

General

Criticism:

- target wrong problem
- lacks formal foundations
- leads to inefficient solutions
- does not differ significantly from other abstractions

Sources:

- https://sourcemaking.com/design_patterns
- https://en.wikibooks.org/wiki/Computer_Science_Design_Patterns
- https://en.wikipedia.org/wiki/Software_design_pattern

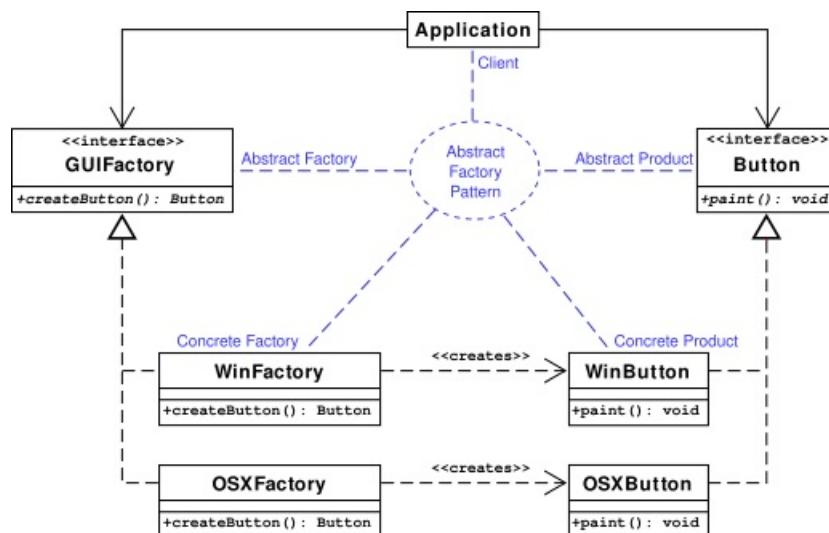
Creation

General:

- deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- These design patterns are all about class instantiation
- This pattern can be further divided into class
- creation patterns and object
- creational patterns
- While class
- creation patterns use inheritance effectively in the instantiation process, object
- creation patterns use delegation effectively to get the job done
- The creational patterns aim to separate a system from how its objects are created, composed, and represented
- They increase the system's flexibility in terms of the what, who, how, and when of object creation

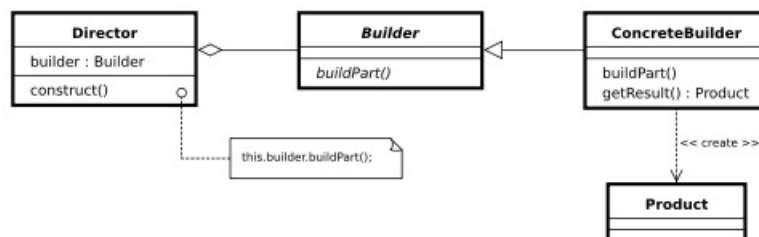
Abstract Factory:

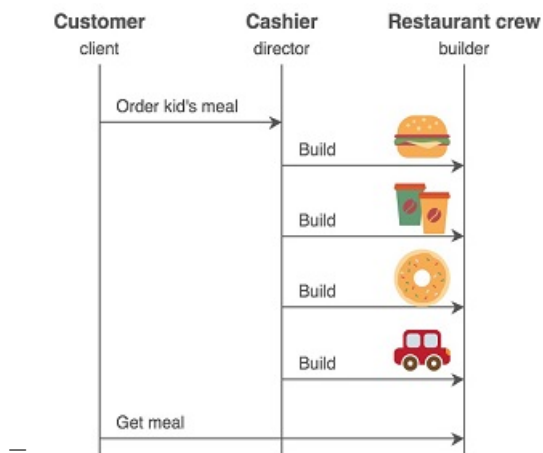
- Definition/Use
 - provides an interface for creating related or dependent objects without specifying the objects' concrete classes
 - provide an interface for creating families of related or dependent objects without specifying their concrete classes
 - encapsulates "new" ex: new product()
 - determines concrete type but returns abstract pointer
 - * client code has no knowledge and isn't burdened by concrete type
 - * adding new concrete types done by modifying client code to use different factory (1 line)
 - can determine concrete type from config file for example
- Structure



Builder:

- Definition/Use
 - separates the construction of a complex object from its representation so that the same construction process can create different representations
 - builder pattern is useful to avoid a huge list of constructors for a class
 - an application needs to create the elements of a complex aggregate
 - use builder to store parameters and then use that builder in constructor
 - separate the construction of a complex object from its representation so same construction can create different representations
- Structure

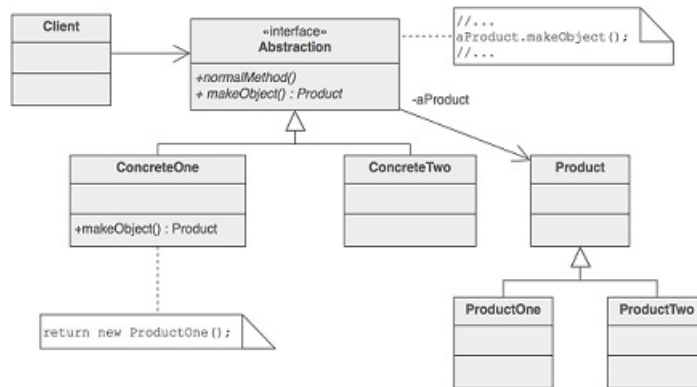




- Example
 - StringBuffer and StringBuilder
- Notes
 - put the builder term in the name of the builder class to indicate the use of the pattern to the other developers
 - if the target class contains flat data, your builder class can be constructed as a Composite that implements the Interpreter pattern

