Head First Design Pattern

Factory Pattern

The Abstract Factory Pattern provides an interface for creating families of related or dependency objects without specifying their concrete class. ★

The Factory Method Pattern defines an interface for creating an object, but lets subclasses decide which class to insantiate.

Factory Method lets a class defer instantiation to subclass.

When you directly insantiate an object, you are depending on its concrete clas.

Depend upon abstractions. Do not depend upon concrete classes. ★

Program to an interface, not an implementation.

It suggests that our high-level components should not depend on our low-level components; rather, they should both depend on abstractions.

For example, class PizzaStore generates various pizza classes. However, PizzaStore is high-level components and it create dozens pizza. It should use abstraction in creating object.

Dependency Inversion Principle

No variable should hold a refrence to a concerete class.

If you use new, you will be holding a reference to a concrete class. **Use a factory to get around that!**

No class should derive from a concrete class.

If you derive from a concrete class, you're depending on a concrete class. Derive from an abstraction, like an interface or an abstract class.

No method should override an **implemented** method of any its base classes.

If you override an implemented method, then your base class wasn't really an abstraction to start with.

All factories encapsulate object creation.

All factory patterns promote loose coupling by reducing the dependency of your application on concrete class.

Factory Method relies on inheritance: object creation is delegated to subclasses which implement the factory method to create objects.

The intent of Factory method is to allow a class to defer instantiation to its subclasses.

Abstract Factory relies on object composition. object creation is implemented in the methods exposed in the factory interface.

The intent of Abstract Factory is to create families of related objects without having to depend on their concrete class.

Compound Pattern

https://github.com/mrtkprc/design-pattern-notes/tree/master/CompoundPattern

Template Method Pattern

Lets assume that you will prepare tea and coffee. Related processes are like that.

Tea Recipe:

1-) Boil some water.

2-) Steep tea in boiling water.

3-) Pour tea in cup

4-) Add lemon

Coffee Recipe

1-) Boil some water.

2-) Brew coffee in boiling water.

3-) Pour coffe in cup

4-) Add sugar and milk

Above process (tead & coffee) contains code duplication.

Code duplicationes are extracted to abstract class. Different steps are renamed as more generic name.

The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps.

Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Template Method Pattern defines the skeleton of an algorithm in a method, deferring some steps to subclasses. ✏️

Sample Structure is like:

Abstract Class
templateMethod()
primitiveOperation1() = 0;
primitiveOperation2() =0;

Concrete Class
primitiveOperation1()
primitiveOperation2()

The Hollywood Principle: Don't call us, we'll call you. ★

Java have interface **Comparable**. This interface supply with method **compareTo(Object)**.

Factory Method is a specialization of Template Method.

Adapter and Facade Pattern

There are two adapter methodology.

Class Adapter

Class Adapter require multiple inheritance.

Object Adapter

The Adapter Pattern converts the interface of a class into another interface the clients expect. ✏️

A facade not only simplifies an interface, it decouples a client form a subsystem.

The Facade Pattern provides a unified interface to a set of interfaces in a subsytem. ✏️

Facade defines a higher level interface that makes the subsystem easier to use.

Principle of Least Knowledge - Talk only to your immediate friends. ★

return station.getThermometer().geTemprature(); =>

How many classes is this code coupled to?

This principle provides some guidelines:

Any componets of the object => Class Variables.

The object itself.

Objects passed in as a parameter to the method

Any object the method creates or instantiates.

A facade decouples a client from a complex subsystem.

Iterator and Composite Pattern

The Iterator Pattern provides a way to access the elements of an aggregation object sequentially without exposing its underlying representation. ✏️

There are two cooks. Theirs menu representations are different from each other. While one uses arraylist, another one use array.

If we've learned one thing in this book, it's encapsulate what varies.

Let's define an interface to manage this changes.

<<interface>>
Iterator
hasNext()
next()

A class should have only one reason to change. (Single responsibility)

If we intend to show menu in menu; therefore, you should use composite pattern.

