Architectural Design Document

Introduction

With an aimto build a simplistic version of a Dungeons and Dragons role-playing game, we implemented a Model View Controller (MVC) architectural design model along with Observer pattern. It was an effort to use extreme programming effort with iterative software development approach to make a modular design and deliver several working coherent modules in small increments or builds.

Model View Controller

Model View Controller design aims at decomposing an interactive system into three components, namely: Model, View and Controller.

Model - The model represents data and the rules that govern access to and updates of this data. In enterprise software, a model often serves as a software approximation of a real-world process.

View - The view renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This can be achieved by using a push model, in which the view registers itself with the model for change notifications, or a pull model, in which the view is responsible for calling the model when it needs to retrieve the most current data.

Controller - The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks or menu selections. Depending on the context, a controller may also select a new view -- for example, an action on a particular click event may result in rendering a completely new view.

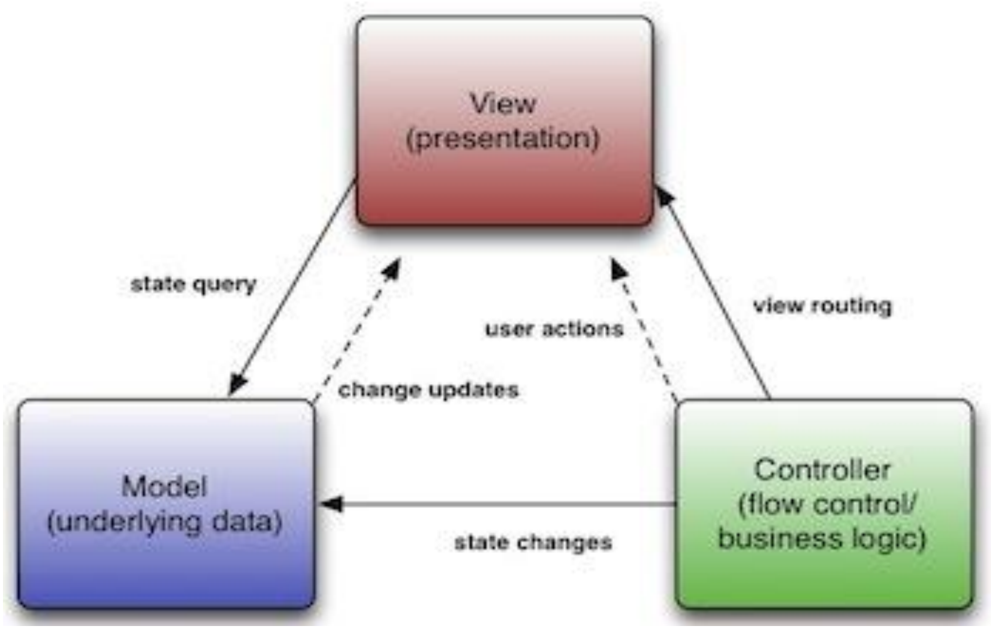


Fig1. Basic MVC architecture

As the figure 2 goes, this is how MVC has been implemented in the project.

- Various **Models** (mapModel, characterModel, etc) manages the behavior and data of the application domain. Once it gets a change state query request from the various Views (mapView, characterView, etc) that are registered to the model, they respond to instructions to change the state from various controllers (mapController, characterController, etc).
- Here we have built an event-driven system where the model notifies observers (usually views) which have been registered to the models, whenever there is change in information or state, so that they can react.

- **Views** on the other hand renders the model into a form suitable for visualization or interaction, in a form of UI (user interface). If the model data changes, the view must update its presentation as needed.
- Here we have developed a push model where view registers itself with the model for the change notifications, thus following observer pattern.

- **Controllers**are designed to handle user input and initiate a response based on the event by making calls on appropriatemodel objects.Thus accept various input from the user and instruct the model to perform operations.
- The controller translates the user's interactions with the view it is associated with, into actions that the model will perform that may use some additional/changed data gathered in a user-interactive view.
- Controller is also responsible for invoking new views upon conditions.

