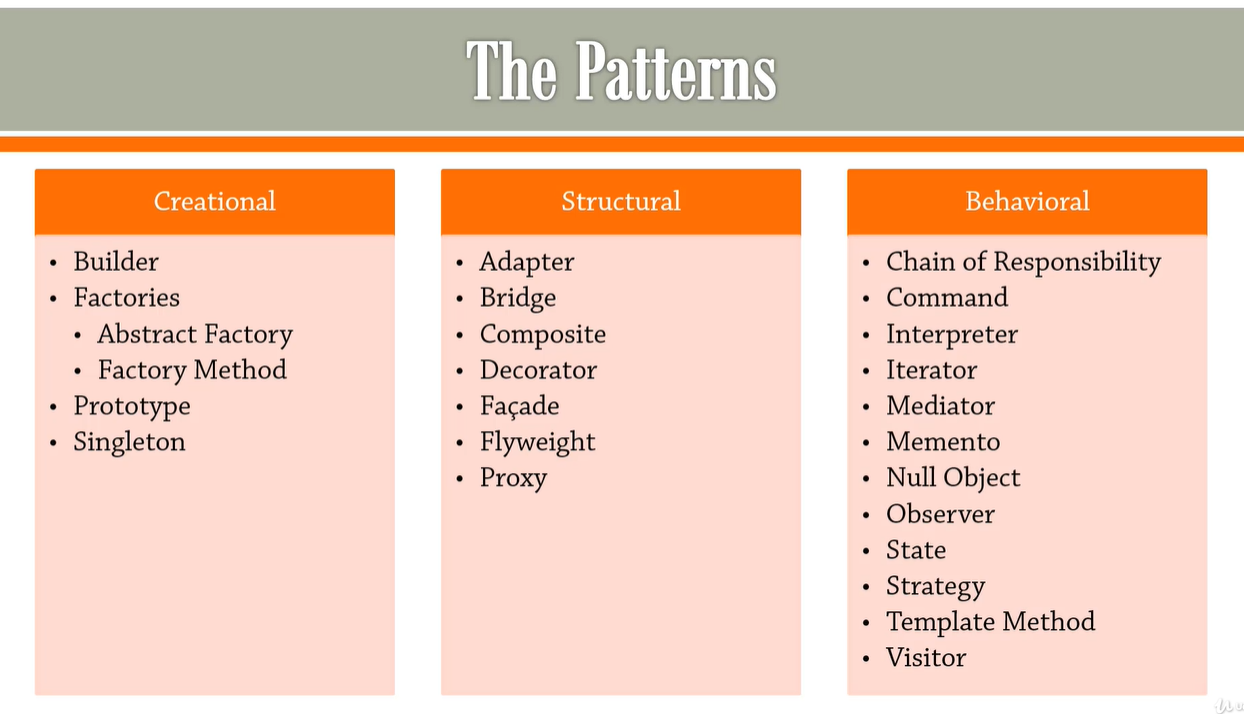
1. Gamma Categorization of Design Patterns
   1. Creational Patterns
      1. Deal with the creation of objects
      2. Explicit (Constructor) vs Implicit (DI, Reflection etc.)
      3. Wholesale (Single statement) vs piecewise (step-by-step)
   2. Structural Patterns
      1. Concerned with the structure (e.g., class members)
      2. Many patterns are wrappers that mimic the underlying class interface
      3. Stress the importance of good API Design
   3. Behavioral Patterns
      1. They are all different, no central theme



1. SOLID Design Principles
   1. Useful principles for object-oriented programming (OOPs)
   2. Single Responsibility Principle (SRP)
      1. SRP states is that a class should have just a single reason to change.
      2. A single class should have one primary responsibility instead of taking on lots and lots of different responsibilities.
   3. Open-Closed Principle (OCP) + Specification
      1. the main point of the open closed principle is classes should be open for extension but closed for modification.
      2. modifications of code that there has already been written tested and deployed is not good.
   4. Liskov Substitution Principle (LSP)
      1. idea of the risk of substitution principle is we should be able to substitute a base class for a subtype class.
      2. So, if we have some API, which takes a base class, we should be able to stick a subclass in there without the things breaking in any way.
   5. Interface Segregation Principle (ISP)
      1. the interface segregation principle is basically a recommendation on how to split interfaces into smaller interfaces.
      2. the interface segregation principle is that instead of sticking everything into

a single interface like we did for the machine interface, what you should do is you should put the absolute minimum amount of code into an interface

* + 1. YAGNI – You Aren’t Going to Need it
  1. Dependency Inversion Principle (DIP)
     1. High-level modules (which contains business logic) should not depend on low-level modules. Both should depend on abstractions.
     2. Abstractions should not depend on details. Details should depend on abstractions.

1. Creational Patterns
   1. Builder
      1. Some objects are simple and can be created in a single constructor call. Other objects require a lot of ceremony to create.
      2. Having an object with 10 constructor arguments is not productive
      3. **Builder** provides an API for constructing an object step-by-step
   2. Factories
      1. Constructor is not descriptive
         1. Name mandated by name of containing type
         2. Cannot overload with same sets of arguments with different names
      2. Wholesale object creation (non-piecewise, unlike Builder) can be outsourced to
         1. A separate function (**Factory Method**)
            1. Make constructor as private and create static factory method to create object and return.
         2. That may exist in a separate class (**Factory**)
            1. Create inner or separate class and add static method to create object.
         3. Can create hierarchy of factories with **Abstract Factory**
            1. create Abstract class or interface and implement in class which override method.
            2. Then create Factory class which return object.
   3. Prototype
      1. When it’s easier to copy an existing object to fully initialize a new one.
      2. An existing (partially or fully constructed) design is a Prototype.
      3. We make a copy (or clone) the prototype and customize it.
         1. Requires “deep copy” support
      4. We make the cloning convenient (e.g., via a Factory)
      5. Three different ways:
         1. Cloneable (Deep Copy)
         2. Copy Constructor
         3. Copy Through Serialization
   4. Singleton
      1. For some application it only makes sense to have one object in the whole application.
         1. E.g.,
            1. object which accesses Database
            2. factory object
      2. Constructor call is expensive
         1. We only do it once
         2. We provide same instance
      3. Want to prevent anyone creating additional copies
      4. Need to take care of lazy instantiation and thread safety
         1. For lazy instantiation – create object when object is null
         2. For thread safety – use doble locking check using synchronized block
      5. Problem with simple singleton
         1. Reflection
         2. Serialization
            1. readResolve method is used for \_replacing\_ the object read from the stream (to fix serialization issue)
2. Structural Patterns
   1. Adaptor
   2. Bridge
   3. Composite
   4. Decorator
   5. Facade
   6. Flyweight
   7. Proxy
3. Behavioral Patterns
   1. Chain of Responsibility
   2. Command
   3. Interpreter
   4. Iterator
   5. Mediator
   6. Memento
   7. Null Object
   8. Observer
   9. State
   10. Strategy
   11. Template Method
   12. Visitor