and conflict avoidance mechanisms.

Despite game theory's obvious applicability to software processes, our survey of the state of the art (Section 3) shows that 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 (Sandholm, 2010).

Our research agenda exploits abstraction 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 extensive form games (Sandholm, 2015). EGTA converts an extended form game into a normal-form game that only considers a subset of the actions available in the full game and the payoffs are expected values obtained through simulation. EGTA relies heavily on data for simulation input analysis and output validation. Fortunately, software engineers can now easily access a wealth of data building and using software products (Sarro, 2018). Our proposed solution — called Game-Theoretic Process Improvement (GTPI) — relies on this data and EGTA models to improve software development processes (Section 4).

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 (Lavallée and Robillard, 2015). We use this problem to showcase GTPI.

Our contributions follow:

- A survey on game-theoretic analysis of software development practices.
- An introduction to EGTA for software engineers; and
- A complete, end-to-end analysis and fix of budget-driven technical debt: we diagnose it using game theory, then apply mechanism design to suggest a simple and inexpensive change — the addition of a single reviewer — and validate the new process via its NE.

2. Background

In this section, we review key concepts from game theory using a software engineering scenario to illustrate them. We also discuss the issue of game representation: the number of players, actions and rounds over time have a large impact on the size of the game-theoretic model.

In game-theoretic context, a *game* is a scenario where multiple self-interested agents, or *players*, interact and their actions impact the perceived utility of all of them. This definition fits nicely with the traditional definition of game. For example, in rock-paper-scissors winning is a function of your play — selecting an *action* — and the play of your opponent.

Table 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.

	B: cooperate	*B: not cooperate*
A: cooperate	$A = \$150,\ B = \$150$	$A = \$50,\ B = \$200$
A: not cooperate	$A = \$200,\ B = \$50$	$A = \$100,\ B = \$100$

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. We represent this game with a *payoff table* — shown in Table 1 — that contains payoff information per player given the actions they performed.

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 (Leyton-Brown and Shoham, 2008). In an NE, all the players adopt the best response to their opponents' strategies: any deviation lowers a player's payoff. This explains the stability of an NE. Nash proved that every finite game has at least one NE (Nash, 1951).

To put this in context, we return to our freelancers. 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 NE for both players is to adopt the same strategy: ignore each other. Any deviation from this strategy allows their colleague to take advantage. Hence, NE 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 our freelancers is an instance of the *prisoner's dilemma*: a game used to explain cooperation — or the lack of it — in real-world scenarios like global warming and cigarette advertisement.

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 (Tadelis, 2013). In contrast, *bounded rationality* approaches model rationality within the limits of attention, information, and mental capabilities of the decision maker (Holyoak and Morrison, 2012). *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 (Sanjab et al., 2017) and to the study of smart-grid adoption (Saad et al., 2016). *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