Factory Method:

- Definition/Use
 - allows a class to defer instantiation to subclasses
 - new operator considered harmful
 - define an interface for creating an object, but let subclasses decide which class to instantiate
 - provide a way for users to retrieve an instance with a known compile-time type, but whose runtime type may actually be different
 - an increasingly popular definition of factory method is: a static method of a class that returns an object of that class' type
 - * unlike a constructor, the actual object it returns might be an instance of a subclass
- Structure



- Example

- a factory method that is supposed to return an instance of the class Foo may return an instance of the class Foo, or an instance of the class Bar, so long as Bar inherits from Foo

```

Color.make_RGB_color(float red, float green, float blue)
Color.make_HSB_color(float hue, float saturation, float brightness)

```

```

Letter.getLetter(char) if vowel return Vowel(char) else Consonant(char)
(vowel, consonant extend letter)

```

- Notes

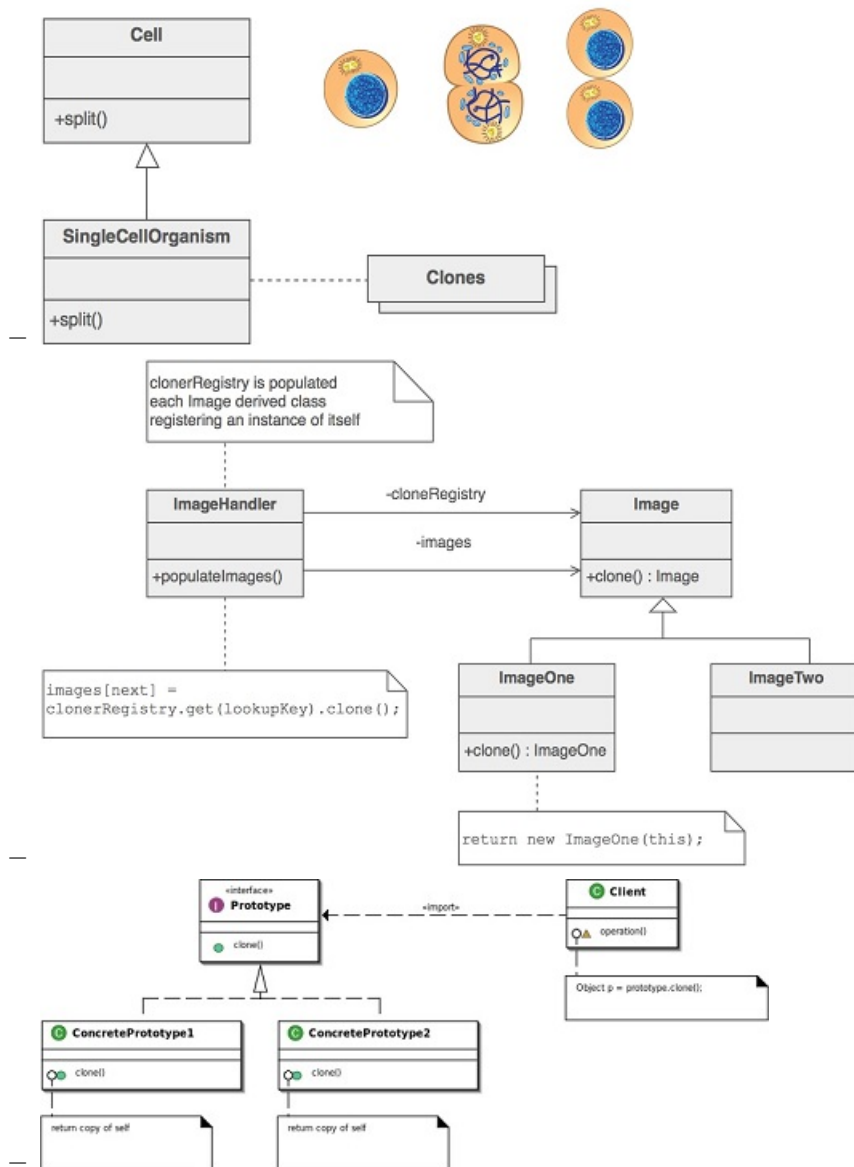
- consider making all constructors private or protected

Prototype:

- Definition/Use

- specifies the kind of object to create using a prototypical instance, and creates new objects by cloning this prototype
- when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects
- application "hard wires" the class of object to create in each "new" expression

- Structure



• Notes

- add `clone()` method
- add registry
- put the prototype term in the name of the prototype classes to indicate the use of the pattern to the other developers

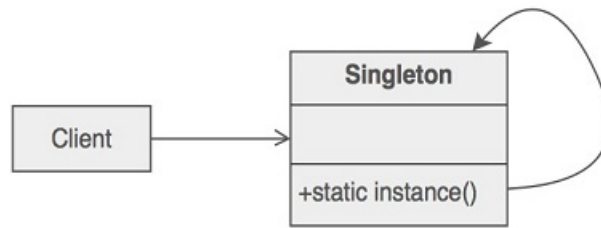
Singleton:

• Definition/Use

- ensures that a class only has one instance, and provides a global point of access to it

- ensure a class has only one instance, and provide a global point of access to it

