Zol: Design Document

J. Halstead, J. Ciurej, A. Exo, N. Jeffrey, E. Christianson, E. Chan
University of Illinois: Urbana-Champaign (CS428)

May 4, 2014

Contents

1 Brief Overview of `Zol`

The primary motivation behind the initial conception and the continued development of the `Zol` project was a desire among the members of the original development team to create a general and extensible game engine that could be used to facilitate rapid game development and game prototyping. That said, the objective of the `Zol` project is to provide game designers and developers with an intuitive and robust game engine groundwork upon which they can quickly and easily develop video games with a wide variety of different rules and behaviors.

While the project implementation has quite a way to go before it can be used to easily generate completely general games[1], the project in its current state supports a great assortment of tools for creating varied two-dimensional experiences. In particular, `Zol` provides the following utilities for two-dimensional games:

- General game entity construction and specification using the finite state machine behavior specification pattern.
- General and overridable collision detection and collision resolution infrastructures.
- Composite hitbox support with abstracted SVG specification and arbitrary collision resolution behavior per hitbox.
- Animation support with specification via the commonly used sprite sheet technique.
- Tile-based game world construction and generation using adjustable input image maps.
- Integrated support for in-game cameras with panning and easing functionality.
- Basic infrastructure for generalized game world rendering with support for user interface widgets and overlays.

In order to demonstrate these capabilities, we've included the implementation of a basic game that mimics the classic title The Legend of Zelda. While this version isn't nearly as fully featured as the original, we believe that this demo game adequately demonstrates both the capabilities of our engine and how to properly utilize these capabilities.

2 `Zol` Development Process

The development process followed by the development team while developing `Zol` was a variant of the Extreme Programming process (as it was presented in CS427: Software Engineering I) adapted for use in an academic setting. The adaptations (which are expounded upon in more explicit detail below) were made to better suit the development process to a group developers with highly variate schedules and (consequently) limited availability to meet.

2.1 Iterative Development

As far as iterative development is concerned, our process doesn't deviate much from its base Extreme Programming model. Each development iteration spans two weeks, beginning with a planning game (involving the resolution and refinement of user stories) and ending with an iteration product.

In order to stress Extreme Programming's core principle of communication, our process also requires that each iteration contain four group meetings: two group requirement/review meetings (one at the beginning of each week to iron out requirements and design), one preparation

[1] See the "Future Work" section for more details

meeting (the day before the end of the iteration to polish the iteration product), and one presentation meeting (the final day of the iteration to demonstrate the product).

2.2 Refactoring

The process we used for the `Zol` project is identical to Extreme Programming process with respect to its approach to refactoring, requiring that development follow a test-code-refactor cycle and that code be refactored whenever and wherever necessary. This requirement led the development cycle of each team member during each iteration to adhere to a schedule similar to the following:

- Refactor the implementation code associated with one's currently assigned task, updating test methods wherever necessary.
- Write the implementation and associated testing code to accomplish one's currently assigned task.
- Perform first-pass refactorings to the previously written code to improve its quality (time-permitting).

2.3 Testing

While our process for `Zol` values and upholds the core Extreme Programming rules related to software testing, it includes one slight modification: test-driven development practices are replaced with test-inspired development practices. Put another way, our process removes the requirement imposed by Extreme Programming for tests to be written before implementation code and instead allows developers to choose on a case-by-case basis on which segment of code should be written first.

The primary reason that our process was adapted in this way was to reduce the volatility of testing code. Many of us found that, unlike advocates of the test-first methodology claim, writing test code first doesn't adequately elucidate implementation design requirements, which often causes tests to be rewritten or overhauled after such requirements are discovered (often by means of writing the implementation code itself). As such, we adopted the more flexible approach of test-inspired development over test-first development to eliminate the amount of time wasted on this unnecessary rehashing, which expedites project development in most cases.

2.4 Collaborative Development

Perhaps the most prominent deviation of our process from its Extreme Programming base is its replacement of the centralized code review technique called pair-programming with the distributed code review scheme afforded by pull requests and the shared-repository model. This change shifts collaborative development to become asynchronous and spatially independent, supporting code reviews in a much more flexible and accessible way.

