OOP Design Patterns

Introduction - Goals

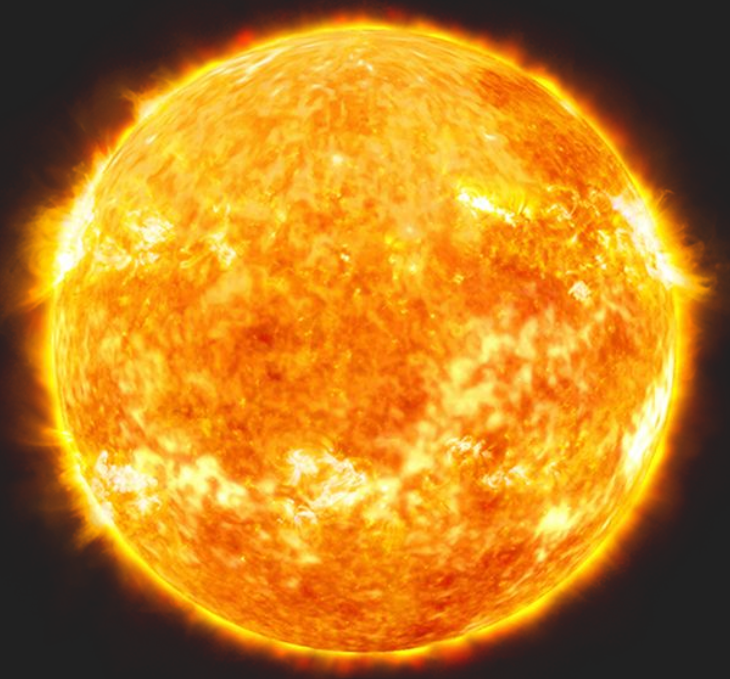
- As we spend much more time/eford for maintenance than implementation, there should be some techniques that help us simplify the maintenance process
- Avoid modifying existing code altogether
- Make the designs extendable in the right directions, with the right tools and the right amount.

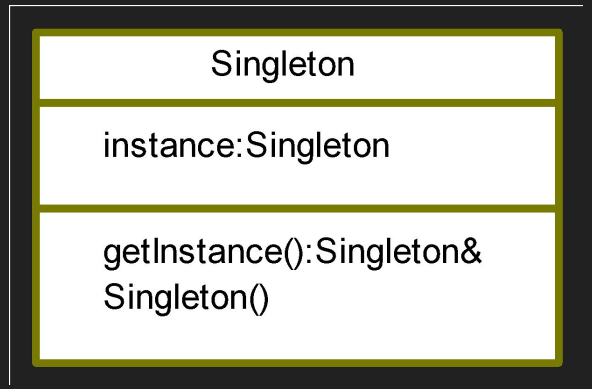
Polymorphism - Revision

- What is Polymorphism?
- Objects of different types (classes) provide single interface
- Using polymorphism we are able to send messages (call methods) to objects without knowing (specifying) their concrete class (type)
- Therefore we don’t depend on many concrete types

Singleton

The one and Only




Singleton
instance:Singleton
getInstance():Singleton&
Singleton()

Singleton - Advantages

- Cannot create more than one instance of a class when it does not make sense (when it models real-world entities)
- Makes use of lazy initialization where resources for the Singleton are only allocated the first time an instance is requested (like global variables)

Singleton - Disadvantages

- Singleton represents a global state that is spread wildly throughout the whole application
- Singletons tend to spoil object oriented design and principles. Undermines encapsulation
- Classes that use Singletons hide dependencies from the Singletons they use (their API is misleading)
- As a result of Singletons, the complexity of the system can easily increase out of control (implementation complexity, unit testing complexity, debugging complexity)

