CS4227: Software Design And Architecture.

# Semester 1 – 2015/2016

Superhero VS Superhero simple Game Framework

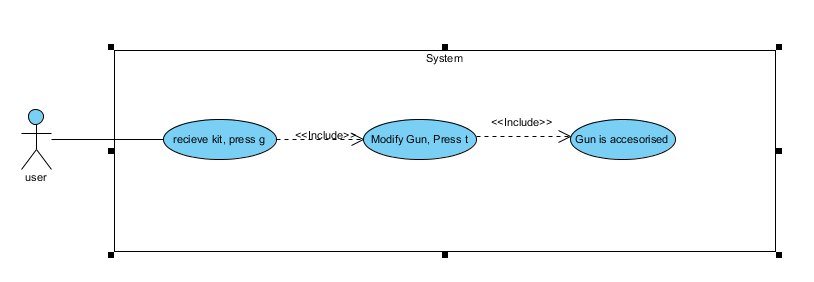
<http://screenrant.com/batman-v-superman-movie-fight-winner/>

**Description of Project**

The framework that I decided to produce is based on a game. The user starts the game and then chooses a character and what colour he is wearing. A character is then created. The user’s character then receives a kit with weapons. The user then picks up a gun. The user then has a choice of accessorising his gun with different type of scopes, magazines etc.… When the gun is assembled, then the user’s character can fire the weapon. Each time a bullet is fired, the user is told how many bullets are left in the magazine. The user can also pause and finish the game with different commands.

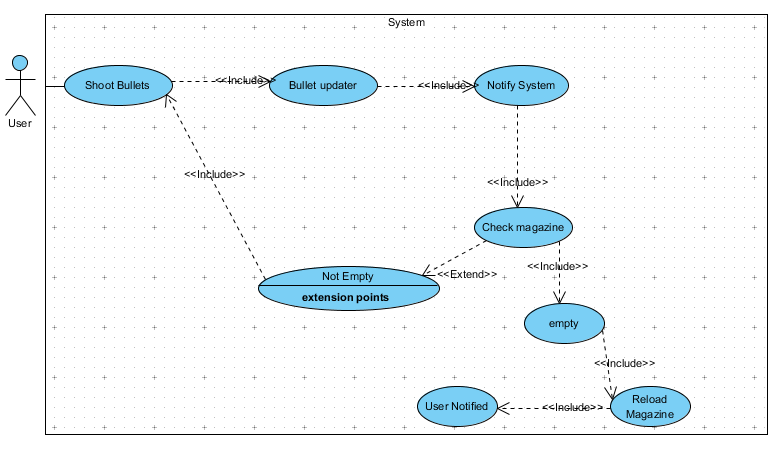
**Use Cases**

|  |  |
| --- | --- |
| Title | Accessorise Gun |
| Actors | User |
| Goal in Context | When the user picks up the gun, user chooses accessories to update the gun. |
| Precondition | Accessories are already attached to kit. |
| Success Postcondition | Accessories are attached to gun. |
| Failed Postcondition | Accessories not attached to gun. |
| Trigger | After receiving kit. |
| Description | Action |
| 1. | Character gets kit.(press g) |
| 2. | Character chooses accessories from kit for gun. (press t) |
| 3. | Gun now has accessories attached. |



Use Case

|  |  |
| --- | --- |
| Title | Updating magazine |
| Actors | User |
| Goal in Context | User shoots the gun, and the bullet count in the magazine is reduced until empty. Then reloads magazine. |
| Precondition | User has magazine attached |
| Success postcondition | Bullet count reduced from magazine once fired. |
| Failed postcondition | Bullet count does not reduce from magazine once fired. |
| Trigger | Gun fired |
| Description | Action |
| 1. | User shoots gun |
| 2. | Check magazine |
| 3. | Reduce bullet by one |
| 4. | User can fire again |
|  |  |
| Variations | Action |
| 4 | The magazine is empty, reload magazine |
|  |  |



**Architectural use cases**

Accessorise:

This can be used by many action games to accessorise their weapon. Games like call of duty have accessories for weapons such as a scope and it is up to the user if he wants to use it. It is also useful for accessorising any object. An example would be a game that wanted to add decorations to a cake (i.e. pick what you want to add) etc…..

