A Perspective on Blending Programming Environments and Games: Beyond Points, Badges, and Leaderboards*

Titus Barik, Emerson Murphy-Hill
North Carolina State University, USA
tbarik@ncsu.edu, emerson@csc.ncsu.edu

Thomas Zimmermann
Microsoft Research, USA
tzimmer@microsoft.com

***Abstract*—Programming environments and game environments share many of the same characteristics, such as requiring their users to understand strategies and solve difficult challenges. Yet, only game designers have been able to capitalize on methods that are consistently able to keep their users engaged. Consequently, software engineers have been increasingly interested in understanding how these game experiences can be transferred to programming experiences, a process termed *gamification*.**

In this position paper, we assert through formal argument that gamification as applied today is predominately *narrow*, placing emphasis on adopting the reward aspects of game mechanics at the expense of other important game elements, such as framing. We argue that more authentic game experiences are possible when programming environments are re-conceptualized and assessed as a holistic, serious games. This *broad* gamification enables us to more effectively apply and leverage the breadth of game elements to the construction and understanding of programming environments.

I. Introduction

Blending programming environments with games is a fascinating, human-centered intersection for exploration by software engineering researchers, because these two contexts appear to share many of the same characteristics: they both have complex rules and strategies to learn, require long hours and deliberate practice to build mastery, and produce boundless obstacles and challenges that must be overcome. Yet we immediately recognize that though programming environments and games are ultimately just software, the *experience* that we get from software as programming environments when we're programmers and software as games when we're players just feels very different. In particular, games are somehow able to offer us an experience in which we are focused, motivated, and productive in a way that is unmatched by the environments that we use to work [1].

Other software engineering researchers have also arrived at the same conclusion.[1] As a result, they too have been increasingly interested in transferring game experiences to programmers, so that programmers can more effectively and efficiently create software artifacts [2]. One mechanism by which researchers have investigated this process is through *gamification* [3], that is essentially: "the use of game design elements in non-game contexts" [3].

In this perspective paper, we argue that the application of gamification has thus far been focused *narrowly* on the game element of incorporating reward-based *mechanics*, such as points, leaderboards, and badges. In doing so, software engineering researchers elide other essential game elements, such as purpose, aesthetic, and framing [4]. We demonstrate that *broad* gamification, or gamification that considers a holistic view of game elements, can provide more substantive game experiences in programming environments.

As a method of inquiry, this paper uses the Toulmin model of argument to present and evaluate its claim [5]. It is a formal method of argument, though not in the mathematical proof sense. Rather, the model's formality lies in the *components* of the argument.[2] Through this argument, we offer the following insights:[3]

- We *claim* that programming environments can be conceptualized as *serious* games, a class of games that do not have entertainment as their primary purpose (Section II).
- Through the use of Mitgutsch and Alvarado's Serious Game Design Assessment (SGDA) framework, we offer a *warrant* of *broad* gamification that enables us to assess programming environments as games (Section III). Consequently, we are better able to express the ways in which programming environments today fall short of meeting the full potential of game experiences.
- Through *facts*, we argue that existing implementations of gamification are *narrow*, largely eliding the body of literature from psychology and game designer, and through *backing*, demonstrate that these implementations focus on reward-based game mechanisms as a means

* **Note to reviewers:** This paper is submitted under the topic "Human aspects and psychology of software development and language design," and supports its claims through "formal argument."

[1] In this paper, we use the term *software engineering researchers*, or sometimes just researchers, in a broad sense to refer to people who have an interest in researching or building tools that help other programmers. We use the term *programmers* to the refer to people who are intended to be the users of these tools.

[2] The six components of this model are described by Besnard and Hunter [6]: 1) *claim*, or the position being argued for; 2) *fact*, or items of information specific to a context; 3) *warrant*, or relating facts to claims; 4) *backing*, or justification for a warrant; 5) *rebuttal*, or exceptions to the claim; 6) *qualification*, or specific limitations to the claim.

[3] *Rebuttals* are addressed as needed, and are interspersed through the paper.

to extrinsically motivate programmers to perform some action (Section IV).

- As further *backing*, we apply the principles learned from broad gamification to two programming environment examples (Section V). The first example demonstrates that novel insights can be gained in existing programming tools by reconceptualizing them as games (Section V-A). The second example demonstrates how the use of game principles can be used to guide the design of new programming environments (Section V-B).
- Finally, we offer implications and *qualify* our arguments (Section VI). For example, we suggest that our current reward-based approaches to gamification may be unintentionally reducing the diversity and pool of programmers who might otherwise be inclined to use our programming environments.

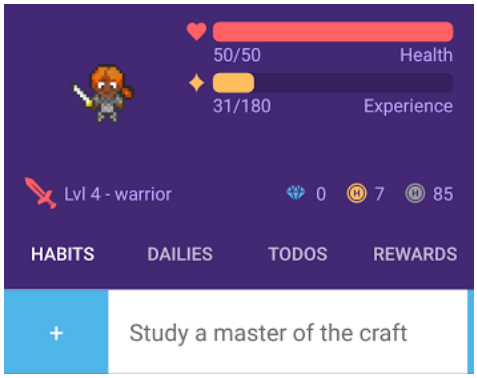


Fig. 1. *Habitica* is serious game designed to build good habits. It gamifies a typical time-tracking application be reconceptualizing it in the form of a role-playing game.

II. Programming Environments as Serious Games

If programming environments can be conceptualized as games, what type of game should they be? On one hand, when we think of games, the notion of *fun* is typically what comes to our mind. In nearly every interview with software engineers who had experience with both game and non-game software, programmers pointed to "fun" as one of the core differences between requirements in games and other types of software [7]. Lewis and Whitehead, in describing research areas between software engineering and games, likewise noted, "the one unique aspect of games, that seems to separate it from traditional software, is the requirement for games to be *fun*" [8].