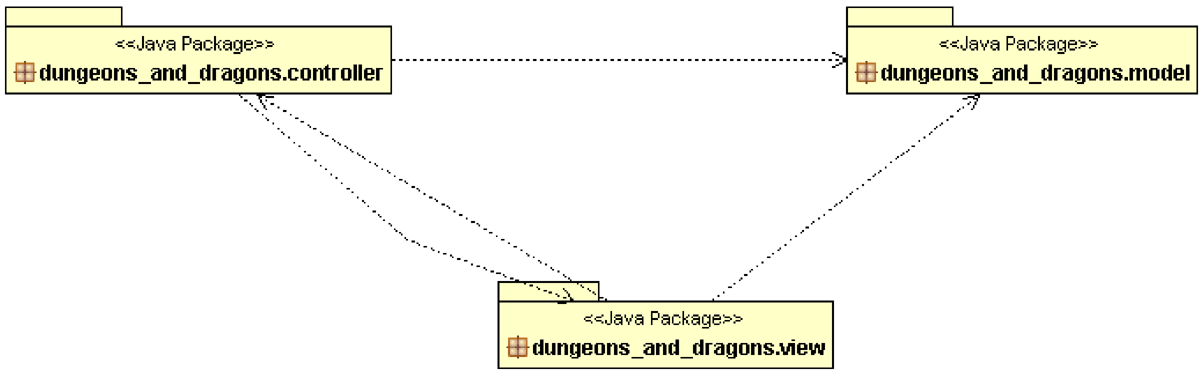

<<Java Package>>
dungeons_and_dragons.controller
<<Java Package>>
dungeons_and_dragons.model
<<Java Package>>
dungeons_and_dragons.view

Fig. 2 Class Diagram for overall MVC structure

Following the layout of figure 2 the figure3, 4 and 5 describe the actual class diagrams inside each and every packages. Viz. Model Package in figure 3, View Package in figure 4, Controller Package in figure 5.

