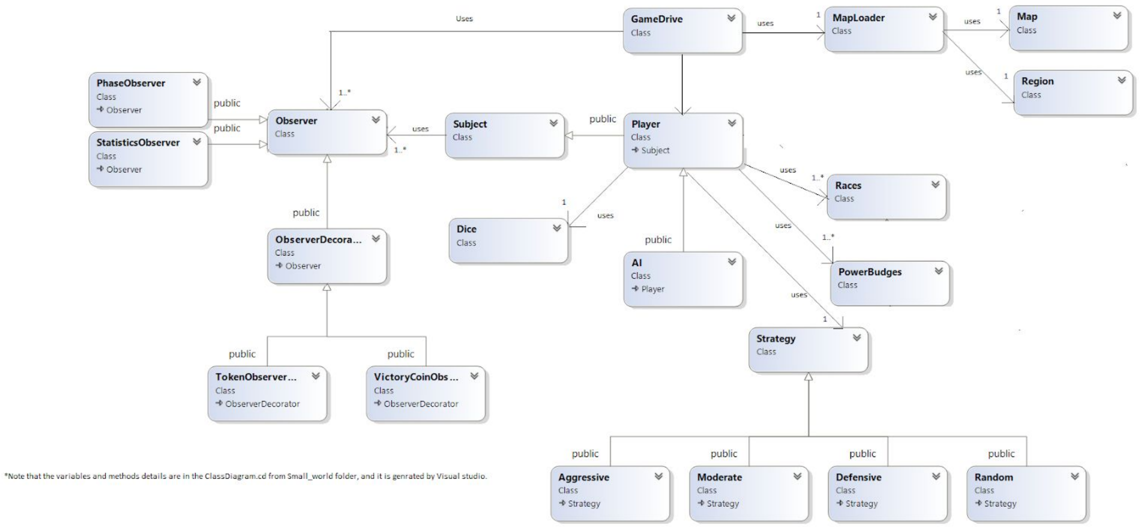
Part 3: UML and brief design decisions

a) Provide the UML class diagram of your game architecture. In which you show the classes and the design patterns used from assignment#3.
The class diagram generated by using Visual Studio is called ClassDiagram.cd. It can be found inside of the SmallWorld folder of our project

The class diagram contains associations are shown below, since the one provided by Visual Studio only generates the inheritances. The detail of the inheritances for the races and powers are from the ClassDiagram.cd. The details of variables and methods for each class can be viewed in ClassDiagram.cd.


GameDrive Class
MapLoader Class
Map Class
Region Class
Uses
uses
1
1..*
PhaseObserver Class Observer
StatisticsObserver Class Observer
public
Observer Class
Subject Class
Player Class Subject
Races Class
PowerBudges Class
Dice Class
AI Class Player
ObserverDecora... Class Observer
Strategy Class
TokenObserver... Class ObserverDecorator
VictoryCoinObs... Class ObserverDecorator
Aggressive Class Strategy
Moderate Class Strategy
Defensive Class Strategy
Random Class Strategy
*Note that the variables and methods details are in the ClassDiagram.cd from Small_world folder, and it is genrated by Visual studio.

b) Briefly explain your key design decisions, in terms of abstraction, inheritance, polymorphism, high cohesion and low coupling of your modules.

The code for generating the game is called inside of the main.cpp file, which will call the gameDrive class to run each step of the game.

In our design,we have 4 different maps that can be used for different numbers of players. The GameDrive class will call the MapLoader, which will read from a specific map.txt file and store the map information inside of the MapLoader class. And all types of the region relation information are saved inside the Map class, which has a high cohesion. A list of players is created and saved inside of the GameDrive class too. All the attributes for players are saved in the Player class, which improve the cohesion. Classes are referencing another class in a single side, which means there are no classes referencing each other.

We are using both inheritance and polymorphism for PowerBudges and Races classes, since all special races and powers have the same attributes. In order to solve a potential risk of having a slicing issue, the parent class contains all the attributes which can be modified at child classes, so that every child class can has its specific way to override it.

