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
December 12, 2018

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.

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

(Working at EA-Bioware) (Associate Developer (Programmer) for game Anthem) (Using modern game engine Frostbite)

Scope of Coverage

(The scope of this report)(Covers many design patterns outlined in "Gang of Four")(Will not cover detailed breakdown, explaination, and specification of existing proprietary tech; i.e. extended code from the game or Frostbite engine)

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.

As the scope of game development or software development extends wide in breadth, this report will not throughly 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

(Gibson, Tim - senior software lead, manager: contribution by supervising the report and giving advice) (Johnson, Chris - supervisor, programmer, assisted in design pattern exampls) (Steadman, Andrew)

Summary

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, templated 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 adoped at Electronic Arts (EA) studios. It was originally developed by Digital Illusions CE (DICE).[2][3] 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.

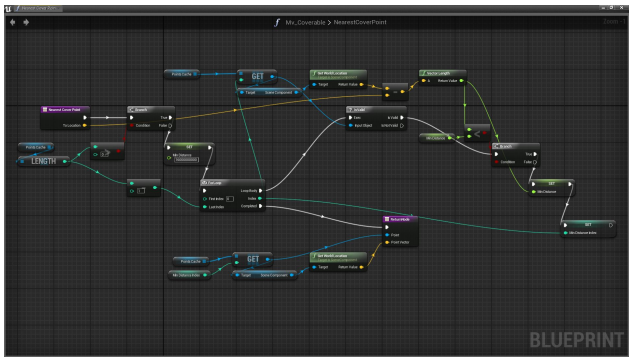

Figure 1: Blueprint (schematics) for development in Unreal Engine[1]

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 widgest 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.[4]

Although we will not go into depth into the subject of object oriented programming, it is vital to undertand that OOP is basis of most modern game development due to its benefits of resuability, refactoring, extensibility/scalability, maintenance, and efficiency.[4] 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 developent.[5] Design patterns are meant to be used when there no problems actually exist – doing so would cause excess undesired complexity.

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. (TODO:)

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.[6]

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.[7] 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 inheritantly 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.[8] 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,[9] 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 paramters 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.2.

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[10] 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.[11]

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.

(Simple factory in game developement ehhhhh).

A sample code for this pattern is available in section A.4.

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.[12] 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.[13]

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.[14]

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

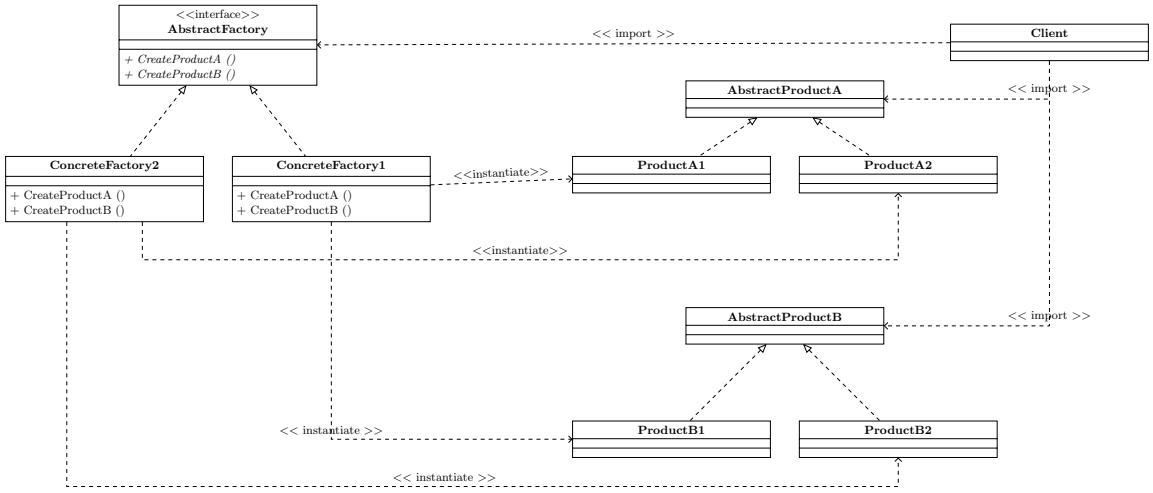

Figure 5: A generic UML diagram of abstract factory pattern

2.3 Object Pool

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 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. Finally, using 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*).

(Objective markers that need to be displayed on the screen: they are persistent in the world, but not all of them needs to be displayed all the time. Only the ones inside players' camera frustrum 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.)

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 procedrally 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.[ii] 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 teh 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 recomputation 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.[15] 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.[16,17]

In game development, singletons can be used for for system managers. i.e. a single manager class

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

that overlooks and orchestrates UI system or a system to track achievements. (see section 5.1 for component-entity-system architecture where a single system manages multiple objects).

Singletons are also useful for configuration classes and shared resource accessing classes.[16] An example in game development is levels and personalization libraries (customizations) 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 implemantion holds a static reference to itself and a static getter that returns that reference (figure 6). In singleton implementations, concept of *Lazy implentations*, where we do not initialize and allocate the memory for the object until client actually requests it.

