Exploration of Design Patterns in Game Development

Technical Work Term Report (Work Term 3)

APSC 310

University of British Columbia

Muchen He
Associate Developer, BioWare Edmonton
Student Number: 44638154
January 6, 2019

Contents

Preface & Foreword

Purpose

The purpose of this report is to explore and observer different kinds of theoretical methods nad patterns and how they're used ot develop components that make up a game.

There exists a discrepancy of knowledge acquired in school versus ones learned in the company. Mainly, because of my areas of studies is in electrical engineering, discussions involving software development, software structure, and methodologies are minimal, if not non-existent. Therefore this exploration and report is an excellent opportunity to broaden my perspective in software development.

As well to consolidate my knowledge as a whole in the industry as design patterns are software development concepts and potentially applicable elsewhere in the tech industry.

Background

This report is produced during the second and third co-op term while working for Electronic Arts (EA)'s division in Edmonton. The studio name is BioWare which produces games with rich single player stories. The technology worked with is shared across many EA studios for various franchises and licenses. My position at Bioware is an *associate developer (programmer)* for the game Anthem, set to release in late Feburary of 2019. Modern EA games are built on the game engine called Frostbite. My contribution is mainly to the game client's code as well as the engine codebase.

Scope of Coverage

The report will cover most of the original design patterns as defined by the book "Gang of Four".[1] Many of the patterns will not be discussed in too much detail as they might not apply. Technical content of the report, including code, diagrams, and models will be general and high level. Any detailed breakdown, explanation, and specification of a existing idea or technology in this report is not confidential or proprietary. This report will not cover any confidential or proprietary data such as code used in the game Anthem or Frostbite as it is a property of EA or BioWare.

The technical coverage cited in this report is limited because the co-op work term position is working in UI/UX. Thus I lack understanding in other areas of the game such as character handling, world rendering, etc. Many of the workflows I describe in this report may only apply to UI/UX development within this project. BioWare, the studio for developming games focuses more on stories and game content rather than research of new and groundbreaking design patterns for games, thus the software methods, patterns, and provided sample code are industry generic and does not expose any confidential information.

As the scope of game development or software development extends wide in breadth, this report will not thoroughly cover topics of object oriented programming, development methodologies, specific syntax in a programming language, specific implementations of algorithms in a programming language, and psychology of UI/UX.

Contributors

Other contributors to this report is as follows:

- **Gibson, Tim** (Senior Software Lead, manager) contribution by supervising my workfly and giving advice on how design patterns are applied in BioWare games, EA Frostbite, and how they can be used outside of games.

- **Johnson, Chris** (Software Engineer, supervisor, manager) contribution by helping me working through Anthem UI/UX projects as Chris is expert in UI programming. Chris Johnson also gave how some of the design patterns work in UI in context. Finally, Chris Johnson oversees my work term and report and provided valuable feedback.

Summary

The software design patterns are templated solutions to be implemented in large software projects. These patterns improve performance and maintainability of the end product by pushing computation and spacial optimizations to object-oriented-programming structures. The design patterns make the components of the software less error prone by making them more flexible and modular through acts of decoupling and mediating. These patterns also improve productivity and organization of programmers as monolithic code are broken down and decoupled into independent modules by function and composition. Video games, or video game engines are large pieces of software that require maximum optimizations. Therefore, the software design patterns is essential in programming of video game engine. However, the design patterns are to be used to solve existing problems by identifying specific symptons and applying the appropriate pattern. Usage in attempt to resolve small to non-existant issues is unproductive due to the relatively large ratio between time and effort required to implement versus benefits.

List of Figures

List of Tables

1 Introduction

This enters the discussion portion of the report. We will analyze the current processes in game development for UI/UX at BioWare. Then look at some of the more theoretical, template solutions known as design patterns in later sections and draw comparisons. We will also look into how the design patterns would be used in a general sense in game development (not specific to BioWare or BioWare's current project, Anthem).

1.1 Game Development Overview

A major portion of the video game market consists of AAA video games. These are titles with very high production value, large budget, and typically consists of a team of hundreds of developers. These developers consist of artists, scripters, programmers, managers, etc. Many of these roles are further grouped into sub-teams such as rendering, physics/simulation, AI, UI/UX, sound design, music production, networking, etc. Individual developers are specialized to work on one part of the game, with the exception of leads, managers, and other executives.

1.1.1 Game Engine

Developers will use a game engine so that many people can work on the project at once. The game engine is generally highly optimized for graphics rendering, physics simulation (such as collision detection), and object handling.

The Frostbite engine is the game engine widely adopted at Electronic Arts (EA) studios. It was originally developed by Digital Illusions CE (DICE).[8][9] As expected, the game engine is a piece of software, thus it is prone to software development problems that design patterns are ought to solve.

1.2 Game Building

Programmers make primitive entities such as an abstract UI widget, as well as the system that manages these entities.

The scripters and artists then use these entities in *schematics* or *blueprints*, which are used for visual scripting blocks of game content that contains logic and input/output together. Figure 1 shows an example of what it looks like.

These schematics are in game levels, UI widgets, and prefabricated logic blocks (LogicPrefab). Thousands of these make up functionalities in a game, and are all handled by the game engine.

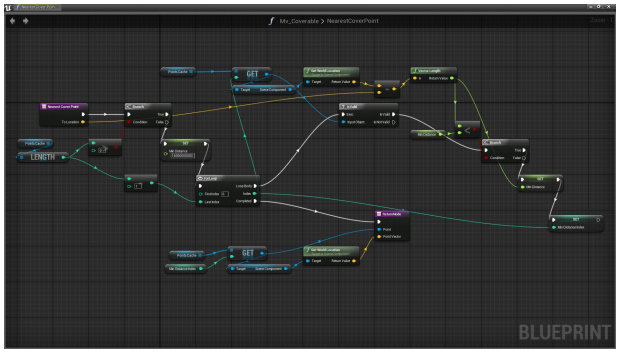


Figure 1: Blueprint (schematics) for development in Unreal Engine[2]

1.2.1 Role of Programmers

At BioWare, most game programmers work with code (in C++ and C#). Programmers are specialized to develop in-game entities, or tools for scripters, designers, and artists. Programmers also integrate systems together, such as making mouse inputs interact with hitzones widgets, which interacts with graphical widgets or animation timelines.[i]

1.3 Object Oriented Programming

Object oriented programming (OOP) is a programming paradigm centered around interaction of encapsulated objects. Objects have its own functions (methods), input and output ((getters) and (setters)), state and memory (member variables). OOP emphasizes on encapsulation which makes software structure easier to organize and debug. OOP allows programmers to define well-defined boundaries between features or objects. OOP also enables *polymorphism* and *inheritance* which are important in abstraction and organization of code.[10]

Although we will not go into depth into the subject of object oriented programming, it is vital to understand that OOP is basis of most modern game development due to its benefits of reusability, refactoring, extensibility/scalability, maintenance, and efficiency.[10] Nearly all of the design patterns explored here are utilizing OOP concepts in some way.

[i] Animation timelines are timeline that describe how a particular object will animate. Example: a fade in.

1.4 Design Pattern

Design patterns are reusable, general, abstract solutions to common problems in software development.[11] Design patterns are meant to be used when there no problems actually exist – doing so would cause excess undesired complexity.[1]

Throughout the exploration in this report, we will see that all the patterns loosely fit into three categories of benefits: performance optimization, readability/maintainability improvement, organization/structure/scalability improvement. We use these three categories to evaluate the usefulness of the design patterns in game development.

1.4.1 Antipatterns

Software written without care are difficult to maintain, prone to bugs and errors, and hard to collaborate. Software written with these "hacks" are known as *antipatterns*, where there is more negative consequences than the benefits of its solution. Antipatterns are counter parts to correctly implemented design patterns.[12]

Avoid antipatterns as much as possible. Exploring the negative effects of antipatterns is not part of the scope of this report.

1.4.2 Gang of Four

In software development, the phrase "Gang of Four" refers to the four authors of the Design Patterns book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides[13].[1] The book features the most common design patterns widely in use today.

The design patterns featured in the book outlines the classical design patterns. Thus, many patterns would not be applicable as more recent revisions of programming languages (Python, C++11, etc.). This is because design patterns are inherently solutions / workarounds to limitations of the programming language. As there are more improvement in the usability of the language, certain design patterns (such as the *template* pattern) are implemented less as it's less complex, less error prone, and easier to use the existing features built-in to a language.

2 Creational Design Patterns

Creational design patterns involves with object creation.[14] This is especially useful in game development, such as when we need to create many trees or bushes in a level of a game. These patterns become more useful when the objects we're trying to create have relationships, such as *inheritance*, to other objects.