Shooting:

The framework can help a game to update their bullets in their gun. This is again useful for action games where the game is made more realistic by running out of bullets. When a player fires a bullet, then the player can see how many bullets he has left.

**Non Functional Requirements**

**Extensibility**: This will be important in my project because I must take future growth of the framework into consideration. I intend to use the factory method to help me improve extensibility. In the system i must make sure that clients can add new characters in the future, while minimizing impact to existing system functions.

**Reliability**: Will my framework perform its required functions under stated conditions for a specific period of time. When the framework gets integrated with third party clients systems, will it perform over a set period of time?

**Performance**: This concerns the speed of the framework. How quick will it take for the system to respond to the end user? How long will it take the system to update the magazine once the bullet is fired? Also will the framework be available for service when requested by end-users?

**Maintainability**: The code for this game must be maintainable. That is why we should use programming to interfaces. This makes it easier when changes have to be made.

**Testability**: I want the framework to be tested. I have to test each feature of the game. That includes creating character, accessorising gun and updating magazine once gun is fired. If one of these fails, then the user can find this problem and hopefully resolve it.

**Scalability**: This is very important in the project. I have designed this framework so as that it would be easy to make it bigger. Will use the decorator pattern for holding the different accessories in the kit bag. If other users of the framework want to add more items to this kit, then this is possible. It should be easy for the client to make the world of the game bigger.

**Reusability**: The framework should be reusable by third party clients. I hope that certain areas can be reused on their own (i.e. low coupling).

**Discussion on Architectural and design patterns**

**The Interceptor architectural pattern**

The interceptor allows users of the framework to change the behaviour of the framework with their own. The interceptor created will make the changes by intercepting at the interception point. When an event occurs, the framework informs the appropriate dispatcher to invoke the callback of the registered concrete interceptor. Context objects can then be passed to concrete interceptors when they are dispatched by the framework. I will use this in the project by creating log files. I will have an interceptor between an invoker and the command class. When a user starts the game, this action will be logged to a file and then the game will start. If the user pauses or ends the game, then the interceptor will also be used to log the file. The advantages for using the interceptor are; that it promotes extensibility and reusability. (These are important NFR’s in the framework).

**The Bridge pattern**

The Intent of the bridge pattern as described by the gang of four is to “decouple an abstraction from its implementation so that the two can vary independently.” I will use this pattern when creating the character for the game. When the character is chosen, then the user has a choice of what colour to have the character. I will have a class called colour which has the method to applyColour(). There will then be other classes (i.e. classes such as blueColour or redColour) which will implement the method applyColour() from the colour class. This makes it easier to add more colours of needed when the framework is implemented by a third party client. The bridge is then also used by the character factory class where the character is created (abstraction).The bridge pattern thus uses inheritance to separate each different colour into different classes.

**The Command Pattern**

The command pattern “encapsulates a request as an object, thereby letting you parametrise clients with different requests, queues or log requests, and support undoable operations” GOF (1995). Succinctly the command pattern encapsulates all the information needed to execute an action at a later time. There are four terms associated with the command pattern. They are command, receiver, invoker and client. The client decides which actions to perform and when. The command implements the execute() performance by invoking the action on the receiver. The invoker tells the command to do the request after an event has occurred. The receiver actually does the work and performs the specific program. I will use this pattern when showing that the game can be started, paused and stopped. When the user wants to start or pause the game, the notification is sent to the system and the action will be performed.

**The Decorator Pattern**

This pattern “allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.”1 I will use a weapon accessory decorator class in this framework, so as to provide a pliable alternative to sub classing. In the framework the weapon accessor class will implement the weapon class. I do not want to modify the weapon class, so I used the decorator class so as if third party clients ever wanted to add anything new to the weapon accessory, it would be done in the decorator class. The decorator pattern creates a decorator class which envelops the original class (weapon) and provides supplementary functionality keeping class methods signature complete. In conclusion I used the decorator method because I wanted the weapon class to be open for extension, but closed for modification if possible. I believe this will be a good principle to keep in mind, as it will keep the class stable, but also leave it open for extension if third party clients wants to add extra behaviour in the future i.e. the open/close principle.