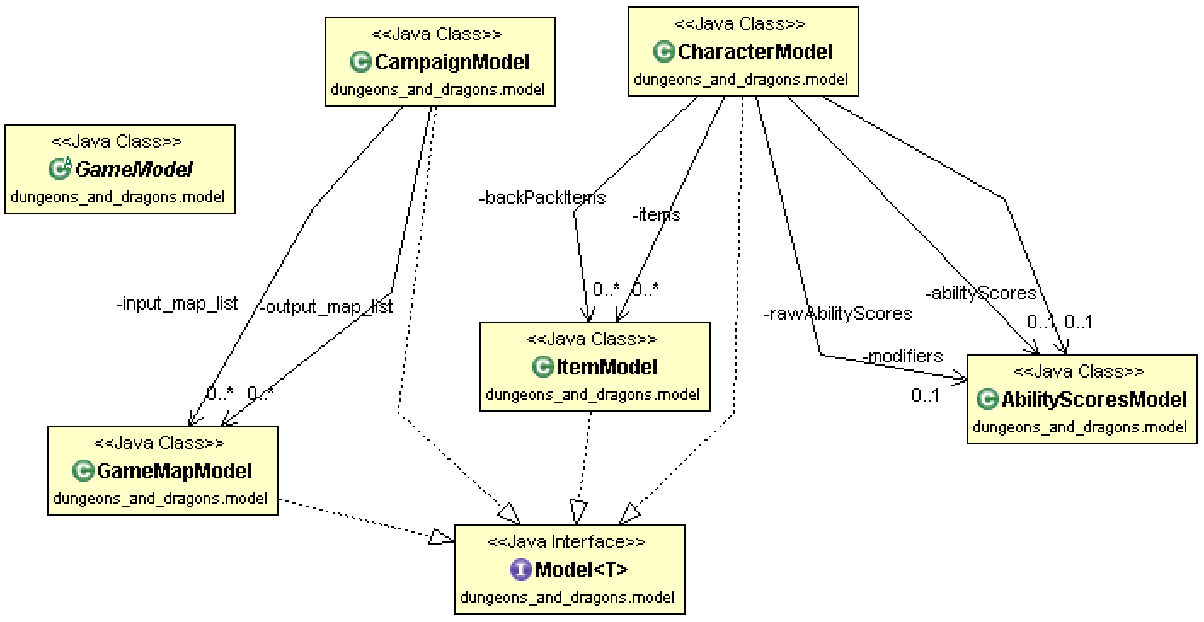


Fig.3 Model Class Diagram