- Structure



- Notes

- make instantiation(`private ThisSingleton()`) private aka define all constructors to be protected or private
- name the method `getInstance()` to indicate the use of the pattern to the other developers
- define a private static attribute in the "single instance" class

Comparison:

- abstract Factory classes are often implemented with Factory Methods, but they can be implemented using Prototype
- factory method: creation through inheritance, prototype: creation through delegation

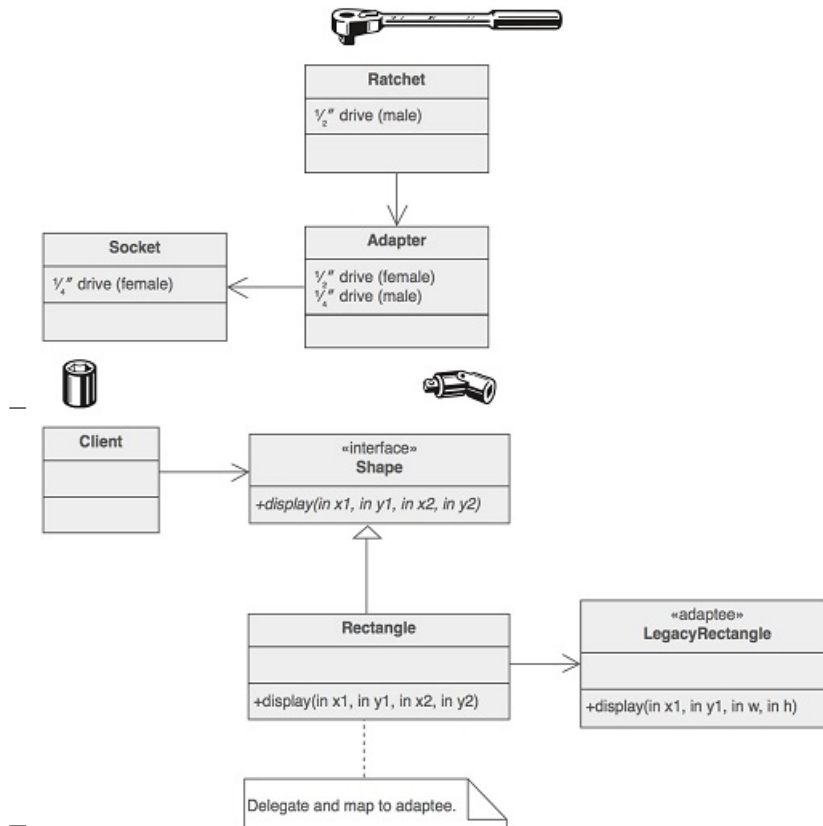
Structural

General:

- ease the design by identifying a simple way to realize relationships between entities
 - about class and object composition
 - use inheritance to compose interfaces
 - define ways to compose objects to obtain new functionality
-

Adapter:

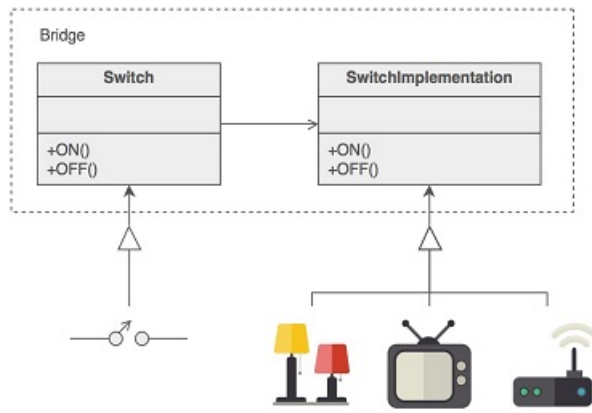
- Definition/Use
 - 'adapts' one interface for a class into one that a client expects
 - used when a client class has to call an incompatible provider class
 - an "off the shelf" component offers compelling functionality but its "view of the world" is not compatible
 - wrap an existing class with a new interface
- Structure



- Notes
 - put the adapter term in the name of the adapter class to indicate the use of the pattern to the other developers

Bridge:

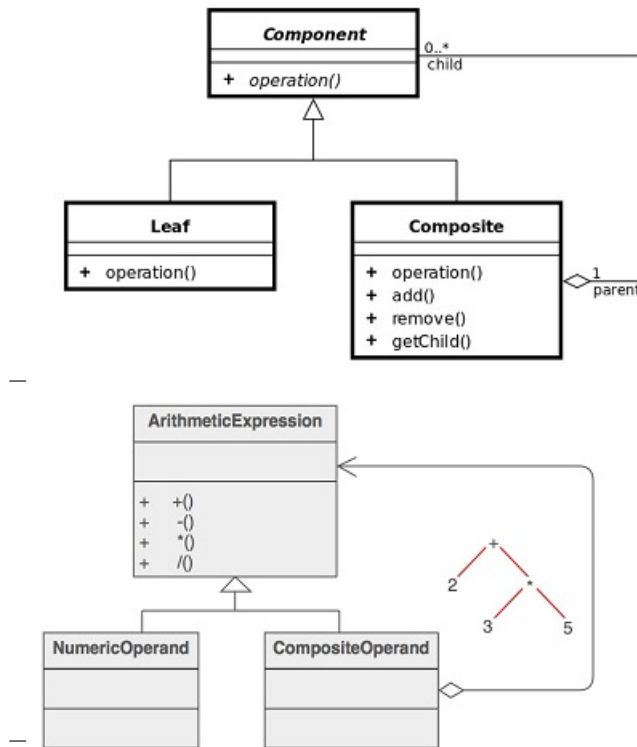
- Definition/Use
 - decouple an abstraction from its implementation so that the two can vary independently
 - useful when a code often changes for an implementation as well as for a use of code
 - decouple an abstraction from its implementation so that the two can vary independently
- Structure