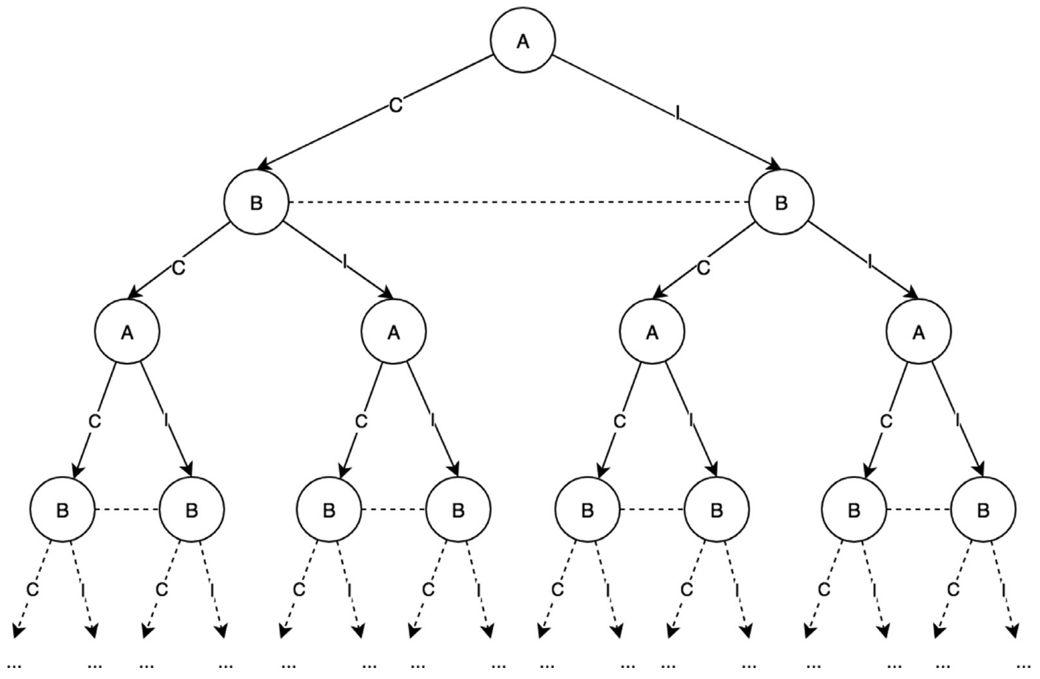

Fig. 1. Extensive form game for a multi-project freelancer's dilemma: this figure contains the part of the tree corresponding to the first two projects. The size of this representation grows with the number of projects, freelancers and actions available to them. EGTA (Section 4) is crucial for managing this explosive growth.

and emotions in the game-theoretic model. In this work, we adopt the conventional game-theory approach that predicts convergence to NE under its rationality assumptions. We believe teams of experienced developers working over well-know codebases, like in open-source projects or in mature software maintenance teams, meet these assumptions. Our results are already strong despite being conservative and imprecise and they can only improve by adopting a bounded rationality approach closer to team behaviour. This is left as future work.

Game theory has been widely applied. It has already been used to optimise student recruitment in American hospitals, improve the auction of telecom operating licences in Britain and to analyse transport networks in New York (Prison Breakthrough, 2018). In *static games*, players select their actions independently and simultaneously, like in rock-paper-scissors. *Dynamic games,* like chess or poker, unfold over time and require players to sequence their actions. The most common representation for dynamic games is the *extensive form* (Tadelis, 2013). Extensive-form games can be described via game trees, like the one in Fig. 1. The non-terminal nodes of the game tree are called *choice nodes*. In each choice node, a player is required to select an action and the available actions correspond to the outgoing edges of the choice node. A *pure strategy in an extensive form game* defines which action to take at each player's choice node, while a *mixed strategy* is a probability distribution over these pure strategies (Leyton-Brown and Shoham, 2008). The size of the game tree can become inconveniently large when dealing with real-world scenarios. These sizes quickly become unmanageable for NE calculation algorithms, hence abstraction is needed.

3. A survey of game theory applied to software engineering

In the Section 2, we used game theory to analyse freelancers in a software project and showed its potential as a formal technique to analyse development practices. Other popular approaches are less formal but based on the consultants' hard-won experience, although they sometimes lack arguments to back their adoption (The End of Software Engineering, 2019). Bayesian networks

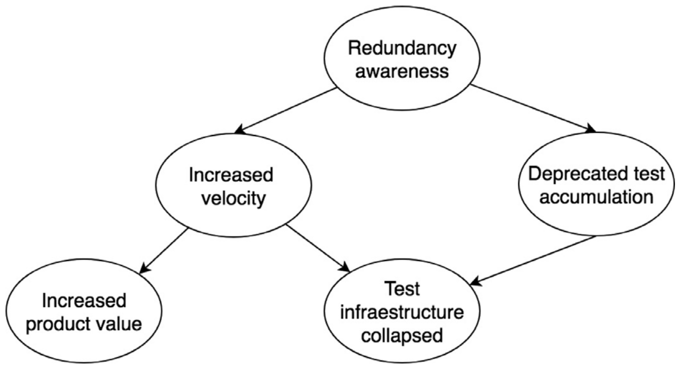

Fig. 2. A Bayesian network for the tragedy of the test suite (Section 5): redundancy awareness, understood as the perception that underperforming can get a developer fired, influences the increase of velocity and the accumulation of deprecated tests. Higher velocity and the accumulation of deprecated tests influence the collapse of test infrastructure.

have also been proposed for process improvement, and due to their use of models and their formal approach they are the closest to game theory (Fenton and Neil, 2000). We start this section by exploring Bayesian networks, then review the work published so far in game-theoretic applications to software engineering.

Bayesian Networks for Process Improvement Scenarios like the tragedy of the test suite (Section 5) can be analysed with tools from decision theory, like the Bayesian network in Fig. 2, but this approach fails to capture strategic interactions *between agents*. Decision theory can be seen as "the study of games against nature", where nature behaves stochastically instead of been driven by self-interest (Parsons and Wooldridge, 2002). Game theory models agents' interests and how their interactions affect these interests. In game theory, players, actions, and strategy profiles are all first-class elements. Model elicitation is the coal face of process improvement, and it is here that game theoretic models shine, because their primitives are easy to understand and are a natural fit for modelling software processes. In contrast, building Bayesian networks rests on eliciting random variables that can have unclear conceptual boundaries. Bayesian networks can, of course, model game theoretic models, at the cost of further complicating their models.