Singleton discussion

Misko Hevery, “Clean Code talks - Global State and Singletons”

https://www.youtube.com/watch?v=-FRm3VPhseI

Strategy

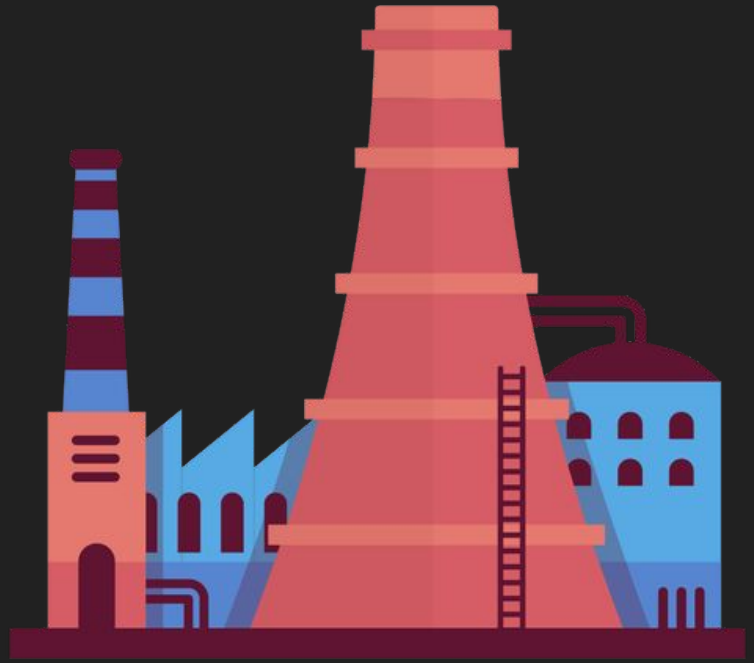
Change object behaviour

Strategy - Objectives

- Examples of changing object's behaviour at runtime
- Clients call the same method but get new behaviour - can be passed in constructor, or made use of *set/change* behaviour method
- Essence: context (dog) delegates its behaviour to Strategies (barking styles). It only assigns strategies, and never implements them on its own