On the other hand, programming environments are also *serious*, sophisticated instruments for the construction of software artifacts. Thus, opponents might reasonably claim that the very notion of "fun" is antithetical to productivity in programming environments.

Our approach to overcoming this potential impasse is to qualify software engineering environments as a form of *serious games*. In its essential form, serious games are games that "do not have entertainment, enjoyment, or fun as their *primary* purpose" [9]. As noted by Djaouti and colleagues, this type of blending of the fun elements and serious elements in an environment allow us to use the artistic medium of traditional entertainment games to educate, inform, train, and influence [10].

We can apply blending to programming environments by shifting our perspectives on these environments and framing them as a domain-specific serious games. Doing so provides two benefits. First, this results is less of a gap to bridge between programming environments and game environments. Second, we can still leverage applicable game design elements from both games and serious games and apply them in the context of programming environments.

III. Serious Game Design Framework Assessment

If programming environments can be conceptualized as serious games, then it suggests that these environments can be evaluated through existing game design frameworks. Such evaluations may be retrospective, enabling us to assess existing tools through the lens of a game designer. Such evaluations may also be generative, as design guidelines for new programming environments.

In this paper, we utilize Mitgutsch and Alvarado's Serious Game Design Framework Assessment (SGDA) framework [4]. In contrast with with prior frameworks that primarily focus on educational games [11], [12], SGDA incorporates not only general game design elements, such as *mechanics*, but also elements unique to serious games, such as *purpose*. To illustrate the framework concretely, we describe the framework elements through *Habitica*[4] (Figure 1), an online time-tracking application. As a serious game, *Habitica* positions itself in the role-playing game (RPG) genre, in which players gain "a sense of growing an ordinary person into a superhero with amazing powers" [13]. The SGDA framework has six elements, and the game elements are intended to be used cohesively to provide a holistic game experience:

Purpose The purpose of the game determines the intended impact on the players, and influences the consideration of other elements within framework. In *Habitica*, the purpose of the game is to motivate the player to form positive habits and complete goals, such as flossing regularly or submitting a VL/HCC paper by the deadline. The remaining elements of the framework serve to support the purpose of the game.

Content/Information This element refers to the data provided to the player within the game. In *Habitica*, this content includes player statistics, such as their level, health, and experience. The content also includes their daily habits and goals, such as "Study a master of the craft."

Mechanics The mechanics of a game include not only the methods by which the players in the game interact with the environment, but also the rules of the game that define the "possibility space" [4]. As such, the mechanics also include any reward mechanisms, such as points or badges.

[4]https://habitica.com/

In *Habitica*, the mechanics of the game involve adding tasks to the game, setting deadlines, and marking which tasks are completed. The reward mechanics include in-game items, "boss fights", and "quests."

Fiction/Narrative The element of narrative introduces the context of fantasy, which includes the setting, story, scenario, or problem or the game. In *Habitica*, the fantasy element is provided within a medieval setting in which mundane, real-life goals transform into heroic in-game events, such as a vanquishing villainous dragons.

Aesthetics/Graphics Niedenthal provides a taxonomic definition specific to games by segmenting aesthetics into three components [14]: 1) "the sensory phenomena that the player encounters in the game," such as visual or aural stimulus, 2) "those aspects that are shared with other art forms," such as visual styles or use of colors, and 3) "an expression of the game experienced emotionally," such as through fear, pleasure, or frustration. In *Habitica*, the game aesthetic uses retro, sprite-based graphics that typified the graphical style of classical 8-bit Nintendo RPGs, such as *The Legend of Zelda* [15].

Framing Framing associates the other elements of the game framework in terms appropriate to the target audience, that is, it captures the *player literacy*. A game that has poor play literacy would make incorrect assumptions about the type of player who plays the game, leading the player to be confused or frustrated. *Habitica* progressively discloses complexity: as "starter quests" are completed, and as the player obtains additional experience, the game unlocks additional quests.

Through the SGDA framework, we argue that *Habitica* uses these game elements appropriately and cohesively and presents a coherent, cohesive, serious game experience.

IV. The Case Against Narrow Gamification in Software Engineering

Having presented the SGDA framework (Section III), we postulate that gamification techniques as applied today are primarily of the narrow form, and argue as such by examining the current literature on blending programming environments and games. We then consider the limitations and potentially harmful implications of narrow, reward-based gamification, using the underlying cognitive psychology and perspectives from game designers.

A. Existing Approaches to Implementing Gamification

We summarize existing approaches and their purpose for papers in which the authors explicitly characterized their work as implementing gamification for programming environments. For this task, we began with the literature review conducted by Hamari and colleagues in 2014 [2]. We then conducted a literature search of the past two years of ICSE, FSE, CHI, and VL/HCC to identify more recent papers on gamification since Hamari and colleagues' original publication. We summarize these papers in Table I, and classify their support for each of the elements in the SGDA framework.

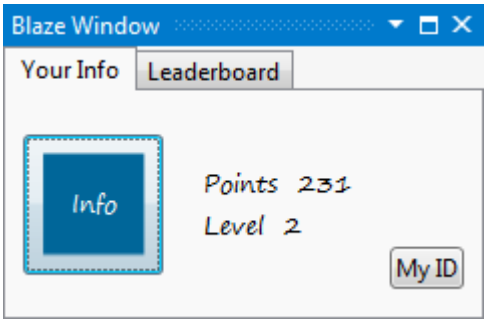


Fig. 2. The Blaze gamification system [16] uses points, levels, and leaderboards, which are characteristic of narrow gamification.