2.1 Builder

In object oriented programming (OOP), the construction of a class instance takes parameters. It is possible that we need a lot of parameters to satisfy more variants of the class. This introduces the telescoping constructor antipattern,[15] where there are numerous variations of constructor methods all delegate to the default constructor. This chains the different constructors together like an upside-down cake, or a telescope shape.

The code is hard to maintain and difficult to use. When creating a new instance (invoking the constructor), it's hard to tell what the actual parameters are if they have the same type. Using default parameters is useful but makes code less usable as it changes the parameter ordering.

The **builder pattern** encapsulates the parameters into a single structure. The structure itself is then passed into the constructor method, where we can access the data inside the structure again. The members of the structure can be accessed and modified by name, thereby eliminating the ambiguity of telescoping constructor.

This is useful in games. For example, a customizable character has many attributes that can be filled in (Figure 2). Realize that as we increase the number of attributes, we also need more parameters in the constructor to populate the attributes.

PlayerCharacter
-name: string -age: int -height: float -weight: float
+PlayerCharacter(string name, int age, float height, float weight)

Figure 2: PlayerCharacter class with various desired parameters

Using the builder pattern, we create a encapsulated structure that would contain all the parameters in figure 3 and then default all of the struct internal variables to some default value. We then have the flexibility to fill in whichever attribute we want to change by modifying the struct's members (`struct PlayerCharacterBuilder builder; builder.age = 25;`). Optional logic could be added to the builder as well.

struct PlayerCharacterBuilder
name: string = "Alex" age: int = 20 height: float = 175.0 weight: float = 180.0

Figure 3: PlayerCharacterBuilder struct

Now, the PlayerCharacter class constructor only uses a single parameter of the builder type reference (figure 4). The constructor then reads all the values in the struct, and assign them to the corresponding member variables.

PlayerCharacter
-name: string -age: int -height: float -weight: float
+PlayerCharacter(struct PlayerCharacterBuilder &builder)

Figure 4: PlayerCharacter class with various desired parameters

Of course, this pattern is applicable widely in game development, from in game objects that have visual variations, to online player save files, and even the build pipelines used during development to build game assets.

A sample code for this pattern is available in section A.3.

2.2 Factories

Factory is a quintessential creational design pattern because it governs the creation of different types of objects at a scalable level. For programming games, memory allocation and management is tricky, therefore it is helpful to not expose that kind of control to the client code.

The three main factories patterns we will explore

is *Simple Factory*, *Abstract Factory*, and *Factory Method*. However, because factory method pattern is very similar to abstract factory[16] in the context we care about, we will only explore abstract factory. The general pattern in all these different factory patterns is that the keyword `new` in C++ is harmful. And that the factory should abstract away the specific steps to instantiation.

Simple Factory
As the name suggests, the *simple factory* is as simple as a factory could get, the simple factory generates an instance for the client but does not expose any instantiation logic.[17]

A real life analogy would be purchasing a burger: the client requests one burger, and the restaurant makes the burger and give it to the client. Without the factory design pattern, the client would need to gather all the ingredients of the burger and make it themselves, which is repetitive and unproductive.

Particle systems could be examples of simple factories used in games. Suppose we have a random sparks generator and each particle is an instance of the *Particle* class which is managed by a *ParticleSystem* parent class. The parent, in this case, act as a factory. The client simply tells the ParticleSystem when to start or stop emitting particles, the ParticleSystem then is responsible for the initialization logic of creating the particles such as setting random displacement and velocity, and random lifetimes.

A sample code for this pattern is available in section A.8.

Abstract Factory
Abstract factories are useful in cases where we need to instantiate abstract objects, but we do not know the specific concrete object yet.[18] This pattern is prevalent in cross-platform UI programming where the identical elements have similar characteristics, but behave slightly differently under the hood. A typical abstract factory resembles a structure depicted in figure 5.[3]

For instance, suppose we have a game that runs on Xbox, PlayStation, and PC, and in it we have a UI widget to enter the player name. The expected behavior would be to open the first-party on screen keyboard for the two consoles, and do nothing for PC (as keyboard is physical). In this case, to make the code more maintainable, we would not program it with two or more variants just to work with different controls.

A similar example is provided in the sample code in section A.1.[19]

Of course, the abstract factory could be generalized to various mechanisms within a game from game objects to player characters.

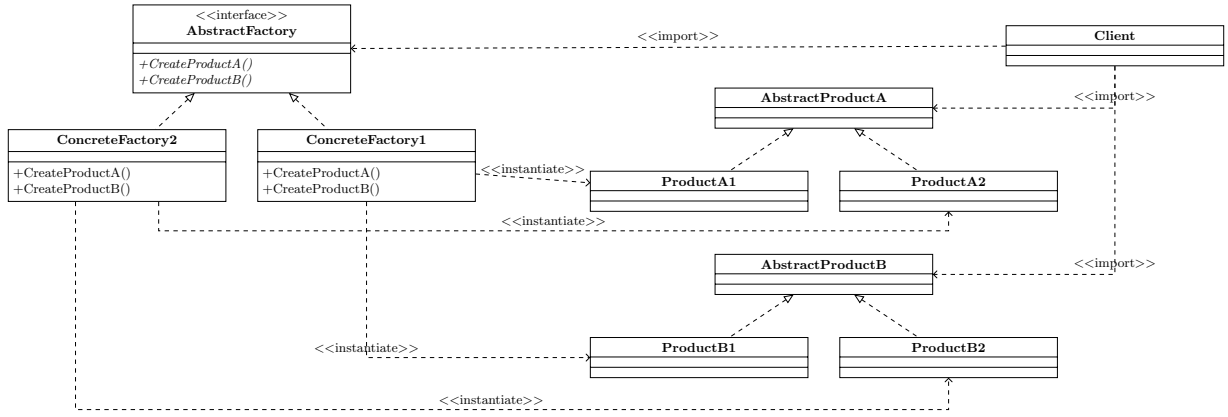


Figure 5: A generic UML diagram of abstract factory pattern (source: pfg-umlcd documentation[3])

2.3 Object Pool

The object pool is a creational design pattern that caches objects for re-use. An object pool can let clients check-out objects from the pool. If a requested object do not exist, the pool can create a new object; although expensive, it only occurs once.

An analogy in real life would be a library of books. When a client wants to access some information, instead of writing a new book to the client to use every time, which is expensive, we check if the book already exists in the library. Then we could lend the book to the client. When the client is finished, they return the book for other clients to checkout.

The object pool better organizes object life time of the objects in the pool. Thus we can free memory by clean up unused objects periodically (often known as *garbage collection*).

The object pool are applied universally in game engines to relieve required memory resources as they are important in larger games to manage tens of thousands of objects. In Anthem as seen in figure 6, the object pool can be used on the on-screen markers (on the top of the image). These heads-up-display (HUD) markers shows the player where they need to go, or where other players are. These markers are persistent in the 3D world but not all of them needs to be displayed at all times. It would be a waste of resources to draw the markers even if they can not be seen. Only the ones inside players' camera frustum are rendered. But that does not mean we should destroy other markers as it would be more expensive to create them again, and could cause performance issues if the player looks around too fast (spinning). So we return the instance of the marker when they're no longer being rendered to the object pool. They will be accessible and can be rendered again when players' camera puts these markers into view.

Figure 6: HUD UI markers from Anthem E3 Gameplay Demo[4]

One possible general use case is manage objects' lifetime based on culling[ii] objects from rendering. Obviously since the player cannot see them when it's outside the view of the screen, we do not want to render them (shading, particles, textures) in order to improve performance. But re-creating the instances of world objects when player puts them into view again could be expensive, and cause stutter and lag when player looks around too fast. An object pool is applicable such that the objects that need to be rendered could be checked out, and returned when they no longer need to be rendered. See figure 7 as an example for it used in Horizon Zero Dawn.[20]

[ii] culling is hiding objects from rendering to improve performance

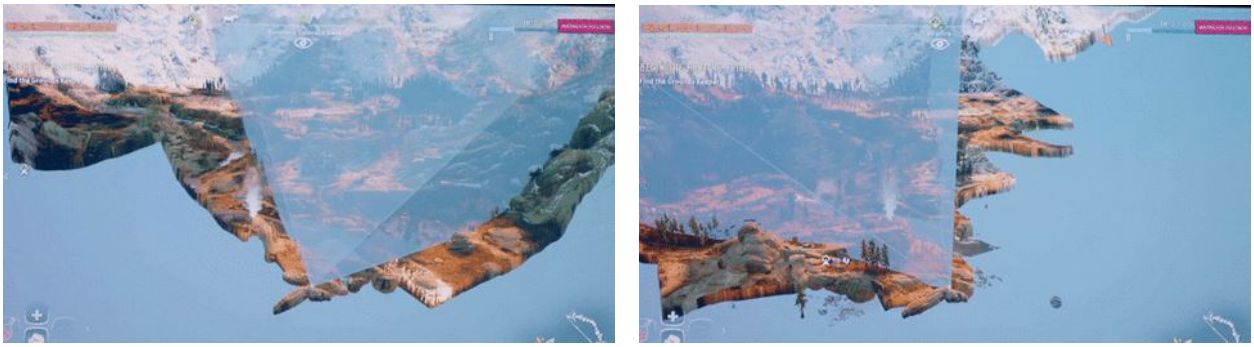


Figure 7: Culling of the world outside of player's field of view in the game Horizon Zero Dawn

2.4 Prototype

The *prototype* design pattern helps when we need to create an object that is similar to an existing one, but instantiating a new instance is relatively expensive.

