Teaching Design Patterns Through Computer Game Development

PAUL GESTWICKI and FU-SHING SUN
Ball State University

We present an approach for teaching design patterns that emphasizes object-orientation and patterns integration. The context of computer game development is used to engage and motivate students, and it is additionally rich with design patterns. A case study is presented based on *EEClone*, an arcade-style computer game implemented in Java. Our students analyzed various design patterns within EEClone, and from this experience, learned how to apply design patterns in their own game software. The six principal patterns of EEClone are described in detail, followed by a description of our teaching methodology, assessment techniques, and results.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-oriented programming; K.3.2 [**Computer and Information Science Education**]: Computer Science Education

General Terms: Design

Additional Key Words and Phrases: design patterns, UML, games in education, assessment

ACM Reference Format:
Gestwicki, P. and Sun, F.-S. 2008. Teaching design patterns through computer game development. ACM J. Educ. Resour. Comput. 8, 1, Article 2 (March 2008), 21 pages. DOI = 10.1145/1348713.1348715. http://doi.acm.org/10.1145/1348713.1348715.

1. INTRODUCTION

As computer science educators, we are always seeking to improve our students' learning experiences. It can be difficult to find examples and case studies that are compelling and appropriate for students. Computer games provide a context that is familiar and motivating to students, and there has been a significant amount of work investigating the educational impact of game programming. This article describes the results of using computer games to teach design patterns for object-oriented programming. The results are based on a series of experiments conducted from Summer 2006 through Spring 2007, and it is a part of ongoing research into the applications of games to computer science and software engineering education.

This builds upon *Computer Games as Motivation for Design Patterns*, which was presented at SIGCSE 2007 by Paul Gestwicki [Gestwicki 2007].

1.1 Design Patterns

Design patterns provide solution strategies for general problems in object-oriented software engineering. These patterns were discovered in successful designs and most famously cataloged by Gamma et al. over a decade ago [Gamma et al. 1995]. Despite an active patterns community and an abundance of research on patterns education and practice, patterns still tend to be underrepresented in undergraduate education.

There has been significant research regarding the teaching of design patterns in the computer science introductory sequence. This work ranges from case studies such as the games Set [Hansen 2004] and Life [Wick 2005] to complete curricular revisions [Decker 2003]. Many of the SIGCSE Nifty Assignments are used to teach specific design patterns and software architectures. The consensus is that through the use of appropriate case studies, combined with effective teaching techniques, students in the introductory sequence can learn patterns. That is, they do not have to be relegated to upper-level courses on principles of software engineering.

Personal experience indicates that students are often unable to comprehend patterns from isolated examples. In a graduate-level seminar on visualization and design patterns, students were asked to develop sample applications to illustrate various patterns. Many of the students were unable to generate accurate, isolated instances of the patterns [Gestwicki and Jayaraman 2005]. This is not a coincidence: it is difficult to translate from the isolated examples of Gamma et al. to the complex and multifaced pattern reifications encountered in practice. Instead, patterns are best taught by presenting them in software of significant scale that contains collaborations among patterns. This belief is supported by the literature: case studies have been used as a means for showing the definition, benefits, and costs of design patterns [Wick 2005; Dewan 2005], and some resources have also shifted from a dictionary presentation of patterns to illustrative examples [Holub 2004].

1.2 Games

The modern college freshman has grown up surrounded by the ever-growing computer game industry, and such games have become an integral part of popular culture. Our students have shown great interest in not only playing games, but also in the interdisciplinary study of game design and development, including visual, audio, business, social, and technical aspects. It is unlikely that this is a local phenomenon, as others have shown that using computer games as learning tools motivates students to learn topics they might otherwise avoid [Claypool and Claypool 2005].

Commercially successful games are usually implemented in C or C++, and the techniques one encounters in game development magazines, books, and forums are usually tricks for squeezing a few more frames per second from a logic and rendering pipeline. In contrast, design patterns bring benefits in clarity and reusability, not (necessarily or by design) execution efficiency. While the vocabulary of patterns is certainly not absent from games, its concerns are frequently supplanted by the need for speed. This primacy of optimization results

in code that is streamlined for one purpose. While these techniques are interesting and perhaps worthy of study for those interested in writing fast and small code, students benefit more from learning the process of abstraction and design than from language- and platform-specific tricks.

The interdisciplinary nature of computer game design makes it difficult to incorporate into traditional computer science curricula. However, it would be a mistake to disregard the study of games out-of-hand whether due to this complexity or the traditional stigma that they are "just games." The key to success for integration into existing programs (as opposed to the generation of game-specific curricula) is to extract the computer science and software engineering aspects from games.

It is important to distinguish between *game design*, which is the process of developing a game idea [Salen and Zimmerman 2003], and *game programming*, which is the process of implementing game software [Gestwicki and Sun 2007]. The latter is clearly a domain of software engineering. There have been attempts to classify "design patterns" within game design [Bjork and Holopainen 2004], but these are distinct from design patterns for object-oriented software development. Since game programming using object-oriented techniques is an instance of object-oriented software engineering, one can reasonably expect the traditional patterns to be applicable.

1.3 Teaching with Games and Patterns

Using games to teach computer science is not a new idea. We have already mentioned some examples that use games to teach patterns, and there are many more. However, many of the examples one encounters in textbooks and on the Web take shortcuts in the name of clarity and simplicity. For example, in a survey of recent Java textbooks, every game example performs all of its rendering and logic processing on the event thread. This leads to simpler code, but it ignores the important programming idiom that the processing on the event thread should be kept to a minimum. In small-scale or turn-based games with little or no animation, the impact of this poor design decision is not apparent. However, this approach does not scale to action-oriented games, which must maintain good frame rates while being responsive to user input.

