Experiences Designing and Validating a Gamified Development Environment for Learning Programming

Jeisson Hidalgo-Céspedes, M.Sc[1], Gabriela Marín-Raventós, Ph.D[1], and Vladimir Lara-Villagrán, Ph.D[1]
Universidad de Costa Rica, Costa Rica, jeisson.hidalgo@ucr.ac.cr, gabriela.marin@ucr.ac.cr, vladimir.lara@ucr.ac.cr

Abstract*– Learning to program in a programming language is a difficult task for Computer Science students. Vygotsky's constructivism theory states that learning is unavoidably done through association of new concepts with existing ones. Based on this theory, students must build upon life experience concepts, abstract computer concepts (like memory indirection and execution threads), and programming language concepts (like pointers and threads). We hypothesize that we can ease the association process and improve the learning of abstract concepts by using metaphors, letting students program them directly through gamified development environments. We propose a methodology to design gamified development environments supporting the concept association principle. We provide an example of a gamified development environment idea using metaphors for learning abstract programming concepts reported as difficult to learn in a student survey. The gamified development environment idea was validated by Programming II (CS2) professors through two focus groups with slightly positive results.

Keywords—Learning; programming language; video game; metaphor.

Digital Object Identifier
(DOI):http://dx.doi.org/10.18687/LACCEI2016.1.1.182
ISBN: 978-0-9822896-9-3
ISSN: 2414-6390

Experiences Designing and Validating a Gamified Development Environment for Learning Programming

Jeisson Hidalgo-Céspedes, M.Sc[1], Gabriela Marín-Raventós, Ph.D[1], and Vladimir Lara-Villagrán, Ph.D[1]
[1]Universidad de Costa Rica, Costa Rica, jeisson.hidalgo@ucr.ac.cr, gabriela.marin@ucr.ac.cr, vladimir.lara@ucr.ac.cr

***Abstract*– Learning to program in a programming language is a difficult task for Computer Science students. Vygotsky's constructivism theory states that learning is unavoidably done through association of new concepts with existing ones. Based on this theory, students must build upon life experience concepts, abstract computer concepts (like memory indirection and execution threads), and programming language concepts (like pointers and threads). We hypothesize that we can ease the association process and improve the learning of abstract concepts by using metaphors, letting students program them directly through gamified development environments. We propose a methodology to design gamified development environments supporting the concept association principle. We provide an example of a gamified development environment idea using metaphors for learning abstract programming concepts reported as difficult to learn in a student survey. The gamified development environment idea was validated by Programming II (CS2) professors through two focus groups with slightly positive results.**

Keywords—Learning; programming language; video game; metaphor.

I. Introduction

In order to obtain their Computer Science degree, students must demonstrate proficiency in several programming paradigms, and deep knowledge in at least one programming language [1]. Nevertheless many students find this difficult and an unpleasant activity [2]. Several universities worldwide have reported a 33% failure rate in the first two programming courses [3]. There is also evidence of students who approve their courses without basic knowledge of programming [4].

To understand why learning to program is a difficult task, we looked into several influential learning theories, and found that Vygotsky's constructivism theory provided a plausible explanation. Vygotsky's constructivism states that learning is done by association of new concepts with existing ones. Nobody can learn a concept without associating it with something [5]. When learning to program a computer, students must mentally construct abstract concepts like pointers, streams, and execution threads, by associating them with other concepts acquired in their life experience. Professors very seldom explore students' previous concepts to explain programming concepts. The objective of this research is to evidence the theoretical importance of previous real world concepts to learn programming, and to propose a methodology to ease the association process of abstract programming concepts with ordinary concepts though metaphors.

Vygotsky's constructivism, discussed in section II, provides a theoretical framework to guide the teaching-learning process. We reviewed existing game based tools for learning a programming language in section III. Following the theoretical guidelines, we propose in section IV, a methodology to create gamified development environments using metaphors for representing abstract concepts. Section V includes the results of a survey done to students asking them to isolate the topics they consider as difficult and useful. Section VI presents Puppeteer++, a gamified development environment idea that proposes metaphors for those topics. It is validated in section VII through a focus group with CS2 professors.

II. Theoretical Background

Learning is the biological capability of the brain to change its structure in order to adapt itself to the environment and ensure the survival of the species [6]. *Constructivism* states that apprentices do not reproduce knowledge, but they mentally reconstruct it by associating each new concept with existing ones [5]. For example, if we ask you to read the remaining of this section and then say what you remember, you will use your own words to build similar ideas. This evidences that the mind does not reproduce the text, but reconstructs it using previous knowledge. Each reader will use different words because he or she has a different life experience to make associations with. [5]

According to Vygotsky's constructivism, learning is a *process* of construction of concepts associating them with existing ones, and forming *concept systems* that can be applied to new situations, for example, to solve problems. We derived from [5] the following steps for a recommended teaching-learning process.

