Software Analysis and Design

Assignment 6

Group members:
Lucas Fredin
Chi Hong Chao

Responsibility analysis:

General responsibilities in the Layered Architecture of our remodelled Asteroids game:

- **UI layer** - Holds classes that have to do with the displaying of the system to the user to see.
    - **AsteroidGraphics**
        - The core class of the UI layer. This class's responsibility is to decide what to show to the user depending on the current events happening in the application class. Coordinates other classes of this layer to perform this function.
    - **Background**
        - Contains information about what to show in the background of the application, including methods on what it can perform.
    - **HUD**
        - Contains information regarding in game text, and the font information of the system.

- **Application layer** - Holds controllers that take input from the user on the UI level, and contains the core logic of the application which makes objects interact with each other to make a cohesive system.
    - **Controller**
        - Takes input of the user and decides what to do/modify according to it, using the logic of GameLogic.
    - **GameLogic**
        - The core class of the Application layer. Contains the core logic of the game, deciding how the different objects of the domain layer interact with each other.
    - **Asteroids**
        - The entry point of the system. This class is entered when the system is opened. Deals with the initialization of the system, which as a result also deals with how threads are run in the system.

- **Domain layer** - Contains the information of each individual class, and what is part of each individual object.
    - **Sprite** - The base object that all the other objects except sounds inherit from. Contains basic information about the definition of a game object.
    - **Asteroid** - Contains information about an Asteroid.
    - **Missile** - Missile Info
    - **Photon** - Photon info
    - **Ship** - contains the definition of a ship. as the ship has thrusters, this class contains its own thruster objects.
    - **RevThruster/Fwd Thruster** - Thruster info, 2 different classes due to different shape
    - **Ufo** - Ufo info
    - **Explosion** - Explosion info
    - **Sounds** - Contains the sounds of the game, and methods on how to manipulate them in game.

- **Foundation layer** - Contains the information that is used by several classes in the above layers, as well as classes or libraries that is needed for the application to function.

    - **Constants**
        - Contains all the unchangeable constants used in different classes regarding different objects and the application.

- **<u>Technical services layer</u>** - Contains the raw data that is not created in the system, and libraries used. Reused code that are not made by the creator of the application is here as well.
    - **Raw sound files** (crash.au, saucer.au, etc)
        - The sounds the Sounds class in the domain package uses, and thus the application uses.

Layered architecture overview

UI	**Application** (Business/logic)	**Domain** (Services/Object)	**Foundation** (Persistence)	**Technical Services** (Database/Information Holders)
Background HUD AsteroidGraphics (the main view of the system)	GameLogic Asteroids Controller	Sprite Missile Ufo Asteroid Explosion Sounds Ship RevThruster FwdThruster Photon	Constants	SoundFiles

Major problems:

There were several things wrong, or bad about the original code, here we have listed the most important ones.

1) **No modularity**
    a) All of the code was contained in one single Java file, which makes everything very dependent on each other. Were we to change something in this file, just a small thing, it might not work at all anymore, and the program would be broken.
2) **No clear responsibilities**
    a) Since there were only 2 classes in one file, those classes had a bunch of different functions to handle, which meant they did not have any single responsibility. This makes it hard to figure out where to find methods you need, and even most methods did not have a single responsibility. (methods that should just do 1 single thing, are doing many different things)
3) **No architectural layout**
    a) There was no structure to this code since everything was in one single class.
4) **Zero Encapsulation**
    a) As there was no structure, and the fact that none of the attributes were private, everything could be accessed by anything.
5) **Long Methods**
    a) Several methods performed several large tasks with no information hidden. This also created unreadable code.
6) **Temporary Fields**
    a) many methods contained 6 or 7 temporary variables because of the lack of getters, setters and proper structure created much needed confusion.
7) **One-Letter Variables**
    a) Many temporary variables were confusing as the function of it was unknown. The fact that the variable was 1 letter long did not help. This created low readability of code.
8) **No use of inheritance**
    a) It was definitely clear that some objects belonged in the same type of group and thus could be generalized into a super type. The class AsteroidsSprite was a good contender for a super type for a number of objects.
9) **No abstraction or use of interfaces**
    a) Information hiding was not a thing in this code, since there not only was no encapsulation, there was also no information hiding by using for example abstract classes or interfaces.