Game Theory for Software Engineering

Game theory has been used before for exploring conflict and cooperation in a software development context. Below, we survey game-theoretic analysis of five types of interactions: interactions between developers, development and testing conflicts, interactions driven by management, role-neutral interactions, and conflicts external to the development team. However, the proposed models do not scale to global software development scenarios with large teams and fast releases (Sandholm, 2015).

Researchers have modelled strategic scenarios between developers on a software team. Lagesse explored task assignment between developers, using cooperative game theory to obtain an ideal assignment considering the developers' preferences (Lagesse, 2006). Hasnain et al. modelled stand up meetings, where developers had the option to work or shirk responsibility (Hasnain et al., 2013). Bavota et al. approached refactoring as a game between developers, where developers compete for refactoring opportunities on a common class to refactor (Bavota et al., 2010). Kitagawa et al. explored the collaboration between code reviewers inside a software team, proposing that this scenario constitutes an instance of the snowdrift game (Kitagawa et al., 2016). These papers succeed in identifying conflicting goals between software developers. In the same fashion, all the games presented in this paper — the freelancer's dilemma (Section 2), technical debt (Section 4), and the test suite tragedy (Section 5) — fall in this category. However, since they use techniques from classical game theory, their models cannot scale to large software teams operating over time, as explained in Section 2.

Other authors have explored scenarios where developers are in conflict with the testing team. Feijs models testing as a game between developers and testers, where the actions for both of them are to do a good job or a poor job (Feijs, 2001). Kukreja et al. modelled testing as a security game: testers are defenders that have a limited budget for test case execution while developers are the attackers who commit poor quality code (Kukreja et al., 2013). The tension between the testing and the developer role fits naturally into the game-theoretic domain, and we expect to study it in the future. As with previous work, they rely on classic game theoretic approaches that limit their scalability.

The connection between software project management and game theory was identified by Boehm and Ross (1989). Models of software process improvement normally have the project manager as player. Hata et al. modelled collaboration in an open-source project as a game between project leaders and contributors; the project leader's goal is to facilitate contribution and his actions include doing nothing, writing documentation, or writing installation scripts (Hata et al., 2015). Yilmaz et al. focused on team members' characteristics and personalty traits and their impact on productivity (Yilmaz et al., 2010; Yilmaz and O'Connor, 2011). Bacon et al. proposed viewing software development as an economy. They present software estimation as an example, where project managers are rewarded by accurate estimations and developers benefit from quick feature delivery (Bacon et al., 2010). Following that vision, Rao et al. suggested a software maintenance game where developers compete by submitting deep or shallow fixes, comparable with the fix-kludge scenario we presented in Section 4 (Rao et al., 2015). Rao et al. also mention the problem of scaling game-theoretic models. They address this by using mean field games, which support a large number of players. Some interesting topics covered by these authors are mechanism design (Yilmaz et al., 2010; Bacon et al., 2010; Rao et al., 2015), data-based validation (Hata et al., 2015) and game abstraction (Bacon et al., 2010). However, they studied each of these topics in isolation. A key contribution of this work is to integrate all these concepts into a formal framework for process improvement at scale, from inception to deployment.

There are also approaches that model team members regardless of their role in the software process. Wang and Redmiles (2016) explore the impact of informal talk in the development of trust using evolutionary game theory. Hazzan and Dubinsky (2005) described how cooperation and defection would impact specific software development practices. Again, since no game abstraction approach was adopted, those game-theoretic models do not scale.

Strategic interactions outside the software team have also been explored, especially with end-users and clients. Since these proposals focus on clients and users, they obviate software development practices, which is a focus of this research agenda. García-Galán et al. reconcile conflicting user configurations in product design by adopting a cooperative game theory approach (García-Galán et al., 2013). Grechanik and Perry model software development as a game with three types of players: managers, developers and customers (Grechanik and Perry, 2004). Oza explores that idea that software outsourcing can be modelled as a game between software vendors and their clients, configuring a prisoner's dilemma scenario (Oza, 2006). Sazawal and Sudan model software evolution as an extensive form game between the software designer and its end-users (Sazawal and Sudan, 2009). Kumar et al. proposed a novel marketplace style for cloud computing providers and customers based on the double auction mechanism (Kumar et al., 2018).