1. *Motivation*. The mind is unable to make associations in passive condition. Educators should first turn student's minds in active state through motivation to allow the creation and association of concepts. Several techniques can be used. For example, structuring the class to keep students doing activities and collaborating with others; making them understand the importance and utility of the new topic (or the risks of ignoring it); using emotive situations like existential problems, games, and stories. [5], [8]
2. *Conceptual contraposition*. New concepts must be built using existing concepts. As stated previously, nobody can construct a concept without associating it with something. When old notions do not help construct new concepts, the conceptual contraposition technique is useful. It consists in making students realize that their old abilities and notions are insufficient or contradictory to reach the objectives they are motived to reach. It creates a cognitive

Digital Object Identifier (DOI): http://dx.doi.org/10.18687/LACCEI2016.1.1.182
ISBN: 978-0-9822896-9-3
ISSN: 2414-6390

uncertainty state in the student's mind that can only be overcome by reorganizing old concepts and constructing new ones. New concepts are desired and welcome by students in order to surpass the uncertainty and reach a gratification state of equilibrium. [5]. The conceptual contraposition technique produces an intrinsic motivation in the learner that does not require an external reinforcement [8].

3. *Concept assimilation*. When the student's mind is requiring a new concept, the professor can present it, *explaining it by using other concepts in the apprentice's knowledge base*. Under normal conditions, students will not appropriate of a new concept immediately, it will be only temporary associated in short-term memory. An iterative analysis-synthesis-application process is required to gradually establish the connections in long-term memory. All iterations must keep the new concept fundamental principle and vary its non-essential aspects. [5]
4. *Concept application*. Once assimilated in abstract thinking, the new concept should be applied to practical situations, otherwise it will not add any meaningful change to student's life experience. Problem solving and artistic education are two rich scenarios to apply concepts. Both require a general method or process to get a solution or product. [5]
5. *Habit acquisition*. A habit is developed as consequence of the refinement of a method through its repeated application to several distinct situations sharing the same fundamental principle. Students feel confused during the first concept applications, requiring analysis-synthesis processes. Since each iteration keeps the concept's fundamental principle, a natural connection is made with the previous iteration, reducing the analysis-synthesis effort. When this effort is almost inexistent, the habit has been acquired. The goal of teaching is to provide a formal method or process in the conscious phase before the habit is acquired. [5]
6. *Concept systems*. Vygotsky states that knowledge is not constructed by isolated concepts, but systems of associated concepts that reflect the relationships to objects and real life phenomena. According to him, professors should organize the learning material to reflect a natural hierarchy. After learning a new concept, students should solve more comprehensive problems that require associating the new concept with previous ones; that is, constructing concept systems. Evaluating a just learnt concept is insufficient It is necessary to evaluate that concept systems are stable over time and that students apply them to new situations. [5]

An individual can learn by his/her direct interaction with objects. But the most natural and effective learning is by interacting with other people. Vygotsky's Social Constructivism theory states, that what an individual learns becomes his/her reality. Each individual implicitly evaluates his/her notions against the knowledge from others. The validated notions in the mind of several individuals conform the collective knowledge, the reality. Collaboration is the richest learning environment because allows learning and validating from others in a natural way. [9]

A. Discussion

Vygotsky's theory places great importance in using existing concepts to create, associate and apply new concepts; that is, to learn new concepts. When students learn to program, they must formalize and apply ordinary concepts like sequence, condition and repetition. But other computer concepts are abstract, like memory segments, pointers and execution threads. Students cannot construct these concepts by associating them directly to concrete objects, because they cannot be sensed. They must resort to imagination in order to have something to associate them with, and this process can be a source of wrong or weak connections.

Students must apply abstract computer concepts to solve real life problems, through their representation in a selected programming language. Therefore students must build a least three levels of associations: (1) life experience concepts with abstract computer concepts, (2) abstract computer concept with its representation in the programming language (rules, syntax), and (3) programming concepts with the problem to solve.

This complex system of associations can be one of the factors that explain the programming learning difficulties reported by several students. We propose to aid this association process by using *metaphors*, or high-level systems of metaphors called *allegories*.

Associations of type (1), previously stated, can be strengthen by representing abstract computer concepts with some colloquial concepts that share most of the characteristics and relationships. For example, the abstract concept of *nodes in a linked list* can be represented by wagons of a train, and the locomotive represents the head of the list. When a wagon must be inserted to some point of the train, the operator must travel from the locomotive to that point, untie the wagons and attach the new wagon among them.

Teachers can naturally use metaphors in lessons when introducing abstract programming language concepts, for example, using toy wagons for illustrating the linked list. But this is a behaviorist approach. Constructivist suggests that students work directly with the metaphors. That is, students should play with a wagon train, and they should deduce the rules for building linked lists.

Playing with toy wagons can help students deduce properties of linked lists. But when they must implement linked lists, they must take a long leap from the wagon toys to the programming language concepts. We want to also undertake associations of type (2), letting students play with the metaphors directly in the programming language. Therefore metaphors must be both, familiar to students and

operable by the computer. Our proposal is to confluence both requirements into a gamified development environment.

