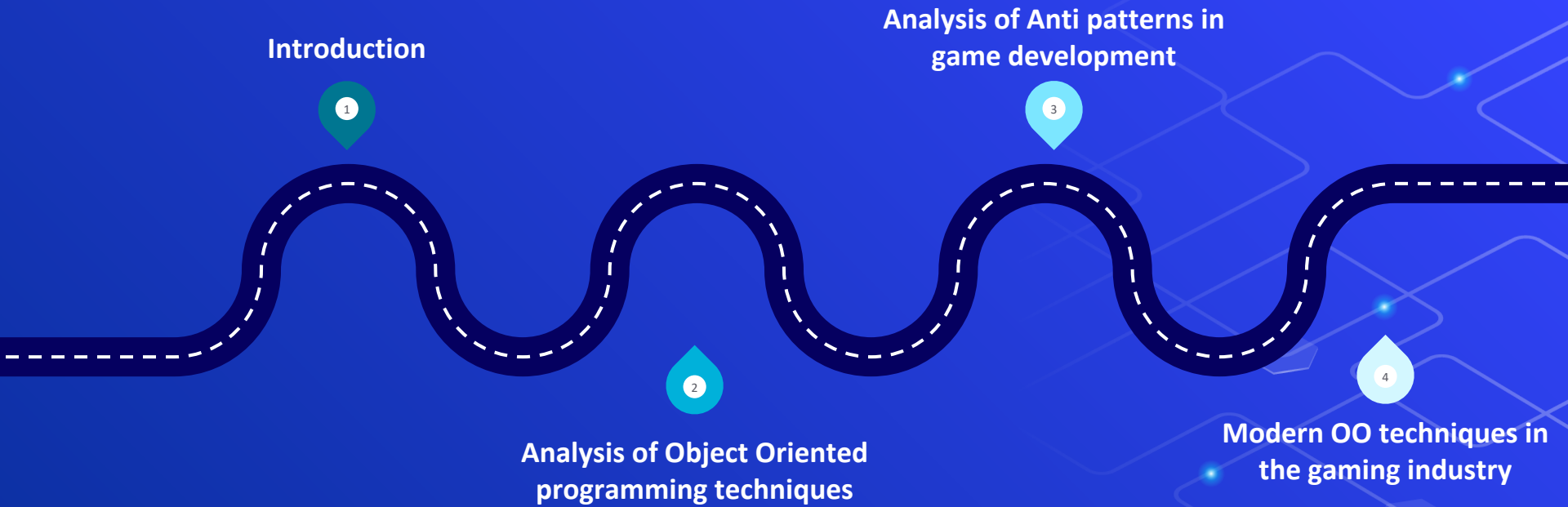
Exploring Object-Oriented programming in game development

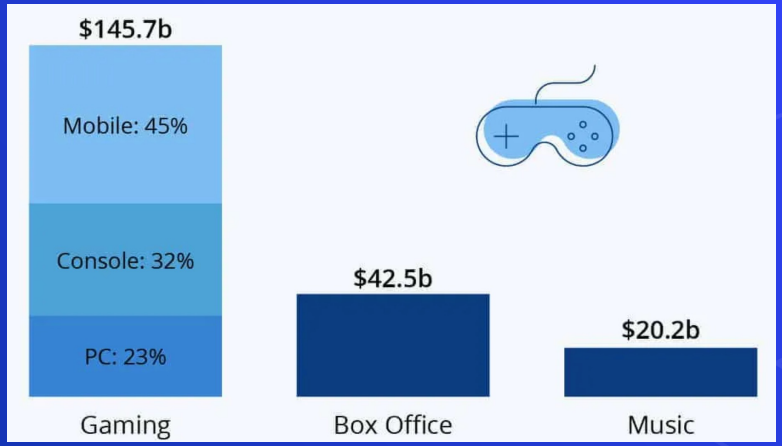
Members:

Srinivas Akhil Mallela
Likhitha Katakam
Vishal Prabhachandar

Agenda


Introduction
1
Analysis of Anti patterns in game development
3
2
Analysis of Object Oriented programming techniques
4
Modern OO techniques in the gaming industry

