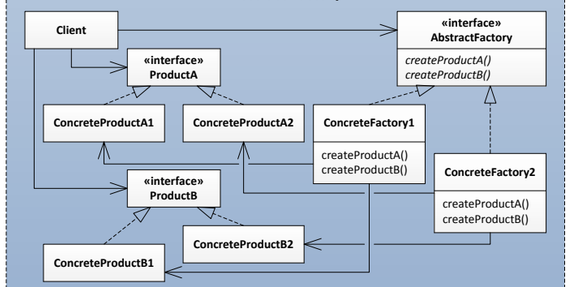
Design Patterns

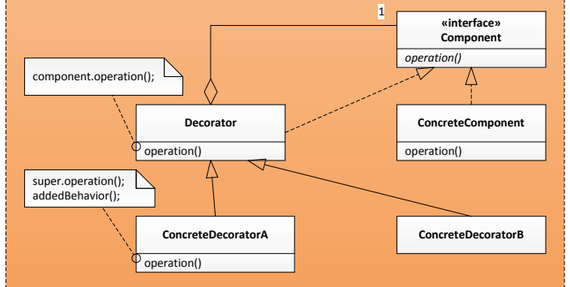
(according to *Flexible, Reliable Software – Using Patterns and Agile Development* by Henrik Bærbak Christensen)

Abstract Factory



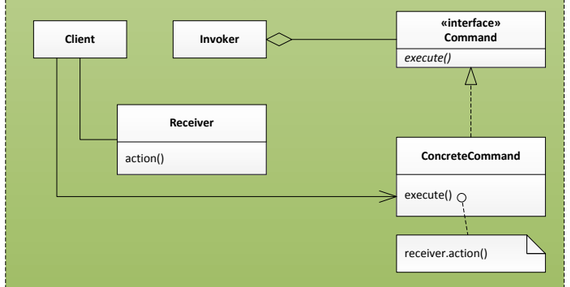
Problem: "Families of related objects need to be instantiated. Product variants need to be consistently configured."
Example of use: Simple configuration of HotGammon where each variant has its own factory

Decorator



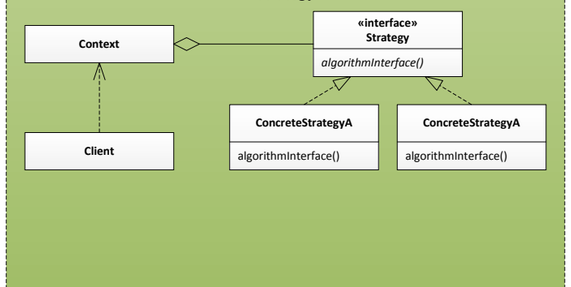
Problem: "You want to add responsibilities and behavior to an object without modifying its class."
Example of use: Add log file output to Paytation

Command



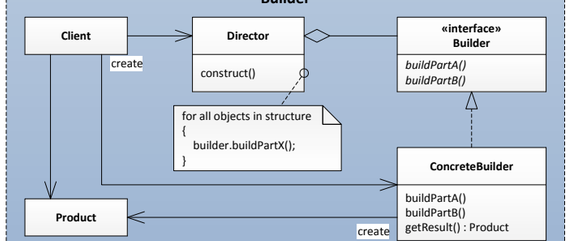
Problem: "You want to configure objects with behavior/actions at run-time and/or support undo"
Example of use: Shortcut and action key parameterization in a GUI

Strategy



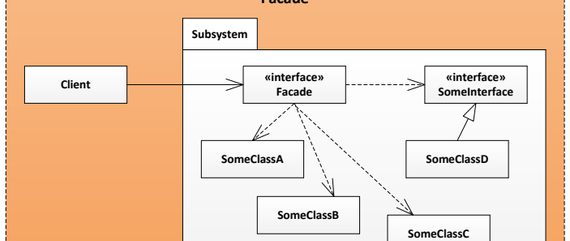
Problem: "Your product must support variable algorithms or business rules and you want a flexible and reliable way of controlling the variability"
Example of use: PayStation Linear and Progressive rate calculation strategies

Builder



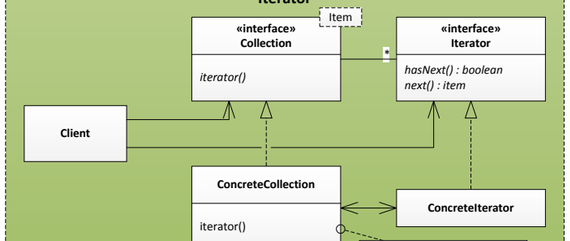
Problem: "You have a single defined construction process but the output format varies"
Example of use: Word processor output formats

