Analyzing Software Development as a Noncooperative Game

Mark Grechanik and Dewayne E. Perry
The Center for Advanced Research In Software Engineering (ARISE)
The University of Texas at Austin
gmark@cs.utexas.edu, perry@ece.utexas.edu

Abstract

A significant number of failures of software projects are widely attributed to poor requirements gathering and making various errors in specifications, choosing an incorrect architecture, following a wrong design and development model, and incurring significant cost in the maintenance stage. While these and other reasons are correct, they are based on an assumption that everyone involved in a software project is driven to make it successful and agrees on the goals and the methods of how to achieve that success. However, each team participant views the ultimate success of the project differently in terms of his/her personal goals. These different views may result in conflicting decisions by team participants that affect the overall success of the project. In this paper we analyze software projects as noncooperative games and show how to use the tools and techniques of game theory to uncover some hidden causes of failures of software projects and we suggest ways to fix them.

1. Issue Statement

A significant number of failures of software projects are widely attributed to poor requirements gathering and making various errors in specifications, choosing an incorrect architecture, following a wrong design and development model, and incurring significant cost in the maintenance stage [2][5][9]. While these and other reasons are correct, they are based on an assumption that everyone involved in a software project is driven to make it successful and agrees on the goals and the methods of how to achieve that success. However, each team participant views the ultimate success of the project differently in terms of his/her personal goals. These different views may result in conflicting decisions by team participants that affect the overall success of the project. In this paper we analyze software projects as noncooperative games and show how to use the tools and techniques of game theory to uncover some hidden causes of failures of software projects and we suggest ways to fix them.

Type of issue

We consider the long-term strategic issues of making individual decisions by each participant of a software project team with respect of the common goals of the project. Despite the fact that desired actions of each team participant are outlined in the job descriptions for a software project, there is a great deal of flexibility in how people can carry out these actions in real-world project settings.

Context

We assert that this strategic issue of conflict between personal decisions and overall software project goals is common in most commercial companies in the US and abroad, and that this issue has not been considered fully yet.

2. Goals of Software Engineering

Every software engineering project pursues two major goals [1][2]. The first goal is to achieve a successful software product, and the second is to conduct a successful software development and maintenance process. In software engineering economics it is assumed that every participant of a software project strives toward achieving these goals. While it is understood that goal conflicts are possible, the space of generally considered goal conflicts includes situations where conflicts arise due to external circumstances, for example, unknown requirements, poorly organized software process, or selecting a wrong tool.

Various analytical models of software economics establish relationships between a programmer's productivity and the cost of a project, help to choose an appropriate pricing strategy, or evaluate economic risks that may impact a software project in a variety of ways [1][2][3][4]. In all models, participants of software projects are considered as a group whose goals are the same as the general goals of software engineering [1]. However, each participant of a software project has three characteristics: individualism, rationality, and mutual interdependence with other participants. By individualism we mean that participants of software projects can choose whether to enter into binding agreements with others, or choose actions to achieve individual goals. Rationality means that individuals are assumed to act in their own self-interest. Finally, mutual interdependence is an important characteristic of software project participants since software development is a team activity [5]. However, it is possible that a single entrepreneur can found a company and write software, and we do not consider such cases in this paper.

When individual goals of the participants of a software project conflict with the general goals of software engineering it may result in failure. Often the blame for the failure is assigned to certain reasons that are not the enablers but rather causes of this failure. For example, an Austin-based company Arrowsmith Technologies, Inc. closed its doors in May 1997. An official reason for this closure was the deregulation of telecommunication industry that wrought havoc on company's marketing effort and led to significant financial losses. However, one of the authors (Grechanik) was employed as a senior engineer with this company and he and his co-workers witnessed the real causes to this failure. Software products sold by the company were buggy, customers complained about their poor quality, and software engineers leaving the company at a rate of up to five programmers a week contributed more to the downfall of this company than any deregulations.

A fundamental problem that we address in this paper is that of analyzing conflicts between the goals of individual team members and general software engineering goals, predicting them, and developing a strategy to eliminate these conflicts. We offer an approach that solves this fundamental problem and enables us to predict software project failures and even avoid them.

3. Proposed Approach

We consider software development activities as strategic interactions that include the constraints on the actions that the project participants can take with respect to their interests, but do not specify the actions that the participants do take. As such, a software development project is a software development game (SD game). A participant of an SD game is a player and is the basic entity in all game theoretic models [7].

We offer the following caveat: we define the general characteristics of the players and ignore all the possible exceptions.

Rationality

Each player of an SD game is a rational decision maker. This statement may be read with certain degree of disbelief since it is a known fact that various failures of software projects have been linked to irrational technical and managerial choices. What we mean by players of SD games being rational is that they make choices in the best of their own self-interests.