As an example, consider the *game loop*. Computer games are driven by a central loop that is similar to the event-processing loop in all modern interactive software. It can be summarized by the following pseudocode:

```
while the game is running
  update the game elements
  redraw the screen
```

This loop demonstrates *active rendering*, a technique in which the main game loop draws the screen directly. Many Java examples found both online and in print use *passive rendering*, where the `paint` or `paintComponent` method of each game element handles its own drawing, as in AWT or Swing applications. When using passive rendering, screen drawing is dependent upon the

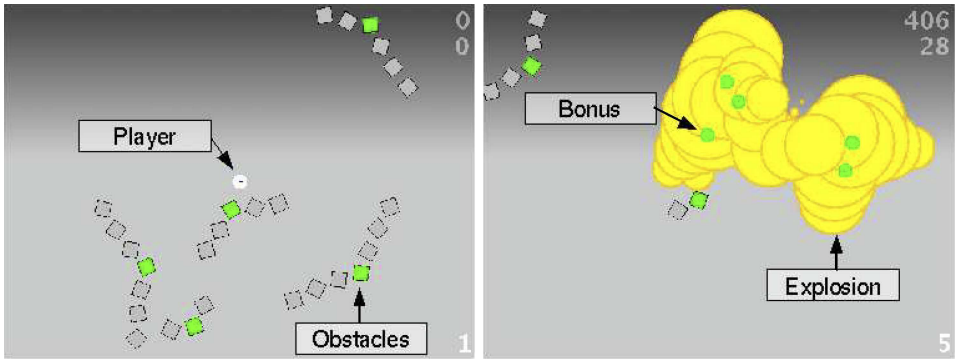

Fig. 1. Two annotated screenshots of EEClone. On the left, the player's sprite is seen in the middle of the screen, dodging the flying block obstacles. On the right, the player has triggered an explosion, and the chain reaction of explosions is evident. The circles emanating from the explosions are the bonus sprites that the player can collect for extra points.

event thread, and so there is no guarantee of frame rate, which leads to unsteady animations.

The pseudocode presented above is deceptively simple: the difficulty in game programming comes from balancing game logic processing and rendering within the game loop. In order for a game to achieve smooth animations, whether it is an action game or an animated puzzle game, the game loop needs to repeat as many as 70 to 90 times per second [Davison 2005]. This upper limit synchronizes the scene drawing with the refresh rate rate of modern display devices. A well-designed game engine will be able to maintain a steady frame rate by a combination of active rendering, double-buffering, and, when necessary, dropping frames.

1.4 Overview of EEClone

EEClone was developed as a case study to both explore and teach design patterns. Two screenshots are provided in Figure 1. It was inspired by Every Extend, a free game for Microsoft Windows published by Omega.[1] Commercial versions of the game have been recently released by Q Entertainment for PlayStation Portable and XBox Live Arcade. In the game, the player controls a bomb in a world filled with flying obstacles. If the player strikes an obstacle, a life is lost. If the player detonates the bomb, a life is still lost, but it also starts a chain reaction of explosions, which earns the player points and more lives. Some obstacles drop bonuses when destroyed, and these can be picked up for extra points. However, the amount of points needed to earn extra lives increases. A key aspect of the gameplay involves dodging obstacles until a large enough chain reaction can be created, all while the timer counts down. Part of what makes the game fun is this trade-off between risk and reward. EEClone is a simplified clone of Every Extend, borrowing its interface paradigm but

[1]http://nagoya.cool.ne.jp/o_mega

using simplified graphics and scoring. It was implemented in Java without using any native libraries.

EEClone was chosen as an example because the rules are simple, it is extensible, and it contains many of the features of more complex games. The game requires smooth animation, score keeping, responsive controls, music, sound effects, and collision processing. From its inception, EEClone was designed to be simpler than Every Extend but also extensible, so that interested students could build on the software to add new features. That is, it was designed specifically to support software reuse and extensibility, two oft-cited benefits of object-orientation and design patterns.

2. PATTERNS IN EECLONE

Several patterns are reified in EEClone. The game's software architecture is a collaboration of patterns, but for clarity, each pattern is first described in isolation. The method by which patterns are presented here mirrors the method used to discuss the patterns within our experiment: a discussion of the domain-specific problem leads to a justification for the design pattern. The software designs are presented using the Unified Modeling Language (UML) [Booch et al. 2005]. Design patterns are identified within class diagrams using the UML collaboration notation. This is an effective and semantically correct use of the notation and useful for highlighting the collaborative nature of design patterns.

2.1 Sprites: The State Pattern

A *sprite* is any non-background element in a game, such as players, enemies, and projectiles. In EEClone, the player, obstacle blocks, explosions, and heads-up display are all considered sprites by the game engine. Sprites lend themselves to object-oriented design: they have state and they exhibit behaviors. The state of a sprite, generally, includes its location, velocity, size, and image. The behaviors of sprites usually correspond to either user input for player sprites or collisions with the player for enemy sprites.

Sprites often exhibit different behaviors based on external or internal game information. For example, Pac-Man behaves differently after eating a Power Pellet than before. A state diagram for the player's sprite in EEClone is presented in Figure 2. The traditional implementation strategy, exhibited in every textbook surveyed, is to use integer constants to represent each possible state. Then, in each method whose behavior is state-dependent, the program uses `if` or `switch` statements to explicitly detect the current state and take the appropriate behavior. This procedural approach carries with it all the usual baggage: state-dependent logic is distributed throughout the code, making the addition of new states both awkward and error-prone.

The patterns-based alternative is the State pattern, where each state is represented by its own object. Figure 3 provides an architectural view of this approach. Each state is implemented in its own class, which encapsulates and localizes each state's behavior. All messages sent to the player sprite are delegated to the current state object. In Java, these states are implemented as

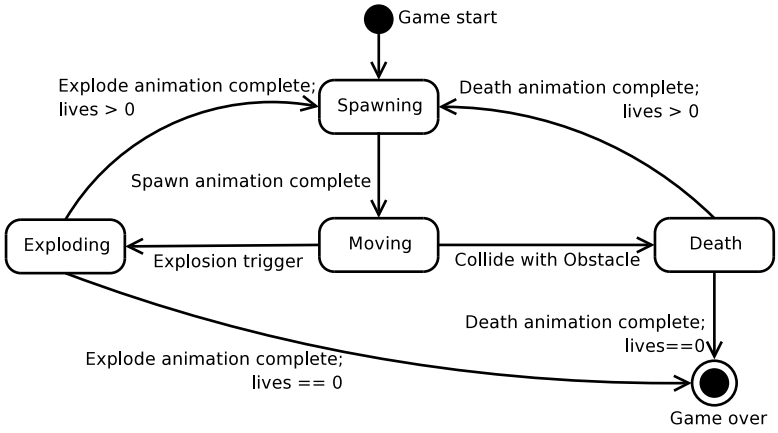

Fig. 2. State diagram for the EEClone player's sprite.

