Assignment 5

Alex Geenen, Bart de Jonge, Mark van de Ruit, Menno Oudshoorn, Stefan de Vringer

Software Engineering Methods, Group 1, 23/10/2015

Exercise 1: 20-time, Revolutions

For this exercise, we first show the new requirements for the new features and improvements we wished to develop. Our focus lay on improving the GUI, and adding sounds.

Functional Requirements

Must haves: none.

- The game must have a multitude of sounds
    - The game must play differing types of music, playing when the player is in a menu, or playing a game, or dead, or has won a game.
    - The game must play representative sounds when a player fires a laser, a spiky laser, an instant laser, destroys a bubble, picks up a coin, powerup, etc.
    - The game must play representative sounds when a player highlights a UI element, and clicks on/selects said element
- The game must have full keyboard support
    - The player must be able to navigate the menus using nothing but the 'Arrow' keys
    - The player must be able to use buttons, textfields, etc, when pressing the 'Enter' key
    - The player must be able to completely empty text fields instantly, when pressing the 'Backspace' key or a dedicated on-screen button.

Should haves: none.

- The game should show a 'splash screen' when a player boots up the game.

Could haves: none.

- The game could have some visual effects, to make the 'old terminal' feel more believable
    - The game could have some consistent screen noise
    - The game could show artifact-stripes running across the screen randomly

Won't haves: none.

Non-functional Requirements: none

Now that we have discussed the requirements for exercise 1, we show how we have continued using Responsibility Driven Design to further develop these features. We do this by creating CRC cards for new and updated classes. We have also kept our (rather humongous) UML class diagram up to date. **Due to the size of this diagram, however, it should be opened separately from this document. See A5-UML.png.** But first, we discuss one of the new features, sound, in detail.

When we first looked at this assignment, we thought to ourselves: 'What is our game still missing?". We concluded that we had failed to build one of the fundamental pillars of any good game: sound. The game was sorely lacking any sound effects, let alone music.

Since we've nearly reached the end of the course, we now have a better understanding of which steps we should take when implementing a new feature. Thus, we started on this feature by carefully designing it bottom-up, so our implementation would be of sufficient quality.

We quickly noticed the following two properties:

- At any given time, only one track of music should and could be played.
- Sound effects can and will be played from almost everywhere in the game.

Due to this, we decided to focus our efforts on one central *SoundPlayer* class.

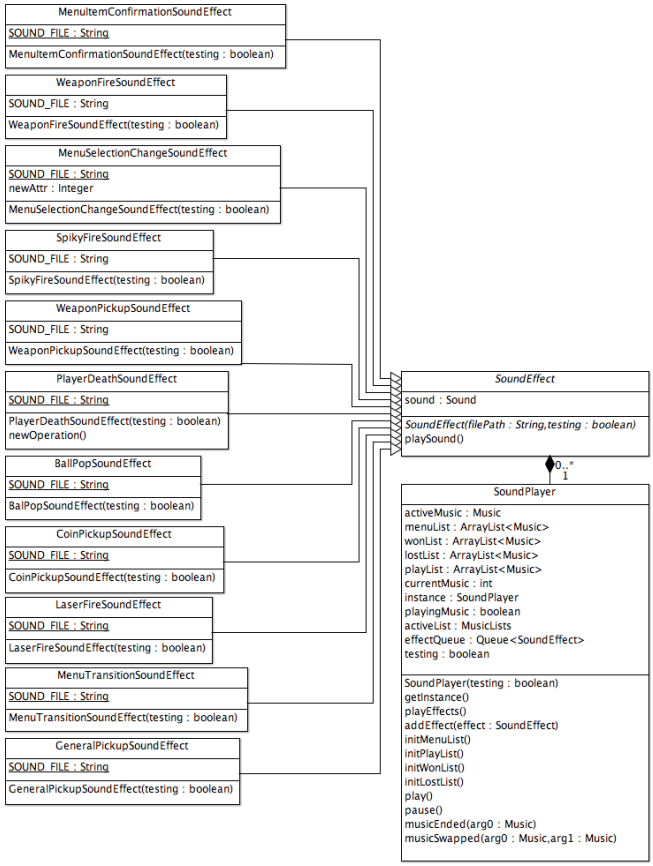
There would be exactly one *SoundPlayer* running at any time, which made it a perfect fit for the singleton pattern. This way, every single class can access the *SoundPlayer* and queue sound effects. To make sure this wouldn't cause any issues, we added a *concurrenysafeQueue* to our design, which will temporarily store the effects. The queue is then be emptied (by playing all sounds one-by-one) at the end of each *update()* loop.