What constitutes rational behavior of SD game players? There is a set of *actions* A from which a player makes a choice, a set C of possible *consequences* of these actions, and a consequence function $g : A \rightarrow C$ that associates a consequence with each action. A player chooses appropriate actions using a preference relation $\prec$ that is also called a *payoff*.

An SD game is a strategic game because in the process of interactive decision-making a player chooses his plan of actions once and for all, and these choices are made simultaneously with other players. Each player has his/her own strategy that is a set of actions. All players choose these actions simultaneously from a nonempty set A.

Players

We divide all SD game players into three categories: management, customers, and developers. We designate them with uppercase letters M, C, and D respectively. Management includes all managers and company executives who are not involved in software development activities directly, however, they are responsible for the ultimate success of the software product that is an outcome of the SD game. Customers are individuals and companies that buy a software product resulting from the SD game. Finally, software engineers or developers include project participants who gather requirements, write specifications, architect and design, code, test, and deploy the system. In short, D stands for every player of an SD game who is directly involved in creating and evolving a software product.

Strategies

We are ready to define strategies for each category of SD players. M's strategy comprises three goals: decrease cost of the software product, increase its price, and ship the product faster. Ideally, when the cost of software product goes to zero and its price skyrockets, M is happy. In the real world the difference between the sales (price of a unit of the product multiplied by the number of units sold to C) and the cost to produce and maintain this product is the margin that defines the profit of a company, and how good an SD game is. Shipping a product faster enables M to sell more units of the product in a shorter interval of time thereby increasing the margin and the company's profit.

C's strategy revolves around three goals: decrease the price of the product, increase a number of features that it offers, and purchase a higher quality product for a smaller price.

D's strategy is less than clear. While M and C have well-defined financial goals based on the successful outcome of the SD game, D's goals are to satisfy job requirements, increase personal marketability, and increase M and C's dependence on D. The first goal of satisfying job requirements is easy to understand. If D cannot do what s/he is hired to do then M fires D. At the same time D strives to improve D's personal marketability in order to stay competitive and be able to obtain a better job. Finally, D want to secure the existing job, and one sure way to achieve it is to make D indispensable for M and C.

Payoff

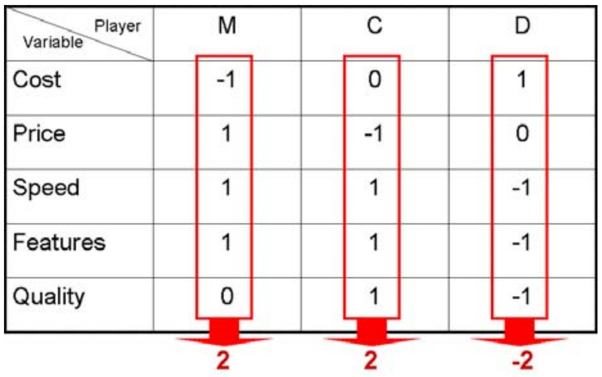
Now that we know strategies and goals of each player of an SD game we define a preference relation (payoff) for each player. For M, if the product is successful the payoff is promotion and a sizable bonus. For example, IBM Corp. allocates a hefty bonus if a project results in a successful product and all milestones are met in time and within budget. A fifth-level manager leaves a sizable chunk of the bonus to him/herself and passes the rest to the fourth level managers overseeing this project. The process repeats with remaining part of the bonus distributed among first-level management and nothing for actual project participants (i.e. D) [8].

C's payoff can be described as satisfaction of C's business goals using a purchased software product. The ultimate goal of any purchased product for C is to make more money or save money using this product. Some products are geared toward increasing worker's productivity that also result in making more money for C.

D's payoff is very prosaic. Besides keeping the job and a slim chance for a limited promotion from a junior software engineer to a software engineer and ultimately to a position of a senior or principal software engineer, there is no payoff that links D to the successful outcome of a software project.

A payoff table is shown in Table 1. Three columns specify SD game players, i.e. M, C, and D. The variables that affect the successful outcome of an SD game are shown in the leftmost column. Variable *cost* defines the total resources used by the project that can be measured in dollars. Variable *price* defines a price of a unit of a software product resulting from a given SD game, in dollars. Variable *speed* measures how fast a product can be shipped to customers. Variable *features* indicates the number of functional units that a product delivers to customers.

The last variable is *quality* of the resulting product. We measure quality as a level of C's satisfaction with a purchased software product. We talk about three levels of quality: C is extremely satisfied with the quality of a product, C is satisfied enough to use a product, and C is dissatisfied with a product.