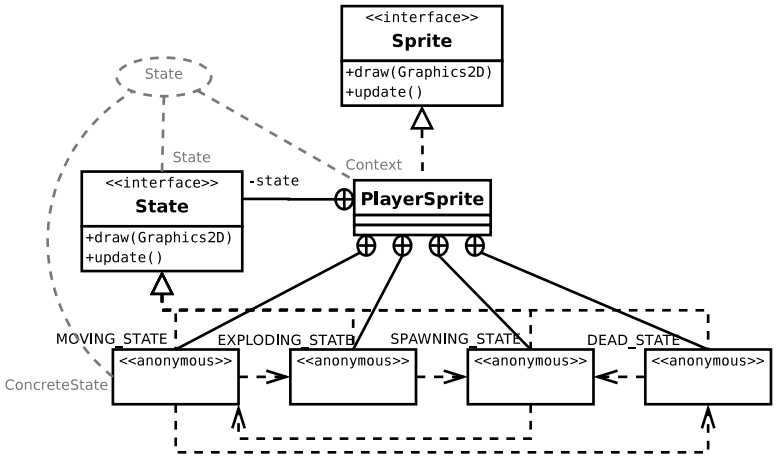

Fig. 3. The State pattern applied to player state representation. The player sprite can be in one of four states, each of which is implemented as an anonymous inner class of `PlayerSprite`.

private anonymous inner classes, which has the benefits of hiding the details of states within the containing class and also ensuring that there is only one instance of each state object. The relationship between a class and its inner classes is indicated using a bidirectional UML *containment* association. This is a relatively uncommon notation, but it is semantically appropriate for representing inner classes [Holub 2004].

This design incorporates an important yet easily overlooked property of the State pattern: states are responsible for their own transitions [Gamma et al. 1995]. The state transitions can be modeled as a directed graph, and each state is only aware of its neighbors.

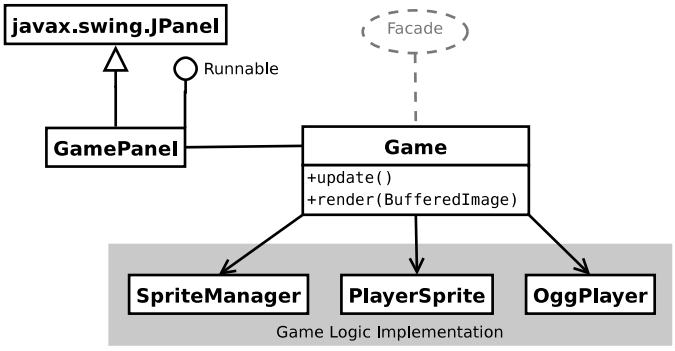

Fig. 4. The Facade pattern, which hides the details of the game implementation behind a single point of entry. The game logic implementation details are hidden from the client. A representative sample of classes within the game logic implementation are shown.

2.2 Logic vs. Presentation: The Facade Pattern

When programming in Java, one may deploy a game as an applet or as an application, and applications can use windowed mode, full-screen exclusive mode (FSEM), or "almost full-screen" mode [Davison 2005]. These decisions are related to the presentation and have only minor effect on the game logic. Also, there are situations when one would want to switch easily between presentations: debugging a game intended for FSEM can be easier in windowed mode, for example. Although the separation of concerns is clear, many code samples in print and on the web combine presentation and game logic into one class, presumably to simplify or shorten the code.

Figure 4 shows how the facade pattern can be used to hide the game logic behind the `Game` class. Using this design, the game's "wrapper" (a full-screen window, an applet, etc.) can be changed without affecting the game logic, and the game logic can be changed without affecting its wrapper. This design can be used to build a game architecture in which the game logic is a replaceable module [Nguyen and Wong 2002].

2.3 Controls: The Observer Pattern

The classic example of the observer pattern in Java uses the `java.awt.event` classes to process events. Listeners are registered with event producers, and events are fired and handled on the AWT-Event Thread. Our experience shows that students have little trouble understanding event-driven programming at this level. However, when there is nontrivial processing to be done in response to an event, the best practice is to push this processing onto a different thread. Many students fail to grasp this lesson, latching instead onto a naïve interpretation of the Observer pattern in which the observer reacts to the stimulus immediately and on the same thread.

Smooth animation and consistent, responsive input can be achieved with a more clever application of the Observer pattern. Java's standard libraries do not provide a platform-independent approach for polling the keyboard's state.

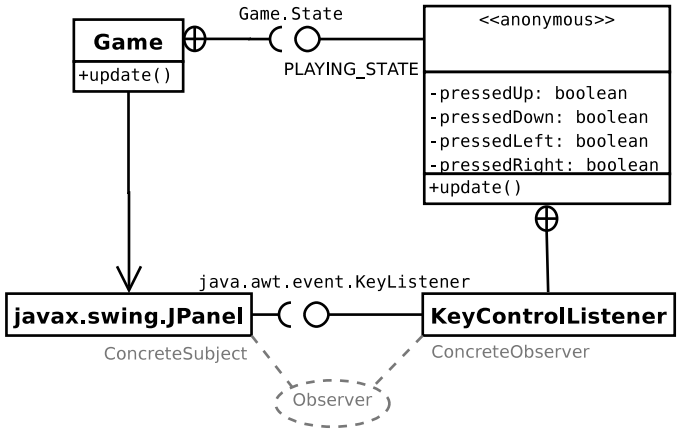

Fig. 5. The Observer pattern applied to asynchronous processing of keyboard input. The event thread posts the status of keys, which is read by each update in the main loop of the game thread.

