GAM250: Advanced Games Programming

2: Design Patterns

Learning outcomes

- **Describe** the concept of Design Patterns
- **Understand** some of the classic 'Gang of Four' Design Patterns
- **Implement** some of the most commmon design patterns

OO Design Basics

Role of Design Patterns

Object orientated systems tend to exhibit recurring structures that promote:

- Abstraction
- Flexibility
- Modularity
- Elegance

Role of Design Patterns

- Therein lies valuable design knowledge.
- The challenge, of course, is to...
  - capture
  - communicate
  - and apply
- ...this knowledge.

Role of Design Patterns

A design pattern...

- Abstracts a recurring design structure
- Comprises class and/or object
  - dependencies
  - structures
  - interactions
  - conventions
- names and specifies the design structure explicitly
- and thereby distils design experience

Components of a Design Pattern

A design pattern is comprised of:

- A name
- Common aliases — *also known as*...
- Real-world examples
- Contexts
- Common problems solved
- Solution
- Structure
- Diagrams
- Consequences

Components of a Design Pattern

- Design patterns are often tacit knowledge made explicit.
- You will develop tacit knowledge of patterns through regular design practice.
- You are expected to engage in constant research and reflection when designing software to learn all of these different patterns.
- They will help you communicate and design in the future.
- Additional research will be required as the number of patterns greatly exceeds those that can be covered in workshops.

Design Patterns

Types of Design Pattern

Design patterns come in three main flavours:

- **creational**: concerned with the process of creating and managing the creation of objects.
- **structural**: dealing with the composition of objects.
- **behavioural**: characterizing the different means by which objects can interact with others.

Types of Design Pattern

- **Creational**
- Singleton
- Typesafe Enum
- Factory
- Prototype
- Builder

- **Structural**
- Adapter
- Bridge
- Proxy
- Facade
- Decorator

- **Behavioural**
- Template
- State
- Observer
- Visitor
- Strategy

Design Patterns

We will now briefly examine these patterns. Throughout this section...

- **Please** make notes
- **Link** to on-line resources
- **Ask** questions
- **Think** about how the patterns may apply to your own projects
- **Conduct** further research

Singleton

- Guarantees that there is only one instance of a class and can be accessed globally
- Usually 'lazily' initialised via a static function that satisfy the statement above
- Used for manager classes which track some sort of Global State
- **Warning!** Some consider Singletons to be an anti-pattern
- Singleton: an anti-pattern? - `https://stackoverflow.com/questions/12755539/why-is-singleton-considered-an-anti-pattern`

Abstract Factory

- Centralises the creation of similar objects
- Decouples the creation of the object from actual object
- This pattern requires several class
  - Abstract Product - Base class for all things created by the Factory
  - Abstract Factory - Base class for all factories, creates Abstract Products
  - Many Concrete Products - Implement Abstract Product
  - Many Concrete Factories - Implements Abstract Factory and creates Concrete Products
- The caller then creates instances of Product through the concrete factory
- Used for spawning objects or the creation of other similar objects

Observer

- When one object is updated, all observers of this object are notified
- A list of observers are mainted by the subject
- When the state of the subject changes then the list of the observers is processed
- Each observer is then notified of the change
- Each observer should register/unregistered itself with a subject
- Very useful for UI, Input or Network systems in games
- Some of this function is already built into C#(delegates & Events) and Unity(Unity Events)

State

- Do you have large amount of if..else or switch statements in your code?
- Have you ever had to change such a system?
- Then the State pattern is here to help
- You define a Base State class which all other States implement
- This Base State will have a method for updating the state, for entering and exiting
- Each Concrete State will then implement these methods and handle its own logic
- Transitions can be handled by a Manager class
- This is generally used to deal with Game State or AI (see Finite State Machines)

Unity Implementations

- Singleton - `https://unity3d.com/learn/tutorials/projects/2d-roguelike-tutorial/writing-game-manager`
- Factory - `http://brightreasongames.com/object-construction-factory-method/`
- State - `http://www.habrador.com/tutorials/programming-patterns/6-state-pattern/`
- Observer - `http://www.habrador.com/tutorials/programming-patterns/3-observer-pattern/`

Live Coding

Singleton & Observer

Exercise 1

- Implement Singleton and Observer in the following project
- Download or fork - `https://github.com/Falmouth-Games-Academy/GAM250-Example-Game`
- The Player score and lives could be tracked using the Observer Pattern
- The Player and Wave Manager could be implemented as Singleton

Exercise 2

- Research Object Pools
- Implement Object Pools for Bullets and Asteroids
- For bonus points, create Factories for these objects

Further Reading

- Game Programming Patterns - `http://gameprogrammingpatterns.com/contents.html`
- Game Programming Patterns in Unity - `http://www.habrador.com/tutorials/programming-patterns/`
- Unity Design Patterns - `https://github.com/Naphier/unity-design-patterns`