Team Lead 3 Presentation

Hayden, Riley, Heath, Zach

Presentation Overview

Introduction

- What is testing?
- Why do we test?
- Verification Vs Validation
- Release Testing
- User Testing
- Agile Methods

Design Patterns

- What are Design Patterns
- Creational Patterns
- Structural Patterns
- Behavioral Patterns
- AntiPatterns
- Arguments against using Patterns
- Requirements for this Class
- Useful Resources
- Examples

Intro to tests + Boundary tests

- EditMode Tests vs PlayMode Tests
- How to set up testing
- Creating boundary tests
- Running tests

Stress tests

- Stress Test Development Demo
- How to interpret Results
- Play mode or edit mode??

What is Testing?

Testing software is just running a program with artificial data.

Tests are intended show that a program does what it is intended to do and to discover defects before it is put into use.

Tests can reveal the presence of errors in your code not the absence.

Tests are apart of a general validation process

Why do we need automated testing?

3 Big reasons:

- Ensure your program can hold up to various inputs
- Communicate to your team whether your code is functional
- Find the limit of your program in order to stay inside of it

Why do we need automated testing? Continued

Testing Goals:

- To Discover situations in which the behaviour of the software is incorrect, undesirable, or not to specification. (defect testing)
- To Demonstrate to the customer and developer that the software meets its requirements. (validation testing)

Verification Vs Validation

Verification: “Are we building the product right”

- The software should conform to its specifications

Validation: “Are we building the right product”

- The software should do what the user needs it to do

Release Testing

Release Testing

Release testing is the process of testing a particular release of a system that is intended for use outside of the dev team

The primary Goal of this testing is to convince the supplier of the system that it is good enough for use.

Thus it has to show the system is dependable, functional, and can perform without fail in normal use

Use Case Testing

Use cases are developed in order to mimic the use of the software as if the user were using it, to help develop test cases for the software.

Example: Login Page

- Enter Login credentials
- Validate credentials
- Give access to User

Requirements based Testing

Requirements Testing involves examining the requirements of the system and developing tests or a test for it

Example: Login Page

-Give access to Password/Username is right

-Reject access if Password/Username is wrong

-limit amount of wrong attempts

Performance Testing

Part of release testing may involve looking at the performance of the system

Tests should reflect the profile of use of the system

Performance tests involves steadily increasing load until performance becomes unacceptable

Stress Tests != Performance Tests

User Testing

User Testing

User testing is a stage in the testing process where the user or customer can provide input on system testing.

User Testing is essential, even when release and system testing have been carried out. The users working environment is essential when tailoring software for performance, usability, and reliability.

Types of User Testing

Alpha Testing

- A User works with a development team to test the software at the developers site.

Beta Testing

- A release of software is made available to gather feedback.

Acceptance Testing

- Customers test a system to determine whether or not it is ready to be accepted and deployed.

Steps in acceptance testing process

- Define acceptance criteria
- Plan acceptance testing
- Derive acceptance tests
- Run acceptance tests
- Negotiate test results
- Reject/accept system

Agile Methods and acceptance testing

In agile methods, the user is part of the dev team and is responsible for making decisions on the acceptability of the system.

Tests are defined by the user/customer and integrated into other tests that are run automatically when changes are made

There is no separate acceptance testing process

Main Problem here is whether or not the customer can represent the typical user

What is required of you?

- Initial Test Plan (Thursday)
    - 1 Stress Test
    - 2 Boundary Tests
- Full test Plan (Oral Exam)
- 2 Patterns somewhere in your code (Oral Exam)
    - Justify the pattern you Choose, talk about where you would not use it

End Section

Questions?