One must use the AWT `KeyListener` to receive keyboard events unless linking with a native library such as JInput.[2] In order to maintain a smooth animation, the speed of a sprite must be defined with respect to the number of updates per second, not the number of key events per second. An elegant solution to this problem is for the key listener to write key state information to a shared object, and for the main game thread to read the state of this object during the update step, as illustrated in Figure 5. This demonstrates how the observer pattern can be used in a multithreaded environment, and this architecture can be applied in nongame applications as well. This is also an example of multithreaded processing that does not require synchronization: thread synchronization would introduce unnecessary overhead considering the frequency of updates and the atomicity of assignment.

2.4 Animation: The Strategy Pattern

In an object-oriented implementation of a game, sprites should be responsible for rendering themselves. One could combine the logic and presentation in a single class, but this would make it difficult to add or modify animations later. For example, one method of animation is to load a series of images and flip through them; another approach is to render each frame of an animation using graphics primitives. If the animation logic is mixed with the sprite's state logic, it would be very difficult to alter.

An alternative is the Strategy pattern. The architecture of such a solution involves an `Animation` interface, attaching animations to sprites at runtime, and delegating the rendering from the sprite to the animation object. This approach is illustrated in Figure 6, where a sprite's animation update and

[2]https://jinput.dev.java.net

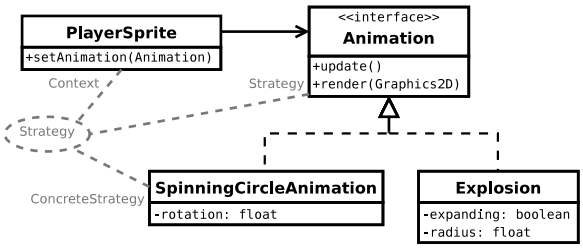

Fig. 6. The strategy pattern applied to sprite animation.

rendering are delegated to an animation strategy, identified by the `Animation` interface. When this design is combined with the state design of Figure 3, each state of the player sprite may have its own animation strategy, and the corresponding animation is only updated and rendered when its state is active. It is worth mentioning that the current version of EEClone uses Java2D graphics primitives, not image-based animations. This is intended as an entry point for students to modify the provided code.

2.5 Collisions: The Visitor Pattern

Games often contain a variety of elements that can collide with each other, and the result of the collision depends on the elements' types. Consider again the classic Pac-Man: our intrepid hero can collide with walls, ghosts, power pellets, dots, and fruit. Using a non-OO approach, the types of game elements can be encoded as integers and then selected via `switch`. The disadvantages of this approach are similar to those for the state problem discussed earlier: the logic for collisions is spread throughout the game, and there is no static verification that all collisions are considered. An alternative in Java is to use `instanceof` to obtain runtime type information, but this requires a rigid type hierarchy where polymorphic dispatch is preferable.

A pattern-based approach uses the Visitor design pattern. Many students balk at this pattern's polymorphic double-dispatch, but it is a natural fit for the problem of handling sprite collisions, where the behavior of each collision is dependent on the sprite's type. We return to the EEClone game for a simple example, since there are only four types of sprites: the player's sprite, explosion sprites, flying block (obstacle) sprites, and bonus sprites. Figure 7 shows how the pattern is applied with respect to the player sprite and flying block sprite implementations. Both of these classes have their own `collisionHandler`, which is itself a `Visitor`. The player sprite's collision handler contains its logic: if it collides with a bonus sprite, bonus points are earned, and if it collides with a flying block, a life is lost. The flying block sprite's collision handler watches for collisions with explosions, which result in the flying block's own explosion and an increase in the player's score. For both of these, collisions with other types of sprites do nothing, and so those `visit` implementations are empty. If special processing were required upon collision at a later date, it could easily be added.

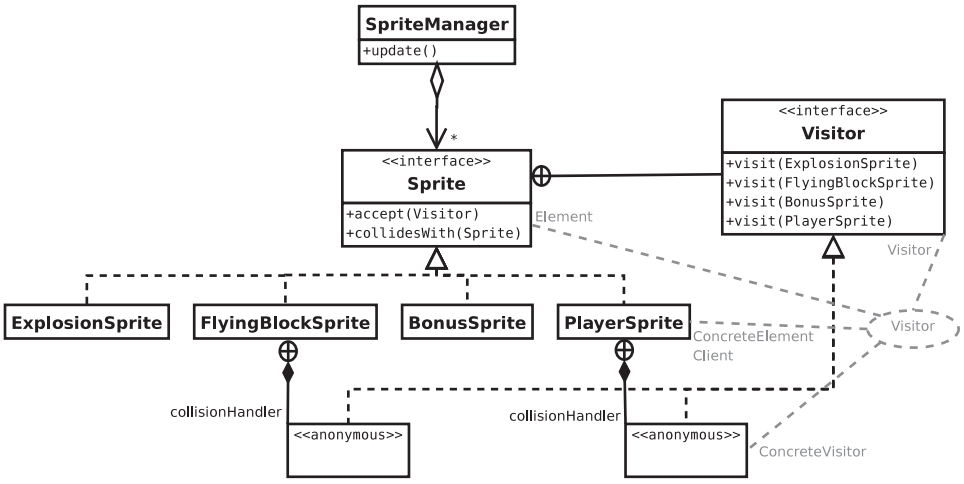

Fig. 7. The Visitor pattern applied to sprite collision processing. The `SpriteManager` checks for bounding box collisions and delegates the handling of collisions to each sprite's `collidesWith` method. The `Sprite` implementors serve as both Concrete Element and Client.

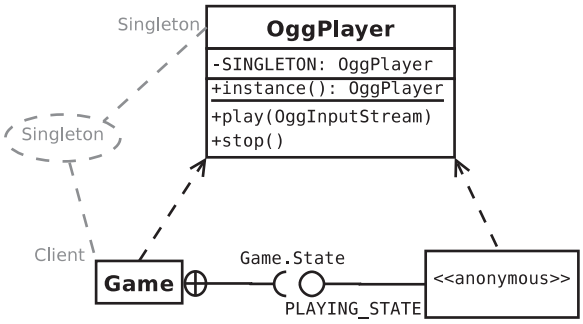

Fig. 8. The Singleton pattern allows for any object in EEClone to modify music playback.