The sound effects that can be added, in turn, extend an abstract superclass named *SoundEffect*. This ensures that all sound effects play by the same rules, meaning that they have an actual sound and a *play()* method.

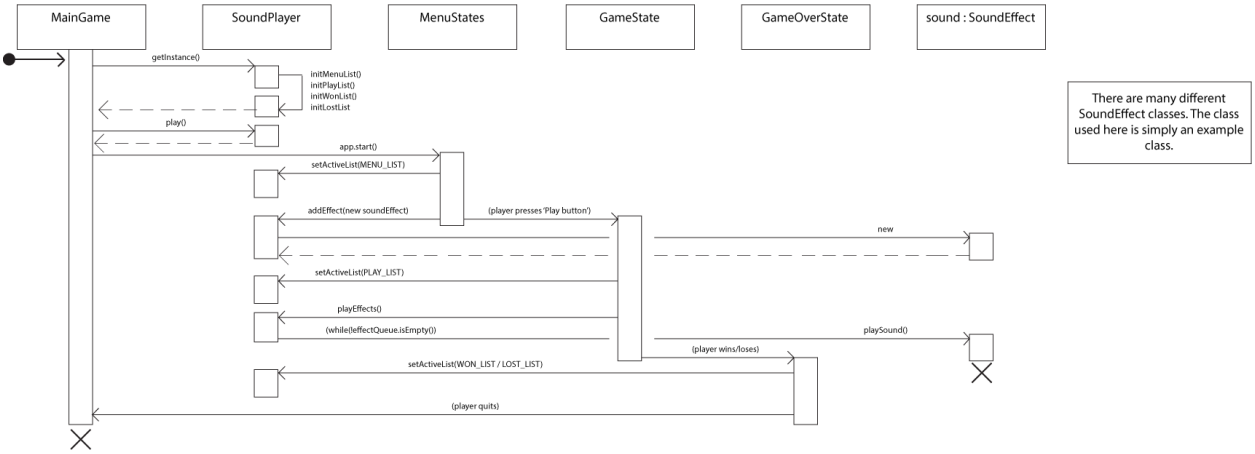
Once we were finished with the design, we implemented it without any real issues. However, one item that should be noted is that the (new) *Sound* package has hardly any test coverage. This is because not all tests (Maven/Travis) have access to an environment in which they can play sounds. Furthermore, the Sound package suffers from the same issues the GUI packages have, in that it is hardly testable, due to *Slick2D*'s allergy to mocking of any kind.

To better help the reader understand how we've implemented the new sound feature, we have also created a separate class and sequence diagram, describing the entire process. These are also **included separately**, for your convenience.

Sound Class Diagram



Sound Sequence Diagram



Another item that should be noted is that we have created CRC cards for all new/updated classes in Exercise 2 and that these are visible here as well.

To clarify: CRC cards contain the class name, superclass, and subclasses. Then the bottom-left part of a crc card describes responsibilities, while the bottom-right part describes collaborations.

