Creational Patterns

- two main methods for parameterizing a system:
  - subclass the classes that create objects
    - Factory Method and offshoots
    - requires continued subclassing which can cascade changes and becomes brittle
  - use object composition by defining an object that is responsible for creating a class
    - Abstract Factory, Builder, Prototype
- considerations:
  - proliferation of subclasses (worst: Factory Method -> Abstract Factory -> Builder -> Prototype)
  - Factory Method provides more customizability
  - Abstract Factory, Builder and Prototype are more flexible but more complicated than Factory Method

Abstract Factory

- intent:
  - provide an interface for creating families of related or dependent objects without specifying their concrete classes
- use when:
  - a system should be independent of how its products are created, composed and represented
  - a system should be configured with one of multiple families of products
  - a family of related product objects is designed to be used together and this constraint needs to be enforced
  - you want to provide a class library of products and you want to reveal just their interfaces not their implementations
- participants:
  - Abstract Factory
  - Concrete Factory
  - Abstract Product
  - Concrete Product
  - Client

- collaborations:
    - single Concrete Factory at runtime for classes of products
    - Abstract Factory defers object creation to Concrete Factory
- consequences:
    - isolates concrete classes
    - makes exchanging product families easy
    - promotes consistency among products
    - supporting new kinds of products is difficult
- implementation:
- related:
    - implemented using factory methods or prototypes
    - concrete factories can be singletons

Builder

- intent:
    - separate the construction of a complex object from its representation so that the same construction process can create different representations
- use when:
    - the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
    - the construction process must allow different representations for the object that's constructed
- participants:
    - Concrete Builder
    - Director (constructs object using the Builder interface)
    - Product
- collaborations:
    - client creates a Director and configures it with a Builder
    - Director notifies the Builder when a product component should be built
    - Builder composes product on request from Director
    - client retrieves product from the Builder

- consequences:
    - can vary a product's internal representation
    - produces code for construction and representation
    - allows more fine grained control over the construction process
- implementation:
- related:
    - Abstract Factory vs Builder:
        - Builder - step by step construction of an object at a time, returns to client on completion
        - Abstract Factory - constructs families of products and returns immediately
    - Builder often builds Composite

Factory Method

- intent:
    - define an interface for building an object but let subclasses decide which class to instantiate
    - allows a class to defer instantiation to subclasses
- use when:
    - a class can't anticipate the class of objects it must create
    - a class wants its subclasses to specify the objects it creates
    - classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate
- participants:
    - Product
    - Concrete Product
    - Creator (Application)
    - Concrete Creator (Application Instance)
- collaborations:
    - creator relies on subclasses to define the factory method so that it returns an instance of the appropriate Concrete Product
- consequences:
    - provides hooks for subclasses

    - connects parallel class hierarchies
- implementation:
- related:
    - Abstract Factory is often implemented using a Factory Method
    - usually called within Template Methods
    - Prototypes don't require subclassing Creator but often require an initialize operation (for Products). Creator uses Initialize to initialize the object but Factory Method does not require this.

Prototype

- intent:
    - specify the kinds of objects to create using a prototypical instance
    - create new objects by copying the prototype
- use when:
    - classes to instantiate are specified at run time (e.g. dynamic loading)
    - avoid building a class hierarchy of Factories to parallel a Product class hierarchy
    - when instances of a class can have one of a few combinations of state (create a handful of prototypes and clone them when necessary)
- participants:
    - Prototype
    - Concrete Prototype
    - Client
- consequences:
    - ability to add and remove classes at run time
    - specify new objects by varying values
    - specify new objects to copy structure
    - reduce subclassing
    - dynamically configure an application with classes
- implementation:
- related:
    - Prototype and Abstract Factory can be used in place of one another but an Abstract Factory can use Prototypes to create instances of Products

    - Composite and Decorator usage benefits from Prototypes

Singleton

- intent:
    - ensure only one instance of a class and provide global access point
- use when:
    - there must be one instance of a class with a well-known access point
    - when the single instance should be extensible by subclassing and clients should be able to use an extended instance without modifying their code
- participants:
    - Singleton
- collaborations:
- consequences:
    - controlled access to sole instance
    - reduced name space
    - permits refinements of operations and representation
    - permits a variable number of instances
    - more flexible than class operations
- implementation:
- related:
    - many patterns can be implemented using a Singleton:
        - Abstract Factory
        - Builder
        - Prototype