2.6 Music Engine: The Singleton Pattern

The Singleton pattern is applied when an object is both global and unique for its type. The apparent simplicity of its intention and implementation leads to its abuse by amateur software designers. We assert that it should be discussed in a course on design patterns precisely because it is so easily abused; a failure to adequately explore this pattern will likely lead to its continued misuse.

In EEClone, a singleton is used for the object that manages music playback. This allows any object in the executable model to change the music that is currently played. The `OggPlayer` class is a singleton, as shown inbreak Figure 8. Although only the game states currently use this singleton, the pattern allows for extensions to the game to be able to change or stop the music as required.

When presenting this pattern, we observed that our students accepted this design without argument. The students were surprised to hear that another professional software designer had debated with us whether this is a good application of singleton or not. Our view is that it is reasonable to assume that there is one global entrypoint into the audio subsystem since the entire engine is based on the assumption that there is a single computer with a single set of speakers or headphones. However, one can imagine a case where a game must support different audio outputs, although our class could not immediately identify why one would want such a thing. This issue leads directly into an important discussion about the broad assumptions one must make when applying the Singleton pattern. It is important in a discussion of design to balance the desire for extensibility with the necessity of solving the problem at hand. In this case, we assert that a singleton is the simplest solution, since extending the engine to a multiheaded or multiuser system significantly changes the requirements and therefore the design of the entire game engine.

The use of the Ogg Vorbis audio format provides a convenient point to initiate discussion of intellectual property. Students tend to have some understanding of the various audio codecs available, but few appreciate the legal implications of codec selection. In this case, the choice of Ogg Vorbis was easy: it is free for perpetuity, and there are several tools available to convert audio into its format. Embedding the codec logic into EEClone optimizes the size of the application and provides the detailed source for students who are interested. However, a more robust game engine could easily make use of an existing audio engine such as OpenAL (http://openal.org).

3. TEACHING, LEARNING, AND ASSESSMENT

Our teaching philosophy is based on active learning principles, which have shown to be useful in teaching computer science [McConnell 1996; Briggs 2005] and particularly software design [Warren 2005; Gibson and O'Kelly 2005]. We engage students with a subject that is of interest to them so that they approach the problems proactively via individual exploration, peer collaboration, small group discussion, and other constructive means. To the students, game programming is both interesting and significant, and so students are motivated to succeed. The instructors serve as facilitators and guides in this process [Bonwell and Eison 1991; Wilkerson and Gijselaers 1996]. To assess the learning results from using these principles, we use formative assessments such as self- and peer-evaluation, a form of portfolio assessment, and presentations in addition to traditional summative assessments [Biggs 1999; Earl 2003].

Design patterns provide a set of best-practices for object-oriented design, having been discovered in successful software systems [Gamma et al. 1995]. One of the most important benefits of design patterns is that they facilitate communication: a potentially complicated software design can be concisely described in terms of interacting patterns. This identifies three roles for design patterns in education: as examplars of object-oriented design, as practical and useful solutions to software design problems, and as tools to communicate

about design. Based on these observations, we have defined the following learning objectives:

(1) The student understands object-oriented design and modeling.
(2) The student can identify and apply the specific design patterns addressed in this experiment.
(3) The student can effectively communicate about software design and design patterns.

We have developed a variety of assessment tools to measure students' progress towards achieving these goals. Specifically, these are a multimodal entry exam, peer evaluation, in-class presentations, project milestone deliverables, and an exit exam.

3.1 Entry Exams

The entry exam was designed to measure students' knowledge about design, including both declarative and procedural knowledge. It is insufficient for a student to only have declarative knowledge about software design since it is inherently a process. Similarly, it is almost useless for a student to be able to design if he or she cannot communicate about the design.

The students' declarative knowledge was assessed using a written exam, which was distributed on the first day of class and collected on the second. This instrument was designed to measure students' knowledge of fundamental object-oriented design principles and design patterns. The exam provided several topic areas, for which the student was asked to provide an estimation of self-efficacy on a five-point Likert scale. Students were also asked to provide an example, in C++ or Java, to demonstrate this understanding. The list of topics in the written portion of the entry exam were: object-oriented software design, inheritance, polymorphism, UML, design patterns, and the difference between procedural and object-oriented design. Four specific design patterns were also mentioned by name: Singleton, State, Strategy, and Visitor.

The results of the written exam met our expectation that the students had little or no knowledge of design patterns (by name). A surprising result was that students reported moderate understanding of principles of object-orientation. However, the examples provided by the students did not demonstrate any such understanding. Students "textbook definitions" were fairly accurate, but their sample code did not reify the concepts. There is a possible flaw in this instrument, since not much space was provided for the example code. It is important to note that there are competing factors: asking students to write code on paper tends to produce small, non-representative blocks of code, while requiring students to submit full applications requires much more time. Additionally, it may be difficult, when following the latter approach, to determine if the program is merely a copy of an assignment from a previous class that explicitly demonstrates the concept.

The shortcomings of the written portion of the exam were foreseen and addressed by complementing it with an interactive exam. In the second class

meeting, the students were presented with four different scenarios and asked to design software solutions for them. Since we had a small number of students, the entire group collaborated in the process; with a larger class, subsets of students could be formed. Both instructors observed the process without interfering, which allowed us to assess students' understanding and communication skills. Students' performance was recorded by marking each student's responses in a variety of categories, including aspects of object-orientation and design patterns. Responses were classified on a scale ranging from strong understanding to misunderstanding.

Each of the scenarios from the interactive exam are presented below, along with commentary regarding the intention of the scenario and our students' reactions.

Scenario 1: In Pac-Man, those who seek world-class high scores know that each of the four ghosts has its own behavior. How would you design a software architecture that supports different AI routines for each of Pac-Man's ghostly enemies?