Name: ElementList	
Superclass: none	
Subclasses: none	
Store, manage and connect all separate interactive UI-elements	All states, Element, all Element-extending classes (eg, button), Popup

Name: Element	
Superclass: none	
Subclasses: Button, Textfield, PlayerButton	
Represent an interactive UI-element which can be connected to other elements	All states, ElementList, all subclasses

Name: Button	
Superclass: Element	
Subclasses: PlayerButton	
Represent an interactive Button in the UI	All states, ElementList, Element, Popup

Name: PlayerButton	
Superclass: Button	
Subclasses: none	
Represent a special interactive Button in the settings UI, used for selecting players	MenuSettingsState, Element, Button, Player, MainGame

Name: Textfield	
Superclass: Element	
Subclasses: none	
Represent an interactive Textfield in the UI, used for filling in names, ip-addresses etc.	All states, Element, Button, Slick2D.TextField

Name: Command	
Superclass: none	
Subclasses: AddDroppedCoinCommand, - PowerupCommand, -FloatingScoreCommand, -PowerupToPlayerCommand, RemoveCircleCommand, -DroppedCoinCommand, DroppedPowerupCommand, -FloatingScoreCommand, -SpeedPoweruCommand, SetCircleListCommand	
Form abstract basis for all Commands. Commands are small classes holding a small item of code, which will influence the game in one way or another.	GameState, Player, Client, Host, Connector, CommandQueue

Name: SoundPlayer	
Superclass: MusicListener	
Subclasses: none	
Manage and help with playback of sounds and music throughout the entire game.	All states, Player and all PlayerHelpers, SoundEffect and children.

Please note, that, due to the sheer number of SoundEffect extending classes, these classes will not get separate CRC cards. Most of them are no more than five lines of code, extending and overriding what SoundEffect itself already does.

Name: SoundEffect	
Superclass: none	
Subclasses: all SoundEffect extenders. Eg: SpikyFireSoundEffect, CoinPickupSoundEffect, GeneralPickupSound, etc.	
Represent a playable sound that can be played in the game.	All states, SoundPlayer, most other classes even a little bit.

Name: MenuSplashState	
Superclass: BasicGameState	
Subclasses: none	
Represent a Menu for introducing the player to a game.	All other states, MainGame, Button, Separator, Slick2D.Image, RND.

Exercise 2: The Iterator Pattern

Natural Language Description of the Iterator Pattern

For this assignment, we have chosen to implement the Iterator Pattern. This fit our project quite nicely, as our code has several classes which contain an *ArrayList* of a certain type of *Object*. All these classes will, from now on, be referred to as "list classes".

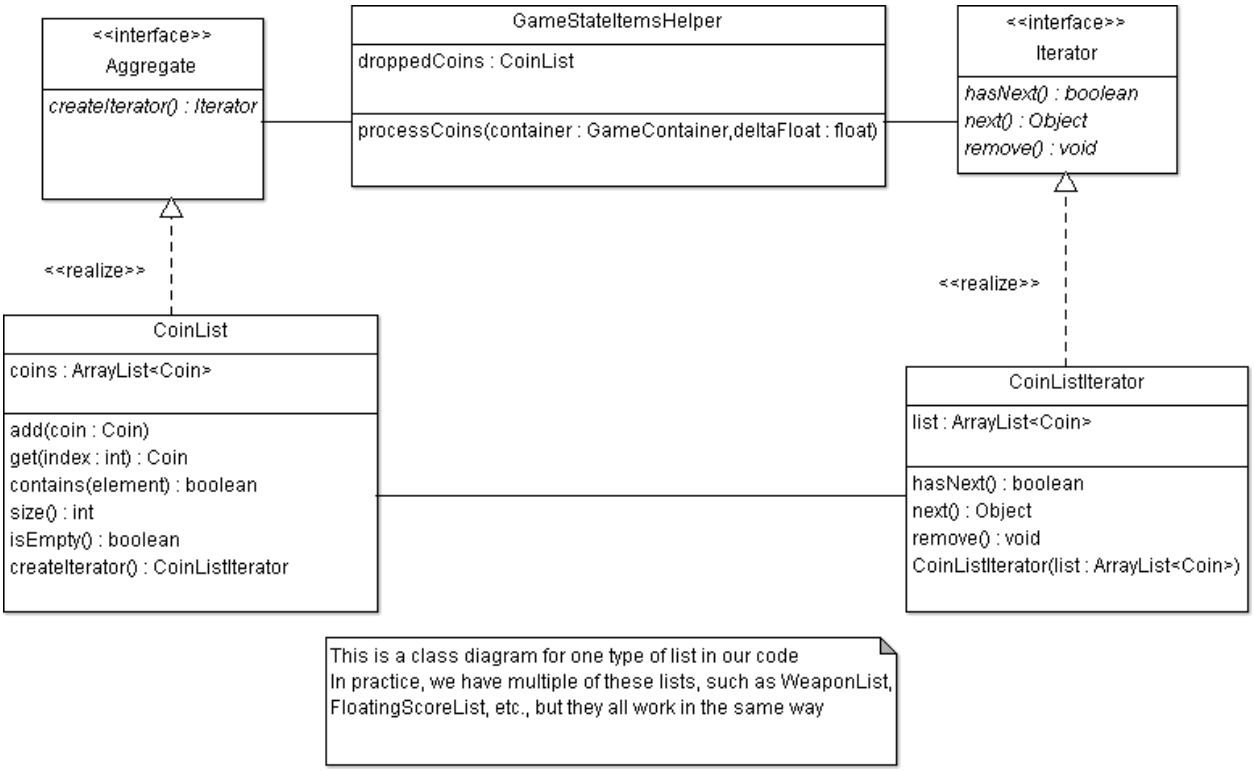
The benefit of implementing the Iterator Pattern is that one does not need prior knowledge of how a list class stores its iterables. Instead of perusing a *for-loop*, in which you have to access every *Object* in a (possibly) convoluted manner, you can always rely on the *createIterator()* method to loop through the *Objects* sequentially, and in the correct manner.