Structural Patterns

- contain many similarities across patterns, differences can sometimes be subtle
- Adapter vs. Bridge:
    - Adapter - resolves incompatibilities, generally after-the-fact
    - Bridge - provides stable interface to possibly differing implementations, generally as an initial design decision

- Facade is not an Adapter as it defines a new interface to unify multiple interfaces
- Composite vs. Decorator vs. Proxy:
    - Composite and Decorator - Decorator is not just a degenerate Composite
    - Decorator - adds responsibilities to objects without subclassing
    - Composite - provides structure to multiple objects
    - often used together
    - Proxy - like Decorator, composes an object to provide an identical interface to clients (i.e. does not change original interface)
    - not used for recursive composition
    - provides access control primarily, hiding various details like location and time of initialization
    - subject provides/defines functionality
    - Decorator remedies the situation where an object's total functionality cannot be determined at compile time (conveniently, i.e. without excessive subclassing)

Adapter

- intent:
    - convert the interface of a class into another interface
    - allows interoperability between classes that would be incompatible
- use when:
    - you want to use an existing class with an interface that does not match your needs
    - create a reusable class that cooperates with unrelated or unknown classes
    - can adapt the parent class for the interface of several subclasses
- participants:
    - Target
    - Client
    - Adaptee
    - Adapter
- collaborations:
    - Clients call operations on the Adapter which in turn calls operations on the Adaptee

- consequences:
    - Class vs Object Adapter (subclassing vs pointer referencing):
        - Class:
            - commits to a single concrete class
            - overrides some of Adaptee's behaviors (as a subclass of Adaptee)
            - introduces only one object with no additional pointer reference to Adaptee
        - Object:
            - single Adapter works with many Adaptees (Adaptee and subclasses)
            - can add functionality to Adaptee
            - more difficult to override behavior
    - exist on a spectrum of work done (and overridden/accessible behavior)
    - pluggable adapters (can subclass adapters and modify behavior as needed)
    - two-way adapters for transparency (multiple inheritance to remedy the potential lack of interface conformance caused by Adapters)
- implementation:
- related:
    - Bridge - similar but meant to allow two interfaces to vary independently (Adapter changes an interface)
    - Decorator - adds to an object but does not change the interface, also supports recursive composition
    - Proxy - provides a surrogate or another object but does not change interface

Bridge

- intent:
    - decouple an abstraction from its implementation so that the two can vary independently
- use when:
    - avoid permanent binding between an abstraction and its implementation
    - both abstraction and implementation should be extensible through subclassing
    - changes in implementation should not affect clients
    - hide implementation completely from clients (PIMPL)
    - proliferation of subclasses necessitates splitting an object into two parts
    - share implementation between multiple objects but hide this from clients
- participants:
    - Abstraction

    - Refined Abstraction
    - Implementor
    - Concrete Implementor
- collaborations:
    - Abstraction forwards requests to the Implementor object
- consequences:
    - decoupling interface and implementation
    - improved extensibility
    - hiding implementation details from clients
- implementation:
- related:
    - Abstract Factory can create and configure a Bridge
    - Adapters are generally created and applied after the fact whereas Bridges are created up front to allow abstractions and implementations to vary independently

Composite

- intent:
    - compose objects into tree structures to represent part-whole hierarchies
    - allows clients to treat individual objects and compositions of objects uniformly
- use when:
    - need to represent part-whole hierarchies
    - need to allow clients to ignore the difference between compositions of objects and individual objects
- participants:
    - Component
    - Leaf
    - Composite
    - Client
- collaborations:
    - Clients go through the Component to access a Composite, which either forwards directly to a Leaf or passes through a hierarchy to a Leaf

- consequences:
    - defines hierarchies consisting of primitive objects and composite objects
    - makes the client simple
    - makes it easier to add new kinds of components
    - can make the design too general
- implementation:
- related patterns:
    - component-parent link can be used for Chain of Responsiblity
    - Decorator and Composite are often used together, with Decorator supporting the Component interface
    - Flyweight allows sharing of components but they can no longer refer to parents
    - Iterator can be used to traverse composites
    - Visitor localizes operations and behavior that would otherwise be distributed across Composite and Leaf classes