This scenario was primarily designed to determine if students understand delegation, polymorphism, and the Strategy design pattern. Even though our students did not study this pattern explicitly, their responses indicated that they understood the benefits of a delegation model over an inheritance model. That is, no one recommended separate classes for each ghost, but rather a single ghost class that delegates its decisions (artificial intelligence) to a dependent class.

Scenario 2: At the start of a game of EEClone, when a bomb "spawns," the player is invulnerable for a few seconds. During that time, the bomb cannot be exploded, and neither can it collide with obstacles, although the player can collect bonuses. After the bomb explodes and a brief a pause, a new bomb spawns as before until the player is out of lives. If the bomb is struck by an obstacle before it explodes, then it is "dead" for a few seconds before a new one spawns. While dead, the bomb cannot collide with obstacles or bonuses, nor can it be moved. Describe a software architecture that manages the many states of the bomb and its varying behaviors.

This scenario is designed to assess students' understanding of stateful systems and the State design pattern. The one graduating senior in the class had previously encountered state diagrams in a course on programming languages, and he immediately saw a connection to this formalism. The students collaborated on a decent state diagram for EEClone, and the rest of the class learned the fundamentals of state modeling from the one student with this knowledge. However, while discussing the implementation design, the students unanimously agreed to use integer constants to represent the states and declared the analysis complete. We returned to this discussion later in the semester when discussing the state design pattern, and at that point, students were able to understand the shortcomings of their original design.

Scenario 3: In EEClone, the effect of a collision between two sprites depends on the types of the sprites. For example, the bomb's hitting an obstacle is different than an explosion's hitting an obstacle. It is desirable to add more types of

sprites in future version of the game; these may include time bombs and accelerating power-ups. How would you design a software architecture that correctly handles collisions between sprites, keeping in mind that we want to add more types of sprites in the future?

This scenario deals with the problem of runtime type recovery: choosing the type-dependent instructions during program execution. This is most elegantly addressed with the Visitor design pattern, but the students had a hard time coming up with a consensus on a good solution. Students with some experience in Java knew of the `instanceof` keyword, and this was the best solution developed.

Scenario 4: Java does not allow polling of the keyboard: that is, you cannot query what keys have been pressed on the keyboard. The only guaranteed way to get keyboard input in Java is to attach a `KeyListener` to your application, and then Java will call the `keyPressed` method of the listener whenever a key is pressed. Usually then, you have one thread that monitors the keyboard (the AWT-Event thread) while another thread handles updating and rendering the game state (the "main" thread). We do not know how many keys a user will press between update-and-render cycles. Describe a multithreaded software architecture that manages keyboard input without polling. For example, consider the input controls for EEClone: arrow keys to move, spacebar to explode.

This scenario is not intended to demonstrate any particular design pattern but rather to illustrate a common problem in game engines: synchronizing user activity with program logic. None of our students had significant experience with multithreaded systems. Although they understood the definition of a thread and the concept of a producer/consumer model, the groups were unable to accurately model or communicate about a solution.

In summary, our students' responses showed some understanding of object-oriented design, but there was very little clarity. Some students referred explicitly to function pointers and bit flags, suggesting to us that the student had some knowledge of the status quo for game development resources, which tend against object-oriented design in favor of C-style procedural solutions.

3.2 In-Class Presentations

During the semester, students were required to give several in-class presentations of their progress. At the beginning of the semester, students developed *concept documents*. These were used to define the scope of students' projects. In the games industry, concept documents are used to provide an overview of a game along with economic and marketing factors, although we omitted the latter. Students were not required to make games that would sell; rather, they were allowed to design any game that they would find interesting and that was within the scope of one semester's work. Both interest and scale were discussed with peers and instructors during class. While this activity could have been done individually in order to define the scope, we found that involving the entire group in discussions of game design was a useful team-building activity. Students learned more about each other and, from our observations, valued each others' opinions. It should be noted that very little time was spent on

Week	Topics	Deliverable
1	Concept documents and entry exam	
2	Review of UML and OOD; State and Facade patterns	Concept documents and peer evaluation
3	Strategy pattern; Java WebStart	Revised concept document
4	Milestone presentations; Observer pattern	Milestone 1
5	Major status reports	
6	Milestone presentations; Visitor pattern	Milestone 2
7	Collision detection; minor status reports	
8	Milestone presentations; Singleton pattern	Milestone 3
9	Major status reports	
10	Milestone presentations; thread pooling and sound effects	Milestone 4
11	Music playback; minor status reports	
12	Milestone presentations; ludology and theories of fun	Milestone 5
13	Major status reports	
14	Conclusions and exit exam	Milestone 6
15	Final presentations	

Fig. 9. Course Schedule for a 15-week Semester

theories of game design, a separate field of inquiry [Salen and Zimmerman 2003]. Ours was a non-academic treatment of game design, leaving decisions up to our students, who seemed to have significant experience with various games and genres. The concept documents were finalized by the second week of the semester, which coincides with the end of a two-week review of Java, object-orientation, and UML. For some of the students, this was the first experience with all of these topics.

For the remainder of the semester, the students were required to give periodic presentations of their progress both through status reports and milestones; the semester's schedule of topics and presentations is provided in Figure 9. The status reports were divided into *major* and *minor* classifications, corresponding to ten- or twenty-minute presentations, respectively; this classification was an artifact of the course schedule and number of participants. During milestone presentations, students were required to demonstrate their program and explain the major changes to the software design and implementation. These explanations clarified students' usage of design patterns. The presentations were informal, akin to "stand-up meetings" in agile development processes, and they served as assessment instruments for the instructors to measure students' progress in both technical and theoretic issues. Specifically, the instructors were able to measure and direct students' progress towards the three primary learning objectives for this course.

The milestones were originally designed to require students to apply the patterns previously discussed in class. For example, the first milestone required the application of the State pattern. The sequence of patterns in the milestones was based on the development of EEClone. However, the students' projects each evolved in different directions, even those within the same genre. This trend was obvious to the instructors due to the frequency of

student presentations, and so the milestone requirements were changed from pattern-specific features to simply demonstrable progress towards completion. This teaching technique is similar to the Scrum model for managing student projects, in which the students are guided towards setting realistic goals and regularly reporting on their progress [Sanders 2007].