For instance, suppose we have a procedurally generated 3D model of a rock and we want to clone this model all over the place N times.

We could just generate N rocks like how we generated the first one using the same random seed.[iii] The problem is for every clone of the rock, there are computational overheads that hinders efficiency: such as re-computing the normal vectors from all the faces of the polygon, or recalculating the UV coordinates for the texture.

Using the prototype design pattern, we implement a clone method to the rock object that copies all its data to a new instance, thereby avoiding the re-computation overheads.

2.5 Singleton

The singleton pattern involves creating a single instance of a class and make sure that only one is created and used.[21] The singleton pattern provides a global accessor to the client such that it can be accessed everywhere. Thus, it is said singletons are glorified global variables.[22, 23]

In game development, singletons can be used for for system managers. i.e. a single manager class that overlooks and orchestrates UI system or a system to track achievements.
Singletons are also useful for configuration classes and shared resource accessing classes.[22] An example in game development is levels and personalization libraries (customization) where we load them into memory during loading screen initially. After-which their instance can be accessed via a static function call.

Typical singleton class implementation holds a static reference to itself and a static getter that returns that reference (figure 8). In singleton implementations, concept of *Lazy implementation*, where we do not initialize and allocate the memory for the object until client actually requests it.

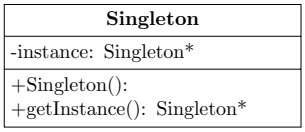


Figure 8: Generic singleton UML class diagram

An example of Singleton implementation for an simple achievement tracker in sample code in section A.9.

[iii] Procedurally generation relies on pseudo-random or noise, which are based off of a seed.

3 Behavior Design Patterns

The behavior design patterns recognizes the problem in common software developments and provided more flexible communication between classes and class hiearchies. The overall benefits are more robust and flexible software.

3.1 Command

The command design pattern decouples action request and action handlers. The callee who makes the request makes no assumptions about the class the receives the command. The decoupling between the two classes means that the client does not know anything about the handler. Likewise, the handler does not know anything about the callee of the command.

A loosely related real life analogy would be managers and programmers. Suppose programmer are specialized and can code a specific kind of software, and the managers tell programmers what to make. In a tightly coupled, without utilizing the command pattern, the manager is specialized too. Different programmer would need a different manager and the specific "manager-to-programmer" mapping cannot get broken (i.e. it does not make sense to have a manager responsible for online-services to give tasks to a graphics programmer).

Using the command pattern, the manager makes no assumption about who will be assigned the task/command they request. The manager simply broadcasts its requests out and moves on. Similarly, the programmer (handler) makes no assumption on who issued the task/command. But if the programmer is responsible for that type of task/command, they will perform the corresponding action.

In video games, the command pattern is most applicable in input mapping. An input is any input player gives to the game, such as controller axis and buttons, mouse presses, mouse move, and keyboard key pressed. In a tightly coupled scenario, the video game features a single control scheme and the player cannot customize it. Many older video games such as Tetris[24] and Super Mario Bros[25] feature relatively simple controls. In addition to the limitation of the hardware which hinders the flexibility of the software, there is no customizable input mapping.

In nearly all of the newer game titles, regardless of they are from AAA production or independent developers, remappable inputs are almost an industry standard. Quality assurance (QA) of game studios rigorously perform compliance testing to ensure that remappable inputs function for customization as well as accessbility to increase player satisfaction.

In the example shown in figure 9,[26] the game Counter-Strike: Global Offensive (CSGO) by Valve demonstrates remappable inputs which is a convention adopted by the video game industry.

Figure 9: Option menus for optional input remapping in the CSGO

On the left side, we have the description of what handler a particular input would be attached to (i.e. "Fire" action is bounded to the controller's *right trigger* button). By selecting this menu item and apply a new input, we could remap the action *fire* to something else. This is extremely useful if the player does not own a controller, has disabilities that prevent them from reaching certain inputs, or just prefers their own input scheme. Note that CSGO is not special and is presented for the sake of the example. Modern games (especially ones that shipped to PC) will most certainly feature remappable inputs.

The command design pattern opens up more flexibility in mapping a requests to a handler. The software programming as well as game design is opened to be flexible. The callee invokes a command object that holds a reference to the *real object* so that the handler can be called. Note that the command object is a *mediator* between the callee and the handler (see section 3.3). Thus, we are not limited to simply remapping inputs, other logic could be implemented in the layer between the callee and the handler. One instance is to also incorporate *memento* design pattern and implement a simple undo/redo system.

3.2 Chain of Responsibility

In extension to *commands* pattern, chain of responsiblity essentially maps a single command to multiple handlers in a chain. When a request is raised, the first handler in the chain will receive and the logic in the first handler can decide if it wants to "lower" or "resolve" the request. If resolves, the request will not be passed down the chain. Otherwise, the next handler will be called and the same request is passed.

A common real life example is the algorithm of giving someone money in exact amount, in cash. Because cash have some specific denominations, we build this chain of command where we first use the largest denomination and then move down the denominations. For example, if I need to pay $255.60, the logic would look like this in figure 10:

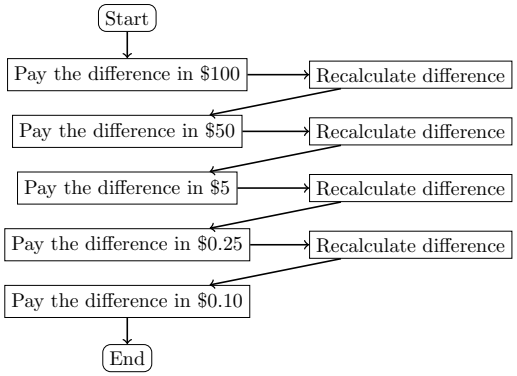


Figure 10: Chain of Responsibility design pattern demonstrated in process of an real life scenario.

In video games, chain of responsiblity, in combination of the *command* design pattern (section 3.1) are often used handle large scale UI. I will use the term **chain of commands** for the combined use of the two patterns.

In the Frostbite engine (and other game engines) there exists elements *InputHandler* and *InputDispatcher*. The input handler elemet utilizes the command pattern and on a high level, listens to a particular user input (mouse, keyboard, or controller) and outputs an event when the input is active. The input dispatcher element may hold references to other input handlers via links. This creates a chain of command as when an input is fired, the event travels up the chain of input handlers. Each input handler has some logic and could be customized to "consume" the input. Only the input handler that "consumes" the input will carry out the logic, and other input handlers down the chain will not receive the command.

The input dispatchers enable/disable input handlers on the chain. This is useful for when we only need certain inputs at certain time. For example, when in game, the mouse cursor is disabled as the player is using the mouse to control the camera, and the input for mouse buttons should correspond to aim and fire of the gun. However, when we activate menus, we show the cursor and disable the input to areas we just discussed. Instead turning on other UI related input handling such as mouse hover etc.

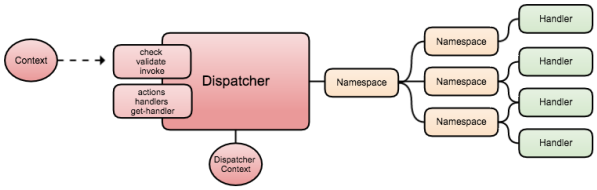


Figure 11: General model of input tree with input handlers and input dispatchers

During handling of mouse press when there are multiple *hitzones*[iv] overlapped (figure 12). Any press or hovering event will be handled by the top-most hitzone entity first. If the handler does not consume the input event, the event will naturally "flow" down to the next level.