Variable \ Player	M	C	D
Cost	-1	0	1
Price	1	-1	0
Speed	1	1	-1
Features	1	1	-1
Quality	0	1	-1
	2	2	-2

Table 1. Preference relation (payoffs) for players of an SD game.

We use three values to associate each variable in Table 1 with SD game players. Value `-1` means that a player is not interested in increasing the value of that variable, value `0` means that a player is neutral on that variable, and value `1` mean that a player is interested in increasing the value of that variable.

Analysis of Payoffs

Let us consider payoff values that we assign in Table 1 for each player with regard to each game variable. M is concerned with the increased of the cost of a project since it has a negative impact on M's goals. Therefore M's strategy is to minimize the cost and we reflect the negative payoff for M when the cost increases. Naturally, M wants to increase the price, speed, and the number of features of a software product, and it is reflected in Table 1 by assigning values `1` to these variables for M. Finally, the effect of quality on M is neutral; M wants sufficient quality for C, however, M does not want to spend significant resources on improving the quality beyond what practically is required for C to be satisfied. Because of that we assign value `0` for quality variable for M.

On the other hand, C is not concerned with the cost of manufacturing of a software product. If M's cost is greater than its revenue from a product (which we observed during the dot com bubble), C is still perfectly happy to purchase this product. Because of this we assign value `0` to the cost variable for C. However, C is concerned about the price of a product and wants to pay less while getting a software product faster, with more features, and of the highest quality. These objectives are reflected by assigning negative value `-1` to price variable and positive values `1` to speed, features, and quality variables.

When considering D's payoffs it becomes clear that they are different from payoffs of M and C. D is interested in the increase of the cost of a project. Why? Because the increase in the cost means that more resources are allocated to D's needs. For example, money is spent on training D, buying better equipment and software tools, and salary increases drive up the cost of the project. However, it satisfies D's goals of becoming better marketable. The price of a product has little effect on D since it is not D who pays this price. The speed, features, and quality variables have negative impact on D since improving these variables requires more work from D.

When we sum up values in columns M, C, and D we obtain values `2` for M and C and `-2` for D. Clearly this result indicates that while M and D end up with positive payoffs, the payoff for D in the same game is negative. This is a strong indicator that the general goals of software engineering conflict with a personal strategy selected by D to maximize its own payoff.

SD is a Noncooperative Game

Game theory defines a game as noncooperative if two following conditions hold:

- Agreements to share payoffs, even if it were practical, are virtually nonexistent, and
- All players are out for themselves.

Let us see why these conditions hold for a vast majority of SD games. Player M does not share business plans with D and often views D as a disposable resource. Moreover, M does not explain business objectives to D and often makes decisions that require significant investments from D. For example, we observed that in many companies D is asked to deliver a certain subsystem ahead of time at the expense of sacrificing D's personal time. D takes such a request seriously and works hard producing the subsystem within the budget and ahead of schedule only to find out that M changed its priorities and this subsystem is not needed any more. Even if it still needed there is a little reward allocated to D.

A reasonable question to ask whether there is a strategy for D such that D can share payoffs and cooperate with M. Our hypothesis is that this is only possible when D becomes M, i.e. D changes his/her career, stops being a software engineer, and becomes a manager. The current settings of noncooperative SD games leaves no choice for Ds but to look for ways to leave software development and either become managers or free-lancing software consultants.

Nash Equilibrium for SD Games

It is a known result in the game theory that trying to maximize payoff to one or two players may result in the loss to all game participants.

In SD games the same game is played many times but with different players. Each player knows the unspoken rules for maximizing payoff for M and C from the prior experience, and attempts a strategy that would maximize his/her payoff often at the expense of the success of the SD game and consequently of the software product. M knows that D cannot be relied upon especially if D knows how M maximizes its payoff. This information can explain a situation when managers try to hire mostly fresh college graduates for large-scale projects. As irrational as it seems this action is perfectly rational considering the complete absence of desire from M's side to share the payoff. M's hopes lie in the belief that programming is not hard and young developers are inexperienced in the structure of SD games and have high motivations to prove themselves at the expense of their personal benefits.

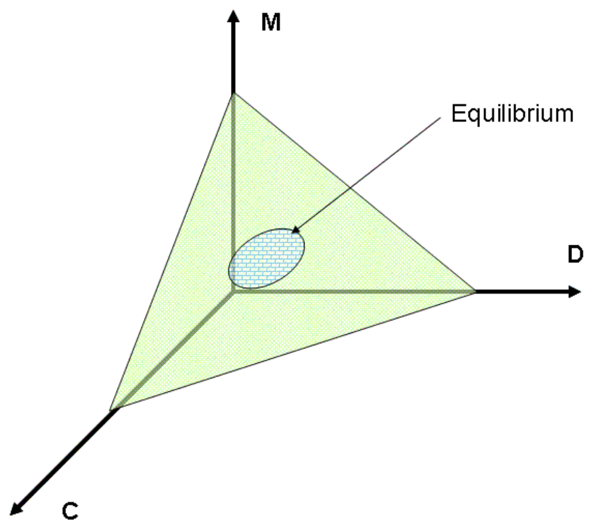


Figure 1. A model of Nash equilibrium for an SD game.