3.3 Observations

Students' comprehension of design patterns and design principles followed the same pattern as the instructors' delivery of the material. For each pattern and core concept, the class first explored the nature of the problem, such as state-based games (addressed by the State pattern) or the desire for online publishing (addressed by Java WebStart). The problems were then framed in the context of EEClone. Traditional, naïve, and non-patterns-based approaches were briefly discussed, followed by an introduction to the pattern's behavior, structure, and implementation. Finally, patterns and concept integration themes were discussed. The stages of students' understanding can be classified into four stages as follows:

(1) **Problem understanding.** Students first tried to understand the nature of the problem itself. For example, all the students were familiar with games that have a menu, a playable game, and a high-score list, but they had not previously formalized the idea of state transitions.
(2) **Structural copying.** After the pattern motivation, behavior, and structure were discussed in class, the students tended to copy the implementation of the pattern from EEClone directly into their own projects. Several times, students mentioned in status reports or milestone presentations that they had successfully used the pattern, although they admitted not understanding it. We observed that this stage lasted approximately one week.
(3) **Customization.** Once the students had working software that integrated the EEClone pattern implementations, they proceeded to customize the implementation to match their game engine designs. This stage lasted approximately two weeks as the students bridged the gap between the problem, the pattern, and the implementation.
(4) **Ownership.** Approximately three weeks after presenting a pattern, including several student presentations and clarifications by the instructors, students showed a distinct ownership of the patterns. This was demonstrated through clarity of communication about the patterns independently of particular problems or implementations.

It is of utmost importance to this approach that students be guided from the second step onwards. The point of design patterns is not the patterns themselves but the problems they solve. It is important that students understand the problem prior to seeing the pattern, and furthermore, that the students return to focusing on the problem after the pattern is implemented.

3.4 Exit Exam

The formative assessment during the semester allowed the instructors to measure and guide students toward the three primary learning objectives for this experiment, and we have explained the process by which students gain ownership over a design pattern. However, mastery of individual patterns is neither necessary nor sufficient to show a mastery of software design in the large. Although we observed that students were successful in integrating some patterns, an end-of-semester exit exam was necessary in order to assess students' understanding of the more holistic aspects of design and patterns.

The exit exam consisted of six essay questions. It was given as a "take-home exam," and the students had two days to complete it. The students were informed that they would be graded on the clarity of their responses, not necessarily the content. Grading the exam was deemed necessary in order to ensure students' full participation. The six assessment questions on this exit exam are presented below along with a brief discussion of their motivation and results.

Question 1. Think back to the beginning of the semester and your original design for your game and software. Since then, we have explored many ideas of software design and game design. If you were to start your project again, would you do anything differently with respect to your software design plans? What about your game design plans?

Our students' responses reflected an increased appreciation for the design phase of software engineering. All of them mentioned that they would have used more formal techniques such as state, static, and dynamic models. This indicates that the students have met our first learning objective.

Question 2. What relationships do you see between the game design and software design processes?

This question is founded in some of our earlier work on the relationships between software design and game design [Gestwicki and Sun 2007]. All of our students claimed that game and software design are intrinsically linked, although this was justified by intuition rather than a scientific approach to design. However, the responses did demonstrate that the students were able to identify the differences between game design and software design, which was not clear from their entry exams.

Question 3. Suppose you are a game software designer working with game designers. What kinds of documents or other communication would you need from a design team in order to implement a game idea?

Question 4. Suppose you are a software designer who is leading a team of programmers. What kind of design document would you give to your team to ensure that they implement the design correctly?

These two questions relate to the third learning objective, and the students' responses indicate greatly improved design communication skills. In contrast with the entry exam, where students struggled to communicate about both game and software design, the exit exam shows that the students understand how to effectively communicate between different stakeholders for such a software development process.

Question 5. Focusing on your game engine, could you produce another game out of it? That is, could you write another game that reuses a majority of the code you have written? Would it necessarily be similar to your current game, or could you produce a vastly different game from your game engine? Could you have designed your game engine differently to make it more reusable?

Our students' responses advocated for both reusability and extensibility, two important aspects of object-oriented software design. They also advocated for more explicit use and identification of patterns within the game engine, and this demonstrates that students have met the first two learning objectives.

Question 6. Imagine you were to develop a new game in a different genre. What lessons from this semester would help your designing or developing a new game?

The students responded with claims and examples of the applicability of design patterns to other game genres. As with the previous questions, the students strongly advocated pattern-based, object-oriented software analysis after having participated in this experiment. This indicates that the students have met the first and second learning objectives.

As a formative assessment tool, the exit exam revealed a profound increase in students' understanding of software design and design patterns. Additionally, the students were able to clearly identify several salient properties of the design process in their responses to the third and fourth questions. These responses stand in contrast to the students' concept documents, their first work of the semester, which contained many vague and conflicting ideas.

4. CONCLUSIONS AND FUTURE WORK

Our formative assessment tools suggest that our approach helps students meet the learning objectives, although larger-scale implementations of this process are required in order to gather sufficient data for statistically-significant summative evaluation. Specifically, we are developing assessment techniques to verify that the four-stage method by which students learn design patterns is accurate and reproducible. However, it is possible that our methodology does not scale well to larger class sizes. Future work will involve formal separation of students' roles as game and software designers.

Our students played many "design" roles within this experiment: they were game designers, graphic designers, interface designers, and software designers. This introduces a significant complication in teaching game and software design. For some of the projects, if the student was unable to correctly design part of the software, the student changed the game design to match the software. In practice, it is rare for one person to have so many design roles, and it would be unacceptable for a software engineer to dictate changes in game design. However, we allowed this practice for our experiment since our students did not have any formal training in game design, and many of the changes were coincidentally beneficial to the game design. One of our future projects involves forming interdisciplinary teams across two courses, one on game design (for non-CS majors) and one on software design (for CS majors).