Figure 12: Mouse interacting with overlapping mouse hitzones

3.3 Mediator

Mediator acts as a middle-man between communicating objects and decouples them.[27]

[iv]Hitzones are interactive areas for mouse that reports data and events such as mouse position, if mouse is hover, mouse button presses and releases

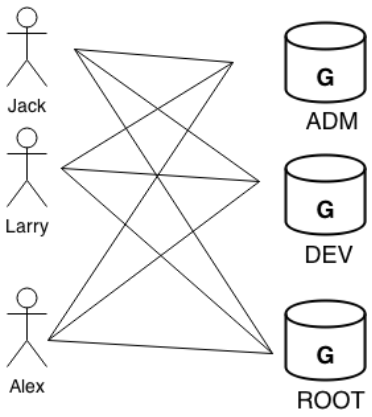


Figure 13: Objects communicating without using mediator pattern

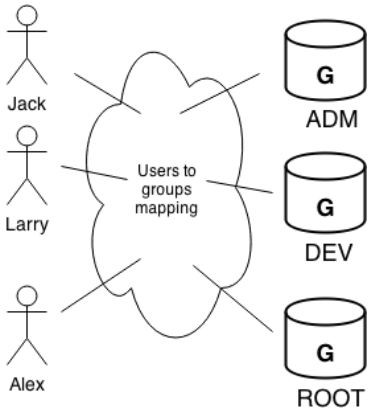


Figure 14: Objects communicating using mediator pattern

As seen in figure 13, complexity of interactions between objects become exponentially large as the software grows.

Such problem occurs often in UI: An example would be when a input signal is mapped to multiple elements. In particular, say we have a button to open up the game menu. Then at the same time there are a few systems we need to talk to during this. First, we need to tell the game to lock all inputs to the character in the world, so that when we're navigating the menu, we don't move the character around. We want to enable mouse cursor (and hide it during game play). We might want to pause the game so that enemies don't attack in the background while we have the menu open. We might want to optimize performance by turning off rendering of the world when we're in the menu, since we might not able to see it anyway.

Without using a mediator, the activated event from the button would need to invoke methods in all these different systems. This *antipattern* breaks abstraction, tightly decouples all these systems together which makes it prone to error and crashes, and makes the code highly unmaintainable.

Instead, we use pattern as depicted in figure 14. The button event should communicate with a middle-man that sets a state. The various systems that would be affected does not know anything about the button, or anything that interacts with the mediator. It merely gets the state stored in the mediator, and performs corresponding action.

3.4 Memento

Memento are design patterns for objects to know how to save and load (externalize) its own internal state and data.

Usability improvement: when used together with Command pattern, we can create undo/redo actions. Not as applicable in the game engine itself, but essential to the development tools and editors artists and scripters use to develop the game.

Difference Between Command and Memento

Command passes a request from the client to be handled while memento passes its own internal state. Using an analogy of how a player interacts in a game menu: a command would be the event of player clicking on the save/load buttons of their game files, and the memento would be the save file objects itself being saved/loaded.

3.5 Observer

The observer design pattern is ubiquitous to software development. Observers define a one-to-many dependency where many clients can "subscribe" to a state change of a particular object. The clients would be notified by the broadcast sent by the subscribed object.

In observer pattern, the object sending the events is the *subject*, and the objects receiving are the *observer*. The observer hierarchy consists of an abstract

observer base class, which has some handler (see figure 15).

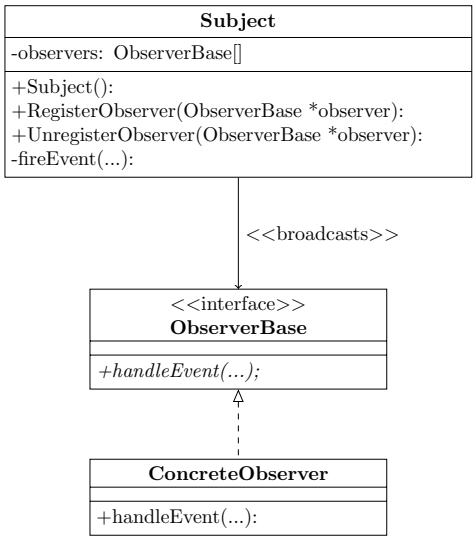


Figure 15: Simple subject-object relationship in observer design pattern

Without using any mediators, the subject is *loosely coupled*[v] to the observer types. The subject holds a collection or list of registered observers, such that when an event should be fired, the subject iterates through all the observers and deliver the event.

In Frostbite (and extends to many game engines and software), many OOP and inter-class relationship relies on events. A player's interaction with the controller, keyboard, or mouse triggers input events to be handled by the Input Handler. For example shown in figure 16, the subject are objects that outputs an event whenever certain key is pressed (red blocks). The output event *Released* is hooked up to some observer. The corresponding observers are the *Move Updated Component* handlers that carries out specific tasks whenever it receives the event/signal.

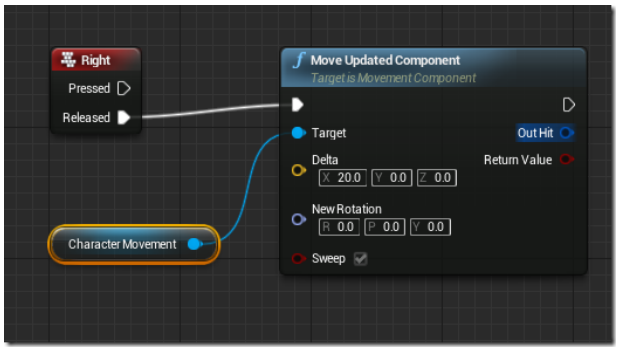


Figure 16: Example of event connections in Unreal Engine which is based on the observer design pattern.[5]

It avoids the antipattern of *spaghetti code* where increase in complexity of the software leads to exponential increase in complexity and unmaintainable code. The design pattern is very useful for organization, maintainability, flexibility, and scalability of code.

There are several approach to the implementation. Mainly, the subject could push data to observers during the function call. Or the observers could pull data from the subject. The observers could receive a subject pointer during construction and register themselves, or a client could register the observers for them. Nonetheless, the observer pattern is flexible and fit many needs in video game programming.

A sample code of an observer pattern example is in section A.7. In this example, we created a simple system to broadcast world events to a multiplayer game.

[v]Loose coupling means dependency to another object via interface or abstract class. Whereas tight coupling means dependency to the concrete class directly.

4 Structural Design Patterns

Structural design patterns are generally utilized to help organizing the structure and relationship of objects.

4.1 Adapter

An adapter is a structural design pattern that wraps an incompatible object such that it can be used/interfaced by the client.[28] Such pattern is commonly used to integrate third party *application program interface* (API) or libraries into the project. In other words, an adapter will take something that has already been designed or built that would not work with current solution, "retrofit" around that and make unrelated classes work together.[28]

A middle layer that transforms the interface of a desired object (usually provided by libraries) to work with client.

An example used in game right now is adapting data from local client to an online database. Suppose we have some player persistence data for an online game stored on a database in a form of a JSON object. When we fetch the player data from the server to be used in gameplay, the JSON object would need to be "parsed" to some usable object in C++. Thus an adapter would be used to read the JSON and populate another object with one-to-one mapping.

When we change the player data locally, we then need another adapter to serialize the data in the client player object to JSON such that it can be posted to the database again.

Another example would be adapting file formats such as 3D model objects and texture formats such as textures.

In appendix section A.2, I demonstrate one of the simplest form of an adapter. Suppose we utilize a graphics library that has a rectangle drawing function that takes the parameter of `(int x, int y, int w, int h)` where they represent x-coordinate, y-coordinate, width, and height respectively. But our client code, instead of knowing the width and height of the rectangle, knows the x and y-coordinate of the opposite corner. Of course, the client could do the conversion itself, but the implementation could get more complex and repetitive as we deal with more complex interfaces.

In essence, the adapter design pattern is beneficial in game development to fit incompatible parts to the project. Thereby reduces development time that it would take for re-implementation.

Difference Between Adapter and Mediator

Note that adapters are different from *mediators* (section 3.3), as mediator manages the communication between two classes, essentially decouples the interaction of the two. Whereas adapter simply translates.

Consider a real life analogy, where two people who speak different languages are fighting. A mediator would be a lawyer such that the two people don't talk to each other directly. An adapter would be a translator so that the two can communicate more directly. Thus, adapters are structural and mediators are behavioral.