Design Patterns

Design Patterns

- Repeatable solutions to common problems
- Not meant for direct implementation
- Helps communication

Design Patterns

- Creational
- Structural
- Behavioral
- AntiPatterns

Creational Patterns

- Patterns that deal mainly with how objects are created

- Abstract Factory
- Builder
- Factory Method
- Object Pool
- Prototype
- Singleton

Structural Patterns

- Simplifies design by recognizing the relationships between objects

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Private Class Data
- Proxy

Behavioral Patterns

- Patterns that allow communication between objects to operate more efficiently

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Null Object
- Observer
- State
- Strategy
- Template Method
- Visitor

Anti-Patterns

- Common design processes that are known to be defective and can be simplified using design patterns

- The Blob
- Continuous Obsolescence
- Lava Flow
- Ambiguous Viewpoint
- Functional Decomposition
- Poltergeists
- Boat Anchor
- Golden Hammer
- Dead End
- Spaghetti Code
- Input Kludge
- Walking through a Minefield
- Cut-and-Paste Programming
- Mushroom Management

Arguments Against Design Patterns

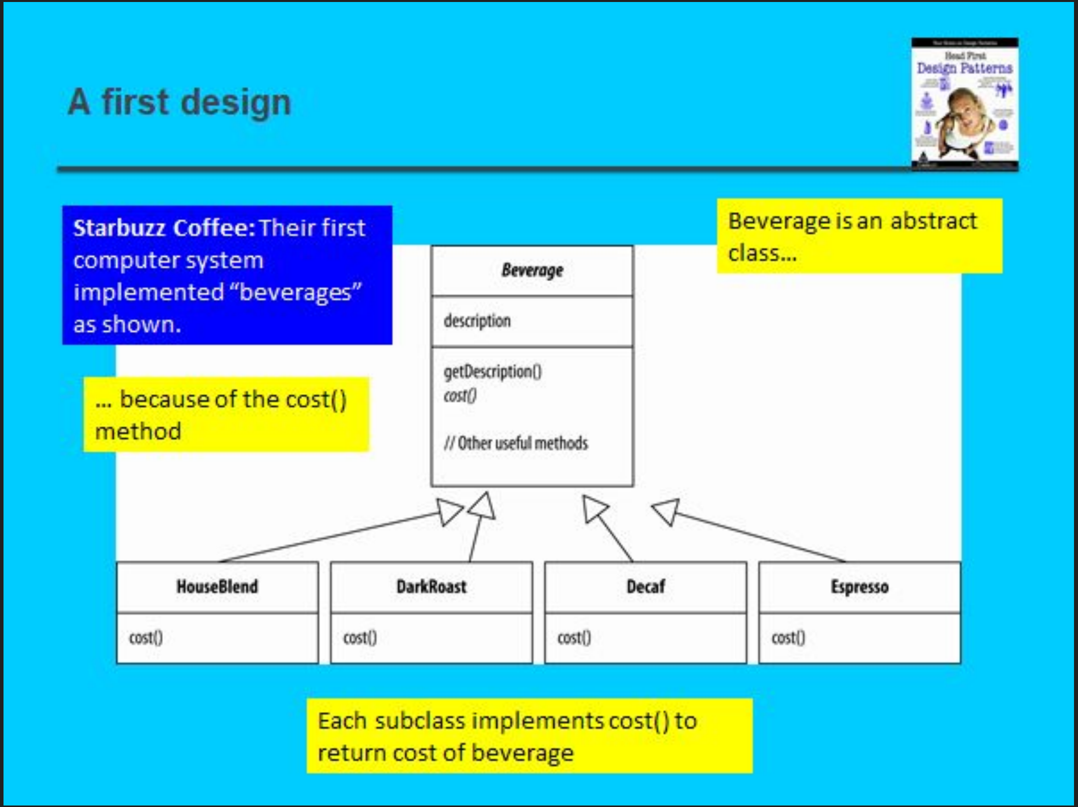
- The need for patterns results from a lack of abstraction in programing languages
- Patterns are not meant to be copied
- Leads to non-optimal solutions

Course Requirements (oral exam)

- 2 patterns implemented in each person’s code
    - Know why you used the specific pattern
    - Be capable of recreating the class diagram of the pattern

Decorator

- Goal is to encapsulate what varies
- Program interfaces without direct implementations



A first design

Starbuzz Coffee: Their first computer system implemented "beverages" as shown.