Design solutions chosen:

Separating the code into classes

To go with the object oriented approach and in order to create high modularity, we define classes based on responsibility, aiming for a single responsibility in each class. This rule was broken a few times due to the close dependency of some of the several responsibilities.

Making use of inheritance

Classes that have a lot in common get their own generalization, in order to further increase modularity. Polymorphism also allowed us to give each respective object their own sound, for example calling missile.sound would be different than revThruster.sound. Of course we have getters and setters, but that is beside the point. Polymorphism also allowed us to have objects inherit from other objects through the use of inheritance. This allowed us to call explode() on any object we wanted, as the parameter could be the base(Sprite) class.

Determining responsibilities of classes and methods

Since the methods often had more than 1 responsibility to take care of, we sometimes had to split them up to force them to contain just a single responsibility.

Private, protected Variables

By having most attributes of the classes private, we increased encapsulation and security of the system. We made certain attributes protected because it was easier to modify the code.

Layered architecture

We put code into layers with their own different responsibilities and roles. This increased encapsulation even further as certain layers could not be accessed by others. This allowed us as engineers to only give more specific access of methods and attributes to different classes. With the additional protection of private variables, the layered architecture gave the system much needed information hiding and protection.
The layered architecture also solved other problems such as no modularity, unclear responsibilities and no architecture, with each layer having its own responsibility, where that responsibility was then further broken down into smaller ones inside each class.

Design Decisions Retrospective

Due to lack of time and lack of experience with similar projects we have made design decisions that are less than optimal, and this may have undermined some of the objectives we tried to achieve. Although we tried our best, we were unable to get the refactored program running. We are also aware that we have made some decisions that are against the best practices of creating a layered architecture, some of which directly break the purpose of the architecture. We would like to list these problems and justify them here.

Unimplemented methods

There were certain methods in the program that were deemed necessary, but that we did not have time to implement. Therefore methods can be found in a number of different classes that are nothing but empty shells. If we had more time and experience we would like to implement these, as they are fitting for the program and help improve its design.

The reuse of inheriting "Applet" in multiple classes

We chose to design our program in such a way that several classes extend the Applet class. With this method we skipped a lot of steps connecting different classes to try to display the game and get everything needed in place, and thus the game has higher coupling and dependencies that we would have liked.

We chose to have multiple classes inheriting from Applet due to several reasons: As we were inexperienced with how the Applet class worked, we found out that some methods did not work unless we extended Applet. However, we then had the conundrum of deciding whether to include the method that is supposedly in the UI layer back in the Application layer. We were also afraid of breaking the method by separating certain pieces of code within the method. We also found that inheriting from the Applet Class directly instead of creating classes to obtain only the methods we used much more easier, and time did not allow us to make extra classes like these.

With all this said, we are aware than classes in the application and UI layer should not have direct access to classes of the Foundation layer, where the Applet class is located. We are also aware that multiple classes inheriting from one class increases coupling and dependency, which is a bad practice.

To fix this, we could create a 'facade' class which would inherit the Applet class from the application layer, and restructure our classes so that only 1 class of each layer in and above the application layer would inherit from that facade class.

Setting attributes as "Protected"

We decided to set the attributes in the Sprite class as protected instead of private. We chose to do this in order to keep some simplicity in the code, since making private attributes with getters and setters would greatly reduce readability of the code, yet not contribute a lot more to the improvement of the system. Changing these to use getters and setters instead would also mean that we would have to change all the calls to these attributes in all of the subclasses, which would be a very time costly thing to do, and time was something we didn't have enough of.

If we'd had more time we would have fixed this by simply making the attributes private and putting getters and setters where necessary.

Constants in the foundation layer

We put all the constants of the game in their own class in the foundation layer so that they could easily be called from any other class, without having to have the class of the constants be instantiated. Because there are only constants in this class, this is not a huge issue even though we have dependencies running through many layers, because constants cannot be changed anyway, but also perhaps not the best way to solve this problem.

A better way to fix it might have been to distribute the constants into the classes where they truly belong. But because we were very unsure about what those classes would be for each of the constants, and also because a lot of constants are used in multiple different classes, we decided against it.