While retaining the pair programming requirement in our process would have certainly been possible, it would have introduced a lot of scheduling overhead. Since the pair programming technique assumes that the members of each development pair have similar schedules, it's far too cumbersome to be useful in an academic project involving multiple, diverse student members (e.g. the `Zol` project). Hosting the reviews through a distributed version control system (i.e. GitHub) eliminates this need for scheduling while retaining all the benefits of code reviews, providing the optimal collaboration management solution.

3 `Zol` Project Requirements

The requirements for the `Zol` project can be found on the project's main wiki page. It's important to note that these requirements are directed towards a game designer client as opposed to a game player client. As such, these requirements detail the required capabilities of a general game engine that a designer may use to create a game (and not any particular game created through this means). In contrast, the given use case document outlines the potential user actions for the given `Zol` example game. This choice was made in order to cut down on the amount of abstraction and to present a more cohesive use case to potential designer users.

The following is a listing of important requirement specification documents found on the main project wiki:

- User Stories
- Use Cases

4 Implementation Design and Architecture

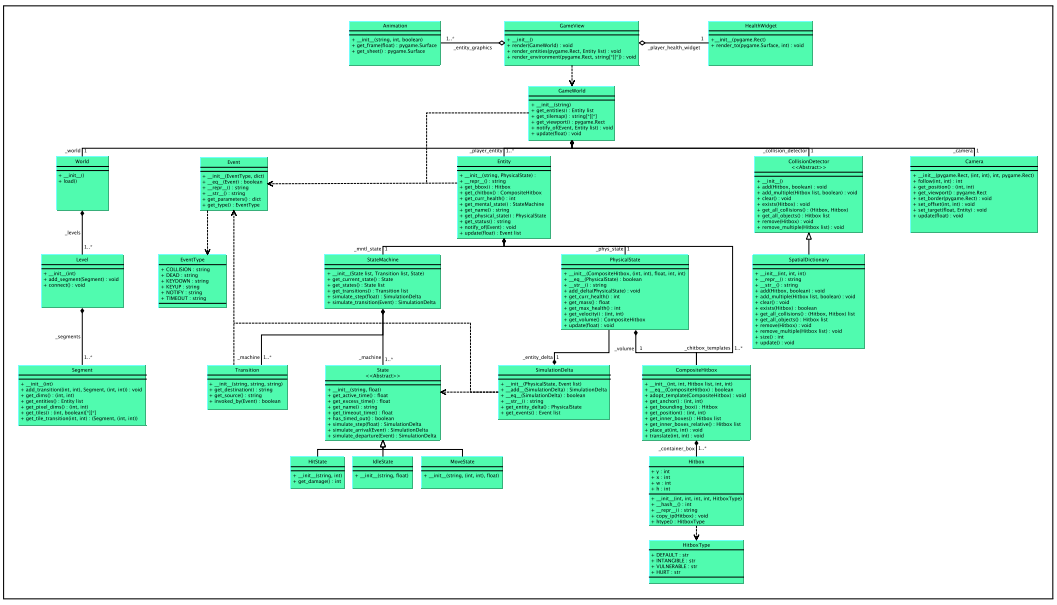

Figure 1: UML Diagram for the `Zol` Structure

At a high level, the `Zol` game engine is an implementation of the model-view-controller design pattern with the following components:

Model: All the logic and data associated with the implementation of the underlying game world, including the logic for updating entities, loading new worlds, detecting entity collisions, resolving entity collisions, etc.

View: All the logic and data associated with the rendering and displaying of the contents of the underlying game world, including the logic for displaying world entities, displaying world tiles, and rendering user interface modules and overlays.

Controller: All the logic and data associated with updating the model and view components based on user input. In contrast to the previous two components, this component is quite sparse, encapsulating only the user key input capture and broadcast logic.

The most interesting and complex of these components is the model component, which contains all the logic for game behaviors. The main class within this component is the `GameWorld` type, which represents a singular and whole instance of a simulated world in which a game may take place. Each of these world instances consists of four primary sub-components: a list of `Entity` objects that live and interact within the world (each of which is represented as a finite automaton), a `World` object to serve as the tile-based backdrop for the space, a `Camera` to keep track of the observer's view into the world, and a `CollisionDetector` to detect and resolve collisions that occur between entities. These components work in tandem to simulate interactions between `Entity` instances within the world, which serves as the basis for game behavior.

4.1 Primary Implementation Components

The core of the `Zol` project consists of the higher-level types used to implement the model and view components within the overall architecture. For each of these primary components, the listing below describes the major functionality encapsulated by that primary component and provides a visual aid for its role in the `Zol` architecture through a UML diagram.