- Notes
 - design the separation of concerns: what does the client want, and what do the platforms provide

Composite:

- Definition/Use
 - a tree structure of objects where every object has the same interface
 - application needs to manipulate a hierarchical collection of "primitive"(leaf) and "composite" objects
- Structure



- Examples

- GUI, widgets organized in a tree and operations (resize, repainting) on all widgets processed using pattern

- Notes

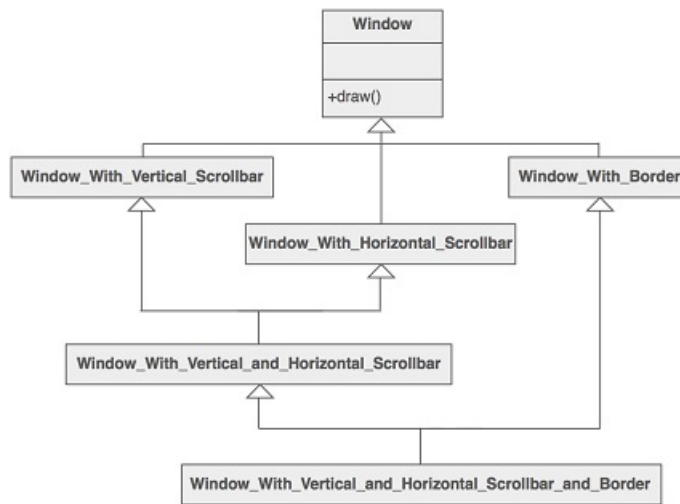
- consider the heuristic, "containers that contain containees, each of which could be a container"

Decorator (Wrapper):

- Definition/Use

- add additional functionality to a class at runtime where subclassing would result in an exponential rise of new classes
- client-specified embellishment of a core object by recursively wrapping it

- Structure



```
*
Widget* aWidget = new BorderDecorator(
    new HorizontalScrollBarDecorator(
        new VerticalScrollBarDecorator(
            new Window( 80, 24 ))));
aWidget->draw();
```

- Notes

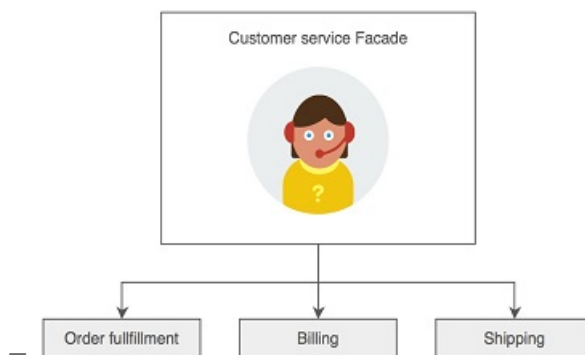
- ensure the context is: a single core (or non-optional) component, several optional embellishments or wrappers, and an interface that is common to all

Facade:

- Definition/Use

- create a simplified interface of an existing interface to ease usage for common tasks
- hides the complexities of the system and provides an interface to the client from where the client can access the system

- Structure



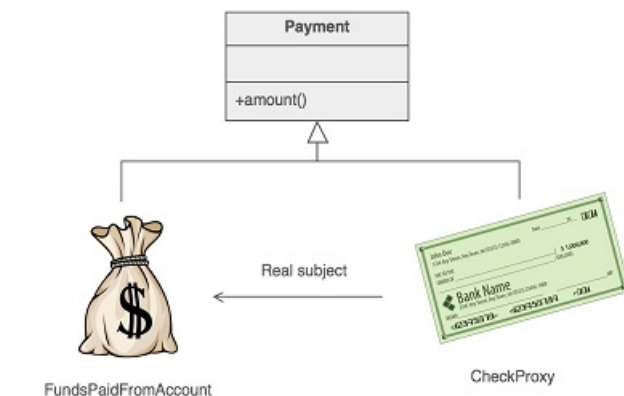
- Notes
 - often singletons because only one facade object is required
 - client uses (is coupled to) the facade only

Flyweight:

- Definition/Use
 - a large quantity of objects share a common properties object to save space
 - each "flyweight" object is divided into two pieces
 - * the state-dependent (extrinsic) part: stored or computed by client objects, and passed to the Flyweight when its operations are invoked
 - * the state-independent (intrinsic) part: stored (shared) in the Flyweight object
- Example
 - in video games, it is usual that you have to display the same sprite (i.e. an image of an item of the game) several times
 - * it would highly use the CPU and the memory if each sprite was a different object
 - * so the sprite is created once and then is rendered at different locations in the screen
 - * this problem can be solved using the flyweight pattern
 - * the object that renders the sprite is a flyweight

Proxy:

- Definition/Use
 - a class functioning as an interface to another thing
 - provide a surrogate or placeholder for another object to control access to it
- Structure



- Example
 - ProxyImage and RealImage

Comparison:

- adapter makes things work after they're designed, bridge makes them work before they are
- composite and decorator have similar structure diagrams, reflecting the fact that both rely on recursive composition to organize an open-ended number of objects
- decorator and proxy have different purposes but similar structures