Reaching Winning Equilibrium

Thus, in every SD game we have the mixture of collective rationality and individual inspirations from the three players. The question is how do we reach a winning equilibrium?

One of the solutions from the game theory to reach a winning equilibrium is to form a new coalition among the players. In case of SD games consider forming coalitions between M and D. In a way such a coalition is attempted in software startups where D and M receive shares of a company. A problem is that an imputation, or payoff, for this coalition should be Pareto optimal -- that is, all players should simultaneously do better. It is illustrated in Figure 1 where axes show payoffs for the three players. A small area in the payoff space is called the equilibrium. For example, if M and D get equal number of company's shares, then a payoff may be optimal. As a general rule M receives more shares than D. However, to reach the winning equilibrium, M cannot get big intermediate bonuses while D is promised a payoff only when company's stock goes up. A large difference in the company shares that M and D receive in no way helps to make an imputation for SD games Pareto optimal.

In order to reach a winning equilibrium in SD games a payoff should be structured in a way that all players win when a project's outcome is a successful product. In reality it is a difficult problem. Most financial systems especially in large software organizations are not set up to structure optimal payoffs. As a proverb says, a war is won by generals and lost by soldiers. The analogy between a war and an SD game is that when it is successful all the glory goes to M, and when it is unsuccessful all the blame goes to D. This mentality dictates unequal payoff distributions with larger chucks going to M.

Software Complexity and SD Games

An interesting side effect of noncooperative SD games with nonoptimal payoffs is a significant accidental complexity in the resulting software products. We assert this claim based on our observations of multiple projects that we have participated in as developers and software consultants. In a number of cases we observed the following situation. A project was successfully moving to its completion and a product was built in time and within the budget. Management was trying to maximize its payoff and wanted to reduce the cost of the project by laying off a number of developers. In an interesting twist managers first let go employees whose systems were less buggy and required less maintenance. Thus, those who cared about the success of a project were punished while those who created imperfect software stayed because the management needed someone to maintain it. Since it was troublesome to fire the creators of buggy software and hire someone instead of them, train these new hires, and pay them to maintain the software, managers preferred to keep the poor programmers.

Recall that the same SD game is played by different players who already know about the rules and unfair payoffs. A natural strategy for D is to maximize the dependence of M on D. But what are the ways to achieve this dependence? One way is prompted by M's actions of laying off those who write less complicated and higher quality software. D realizes that it is in his/her interest to write complicated software that would require M to incur a high switching cost when trying to replace a D. This perspective may explain why Ds often argue to use software and tools that add complexity to their daily tasks and negatively impact software projects. For example, programmers who insist on using C++ when a better domain-specific language is available may have this as their rationale.

Previous work

We base our work on the foundational goals of software engineering defined in [1] and we use the game theory to explain our observations [2]. We are not aware of any prior results that use game theory to evaluate the effect of goal conflicts on the success of software engineering projects.

Acknowledgment
We would like to warmly thank Don Batory for discussions and comments that significantly improved this paper.

7. References

[1] B. Boehm, *Software Engineering Economics*, Prentice Hall, Upper Saddle River, NJ, 1984.

[2] B. Boehm, "Software Risk Management: Principles and Practices", IEEE Software, Jan. 1991, p.32-41.

[3] E. DeGarmo, W. Sullivan, and J. Bontadelli, *Engineering Economy*, Prentice-Hall, Upper Saddle River, NJ, 1993.

[4] E. Grant, W. Ireson, and R. Leavenworth, *Principles of Engineering Economy*, Wiley, New York, 1990.

[5] F. Brooks, *The Mythical Man-Month*, Addison-Wesley, 2nd edition, August 1995.

[6] L. Levy, *Taming the Tiger - Software Engineering and Software Economics*, Springer-Verlag, New York, 1987.

[7] M. Osborne and A. Rubinstein, *A Course in Game Theory*, MIT Press, August 1994.

[8] Private conversations with IBM, Schlumberger, and KLA-Tencor employees.

[9] S. Flowers, Software Failure: Management Failure: Amazing Stories and Cautionary Tales, John Wiley & Sons, December 1996.