Decorator

- intent:
    - attach responsibility to an object dynamically
    - flexible alternative to subclassing for extended functionality
- use when:
    - add responsibility to an object dynamically without affecting other objects
    - responsibilities of an object/class can be withdrawn
    - extension by subclassing is impractical (large number of independent extensions are possible or a class is hidden or not capable of being subclassed)
- participants:
    - Component
    - Concrete Component
    - Decorator
    - Concrete Decorator
- collaborations:
    - forwards requests to Component

- consequences:
    - more flexible than static inheritance
    - avoids feature laden classes high in a hierarchy
    - a decorator and its component are not identical
    - lots of little objects
- implementation:
- related:
    - Adapter - changes an interface whereas a Decorator changes an object's responsibilities
    - Composite - a Decorator is a Composite with only one Component
    - Strategy - Decorator changes the skin, Strategy changes the guts

Facade

- intent:
    - provide a unified interface to a set of interfaces in a subsystem
    - provides a higher level interface that makes a system easier to use
- use when:
    - want to provide a simple interface to a complex system/subsytem
    - want to decouple a client from a dependency-laden abstraction/implementation class network
    - want to provide a (simplified/single) entry point to a layered subsystem
- participants:
    - Facade
    - subsytem classes
- collaborations:
    - forwards requests from clients to various subsystem objects
    - clients do not need to access subsystem directly
- consequences:
    - hides subsystem from clients, simplifying client code/interactions
    - promotes weak coupling/decoupling
    - does not prevent an application from using subsystem classes if necessary

- implementation:
- related:
    - Facade can provide an interface to an Abstract Factory or an Abstract Factory can be used in place of a Facade
    - Mediator is similar but is used to coordinate various Colleagues without reducing awareness between Colleagues
    - Facades are often Singletons

Flyweight

- intent:
    - use sharing to support large numbers of objects efficiently
- use when:
    - an application uses a large number of objects
    - storage costs are high because of the sheer quantity of objects
    - most objects can use extrinsic state
    - many groups of objects may be replaced by relatively few shared objects after removing extrinsic state
    - application does not depend on object identity
- participants:
    - Flyweight
    - Concrete Flyweight
    - Unshared Concrete Flyweight (row, column)
    - Flyweight Factory
    - Client
- collaborations:
    - Flyweight state is characterized as extrinsic or intrinsic
        - intrinsic state is stored in the Concrete Flyweight
        - extrinsic state is calculated by clients and passed to Flyweight instances when their operations are invoked
    - clients should obtain Flyweights exclusively through a Flyweight Factory (which manages sharing)
- consequences:
    - may incur run time cost of generating extrinsic state

    - promotes storage savings by:
        - reduction in total number of instances
        - amount of intrinsic state per object
        - computation of extrinsic state vs. storage of extrinsic state
    - Flyweights are often combined with Composite to create a hierarchy
- implementation:
- related:
    - combined with Composite to create hierarchy (DAG with shared leaf nodes)
    - best to implement State and Strategy as Flyweights

Proxy

- intent:
    - provide a surrogate or placeholder for another object to control access to it
- use when:
    - remote proxy:
        - representative for an object in a different address space (i.e. machine, network)
    - virtual proxy:
        - creates expensive objects on demand
    - protection proxy:
        - controls access to an object (and manages access rights)
    - smart reference:
        - replaces bare pointer
        - reference counting
        - lazy loading, caching, memory management, etc
        - locking/access control
- participants:
    - Proxy
    - Subject
    - Real Subject
- consequences:
    - can hide location (address space) of object
    - can optimize with lazy creation/caching
    - allows additional housekeeping for object access
    - can use for copy-on-write

- implementation:
- related:
    - Adapter - provides a different interface whereas Proxy provides the same interface (NOTE: access control can result in a subset of the interface)
    - Decorator - similar but adds responsibilities whereas Proxies control access both vary in degree to their implementation