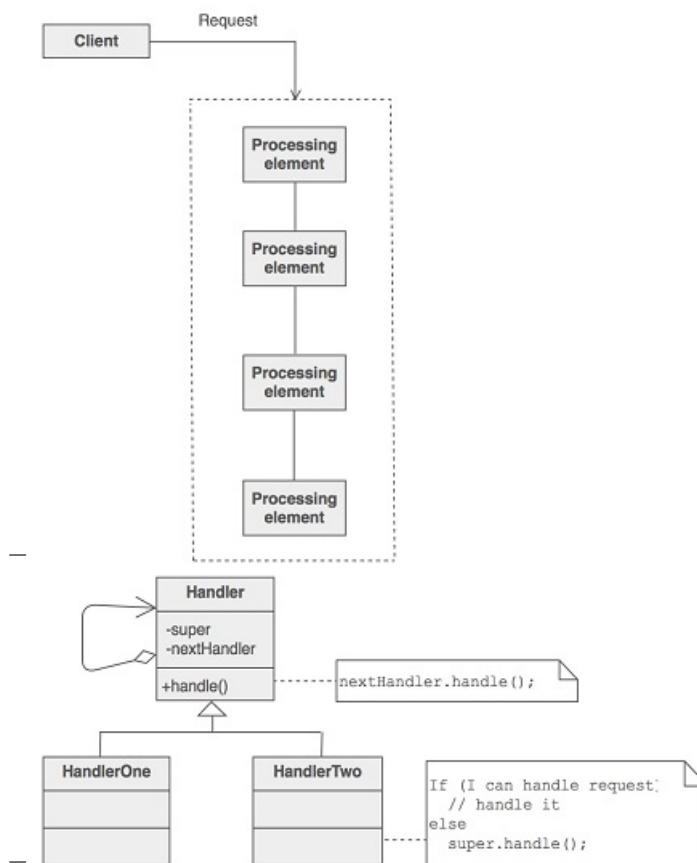
Behavioral

General:

Identify common communication patterns between objects and realize these patterns

Chain of responsibility:

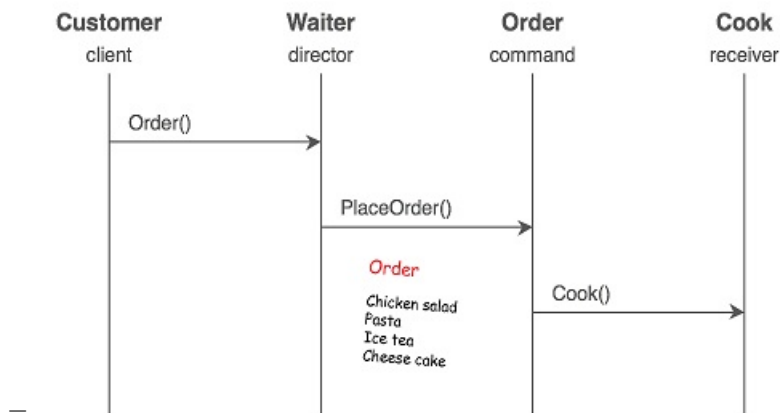
- Definition/Use
 - command objects are handled or passed on to other objects by logic-containing processing objects
 - avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request
- Structure



- Notes
 - base class maintains a next pointer
 - if the request needs to be "passed on", then the derived class "calls back" to the base class, which delegates to the "next" pointer

Command:

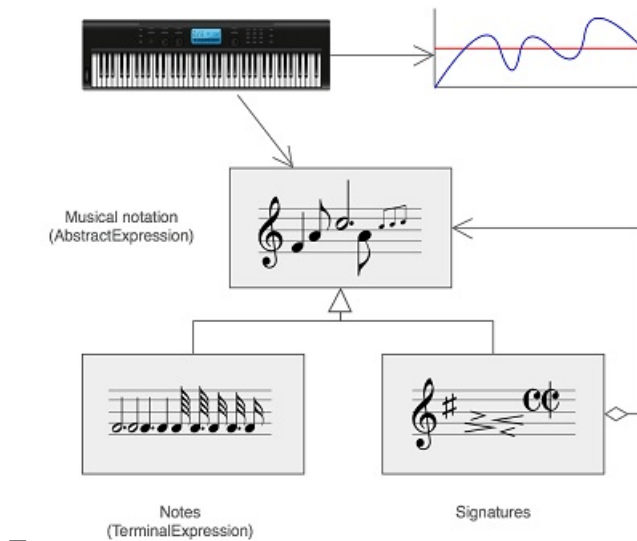
- Definition/Use
 - command objects encapsulate an action and its parameters
 - Need to issue requests to objects without knowing anything about the operation being requested or the receiver of the request
 - separation provides flexibility in the timing and sequencing of commands
 - command objects can be thought of as "tokens", created by one client that knows what need to be done, passed to another client that has the resources for doing it
- Structure



- Notes
 - define a Command interface with a method signature like `execute()`

Interpreter:

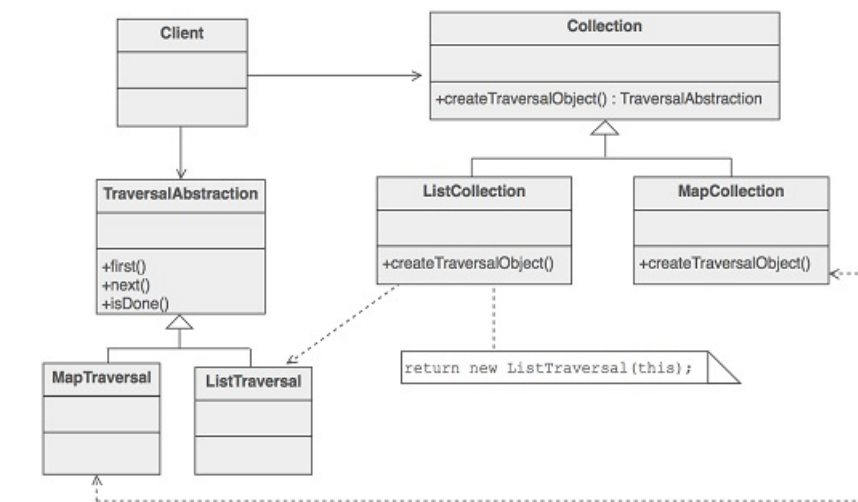
- Definition/Use
 - implement a specialized computer language to rapidly solve a specific set of problems
 - map a domain to a language, the language to a grammar, and the grammar to a hierarchical object
 - oriented design
- Structure



- Notes
 - the pattern doesn't address parsing. When the grammar is very complex, other techniques (such as a parser) are more appropriate

Iterator:

- Definition/Use
 - iterators are used to access the elements of an aggregate object sequentially without exposing its underlying representation
 - need to "abstract" the traversal of wildly different data structures so that algorithms can be defined that are capable of interfacing with each transparently
- Structure



- Notes

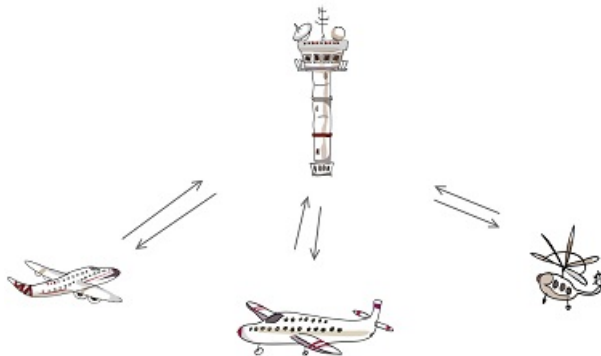
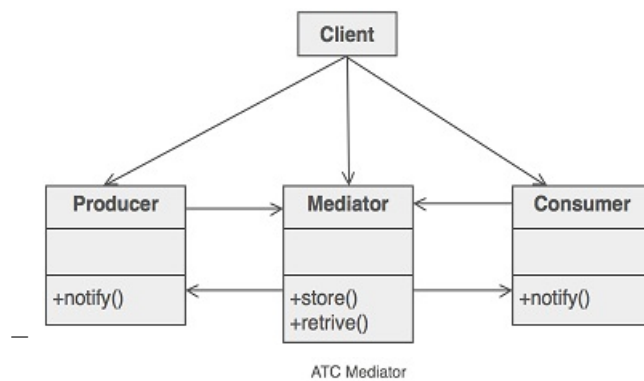
clients use the `first()`, `is_done()`, `next()`, and `current_item()` protocol to access the elements of the collection `class`

Mediator:

- Definition/Use

- provides a unified interface to a set of interfaces in a subsystem
- promotes loose coupling by keeping objects from referring to each other explicitly

- Structure



- Notes

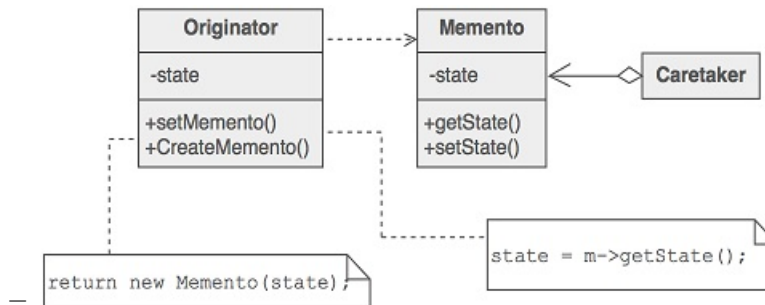
- be careful not to create a "controller" or "god" object

Memento:

- Definition/Use

- provides the ability to restore an object to its previous state (rollback)
- pattern defines three distinct roles
 - * originator : the object that knows how to save itself
 - * caretaker : the object that knows why and when the originator needs to save and restore itself
 - * memento : the lock box that is written and read by the originator, and shepherded by the caretaker

- Structure



- Notes

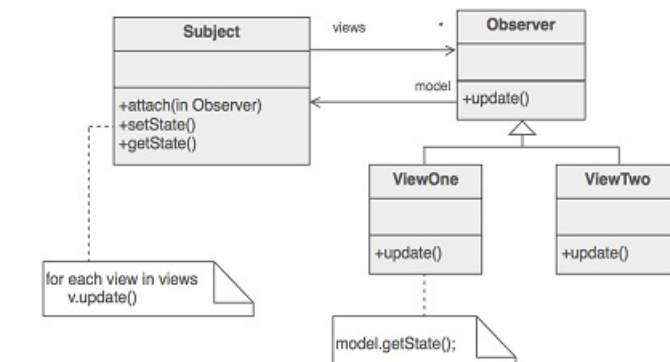
- identify the roles of caretaker and originator