Current state of the gaming industry



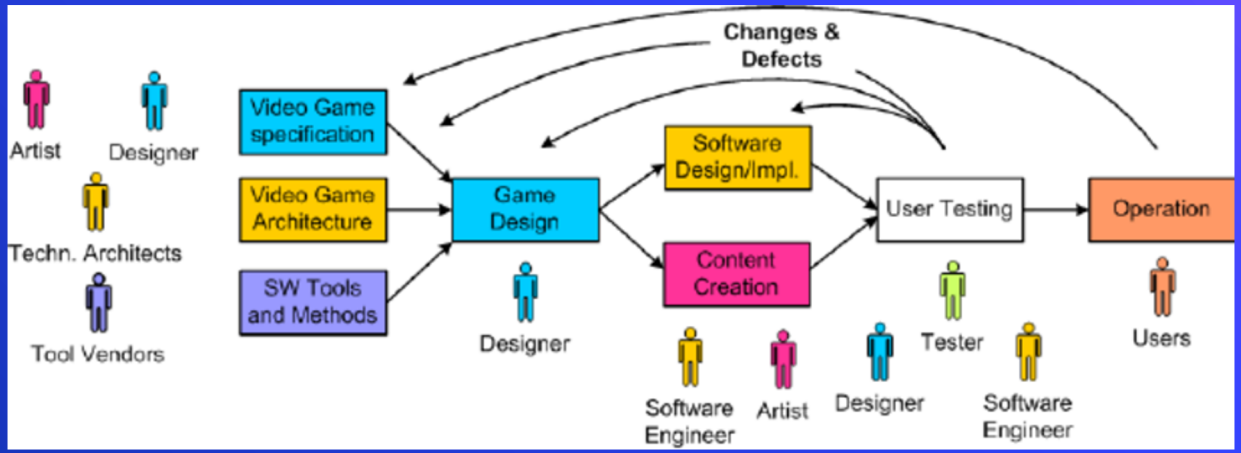
Gaming and game development has been in the forefront of many breakthroughs and advancements. Being one of the top revenue generators in the entertainment industry, they have had a big impact in the software development world as well.

Motivation and reasoning

Multi-billion dollar industry

Customer facing applications

Object oriented principles
and patterns

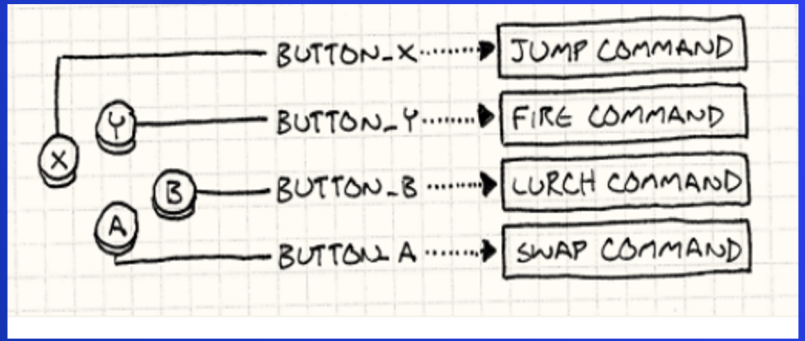


Game development combines control system, AI, sound and art, which makes them quite different from the traditional software development. However the practices followed to keep up with the tightly knit project plans are similar to software engineering techniques used commonly in the industry to maintain flexibility, increase maintainability, lower costs and improve design.

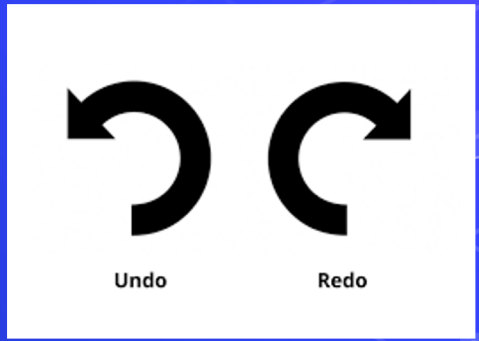
Command - Situation

Let’s consider the following two scenarios commonly found in games.