Associations of type (3), related to solving real life problems by applying abstract programming concepts, are the most challenging to address. Gamified development environments must provide a rich set of situations where the programming concepts can be applied. We suggest designing the gameplay of the gamified development environments to support the learning process presented in section II.

III. Related work

A gamified development environment is defined in this paper as a software tool designed following gamification principles to help students learn to program using a specific programming language. Some systems reported as video games, can be classified gamified development environments. The following list describes existing video game or gamified systems designed to help students learn a programming language.

1. Robocode (2001) is a tank battle simulator. Students must create their own tank by inheriting from a Java class (Robot) and build in some survivor and attack logic, in order conquer battles against tanks trained by other programmers. [10]
2. Greenfoot (2003) is an IDE (Integrated Development Environment) based in BlueJ. It provides sceneries with graphic and event handling capabilities for building graphic intensive applications in Java, such as visualizations and video games. [11]
3. *Bomberman* (2008) is actually a system that presents slides containing C programming information, examples and exercises. Students must solve graphical exercises inspired in the original Bomberman game created by Hudson Soft in 1983 [12]. This system was designed under the programmed instruction principle of behaviorism theory.
4. CodeCombat (2013) is an online strategy game. Players must train their troops by writing strategy logic in JavaScript, in order to conquer multiplayer battles.
5. CodeSpells (2013) is a role-playing video game. Players are apprentice wizards that create magic spells in Java, and use them to help villagers. [13], [14]

In all of these systems, game concepts are not metaphors of abstract concepts of a programming language, their properties and relationships. For example, a tank object in Robocode tries to resemble a real battle tank, its properties, actions, limitations and relations with others tanks in the battle scenery. But a tank does not represent an abstract programming concept. It does not try to explain a node, a reference, or a function call, nor its properties and relationships.

Two exceptions can be remarked. First, a code block is represented as a magic spell in CodeSpells. This metaphor helps students associate the concept of program with a more familiar concept, but some properties are missing, like debugging. Second, visual objects in Greenfoot must be inherited from World or Actor, and relationships are diagrammed using arrows.

Our proposal suggests creating gamified development environments where objects represent simultaneously both worlds: a familiar real life concept and an abstract programming concept. So, when students follow natural gameplay rules for the objects, they automatically infer and learn abstract programming rules.

IV. Proposed methodology

We propose the following steps to develop gamified development environments that represent abstract concepts with colloquial concepts (metaphors).

1. Choose the abstract concepts that students must learn. Describe their fundamental properties: attributes, behaviors and relationships.
2. Look for some colloquial concepts (a family of concepts) that share similar properties with abstract concepts. Map their attributes, behaviors and relationships. If more than one family of concepts is found, choose the family that shares most properties with abstract concepts.
3. Propose one game idea built upon the colloquial concepts. Structure its gameplay following a learning theory, like the steps derived from Vygotsky's constructivism (section II). Players must program in order to overcome the game challenges. The product of this step is a gamified programming environment.
4. Determine if the gamified programming environment idea can support other abstract concepts.
5. Validate the gamified programming environment idea with experts.
6. Implement the gamified programming environment following a software or game development process.
7. Evaluate the gamified programming environment under a learning environment.

V. Survey results

A programming language has many abstract concepts that can be represented with gamified metaphors. In order to choose the programming language and its abstract concepts to implement (step 1 in the proposed methodology), we surveyed our students at our School of Computer Science, the target population of our proposal. We asked them for the programming language they use the most, and the topics they consider difficult and useful.

The survey was conducted to all the students that completed the five courses listed in Table 1. These courses are

highly related to programming topics, and they range from second to fourth year of our Computer Science Bachelor. Students were not surveyed twice.

TABLE 1.
PROGRAMMING RELATED COURSES THAT WERE SURVEYED

Acr.	Name	Year	Req.
CS2	Programming II	2.I	CS1
DSA	Data Structures and Algorithms	2.II	CS2
DB1	Data Bases I	3.I	DEA
SE1	Software Engineering I	3.II	DB1
SE2	Software Engineering II	4.I	SE1

The anonymous self-administered questionnaire was answered by 144 students. There were two incomplete instruments and one student that did not report sex. Men answered 114 cases (81%) and women 27 cases (19%). These response rates reflect the gender distribution of the student population.

The Programming I course (CS1) teaches Java and Programming II (CS2) teaches C++. We asked students the approximate usage percent of these and other programming languages. In average, C++ was reported as the most used programming language (51%), followed by Java (33%). The usages vary through courses mainly due to the professors' preferences for assignments as shown in Figure 1. Professors of Data Structures and Algorithm Analysis (DSA) prefer C++, and professors of Software Engineering (SE) prefer Java. As expected, the usage of other programming languages –such as SQL, JavaScript, Lisp, and C#– increases as students advance through the major.