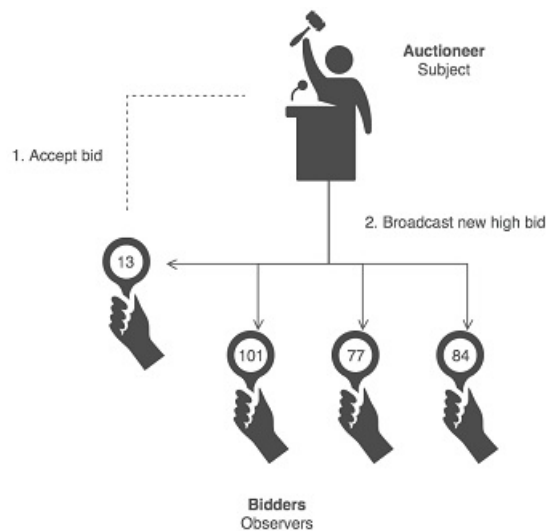
Observer(Publish/Subscribe or Event Listener):

- Definition/Use

- objects register to observe an event that may be raised by another object
- defines a one
- to
- many relationship so that when one object changes state, the others are notified and updated automatically

- Structure

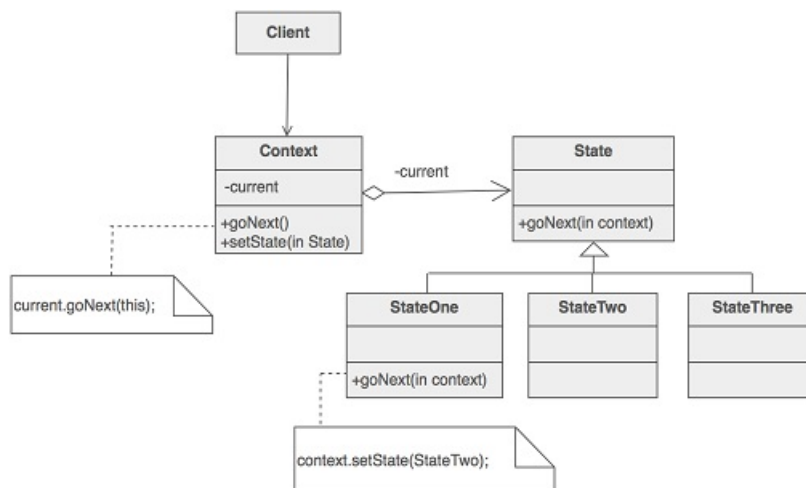


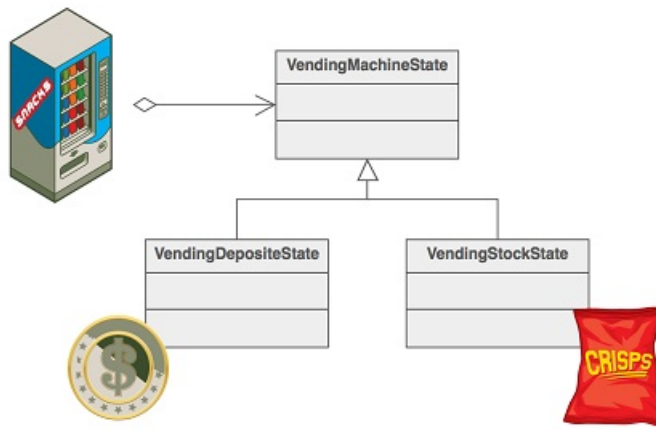


- Notes
 - subject broadcasts events to all registered observers

State:

- Definition/Use
 - a clean way for an object to partially change its type at runtime
 - a monolithic object's behavior is a function of its state
- Structure





- Notes

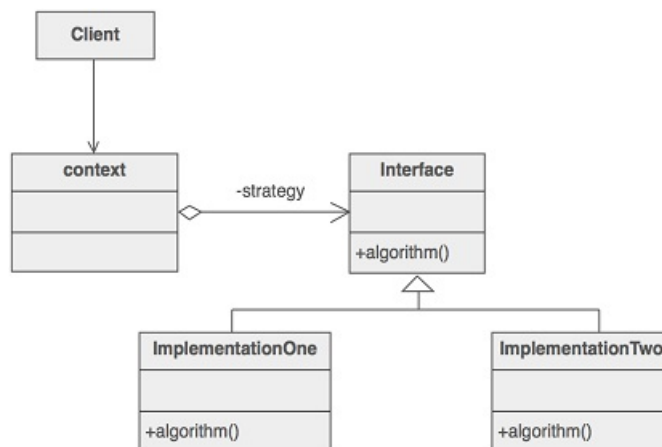
- pattern does not specify where the state transitions will be defined
 - * the "context" object
 - * each individual State derived class
 - advantage is ease of adding new State derived classes
 - disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses

Strategy:

- Definition/Use

- algorithms can be selected on the fly
- defines a set of algorithms that can be used interchangeably