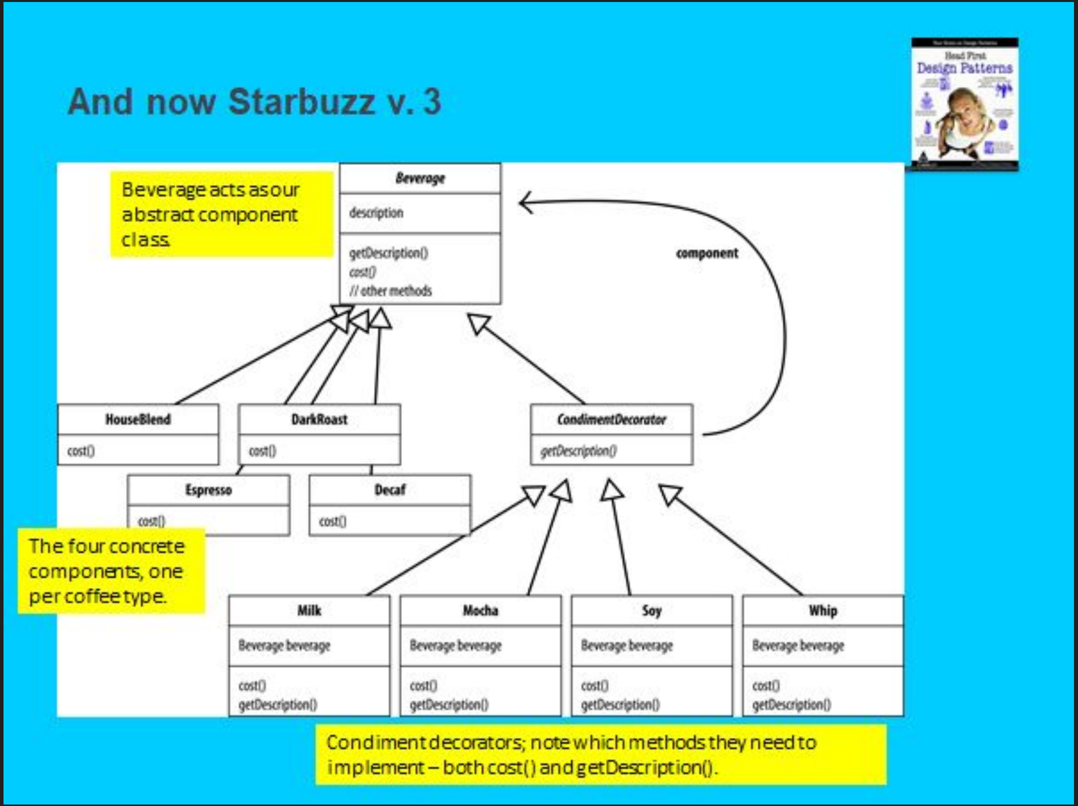
Beverage is an abstract class...

... because of the cost() method

Each subclass implements cost() to return cost of beverage

Decorator

- Decorators wrap objects in order to add additional behavior to them
- An alternative to inheritance
- Components can have multiple decorators



