During the course of software development, we often encounter issues when it comes to the key aspects of game it order to deal with these major issues within game development, we use design patterns to be able to solve many of these common issues and problems in order to increase productivity and to be able to save time and effort into the creation of software and game development.

**PROBLEM 1 : OBJECT CREATION**

One of the very common problems when faced in game development is the task of object creation, where game objects are actually created in the game when run time occurs, and though the use of a design pattern, this task can be made much easier and much simpler with the use of one. as without the use of common design patterns, object creation can be very difficult to understand and can also be very hard to manage. Two of the most common design patterns that are used are the "Factory" pattern and the "Prototype" pattern.

**FACTORY**

The "Factory" design pattern makes use of a superclass (aka, Entity/Object) that specifies all the standard and generic behaviours across all object's with the use of pure virtual placeholders and then the creation of the a specific object within the superclass is handled within subclasses(GameItem,Enemy) that only handle a specific object. when the creation of a specific object is done, it will create the object with the general behaviours from the superclass as well as special behaviour, attributes and functions that is only done by the subclass, this is though the use of inheritance, were subclasses that are lower down the inheritance tree with inherit general attributes and behaviours with classes that are higher up the inheritance tree. The main benefit of using a Factory design pattern is while most other design patterns require you to make new classes for their implementation, the Factory pattern only requires you to incorporate a new operation. This design pattern also allows us to create the object without showing the logic of how to create the object to the user.

The structure of the factory method is shown below, noticing the use of inheritance, InectionMold will have its own void function of inject() which is then overridden based of if the object that is getting molded is a duck or toy car.

And the sample code for what a factory would look like in C++ is listed below.

**PROTOTYPE**

Another popular design pattern is the "Prototype" design pattern. This design pattern is used when you want to instantiate a class by cloning the properties of an existing object within your game. when this new object is created, it will be a exact copy of the prototype object but this newly copied object permits the modification of itself without changing the behaviours, attributes or values of the original object, and because of this object being an entirely independent object when it gets cloned, if it is changed at all, it will not alter anything within the base prototype object. When a prototype is copied, it is either a shallow copy or a deep copy. if it is a shallow copy, it will duplicate all the prototypes properties and if the prototype has any references, the reference is also copied. If something is deep copied, it will clone the prototype and all its child objects and will also clone any references tied into that prototype. This will give a deep copied object its own individal copies to its references.

Below is the use of Prototyping in a UML, notice how ProductOne and ProductTwo are just copys of the prototype Product.

And the sample code of the implementation of the prototype design pattern in C++ is shown below.

**Problem 2: Inter Object communication**

Another problem that faces game creators when it comes to game development for objects to be able to communicate with other objects. This is to ensure that objects are able to interact with each other and to be able to perform certain behaviours with one another. This is important within a game as it will allow for game mechanics and animation to be made. Two common design patterns that are the "State" and the "Strategy" design patterns.

**STATE**

A common design pattern that is used for the behaviour between objects in game development is the use of a "State" design pattern. The state design pattern makes use of a state machine in a object orientated fashion. This state machine will be used to pick one state from a list of states and can only choose to run one of these states at any one time. it then uses transitions to be able to change from different states and this would be used for when a game changes from a running state to a paused state for example. Each state is considered a derived class of the state interface and these transitions are done by using method's within the patterns superclass. By using the "State" design pattern, we are able to have different instances of a program running within the same program, but running with different behaviours based on which state it is in. (a Pause state can ensure that the same Game state running, but the game just does not update like it would if it was running in the game state.) The transitions within these states are generally done using method's and these transitions can do to the next or previous state, And because of the use of the State pattern, this allows any object within the game to be able to change its behaviour independently as well based on which state.

Below is a UML of the State Design Pattern being implemented. this has the superClass State. followed by its sub-classes (StateOne, StateTwo and StateThree) with a method that allows the transition of the states with the use of the setState function.

and the Sample code for the use of a State design pattern in C++ is shown below.

**Strategy**

Another design pattern that allows for the communication between game objects is the "Strategy" Design pattern. This pattern allows for an objects behaviour to change at runtime, it will first define a family of algorithims that all of the object's will use within the game, it will the encapsulate every single one of the algorithims and then makes the all the algorithims interchangeable within the family that was previously created. This will allow an objects behavior to change dynamically via user inputs or preferences and increases flexiblity by allowing new algorithms to be easily implemented should they so wished to be implemented at a later date.

Below is a UML example of the Strategy Design pattern.

**WHEN TO USE**

You will want to use a Factory pattern when you want to make use of inheritance and polymorphism in order to be able to create new instances of Objects and when you want a centralized class that handles the creation of new objects.

You will want to use the Prototyping pattern when you would rather clone an existing base object and then modify them. which could yield better performance in the case that a Factory would prove to be more performance hitting in a game project.

You will want to use the State pattern when you want a more dynamic binding pattern and in the State pattern, we pass the context itself as a parameter to the method.

You will want to use the Stratergy pattern when you want the binding to happen only once at the start of runtime and when you want the class to be relieved from the knowledge of how to perform a certain task.

**CONCLUSION**

In conclusion, with the use of these design patterns, we are able to make better use of our time and efforts in order to create games that are not only easier to code but also easier to maintain. and the use of these common design patterns makes it alot easier to make maintainable code with the use of common functions and patterns. It is very important to be able to use these patterns when designing new games and is an important thing for programmers to know how to do. espically with these common patterns. The factory is very useful in making new game objects based upon a root object and using inheritance and polymorphism, prototyping is very helpful when you want to clone the instance of a root class when the creation of a new factory would hinder your performance and these two patterns are common in the use of creating new objects. and the State pattern and Stratergy patterns are very important for allowing objects to behave with each other. with the State pattern being used if you want more dynamic behaviours and Strategy being used if you wish for object to maintain a certain behaviours only once during runtime. All of these patterns are very important in the use of game projects and these 4 design patterns are what can be used to solve two common problems of game design, Object creation and inter-object communication.