Our implementation features two interfaces. The first one is called the 'Aggregate' interface, which every list class must implement. It ensures that list classes have a *createIterator()* method. The second one is called the 'Iterator' interface. Any iterator class must implement this interface, meaning every iterator is bound to have the *hasnext(), next()* and *remove()* methods.

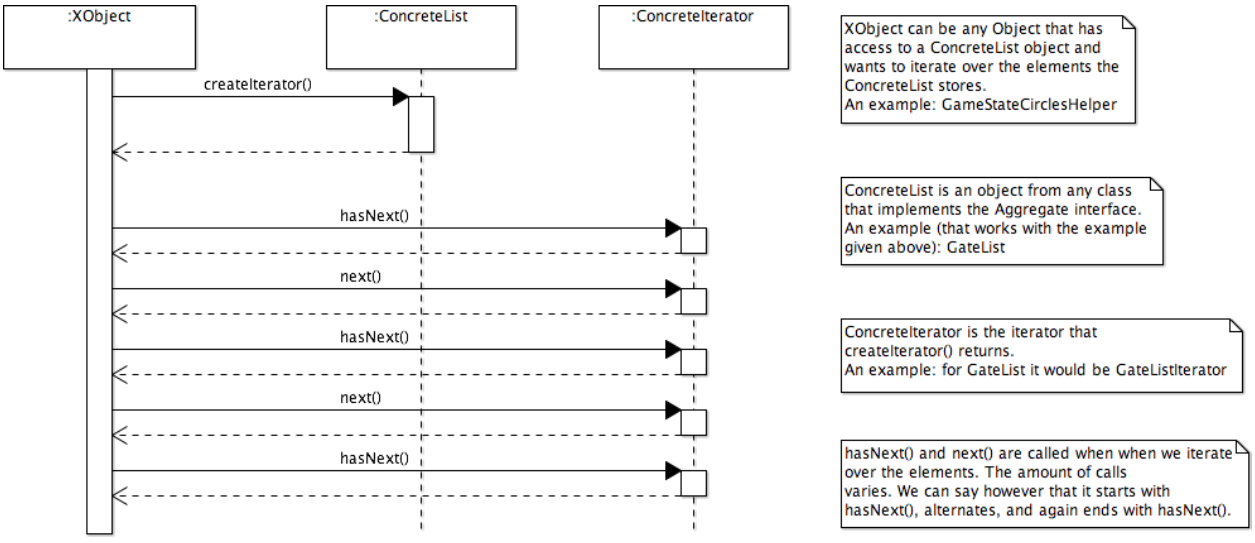
Now, in order to loop through all *Objects* stored in a list class, one first calls the *createIterator()* method, which then returns the correct implementation of *iterator*. This returned *iterator* then knows how to loop through the *Objects* properly. We can then use it to loop through our *Objects* in the following manner: '*while (iterator.hasNext()) { iterator.next(); }*'.