**The Factory Pattern**

I will also use the factory method in the framework. The factory methods intent is to define an interface for creating an object, but let subclasses decide which to instantiate. The factory method lets a class postpone instantiation to subclasses. I will use this when creating the characters for the game (character factory class). The advantage will be that new characters can be added without changing a single line of code in the framework. You just have the change the code in the factory class and nowhere else. If a character is changed then only this class needs to change to reflect this. This will again help with extensibility and make it easier for third party clients to add to their system if using the framework.

**The Observer Pattern**

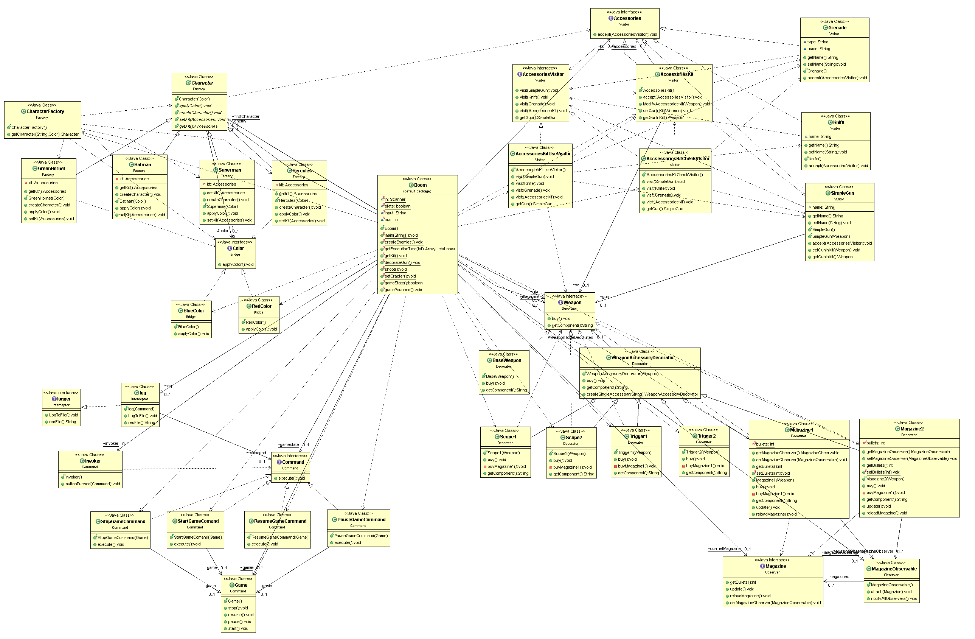
The observer pattern is about one to many relationships. The dependents are informed when a subject changes. The observer pattern is made up of three main classes; the observer, the subject and the client. The subject will maintain a list of the observers and can add or remove them from the list. Once notify is called, the subject will go through the list and call on update the observers that it is integrated with. The observers then become aware of this change and can update accordingly. The observer has an interface which has an update declaration and the rest of the classes can use this declaration. In the framework I will use the observer to observe when bullets are fired from the magazines. I will have two different magazines, one with 20 bullets and the second with 30. When a bullet is fired, the magazine is notified and the count of bullets in the magazine is updated.

**The Visitor Pattern**

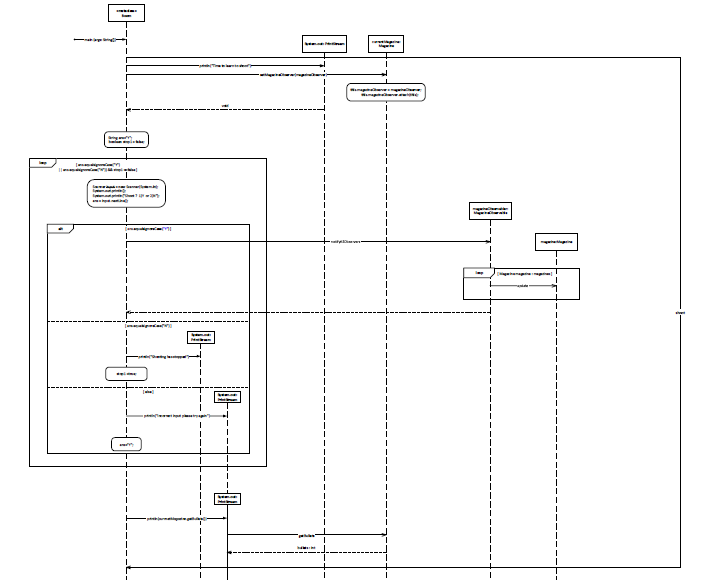
The intent of the visitor pattern is to “Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”2 I am going to use the visitor pattern to visit the accessories in the game framework. I am going to create an accessory interface defining the accept operation. Grenade, knife, kit and gun will be concrete classes that implement the accessory interface.I will also define another interface called accessoryVisitor which will define a visitor class operations. The accessory then uses concrete visitor to do the corresponding action.