We asked the students to grade the learning difficulty they experienced with several C++ programming topics, using a scale ranging from 1, meaning not difficult at all, to 10, meaning the highest difficulty. They also graded the perceived usefulness of the topics. The findings are summarized in Figure 2. Since we did not want to address very difficult topics that were not useful, and vice versa, we multiplied the

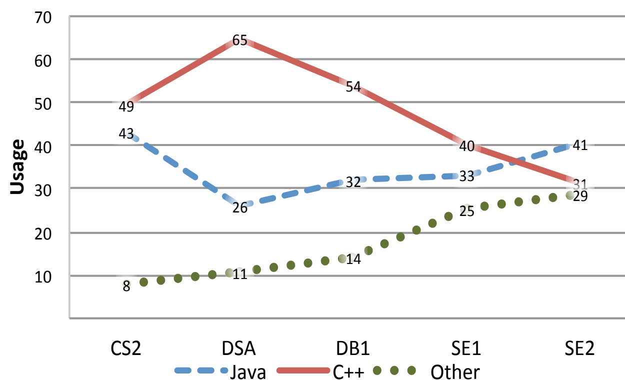

Figure 1. Programming language usage through courses

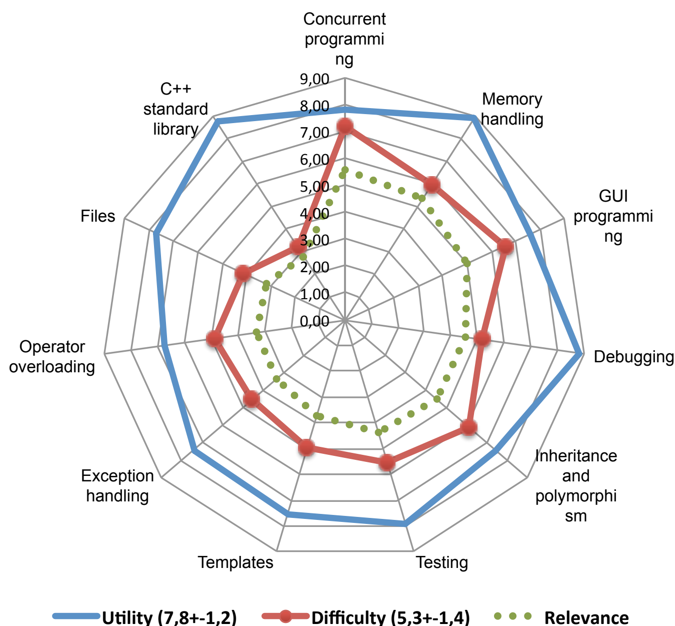

Figure 2. Utility and learning difficulty of some C++ topics

usefulness by the learning difficulty of each topic, and called it *learning relevance*. Figure 2 is ordered clockwise by this metric. Parallel/concurrent programming was considered as the most difficult topic to learn, with a utility of 7,8 of 10. Memory handling was the second most useful (8,9 of 10), and at the same time, as difficult (6,0 of 10). In the next section we will propose a gamified development environment idea for addressing these two topics.

VI. GAMIFIED DEVELOPMENT ENVIRONMENT IDEA

Survey results reported that C++ is the most used programming language by our students, and that "parallel/concurrent programming" and "memory handling" are considered the two most relevant topics to learn. In this section we design a gamified development environment that supports abstract programming concepts for these two topics, following the proposed methodology in section IV.

Step 1: choose the abstract concepts to represent with metaphors

The abstract concepts to implement are the main concepts from the selected programming topics. They are listed in Table 2 with a brief description.

Step 2: find colloquial concepts that resemble abstract ones

Let's begin with memory segments. They resemble areas to store objects. Segments are size limited except heap, which is huge. An execution thread resembles a worker. Workers could be interested in accomplishing some tasks, but their working area is limited (stack segment). In order to have

access to the big area they require a special mechanism (a pointer). Two or more workers can access simultaneously objects in the huge area, but each worker does not share its own direct working area (stack segment).

TABLE 2.
MAIN ABSTRACT PROGRAMMING CONCEPTS TO SUPPORT

Concept	Description
Execution thread	A set of instructions that can be run independently from other running instructions.
Shared memory	Some memory that can be accessed simultaneously by two or more running threads.
Memory segment	Program's memory is distributed in segments like code, data, stack and heap. All of them are very limited in size except heap. Pointers are mandatory for accessing the heap. Threads of a same program share all segments except their stacks.
Pointer	An integer variable that stores the address of an object allocated in another place of the memory. Its value allows accessing the pointed object.
Function call	Action of running a function or method providing values for its parameters and waiting for its result.

A number of families of colloquial concepts can be fitted to the previous descriptions. We propose one related to puppetry (Figure 3). Puppets are unanimated characters that act in the scenery (heap segment) controlled by puppeteers (execution threads). Puppeteers are not supposed to act, therefore they never appear in scenery. They work over a platform (stack segments) at the top of the theatre, hidden from the audience. A puppeteer controls its puppet in the scenery through strings (pointers). Puppeteers animate their marionettes following step by step a script (code segment).