This structure should better simulate a professional game development experience, and we plan to adopt established practices of agile development and team management into the course.

Design is the process of considering alternatives, and good software engineers must be able to weigh each option on its merits. The application of patterns in EEClone can be used as a springboard for classroom discussion. Consider again the state pattern as applied to sprites, presented in Section 2.1: the resulting executable may be nominally faster if integer constants are used rather than indirection through state objects. The advantage of the state pattern becomes clear when adding new states or maintaining the software. Students should be encouraged to experiment with these conflicting concerns and develop for themselves an appreciation for patterns; as with problem-based learning, this usually results in students taking more ownership of the material than when it is only presented in lectures [Barg et al. 2000].

We have used EEClone to explore how the State, Facade, Observer, Strategy, Visitor, and Singleton patterns are reified in game software design and implementation. The case for these patterns was found to be particularly compelling, but this is not meant to be an exhaustive list. Other games and genres engender other specific patterns. For example, one of our students identified the Decorator pattern as being applicable in a sprite animation library: the "power-ups" collected by a spaceship can decorate the drawing of the ship, eliminating the need for an unmanageable number of customized sprite images. Future work includes investigation of patterns that are intrinsic to various types and genres of games, including architectural design patterns [Buschmann et al. 1996].

No formal longitudinal assessment has been conducted or has been planned for the students involved in this experiment. The number of participants is unfortunately too small to yield significant statistics, but we have remained in contact with most of the students. One of the students is working on a large-scale project to develop games for science education. He adapted most of the design patterns we studied into C++ and Flash, and he claims that these have made his team's implementation much more manageable. Another student is now taking a software engineering course and reports that he is able to lead his team's development by showing them design patterns in action prior to their being discussed in lecture. These anecdotes suggest that the software design lessons learned in this course are significant and have lasting impact on the students' software design processes.

A playable version EEClone, its reference implementation, and selected projects from our research group are available from http://www.cs.bsu.edu/~pvg/games.

REFERENCES

BARG, M., FEKETE, A., GREENING, T., HOLLANDS, O., KAY, J., KINGSTON, J. H., AND CRAWFORD, K. 2000. Problem-based learning for foundations computer science courses. *Comp. Sci. Educ. 10,* 2, 1–20.

BIGGS, J. 1999. *Teaching for Quality Learning at University*. Society for Research into Higher Education and Open University Press.

BJORK, S. AND HOLOPAINEN, J. 2004. *Patterns in Game Design*. Charles River Media, Hingham, MA.

BONWELL, C. C. AND EISON, J. A. 1991. *Active Learning: Creating Excitement in the Classroom*. ASHE-ERIC Higher Education Report No. 1, Washington, D.C.: The George Washington University, School of Education and Human Development.

BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 2005. *The Unified Modeling Language User Guide*, 2nd ed. Addison-Wesley Professional.

BRIGGS, T. 2005. Techniques for active learning in CS courses. *J. Comput. Sci. Coll. 21,* 2, 156–165.

BUSCHMANN, F., MEUNIER, R., ROHNERT, H., SOMMERLAD, P., AND STAL, M. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. John Wiley and Sons.

CLAYPOOL, K. AND CLAYPOOL, M. 2005. Teaching software engineering through game design. In *Proceedings of the 10th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'05)*. ACM Press, New York, NY, 123–127.

DAVISON, A. 2005. *Killer Game Programming in Java*. O'Reilly Media, Inc.

DECKER, A. 2003. A tale of two paradigms. *J. Comput. Sci. Coll. 19,* 2, 238–246.

DEWAN, P. 2005. Teaching inter-object design patterns to freshmen. In *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'05)*. ACM Press, New York, NY, 482–486.

EARL, L. M. 2003. *Assessment As Learning*. Corwin Press, Inc.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.

GESTWICKI, P. AND JAYARAMAN, B. 2005. Methodology and architecture of jive. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis'05)*. ACM Press, New York, NY, 95–104.

GESTWICKI, P. AND SUN, F. 2007. Game software and the design process. In *Proceedings of the Design Science Research in Information Systems and Technology Conference (DESRIST'07)*. 362–378.

GESTWICKI, P. V. 2007. Computer games as motivation for design patterns. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07)*. ACM Press, New York, NY, 233–237.

GIBSON, J. P. AND O'KELLY, J. 2005. Software engineering as a model of understanding for learning and problem solving. In *Proceedings of the International Workshop on Computing Education Research (ICER'05)*. ACM Press, New York, NY, 87–97.

HANSEN, S. 2004. The game of set: an ideal example for introducing polymorphism and design patterns. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'04)*. ACM Press, New York, NY, 110–114.

HOLUB, A. 2004. *Holub on Patterns: Learning Design Patterns by Looking at Code*. APress.

MCCONNELL, J. J. 1996. Active learning and its use in computer science. In *Proceedings of the 1st Conference on Integrating Technology into Computer Science Education (ITiCSE'96)*. ACM Press, New York, NY, 52–54.

NGUYEN, D. Z. AND WONG, S. B. 2002. Design patterns for games. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education (SIGCSE'02)*. ACM Press, New York, NY, 126–130.

SALEN, K. AND ZIMMERMAN, E. 2003. *Rules of Play: Game Design Fundamentals*. MIT Press.

SANDERS, D. 2007. Using scrum to manage student projects. *J. Comput. Sci. Coll. 23,* 1, 79–79.

WARREN, I. 2005. Teaching patterns and software design. In *Proceedings of the 7th Australasian Conference on Computing Education (ACE'05)*. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 39–49.

WICK, M. R. 2005. Teaching design patterns in cs1: a closed laboratory sequence based on the game of life. In *Proceedings of the 36th Technical Symposium on Computer Science Education (SIGCSE'05)*. ACM Press, New York, NY, 487–491.

WILKERSON, L. AND GIJSELAERS, W. H. 1996. *Bring Problem-Based Learning to Higher Education: Theory and Practice*. Number 68, Winter 1996, Jossey-Bass Publishers, San Francisco, CA.

Received June 2007; revised October 2007; accepted December 2007