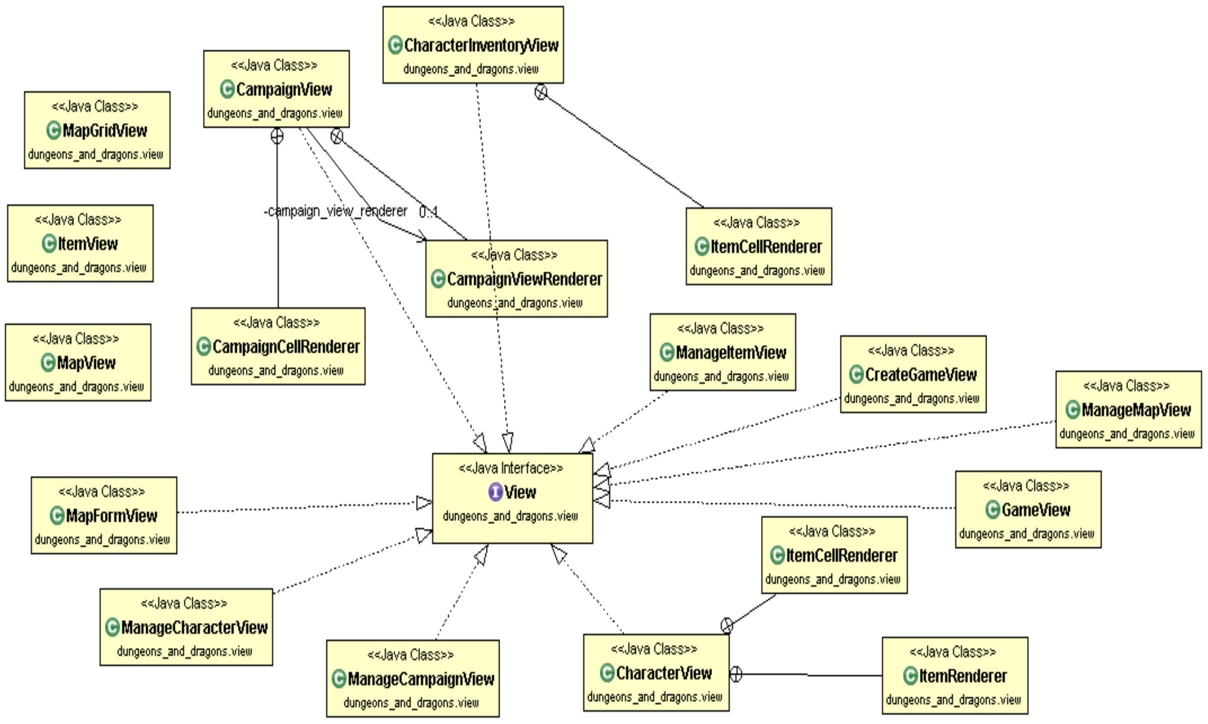
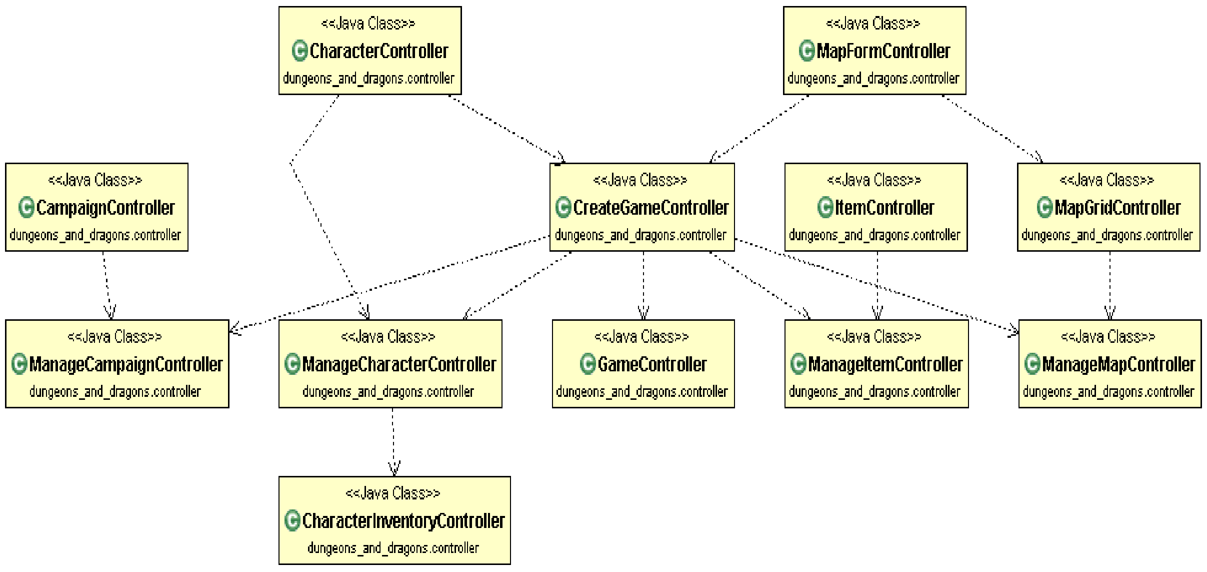