Please note that the following diagrams use *CoinList*, and *Coins*, as an example. In our implementation, the pattern is applied to far more than just coins

Class Diagram of the Iterator Pattern



Sequence Diagram of the Iterator Pattern



Exercise 2: The Command Pattern

Natural Language Description of the Command Pattern

For this assignment we were also allowed to discuss design patterns that we had previously implemented, outside of assignment 3. Therefore, we chose to discuss our use of the Command Pattern, which we had already implemented during assignment 4.

We previously implemented this, because it could assist us in solving a set of concurrency issues, which occurred on practically every list that was modified/replaced by the Host and Client classes during a LAN-multiplayer game.

More precisely, these issues only occurred during a very specific set of events. While one of the regular game/logic classes was iterating over, say, an *ArrayList*, the *Host/Client* classes could receive a message to delete or modify an element from that very list. As the *Host/Client* had deleted/modified that element, when the regular gamelogic reached it, it would run into issues. This would cause our game to throw a *ConcurrentModificationException*, which brought our game to a halt.

Initially, we were unable to fix this (we tried all sorts of synchronizing, to no avail), until our TA presented us with some possible fixes. This got us thinking, and soon one of us noted how one of these solutions was similar to the Command Pattern, as discussed in the lectures.

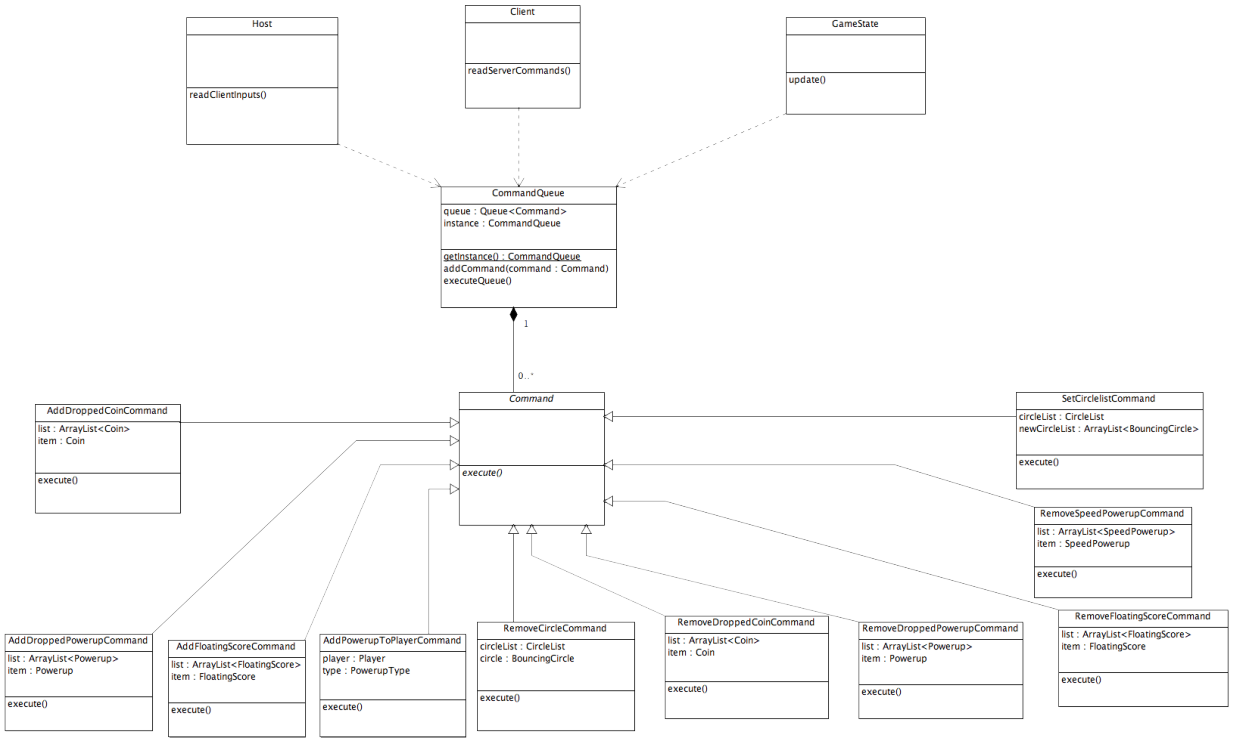
The Command Pattern describes a way to define a set of instructions/executions which can be called on afterwards. We decided to apply this in the following form:

- Game/Logic classes will loop over lists during *GameState.update()*. During this, no elements in the list may be added or deleted by an outside source.
- When the *Host/Client* receives an instruction to delete/add/modifiy an element, it will create a *Command* containing two variables: the list which is being modified and the element in question. This command is stored in a queue.
- Once *GameState.update()* has run its course, all commands in the queue will be executed in order (with checks in place to make sure elements are still accessible). At this point, all regular classes have already iterated over the lists. No more concurrency issues should occur.

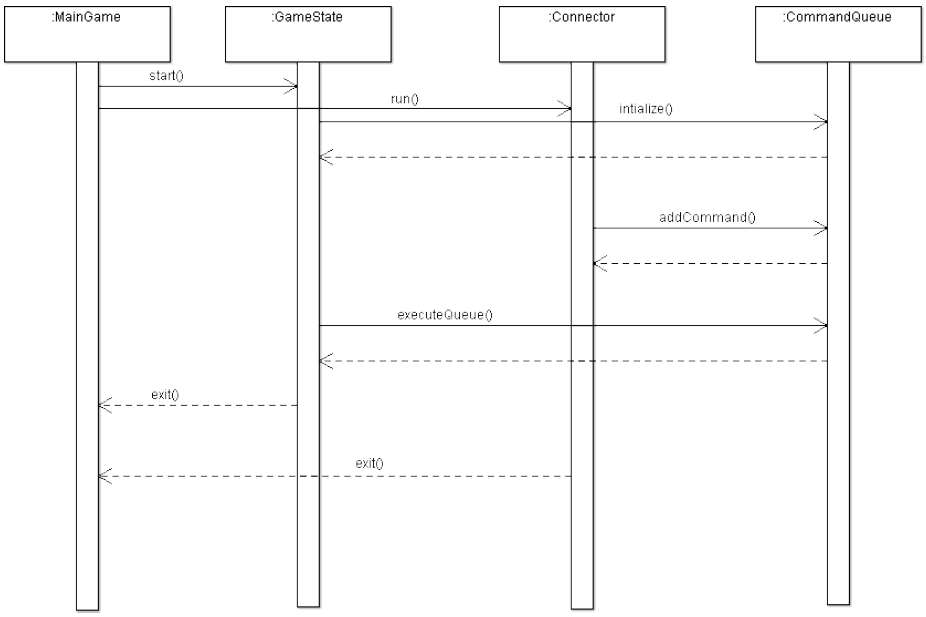
Our implementation therefore consists of three main parts: a singleton *CommandQueue* containing a concurrency safe queue, an abstract superclass *Command*, and a number of child classes of this superclass. All child classes are bound by rules set by *Command*, meaning that they can be added to the queue and executed.

Please note that these diagram are also **included separately**, for readability.

Class Diagram of the Command Pattern



Sequence Diagram of the Command Pattern



Exercise 3: Reflection

If one were to boot up the initial version of our game, and compare it to our current product, one would be inclined to say that outwardly not much has changed between the two. That no massive changes have been made, and no enormous overhauls have been done. Indeed, when seen from afar, the two appear to be almost identical. However, if one looks more closely at our game - not just the source code, which we will discuss later on - one will find that the two games are as dissimilar as they could possibly be!