**Structural Diagrams**

**Class diagram**

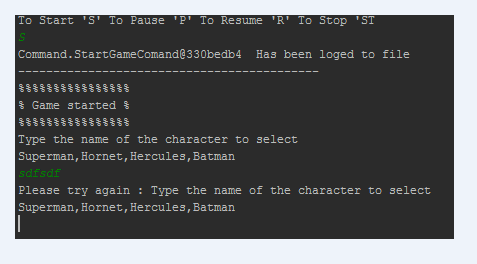


**Sequece diagram updating magazine use case**



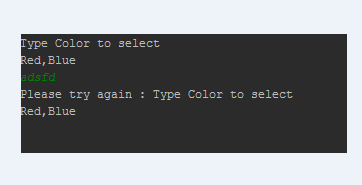
**Testing**

**Design Test Cases** 1**.** Type incorrect Character name:



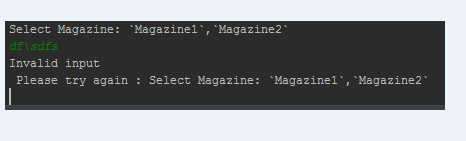
Result: User types in wrong character and gets a message saying that it is incorrect.

2. User types incorrect colour.



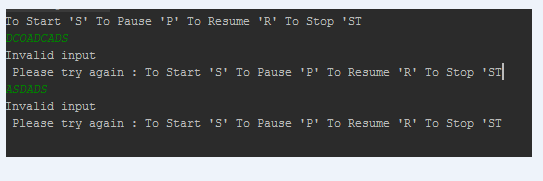
Result: User types in wrong colour and gets message saying that it is incorrect.

3. User types incorrect magazine.



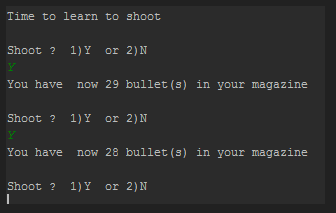
Result: User types in correct information and gets message saying that it is incorrect.

4. User types incorrect letter to start/pause/stop game.



Result: User types incorrect information and gets message saying that it is incorrect.

5. Is the magazine updating once the user fires a bullet.

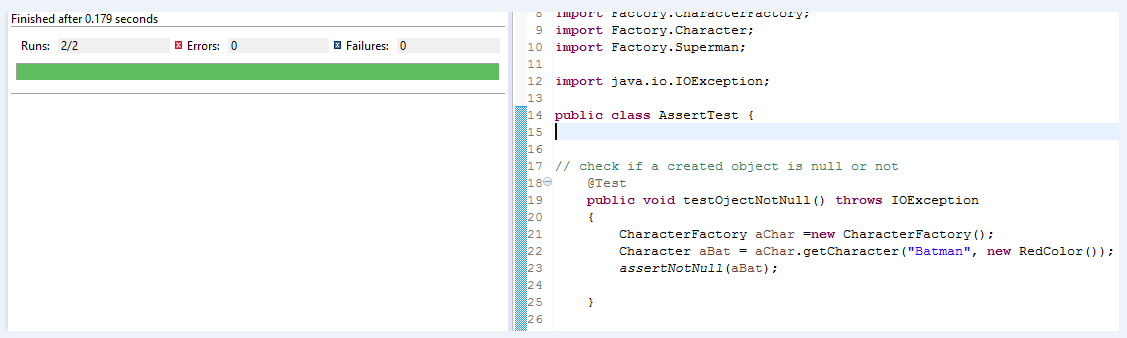


Result: User shoots a bullet and the magazine gets updated correctly.