A puppeteer could control several puppets, but not at the same time. Switching from one puppet to another introduces a visible delay. Also the puppeteer requires a "handle line rack" (stack segment) to hold inactive puppets (Figure 4). A puppeteer could perform several different tasks (function calls), depicted as several stacked handle lines in Figure 4. Puppeteer only works with the topmost line. If several puppets

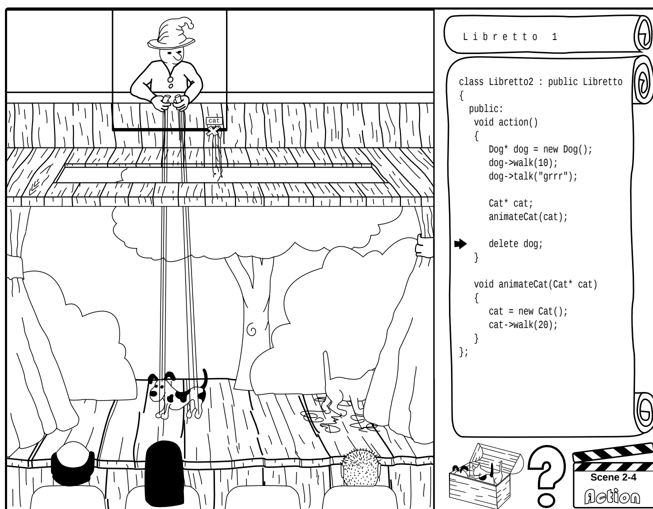

Figure 3. Paper prototype of a puppet theatre game

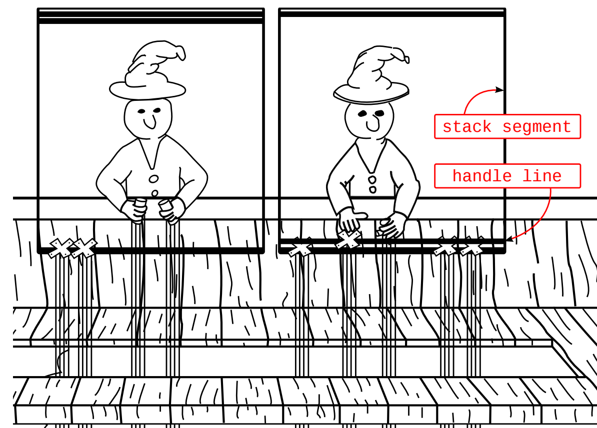

Figure 4. Two puppeteers controlling several puppets

must act simultaneously in the same scene, several puppeteers must work together (concurrence), as depicted in Figure 4. A large puppet, for example a Chinese dragon, requires several coordinated puppeteers (shared memory).

Step 3: build a gamified idea with colloquial concepts

We follow the constructivist principles suggested in section II (Theoretical background). Principle 1 states that through *motivation* the gamified development environment must get student's mind in active condition in order to learn. The game story must challenge the students to arouse their intrinsic motivation, and the gameplay must propitiate that they stay active. The game story will show short videos of entertaining theatre plays and leave an open question: Do you want to create your own play?

The gamified idea empowers students to build their own theatre plays. Players are active playwrights. The welcome screen in Figure 5(a) shows the available plays. Students can create new plays by pressing the "plus" button. Initially they will not know how to write theatre plays, which follows the principle 2 of *conceptual contraposition*. They need scaffolding to construct the concept systems required to write scripts (programming).

After selecting a play in Figure 5(a), the game shows its scenes depicted using cinema claps in Figure 5(b). "Training" play scaffolds students by providing a story in natural language (English) that students must translate to the dialect that puppeteers understand (C++). Initially each scene is incomplete, not translated, which is represented by an open clap in Figure 5(b). Completed scenes can be played in sequence by pressing the play button at the top right of Figure 5(b).

Scenes are grouped in acts. Following principles 3 through 6, the learning material must be logically organized. In "Training" play each act introduces a new concept. For example, Act 1 introduces method calls. Scene 1-1 automatically places a puppet in the scenery, and asks students to greet the audience (say *hello world* by calling a method). Scene 1-2 asks to move the puppet around scenery, and so on.

Act 2 introduces the concept of creation and removal of puppets in the scenery (heap segment).

Each scene applies several learning principles. For example, scene 2-1 directly asks students to create a puppet, to say some words to the audience, and to remove it. Students trying to construct their own theatre play, realize that puppet creation is important. But at this point, they are facing cognitive uncertainty due to a *conceptual contraposition*, "how a puppet is created?" They are requiring a new concept. By pressing the question mark button in Figure 3, the game provides some short visual information about object creation and some examples (principle 3 of *concept assimilation*). The given information will only be associated in short-term memory. Students return to the scene in order to apply the new concept (principle 4, *concept application*).