4.2 Bridge

The *bridge* pattern decouples abstraction from its implementation which enables orthogonal hierarchies. One could think of orthogonal hierarchies like orthogonal basis vectors from linear algebra. Using the basis vectors, we could construct any vector of the same domain using linear combination. Likewise for orthogonal hierarchies, the separation of interface and implementation allow us to have a combination of the different derived types.

The bridge patterns are notably used to encapsulate classes that would need to work on multiple platforms, which would require multiple implementations.

Consider a scenario where we have multiple types of subclasses that has an orthogonal property such as one depicted in figure 17. Notice that if we were to add more types of guns or more gun attachments, the number of derived classes that need to be implemented grows geometrically. Resulting in highly un-maintainable and repetitive code.

Note that all the derived classes share some commonalities. In particular, any derived gun class inherits one of pistol, assault rifle, or LMG. And has modification of one of scope, suppressor, and ergonomic grip. By using the bridge pattern to decouple the implementation from the interface, we can make *orthogonal hierarchies* (figure 18) which is more manageable.

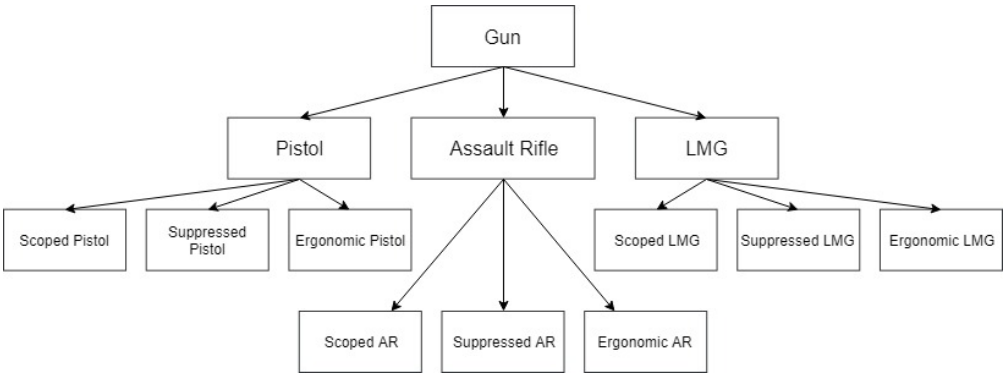


Figure 17: Structure of orthogonal hierarchy before using bridge pattern

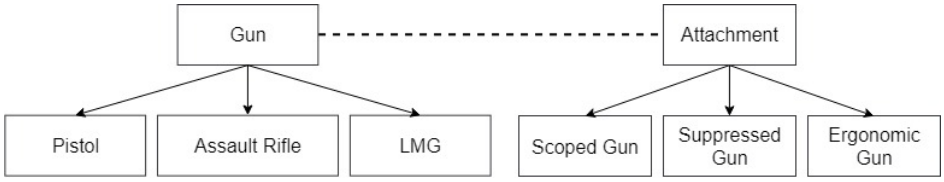


Figure 18: Structure of orthogonal hierarchy after using bridge pattern

4.3 Composite

The composite design pattern allow objects to be composed into nested hierarchies or tree structures. The client can treat the composited object or the composition of objects the same way.[6]

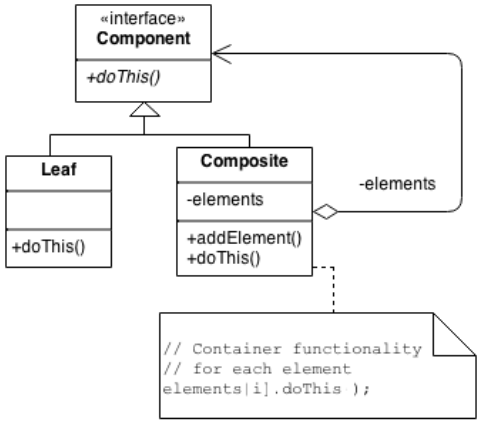


Figure 19: Simple structure of composite design pattern[6]

As seen in figure 19, the client would interact with the root interface *Component*. The derived *Composite* class holds a list of elements with the type *Component* such that a recursive/tree-like structure could be constructed. Alongside, the the derived *Leaf* class has more unique data and behaviors, but still desired to be part of the composition.

The result is that we can have nested relationships abstracted. The interface allows all nested classes to be treated the same by the client. Such as the example shown in figure 20 of a composited mathematical expression. Since the composite is recursive, the interface could expose a single call to the client to evaluate all children composites and leaf objects.

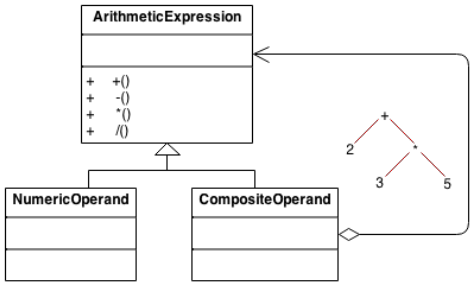


Figure 20: Composite pattern used for representing mathematical expression[6]

The composite pattern is apparent in UI layouts. For instance, a standard desktop application window has the main window widget itself, but it has nested components inside it. These nested components are also sometimes expected to hold items inside themselves.

As seen in figure 21, the main window has a menu bar, the menu bar contains menu buttons which open to another list of menus and the nesting goes on.

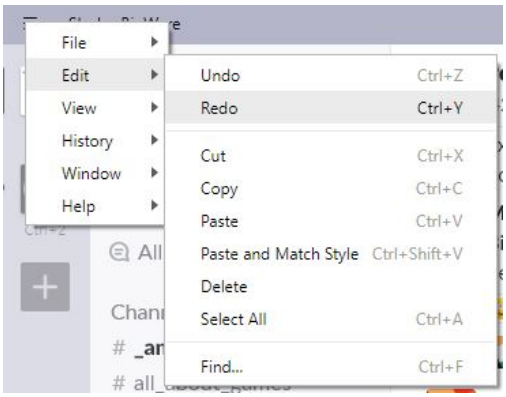


Figure 21: Composite pattern in menu systems in UI

Similar ideas apply to in-game UIs, where elements that composes the game UI are made of smaller components known as *UIWidgets*. Designers put multiple UIWidgets together, sometimes reusing existing ones to create a screen of UI, as seen in figure 22.

Figure 22: UI menus in Anthem[7]

4.4 Decorator

Consider the following problem: suppose we have a "gift" class whose purpose is to hold some intrinsic information about the gift, such as what it is and its value (i.e. an avocado, with value of $1.20). However, there exists times when we need to delegate responsibilities the *gift* class dynamically (during run-time), such as concealing the gift value or wrapping the gift and putting it in a box. In this case, the wrappers and boxes would be extra responsibilities delegated to the gift. However, we would not want to define this statically as it is not flexible (what if we want to different wrapping paper depending on the recipent). Thus, it would be wise to utilize the decorator design pattern.

As the name suggests, the decorator design pattern "decorates" around the class that we want to assign new responsibilities to. To apply this pattern in software, we first create an *interface class* that holds the lowest common denominator attributes and functions of the **core class** (the class that holds the core functionalities) and the **decorator class** (the class that decorates the core classes). We then create the *decorator* base class that implements the interface a level down. The interface allows programmers to establish a loosely coupled relationship in code, and being a base class opens the option for more decorator variations.

Note that both *core* and *decorator* classes extends the interface via "is a" relationship. In addition, *decorator* holds a memeber variable of the interface type. This way, the decorator classes can actually perform the tasks that delegates to the core classes or more decorator classes.

Lastly, the client is responsible for managing the creation of the decorators as well as the composition of the decorator classes added to the core classes.

In games, such pattern is applicable in game assets that have the ability to be modified. To name a few, these game objects could be potion effects to playable or non-playable characters (NPCs) (figure 23),[29] modifications to vehicles (figure 24),[30] or enchantment on tools or weapons (figure 25).[31]

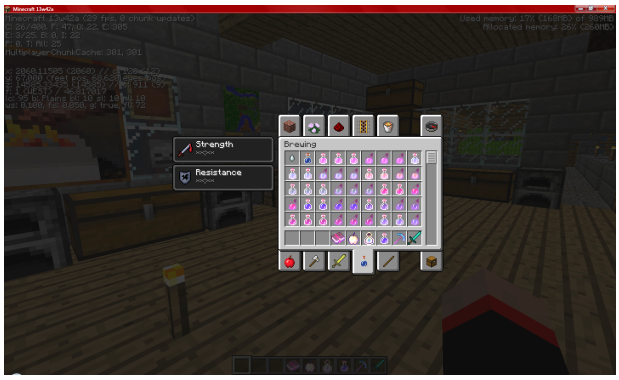


Figure 23: Potion effects on the player in the game Minecraft

Figure 24: Vehicle customization in Battlefield 4

Figure 25: Enchantment that modifies tool or weapon behaviors in Minecraft

The decorator is a structural design pattern to enhance the flexibility of OOP software by allowing extra behaviors to be attached to the core class during run-time. The decoupling of core class functionalities from the potential (but optional) decorator functionalities reduces the tedious need of programming every permutation of core and decorator classes. Thus, the codebase is more maintainable as well. The decorator design pattern shows no change in performance.