Factory

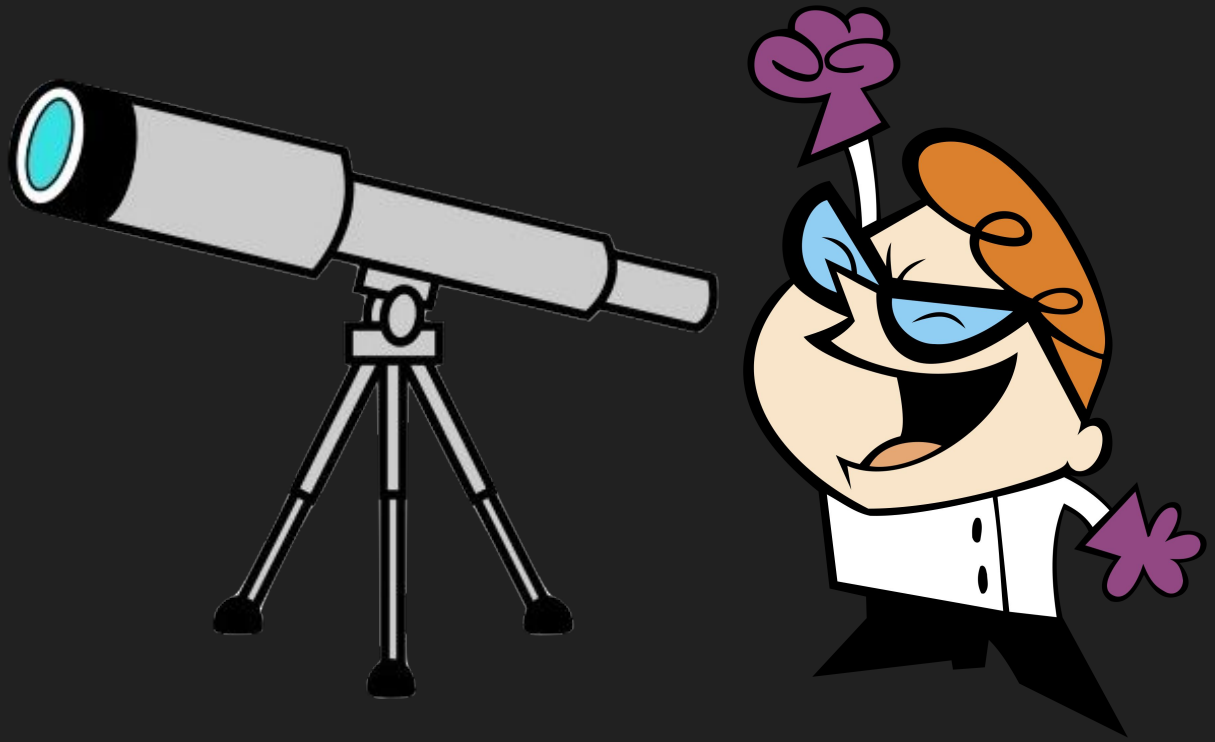
Create me an object please



Factory - Objectives

- Examples of object creation - Umm, each and every application
- Everyone knows how to create objects right?
- And everyone makes it everywhere in the program, which is a disaster, also violates the principle of least knowledge
- Factories encapsulate class instantiation
- We strive to extend without modification
- That’s why we don’t want to have object instantiations all over the application

Observer



Observer - Roles & Responsibilities

- Subject Interface - implemented by all classes that wish to become Subjects (to be observed via subscribe/register methods, i.e. DataElement)
- Observer Interface - implemented by all classes that wish to be Observers (to be able to subscribe for concrete Subjects, i.e. DisplayElement)
- Concrete Subjects - classes that act as subjects (support notify to all observers, i.e. WeatherStation)
- Concrete Observers - classes that need to observe concrete Subjects and receive notifications when needed.

Composite

Represent composition of objects as object

Composite - Objectives

- Sometimes we need to work with tree-like structures, where “leaf” node represents a simple (non-composite) items and “branch” nodes represents composition (group of items, that can also contain a “leaf” node or a “branch”)
- Examples of nested Object Compositions: Directories containing Files & nested Directories, etc.; Document with Sections with Content & nested Sections

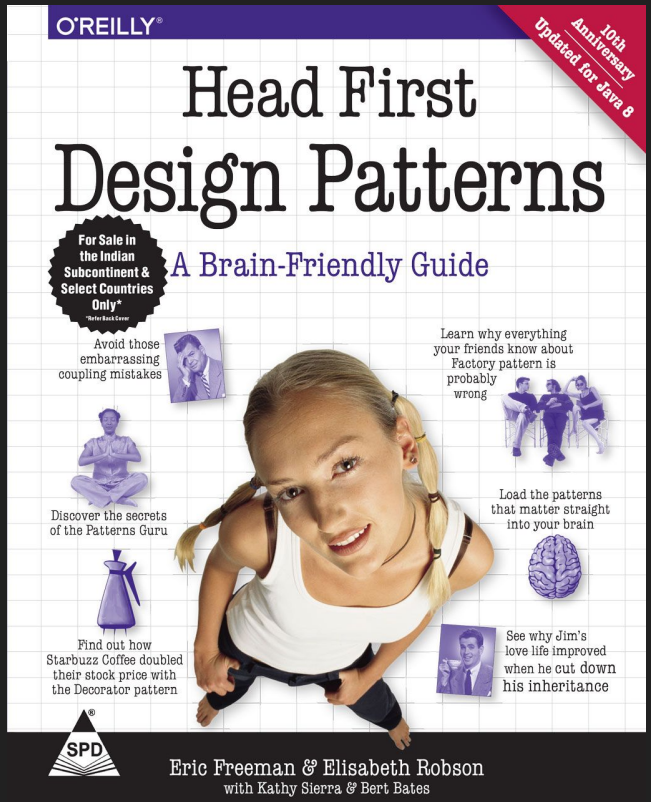
Composite - Responsibilities

- However, in reality, most of the operations (i.e. delete a file/folder) can be applied to both leafs and branches
- Therefore, it would be great if we could find a way to treat a group of items as a single item and avoid checks
- In software terms, we need a Composition (as a whole) to expose the same Interface as its Objects
- This way, we can perform the same operations on the Composition as on its individual items

And many, many others …

… State, Decorator, Adapter, Bridge, Command, Proxy, Template Method, Facade, Iterator … at ….

“Head First Design Patterns” - Eric Freeman



That’s all folks!