4. Research agenda: game-theoretic software process improvement

Since 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. Dynamic games and the extensive-form (Section 2) are more suitable for representing software processes than static games, given static games inability to handle time (Tadelis, 2013). However, real-world software development — with large teams working over multiple releases — generate immense game trees. Hence, abstraction is needed. *Empirical game-theoretic analysis (EGTA)* is one of these abstraction approaches (Wellman, 2006; Walsh et al., 2002). 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 our freelancer scenario, let us assume that the freelancers work together for an undetermined number of projects under the contractual conditions of Table 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 handcrafted following first principles, based on the properties of the system under study (Walsh et al., 2002).

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.

Game-Theoretic Process Improvement (GTPI) is our proposed software process improvement framework. It relies on EGTA ab-

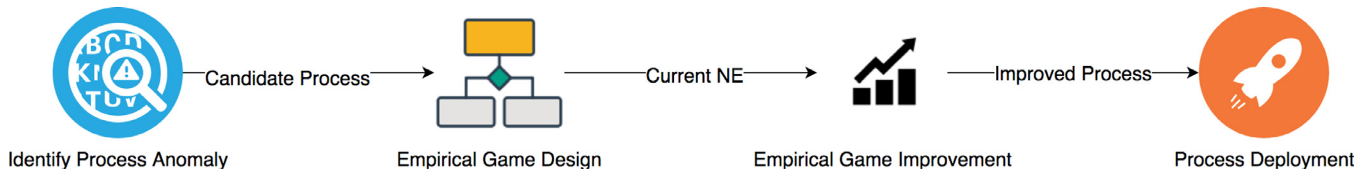

Fig. 3. The GTPI approach starts by detecting a candidate process that exhibits an aberrant behaviour. Such a process is then modelled using EGTA and the Nash equilibrium is obtained. In case the equilibrium is not the one desired, the EGTA model is updated iteratively until a desirable one is obtained. Finally, the improved process is adopted by the team.

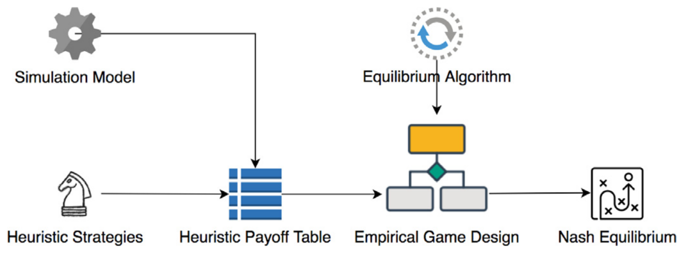

Fig. 4. Empirical game design: the process engineer builds a heuristic payoff table using heuristic strategies as actions: The entries of this table are expected payoff values obtained via simulation. They can later use any algorithm to calculate the NE of the empirical game, once the payoff matrix has been built.

stractions to model software processes, diagnose process anomalies and prescribe solutions to remove them. It is summarised in Fig. 3. Anomalous processes are often itchy: something about them mystifies or annoys their participants. Some behaviours that are apparently irrational appear due to incorrect incentives. *Identifying process anomalies* is the first step of our 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 (Lavallée and Robillard, 2015). Practitioners (Goodliffe, 2014; Stellman and Greene, 2014) and researchers (Lavallée and Robillard, 2015; Rao et al., 2015) 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 solution.[1]

The next step is the *empirical game design*, where we model the process to improve. Underlying Fig. 4 are two models. One is a simulation model designed to capture real world behaviour. To the simulation model, we appliy 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 (Münch et al., 2012). Here, we adopted discrete-event simulation because it is easy to reproduce and evaluate (Greasley, 2009).

The parameters for our simulation model are described in Table 2. Technical debt can arise because developers write sloppy code under pressure. Why? 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 we look to the output of the process; F_i features per release. We 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 our model works a Kanban board with three columns: To-Do, In-Progress and Done[2] Tasks appear as To-Do’s with probability I. On average, a task project takes T days to reach “Done” and increase F_i.

Table 2
Input and output variables of the simulation model of the development team: 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

Technical debt is a *tragedy of the commons*: the codebase quality, the shared resource or *common*, progressively degrades as the team pushes kludges. So, we define our model to make kludges faster to code than fixes at expense of codebase quality and with a higher risk of rework. To this end, we 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 actions developers can take, our simulation model gives a developer only two: write a kludge or code a time-consuming fix, as Fig. 5 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 compare the simulation outputs with actual process data (Banks et al., 2010). Consolidating data across sources is non-trivial and time-consuming: we discuss how one might approach this problem in Section 5.

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,

[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.

[2] Kanban boards visualise workflows. They have several columns and work items flow between them (Stellman and Greene, 2014).