4.1.1 The `Event` Component

The `Event` class is a representation of an event that occurs within the context of a `GameWorld`. An `Event` instance is generated whenever an event of interest occurs (e.g. a collision, a user input, etc.) with an appropriate type (as specified by its `EventType`) and propagated through the world to the relevant (i.e. affected) `Entity` objects. Instances of this type drive the state changes and (consequently) behaviors of the `Entity` objects to which they're propagated.

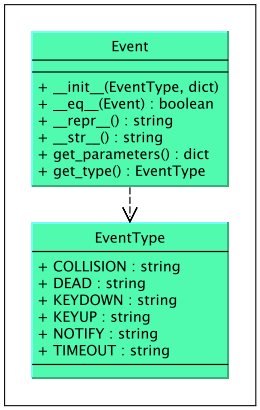

Figure 2: UML Diagram for the `Event` Structure

The primary function of the `Event` class is to serve as a form of message that encapsulates information about an unusual occurrence within the simulation that's likely to cause change. Instances of this type are passed back and forth between the `GameWorld` and its contained `Entity` objects to facilitate communication without coupling the two types.

4.1.2 The `PhysicalState` Component

The `PhysicalState` class represents the physical components of any object within the context of the `GameWorld`, such as their position, velocity, mass, and collision volume (or hitbox). The properties for any `PhysicalState` can be modified by applying state deltas, which are

encoded instances of the `PhysicalState` containing delta values for each property. This functionality of the type is often exercised when incremental changes are being applied to a persistent instance (which is often the case for `Entity` updates).

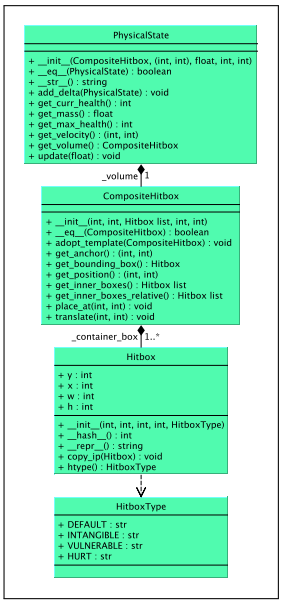

Figure 3: UML Diagram for the `PhysicalState` Structure

The `PhysicalState` type is used primarily by the `Entity` class to represent the physical properties of each individual `Entity` instance. This type is also utilized by the `StateMachine` class in order to indicate physical changes generated by the behaviors of contained `State` instances.

4.1.3 The `StateMachine` Component

The `StateMachine` class serves as an implementation of the finite state machine behavior specification pattern for the `Zol` project. In this particular implementation of the pattern, each `State` represents a particular behavior codified by a change in `PhysicalState` (e.g. standing still, moving, etc.) and each `Transition` represents a change of state invoked by a particular `Event` or class of `Event` instances.

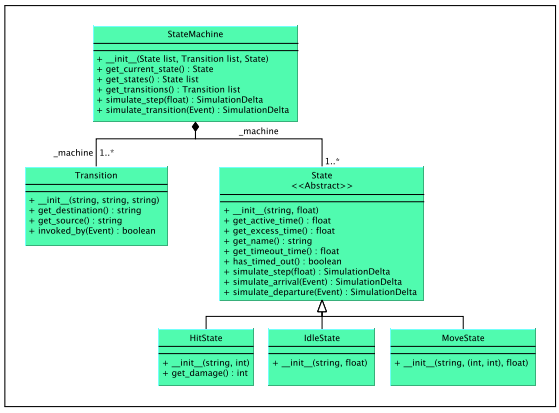

Figure 4: UML Diagram for the `StateMachine` Structure

The main purpose of the `StateMachine` type is to structure and specify the behavioral patterns of `Entity` class instances in a completely general way. Essentially, the `StateMachine` type serves as the 'brains' for the `Entity` instances in the world, dictating how these objects should behave and react under certain circumstances.

4.1.4 The `Entity` Component

The `Entity` class represents any dynamic object that exists within the game world and interacts with other similar objects. Each instance of this type contains two distinct sub-components: a `PhysicalState` (which represents the object's current physical properties) and a `StateMachine` (which represents the object's current behavioral properties). The `StateMachine` is primarily responsible for governing how the `Entity` behaves and changes as it interacts with other objects within the same `GameWorld` and responds to events.