If for some reason students create a local object in scene 2-1, for example the C++ declaration `Dog dog` in Figure 3, the puppet will appear in the puppeteer's handle line (stack segment), the audience will not see it, and the scene's goal is not accomplished. Students will have an immediate visual cue of the problem. They are still facing a *conceptual contraposition*; their old notions contradict new ones. Using metaphors solving the theatrical defect naturally leads to fixing the code. According to principle 1 (*motivation*), the gamified development environment could also give an explication of the problem and encourage students to try a different approach.

When students use dynamic memory in scene 2-1, for example `Dog* dog = new Dog()`, a puppet will appear in the scenery as expected. The puppeteer will hold in his hands a handle named `dog` connected to the new puppet by some strings. The scene 2-1 is not complete yet. After getting the curtain closed, cleaning the scenery is mandatory in order to finish any scene. When students finally remove the puppets (with the `delete` operator) the scene will be completed, and they will receive the applause from the audience. The praise for the success is immediate (principle 1, *motivation*).

Scene 2-2 asks students to create different types of puppets. Scene 2-3 asks them to create a puppet and animate it by calling some of the methods used in Act 1. Scene 2-4 asks the students to create two puppets and animate one before the other. And so on. Each scene keeps the fundamental principle of object creation and removal. It is incrementally applied to several situations helping establish the associations according to principle 4 of *concept application*. After creating and deleting objects over and over, the diffuse and slow first reactions become almost mechanical by the end of the Act, leading to the acquisition of a habit (principle 5).

Act 3 introduces the concept of concurrence: two or more puppeteers animating puppets. Concepts from Act 1 and 2 are reapplied in Act 3. Thus, new notions will be associated with existing ones forming a *concept system* (principle 6). Act 4 asks students to create their own puppets. Act 5 helps students create their own sceneries. After completing the Training play, students can apply their learning to build a real theatre play

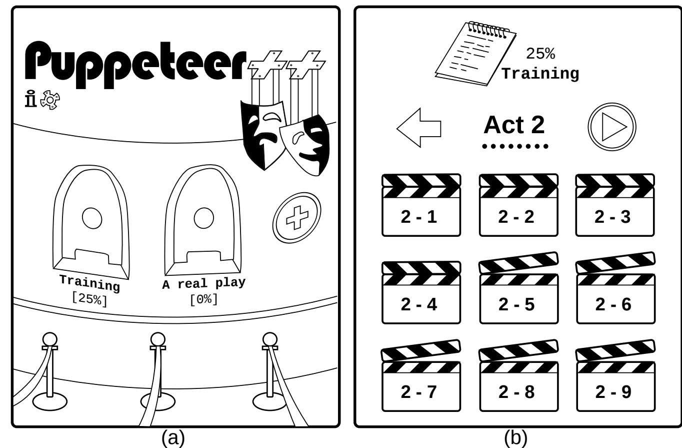

Figure 5. (a) Selecting a play. (b) Selecting a scene

under the generic title of "A real play" in Figure 5(a). They have finally acquired the basic concept system to build their own theatre plays. Creating a theatre play is a laborious task if done individually. More elaborated theatre plays can be built in collaboration. The game can support Vygotsky's social constructivism theory letting a team of students work with a theatre play simultaneously.

Step 4: support other abstract concepts, if applicable

The puppet theatre idea can also support "debugging" and "inheritance and polymorphism". These are the fourth and fifth most relevant topics according to the survey results (Figure 2). *Debugging* is a methodical process of detecting and correcting defects. Each time a script is in action, the running line is remarked. Students can pause the run, and execute the script line per line. When something is wrong, graphical feedback is natural and not overwhelming as in traditional debuggers. For example, in Figure 3 the pointer in stack memory to the cat puppet was lost when its method finished executing, but the pointed object was not deleted (a memory leak). The strings tying the handle with the cat are just visually cut. *Inheritance and polymorphism* is required when students want to create their own puppets and sceneries. They have to inherit from `Puppet` and `Scenery` classes, and override some functionality.

The puppet theatre metaphor can also support some advanced programming challenges. For example, puppeteers follow the script literally. If students place some decisions based in random variables, it will provide different courses of events each time the play is run. In some real theatre plays, actors invite people from audience to participate in the play. In the game context, the audience can be the real users in front of the computer.

Step 5: Validate the gamified development environment idea with experts

A validation of the *gamified development environment* idea is presented in the following section.

VII. Gamified Development Environment Idea Validation

Before implementing the *gamified development environment idea*, the metaphors involved must be tested. The process of testing the ludic metaphor system is as important as its creation process.