TABLE I
Overview of Existing Software Development Environments under SGDA Framework

Tool	Framework Element					
	Pur	Aes	Fic	Mec	Fra	Con
Blaze [16]	●	○	○	●	○	◐
Teamfeed [17]	●	○	○	●	◐	●
Beehive [18]	●	◐	◐	●	○	●
OO Practices [19]	●	○	○	●	○	◐
Stack Overflow [20]	●	●	◐	●	◐	◐
VS Achievements	○	●	◐	●	●	●
Free Hugs [21]	●	●	◐	●	●	●

[1] Circles indicate: ● — Element supported, ◐ — Element partially supported, ○ — Element not supported. Framework elements: Purpose (Pur), Aesthetic (Aes), Fiction (Fic), Mechanics (Mec), Framing (Fra), Content (Con).

Reflecting on our own foray in gamification, we introduced a system system called *Blaze* (see Figure 2) to motivate programmers to use structured navigation tools when exploring source code [16]. Thus, *Blaze* has a clear purpose. As a game mechanic, *Blaze* uses rewards such as points and leaderboards to notify the programmer about their use of these tools. However, the system has only partial support for content. Although the game notifies the programmer about points and levels, the game lacks a clear association between how tool usage maps to obtaining these rewards. *Blaze* has no discernible, gameful aesthetic of its own; it merely uses the default components that come with Visual Studio. *Blaze* also has no fictional or fantasy element, and does not provide any framing. *Blaze* is emblematic of narrow gamification: it relies primarily on a single dimension of the SGDA framework, reward-based game mechanics.

We conducted a similar analysis on the remaining gamification papers, but for purposes of space, do not elaborate on their details element-wise. Singer and Schneider motivated students to make more frequent commits to version control through a newsfeed (as used by social network games) of commits and a leaderboard [17]. Farzan and colleagues implemented a point-based reward system in their internal social networking system at IBM to motivate users to contribute content [18]. Dubois and Tamburrelli motivated students to use object-oriented

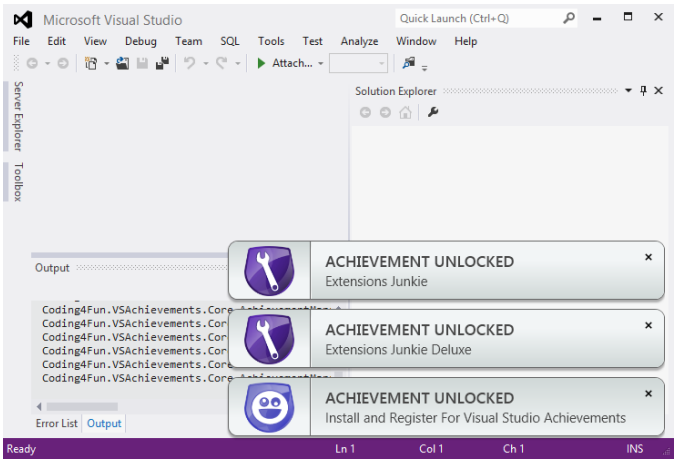


Fig. 3. The Visual Studio Achievements extension introduces playful elements within the programmer's environment. We unlocked the above achievements during the process of creating this figure.

best practices that are expressible as metrics, such as test coverage; these metrics are converted to scores and presented as leaderboards [19].

When we expanded our search outside of programming environments, a literature review of 24 empirical studies on gamification revealed the same trend — leaderboards (42%), points (38%), and achievements and badges (38%) dominate the gamification landscape, across many disciplines [2]. At least, software engineering researchers may have some small comfort in knowing that we aren't alone in our underuse of broad gamification.

With that said, we did find two exceptions to this trend that we think are worth highlighting as examples of broad gamification that use multiple game elements. First, consider the Visual Studio Achievements extension[5], shown in Figure 3. By deliberate design, the extension offers a *ludic*, or playful, space for the programmer — although not a full-blown fantasy, these playful elements do add to the fictional element by compelling players to perform unconventional actions. To illustrate, the "Lonely" achievement is earned when the programmer is coding on a Friday or Saturday night, and the "Potty Mouth" achievement is unlocked when the programmer uses five different curse words in a file. Visual Studio Achievements provide adequate support for framing by unlocking certain achievements when the programmer obtains the simpler, pre-requisite achievements. Visual Studio Achievements is intentionally purposeless, 'add[ing] some humor to the levity of coding" [22]. As such, it is more like a game than a serious game.

As a second exception, consider the micro-gamification environment *Free Hugs* [23], in which the authors sketch the idea of an environment where programmers creates fictitious *alter egos* within a *development empire*. The alter egos in this empire evolve as programmers use best practices in the IDE. The environment has the potential to provide a richer game experience, because it introduces additional game elements, such as fiction, into its design.

B. Undermining Motivation: Challenges of Reward-based, Narrow Gamification

So what's the problem with narrow, reward-based game mechanics? Cognitive psychologists Ryan and colleagues have identified and explained the underlying motivational pull of video games through *self-determination theory*, a theory of *intrinsic motivation* "based on inherent satisfactions derived from action" and one in which "most players do not derive extra-game rewards or approval" [24].

In contrast to intrinsic motivation, *extrinsic motivation* is "done in order to attain a separable outcome," for example, to comply with an external control or to pursue an external reward [25]. Research in psychology consistently demonstrates that external rewards undermine intrinsic motivation [26], [27], [28]. For example, Deci and colleagues conducted an extensive meta-analysis that showed that tangible rewards, such as gold stars, honor roles, and other reward-focused incentive systems have a substantial undermining effect on the intrinsic motivation to learn [26].

This finding has been since been replicated in a variety of software engineering contexts. Farzan and colleagues, in their enterprise social networking system at IBM, found that users who saw points contributed more content, though not indefinitely, and that user contributions dropped precipitously immediately after attaining certain levels [18]. Mamykina and colleagues observed a similar type of behavior from some users of the question and answer site StackOverflow[6]. Initially drawn by the extrinsic motivation provided by points and badges, these "shooting stars," had a single, short period of high activity followed by low activity after obtaining a reward. [29] And out of their four user classifications, only the intrinsically motivated group, *community activists*, converted to highly active, long-term contributors. In addition, the reputation points system in StackOverflow caused some users to provide faster but shorter answers. This allowed them to gain reputation points more quickly, but at the expense of readers, since detailed answers are more informative [29]. External rewards can change behavior, but sometimes in unexpected ways.