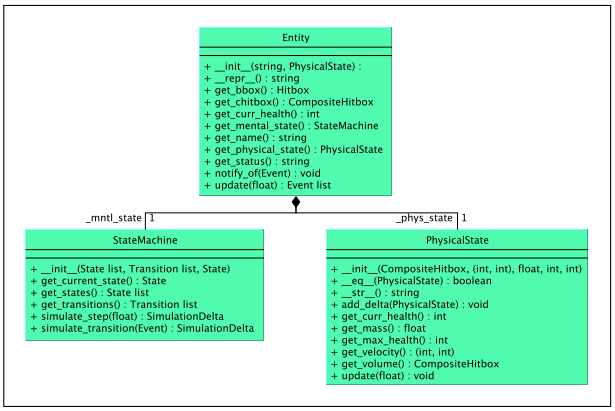

Figure 5: UML Diagram for the `Entity` Structure

All `Entity` objects have static asset binding support by default based on their classification, which is used to specify an instance's `StateMachine` infrastructure and animation/hitbox bindings to inner `State` instances. This feature can be used to easily generate new `Entity` types on the fly with unique behaviors and visual elements. The following list describes the paths and formats for these static bindings:

- **State Machine:** $(ZOL_DIR)/assets/data/entities/(entity-class).json
- **State Animation:** $(ZOL_DIR)/assets/graphics/entities/(entity-class)/(entity-state).png
- **State Hitbox:** $(ZOL_DIR)/assets/data/hitbox/(entity-class)/(entity-state).svg

4.1.5 The `World` Component

The `World` class is responsible for loading all of the levels (we call the segments). A level in our game engine is a collection of segments. Segments are areas that the player can explore and they contain obstacles and enemies. Players can transition between segments by walking over transition tiles. To make segment creation easy we save them as .gif files. Each pixel represents a tile, and we also define which tiles are tangible (the player collides with them). Additionally we define where entities such as the player and enemies spawn. The `Segment` class loads these image files and provides and interface for retrieving this information. The `Level` class is a container for the segments. It matches up the transition tiles between segments in the same level. The `World` class searches the directory where we save segments, and creates levels out of them.

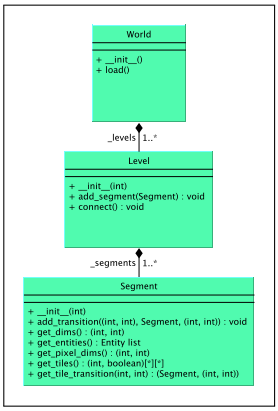

Figure 6: UML Diagram for the `World` Structure

4.1.6 The `GameWorld` Component

The `GameWorld` type is the core type within the model component of the overall project architecture, encapsulating all the functionality associated with simulating game logic. Each `GameWorld` is comprised of a collection of distinct `Entity` instances, which represent the dynamic contents of the environment, and a single `World`, which represents the static, tile-based backdrop for the environment. Each `GameWorld` instance is responsible for detecting and resolving the interactions between its contained objects, which involves generating and propagating `Event` instances when extraordinary events occur (e.g. collisions, user inputs, etc.), intercepting and interpreting `Event` instances generated by contained `Entity` objects, and detecting/resolving collisions between `Entity` objects and the background environment.

