# Exploring Game Architecture Best-Practices with Classic Space Invaders

Ed Keenan
DePaul University
243 South Wabash
Chicago, IL, USA
ekeenan2@cs.depaul.edu

Adam Steele
DePaul University
243 South Wabash
Chicago, IL, USA
asteele@cs.depaul.edu

## ABSTRACT

The classic arcade game Space Invaders provides an ideal environment for students to learn about best practices in game software architectures. We discuss the challenges of creating a good game architecture, and show how our problem space is an ideal environment in which to experiment with the challenges and tradeoffs inherent in any software design. We discuss in detail how each student created and engineered their game using good architectural design principles in general and gang-of-four design patterns in particular.

## Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures-*Patterns,* K.3.2 [**Computing Milieux**]: Computer and Information Science Education - *Computer science education*, D.3.3 [**Programming Languages**]: Language Constructs and Features - *Frameworks*, I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems - *Games*

## General Terms

Algorithms, Documentation, Design

## Keywords

Software Architecture, Design Patterns, Education, Framework, Games Engine, Software Engineering

## 1. INTRODUCTION

In this paper we discuss how the classic arcade game Space Invaders[9] can be used to highlight the challenges and choices that need to be addressed by game software architect students attempting to recreate the game with modern software engineering best-practices.

Traditional game programming is about wringing the maximum performance from the underlying hardware, and this coupled with the tight-time constraints for delivering computer games means that in many cases games are delivered with a codebase that does not conform to the precepts of good software engineering[7] In the SE456 – Game Architecture course at DePaul University's School of Computing and Digital Media (CDM), we use the traditional Space Invaders arcade game as a means to explore how architectural decisions impact the engineering characteristics of a software game system.

We first examine the problems with the software architecture of many computer games, problems that are not uncommon in other production software systems. We then discuss in detail how the students in the Game Architecture course implemented Space Invaders using a number of pre-written components and modules using design patterns as micro-architectures in order to create a high quality high-level architecture[3].

## 2. ARCHITECTURE BEST-PRACTICES

### 2.1 Problems with Game Architectures

Based on previous experiences with game development classes, when students are left to their own devices they often implement game systems as a monolithic code base with very little architectural structure. In addition, the students also tend to develop the software in a short-term fashion, focusing on the next feature at hand without concern for the overall design of the system.

The architectural problems of the student's systems are the same ones that many software systems fall prey to, in that their components are highly *coupled* (*i.e.* dependent) on each other. This gives rise to systems that are *fragile*, where small changes in one part of the codebase affect other parts in non-obvious ways. It also results in *brittle* systems that discourage programmers from making minor modifications and/or refactoring the code to increase its quality.

This is a problem, since in many cases a game's play can't be evaluated for the fun-factor until the feature is prototyped and implemented. If the underlying system is brittle, adding a new feature may break the existing system, and the students approach to change is driven by the need to minimize the impact of the new modified code, rather than to modify the game to produce a better play experience.

The need to apply software engineering principles such as modular and orthogonal underlying systems is critical to allow for continuous game play testing during development. Continuous refactoring of systems to explore new game play features and to preserve the decoupling of system is needed to produce game systems that are high quality from both the game play and software engineering perspectives.

A flexible system, in which changes can be made easily allows the system to be redesigned as necessary, and doesn't lock the development team into poor legacy decisions. Many game engines, in particular, are examples of brittle systems because much of their functionality is hard-wired and depends on magic numbers and other critical information scattered throughout the code so they don't scale well[10].

Furthermore, the lack of modularity limits the opportunities for distributed development and sophisticated versioning. This also means that unit testing has limited utility because so much of the systems behavior can only be tested at the system-wide level. An unappreciated problem is that a messy, convoluted architecture gives rise to documentation that is equally confused and hard to comprehend.

The problems encountered in designing and modifying game architectures mirrors that of many other software systems, and as such is an ideal environment for teaching students best practices in software architecture design, and general software engineering principles.

## 2.2 Teaching a Game Architecture Course

Teaching a Game Architecture course is challenging because unless you are careful it is possible to get bogged down in the low-level minutiae of the graphic system, sound system, the memory system or the file system. As such, we chose to provide the students with the basic framework that is described and provided in "Killer Game Programming in Java"[2].

This leaves us free to focus on the middle and high-level architectural concerns. This is also a common pattern in Industry where most of the low-level functionality is provided by commercial or proprietary frameworks.

In particular, we challenge our students to be able to connect and coordinate the many low-level systems flexibly within the higher-level architecture; how to avoid the problem of having game logic seep into the design of the overall architecture and that of the low-level system; and finally, how to engineer a software system that is robust and data driven[4].