Game experts appear to have an intuitive understanding of motivation and have found effective ways to incorporate motivation in their games. Bissell, a games journalist, notes that games don't seem to suffer from motivation problems, despite having tasks that resemble the tedium and repetition that is sometimes present in programming activities. Other game designers, such as Bogost, have also weighed in on the discussions surrounding gamification. Bogost argues that gamification as being applied is essentially a formulaic "just add points" approach [30]. Robertson explains, "gamification is in fact the process of taking the thing that is least essential to games and representing it as the core of the experience" [31]. She continues, "gamification, as it stands, should actually be

[5]http://channel9.msdn.com/achievements/visualstudio

[6]http://stackoverflow.com/

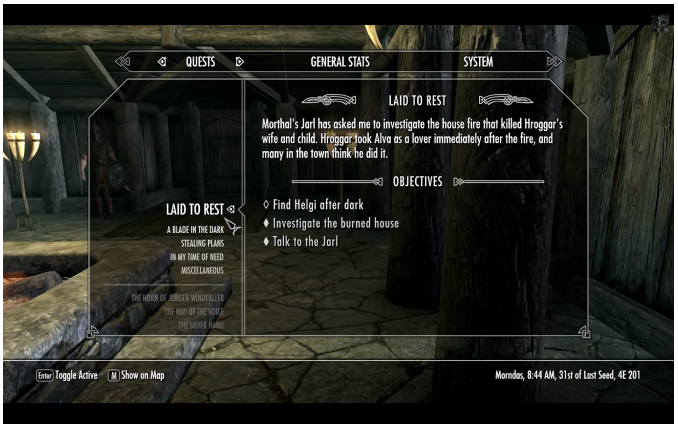


Fig. 4. *The Elder Scrolls V: Skyrim* provides quests for the player to complete. The quests are broken down into manageable subgoals, and the completion of these subgoals is explicitly marked as visual overlays on the interface.

called pointsification" [31]. Jadoga adds, "play, even within the parameters of a design game and its constraints . . . is a space of potential, one that is so often undone by the ludic sterility of [points-based] gamification" [32]. Hamari goes even further by suggesting that perhaps achievements should not be integrated at all, but instead, that "achievement systems should be viewed as games of their own" [33].

Yet, from our perspective, it appears programming environments predominantly include the narrow, reward-based *mechanics* of games, rather than applying broader game design elements to support of the game's purpose. Software engineering researchers are well-intentioned and have a significant interest in enhancing motivation — but leveraging the *right* type of motivation is important.

V. Rethinking Programming Tools as Games

Thus far, we've examined the SGDA framework and how programming environments can be evaluated as serious games. We critically examined how the existing uses of gamification in software engineering narrowly focus on reward-based mechanics, and how doing so limits game experiences.

In this section, we present two examples of how SGDA, and thinking of programming environments as serious games, can surmount the limitations of narrow gamification. The first example demonstrates that novel insights can be gained *retrospectively* when evaluating existing programming environments as serious games. The second example demonstrates that SGDA can be used *generatively*, and without relying on explicit reward-based mechanics.

A. Tests as Quests

Test-first development is an agile methodology through which programmers first establish a set of *test cases* to computationally describe a small increment of the specification before coding the actual implementation [34]. Generally, tests are added gradually and co-evolve with the implementation.

We think that the process of test-first development, when blended with games, can be cast narratively as *quest games*.

```
public void Debit(double amount)
{
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    m_balance += amount;
}
```

(a) Programmer finds a defect.

```
public void Debit(double amount)
{
    if (amount < 0)
    {
        throw new ArgumentOutOfRangeException("amount");
    }

    m_balance -= amount;
}
```

(b) Programmer resolves the defect.

Fig. 5. In deep gamification, the unit testing tool *NCrunch* can be examined as a quest game like Skyrim. NCrunch adds markers to the margin to indicate the coverage for each line: not covered by tests (black circle), covered by tests and pass (green circle), covered by tests and fail (red circle).

Aarseth defines a *quest game* as "a game with a concrete and attainable goal, which supersedes performance or the accumulation of points. Such goals can be nested (hierarchic), concurrent, or serial, or a combination of the above" [35]. Aarseth's framework offers three basic quest types: 1) *place-oriented*, where the player must go from one place in the game world to another, 2) *time-oriented*, where one must satisfy a requirement during or within a certain time interval, 3) and *objective-oriented*, where the task is to achieve a concrete result. Quest types typically do not exist in isolation, but are combined in various ways to create interesting challenges for players.

Let us consider the game *The Elder Scrolls V: Skyrim* (Skyrim) [36], an action role-playing game consisting of main quests (typically required to complete the game), and side quests (which are optional challenges for the player). Figure 4 shows one such side quest, in which the player is offered the main goal of investigating a house fire. They can accomplish the main goal by completing the three subgoals. Note that, unlike unit tests, the full specification is often intentionally elided for purposes of increasing dramatic tension in game. In addition, it is sometimes possible to complete the main goal without completing all of the subgoals. For example, continuing our Skyrim quest explanation, the player might directly choose to confront Hrogger without investigating the burned house. Of course, programmers can do this too[7], but some organizations require that all tests pass before allowing the code to be submitted.

Although not explicitly designed using game elements, and lacking the game element of framing, we argue that *NCrunch*[8], an automated testing tool for Visual Studio, resembles a quest

[7]That is, they can skip subgoals, not necessarily confront people who set houses on fire.