Configuring inputs

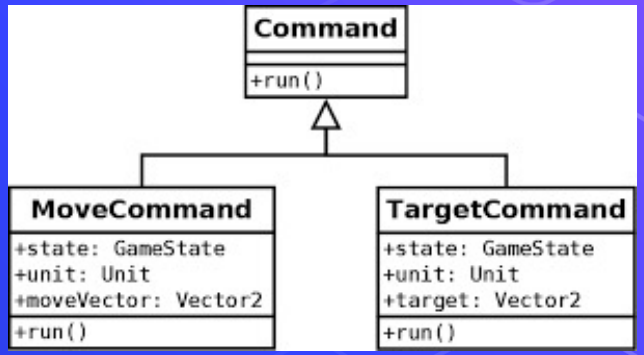


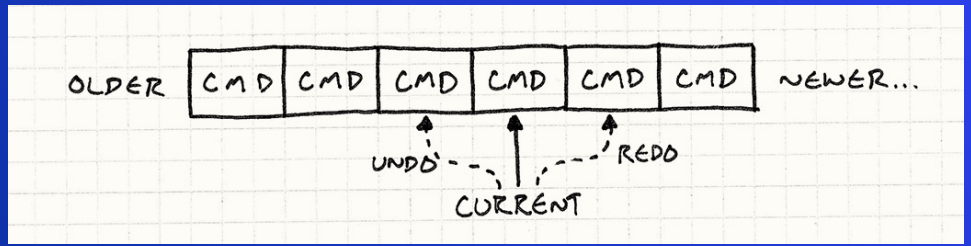
Undo and Redo



Command - Usage

We can create a base class which represents triggerable game commands, further create subclasses for each of the different game actions.



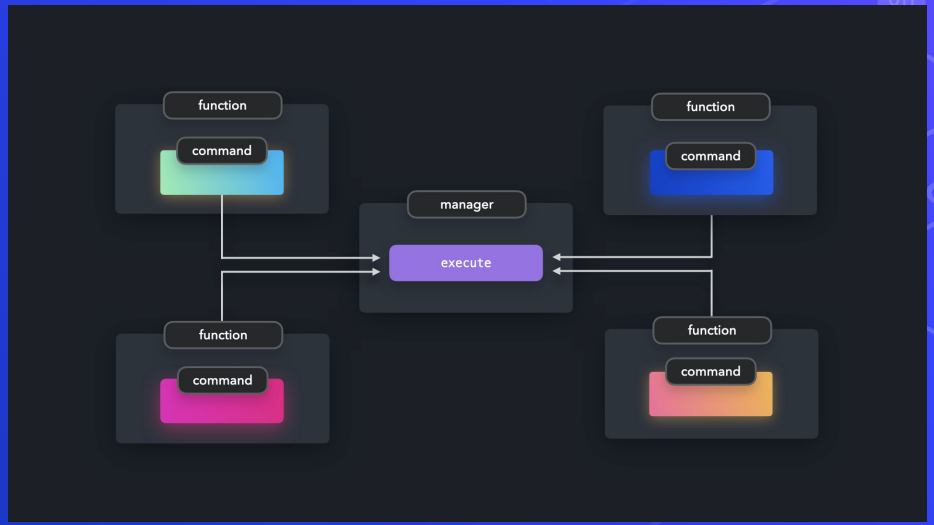


To undo, we need to remember the state and location of the last action. And subsequently, to redo, we execute the command again. We can keep a list of the actions executed in the data structure and support undo/redo through command pattern.

Command – Pitfalls

- There are a high number of classes and objects working together to achieve a goal. Application developers need to be careful developing these classes correctly.

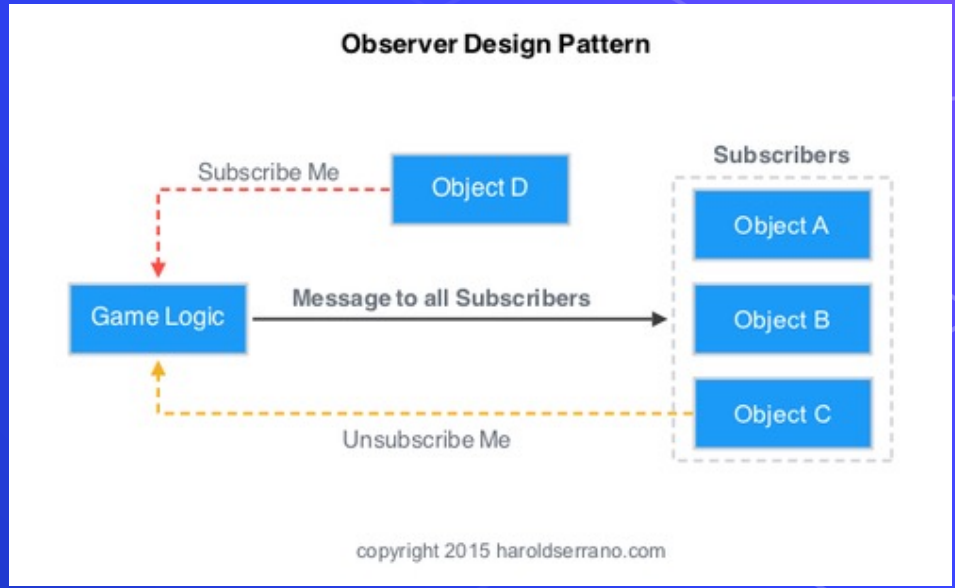
- Every individual command is a ConcreteCommand class that increases the volume of classes for implementation and maintenance