In order to validate the strengths and weaknesses of the proposed *idea*, and try to predict its impact, all our professors of CS2, except one of the authors of this article, participated as experts in a focus group. This group of seven professors is heterogeneous in teaching experience and CS2 involvement. The youngest has nine years of teaching experience and the oldest 34. The professor that has taught CS2 the most has done it for 28 semesters, and there is one professor that is teaching it for the first time. Due to schedule restrictions two focus groups were conducted, with four and three professors respectively. Sessions were organized with the following protocol, where explanation activities are shown within parenthesis:

1. (Introduction) Why is learning difficult in CS2?
2. (Learning theories and metaphors) What metaphors have you used in your classes? Have they been useful?
3. What characteristics do you believe make a good or bad metaphor for the programming learning process?
4. (Presentation of the game idea) What strengths and weaknesses do you find in the puppetry metaphor to teach/learn memory management and concurrency concepts?
5. What would you improve?
6. Any alternative metaphors for learning memory management and concurrency concepts?
7. Do you believe that playing with the proposed tool will positively or negatively impact the learning or motivation of students? Please list the reasons.

Questions were displayed on overhead slides. The moderator made a short introduction or presentation before questions 1, 2 and 4, indicated in parenthesis in the previous list. Participants individually wrote down their answers for questions 4 and 7 before discussing them, to ease their posterior recall and avoid pollution from other professors´ opinions. Professors were informed that the discussion would be recorded and they agreed. Both sessions lasted one hour and 15 minutes each.

A. Data analysis

We followed the same analysis process indicated by [15]. Recordings were listened several times while authors took notes. Notes were analyzed and grouped into themes. A paragraph briefing the main idea was written for each theme. Paragraphs were translated from Spanish to English for this article. The following subsections show each discussed theme.

1. *Learning difficulty in CS2*

The most mentioned reasons were: learning deficiencies in the previous programming course (CS1), the complexity of the programming language (C++), immaturity of students, their lack of discipline and studying strategies.

2. *Metaphors used by participants in their lessons*

Most participants expressed metaphors are important for learning. They use mainly visual metaphors, i.e. drawing abstract geometric figures in the blackboard like rectangles for variables and arrows for pointers. They expressed that other professors use dramatized metaphors. A few metaphors for explaining mechanisms were cited: C++ templates are like rubber stamps or copy-paste-search-replace processes, and computational machine is like the human mind.

3. *Characteristics of good metaphors for learning*

Participants cited the following characteristics for good metaphors:

1. Familiar, relevant for students.
2. Graphical, visual, or dramatized.
3. Abstract, simple, like the program visualization application Jeliot 3. Hide unnecessary details such as standard or third party library internals.
4. Didactic. For example draw complex structures like linked lists, trees or iterators, similar to how books illustrate them.
5. Simple to be able to measure its possible impact (for experimentation purposes).

4. *Strengths and weaknesses of puppetry metaphor*

Participants cited some strengths and no debate was generated about them:

1. Puppetry metaphor is clear, mainly for illustrating instance creation and method calls.
2. It is graphic, visual.
3. It is entertaining; therefore, it will raise students' interest.
4. It is easy to use.
5. It provides immediate feedback.
6. It increases student motivation.

Extensive discussions were hold on its weaknesses. The following list is ordered from the most discussed weakness to the least one.

1. It does not illustrate collections, indexes and iterators.
2. It does not support C++ complicated declarations and their usage, such as pointers to pointers, or pointers to vector of instances.
3. Metaphor is not self–explanatory. Students must invest time understanding the metaphor, afterwards understanding the reality (the machine).

4. Limited coverage of CS2 course topics.
5. Scenery capacity is too limited for lots of actors, for example 101 Dalmatians.
6. Is it ludic?
7. Only a subset of C++ is supported, therefore students can build only limited programs. For example, not all C++ programs require heap allocation.
8. It is not clear on sending and receiving messages between objects (puppets) and sharing information.

5. Improving puppetry metaphor

The following ideas were suggested to enhance the puppetry metaphor:

1. Collaboration and reuse of resources. Several students can create different puppets separately, then import those puppets to build the complete play in a similar way a director does. An official repository of puppets would allow students to share or hire actors, speeding up playwriting.
2. Simplify the metaphor. Remove some scenery distractors like shrubs and curtains.
3. Allow students to run sentences without a full compilation process (code interpretation).

6. Alternative metaphors

Participants of the second focus group suggested as an alternative metaphor any scenario where resource management (memory handling) and quick task completion (concurrency) are required, for example, a building construction such as Minecraft (2011). Students must manage simultaneously several workers (execution threads), such as carpentries and bricklayers; and resources like materials and money.

The four professors of the first focus group, which have had more experience teaching CS2, were more conservative. They suggested *a visual symbolic debugger that shows an abstraction of the computational machine*. Jeliot 3 has this idea, but it is very limited, for example, it does not didactically illustrate data structures such as stacks or trees. The new metaphor should overcome limitations of Jeliot 3.

7. Impact of the puppetry metaphor

Three of the seven professors expressed that the game idea would positively impact the learning and motivation of students; one professor said there would be both positive and negative effects; and three professors were unsure. Professors expressed that they would require empirical results to be convinced. Five participants emphatically suggested using a simplified visual model of the machine, for the empirical evaluation.