[8]http://www.ncrunch.net/

Fig. 6. *Civilization V* is a turn-based city-building strategy game. In the game, players can elect to give worker units instructions at every turn of the game. As an alternative, players can request that one or more worker units perform automated improvements, leaving the computer to pick reasonable defaults through its built-in recommendation system.

game — the purpose of which is to increase test coverage of the project. As shown in Figure 5, its salient feature is the aesthetic of dynamically updating visual overlays in the margin, called *markers*, that indicate lines of the source code are: 1) not covered by any tests ● (black circle), 2) covered by tests and pass ● (green circle), and, 3) covered by tests and fail ● (red circle). The markers act as affordances to let the programmer know what needs or does not need to be fixed. As in quests, a test suite represents the main goal, and the individual markers represent subgoals. Some of these subgoals may be optional. As a game mechanic, the programmer has the agency to explicitly choose to ignore black circles, because they look uninteresting compared with the more important failures that they see in Figure 5a. By examining the failed lines against the test specification (not shown in figure), the programmer sees that he is accidentally adding to the `m_balance`, instead of subtracting. Upon modifying the code, the system confirms the programmer's agency through immediate feedback, or content, that updates the markers to green (see Figure 5b).[9]

Cockburn argues that many practices in agile methodologies are better explained as games than as engineering [37]; we think test-first development is a good starting point for exploring this space. Erdogamus and colleagues already advocate a test-first approach because they provide the programmer with instant feedback, which informs the programmer of the correctness and implementation of functionality and task-orientation [34]. But more importantly, they show that test-first development helps programmers maintain focus and provides them with an indicator of steady, measurable progress [34]. By thinking as game designers, we are afforded a vocabulary that explains why the interface provides these benefits to the programmer.

B. Progressive Complexity

Now that we've seen how thinking in terms of serious games can help us to retrospectively understand existing environments, consider how we can *constructively* use game elements to enhance the game experience in a programming environment.

Integrated development environments have a non-trivial amount of features and customizable settings, yet programmers need only a particular subset of these features at any given time. Thus, there is a framing mismatch in the available options and the requirements of the programmer. Game designers would characterize this as an imbalance between material affordances, the opportunities for action that are presented, and the formal affordances, the motivation to pursue one particular action out of all actions that are offered [38]. We propose that programming tools could benefit from techniques used in games to reduce the cognitive burden presented by the complexity of the interface.

As an example of a game that manages progressive complexity in an effective manner, consider the game *Sid Meier's Civilization V* (Civ V) [39], a turn-based city-building strategy game, shown in Figure 6. In the game, the player must lead a civilization from its inception, achieving one of a number of different victory types. Each of these victory types requires the player to commit to a subset of available game actions, among them scientific research, world exploration, diplomacy, and military conquest.

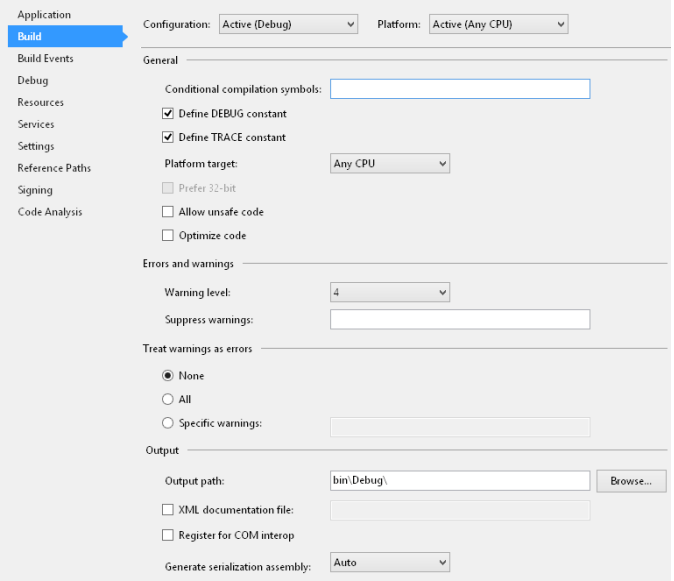
For example, early in the game, managing city internals, such as allocating labor to produce specific resources, can be automatically managed by the game (see Figure 6). As a mechanic, the game is capable of performing reasonable actions on the player's behalf, leaving the player to focus on the higher-level strategies of managing their civilization. Later in the game, when resource management becomes critical for success, players can fine-tune city production parameters to achieve specific resource outputs.

On the contrary, consider the screenshot from the Build Properties page of Microsoft Visual Studio[10] in Figure 7a – the purpose of which is to configure the build. Early in the project, a programmer is likely to first add the project to version-control, and second, specify the location of project build-path binaries to be set outside of version-control tracked folders. In all likelihood, the programmer does not care about "conditional compilation symbols," specifying "`DEBUG` and `TRACE` constants," or specifying "specific warnings" to treat as errors, even though these options are shown to them in the page settings. If instead the IDE was a game, the interface would only show material affordances aligned with the intentions (formal affordances) typically associated with this stage of project development. A mockup of project settings supporting progressive complexity for a code base that is in its early stages of development is shown in Figure 7b. Here, only the options, or content, that are likely to be useful to the programmer are afforded to them.

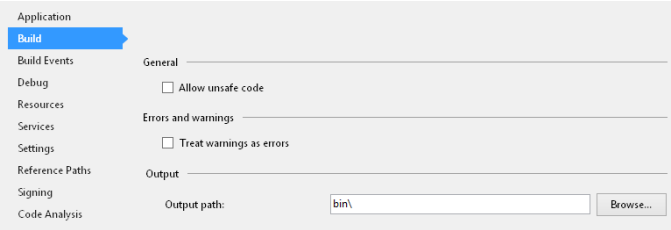
Of course, we expect that the programmer will eventually want more advanced features. Civ V manages this progressive complexity through the technology tree, seen in Figure 8. Branches in the technology tree will enable different available

[9]Another aesthetic, provided directly by Visual Studio, is the green rectangular marker that indicates which lines of code have been changed (see Figure 5b). This lets the programmer know *which* changes helped correct the defect.