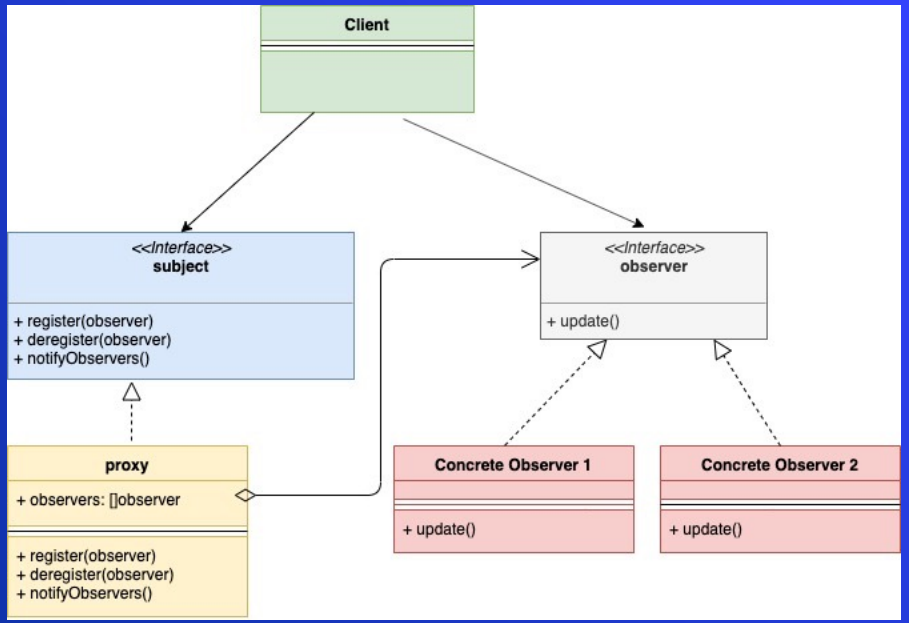
Observer - Situation

Consider a game where achievement badges are unlocked based on the different milestones completed. Coupling the achievement code into the gameplay code is not the proper way. As achievements are unlocked by different sorts of gameplay behaviors.

Observer pattern simplifies the case in that it allows objects to keep other objects informed about events occurring within a software system. It's dynamic in that an object can choose to receive or not receive notifications at run-time. Here, the publisher is called the subject and the subscribers which listen to notifications are observers.