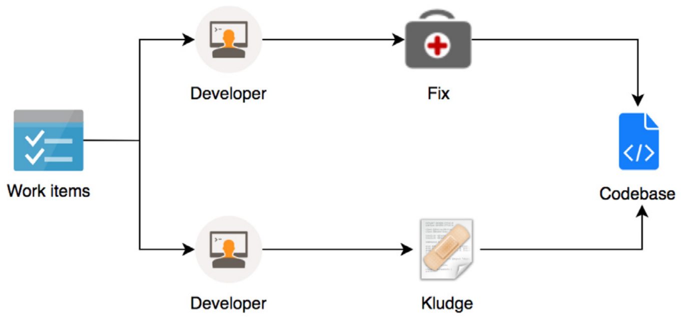

Fig. 5. Developers in our technical debt simulation: the implementation of work items can be fixes or kludges. Fixes demand more time but are less likely to require rework. Kludges are quick but are more likely than fixes to be reworked. Also, kludges negatively impact codebase health.

Table 3
Payoff matrix after the empirical game design stage in Fig. 3: it has a single Nash equilibrium where developer A and developer B adopt only the kludge-intensive heuristic strategy.

	B: fix-intensive	*B: kludge-intensive*
A: fix-intensive	$A = 9.80,\ B = 9.80$	$A = 8.34,\ B = 10.77$
A: kludge-intensive	$A = 10.77,\ B = 8.34$	$A = 9.06,\ B = 9.06$

Table 4
Payoff matrix after the empirical game improvement stage in Fig. 3: it has a single Nash equilibrium where developer A and developer B adopt only the fix-intensive heuristic strategy.

	B: fix-intensive	*B: kludge-intensive*
A: fix-intensive	$A = 9.79,\ B = 9.79$	$A = 8.40,\ B = 7.77$
A: kludge-intensive	$A = 7.77,\ B = 8.40$	$A = 6.83,\ B = 6.83$

keeping in mind that the number of strategies determines the size of the game.

Our model uses two strategies that we believe are worth 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, we have picked two extremal strategies to study how to reduce kludging. In focusing on extremal strategies, we are simply seeking to make our process resilient to unwanted behaviour, following a long tradition in mechanism design (Russell and Norvig, 2016).

Our development model is simple by design in order to capture fundamental behaviour. Our key assumption is that the aberrant behaviour that we 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; we believe customers and process performers 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 model generates more flexible models. Indeed, cutting corners is not unique to software. With just a few modifications, we could easily port our simulation model to non-software domains. Below, we show how our 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, we 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 our game-theoretic model. Once we have the payoff matrix, we obtain its equilibria. Nash equilibrium calculation is a vast and diverse area with many options (Nisan et al., 2007). In this paper, we rely on the Gambit game solver: a software tool that contains ready-to-use implementations of equilibrium calculation algorithms, facilitating quick experimentation (McKelvey et al., 2019).

An Eclipse Platform annual release has the simulation parameters $T = 29.79$ and $N = 360$ after analysing its Bugzilla data (Mi and Keung, 2016). Using our heuristic strategies and setting $D = 2, I = 1.0, T_f = 1.1 \times T, R_f = 0.9 \times R, T_k = 0.75 \times T, R_k = 1.05 \times R, Q_k = 0.05 \times T$, where $R = 0.069$ is raw rework probability in the Eclipse data. We have extracted these parameters to use real-world values. We want to emphasise that we do *not* assume that the Eclipse platform is kludge-prone. We simulated each heuristic payoff table entry for 100 releases and recorded the average F_i per developer. Table 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 our empirical game model offers an explanation: developers have an incentive to kludge instead of generating proper fixes.

Following design, we proceed with the *empirical game improvement* step. Our goal is a process whose corresponding empirical game has a single 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 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, we knew any solution should make kludges more expensive. Adopting pair programming can accomplish this but, due to its cost, we did not consider it. We posited that adding a part-time experienced code reviewer who can detect 10% of the kludges would be sufficient and also cost effective. So, we 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 we set $R_k = 25\%$ and rebuild the payoff matrix, shown in Table 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. Our 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.

GTPI is a general approach to improving software processes. Here, we applied it to the technical debt problem. In other work, we built a tool, which we christened TaskAssessor, that applies GTPI to diagnose and remove priority inflation (Gavidia-Calderon et al., 2019).

5. The challenges of applying GTPI

Adopting GTPI is challenging; data gathering, technical validation, and securing customer and performer acceptance is hard. We 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.