Behavioral Patterns

- Strategy encapsulates an algorithm
- State encapsulates state-dependent behavior
- Mediator encapsulates the protocol between objects
- Iterator encapsulates movement across an aggregate object
- all relate to aspects of a program that are likely to change
- many use objects as arguments:
    - Visitor
    - Comand with Context
    - Memento
- Mediator and Observer are competing patterns -> Observer distributes communication by introducing Subject and Observer, Mediator encapsulates communication between them
- Mediator centralizes communication distribution
- Observers are more flexible to reuse then Mediation
- Mediators are easier to understand the communication flow
- Decoupling senders and receivers:
    - Command, Observer, Mediator, and Chain of Responsibility
    - Command simplifies interface for generally one-to-one-to-one (invoker -> command -> receiver)
    - Observer can have multiple observers and subjects
    - Mediator sits between multiple Colleagues, generally routing requests
    - Chain of Responsibility sends it through many objects to decouple sender from multiple receivers
- behavioral design patterns complement and support each other

- most well-designed object oriented systems work are composed of multiple layered patterns
- it is often better to think at the pattern level than the class level when architecting an object oriented system

Chain of Responsibility

- intent:
    - avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
    - chain the receiving objects and pass the request along the chain until an object handles it (Cocoa first responder pattern)
- use when:
    - more than one object can handle a request and the handler isn't known a priori
    - want to send a request to multiple objects without specifying the receiver explicitly
    - the set of request handlers should/can be specified dynamically
- participants:
    - Handler
    - Concrete Handler
    - Client
- collaborations:
    - request is propagated along a chain to a Concrete Handler
- consequences:
    - reduced coupling
    - added flexibility in assigning responsibilities to objects
    - receipt isn't guaranteed
- implementation:
- related:
    - often used in conjunction with Composite

Command

- NOTE: the Command pattern is different than functors in C++

- intent:
    - encapsulate a request as an object, allowing clients parameterization with different requests, queueing or logging of requests, and support of undoable operations
- use when:
    - parameterize objects by an action to perform (object oriented replacement for callbacks)
    - specify, queue, and execute requests at different times
    - support undo
    - support logging changes so that they can be reapplied in case of fatal error
    - structure a system around high level operations built around primitive operations
- participants:
    - Command
    - Concrete Command
    - Client
    - Invoker
    - Receiver
- collaborations:
    - client creates a Concrete Command and specifies the receiver
    - Invoker stores the Concrete Command
    - Invoker executes and stores state for undoable commands
    - Concrete Command invokes operations on the receiver
- consequences:
    - decouples the invoker from the object that performs the operation
    - Command objects are first-class and can be manipulated and extended as such
    - Commands can be assembled into Composites
    - new Commands can be added easily
- implementation:
- related:
    - Composite can be used to implement MacroCommands
    - Memento can keep state required to undo a Command
    - copying Commands onto the history list uses the Command as a Prototype

Interpreter

- intent:
    - given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sequences in the language
- use when:
    - the grammar is simple (complex grammars are better served by parser generators or custom, full scale parsers)
    - efficiency is not a critical concern
- participants:
    - Abstract Expression
    - Terminal Expression
    - Nonterminal Expression
    - Context
    - Client
- collaborations:
    - client uses an abstract syntax tree of terminal and nonterminal expressions
    - uses the context to invoke the interpret operation
    - interpret operations use the context to store and access the state of interpreter
- consequences:
    - its easy to change and extend the grammar
    - implementing the grammar is easy
    - complex grammars are difficult to maintain
    - facilitates the ability to add new ways to interpret expressions
- related patterns:
    - Composite - AST is an instance of the Composite pattern
    - Flyweight - shows how to share terminal symbols in the AST
    - Iterator - interpreter uses an iterator to traverse the structure
    - Visitor - used to maintain behavior in each node in an AST

Iterator

- intent:
    - provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- use when:
    - there is a need to access an aggregate objects contents without exposing the internal representation
    - there is a need to support multiple traversals of aggregate objects
    - to provide a uniform interface to traverse different aggregate structures (polymorphic iteration)
- participants:
    - Iterator
    - Concrete Iterator
    - Aggregate
    - Concrete Aggregate
- collaborations:
    - Concrete Iterator keeps track of current element and can compute the next element
- consequences:
    - it supports variations in traversal of an aggregate
    - iterators simplify the aggregate interface
    - more than one traversal can be pending on an aggregate
- implementation:
- related patterns:
    - Composite - Iterators are often applied to recursive structures such as Composites
    - Factory Method - used to instantiate the proper polymorphic iterator instance
    - Memento - used to capture the state of an iteration, stored by Iterator internally