Observer - Usage

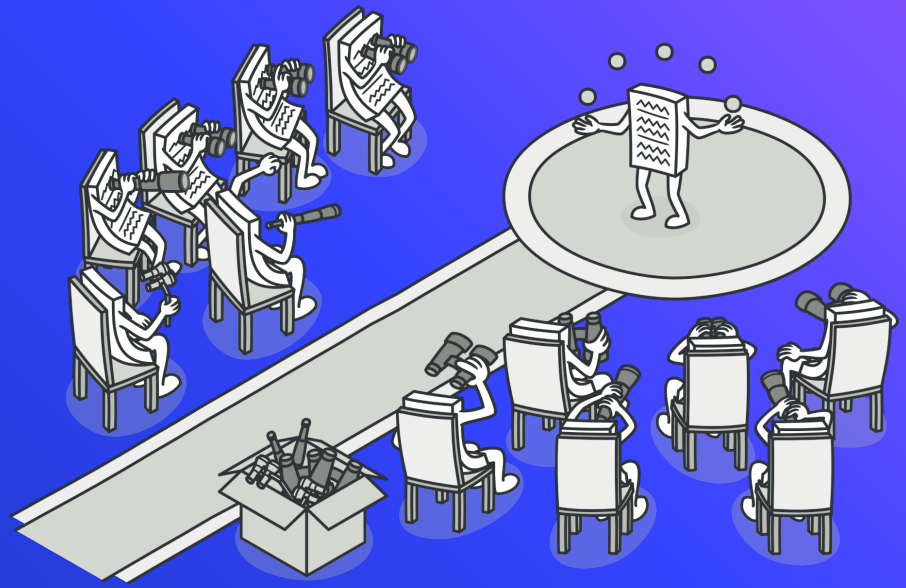


For example, the physics code(subject) has all the gaming-related aspects. The achievement system registers itself so that whenever the physics code sends a notification, the achievement system receives it. Based on the conditions, it unlocks the proper achievement with associated and it does all of this with no involvement from the physics code.

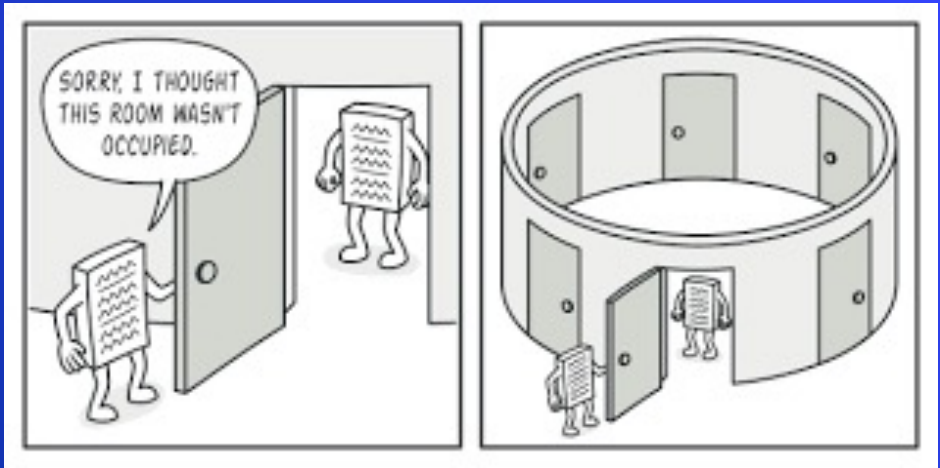
Observer - Pitfalls

Observers are slower than a statically dispatched call, but the cost is minute expect when performance is extremely critical.

Since the observer pattern is synchronous, where the subject directly invokes the observers, the subject doesn't resume its work until all observers return from their methods. This could lead to a case where a slow observer blocks the subject from continuing its work. To avoid this, the slow work has to be pushed onto another work queue or thread.

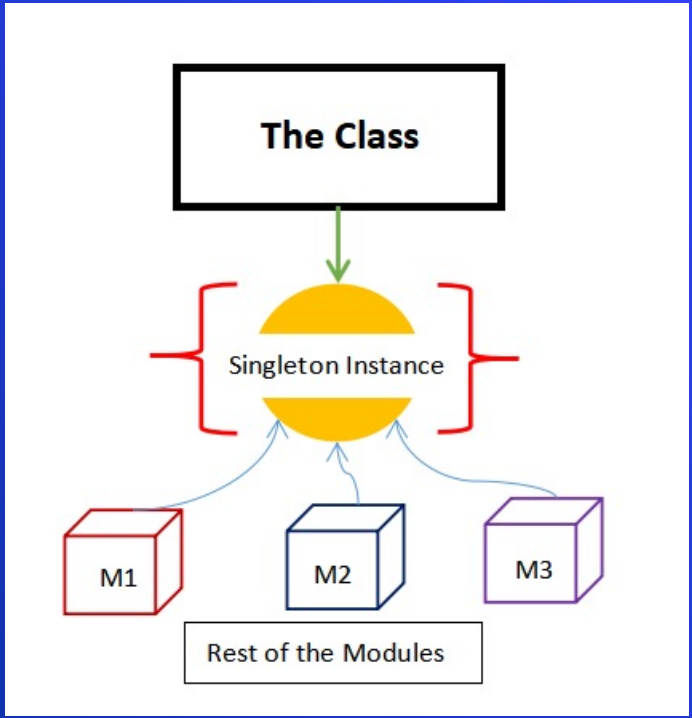


Singleton - Situation



Sometimes a class cannot perform correctly if there are multiple instances of it because we need to have idea on what previous instances are doing for the current instance to be implemented correctly. Thats were singleton comes into play it only has one instance of a class globally that we can use whenever we want to use that class. This way whenever we use an instance we can have previous information on when the instance was used.