And now Starbuzz v. 3

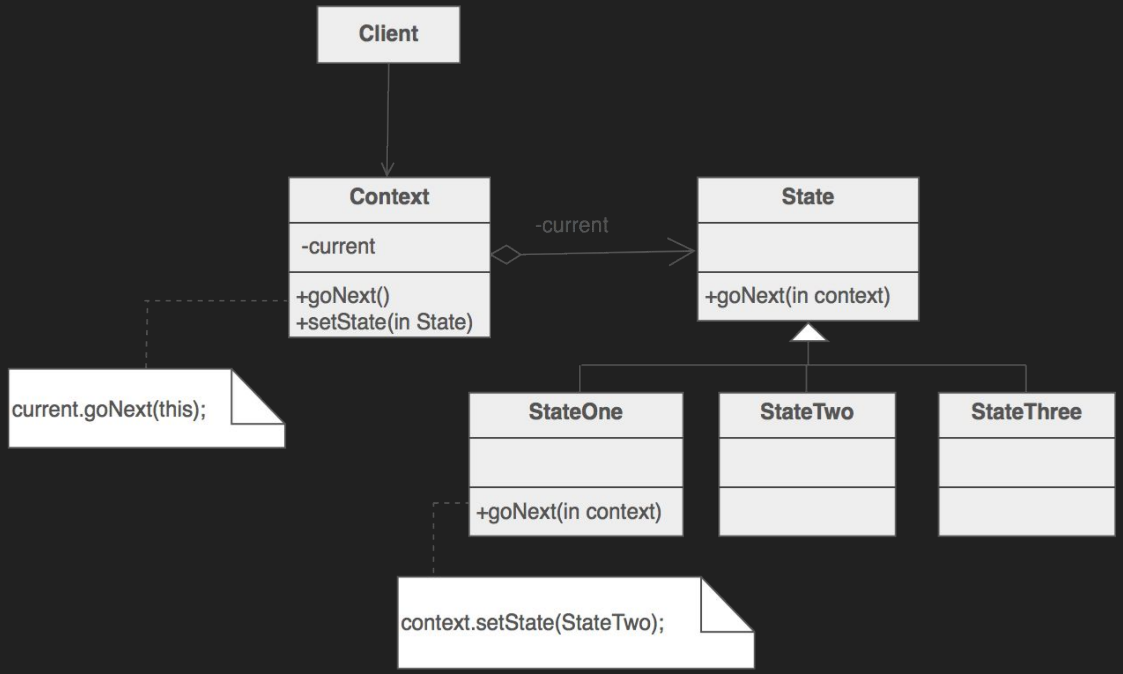
Beverage acts as our abstract component class.

The four concrete components, one per coffee type.

Condiment decorators; note which methods they need to implement – both cost() and getDescription().

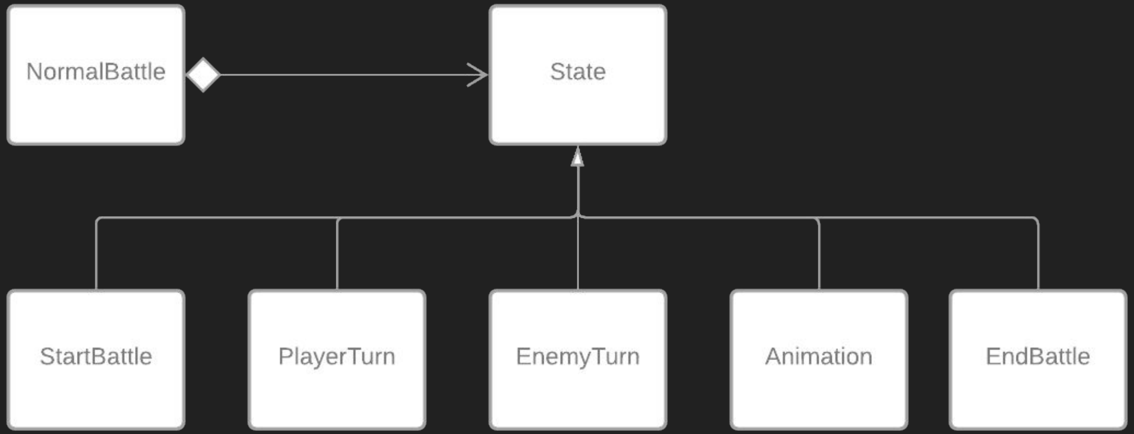
State

- Define an abstract base state class
- Represent different states in classes derived from the abstract state class that derive specific behavior