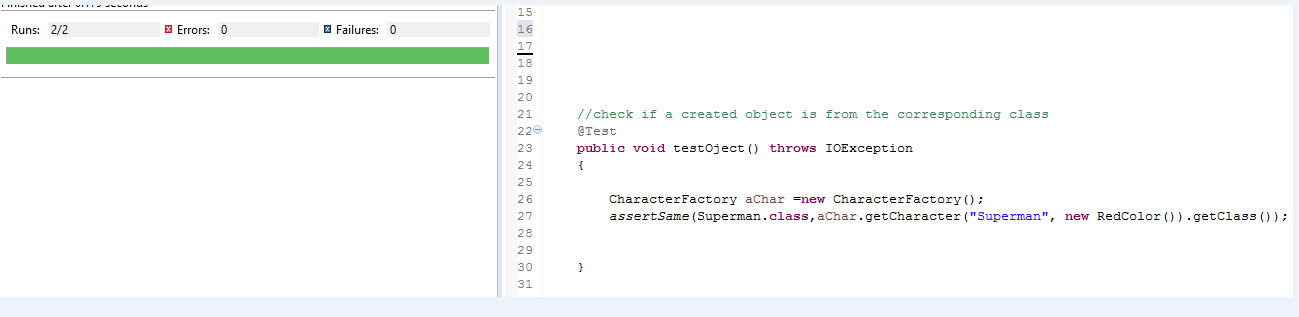
**Automated testing**

I used J-unit to test the frame work. I used this because it is very good for repeatable testing. You can test a unit of your frame work to make sure that it works correctly. The results can be seen below.

Test 1



Test 2



**Analysis**

I used j-unit in the character factory class. The first test was to test that the created object is null or not (i.e. if null the test should fail). The result was that the j unit test passed in the character factory class and that the object was not null. We also tested that the created object is from the corresponding class. Again the framework passed this test as can be seen from above.

I also did manual testing which tested that the program responded to the user’s input. If the user entered the correct information during the game, that it worked correctly. Also if the user entered the wrong information, then the system should return a message to the user to make him/her aware of their mistake. The results have been entered in the above section on design test cases.

**Evaluation / Critique of NFR’s**

*Extensible*: The patterns chosen in this framework have integrated well in the final product. I can extend the framework by the use of the factory method. This means that more characters can be added to the game without having to change lines of code in different classes. The change needs to be done in only one class. Also new magazines can also easily be added to the system without little trouble.

*Scalability*: I also made the system scalable by the use of the decorator pattern. This allows for further accessories to be added on if needed in the game. So that if I had multiple characters in the future, each character could have different accessories with them.

*Reusability*: Parts of the framework can be used on other systems. An example of this reusability is the command pattern.I use the command pattern to start, pause and stop a game. This can be used in another system because of low coupling of this pattern in the framework. This action is useful because it is used in a lot of games.

*Testability*: I validated the user’s input in the last few days of this project, so as to make sure that the framework would pass the test. The programme doesn’t have many actions, so it was important to make sure the users input was correct. If it was not correct, then the user was made aware of their mistake and told to try again. I wanted to make sure that the simple actions in this framework actually worked and was informative to the user in actual time.

Maintainability: The framework is also easily maintainable. It should be easy to maximise this frameworks life. Also if some component does break down or becomes obsolete, the working parts of the framework should still be available to use. There should be little overhead associated with this framework in the future.

*Critique*: If I had more time for this project I would have introduced more actions in the framework such as multiple characters fighting each other and different types of weapons and how to use them. However I have created a framework that can be used by many action games. The counting of bullets in the magazine, is something that is used in most games with guns. In conclusion, I showed that this simple framework integrated seven design patterns and showed how the framework could be used by a third party client.