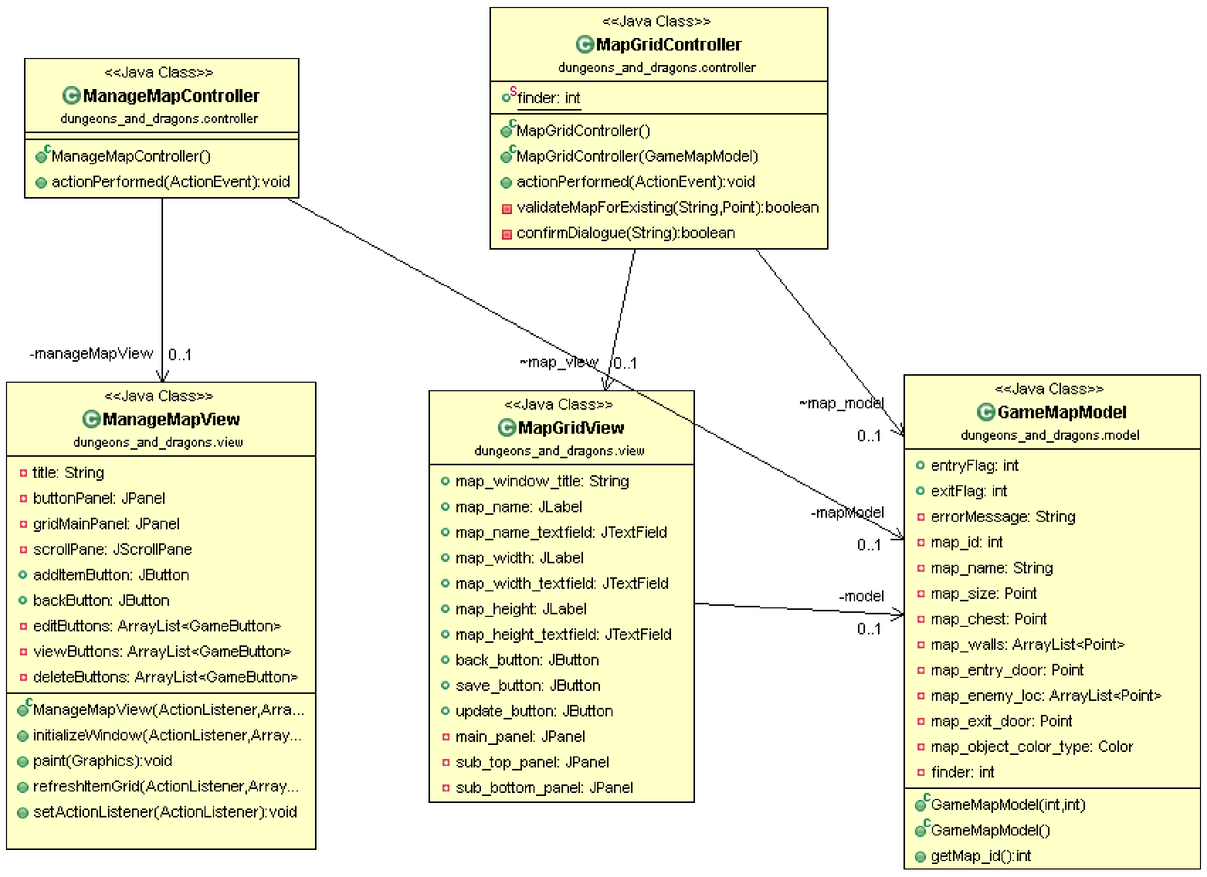
Fig.4 View Class Diagram



<<Java Class>> CharacterController dungeons_and_dragons.controller
<<Java Class>> MapFormController dungeons_and_dragons.controller
<<Java Class>> CampaignController dungeons_and_dragons.controller
<<Java Class>> CreateGameController dungeons_and_dragons.controller
<<Java Class>> ItemController dungeons_and_dragons.controller
<<Java Class>> MapGridController dungeons_and_dragons.controller
<<Java Class>> ManageCampaignController dungeons_and_dragons.controller
<<Java Class>> ManageCharacterController dungeons_and_dragons.controller
<<Java Class>> GameController dungeons_and_dragons.controller
<<Java Class>> ManageItemController dungeons_and_dragons.controller
<<Java Class>> ManageMapController dungeons_and_dragons.controller
<<Java Class>> CharacterInventoryController dungeons_and_dragons.controller

Fig.5 Controller Class Diagram

Figure 6 shows one particular class diagram (of many such possible class linkage possible) where in it is shown how MVC structure is implemented in the actual sense. With each and every package of model controller and view linking with each other in event-driven system.



<<Java Class>> ManageMapController dungeons_and_dragons.controller
ManageMapController()
actionPerformed(ActionEvent):void
<<Java Class>> MapGridController dungeons_and_dragons.controller
finder: int
MapGridController()
MapGridController(GameMapModel)
actionPerformed(ActionEvent):void
validateMapForExisting(String,Point):boolean
confirmDialogue(String):boolean
-manageMapView 0..1
~map_view 0..1
~map_model 0..1
-mapModel 0..1
-model 0..1
<<Java Class>> ManageMapView dungeons_and_dragons.view
title: String
buttonPanel: JPanel
gridMainPanel: JPanel
scrollPane: JScrollPane
addItemButton: JButton
backButton: JButton
editButtons: ArrayList<GameButton>
viewButtons: ArrayList<GameButton>
deleteButtons: ArrayList<GameButton>
ManageMapView(ActionListener,Arra...
initializeWindow(ActionListener,Array...
paint(Graphics):void
refreshItemGrid(ActionListener,Array...
setActionListener(ActionListener):void
<<Java Class>> MapGridView dungeons_and_dragons.view
map_window_title: String
map_name: JLabel
map_name_textfield: JTextField
map_width: JLabel
map_width_textfield: JTextField
map_height: JLabel
map_height_textfield: JTextField
back_button: JButton
save_button: JButton
update_button: JButton
main_panel: JPanel
sub_top_panel: JPanel
sub_bottom_panel: JPanel
<<Java Class>> GameMapModel dungeons_and_dragons.model
entryFlag: int
exitFlag: int
errorMessage: String
map_id: int
map_name: String
map_size: Point
map_chest: Point
map_walls: ArrayList<Point>
map_entry_door: Point
map_enemy_loc: ArrayList<Point>
map_exit_door: Point
map_object_color_type: Color
finder: int
GameMapModel(int,int)
GameMapModel()
getMap_id():int

Fig.6 Map architecture class diagram

Builder Pattern

The builder pattern is an object creation software design pattern. Unlike the abstract factory pattern and the factory method pattern whose intention is to enable polymorphism, the intention of the builder pattern is to find a solution to the telescoping constructor anti-pattern. The telescoping constructor anti-pattern occurs when the increase of object constructor parameter combination leads to an exponential list of constructors. Instead of using numerous constructors, the builder pattern uses another object, a builder that receives each initialization parameter step by step and then returns the resulting constructed object at once.

Builder - Specifies an abstract interface for creating parts of a Product object. As shown in the figure 7, in our case FighterBuilder is the Builder interface.

ConcreteBuilder - Constructs and assembles parts of the product by implementing the Builder interface. Also, it defines and keeps track of the representation it creates and provides an interface for retrieving the product. In our case ConcreteBuilder are BullyFighterBuilder, NimbleFighteBuilder, TankFighterBuilder.

Director - Constructs an object using the Builder interface, which in our case is Explorer class.

Product - Represents the complex object under construction, which in our case will be an object of a FighterBuilder which is CharacterModel in our case.