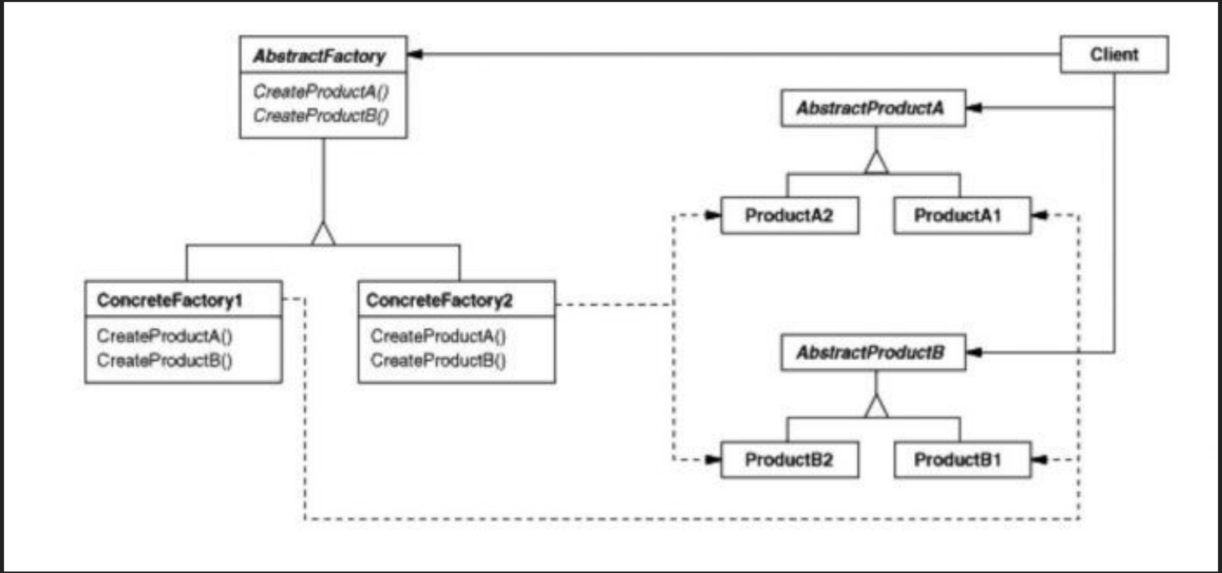
State

- Allows functionality to change along with the state
- Identical structure to bridge but solves different problems