- Structure



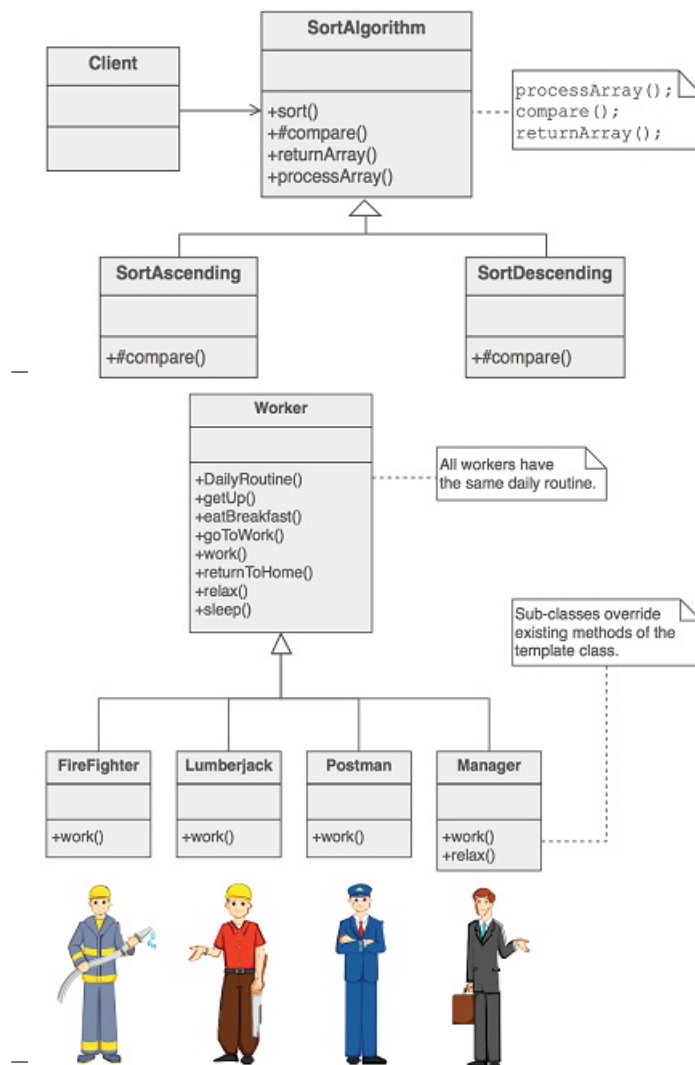
- Notes

- identify an algorithm (i.e. a behavior) that the client would prefer to access through a "flex point"

Template method:

- Definition/Use
 - describes the program skeleton of a program
 - component designer decides which steps of an algorithm are invariant (or standard), and which are variant (or customizable)

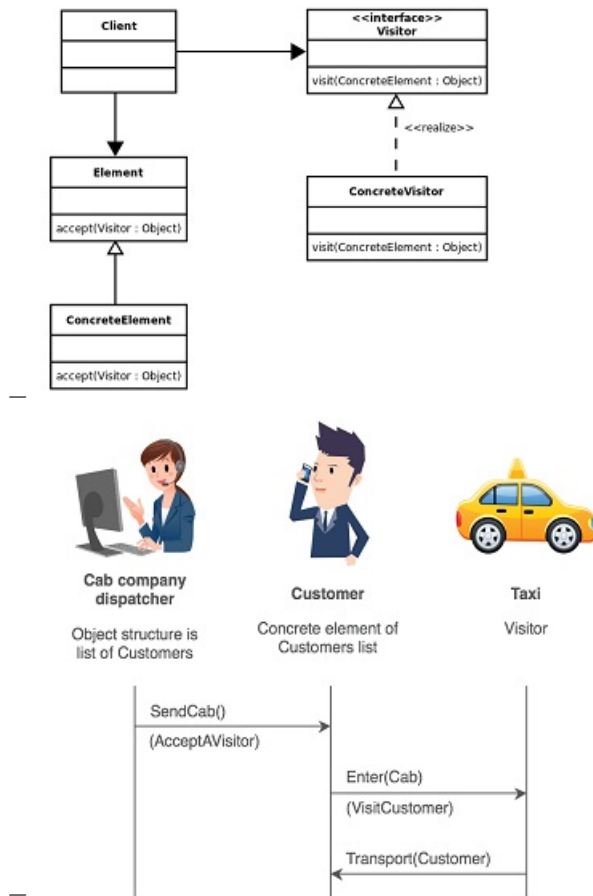
- Structure



- Notes
 - examine the algorithm, and decide which steps are standard and which steps are peculiar to each of the current classes

Visitor:

- Definition/Use
 - a way to separate an algorithm from an object
- Structure



- Example

```

public interface CharacterVisitor {

    public void visit(char aChar);

}

public class MyString {

    // ... other methods, fields

    // Our main implementation of the visitor pattern
    public void foreach(CharacterVisitor aVisitor) {
  
```

```
int length = this.length();
// Loop over all the characters in the string
for (int i = 0; i < length; i++) {
    // Get the current character, and let the visitor visit it.
    aVisitor.visit(this.getCharAt(i));
}
}

// ... other methods, fields

} // end class MyString

public class MyStringPrinter implements CharacterVisitor {

    // We have to implement this method because we're implementing the
    // CharacterVisitor
    // interface
    public void visit(char aChar) {
        // All we're going to do is print the current character to the standard
        // output
        System.out.print(aChar);
    }

    // This is the method you call when you want to print a string
    public void print(MyString aStr) {
        // we'll let the string determine how to get each character, and
        // we already defined what to do with each character in our
        // visit method.
        aStr.foreach(this);
    }

} // end class MyStringPrinter
```

- Notes

- if you have and will always have only one visitor, you'd rather implement the composite pattern

Comparison:

- chain of Responsibility, command, mediator, and observer, address how you can decouple senders and receivers, but with different trade
- offs
 - chain of Responsibility passes a sender request along a chain of potential receivers
- command and memento act as magic tokens to be passed around and invoked at a later time
 - in command, the token represents a request

- in memento, it represents the internal state of an object at a particular time
- polymorphism is important to command, but not to memento because its interface is so narrow that a memento can only be passed as a value