The Composite Pattern allows you to compose into tree structures to represent part-whole hierarchies. ✏️

Sample Stucture is like that.

Component
operation()
add(Component)
remove(Component)
getChild(int)

Composite : Component
add(Component)
remove(Component)
getChild(int)
operation()

Leaf : Component
operation()

Strategy Pattern

It defines a family of algorithm encapsulates each one and makes them interchangeable. ✏️

Identify the aspects of your application that vary and seperate them from what stays same. ★

Take what varies and encapsulate it so it won't affect the rest of your code.

What is the one thing you can always count on in software development? => (Change)

HFDP, Page 22 shows related class diagram over Duck analogy. 🖍

Favor composition over inheritance. ★

Behaviours will live in a seperate class - a class that implements a particular behaviour interface.

Program to an interface, not an implementation. ★

Observer Pattern

Publisher(Subject) + Subscribers(Observer) = Observer Pattern

Observer Pattern provides an object design where subject and observers are loosely coupled.

The only thing the subject knows about an observer is that it implements a certain interface.

We can add new observers at any time.

We never need to modify the subject to add new types of observers.

We can reuse subjects or observers independently of each other.

Changes to either the subject or an observer will not affect the other.

When the state of one object changes, all of its dependents are notified.

It defines a one to many dependecy between objects so that when one object changes state, all of its dependents are notified and updated automatically. ✏️

Strive for loosely coupled designs between objects that interact ★

HFDP, Page 56 shows related class diagram over Weather Data analogy. 🖍

Decorator Pattern

Classes should be open for extensions but closed for modification. ★

Our goal is to allow classes to be easily extended to incorporate new behavior without modifying existing code.

Decorators have the same supertype as the objects they decorate.

The decorator adds it own behavior either before and/or after delegating to the object it decorates to do rest of the job.

Objects can be decorated at any time, so we can decorate objects dynamically at runtime with as many decorators as we like.

Decorator pattern, attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Base Class: **Component.**
ConcreteComponent is inherited from *Component*.
Decorator is inherited from *Component*.
ConcereteDecorator[A,B] is inherited from *Decorator*.

Decorator Pattern usually uses abstract classes.

You can wrap a component with any number of decorators.

In designs, we should allow behaviour to be extended without the need to modify existing code.

The Decorator Pattern involves a set of decorator classes that are used to wrap concrete components.

Decorators can result in many small objects in our design and overuse can be complex.

Proxy Pattern

Reason of usage of Proxy Pattern

Protection Proxy

Virtual Proxy

Remote Proxy

The Proxy Pattern provides a surrogate or placeholder object to control access to it. ✏️

Use proxy Pattern to create a representative object that controls access to another object, which may be remote, expensive to create or in need of securing.

Singleton Pattern

The Singleton Pattern ensures a class has only one instance, and provides a global point of access to it.

Command Pattern

The Command Pattern encapsulates an object thereby letting you parameterize other objects with different requests, queue or log requests, and support undoable operations. ✏️

Command(Interface) declares an interface for all commands. As you already know, a command is invoked through its execute() method, which asks a receiver to perform an action.

We apply Command pattern to many useful applications such as schedulers, thread pools, and job queues

Imagine a job queue: You add commands to the queue on one end.

On the other end sit a group threads.

Threads run the following the following script: they remove a command from the queue, call its execute() method, wait for the call to finish, then discard the command object and retrieve a new one.

Sample Implementation

ICommand contains method execute().

LightCommand inherits ICommand and take related object as parameter. In method execute(), required operation is performed by means of using parametirezed object.

LightCommand is passed to another class. For example, **RemoteControl**

State Pattern

State pattern employs logic of state machines.

For example, there is a gumball machine. The machine have four operation such as **insertQuarter(), ejectQuarter(), turnCrank(), dispense()**

<<interface>>
State
insertQuarter()
ejectQuarter()
turnCrank()
dispense()

NoQuarterState : State
... the four methods

SoldState : State
... the four methods

SoldOutState : State
... the four methods

If Required operations is performed on appropriate state, implementation pass the next state.

The State Pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class.