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
November 28, 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.

Contributors

(Tim Gibson - senior software lead, manager: contribution by supervising the report and giving advice)

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 (Triple-A) 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).[1][2] 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*, which are blocks of game content that contains logic and output, given some input. 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.

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.[1]

1.3 Design Pattern

Design patterns are reusable, general, abstract solutions to common problems in software developent.[3] Design patterns are meant to be used when there no problems actually exist – doing so would cause excess undesired complexity.

1.3.1 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.[4]

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

1.4 Gang of Four

In software development, the phrase "Gang of Four" refers to the four authors of the Design Patterns book. The book features the most common design patterns widely in use today.

2 Creational Design Patterns

Creational design patterns involves with object creation.[5] This is especially useful in game development, such as when we need to create many

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

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, the construction of an instance of an object 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,[6] where there are numerous variations of constructor methods all delegate to the default constructor. The problem is having this many constructor methods with slight variations 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.

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 would have many parameters to be varied (Figure 1).

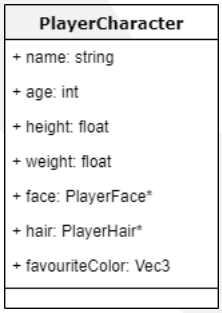

Figure 1: PlayerCharacter class with various desired parameters

2.2 Factories

2.3 Object Pool

2.4 Prototype

2.5 Singleton

In a way, they're just glorified global variable.

In game development, singletons are often reserved for system managers. i.e. a single manager class that overlooks and orchestrates UI system.

Singleton patterns are also used in expensive assets such as levels and personalization libraries (customizations) where we load them into memory during loading screen initially. Then their instance can be accessed via a static function call.

3 Behaviour Design Patterns

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*[2] overlapped (figure 2). 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.

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

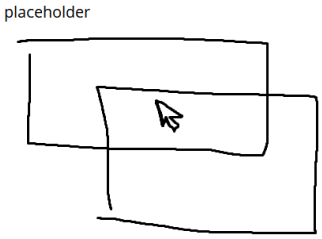

Figure 2: Mouse interacting with overlapping mouse hitzones

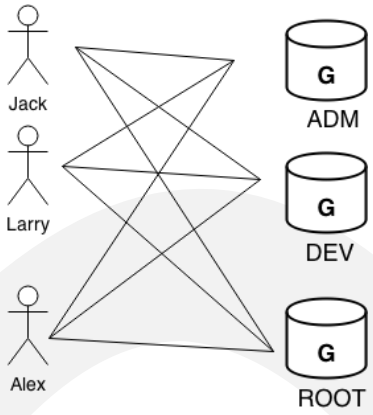

Figure 3: Objects communicating without using mediator pattern

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.

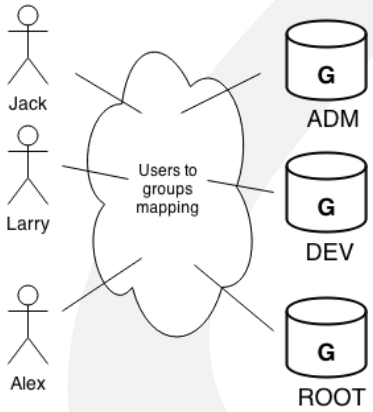

Figure 4: Objects communicating using mediator pattern

As seen in figure 3, 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

3.3 Mediator

Mediator acts as a middle-man between communicating objects and decouples them.[7]

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 4. 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.

In Frostbite, a similar mechanism is called a *SharedLock*.

4 Structural Design Patterns

4.1 Adapter

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

4.1.1 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

4.3 Composite

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

4.4 Decorator

4.5 Facade

4.6 Flyweight

4.7 Proxy

Smart pointers, etc.

5 Architectural Patterns

5.1 Entity-Component-System

Used in physic engines, and when there are a lot of things to handle.
In UI Systems as there are also lots of things to handle. UI widgets do not necessary need to interact with other UI widgets. So using this architecture enables performance optimizations.
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

OOP principle that makes inheritance based on composition / behavior rather than inheritance from parent.

Consider code in section A.1.1, rather than defining class by their "likeness", we define it by its function.

Then in

5.2 Model-View-Controller

Mostly used in UI development.

6 Conclusions

References

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

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

[3] SourceMaking, "Design Patterns." https://sourcemaking.com/design_patterns, 2018. Online; accessed Nov 2018.

[4] SourceMaking, "AntiPatterns." https://sourcemaking.com/antipatterns, 2018. Online; accessed Nov 2018.

[5] SourceMaking, "Creational Patterns." https://sourcemaking.com/creational_patterns, 2018. Online; accessed Nov 2018.

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

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

A Sample Code

A.1 Composition over Inheritance

A.1.1 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.1.2 Usage

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

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

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