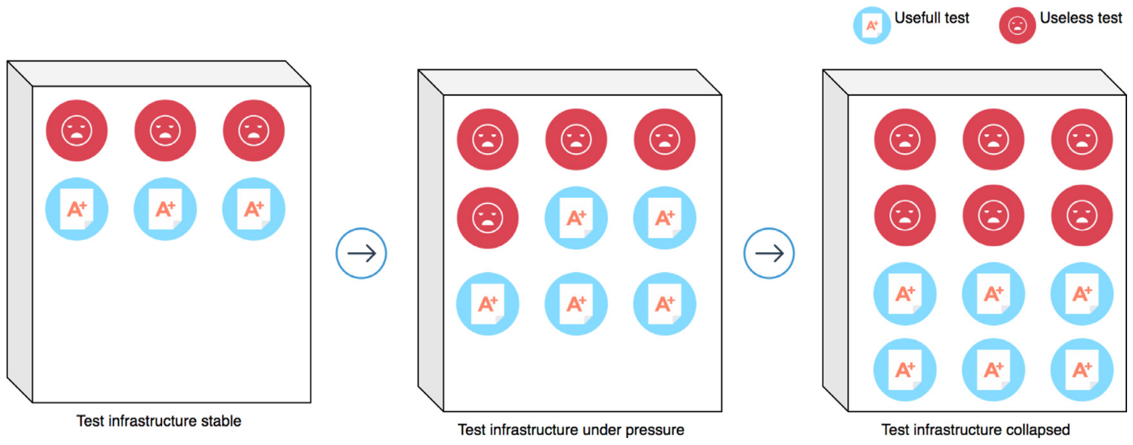
Usefull test
Useless test
Test infrastructure stable
Test infrastructure under pressure
Test infrastructure collapsed

Fig. 6. 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.

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 Fig. 6. This wastes testing resources.

We 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 prioritization 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. We believe it is more efficient, and therefore cheaper, to incentivise software engineers to keep the test suite clean and correct. GTPI 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.

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 (Yoo and Harman, 2012). Our intuition is that the test prioritization 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.

As discussed in Section 4, 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 (Münch et al., 2012; Madachy, 2007), 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 Fig. 3 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. We 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 (Banks et al., 2010, 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 (Banks et al., 2010, 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.

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.

The empirical game improvement stage requires exploring the space of potential games with a desirable NE as discussed in Section 4. 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. Also, 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, we 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.

6. Conclusion

The opportunities for applying game-theoretic tools to software process improvement are boundless. We call on the community to join us in their pursuit at https://www.researchgate.net/project/Improving-Software-Processes-via-Empirical-Game-Theory.

Acknowledgements

This research is funded by the Dynamic Adaptive Automated Software Engineering Programme Grant (EP/J017515).