Where the initial version of our game reflects the well-known arcade game that is Bubble Trouble - including its lasers, local multiplayer, and bouncing namesake - the final product has managed to build on, refine, and expand upon everything the old game represented with a plethora of new features. This includes simple things, such as new and improved pickups, customizable color schemes, and player skins. Many far more complex improvements have also been made, foremost of which is the ability to play multiplayer games over a local network.

What impacts did these changes have on the development of our project? What did we learn from the project as a team? Would we, when starting on another project, work in the same way, or would we take a different approach?

These questions will be addressed later on. First, we will look more closely at the source code of our project. Given that our game has evolved so much, one could assume that our source code has done the same. Is this actually true, and if so, what exactly has happened to our code?

Code changes

While our game has undergone massive changes, its source code has grown to four times its original size. Let us do a simple comparison: the first version of our game had 26 classes split into two packages, testing classes excluded. As the time of writing this reflection, our game has grown into over one hundred separate classes, divided amongst 11 different packages.

New packages have been created to support added features, such as LAN multiplayer and audio. Old packages were split up when they became too large. The GUI package, for example, has been split into three packages: one for the game's main GUI, another for the menus themselves, and a third for all miscellaneous classes.

While this massive growth contributed new features, it came at a cost. The complexity of our code has, between our initial and our current versions, massively increased. The quality of our code, meanwhile, has suffered just as much.

For example, during one of the later assignments, we were tasked with fixing as many issues as an automated issue detector called inCode could find. inCode reported that we had five (five!) God Classes; that is, classes which are too complex, have too many responsibilities, and hold too much data.

We managed to remove three of these by splitting up the classes into a set of smaller classes which divide tasks and responsibilities. While this negates the issue of a God Class, it has made our code that much more complex. Instead of one class, now there are a dozen!

What we've learned

One can image that, in our eyes, this has gone from a small homework assignment to a project which requires thorough design, thought, and input, before anything can be implemented.

Although everyone in our team initially 'piled on' features and classes, we've come to the point where we work together on features. We now apply newfound knowledge - such as some of the design patterns we've learned about during the course - to improve the existing code and better implement additional features.

This newfound knowledge, in all honesty, would have been useful to have had from the start. If we had had knowledge about software design patterns in the beginning, we would have been able to plan ahead. Some patterns, such as the singleton and the command pattern, have proven to be so incredibly useful, that we've gone back and refactored parts of our code to use these patterns, even when it wasn't a part of an assignment.

Another thing we've come to appreciate is the power of SCRUM. While we can see that it has downsides, and even limits, we do realize that by using SCRUM we have managed to incrementally complete our game in a matter of weeks. By working on nothing but a small set of features and improvements during every sprint, we've been able to completely focus our efforts on said features, which, in our opinion, led to every single feature being complete and polished after the sprint.

The future

The next time we have to design and implement a system on this scale - or even a larger one - we'll certainly start out differently. We now see that preparing a proper design, then implementing it, leads to a far more stable system than the other way around, which is how we started out.

Furthermore, we will make far more aggressive use of the design patterns we've learned about in class. Also, we will preventively act against classes which carry a risk of becoming too large. And finally, we will better prepare ourselves for change.

What do we mean with change? One thing we have noticed is that almost every part of our system, at one point or another, has been subject to modification. This oftentimes led to issues where we had to refactor said part before we could modify its functionality or add new features to it.

We now know that we should accept this change, and, more importantly, prepare for it. Some of our more recent code already reflects this. For example, the LAN multiplayer code now works with a set of commands, to which new ones - with vastly differing content - can be added at any time. The GUI was refactored to work with 'elements', from which all interactive user-interface objects stem. New objects can be added with ease.

The next time we implement a system similar to our own, preparing for changes and additional features should be done from the very beginning.

Wrap-up

Within a week, we will deliver the final version of our project. It will represent the cumulative effort, knowledge and work of all the members of this team. We've learned such a great deal about software engineering in such a short time, and hope that this project, this game that we have made, properly reflects this.

While we've run into enough issues to get a small taste of real-world software engineering, in the end we've really enjoyed working on this project.

We hope that, by playing our game, you can enjoy it too.