In order to use the observer pattern to update the information and changes during the game, we added a subject class, which contains a list of observers. The player class extends from the subject so that a player can choose to attach observers. The subject class is using the Observer interface. There are PhaseObserver and StatisticObserver which implement from the Observer interface, and they override the some virtual methods from the Observer interface.

In order to introduce the decorator pattern to let players choose to decorate different observers, an interface called ObserverDecorator is created. It extends from the Observer interface. Besides the methods that inherited from the Observer class, it has an Observer object called DecoratedObserver. There are two decorator observers that can be decorated, which are VictoryCoinObserverDecorator and TokenObserverDecorator. (We didn’t add Domination observer, since the description for part 4 of the assignment 3 is confusing, and the domination observer observes the same thing as the StatisticObserver, which is implemented in the part 2 of assignment 3, and it is always available throughout the game.) PhaseObserver is working as the base Observer for the decorator pattern, which means the decorated observers can decorated themselves on top of the base observers. For instance, it can be created as TokenObserverDecorator* tokenObserverDecorator = new TokenObserverDecorator(PhaseObserver()).

In order to introduce the strategy pattern to let user to choose different types of player, we created a Strategy interface class. Then there are four child class that extends from it, which are Aggressive class, Defensive class, Moderate class, and Random class. Each of them inherits and overrides the virtual method declared inside of the Strategy interface class. These child classes will be used when players choose their playing strategy. We use the strategy as a pointer, when the player choose their play strategy at beginning, for example, when player choose Aggressive, by using Strategy *strat = new Aggressive() to load the methods in Aggressive class.

Part 4: Game Key C++ Concepts and Libraries

Concepts used	**Brief definition/description**	**Give: class/ function/ filename including extension**

pointers/smart pointers	Strategy *strat;
	Player* plys = &j;
	Observer * victoryCoinObserver;
	Observer * tokenObserverDecorator;
		● GameDrive.h and gameDrive.cpp
		● Player.h and player.cpp
		● Subject.h and subject.cpp

memory management	delete plys;
	delete strat;
	~Subject();
		● GameDrive.h and gameDrive.cpp
		● Player.h and player.cpp
		● Subject.h and subject.cpp

vectors	GameDrive::vector<Player> player Player::vector<Races> race Player::vector<PowerBudges> powerBudge MapLoader::vector<Region> regions Map::vector<list<int>> mps Subject::vector<Observer*> observers	● GameDrive.h and gameDrive.cpp ● Player.h and player.cpp ● MapLoader.h and mapLoader.cpp ● Map.h and map.cpp ● Subject.h and subject.cpp
data structure	To store the map, this game used the adjacent list to store the relation between each regions: vector<list<int>>, where int is for the region id; every list mentions the relation between the first regions with the rest of them; and the vector gathers all those relationship as a graph.	● GameDrive.h and gameDrive.cpp ● MapLoader.h and mapLoader.cpp ● Map.h and map.cpp

operator overloading	virtual void Subject::Notify();
	virtual void Subject::Notify(int numRegion, Player* p);
	virtual void Subject::Notify(Player*);
		● Subject.h and subject.cpp

file I/O	The four maps information are saved into four different map file. After the player enter the number of players, the MapLoader class will load the correspond map file to get the map information.
	void MapLoader::mapReader(string);
	void MapLoader::getFileRead(ifstream&);
	void MapLoader::getRegions(ifstream&);
	void MapLoader::getEdges(ifstream&);
		● MapLoader.h and mapLoader.cpp

exception handling	When there is a potential risk that an input from user can cause an error in the system, try and catch blocks are implemented to avoid this exception. GameDrive::start();	● GameDrive.h and gameDrive.cpp ● Player.h and player.cpp
templates	NONE	NONE
Any library including GUI	NONE	NONE
Other	NONE	NONE