[10]http://www.visualstudio.com/



(a) Current Build Properties page.



(b) Build Properties page incorporating progressive complexity.

Fig. 7. In Visual Studio, the Build Properties page contains an overwhelming number of options. For many of these options, the programmer is likely to be indifferent. As with games, the programmer should be able to trade-off performance or configurability for simplicity.

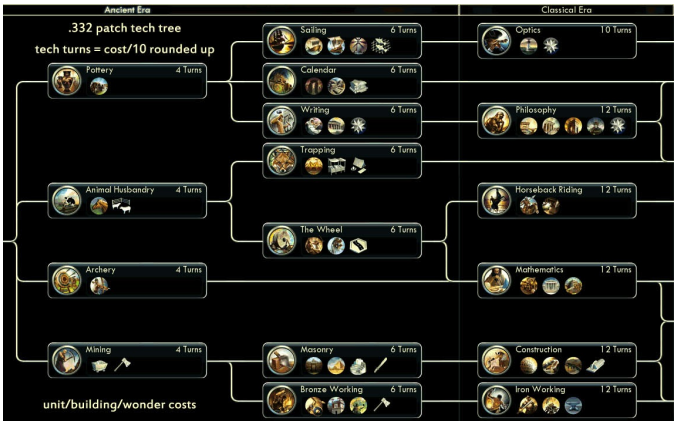
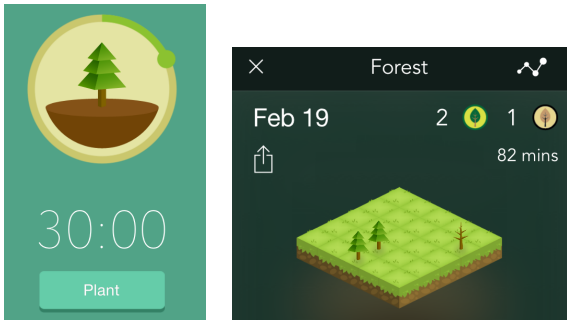


Fig. 8. The technology tree within *Civilization V*. The technology tree enables different in-game actions depending on the player's intentions as they search the tree space. This metaphor could be applied to programming, allowing programmers to edit their environment to enable specific workflows.

in-game actions. At the game's onset, generic technologies are available for use, and as the player's intentions grow in scope, so do the branches of the technology tree. A similar approach could be used in an IDE whereby the tools become available in



(a) (b)

Fig. 9. The *Forest* mobile application helps users put down their phone and minimize phone-related distractions. In (a), the user sets a time period in which they will not use their device. The duration of the timeout determines the type of tree that is planted for the session. In (b), a tree is planted in the forest for each successful session; an interrupted session is rendered as a killed tree.

specific development contexts that programmers explicitly add. As intentions change, they could select specific technologies they want their IDE to accommodate.

VI. Implications

Our argument that gamification today is narrow, and that more meaningful game experiences can be achieved through broad gamification by conceptualizing programming environments as serious games, presents opportunities and challenges for practitioners and researchers who wish to employ gamification in their own work. We describe three key implications that follow from our findings:

Promote fiction and narrative in gamification. As we identified in Section IV, programming environments incorporating gamification have either ignored or found it difficult to integrate immersive narrative elements into their design. Although the nature of programming may limit our ability to fully realize fantasy in programming contexts, we can perhaps "rebuild" narrative bottom-up to offer the programmer *some* level of narrative. Minimally, the narrative should provide the player with a sense of a beginning-middle-end structure [40]. Furthermore, the narrative can be separated into its constituent elements of *story* — that is, "What is told?" and the *discourse* — that is, "How is it told?" [40].

Concretely, consider the mobile phone application, *Forest*[11], illustrated in Figure 9. *Forest* helps users put down their phone and minimize phone-related distractions. It does so by asking the user to specify an arbitrary amount of time, which determines the type of tree that is planted in the game (Figure 9a). If the user does not leave the application within this time, a tree is planted in their forest; if the application is interrupted, the tree is killed (Figure 9b).

We can describe *Forest* through a narrative lens, and therefore a game design lens, using story and discourse. In the case of *Forest*, the *story* is a lack of an event — the user should not use the phone during the duration of the story. Progress within the story is represented as the growth of the tree. The tree

[11]http://www.forestapp.cc/

begins life as a small seed, and if uninterrupted, eventually grows to maturity. The *discourse* through which this story is presented is a metaphor that represents time as a spatial forest. Although the fantasy in this narrative is not extravagant, it is effectiveness nevertheless.

Like *Forest*, which materializes, or *reifies*, abstract concepts such as time into explicit beginning-middle-end structures, we too can support programmers through narrative by artificially introducing explicit beginning-middle-end structures into their programming environments. For example, we could introduce the forest metaphor and mechanics directly into an IDE, allowing the programmer to select a time block for the tree they wish to plant. The tree could to grow to maturity as long as the programmer avoids distracting resources, such as social websites, during the time block. As the programmer accumulates successfully completed time blocks throughout the work week, they would be rewarded with the generation of a charming, procedurally-generated forest.

Frame programming environments tailored to programmer literacy. A second observation is that gamification today does not appear to offer framing as a game design element. That is, programming environments assume that all programmers are interchangeable and come into the environment having the same *programmer literacy*. As Gee demonstrates, good games are "learning machines," and operationalize this principle in many ways, for example, by providing information "on demand" and "just-in-time," as well as through explicit *customization*, which allows the player to directly specify their skill level.