Singleton
-instance: Singleton*
+Singleton(): +getInstance(): Singleton*

Figure 6: Generic singleton UML class diagram

An example of Singleton implementation for an simple acheivement tracker in sample code in section A.5.

3 Behaviour Design Patterns

TODO:

3.1 Chain of Responsibility

(InputHandlers and InputDispatchers) in Frostbite engine. A series of input handlers are chained together. Each input handler may hold reference to other input handlers. When an input, such as mouse, keyboard, or controller action occurs. The calls traverses down the chain of input handlers until one of the input handler entities "consumes" it, which ends the chain.

Another chain of command pattern observed in the Forstbite engine editor, is handling mouse press when there are multiple *hitzones*[iii] overlapped (figure 7). 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.

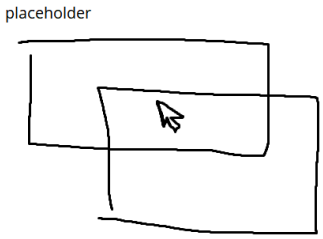

Figure 7: Mouse interacting with overlapping mouse hitzones

3.2 Command

Decouples action request and action handlers. The callee who makes the request makes no assumptions about the class the receives the command. Effectively decoupling them. This opens up more flexibility in mapping a requets to a handler. An example is remappable input handling. Instead of listening for a key press, then calling a specific function that the key press corresponds to. It invokes a command object that holds a reference to the *real object* so that the handler can be called. In between we could do more things. For instance, keep track of how many times the action has requested. Command pattern also opens up for redo/undo operations.

3.3 Mediator

TODO:

Mediator acts as a middle-man between communicating objects and decouples them.[18]

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

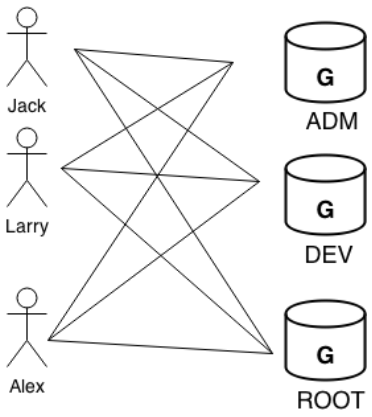

Figure 8: Objects communicating without using mediator pattern

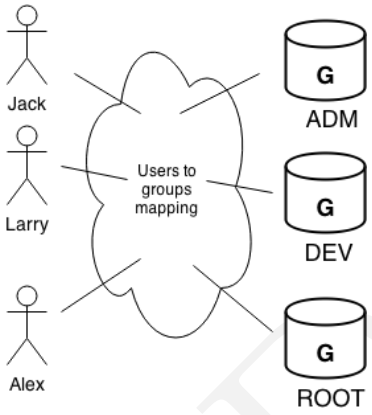

Figure 9: Objects communicating using mediator pattern

As seen in figure 8, 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 9. 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

TODO:
(Desc. of 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.

(How observer works) 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,
(How it's applied in games)
(How it benefits)

3.6 Visitor

TODO:

4 Structural Design Patterns

TODO:

4.1 Adapter

TODO:
A middle layer that transforms the interface of a desired object (usually provided by libraries) to work with client.
Difference Between Adapter and Mediator

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, adapter is structural. And Mediator is behavioral.

4.2 Bridge

TODO:

4.3 Composite

TODO: Useful in architectures such as component-entity-systems model which centers around the idea of "*composition over inheritance*" (section 5.1).

4.4 Decorator

TODO:

4.5 Facade

TODO:

4.6 Flyweight

TODO:

4.7 Proxy

TODO:
Smart pointers, etc.

5 Architectural Patterns

TODO:

5.1 Entity-Component-System

Used in physic engines, and when there are a lot of things to handle.
Other examples include making certain objects in the physics engine adopt a characteristic: i.e. solid/has collision, soft body/hard body, and other physics-aware components.

It could be argued that a "controllable" component could exist to make certain object player-controllable (i.e player characters, car, etc.).

Objects could also inherit components that made them aware of inputs such as mouse. Before Frostbite uses input trees and hit-zone entities, certain UI element could attach mouse interaction components to make them respond to mouse events such as hover, enter and leaving hitzones, click, etc.

Update calls can be queued for all objects at once. Update states can be cached. Cached data are much faster.

5.1.1 Composition over Inheritance

TODO: OOP principle that makes inheritance based on composition / behavior rather than inheritance from parent. Consider code in section A.3.1, rather than defining class by their "likeness", we define it by its function. Then in

5.2 Model-View-Controller

TODO: Mostly used in UI development.

6 Conclusions

TODO:

6.1 Recommendations

TODO:

References

[1] 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.

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

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

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

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

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

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

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

[9] 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.

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

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

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

[13] 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.

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

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

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

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

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

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 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.3 Composition over Inheritance

A.3.1 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.3.2 Usage

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

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

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

A.4 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.4.1 Not using factory

```
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.4.2 Using factory

```
void AfterSimpleFactory()
{
    Shape* shapes[] = {
        SimpleShapeFactory::makeRect(5, 4),
```

A.5 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;
};
```