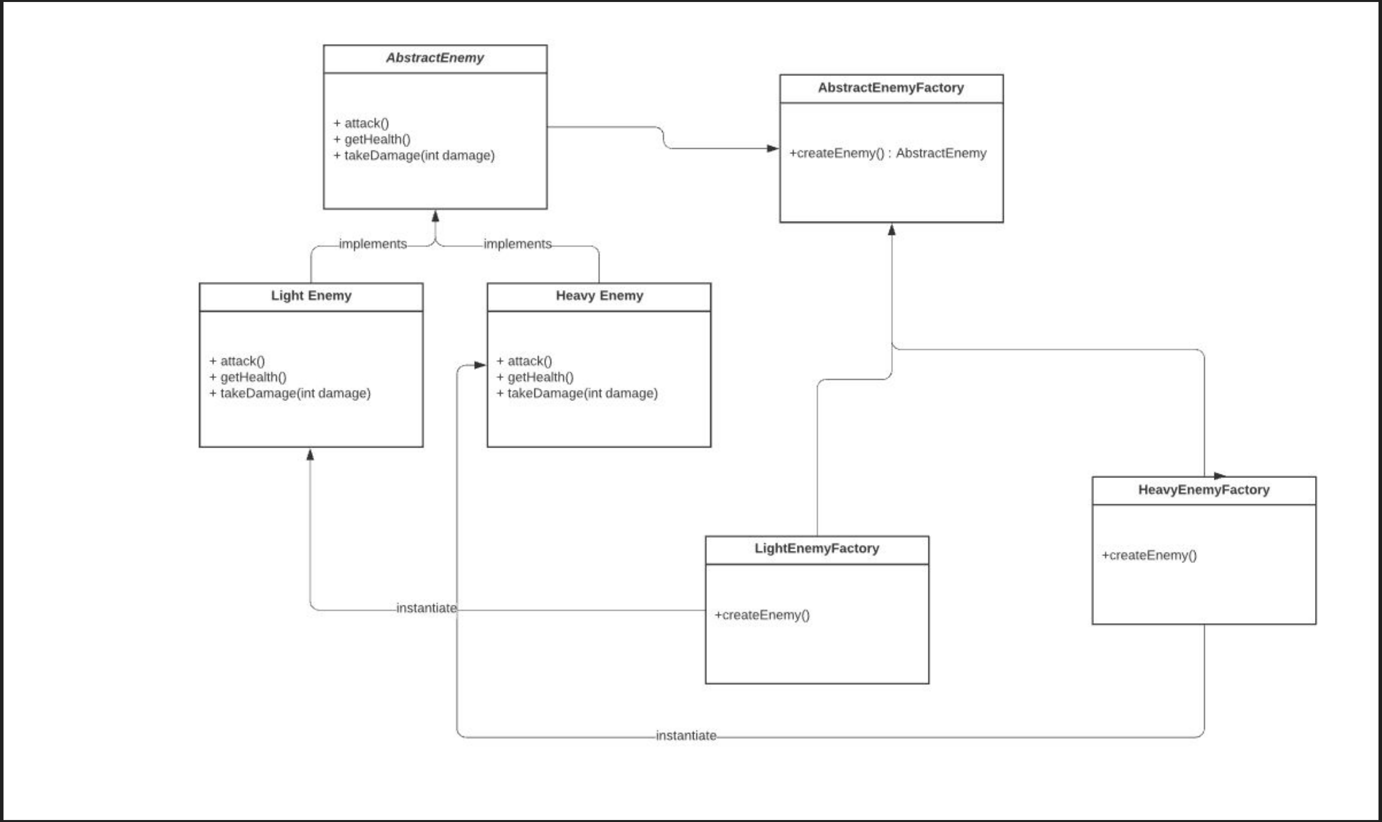


Abstract Factory

- Useful for creating objects of multiple types
- Object type is abstracted from user

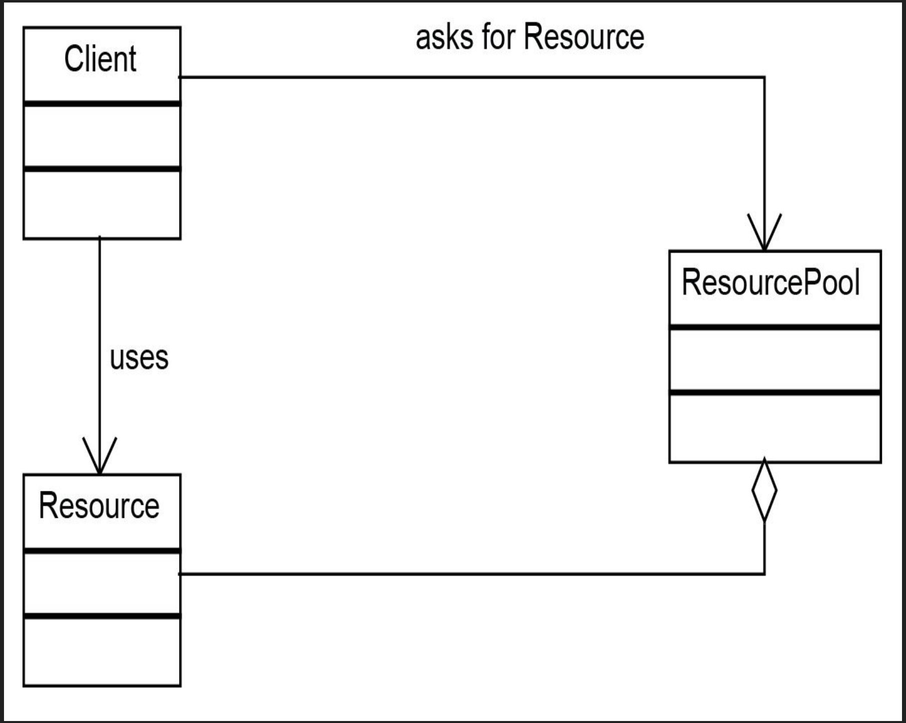


My Example of an Abstract Factory



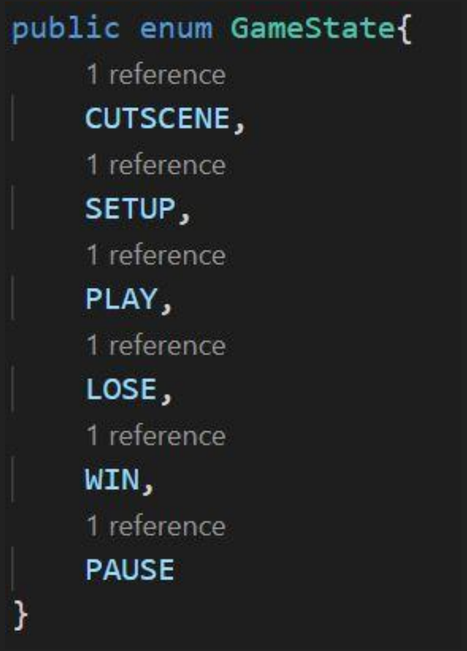
Example of Object Pool

- Used for recycling objects
- When to use:
  - Frequent object instantiation
  - Object’s usage is minimal
  - Object Instantiation is costly
  - Ex. Spawners