Sample code for an exmaple of how the decorator pattern is used in a generic UI scenario is available in section A.5.

4.5 Flyweight

Flyweight is a structural design pattern that optimizes computation efficiency by making clients use shared resources that are computationally expensive.[32]

Consider when we want to load a website and the web page we are loading into may contain multiple references of an image. (i.e. the website logo, or an advertisement). Usually, these images are external references to another URL, so it takes time to access that URL and retrieve the image, making it relatively timely expensive. Therefore, it is undesirable to fetch the image one hundred times if the webpage displays the same image one hundred times. It would be reasonable to cache the first instance, then reuse the resource for all subsequent referenes.

We also want to have some unique information about each instance. In the website example, even though all one hundred images are identical (the data they shared are considered *intrinsic*), they have to have some unique (*extrinsic*) information about them, such as position on the page, size, etc.

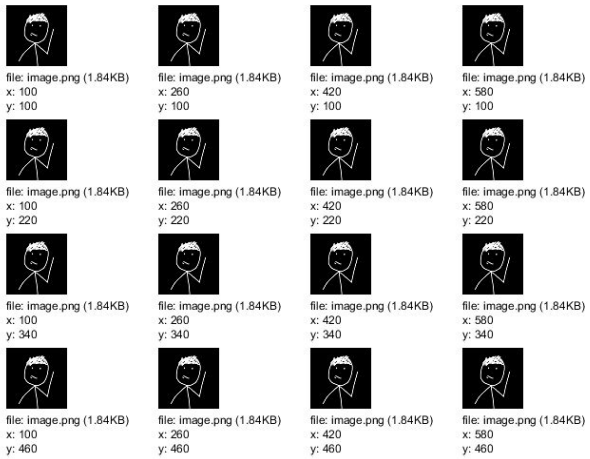
The flyweight states that we separate the intrinsic and extrinsic part of a particular class.[32] The intrinsic data is used for comparison and cached. To check if a particular instance of a class with same intrinsic data exists, a hashing algorithm may be utilized. [vi]

Lastly, a factory (section 2.2) is used to create new instances to provide an extra process for instance creation.[32]

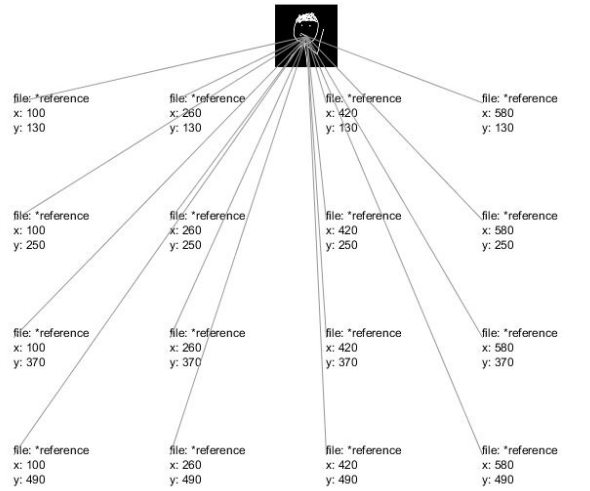
In video games, the flyweight design patterns are used to cut down spacial resources. In most cases, these resources are data and assets that uses relatively high memory, such as high-definition textures, sound, or other forms of data such as timeline, animation keyframes, and motion tracking.

I setup a small example in figure 26 on how flyweight could apply to textures. In my setup, I display 16 textures located at different (x, y) coordinates. In the former example (figure 26a), the image is loaded total of 16 times. In the second example in figure 26b, the image is only loaded once upon the initial draw request, but all subsequent calls simply refers to the image cached from the first time.

[vi] A hashing algorihtm takes an object and outputs a unique number; this process is fast and cannot be reversed.



(a) Without using flyweight



(b) Using flyweight

Figure 26: Visual comparison of not using and using the flyweight design pattern

Assuming the following data sizes in table 1, the reduction in spacial resources are significant (see table 2). In our experimental setup, the memory used is up to 95% less due to elimination of repeated resources. With less memory resource to process, the pattern naturally also reduces computation resources leading to faster performances.

Data type	Size (B)
Image	1840
int	2

Table 1: Assumed data type sizes in the flyweight experiment

	# Images	Image memory (kB)	`int` memory (B)	Total memory allocated (kB)
No flyweight	16	29.44	64	29.504
With flyweight	1	1.84	64	1.904
			% Reduction	-93.5%

Table 2: Performance differences of not using vs. using flyweight

5 Conclusions

The classic software development design pattern as described by the *Gang of Four*[1] is undoubtly essential in even modern software development. In video game technology and design, the need is amplified. The need to desgin pattern is important as video games are computationally demanding, both in terms of process as well as memory. As consumers (players) push for more realistic and immersive experience, the technology becomes more complex. So it is also important to use these "basis" solutions to simply more complex solutions.

The most common, and perhaps important, benefit of certain design patterns is that they decouple code. Making otherwise *spaghetti code* or codebase filled with tech-debt usable in modern applications. Creation design patterns helps manage the creation and lifetime of objects. Behavior design patterns

5.1 Recommendations

Creational design patterns should be mostly applied only where there are many variation of a particular class. These classes often have the property of holding some generic logic and IO, such as a button, but have different implemenation or extra bahavior or data. Creational design patterns should not be used if not needed as it might create unncessary levels of abstraction such as abstract factories from section 2.2, or breakage of abstraction such as a singleton from section 2.5.

Behavioral design patterns assist in managing communication and interaction between classes or objects. These communication Behavior design patterns provides bettered managed communication or interaction between classes and emphasizes on decoupling.

Structural patterns improves performance and maintainability of the software while keeping the code clean and organized. Structural design patterns should be applied when the software is large and difficult to debug. Structural patterns should also be used to conserve resources. Structural patterns should not be used to resolve poor design as they are not solutions to design problems but rather problems due to limitation of the programming language (the technology).

All design patterns investigated here have some prerequisit, meaning that each are meant to solve one type of problem and one type of problem only. Combinations of patterns can be used to build clean code, however one should never try to *start* a code base with design patterns. Design patterns are meant to optimize and polish existing code and solve problems that are often only related to memeory, computation overhead reduction and organization. Design patterns should never be applied on non-issue or problems that do not yet exist.

References

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns, Element of Reusable Object-Oriented Software." Addison-Wesley Professional Computing Series, 1994. Online; accessed Oct 2018.

[2] D. Sector, "Unity vs. Unreal Engine - Begineers Guide to Game Development." https://www.xiaomitoday.com/unity-vs-unreal-engine-comparsion-review/, Jun 28 2017. Online; accessed Dec 2018.

[3] Y. Xu, "Drawing UML Class Diagram by using pgf-umlcd." https://mirror.hmc.edu/ctan/graphics/pgf/contrib/pgf-umlcd/pgf-umlcd-manual.pdf, Jan 31 2012. Online; accessed Dec 2018.

[4] "Anthem Full Gameplay Demo - E3 Best Action Game Winner." https://www.youtube.com/watch?v=p-nKttWT7JM, Jun 2018. Online; accessed Dec 2018.

[5] "Unreal Engine Tutorial Part Five: Pawns, Character Controllers and Handling Input." https://www.gamefromscratch.com/post/2015/05/29/Unreal-Engine-Tutorial-Part-Five-Pawns-Character-Controllers-and-Handling-Input.aspx, May 29 2015. Online; accessed Dec 2018.

[6] "Composite Design Patteern." https://sourcemaking.com/design_patterns/composite, 2018. Online; accessed Dec 2018.

[7] "Javelin Personalization - Anthem Developer Livestream from November 15." https://www.youtube.com/watch?v=ivGNly4umo8, Nov 16 2018. Online; accessed Dec 2018.

[8] "Frostbite." https://www.ea.com/frostbite, 2018. Online; accessed 2018.

[9] "Frostbite (game engine)." https://en.wikipedia.org/wiki/Frostbite_%28game_engine%29, 2018. Online; accessed 2018.