We have taken a fairly radical approach to teaching game architecture, in that we use a traditional game and have them re-engineer it, rather than have them create a new game from scratch. We have found this to be a productive approach as the students can focus on the software engineering problems rather than the challenges associated with designing a new game. Even though Space Invaders is a classic arcade game there is a large amount of reference information available, for example there are a number of YouTube videos that provide a reference specification for the game's behavior[1]. Furthermore the design of the invader sprites is commonly available.

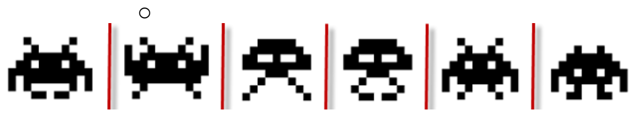

Figure 1. Invader Sprites

The original Space Invaders game was designed and programmed by Toshihiro Nishikado for Taito, Japan in 1978[9].

The game was originally programmed in assembly language on the Intel 8080 CPU. The design of program was built around a main with branching logic to execute the appropriate subroutines. The difficulty in maintaining the 60Hz graphics update rates meant that the original game was extremely tightly coupled, and was not at all modifiable.

[1] http://www.youtube.com/watch?v=VP2T3YlTDG8

## 2.3 The Course Project

During the first part of the class there were 12 students in the class and each of them was assigned to develop one of six different components: *creation of alien sprites*, *movement of the grid of alien sprites*, *shield collision and display system*, *missile collision system*, *sprite animation system* and the *sound system*. Each student was responsible for doing a modern design of their component, e.g. UML diagrams, use cases, design documents and a stand-alone implementation. The students were required to use one or more Gang-of-Four[5] design patterns in their implementation.

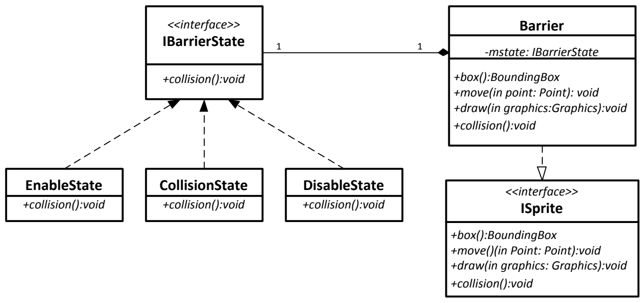

**Figure 2. Partial UML Diagram of the Collision System.**

Students were free to design and implement each component as they see fit, the only requirement is that they use design patterns where appropriate. Since there are many ways to solve and architect a problem, the students' designs varied greatly. The students were able to view and critique all of their fellow student's designs and documentation. Each component was built with the following design responsibilities:

| | |
|---|---|
| ***Creation of the Alien Sprites*** | • Creation of the grid and reuse of similar images to reduce the number of images resident in memory |
| ***Movement of the Grid of Alien Sprites*** | • Determine how to manage and move the grid as a collective<br>• Updating each individual movement to appear uniform<br>• Collision of the grid with the screen boundaries<br>• Changing the speed of descent of the aliens and the movement of the rows |
| ***Shield Collision and Display System*** | • Determine how to have the shield be eroded by missiles from the aliens<br>• Determine how the shield be eroded by missiles from the player<br>• Shield as a protector from alien missiles<br>• Determine the underlining collision grid and update mechanism |
| ***Collision system*** | • Create a collision system that determines collisions between the missiles and game objects,<br>• such as the aliens, player ship, UFOs<br>• The collision system should be able to determine if any missile hit the collision box and intersected the missile box with the target box<br>• Determine the state of the collision (non-intersecting or intersecting) |

| | |
|---|---|
| ***Sprite/Animation system*** | • Create a sprite system that displays the sprite image<br>• Animates the series of images of sprite images<br>• Ability to reuse sprite images instead having duplicate images loaded at the same time.<br>• Should be general enough to display any sprite used in the game, such as alien ships, shields, player ship, UFOs and missiles |
| ***Sound system*** | • Create a system that loads and plays static game wavs<br>• Allow other systems to call single of sounds for their respective effects<br>• Ability to control individual volumes of each sound playing<br>• Allow many different sounds to be playing at once, i.e. multiple overlapping explosions<br>• Mute and overall volume |

**Figure 3.Component Design Responsibilities**

Using the components Students were responsible for creating a completed game that mimicked the original arcade game as faithfully as possible. This included the cycling and transitions between the **Select Screen**, **Enter Game** and **Game Over** screens. The game was required to be played on several levels, the high score was track, one or two players, and keep track of credits[1].