B. Validation results

Focus group results show a divided position between positive and unsure impact of the proposed gamified development environment idea. Almost no negative impacts were predicted. Findings strongly claim for empirical evaluation on students considering three treatments:

1. The puppetry metaphor.
2. An abstract visualization of the machine.
3. Traditional learning (control group).

An abstract visualization of the machine is a metaphor also, and it can be built following the proposed methodology of section IV. This is part of our current work.

VIII. CONCLUSIONS AND FUTURE WORK

Learning theories show the importance of associating new notions with previous concepts. Based on this principle, we have proposed a methodology to design, and test before implementation, gamified development environments. The methodology focuses on the association process between abstract programming concepts and ordinary concepts. We hypothesize that letting students play with gamified development environments based in this principle will lead to better learning of underlying programming concepts.

We proposed a gamified development environment idea named Puppeteer++ following the given methodology. It associates abstract programming concepts considered as difficult and useful by our students, with concepts from puppetry. Puppeteer++ idea was validated by seven CS2 professors. In general, experts provided valuable suggestions to improve the metaphoric associations. Experts recommended empirically evaluating the proposed system, and incorporating a visual symbolic debugger that shows an abstract model of the machine as an extra treatment for the experiment.

We are working in enriching the proposed methodology to incorporate visualizations. In the future we will try to test, by a quasi-experiment, if using concrete metaphors for abstract programming concepts in gamified development environments or visualizations influence motivation and aids the learning of those concepts. We hope to have found a means to ease the learning curve of programming language students.

ACKNOWLEDGMENT

This research is supported by the Centro de Investigaciones en Tecnologías de la Información y Comunicación (CITIC), the Escuela de Ciencias de la Computación e Informática (ECCI), both from Universidad de Costa Rica (UCR), and the Ministerio de Ciencia Tecnología y Telecomunicaciones de Costa Rica (MICITT). We thank our colleagues and CS2 professors for their valuable cooperation.

REFERENCES

[1] ACM and IEEE Computer Society, “Computer Science 2013: Curriculum Guidelines for Undergraduate Programs in Computer Science,” 2013.
[2] F. W. B. Li and C. Watson, “Game-based concept visualization for learning programming,” in *Proceedings of the third international ACM*

workshop on Multimedia technologies for distance learning - MTDL '11, 2011, p. 37.
[3] J. Bennedsen and M. E. Caspersen, "Failure rates in introductory programming," *ACM SIGCSE Bull.*, vol. 39, no. 2, p. 32, Jun. 2007.
[4] M. McCracken, V. Almstrum, D. Diaz, L. Thomas, M. Guzdial, I. Utting, and D. Hagan, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students A framework for first-year learning objectives," *ACM SIGCSE Bulletin, Volume 33 Issue 4*, pp. 128–180, Dec-2001.
[5] D. N. Bogoyavlensky and N. A. Menchinskaya, "La psicología del aprendizaje desde 1900 a 1960," in *Psicología y pedagogía*, 4th ed., Sevilla: Ediciones Akal, 2011, pp. 119–188.
[6] S. Francis, *El conocimiento pedagógico del contenido como modelo de mediación docente*. Coordinación Educativa y Cultural, 2012.
[7] W.-H. Wu, W.-B. Chiou, H.-Y. Kao, C.-H. Alex Hu, and S.-H. Huang, "Re-exploring game-assisted learning research: The perspective of learning theoretical bases," *Comput. Educ.*, vol. 59, no. 4, pp. 1153–1161, Dec. 2012.
[8] M. Cecchini, "Introducción," in *Psicología y pedagogía*, Sevilla: Ediciones Akal, 2011, pp. 7–20.
[9] S. W. Harmon, "A Theoretical Basis for Learning in Massive Multiplayer Virtual Worlds," *J. Educ. Technol. Dev. Exch.*, vol. 1, no. 1, pp. 29–40, 2008.
[10] J. O'Kelly and J. P. Gibson, "RoboCode & problem-based learning," in *Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education - ITICSE '06*, 2006, vol. 38, no. 3, p. 217.
[11] P. Henriksen and M. Kölling, "Greenfoot: Combining object visualisation with interaction," in *OOPSLA '04 Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, 2004, pp. 73–82.
[12] W. Chang, Y. Chou, and K. Chen, "Game-based digital learning system assists and motivates C programming language learners," in *2010 Sixth International Conference on Networked Computing and Advanced Information Management (NCM),* 2010, pp. 704–709.
[13] S. Esper, S. R. Foster, and W. G. Griswold, "On the nature of fires and how to spark them when you're not there," in *Proceeding of the 44th ACM technical symposium on Computer science education - SIGCSE '13*, 2013, p. 305.
[14] S. Esper, S. R. Foster, and W. G. Griswold, "CodeSpells: Embodying the Metaphor of Wizardry for Programming," in *Proceedings of the 18th ACM conference on Innovation and technology in computer science education - ITiCSE '13*, 2013, p. 249.
[15] J. Aberg, "Challenges with teaching HCI early to computer students," in *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education - ITiCSE '10*, 2010, pp. 3–7.