References

Bacon, D.F., Bokelberg, E., Chen, Y., Kash, I.A., Parkes, D.C., Rao, M., Sridharan, M., 2010. Software economies. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research. ACM, pp. 7–12.
Banks, J., Carson, J., Nelson, B., 2010. Discrete-event System Simulation. Prentice Hall.
Bavota, G., Oliveto, R., De Lucia, A., Antoniol, G., Gueheneuc, Y.-G., 2010. Playing with refactoring: identifying extract class opportunities through game theory. In: 2010 IEEE International Conference on Software Maintenance. IEEE, pp. 1–5.
Boehm, B.W., Ross, R., 1989. Theory-w software project management principles and examples. IEEE Trans. Softw. Eng. 15 (7), 902–916.
Butcher, P., 2009. Debug It!: Find, Repair, and Prevent Bugs in Your Code.
Feijs, L., 2001. Prisoner dilemma in software testing. Comput. Sci. Rep. 1, 65–80.
Fenton, N.E., Neil, M., 2000. Software metrics: roadmap. In: Proceedings of the Conference on the Future of Software Engineering. ACM, pp. 357–370.
García-Galán, J., Trinidad, P., Ruiz-Cortés, A., 2013. Multi-user variability configuration: a game theoretic approach. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. 574–579.
Gavidia-Calderon, C., Sarro, F., Harman, M., Barr, E.T., 2019. The assessor's dilemma: Improving bug repair via empirical game theory. IEEE Trans. on Soft. Eng. doi:10.1109/tse.2019.2944608.
Gibbons, R., 1992. A Primer in Game Theory. Harvester Wheatsheaf.
Goodliffe, P., 2014. Becoming a Better Programmer: A Handbook for People Who Care About Code. O'Reilly Media.
Greasley, A., 2009. A comparison of system dynamics and discrete event simulation. In: Proceedings of the 2009 Summer Computer Simulation Conference. Society for Modeling & Simulation International, pp. 83–87.
Grechanik, M., Perry, D.E., 2004. Analyzing software development as a noncooperative game. In: IEE Seminar Digests, 29. IET, pp. 1–32.
Hasnain, E., Hall, T., Shepperd, M., 2013. Using experimental games to understand communication and trust in agile software teams. In: Cooperative and Human Aspects of Software Engineering (CHASE), 2013 6th International Workshop on. IEEE, pp. 117–120.
Hata, H., Todo, T., Onoue, S., Matsumoto, K., 2015. Characteristics of sustainable oss projects: atheoretical and empirical study. In: Proceedings of the Eighth International Workshop on Cooperative and Human Aspects of Software Engineering. IEEE Press, pp. 15–21.
Hazzan, O., Dubinsky, Y., 2005. Social perspective of software development methods: the case of the prisoner dilemma and extreme programming. In: International Conference on Extreme Programming and Agile Processes in Software Engineering. Springer, pp. 74–81.
Holyoak, K., Morrison, R., 2012. The Oxford Handbook of Thinking and Reasoning. Oxford Library of Psychology, OUP USA.
Kitagawa, N., Hata, H., Ihara, A., Kogiso, K., Matsumoto, K., 2016. Code review participation: game theoretical modeling of reviewers in gerrit datasets. In: Cooperative and Human Aspects of Software Engineering (CHASE), 2016 IEEE/ACM. IEEE, pp. 64–67.
Kukreja, N., Halfond, W.G., Tambe, M., 2013. Randomizing regression tests using game theory. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. 616–621.
Kumar, D., Baranwal, G., Raza, Z., Vidyarthi, D.P., 2018. A truthful combinatorial double auction-based marketplace mechanism for cloud computing. J. Syst. Softw. 140, 91–108.
Lagesse, B., 2006. A game-theoretical model for task assignment in project management. In: Management of Innovation and Technology, 2006 IEEE International Conference on, 2. IEEE, pp. 678–680.
Lavallée, M., Robillard, P.N., 2015. 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, pp. 677–687.
Leyton-Brown, K., Shoham, Y., 2008. Essentials of Game Theory: A Concise, Multidisciplinary Introduction. Morgan & Claypool Publishers.
Madachy, R., 2007. Software Process Dynamics. Wiley.
McConnell, S., 2006. Software Estimation: Demystifying the Black Art. Pearson Education.
McKelvey, R. D., McLennan, A. M., Turocy, T. L., 2019. Gambit: Software Tools for Game Theory, Version 15.
Mi, Q., Keung, J., 2016. 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, p. 37.
Münch, J., Armbrust, O., Kowalczyk, M., Soto, M., 2012. Software Process Definition and Management. Springer Berlin Heidelberg.
Myerson, R., 1991. Game Theory: Analysis of Conflict. Harvard University Press.
Nash, J., 1951. Non-cooperative games. Ann. Math. 286–295.
Nisan, N., Roughgarden, T., Tardos, E., Vazirani, V., 2007. Algorithmic Game Theory. Cambridge University Press.
Oza, N.V., 2006. Game theory perspectives on client: vendor relationships in offshore software outsourcing. In: Proceedings of the 2006 international workshop on Economics driven software engineering research. ACM, pp. 49–54.
Palacios-Huerta, I., 2003. Professionals play minimax. Rev. Econ. Stud. 70 (2), 395–415.
Parsons, S., Wooldridge, M., 2002. Game theory and decision theory in multi-agent systems. Autonom. Agents Multi-Agent Syst. 5 (3), 243–254.
Prison breakthrough, 2018. Accessed: 10-09-2018 https://www.economist.com/economics-brief/2016/08/20/prison-breakthrough.
Rao, M., Parkes, D.C., Seltzer, M.I., Bacon, D.F., 2015. A framework for incentivizing deep fixes. In: Proceedings of the AAAI Workshop on Incentives and Trust in E-Communites.
Russell, S., Norvig, P., 2016. Artificial Intelligence: A Modern Approach. Pearson.
Saad, W., Glass, A.L., Mandayam, N.B., Poor, H.V., 2016. Toward a consumer-centric grid: a behavioral perspective. Proc. IEEE 104 (4), 865–882.
Sandholm, T., 2010. The state of solving large incomplete-information games, and application to poker. AI Mag. 31 (4), 13–32.
Sandholm, T., 2015. Abstraction for solving large incomplete-information games.. In: AAAI, pp. 4127–4131.
Sanjab, A., Saad, W., Başar, T., 2017. Prospect theory for enhanced cyber-physical security of drone delivery systems: a network interdiction game. In: 2017 IEEE International Conference on Communications (ICC). IEEE, pp. 1–6.
Sarro, F., 2018. Predictive analytics for software testing: keynote paper. In: Proceedings of the 11th International Workshop on Search-Based Software Testing. ACM, New York, NY, USA doi:10.1145/3194718.3194730. 1–1.
Sarro, F., Petrozziello, A., Harman, M., 2016. Multi-objective software effort estimation. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 619–630. doi:10.1145/2884781.2884830.
Sazawal, V., Sudan, N., 2009. Modeling software evolution with game theory. In: International Conference on Software Process. Springer, pp. 354–365.
Stellman, A., Greene, J., 2014. Learning Agile: Understanding scrum, XP, lean, and kanban. "O'Reilly Media, Inc.".
Tadelis, S., 2013. Game Theory: An Introduction. Princeton University Press.
Talk of tech innovation is bullsh*t. shut up and get the work done says linus torvalds, 2018. Accessed: 17-09-2018 https://www.theregister.co.uk/2017/02/15/think_different_shut_up_and_work_harder_says_linus_torvalds/.
The end of software engineering and the last methodologist, 2018. Accessed: 22-01-2019 https://cacm.acm.org/blogs/blog-cacm/224352-the-end-of-software-engineering-and-the-last-methodologist/fulltext.
Walker, M., Wooders, J., 2001. Minimax play at wimbledon. Am. Econ. Rev. 91 (5), 1521–1538.
Walsh, W.E., Das, R., Tesauro, G., Kephart, J.O., 2002. Analyzing complex strategic interactions in multi-agent systems. In: AAAI-02 Workshop on Game-Theoretic and Decision-Theoretic Agents, pp. 109–118.
Wang, Y., Redmiles, D., 2016. Cheap talk, cooperation, and trust in global software engineering. Empirical Softw. Eng. 21 (6), 2233–2267.
Wellman, M.P., 2006. Methods for empirical game-theoretic analysis. In: AAAI, pp. 1552–1556.
Yilmaz, M., O'Connor, R.V., 2011. A software process engineering approach to improving software team productivity using socioeconomic mechanism design. ACM SIGSOFT Softw. Eng. Notes 36 (5), 1–5.
Yilmaz, M., O'Connor, R.V., Collins, J., 2010. Improving software development process through economic mechanism design. In: European Conference on Software Process Improvement. Springer, pp. 177–188.
Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verification Reliab. 22 (2), 67–120.