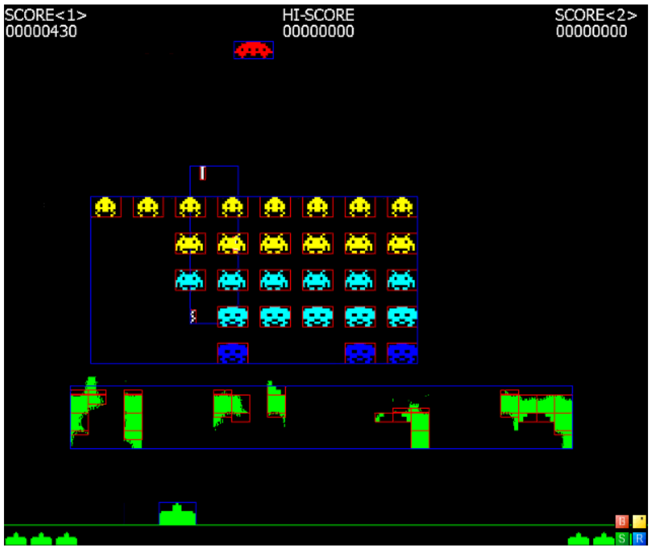

**Figure 4. Game with Debug Collision Boxes**

The students could freely use any part of the components that the other students developed, including their source code. This didn't deter from the challenge of the assignment, as everyone's implementation of their components varied so greatly. Any reused components from other students needed to be significantly refactored to be incorporated into a student's game. Therefore the original component design served more as a reference design.

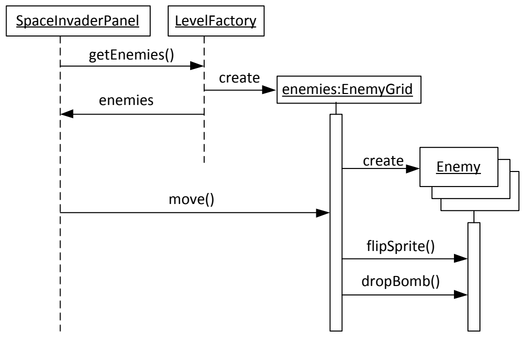

**Figure 5. Sequence Diagram of Enemy Interactions**

Throughout this process, there were three fundamental goals of our new version of the Space Invaders. The new architectural design had to have its components decoupled (orthogonal) from each other; any feature needed the ability to be scalable in quantity, and the components had to define as much the game behavior through its objects' data (data driven architecture)[8].

By focusing on these goals and debating and analyzing the components of the game, several design patterns [5] emerged in common use:

***Singleton*** – Global singletons were used to control player's state, number of lives, points. Similar structures were used to hold the state of the bombs, player's missiles and the controlling data for the collision system and timers.

***Composite*** – The collection of the aliens moving as one group. The groups of aliens were broken into a hierarchy of columns, allowing the group to move uniformly and collide as a collection against the extremes of the screen.

***Factory*** – Create the aliens objects, based on type of aliens, its location, collision object and sprite data. Also factory pattern for the shields.

***Flyweight*** – Easy way to replicate sprites with similar attributes (e.g. collision and textures), instead of creating many instances of fully formed data.

***Strategy*** – Missile behave differently depending on whether their targets are shields, aliens, UFOs or off screen.

***Observer*** – Call back mechanism for collisions and explosion effects of the aliens, and I/O from the keyboard controller.

***Command*** – Parameterization of the collision behavior depending on what type of object was hit. Difference between shields, alien, alien's bomb, UFO, off screen, and ground.

***Iterator*** – Used in the collision system to collide with moving objects on the screen. Moving object versus static objects, broken into hierarchy layers to minimize the number of active checks.

***Null Object*** – Insures unified behavior on all objects, whether they are specialized or not, thereby increasing the robustness of the design.

***State*** – State of the missiles, individual aliens, and shields, including the transitions effect of explosions.

***Memento*** - Switching between player 1 and player 2 as to preserve the same state of the two games.

## 2.4 Shield System Component

In this section we provide an analysis the student shield system. Each shield is constructed of collection of smaller barriers. These smaller units are responsible for collision detection, and each unit keeps track of the state of the shield. The shields slowly erode away with each collision until the division is disabled. At this point, the unit is not included in any future collision detection.

- The shield manager contains the list of shields (these are the units).
- The shield contains the shield state (this determines the erosion of the unit).

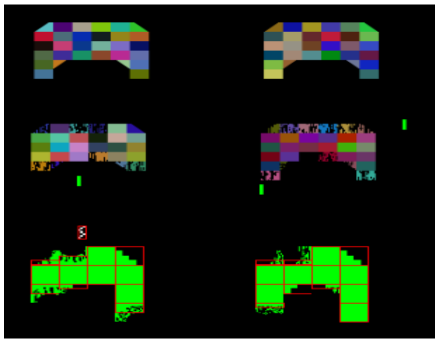

**Figure 6. Evolution of Shield System**

The erosion system in the collision module went through several iterations. Initially the shield was subdivided into rectangular grids. Random colors indicate the division positions (top image) in Figure 5. Each division would have random deterioration of the grids, until they took enough damage to be eliminated.