Mediator

- intent:
    - define an object that encapsulates how a set of objects interact

    - promotes loose coupling
    - allows independent interaction
- use when:
    - a set of objects communicate in a well defined but complex way
    - reuse of an object is difficult because it communicates with many other objects
    - a behavior thats distributed between several classes should be customizable without a lot of subclassing
- participants:
    - Mediator
    - Concrete Mediator
    - Colleague classes
        - each Colleague knows its Mediator object
        - each Colleague communicates with its Mediator in place of another Colleague
- collaborations:
    - colleagues send and receive requests from a Mediator object
    - the Mediator implements the cooperative behavior by routing requests between appropriate colleagues
- consequences:
    - it limits subclassing
    - it decouples colleagues
    - it simplifies object protocols
    - it abstracts how objects cooperate
    - it centralizes control
- implementation:
- related patterns:
    - Facade - unidirectional simplification of interaction with many objects
    - Colleagues can communicate with the Mediator using the Observer pattern

Memento

- intent:
    - capture and externalize and object's internal state so that the object can be restored to this state later without violating encapsulation

- use when:
    - a snapshot of some object's state must be saved so that it can be restore later
    - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation
- participants:
    - Memento
    - Originator
    - Caretaker
- consequences:
    - preserving encapsulation boundaries
    - it simplifies Originator (keeps versions of the internal state requested by clients)
    - using mementos might be expensive
    - defining narrow and wider interfaces
    - hidden costs in caring for mementos
- implementation:
- related patterns:
    - Comand - use Mementos to maintain state for undoable patterns
    - Iterator - can be used for Iteration as described earlier

Observer (aka Publish-Subscribe)

- intent:
    - define a one-to-many dependency between objects so that when one object changes state all dependents are notified and updated automatically
- use when:
    - an abstraction has two aspects, one dependent on the other (and they can vary independently)
    - when a change to one object requires changing others (and you don't know how many objects need to be changed)
    - when an object needs to be able to notify other objects without making assumptions about who those objects are (i.e. need loos coupling)
- participants:
    - Subject

    - Observer
    - Concrete Subject
    - Concrete Observer
- collaborations:
    - Concrete Subject notifies observer when pertinent changes occur
    - Concrete Observers may query the Subject for more info after a notification
- consequences:
    - abstract coupling between subject and observer
    - support for broadcast communication
    - unexpected updates
- implementation:
- related patterns:
    - Mediator - Change Manager acts as a Mediator between Subjects and Observers
    - Singleton - the Change Manager may be a Singleton

State

- intent:
    - allow an object to alter its behavior when its internal state changes
    - the object will appear to change its class
- use when:
    - an object's behavior depends on its state and it must change its behavior at run-time based on its state
    - objects have large multipart conditional statements that depend on the object's state (usually represented by one or more enumerated constants) -> puts each branch of the conditional in a separate class
- participants:
    - Context
    - State
    - Concrete State subclasses
- collaborations:
    - delegates state-specific requests to current Concrete State object
    - can pass Context to the State object handling the request

    - Context is the primary interface for clients and can be configured
    - the Context or Concrete State subclasses can decide which state succeeds another and under what circumstances
- consequences:
    - it localizes state specific behavior and partitions behavior for different states
    - it makes state transitions explicit
    - state objects can be shared
- implementation:
- related patterns:
    - Flyweight manages when and how State objects can be shared
    - State objects are often Singletons

Strategy

- intent:
    - define a family of algorithms, encapsulate each one, and make them interchangeable
    - allows algorithms to vary independently of the clients that use them
- use when:
    - many related classes need a way to be configured with one of many behaviors
    - you need different variants of an algorithm
    - an algorithm uses data that clients shouldn't know about
    - a class defines many behaviors, and these appear as multiple conditional statements in its operations
- participants:
    - Strategy
    - Concrete Strategy
    - Context
- collaborations:
    - Strategy and Context interact to implement the chosen algorithm
    - Context forwards requests from its clients to its strategy
- consequences:
    - families of related algorithms
    - an alternative to subclassing

    - strategies eliminate conditional statements
    - provide a choice of implementations
    - clients must be aware of different strategies
    - communication overhead between Strategy and Context
    - increased number of objects
- implementation:
- related patterns:
    - Flyweight - Strategy objects make good Flyweights