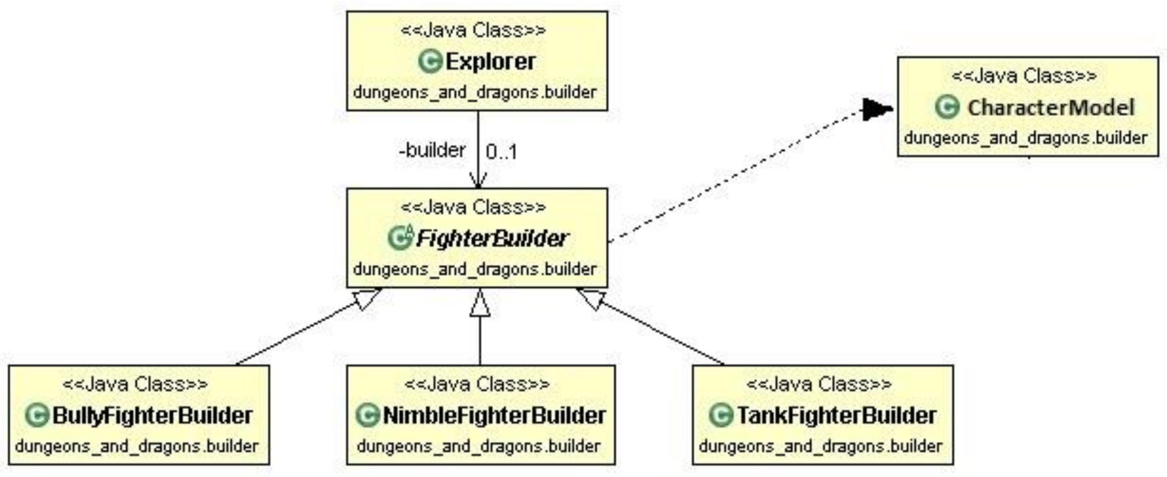


Fig.7 Builder Pattern class diagram

Importance:

The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.

The construction process must allow different representations for the object that is constructed.

As the fighterBuilder types are common though they differ internally with respect to their behavior and structure and thus builder pattern goes best with the flow.

Strategy Pattern

Motivation:

Sometimes we may want to change the behaviour of an object depending on some conditions that are only to be determined at runtime, or to easily add new definitions of certain behaviour without altering the class that is using it.

Intent:

Define a group of algorithms that applies to a family of classes. Encapsulate each algorithm separately. Make the algorithms interchangeable within that family

Consequence:

The Strategy Pattern lets the specific algorithm implemented by a method vary without affecting the clients that use it.

Elements of the strategy pattern:

Context Class: class that uses a certain behaviour that is to be changed during execution. It contains a Strategy object and provides a setStrategy() method to change its own strategy. The strategy is to be called through a executeStrategy() method that will delegate to a concrete strategy method.

Strategy Abstract Class: Superclass of all strategies containing the abstract executeStrategy() method to be implemented by all its subclasses.

Concrete Strategies: Subclasses of Strategy that provide a different implementation for the executeStrategy() method. In your project this concrete strategy classes are AggressiveNPS, CpmputerPlayer, FriendlyNPC and HumanPlayer.