Methods in Possibly Wrongful Places

We had a few arguments about where certain methods and certain parts of methods belonged, and how methods should be split up, and since neither of us are experienced enough with coding and architecture to always truly know where everything belongs, there is always the chance that we messed up in a few places.

If we had had more time (and more experience of course) we would probably have discovered more methods and parts of methods out of place, that we could fix.

Possible design patterns:

There were several patterns and decisions we had to cut from our refactoring due to the lack of time and understanding of the Java Applet class. There were also other design choices we tried to implement but failed to completely integrate due to the lack of in depth technical understanding of how certain concepts worked, such as Threads. As our refactored program does not run completely, we have decided to explain our ideas and patterns here.

Factory

A possible design pattern that could be implemented for this type of program would be the Factory method. With this we could create different objects based on the situation of the game. In a game there are always different objects in different places all around the screen and what these objects are going to be depends on many different factors in the game. Using this we could easily get a hold of the different objects like small asteroids, big asteroids, ships and so on.

Singleton

Another design pattern we could have used for certain objects in game would be the singleton. In the game, there can only be one Ship and Missile at any single point in time. If we had more time, this pattern could be implemented for objects like these, so that we can ensure that only one instance of the ship is run at a time.

Another object that can implement this pattern is Thread. While there are multiple threads running in our system, they are all distinct Threads running different things. There can also only be one of each of these distinct threads, or else there can be multiple instances of the game running without us knowing.

Observer

We also thought of ways to implement the observer pattern, especially when it came to user input. In the game, when user input is given, there are many things happening in the background. Updates for each object are made, and depending on what happens the screen is then updated. Thus, the screen has to be constantly updated on what is going on in the game logic to know what to show. It was too difficult for us to implement, however, and we thus decided to drop this idea.

Facade

While our subsystem is not entirely too complex, we tried to integrate this pattern using our class GameLogic, as it contained the core logic which also meant that it would be the only one to access the objects in the domain layer. We tried to then have elements in the UI layer only use GameLogic, and only GameLogic as much as possible.

A closer attempt of this pattern was the Sounds Object in the domain package. As we tried to map each sound to their respective objects, the objects in the domain class would have to directly reference their own raw sound file if we hadn't created a sounds class, which had references to all the sounds. The objects would then just get their sounds from the Sounds class. This then allowed missile..getSound().play() play a different sound than explosion.getSound().play(), for example.

Benefits and limits of our system and design solutions:

Benefits:

As mentioned above, there are multiple advantages of our system.

Encapsulation and information hiding

Encapsulation and information hiding is a big advantage of our system. As we have tried to constrict each layer to only be able to access the layer beneath it, we have increased the security of the system, making it harder for the user to modify certain parts of the code.

Code structure

Giving the code structure also allowed easier readability and modularity. We can now easily identify which classes contain information about those objects, which classes control or coordinate other classes, and which classes display the information needed to the user. This allows easier maintenance down the road.

Responsibility driven design

Separating classes by responsibility also ties in the idea of greater modularity and ease of maintenance. Before, changing one attribute would mean having to change other methods and several other places as well. However, now that everything is sectioned, we only need to change one piece of the code instead of several.

Limited duplicates

There are also less duplicate code now that everything has their own responsibility. There are still existing duplicate code due to the way the original methods were made, and we were not able to find another way to perform the same method due to the lack of time and experience. However, as we know which classes contain which information, we simply need to reference that now instead.

Limitations:

Low scalability

One limitation we quickly realised with Layered architecture was how much code we had to add in comparison to the original file. With the original class having 1500 lines of code, our refactoring quickly made it past 1500 lines in total. We realised quickly how difficult it would be to scale this application if we wanted to make the game bigger and add more functionality. More classes would have to be made to both coordinate, control, and contain the new information that we would have to create.