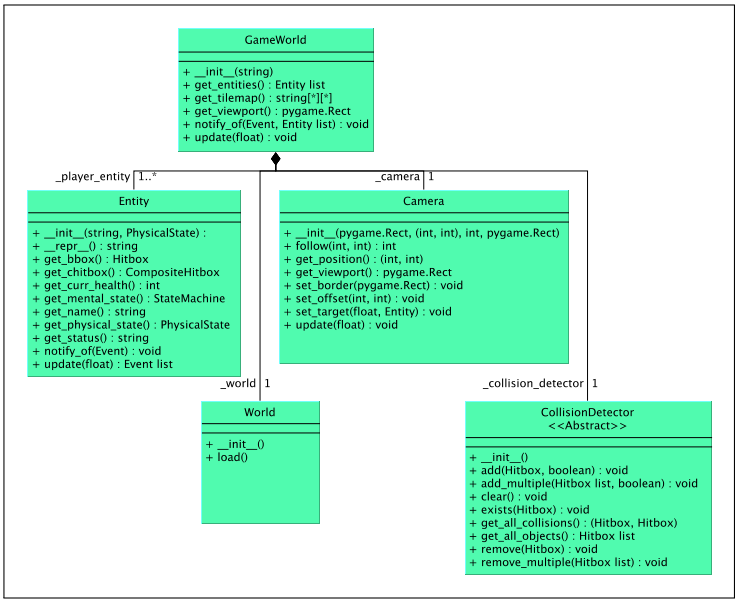

Figure 7: UML Diagram for the `GameWorld` Structure

The primary purpose of the `GameWorld` type is to amalgamate all the model components of the `Zol` architecture into a single, cohesive type. This type is used by the controller and view components of the architecture to capture an overview of the current state of the game model at any point in time and to modify this state as needed.

4.1.7 The `GameView` Component

The `GameView` type serves as the the core type of the view component of the overall project architecture, containing all the logic associated with rendering model information and user interface elements. Given an instance of the `GameWorld` type, the `GameView` type is capable of rendering all the `Entity` and `World` contents of this instance relative to the world's current viewport. This type is also capable of rendering arbitrary model-dependent user interface overlays to provide the player with more precise information about the game's current state.

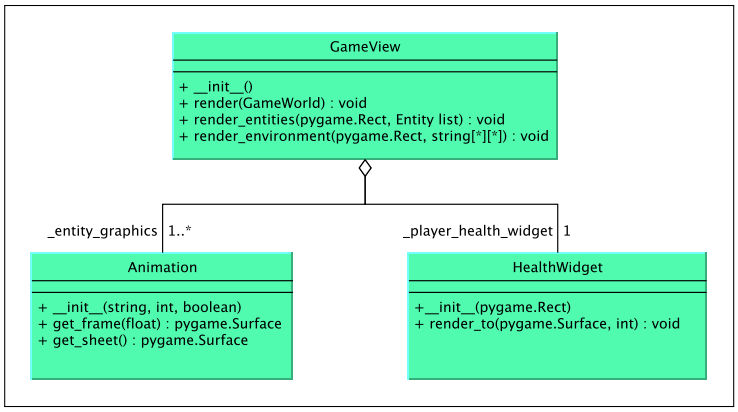

Figure 8: UML Diagram for the `GameView` Structure

Much like the `GameWorld` type for the architecture's model components, the primary purpose of the `GameView` type is to combine all the view components of the `Zol` architecture into a single, cohesive type. This type is used solely by the controller to render the contents of the `GameWorld` after these contents have been updated each frame.

4.2 Framework/Library Integration

For the implementation of our project, we consciously avoided using any existing game development frameworks in order to maintain complete implementation freedom. One of the primary goals of our project was to implement a generalized game engine, and using an existing game framework would only serve to complicate and limit this task. Instead, we decided to implement the entirety of our system without a framework, only using supplementary libraries such as pygame to help with lower-level operations (e.g. image loading, image rendering, etc.). Thus, the `Zol` project's design is completely independent of any existing game framework's design/architecture.

5 The Future of `Zol`

During our initial development period, we found that we were not able to fully realize many of the features we wished to incorporate into the `Zol` game engine. The following section details the major additions we wish to integrate into the engine in the future as well as future plans we have for the project.

5.1 Future Functions and Features

There were many features that we considered during the initial development period that couldn't be fully implemented due to time constraints. These features primarily involve improving the generality of the existing game engine, extending the basic functions provided by the game engine (e.g. adding to the list of provided state behaviors), and providing an enhanced user interface for specifying world entity behavior and game world infrastructure.

The following is a listing of all the most important features to be added to the `Zol` project in the future:

- Creating a user interface to facilitate the full manipulation of entity state machines.
- Creating a user interface to facilitate more user-friendly creation and manipulation of tile-based game worlds.
- Further generalization of the collision detection and collision resolution systems.
- Further generalization of the rendering engine to incorporate arbitrary user interface overlays and widgets.
- Adding support for arbitrary game state saving and loading.
- Adding support for hot-swap game loading to facilitate more rapid prototyping and game parameter adjustment.
- Expanding the functionality of the in-game camera utility to allow for behaviors such as zooming.
- Expanding of the in-game menu system to support more general menus with a wider variety of options.
- Restructuring the existing architecture to facilitate multi-threading (e.g. separate threads for the model-related and the view-related logic).