[10] "Object Oriented Programming (OOP)." https://www.techopedia.com/definition/3235/object-oriented-programming-oop, 2016. Online; accessed Dec 2018.

[11] A. Shvets, G. Frey, and M. Pavlova, "Design Patterns." https://sourcemaking.com/design_patterns, 2018. Online; accessed Nov 2018.

[12] A. Shvets, G. Frey, and M. Pavlova, "AntiPatterns." https://sourcemaking.com/antipatterns, 2018. Online; accessed Nov 2018.

[13] "Design Patterns." https://en.wikipedia.org/wiki/Design_Patterns, 2018. Online; accessed Dec 2018.

[14] A. Shvets, G. Frey, and M. Pavlova, "Creational Patterns." https://sourcemaking.com/creational_patterns, 2018. Online; accessed Nov 2018.

[15] R. Hughes, "The Telescoping Constructor (Anti)Pattern." http://www.captaindebug.com/2011/05/telescoping-constructor-antipattern.html#.W_XGGUxFwis, May 21 2011. Online; accessed Nov 2018.

[16] "Factory Method Design Pattern." https://sourcemaking.com/design_patterns/factory_method, 2018. Online; accessed Dec 2018.

[17] "Simple Factory - Design Patterns for Humans." https://github.com/kamranahmedse/design-patterns-for-humans#-simple-factory, 2017. Online; accessed Nov 2018.

[18] "Abstract Factory Design Pattern." https://sourcemaking.com/design_patterns/abstract_factory, 2018. Online; accessed Dec 2018.

[19] "Abstract Factory in C++." https://sourcemaking.com/design_patterns/abstract_factory/cpp/2, 2018. Online; accessed Dec 2018.

[20] "Horizon Zero Dawn." Sony Entertainment, 2017.

[21] "Design Pattern - Singleton Pattern." https://www.tutorialspoint.com/design_pattern/singleton_pattern.htm, 2016. Online; accessed Dec 2018.

[22] "Singleton Pattern." https://www.oodesign.com/singleton-pattern.html, 2015. Online; accessed Dec 2018.

[23] "Singleton Design Pattern." https://sourcemaking.com/design_patterns/singleton, 2018. Online; accessed Dec 2018.

[24] "Tetris (Nintendo) NES Manual." https://www.retrogames.cz/manualy/NES/Tetris_(Nintendo)_-_NES_-_Manual.pdf, 1989. Online; accessed Jan 2019.

[25] "Super Mario Bros. Instructions Booklet." https://legendsoflocalization.com/media/super-mario-bros/manuals/Super-Mario-Bros-Manual-US.pdf, 1985. Online; accessed Jan 2019.

[26] "Game Accessibility Guidelines." http://gameaccessibilityguidelines.com/counterstrikego-control-options/, 2012. Online; accessed Jan 2019.

[27] "Mediator." https://sourcemaking.com/design_patterns/mediator, 2018. Online; accessed Nov 2018.

[28] "Adapter Design Pattern." https://sourcemaking.com/design_patterns/adapter, 2018. Online; accessed Dec 2018.

[29] "Items with enchantment effect cause blocks and potion effects indicator to not render." https://bugs.mojang.com/browse/MC-35643, 2013. Online; accessed Jan 2019.

[30] P. Haas, "Battlefield 4 Vehicle Customization: Boats Don't Suck Anymore." https://www.cinemablend.com/games/Battlefield-4-Vehicle-Customization-Boats-Don-t-Suck-Anymore-56779.html, 2014. Online; accessed Jan 2019.

[31] "Tim the Enchanter." https://dev.bukkit.org/projects/enchanter, 2015. Online; accessed Jan 2019.

[32] "Flyweight Design Pattern." https://sourcemaking.com/design_patterns/flyweight, 2018. Online; accessed Jan 2019.

A Sample Code

A.1 Abstract Factory

```
}

/// Abstract factory: useful for platform specific things with different concrete
/// implementations
class Button
{
public:
    virtual void draw() = 0;
};

// Derived platform specific
class KeyboardButton : public Button
{
public:
    void draw() override { /* draws a key */ }
};

class ControllerButton : public Button
{
public:
    void draw() override { /* draws a controller button */ }
};

// The abstract factory to create all related objects
class ButtonAbstractFactory
{
public:
    virtual Button* makeButton();
};

// Concrete factory to create concrete objects in the type of abstract
class KeyboardButtonFactory : public ButtonAbstractFactory
{
public:
    Button* makeButton() override { return new KeyboardButton(); }
};

class ControllerButtonFactory : public ButtonAbstractFactory
```

A.2 Adapter

```
// Suppose the rectangle that comes in the library is defined as:
class Rectangle
{
public:
    Rectangle(int x, int y, int w, int h)
    {
        // ...
    }
};

// An adapted rectangle class to fit client need
class Rect : public Rectangle
{
public:
    Rect(int x1, int y1, int x2, int y2)
    : Rectangle(x1, y1, x2 - x1, y2-y1)
    {
        // Legacy rectangle created; constructor interfacing correctly
    }
};
```

A.3 Builder

```
typedef struct
{
    string name = "Alex";
    int age = 20;
    float height = 170.0;
    float weight = 180.0;
} PlayerCharacterBuilder_t;

class PlayerCharacter
{
public:
    PlayerCharacter(PlayerCharacterBuilder_t &builder)
    : m_name(builder.name)
    , m_age(builder.age)
    , m_height(builder.height)
    , m_weight(builder.weight)
    {
        // do stuff...
    }

private:
    string m_name;
    int m_age;
    float m_height;
    float m_weight;
};
```

A.4 Composition Over Inheritance

A.4.1 Components

```
class GameObject
{
public:
    virtual ~GameObject() {}
    virtual void update() {}
    virtual void draw() {}
    virtual void collide(GameObject objs[]) {}
};

class Visible : public GameObject
{
public:
    void draw() override{};
};

class Solid : public GameObject
{
public:
    void collide(GameObject objs[]) override {}
};

class Movable : public GameObject
{
public:
    void update() override {}
};
```

A.4.2 Usage

```
// Composed classes
```

```
class Player : public Visible, public Solid, public Movable
{
    //...
};

class Building : public Visible, public Solid
{
    //...
};

class Ghost : public Visible, public Movable
{
    //...
};
```

A.5 Decorator

A.5.1 Decorator Interface

The interface class establishes that both core and decorator classes would be a widget, and would be drawn to the screen.

```
class IWidget
{
public:
    virtual void draw() = 0;
}
```

A.5.2 Core

The core class, without any added "decorations" should hold a width, height, and some text information to display.

```
class TextFieldWidget : public IWidget
{
public:
    TextFieldWidget(int w, int h)
    : m_width(w)
    , m_height(h)
    , m_text("")
    {
    }

    void draw() override
    {
        std::cout << "Draws textfield: " << m_text << "\n";
    }

    void setText(const std::string& text) { m_text = text; }

protected:
    int m_width;
    int m_height;
    std::string m_text;
};
```

A.5.3 Decorator

The decorator classes holds a member with a type of widget interface such that it can delegate the `draw()` function call to the other decorator or core classes. In addition, the derived class also perform extra functionalities that is unique to that decorator class.

```
class DecoratorBase : public IWidget
{
public:
    DecoratorBase(IWidget *w) : m_widget(w) { }
    void draw() override { m_widget->draw(); }
protected:
    IWidget *m_widget;
};

class BorderDecorator : public DecoratorBase
{
public:
    BorderDecorator(IWidget *w) : DecoratorBase(w) { }
    void draw() override
    {
        // Call parent
        DecoratorBase::draw();

        // Does own functionalities (extra decorator responsiblities)
        std::cout << "Draws border\n";
    }
};

class ScrollbarDecorator : public DecoratorBase
{
public:
    ScrollbarDecorator(IWidget *w) : DecoratorBase(w) { }
    void draw() override
    {
        DecoratorBase::draw();
        std::cout << "Draws scroll bar\n";
    }
};
```

A.5.4 Usage

```
int main(void)
{
    // Core widget
    IWidget* textfield = new TextFieldWidget(200, 50);
    textfield.setText("Hello APSC!");

    // Decorators that wrap it in a "chain" of wrappers
    IWidget* wrappedTextfield = new BorderDecorator(
        new ScrollbarDecorator(textfield)
    );

    // Interact with wrapped widget with added functinoalities
    wrappedTextfield->draw();

    return 0;
}
```

A.6 Flyweight

This section of sample code demonstrates how a flyweight design pattern may be applicable in a simple texture resources management scenario, where memory allocation is important.