Template Method

- intent:
    - define the skeleton in an operation, deferring some steps to subclasses
    - allow subclasses to redefine certain steps in an algorithm without changing the algorithm's structure
- use when:
    - a class of algorithms share a common structure and subclasses should implement the variant part
    - a common class should be used to avoid code duplication
    - to control subclass extensions
- participants:
    - Abstract Class
    - Concrete Class
- collaborations:
    - Concrete Class relies on Abstract Class to implement the invariant steps of the algorithm
- consequences:
    - leads to inverted code structure
    - Template Methods call the following kinds of operations:
        - concrete operations
        - operations that are generally useful to subclasses
        - primitive operations (i.e. abstract method operations)
        - Factory Methods
        - hook operations with default behaviors for subclasses

- implementation:
- related patterns:
    - Factory Methods are often called by template methods
    - Strategy - Template Methods use inheritance to vary part of an algorithm whereas Strategies use delegation to vary the entire algorithm

Visitor

- intent:
    - represent an operation to be performed on the elements of an object structure
    - Visitor lets you define a new operation without changing the classes of the elements on which it operates
- use when:
    - an object structure contains many classes with differing interfaces
    - many distinct and unrelated operations need to be performed on objects in an object structure
    - the classes defining the object structure rarely change but new operations are often defined on the structure
- participants:
    - Visitor
    - Concrete Visitor
    - Element
    - Concrete Element
    - Object Structure
- collaborations:
    - a client that uses the Visitor pattern must create a concrete instance to traverse a structure
    - an object is passed as an argument to the Visitor and the operation that corresponds to that object's class is called
- consequences:
    - visitor makes adding new operations easy
    - a visitor gathers related operations and separates unrelated ones
    - adding new Concrete Element classes is hard
    - visiting can occur across class hierarchies

- implementation:
- related patterns:
    - Composite - Visitors can be used to apply an operation over an object structure defined by the Composite pattern
    - Interpreter - Visitor is applied to an AST to do the interpretation

Overview

creational

- abstract factory
- builder
- factory method
- prototype
- singleton

structural

- adapter
- bridge
- composite
- decorator
- facade
- flyweight
- proxy

behavioral

- chain of responsibility
- command
- interpreter
- iterator
- mediator
- memento
- observer
- state
- strategy

- template method
- visitor

(from Wikipedia - see https://en.wikipedia.org/wiki/Software_design_pattern for more)

creational

- abstract factory
  - provide an interface for creating families of related or dependent objects without specifying their concrete classes
- builder
  - separate the construction of a complex object from its representation, allowing the same construction process to create various representations
- factory method
  - define an interface for creating a single object, but let subclasses decide which class to instantiate
  - Factory Method lets a class defer instantiation to subclasses (dependency injection)
- prototype
  - specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum
- singleton
  - ensure a class has only one instance, and provide a global point of access to it

structural

- adapter
  - convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces
  - the enterprise integration pattern equivalent is the translator
- bridge
  - decouple an abstraction from its implementation allowing the two to vary independently
- composite
  - compose objects into tree structures to represent part-whole hierarchies
  - composite lets clients treat individual objects and compositions of objects uniformly
- decorator
  - attach additional responsibilities to an object dynamically keeping the same interface
  - decorators provide a flexible alternative to subclassing for extending functionality
- facade
  - provide a unified interface to a set of interfaces in a subsystem
  - facade defines a higher-level interface that makes the subsystem easier to use

- flyweight
  - use sharing to support large numbers of similar objects efficiently
- proxy
  - provide a surrogate or placeholder for another object to control access to it

behavioral

- chain of responsibility
  - avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
  - chain the receiving objects and pass the request along the chain until an object handles it
- command
  - encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests
  - also allows for the support of undoable operations
- interpreter
  - given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language
- iterator
  - provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- mediator
  - define an object that encapsulates how a set of objects interact
  - Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently
- memento
  - without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later
- observer
  - define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically (remember push vs. pull)
- state
  - allow an object to alter its behavior when its internal state changes
  - the object will appear to change its class
- strategy
  - define a family of algorithms, encapsulate each one, and make them interchangeable
  - strategy lets the algorithm vary independently from clients that use it
- template method
  - define the skeleton of an algorithm in an operation, deferring some steps to subclasses

    - template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure
- visitor
    - represent an operation to be performed on the elements of an object structure
    - visitor lets a new operation be defined without changing the classes of the elements on which it operates