5.2 Future Deployment Details

At the end of the initial development period, we plan to publish the `Zol` project as a standalone application to a new GitHub project repository. Once the project has been republished, the development process for the project will shift to a Bazaar-based process with Joe as the primary project manager and a new phase of development will begin. During this phase and all future stages of development, the project repository will be made publicly available so that any existing team member or open-source contributor may help to expand the existing implementation into a more robust game engine.

5.3 Team Reflections on the Project

The following section contains a brief overview of each team member's personal experience with the project, including their personal reflections on both the project's development (especially in terms of the development process) and on the final product.

Josh's Reflection

When Joe and I wrote the proposal for Zol we knew we wanted to use a development process that was similar to XP but offered more flexibility in terms of TDD and pair programming. Our compromise was to use Git, adopt code reviews via GitHub's pull requests, and adhere to "test-inspired development".

What (pleasantly) surprised me about those small adaptations of XP was how much more connected to the entire team and project I felt. The code review process especially was transparent and open to the entire team. It offered me the chance to see and comment on changes that would have been masked by Subversion and pair programming. This alone left me with a far greater understanding and feeling of ownership of the project as a whole. And I would gladly use a similar process in the future.

As for the final product, I am quite pleased with the architecture of the engine. The quality of which is demonstrated by how quickly we can create demos and alter the game environment. What I learned in creating the final demo is that even the most well thought out systems need modifications once they are actively being used. That lesson is something I will take with me as I work on future products and struggle with getting it just right the first time vs. just quickly getting a product and refactoring from there.

Andrew's Reflection

I enjoyed working on this project, and I was surprised by how effectively we were able to work together as a team. The elements of our process that I found to be the most beneficial were the pull-request/code review contribution method and coveralls. Github's pull-request feature made it really simple to review the work that each member did. We could give feedback to each other, and it also provides a form of documentation of our process. Coveralls provided us with code coverage information. This made it easy to see which areas of our code our tests were missing. Even though we allowed tests to be written after code, we ended up with 95 percent code coverage.

Now that we're done, I'm satisfied with what we created. It's not exactly a game, but it is a robust game engine. Creating new levels, animations, or even character behaviour is incredibly easy. All of these things can be added without touching the code.

Overall I feel like I have learned a lot about being part of a development team. I tend to work alone even when given the option to work with others, so this was a very valuable experience for me. Now I feel much more comfortable contributing to a shared repository using a version control system. In the future I will use this experience to more effectively and more confidently contribute to open source projects.

Nick's Reflection

I think that the development process we used for this project was really swell. We made excellent use of branches in Git to work through our assigned tasks without breaking things for other people working on their tasks, but while still being able to push things to Gitub for version control. We also had Travis for continuous integration, which helped avoid pushing broken code to master, and coveralls to help us know which classes needed better test coverage. Coveralls is by no means perfect though, since it only tracks whether lines were touched during testing, not whether they behave as intended, but it served as a very good heuristic that kept everyone writing good tests for their code. These factors contributed to a final product that I think is very stable and well tested.

Edwin's Reflection

I've been very passionate about video games for a long time and this is the first time I've taken part in the creating side of it with a group of others. Overall I thoroughly enjoyed the entire experience and learned a lot about software development process. I also learned a lot about the the fun and pain of game development. The development process was very good and orderly in that I learned a lot about using a shared repository and all the goods it can bring as well as the headaches. The final product is something I am really proud of, if we had more time I wish we could have added sound effects but other than that, it is something I am satisfied with.

Eric's Reflection

I like the development process that we used worked out pretty well, and it was a very good decision to switch to Git for version control. I am pretty satisfied with how our end result came out, because its a very well structured game engine that can be repurposed for multiple games. It took more time to have a game to show for it, but it was a good decision overall.

Joe's Reflection

Overall, I'd say that I'm quite pleased with the outcome of this project. While there were certainly a few high points and low points during development, I'm happy with the experience overall because it taught me a lot about collaborative software development and game development technologies.

In particular, I learned that the shared repository model is extremely effective in facilitating distributed collaborative development and that test-inspired development serves as a great testing technique alternative to test-driven development when dealing with unfamiliar designs and technologies. Additionally, I thought that the adapted development process that our group used for this project worked really well, and I hope that I can utilize similar processes effectively in my future projects.