Singleton - Usage



Creating a single instance of a class and proving a global access point to it so that any system can use the class without creating it own instance.

Some features:
It doesn't create an instance when no one uses it: this will save memory and CPU cycles.

It's initialized at run time: It's initialized only when used unlike static variables which are initialized even before calling the main function.

Sample Code

```
class FileSystem
{
public:
  static FileSystem& instance()
  {
    static FileSystem *instance = new FileSystem();
    return *instance;
  }

private:
  FileSystem() {}
```

The static *instance_* member holds an instance of the class, and the private constructor ensures that it is the *only* one.
instance() method grants access to the instance from anywhere in the codebase.

Singleton - Pitfalls

They make it harder to reason about code:

If we are trying to understand a code. If the function does not touch any global variables then we can easily get a grasp of what is going on. But lets for example imagine that there is a global variable being used. To figure out what's going on with the variable we need to hunt the entire code base just to find out one bad call that's setting a static variable to wrong value.

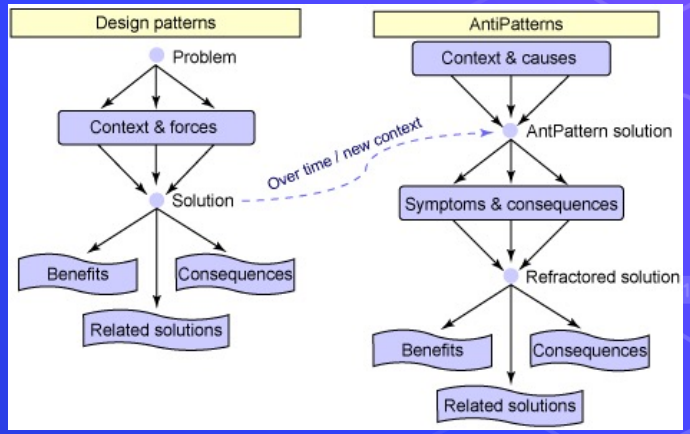
They aren't concurrency-friendly:

Games these days generally run on multi core CPUs. When we create a global variable every thread will have access to that and they can make changes to this critical section. This can lead to deadlocks, race conditions and other synchronisation bugs.

AntiPattern - Premature/Speculative Optimization

- Unnecessary optimization of non performance critical code can make the code unreadable/unmaintainable and fixing bugs in large gaming applications is never easy!

- Especially if the order of run time complexity doesn't change, micro optimizations are not of significant benefit when dealing with modern game frameworks. If a O(n) algorithm stays O(n) it's probably not worth it.




```
def checkHealth(healthBar, staminaBar):
    for i, z in zip(healthBar, staminaBar):
        p = sorted(map(i, i => min(i)), key = z: -z[0])
```

Resolution

- Attempting to optimize prior to having empirical data is likely to end up increasing code complexity and room for bugs with negligible improvements.
+ Profile at every step! If there is no discernable memory or time improvement, do not make speculative code optimizations.

```
def checkHealth(healthBar, staminaBar):
    minHealth = float('-inf')
    for health in healthBar
        minHealth = min(minHealth, health)
    minStamina = float('-inf')
    for stamina in staminaBar
        minStamina = min(minStamina, stamina)
```

AntiPattern - Once

- If a method needs to be called only once, but accidentally gets invoked again dynamically through reflection, it can cause misbehaviour
- Game engine behaviour can sometimes be unpredictable and debugging can be difficult when time is of the essence.
- Special functions have to be executed only once in the current load file or game's instance.

```
def callMeOnce():
    //Special rendering
```