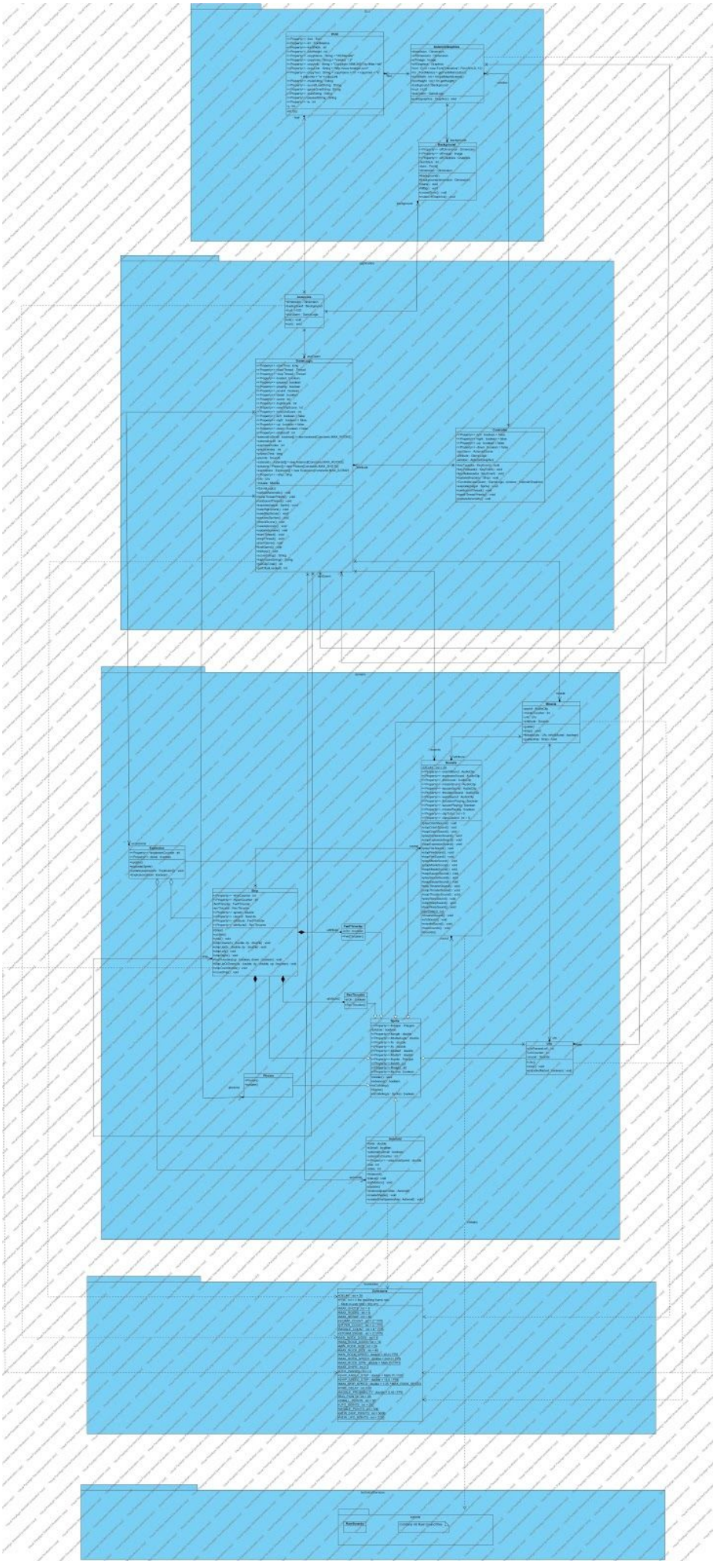
Decreased readability

Separating responsibilities into different classes with private variables also presented another problem: longer lines of code to describe the same thing. As we had to relay information from one class to another, obtaining the same attribute had to go through getters and setters, along with instances of other classes. While this is not too much of a problem, it does reduce readability in some places.

Risk of High Coupling

Tying in with how quickly the application can grow in size, the complexity of the program will also (probably) scale proportionally with the horizontal growth of the layers in the program. More dependencies, compositions, and relations will have to be added. As a result, the risk of high coupling can be very high if the application is not managed carefully.

Class diagram



Brief description:
The diagram is in the form of a layered architecture, where the things more closely related to the system itself are in the upper layers, and things more loosely related to the system are in lower layers.
In the GUI layer we have the classes related to user interface.
In the Application layer we keep the logic of the game, it holds the classes that control the different objects in a cohesive manner.
In the Domain layer we put general game objects used by the system.
In the Foundation layer we keep the constants that the game uses.
In the Technical services layer we keep the raw sound files needed to get some sound to the game.

Imported classes from the Java library: Applet, Runnable, KeyListener