Specifically, Desuvire and colleagues argue that upon turning on the game, the player should have enough information to get started in play, and that, players should not need a manual to play the game [41]. One way to accomplish this is by providing in-game, rather than out-of-game information. In one study, Anderson and colleagues found that when features could not be discovered easily through experimentation, providing tutorial information as closely as possible to when that information is needed increased play time by 16% and progress by 40% [42].

To understand how programming environments might apply framing, let us take the scenario of teaching the programmer to use the *rename refactoring* tool in their IDE. Using this tool, the programmer can automatically and safely rename program elements in their code, yet refactoring tools in generally are rarely used [43]. One reason for this underuse may be that the programmer is not aware that such a tool even exists.

To address this mismatch, let us considering *reversing* the situation from Civ V's automated workers (Figure 6), in which the game automatically managed worker units until it became necessary for the player to fine-tune them. Now, the programmer will manually perform rename refactoring actions as they are currently doing, and behind the scenes we will implement a refactoring detector to identify if they are performing a rename task by hand [44]. If the rename task is a simple one, the programmer is likely to succeed and nothing needs to be done. But consider a case when the programmer attempts to perform a significantly more complex rename refactoring — and fails to do, resulting in a compiler error. If the environment is able to detect this scenario, it would be a perfect time to make the programmer aware of the automated rename refactoring tool: the information would be presented as closely as possible at a time when the programmer is likely to be frustrated and looking for help. Through the use of framing, we are able to increase the likelihood that the programmer will adopt the rename refactoring tool.

Incorporate multiple game elements to increase diversity of users. A third observation is that existing software engineering gamification approaches have a social component. Yet, the majority of game frameworks — with the notable exception of McGonigal [1] — do not emphasize the social component as a first-class principle of game design. While important, perhaps it is the case that software engineering researchers are overemphasizing socialness as an effective gamification technique. For example, Derex and Boyd found that *full* social connectedness is not always better and that "partially connected groups produce more diverse solutions, and this diversity allows them to develop complex solutions that are never produced in fully connected groups" [45]. Furthermore, Bergeron found that too much social connectedness yields a "paradox of organizational citizenship," in which socially-rewarded and publicly visible "good citizenship" actions come at the expense of individual task performance [46].

Existing approaches also appear to use competitive, reward-based game mechanics as their primary gamification approach. However, research in game studies reveal significant age and sex differences in video game play [47], [48]. For instance, Lucas found that female respondents reported "less motivation to play in social situations, and less orientation to game genres featuring competition" [48].

This evidence may explain why men participate more than women in communities such as a Stack Overflow, which are more socially-oriented and reward-based [49]. By incorporating a wider breadth of game elements in programming environments, tool designers can potentially increase the inclusiveness for the kinds of programmers who would use their tools.

VII. Conclusion

The work in this paper encourages software engineering researchers to re-examine their assumptions and biases about the way in which they incorporate game elements into their programming environments. We have argued that gamification as implemented today is predominantly narrow, focused almost exclusively on reward-based game mechanics — a single dimension of the SGDA framework.

We argue that examining programming environments as serious games serves as a starting point for investigating and influencing the way in which we think of gamification today. Through incorporating broader game elements, our thinking about programming environments can potentially go well beyond points, badges, and leaderboards. In turn, programming environments can promote more immersive game experiences that are better tailored to individual programmer literacy, and more inclusive of programmer diversity.

REFERENCES

[1] J. McGonigal, *Reality Is Broken; Why Games Make Us Better and How They Can Change the World*. Penguin Books, 2011.

[2] J. Hamari, J. Koivisto, and H. Sarsa, "Does gamification work? – A literature review of empirical studies on gamification," in *Hawaii International Conference on System Sciences*, Jan. 2014, pp. 3025–3034.

[3] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "From game design elements to gamefulness," in *MindTrek '11*, Sep. 2011, pp. 9–15.

[4] K. Mitgutsch and N. Alvarado, "Purposeful by design? A Serious Game Design Assessment Framework," in *Foundations of Digital Games*, May 2012, p. 121.

[5] S. Toulmin, *The Uses of Argument*. Cambridge University Press, 2003.

[6] P. Besnard and A. Hunter, *Elements of Argumentation*, 2008, vol. 1.

[7] E. Murphy-Hill, T. Zimmermann, and N. Nagappan, "Cowboys, ankle sprains, and keepers of quality: How is video game development different from software development?" in *ICSE*, May 2014, pp. 1–11.

[8] C. Lewis and J. Whitehead, "The whats and the whys of games and software engineering," in *GAS '11*, May 2011, pp. 1–4.

[9] D. R. Michael and S. L. Chen, *Serious Games: Games That Educate, Train, and Inform*. Muska & Lipman/Premier-Trade, Jul. 2005.

[10] D. Djaouti, J. Alvarez, and J.-P. Jessel, "Classifying serious games: The G/P/S model," *Handbook of Research on Improving Learning and Motivation through Educational Games*, pp. 118–136, 2011.

[11] B. Winn, "The design, play, and experience framework," *Handbook of Research on Effective Electronic Gaming in Education*, vol. 3, pp. 1010–1024, 2008.

[12] L. A. Annetta, R. Lamb, and M. Stone, "Assessing serious educational games," in *Serious Educational Game Assessment*. Springer, 2011, pp. 75–93.

[13] E. Adams, *Fundamentals of Game Design*. Pearson Education, 2013.

[14] S. Niedenthal, "What we talk about when we talk about game aesthetics," in *DiGRA*. DiGRA Online Library, 2009.

[15] Nintendo, *The Legend of Zelda: A link to the Past*, 1986.

[16] W. Snipes, A. R. Nair, and E. Murphy-Hill, "Experiences gamifying developer adoption of practices and tools," in *ICSE SEIP*, 2014.

[17] L. Singer and K. Schneider, "It was a bit of a race: Gamification of version control," in *International Workshop on Games and Software Engineering (GAS)*, Jun. 2012, pp. 5–8.