A.6.1 Core Class (Intrinsic)

The intrinsic data, or core class contains the *shared* information.

```
class Texture
{
public:
    Texture(const str::string& path) : m_texture(path) { }
private:
    str::string m_texture;
};
```

A.6.2 Wrapper Class (Extrinsic)

Extrinsic data in wrapper classes contain information that is unique to that particular instance. In this case, position in `(x, y)`.

```
class TextureWrapper
{
public:
    TextureWrapper(Texture* texture) : m_texture(texture) { }
    void setPos(int x, int y)
    {
        m_x = x;
        m_y = y;
    }
private:
    Texture* m_texture;
    int m_x;
    int m_y;
};
```

A.6.3 Flyweight Creation and Logic

The flyweights creation are managed by a simple factory.

```
class TextureFlyweightFactory
{
public:
    static TextureWrapper* getTexture(const std::string& path, int x, int y)
    {
        TextureWrapper newTexture;

        // Check if texture is cached by comparing hashes
        const int textureHash = getStringHash(path);
        if (m_cachedTextures.find(textureHash) != m_cachedTextures.end())
        {
            // Exists
            newTexture = new TextureWrapper(m_cachedTextures[textureHash]);
        }
        else
        {
            // Does not exist, create new then add to cache
            Texture *texture = new Texture(path);
            newTexture = new TextureWrapper(texture);
            m_cachedTextures[textureHash] = texture;
        }

        newTexture->setPos(x, y);
        return newTexture;
    }

private:
    static std::unordered_map<int, Texture* m_cachedTextures;
};
```

To make comparison of the requested instance versus the list of cached instances, we compare their hash value. Therefore a simple hashing algorithm can be used:

```
int getStringHash(const std::string& str)
{
    int hash = 13;
    for (int i = 0; i < str.length(); i++)
    {
        hash = hash * 101 + str[i];
    }
    return hash;
}
```

A.6.4 Usage

Example of how to use the design pattern.

```
int main(void)
{
    std::string texture1 = "player.png";
    std::string texture2 = "enemy.png";

    // Create texture
    TextureWrapper textures[5] = {

        // First instance (expect creating new texture resources)
        new TextureFlyweightFactory::getTexture(texture1, 0, 0),
        new TextureFlyweightFactory::getTexture(texture2, 10, 10),

        // Repeated instances (expect sharing Texture resource)
        new TextureFlyweightFactory::getTexture(texture1, 10, 0),
        new TextureFlyweightFactory::getTexture(texture1, 0, 10),
        new TextureFlyweightFactory::getTexture(texture2, 20, 20)
    };

    return 0;
}
```

A.7 Observer

A.7.1 Subject Abstract Observer Class

The simple implementation of a observer pattern is as follows. Notice that the subject is loosely coupled to the observer. But the observer is tightly coupled to the subject.

```
#include <iostream>
#include <string>
#include <vector>

class ObserverBase;
class Subject
{
public:
    Subject() : m_data(0) { }
    void setData(int data)
    {
        if (m_data == data)
            return;

        m_data = data;
        sendEvent(m_data);
    }

    // Getter could be for when observers want to pull data
    int getData() const { return m_data; }
```

```
    // Register/unregister observers
    void registerObserver(ObserverBase* observer) { m_observers.push_back(observer); }
    bool unregisterObserver(ObserverBase* observer)
    {
        for (auto observerIter = m_observers.begin();
        observerIter != m_observers.end(); observerIter++)
        {
            if (*observerIter == observer)
            {
                m_observers.erase(observerIter);
                return true;
            }
        }
        return false;
    }

protected:
    void sendEvent(int data) const
    {
        for (auto observer : m_observers)
        {
            observer->handleEvent(data);
        }
    }

private:
    int m_data;
    std::vector<ObserverBase*> m_observers;
};

class ObserverBase
{
public:
    // Constructor
    ObserverBase(Subject *subject)
    {
        m_subject = subject;
        m_subject->registerObserver(this);
    }

    // Unregister observer from subject to avoid invalid memory access
    ~ObserverBase() { m_subject->unregisterObserver(this); }

    virtual void handleEvent(int data) = 0;

protected:
    const Subject* getSubject() { return m_subject; }
private:
    Subject* m_subject;
};
```

A.7.2 Concrete Observers

Here are some examples of concrete observers that inherits the observer base class. They override the virtual `handleEvent` method to handle whatever they're designed to do. In this case, any instances of *ZeroObservers* registered to a subject would print out "Subject received data=0" whenever the something calls subject's `setData()` method and passes in 0.

```
class ZeroObserver : public ObserverBase
{
public:
    ZeroObserver(Subject* subject) : ObserverBase(subject) { }
    void handleEvent(int data) override
    {
        if (data == 0)
            std::cout << "Subject received data=0\n";
```

```
    }
};

class EvenNumberObserver : public ObserverBase
{
public:
    EvenNumberObserver(Subject* subject) : ObserverBase(subject) { }
    void handleEvent(int data) override
    {
        if (data % 2 == 0)
            std::cout << "Subject received data divisible by 2\n";
    }
};
```

For more flexible applications, the observer also could also get a subject pointer and pull the data from the subject upon an event. Consider the following concrete observer class:

```
class NegativeObserver : public ObserverBase
{
public:
    NegativeObserver(Subject* subject) : ObserverBase(subject) { }
    void handleEvent(int data) override
    {
        if (data < 0)
        {
            const int subjectData = getSubject()->getData();
            std::cout << "Subject received negative data: " << subjectData << "\n";
        }
    }
};
```

We invoke `getSubject()` method (to ensure we can make no changes to the subject itself) and call the getter directly.

```
void example()
{
    // Subject
    Subject* subject = new Subject();

    // Observers
    ZeroObserver zObs(subject);
    EvenNumberObserver evenObs(subject);
    NegativeObserver negObs(subject);

    // Only negative observer should print
    subject->setData(-1);

    // Only even number and zero observer should print
    subject->setData(0);

    // Only even number and negative observer should print
    subject->setData(-4);
}
```

A.8 Simple Factory

```
// Base class
class Shape
{
public:
    virtual float getArea();
    void setId(int id) { m_id = id; }
protected:
    float m_width;
    float m_height;
```

```
    float m_id = 0;
};

// Derived classes
class Rectangle : public Shape
{
public:
    Rectangle(float w, float h)
    {
        m_width = w;
        m_height = h;
    }

    float getArea() override { return m_width * m_height; }
};

class Triangle : public Shape
{
public:
    Triangle(float w, float h)
    {
        m_width = w;
        m_height = h;
    }

    float getArea() override { return 0.5 * m_width * m_height; }
};

// Factory
class SimpleShapeFactory
{
public:
    static Shape* makeRect(float w, float h)
    {
        Shape* newRect = new Rectangle(w, h);
        newRect->setId(m_idCount++);
        return newRect;
    }

    static Shape* makeTri(float w, float h)
    {
        Shape* newTri = new Triangle(w, h);
        newTri->setId(m_idCount++);
        return newTri;
    }
private:
    static int m_idCount;
};
```

A.8.1 Not Using Factory

Not only that we need to create the objects at first, we also need to ensure that every object is properly initialized correctly. In this case, we're using a for-loop. However, one can see that as number of things to initialize increase, and initialization gets more complex, the client would need to do much more work.

```
void BeforeSimpleFactory()
{
    Shape* shapes[] = { new Rectangle(5, 4), new Triangle(3, 1) };

    // Then we need to apply any applicable instantiation logic to each one
    for (int i = 0; i < 2; i++)
    {
        shapes[i]->setId(i);
    }
}
```

A.8.2 Using Factory

Notice that the initialization logic is hidden away at this level. So using the code is much simpler.

```
void AfterSimpleFactory()
{
    Shape* shapes[] = {
        SimpleShapeFactory::makeRect(5, 4),
        SimpleShapeFactory::makeTri(3, 1)
    };
}
```

A.9 Singleton

```
// i.e. we want to have an achievement tracker, but the entire game should only have one
class AchievementTracker
{
public:
    // public static method getter
    static AchievementTracker *getInstance()
    {
        // Lazy initialization
        if (!m_instance)
        {
            m_instance = new AchievementTracker();
        }

        return m_instance;
    }

    // Other stuff
    void reset() { m_counter = 0; }
    void increment(int value) { m_counter += value; }
    bool getAchievementUnlocked() { return m_counter >= 5; }

private:
    // Private constructor
    AchievementTracker()
    {
        m_counter = 0;
    }

private:
    // Private static member to hold the single instance
    static AchievementTracker *m_instance;
    int m_counter;
};
```