Carlos Gavidia-Calderon is a Ph. D. student in Software Engineering at University College London, under the supervision of Dr Earl T. Barr, Dr Federica Sarro and Prof. Mark Harman. His research interests are game-theoretic applications to software engineering, reinforcement learning and agent-based models.

Federica Sarro is an Associate Professor at University College London in the Department of Computer Science. Her research covers Predictive Analytics for Software Engineering (SE), Empirical SE and Search-Based SE, with a focus on software effort estimation, software sizing, software testing, and mobile app store analysis. Dr Sarro has published more than 65 papers in peer-reviewed software engineering conferences and journals, including ICSE, FSE, TSE, TOSEM, ISSTA. She has also received several international awards, including 3 best paper awards, the GECCO-HUMIES awarded for the human-competitive results achieved by her work on multi-objective effort estimation, and an ACM distinguished reviewer award received at ICSE'18. She has also served on several steering, organisation and programme committees, and editorial boards of well-renowned venues such as ICSE, FSE, IEEE TEVC, ACM TOSEM, EMSE.

Mark Harman is an engineering manager at Facebook London, where he manages a team, working on Search Based Software Engineering (SBSE). He is also a part time professor of Software Engineering in the Department of Computer Science at University College London, where he directed the CREST centre for ten years (2006-2017) and was Head of Software Systems Engineering (2012-2017). He is known for work on source code analysis, software testing, app store analysis and empirical software engineering. He was the co-founder of the field SBSE, which has grown rapidly with over 1,700 scientific publications from authors spread over more than 40 countries. SBSE research and practice is now the primary focus of his current work in both the industrial and scientific communities. Prof. Harman, together with his colleagues Dr. Jia and Dr. Mao from the SBSE testing start-up Majicke, were acquired by Facebook in February 2017 (http://www.engineering.ucl.ac.uk/news/bug-finding-majicke-finds-homefacebook/).

Earl Barr is a senior lecturer (associate professor) at the University College London. He received his Ph.D. at UC Davis in 2009. Earl's research interests include bimodal software engineering, testing and analysis, and computer security. His recent work focuses on automated software transplantation, applying game theory to software process, and using machine learning to solve programming problems. Earl dodges vans and taxis on his bike commute in London.