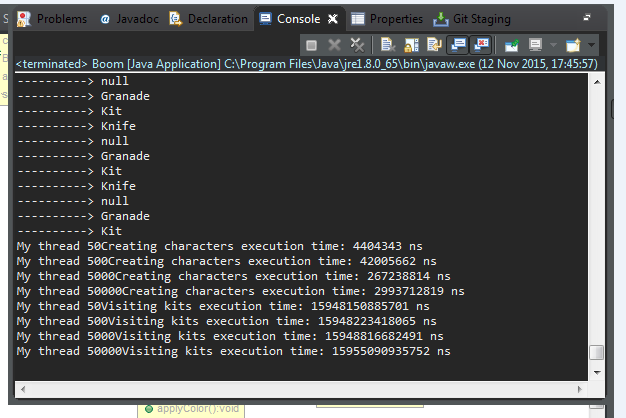
**Performance Engineering**

**Tactics to support performance engineering**

*Open MP: “*OpenMP is a set of compiler directives and callable runtime library routines that extend Fortran (and separately, C and C++) to express shared memory parallelism.”4 OpenMp is an API for writing multithreading applications. Threads communicate by sharing variables. OpenMP is useful in parallelising existing code one part at a time. OpenMP is very useful for shared data, however this can be also a negative as you must think what data you want private. It has easy to use directives but is not suited to non-structured data like trees etc.… I did not use OpenMP because the framework uses java language.

*Java Thread:* The thread is used when the program is executed. The java virtual machine allows multiple threads to execute together concurrently. Every thread has a priority. Some threads have high priority while others have low ones (i.e. which thread should execute first). To identify what code the thread should execute is to pass an object that implements *runnable* interface to the thread constructor. There are some important thread methods in the thread class which include Start(), run(), setPriority() and interrupt() etc.… one must also be aware of java synchronization when multiple threads work concurrently and try to access the same resources, then this can lead to unforeseen errors. I decided to use this tactic to support performance engineering because if fitted my java framework the best.

**Real time for sequential version.**

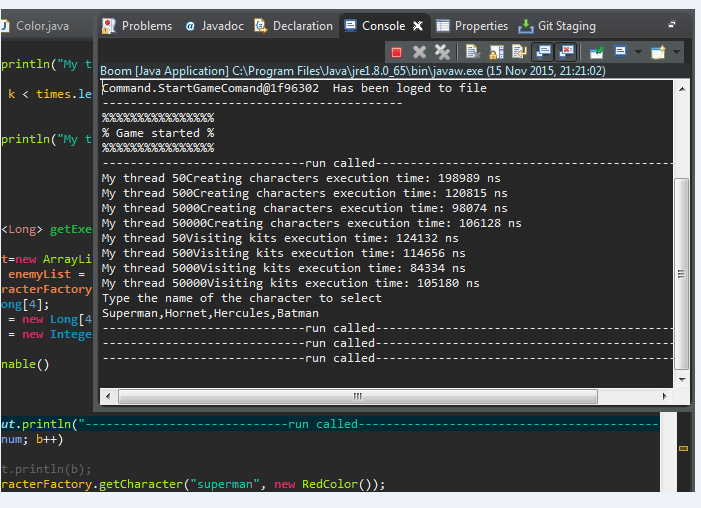


This screen shot shows the threads running sequentially for characters creating and visitors visiting the kits.

**Justification of code for selected for the Java thread.**

The framework did not have much work for java threads. So I decided to create multiple different characters and visitors to the kit independent from the framework. I created the method in the boom class of the framework. I saw the thread time for 50, 500, 5000 and 50000 characters created.

**Runtime for concurrent version.**



**Analysis of performance gain/loss.**

The results above show that the running time improved with the concurrent thread. The time for 50 characters created with sequential threading was 4404343 Nano seconds. The concurrent thread’s time improved sharply for the 50 characters created to 198989 Nano seconds. In conclusion there was a gain in the use concurrent threading.

**References**

# Tutorials point - Design Patterns - Decorator Pattern <http://www.tutorialspoint.com/design_pattern/decorator_pattern.htm> [online] [accessed 14-10-15]

# Source making – Visitor design pattern. <https://sourcemaking.com/design_patterns/visitor> [online] [accessed 14-10-15]

# ReadWrite – Github for Beginners <http://readwrite.com/2013/09/30/understanding-github-a-journey-for-beginners-part-1> [online] accessed [06-11-15]

# Openmp – What is openMP? <http://www.openmp.org/mp-documents/paper/node3.html> [online] [accessed 14-11-15]

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

# JJ Collins notes CS4227 <http://www.csis.ul.ie/coursemodule/CS4227> [online] [accessed through out semester 1 2015/16]