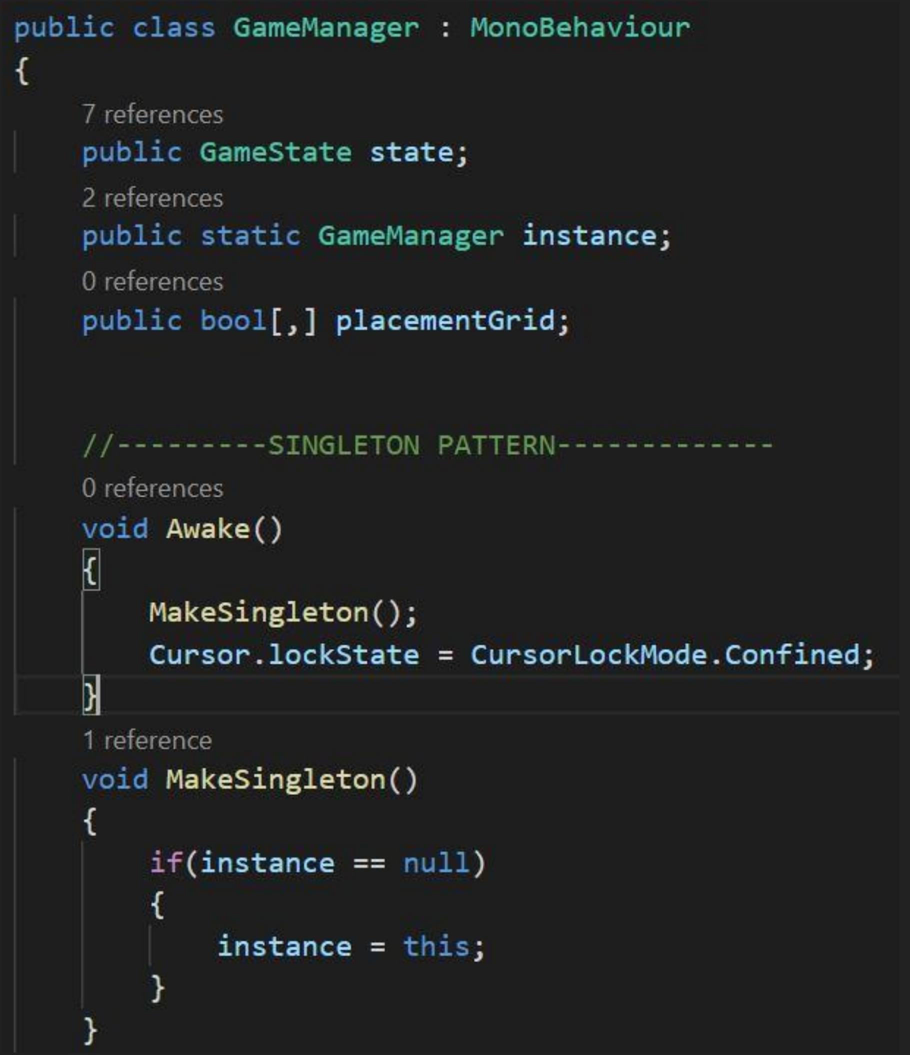


https://subscription.packtpub.com/book/application_development/9781786463593/2/ch02lvl1sec23/object-pool-pattern

Singleton: Game Manager



```
public enum GameState{
        1 reference
        CUTSCENE,
        1 reference
        SETUP,
        1 reference
        PLAY,
        1 reference
        LOSE,
        1 reference
        WIN,
        1 reference
        PAUSE
}
```



```
public class GameManager : MonoBehaviour
{
    7 references
    public GameState state;
    2 references
    public static GameManager instance;
    0 references
    public bool[,] placementGrid;


    //---------SINGLETON PATTERN------------
    0 references
    void Awake()
    {
        MakeSingleton();
        Cursor.lockState = CursorLockMode.Confined;
    }
    1 reference
    void MakeSingleton()
    {
        if(instance == null)
        {
            instance = this;
        }
    }
```