This approach created an issue in a region where there was collision hit was sufficient enough to create complete destruction. One would expect the neighboring areas to have some damage as well, simulating an explosion as the missile or bomb hits the shield creating a crater effect.

To imitate this crater damage effect, the semi-random deterioration of the shield needs to be in a circular density pattern. More damage towards the center of the explosion with a spread of less damage further away from the center. This is effect was constructed by random pixelation within an outlying circle with different densities. These circles of random damage spread across the neighboring grids showing damage.

To insure that only damage to the specific grid that had a direct hit accumulates damage. The grid size of each subsequent hit reduces the size of the collision box in three steps before the fourth hit completely removes the grid. The neighboring grids are not effect by this subdivision, but do show random deterioration by the crater effect.

Similar problem solving experiences were achieved the students developing the other components of the game. They focused on reverse engineering the game by reviewing the reference videos. As they were able to solve the problem, they refactored their solutions to better use design patterns in their designs.

## 3. STUDENT EXPERIENCE

Students' final submission included an updated design document with critical systems diagrammed in UML. A reflection paper describing their experience on the project was also submitted, including their development blogs[6]. Reflection paper asked several questions: what was the most difficult task, description and insights on the design patterns used, view point on the software complexities and issues of video game development, and lessons gained in this class.

Most of the students weren't aware of the complexities of game software development. Extrapolating on their experience for this small classic arcade game, yielded better appreciation and respect to the modern games of this generation.

Most students now see a direct correlation between the use and need of Deign Patterns to help organize, structure, and implement complex systems. They stated that they would continue to look for Design Pattern uses in future software development independent of the domain (game or general purpose).

Students also reflected about the nature of software development, stating that there numerous ways to solve a problem. They understood their tradeoff for each design decision, both in the complexity and efficiency. The scheduling and planning were also constant companions on their journey, gaining insights on the different ways to organize and execute their respective projects.

## 4. CONCLUSION

Using the classic arcade game Space Invaders as a reference, our students have re-engineered the game to gain practical experience using good software engineering principles. They developed fundamental understanding of game engine architecture through design and implementation of complex game systems. The students saw in detail how the use of design patterns gave rise to a software architecture that was decoupled, scalable and data-driven. The principles learnt in this restricted problem can certainly be applied providing best practices for building more general software architectures.

## 5. ACKNOWLEDGMENTS.

We wish to thank the DePaul University's Architecture of Computer Games class, administration of DePaul and DePaul College of CDM, and the School of Computing's Software Engineering program who have supported the course

## 6. REFERENCES

[1] Blumenthal, R. 2009. "Space Invaders: A UM L Case Study', Regis University, Avail at http://www.docstoc.com/docs/15330021/Space-Invaders (downloaded 10/17/10)

[2] Davison, A. 2005. Killer Game Programming in Java, O'Reilly Media, Inc.

[3] Fan, M., Stallaert, J., and Whinston, A.B. 2000 "The Adoption and Design Methodologies of Component-Based Enterprise Systems," European Journal of Information Systems (9:1) pp. 25-35.

[4] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 2002. Design patterns: abstraction and reuse of object-oriented design. In *Software pioneers*, Manfred Broy and Ernst Denert (Eds.). Springer-Verlag New York, Inc., New York, NY, USA 701-717.

[5] Gamma, E., Helm, E., Johnson, R., Vlissides, J. 1995. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[6] Keenan, E., Steele, A. and Jia, X. 2010. "Simulating Global Software Development in a Course Environment," Global Software Engineering, International Conference on, pp. 201-205, 2010 5th IEEE International Conference on Global Software Engineering,.

[7] Kruchten, P.; Obbink, H.; Stafford, J. 2006. "The Past, Present, and Future for Software Architecture," *Software, IEEE* , vol.23, no.2, pp. 22- 30, March-April 2006

[8] Paulisch, F. 1994.; , "Software architecture and reuse-an inherent conflict?,", Software Reuse: Advances in Software Reusability, *Proceedings., Third International Conference on* , vol., no., pp.214, 1-4 Nov 1994

[9] Retro Gamer. 2007. "The Definitive Space Invaders". *Retro Gamer* (Imagine Publishing) (41): 24–33. September 2007.

[10] Sharp, J. Ryan S. 2010, "A theoretical framework of component-based software development phases", SGMIS Database, Volume 41 Issue 1