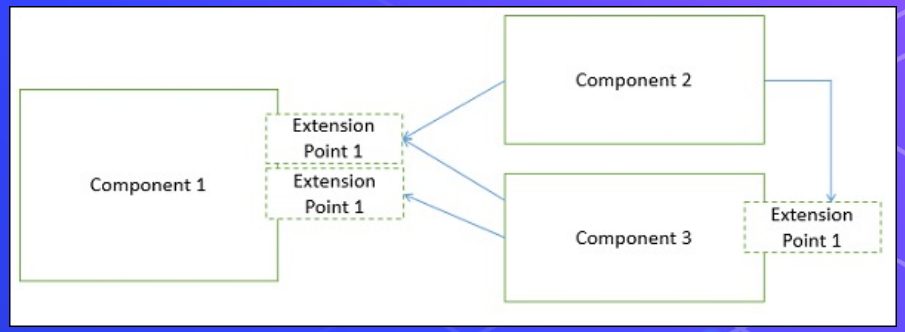
Resolution

- Use a helper function to check if the function has been accessed previously
- Implementing a standardized check for such methods will help in preventing this. Additional checks need to be implemented for multi-threading gaming apps.

```
var called;
def callMeOnce():
    if(called) return
    //Special rendering
    called = True
```

Modern object oriented techniques in the gaming industry

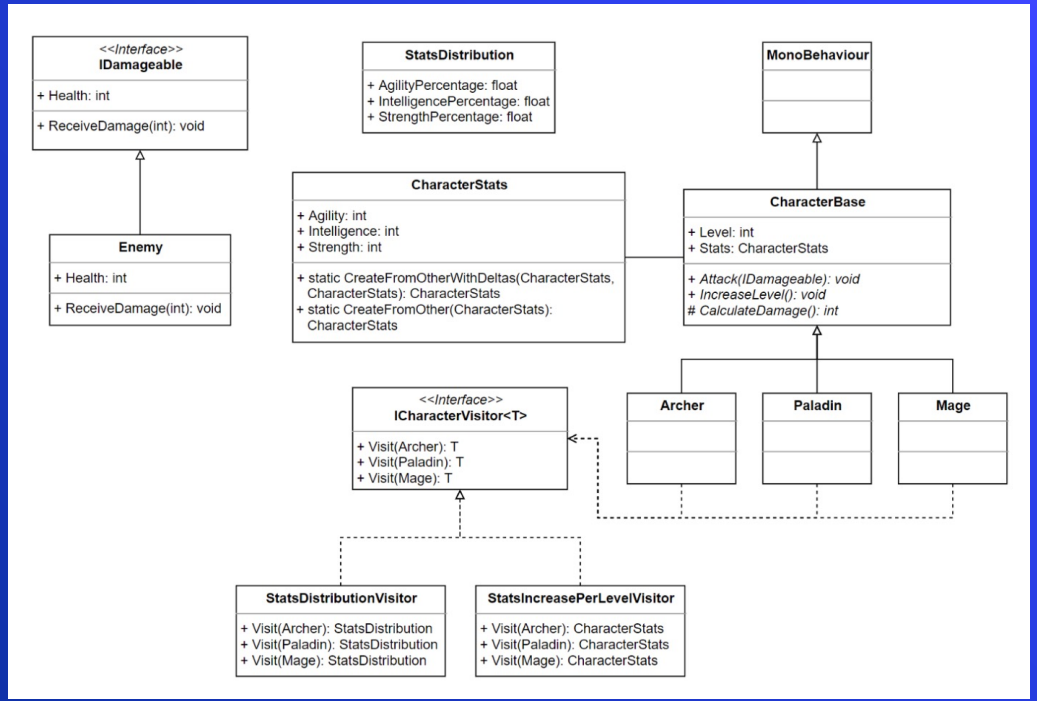
Considering the amount of effort it takes to implement game logic in game development, it is essential to find an efficient way to do that. Using software design patterns is considered good practice but the patterns have to be applied with a reason and caution. It is often tempting to over-complicate design and implementation, which, in result, puts the success of a product at risk



Unity3d coding approach follows a component-based design

Component-based architecture focuses on the decomposition of the design into individual functional or logical components that represent well-defined communication interfaces containing methods, events, and properties. It provides a higher level of abstraction and divides the problem into sub-problems, each associated with component partitions

Modern object oriented techniques in the gaming industry



It is quite challenging to follow object-oriented principles in games developed in today's day and age. Especially given the current engines and dev kits, the architecture works against you.

However, there still exists situations where certain isolated components can be tackled using common best practices.

The visitor pattern is useful in scenarios when we have to deal with multiple objects of the same structure that have to implement certain operations. Visitor lets us keep related operations together by defining them in one class. This opens additional opportunities to make use of the reach feature set of the engine when we want to influence the data of the visitor classes