Resources

- https://sourcemaking.com/design_patterns
    - Also available under “useful links” on Dr. Bc’s website
- Design Patterns

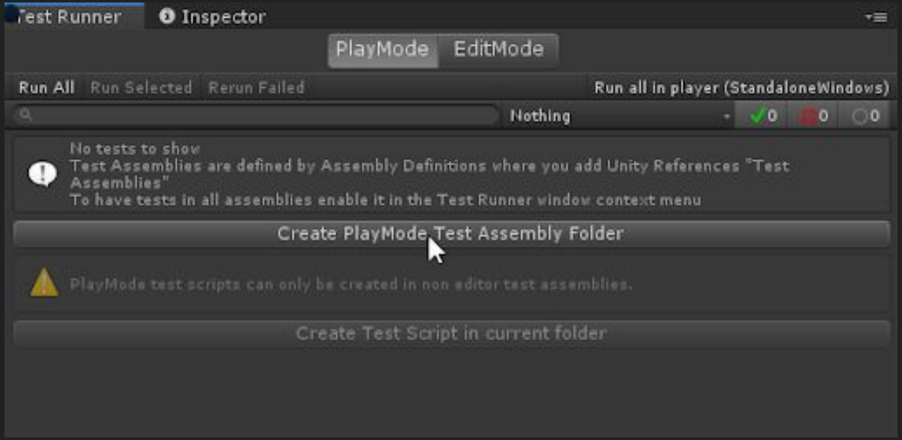
Edit Mode tests vs Play Mode tests

Edit Mode

- For any code that does not need the game to be ran to be tested
- Nothing should inherit from MonoBehaviour

Play Mode

- My preferred option
- Able to actually load a scene
- Can test anything
- More powerful


Test Runner
Inspector
PlayMode
EditMode
Run All
Run Selected
Rerun Failed
Run all in player (StandaloneWindows)
Nothing
0
0
0

No tests to show
Test Assemblies are defined by Assembly Definitions where you add Unity References "Test Assemblies"
To have tests in all assemblies enable it in the Test Runner window context menu

Create PlayMode Test Assembly Folder

PlayMode test scripts can only be created in non editor test assemblies.

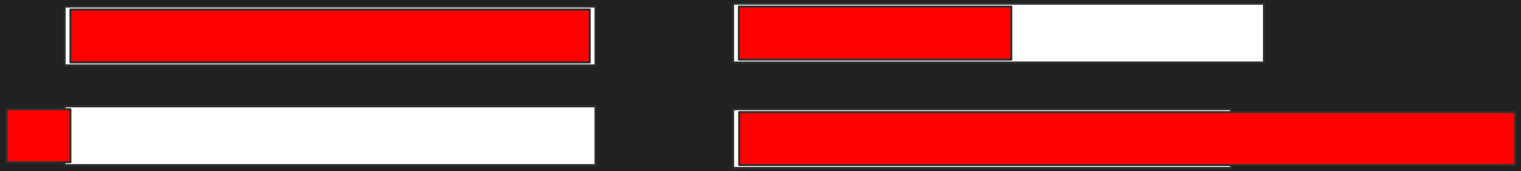
Create Test Script in current folder

What is a boundary test?

Any test where you are checking if a value is in a certain range

Example: Player health

Player health should be in [0,100]



Our How to doc

https://docs.google.com/document/d/1tHVF3wdHjWnRsRIUK07EPtFTcwzeqWEIfSI-SfFEzyI/edit?usp=sharing

DEMO

What is a Stress Test?

What is a Stress Test

- A stress test is any test applies stress incrementally until the code breaks

- Examples:
  - Object Instantiation
  - Velocity Expansion

- Failure Examples:
  - Collision Detection Failure
  - Frame Rate Reduction

https://www.deadlyapps.com/2011/09/unity-td-stress-test.htm

DEMO