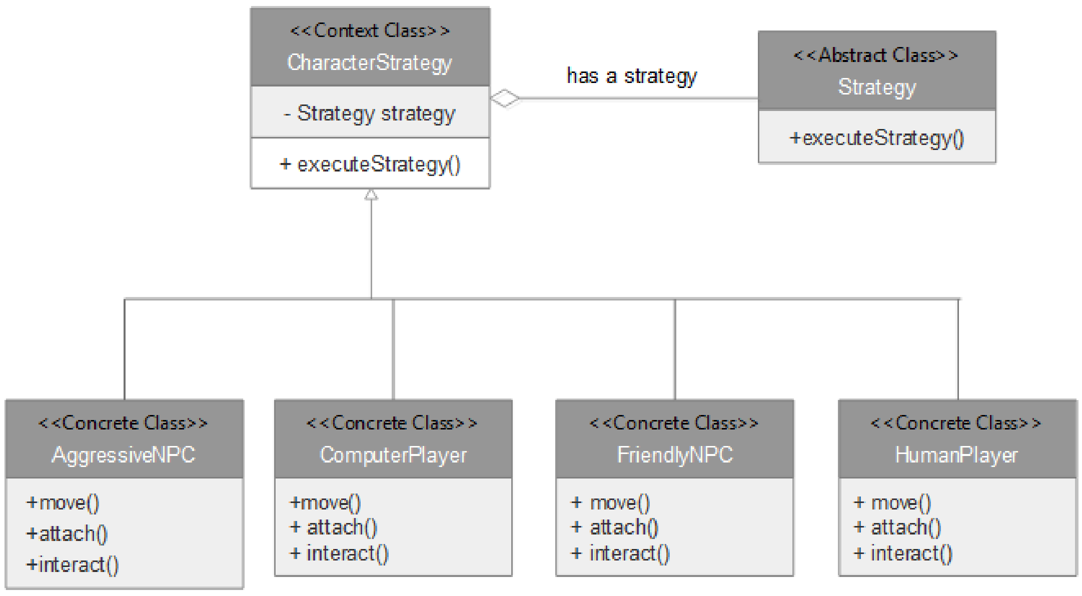


Fig.8 Strategy Pattern class diagram

Decorator Pattern

Motivation:

Sometimes we may want to dynamically add some data members or methods to an object at runtime, depending on the situation.

Intent:

Allow to add new functionality to an existing object without altering its structure. Create a Decorator class that wraps the original class. Provides additional functionality while keeping the class' methods' signatures intact.

Elements of the Decorator pattern:

Component: Abstract class representing the objects to be decorated by the various Decorators which is here represented by Weapon class.

Concrete Component: The potentially many sub-classes that can be decorated.

Decorator: Abstract class that wraps a Component and will have some of its subclasses to decorate it which here represented by WeaponsDecorator class.

Concrete Decorator: Different decorators that add different members to the Component. For our project these concrete decorator classes are Burning, Freezing, Frightening, Pacifying and Slaying.

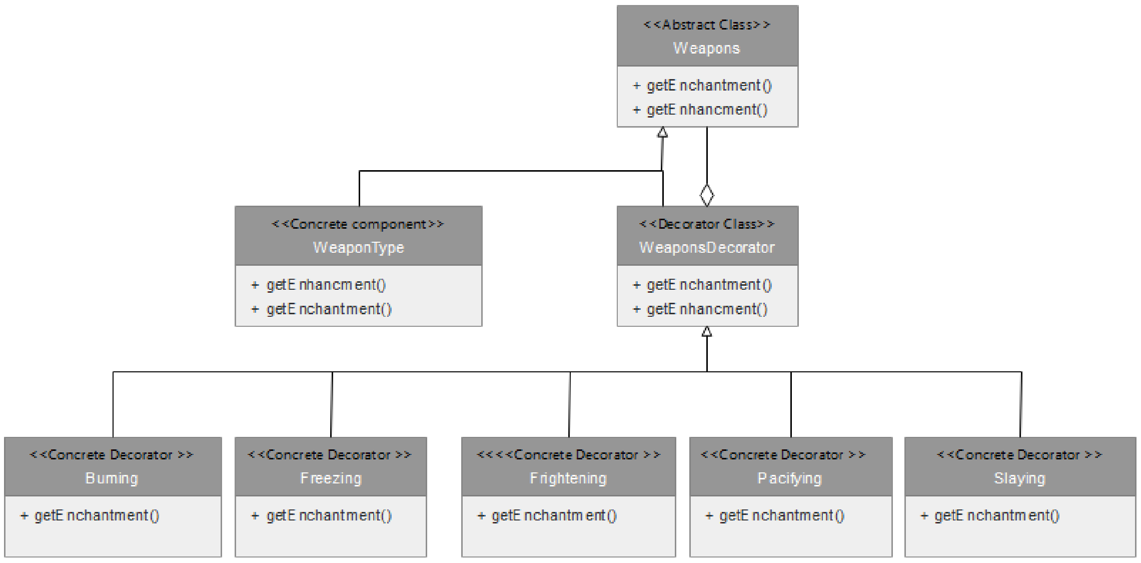


Fig.9 Decorator Pattern class diagram

References

- http://www.oracle.com/technetwork/articles/javase/index-142890.html
- http://www.java-forums.org/attachments/ocmjea/3449d1333636384t-tutorial-review-web-tier-application-architecture-java-architect-exam-c5-conceptualmvc.jpg
- https://en.wikipedia.org/wiki/Builder_pattern