[18] R. Farzan, J. M. DiMicco, D. R. Millen, C. Dugan, W. Geyer, and E. A. Brownholtz, "Results from deploying a participation incentive mechanism within the enterprise," in *CHI*, Apr. 2008, p. 563.

[19] D. J. Dubois and G. Tamburrelli, "Understanding gamification mechanisms for software development," in *ESEC/FSE*, Aug. 2013, pp. 659–662.

[20] B. Vasilescu, V. Filkov, and A. Serebrenik, "StackOverflow and GitHub: Associations between Software Development and Crowdsourced Knowledge," in *International Conference on Social Computing*, Sep. 2013, pp. 188–195.

[21] T. Dong, M. Dontcheva, D. Joseph, K. Karahalios, M. Newman, and M. Ackerman, "Discovery-based games for learning software," in *CHI '12*, may 2012, p. 2083.

[22] K. Januszewski. (2012, Jan) Visual Studio Achievements FAQ. [Online]. Available: http://channel9.msdn.com/Blogs/C9Team/Visual-Studio-Achievements-FAQ

[23] R. Minelli, A. Mocci, and M. Lanza, "Free hugs: Praising developers for their actions," pp. 555–558, May 2015.

[24] R. M. Ryan, C. S. Rigby, and A. Przybylski, "The motivational pull of video games: A self-determination theory approach," *Motivation and Emotion*, vol. 30, no. 4, pp. 344–360, Nov. 2006.

[25] R. Ryan and E. Deci, "Intrinsic and extrinsic motivations: Classic definitions and new directions," *Contemporary educational psychology*, vol. 25, no. 1, pp. 54–67, Jan. 2000.

[26] E. L. Deci, R. Koestner, and R. M. Ryan, "Extrinsic rewards and intrinsic motivation in education: Reconsidered once again," *Review of Educational Research*, vol. 71, no. 1, pp. 1–27, Jan. 2001.

[27] E. L. Deci, "Effects of externally mediated rewards on intrinsic motivation." *Journal of Personality and Social Psychology*, vol. 18, no. 1, pp. 105–115, 1971.

[28] E. S. Elliott and C. S. Dweck, "Goals: An approach to motivation and achievement." *Journal of Personality and Social Psychology*, vol. 54, no. 1, pp. 5–12, Jan. 1988.

[29] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *CHI*, May 2011, p. 2857.

[30] I. Bogost. (2011, August) Gamification is Bullshit. [Online]. Available: http://bogo.st/wm

[31] M. Robertson. (2010, August) Can't play, won't play. [Online]. Available: http://www.hideandseek.net/2010/10/06/cant-play-wont-play/

[32] P. Jagoda, "Gamification and other forms of play," *boundary 2*, vol. 40, no. 2, pp. 113–144, Jul. 2013.

[33] J. Hamari and V. Eranti, "Framework for designing and evaluating game achievements," in *DiGRA*, 2011.

[34] H. Erdogmus, M. Morisio, and M. Torchiano, "On the effectiveness of the test-first approach to programming," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 226–237, Mar. 2005.

[35] E. Aarseth, "From Hunt the Wumpus to EverQuest: Introduction to Quest Theory," *Entertainment Computing - ICEC 2005*, vol. 3711, pp. 496–506, 2005.

[36] B. Softworks, *The Elder Scrolls V: Skyrim*, 2011.

[37] A. Cockburn, *Agile Software Development: The Game*, 2nd ed. Addison-Wesley, 2006.

[38] M. Mateas, "A preliminary poetics for interactive drama and games," *Digital Creativity*, vol. 12, no. 3, pp. 140–152, Sep. 2001.

[39] K. Games, *Sid Meier's Civilization V*, 2010.

[40] S. Chatman, *Story and Discourse: Narrative Structure in Fiction and Film*. Cornell University Press, 1978, vol. 1.

[41] H. Desurvire, M. Caplan, and J. A. Toth, "Using heuristics to evaluate the playability of games," in *Extended abstracts - CHI '04*, Apr. 2004, p. 1509.

[42] E. Andersen, E. O'Rourke, Y.-E. Liu, R. Snider, J. Lowdermilk, D. Truong, S. Cooper, and Z. Popovic, "The impact of tutorials on games of varying complexity," in *CHI '12*, May 2012, p. 59.

[43] E. Murphy-Hill, C. Parnin, and A. P. Black, "How We Refactor, and How We Know It," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 5–18, Jan. 2012.

[44] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *ICSE '14*, may 2014, pp. 1095–1105.

[45] M. Derex and R. Boyd, "Partial connectivity increases cultural accumulation within groups," *Proceedings of the National Academy of Sciences*, vol. 113, no. 11, pp. 2982–2987, Feb. 2016.

[46] D. M. Bergeron, "The potential paradox of organizational citizenship behavior: Good citizens at what cost?" *Academy of Management Review*, vol. 32, no. 4, pp. 1078–1095, oct 2007. [Online]. Available: http://amr.aom.org/content/32/4/1078.full

[47] B. S. Greenberg, J. Sherry, K. Lachlan, K. Lucas, and A. Holmstrom, "Orientations to Video Games Among Gender and Age Groups," *Simulation & Gaming*, vol. 41, no. 2, pp. 238–259, jul 2008. [Online]. Available: http://sag.sagepub.com/content/41/2/238.short

[48] K. Lucas, "Sex differences in video game play: A communication-based explanation," *Communication Research*, vol. 31, no. 5, pp. 499–523, Oct. 2004.

[49] B. Vasilescu, A. Capiluppi, and A. Serebrenik, "Gender, Representation and Online Participation: A Quantitative Study of StackOverflow," in *2012 International Conference on Social Informatics*, Dec. 2012, pp. 332–338.