Team Number: Team 2

Team members: Daniel Pobidel, Radosław Konopka, Trung Minh Tri Nguyen, XiaoWei Chen

- Pattern name: **Composite**
    - CellComponent (Component), CellComposite (Composite), Cell (Leaf), Tower (Leaf), EnemyPrototype (Leaf)
    - The benefit of using the Composite pattern in our application is that it groups every smaller component together into a tree structured object and treats them as a single large component. In our application, the CellComposite is used to group the cell components and allows the invoking of a method on multiple components at once. With the Cell Class, it holds the position of a tower or an enemy object that is placed on the board. And the CellComposite class is used to group the Enemy, Tower, and the Cell objects. The CellComponent provides a common framework for CellComposite and the leafs.
- Pattern name: **Abstract Factory**
    - <<Interface>> AbstractFactory(AbstractFactory), EnemyFactory(concreteFactory), TowerFactory(concreteFactory)
    - The benefit of using the AbstractFactory patterns in our application is that it provides an interface for creating related objects without defining any concrete classes. In our application, we used AbstractFactory Interface to define the method and provide us the ability to create a CellObject (Tower or Enemy). The EnemyFactory is used to create a specific enemy, fast or slow. With TowerFactory, it will create and return a specific tower, either a weak tower or a strong tower.
- Pattern name: **Decorator**

1. EnemyPrototype(Component), EnemyDecorator(Decorator), HealthDecorator(Concrete Decorator)
2. Tower(Component), TowerDecorator(Decorator), SpeedTowerDecorator(Concrete Decorator A), DamageTowerDecorator(Concrete Decorator B)
    a. The benefit of using the decorator patterns in these two places is that it allows us to be flexible with making changes to the components. In our board class we are dynamically modifying these components by passing them through their respective decorators to return the components with their updated fields. We can increase the enemies health as the waves increase to make the game progressively more difficult. Also we use damage decorator to upgrade the towers' damage during the game.

- Pattern name: **Template**
    - BaseTower(Generic Class), WeakTower(Concrete Class A), StrongTower(Concrete Class B)
    - The template pattern allows us to localize the common behavior of tower subclasses to avoid repetitive code, leaving implementation to the subclasses (weak and strong towers). We create a template method shoot() in the BaseTower class and have different implementations for the method in each subclass, otherwise the two subclasses inherit the rest of their behavior from their template parent class. In our game a user can place a weak tower with a left click and a strong tower with a right click, where the strong tower is slightly more expensive but does more damage and has a 10% chance to shoot twice.
- Pattern name: **Prototype**
    - EnemyPrototype (Prototype), Enemy(ConcretePrototype), SlowEnemy(ConcretePrototype1), FastEnemy(ConcretePrototype2)

    - The benefit of using a Prototype pattern is that it is specifying the kinds of objects to create using a prototypical instance and creating a new concrete object by cloning the prototype that is being passed. In our application, we have the EnemyPrototype class that is used to define the structure for the enemy, and defines the clone method for the concretePrototype class to use. With the FastEnemy and SlowEnemy both as concretePrototype classes, they are used to clone the instance of enemies. In the game, we make a single enemy stronger every single round by increasing its health and then we use cloning to create multiple instances of these enemies for the next wave.
- Pattern name: **Iterator**
    - IteratorInterface (Interface) , CellList (ConcreteCollection), CellPathIterator(ConcreteIterator), LinkedList (Interface)
    - The benefits of using an Iterator pattern in the application is that it provides ways to access the element sequentially. In our application, the IteratorInterface is created as an interface for specifying the methods that are needed to loop the list. The CellList contains a list of objects and an iterator to loop through the list, and also contains a nested class CellPathIterator to which contains a linked list object for the Path Iterator. In this game, we use Iterator to get the next path for enemies to follow.
- Pattern name: **Builder**
    - BoardBuilder (Builder), Board (Concrete Builder), BoardDirector (Director)
    - The benefit of this is that we can use a builder class, BoardBuilder, to define the interface for creating our board objects. This way, we can separate the construction from its representation and be able to reuse it to construct different versions of the board, such as an easy or a hard board. We used the concrete

board builder(Board) to construct and assemble all the individual parts that make up the board, which is the product object.

- Pattern name: **Singleton**
  - Builder (Singleton)
  - The benefit of using singleton is it ensures that only one board is created and it allows it to be accessed anywhere. Important because everyone needs to access our board instance.
- Pattern name: **MVC**
  - Classes: GameGui(View), Game(Model), Board(Controller)
  - The benefit of using MVC in our application is that we could use the Model, View, Controller to handle the request from the user on the UI and display the result of that request back to the user onto the same UI but with different information. In our code, we used the Swing library in GameGui class to display UI and create components to be shown, which acts as View. From the Board class it allows the user to perform an action whenever they click on the board or on any button, which acts as Controller. And all the other classes (almost all classes in this case) that provide Objects, logic, and data act as Model.