Facade



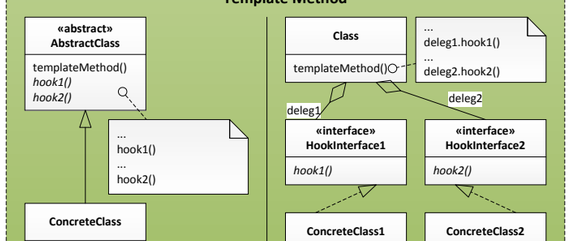
Problem: "The complexity of a subsystem should not be exposed to its clients."
Example of use: GUI for PayStation

Iterator



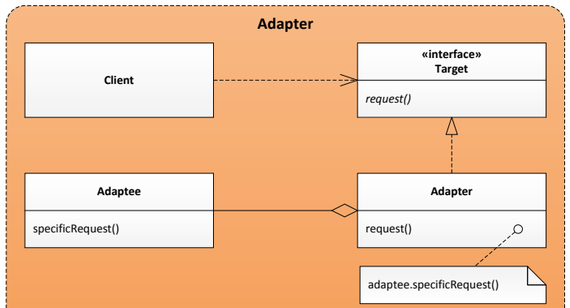
Problem: "You want to iterate a collection without worrying about the implementation details of it"
Example of use: Iteration over possible Knight moves in a game of Chess

Template Method



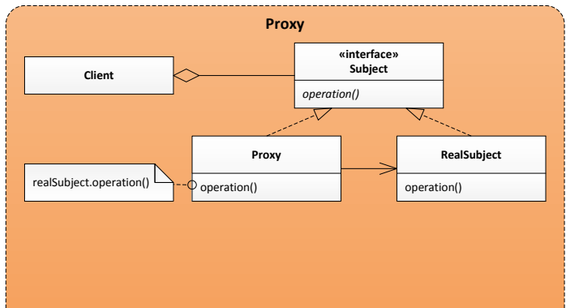
Problem: "There is a need to have different behaviors of some steps of an algorithm but the structure of the algorithm is otherwise fixed."
Example of use: Alter parts of the validate coin, add coin, calculate minutes paystation algorithm

Adapter



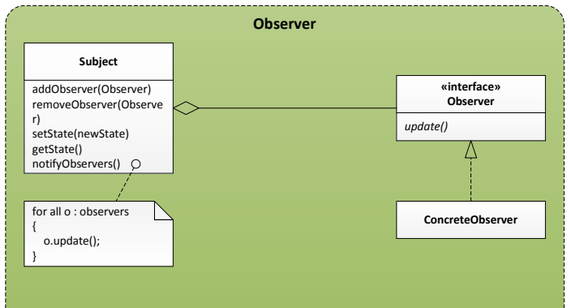
Problem: "You have a class with desirable functionality but its interface and/or protocol does not match that of the client needing it"
Example of use: Lunatown rate calculator integration

Proxy



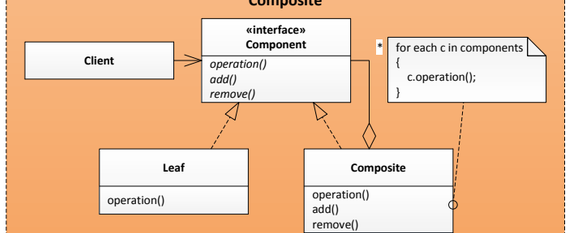
Problem: "A resource demanding object negatively affects the resource requirements of the client even if the object is not used at all; or new housekeeping when clients access the object is needed."
Example of use: Allow a word processor to load a document with large number of objects faster

Observer



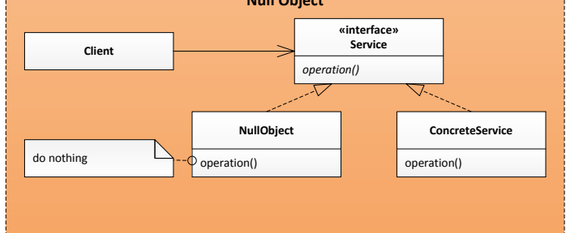
Problem: "A set of objects needs to be notified if a common object changes state to ensure system-wide consensus and consistency. You want to ensure this consistency in a loosely coupled way."
Example of use: Central computer monitoring multiple pay stations on a parking lot

Composite



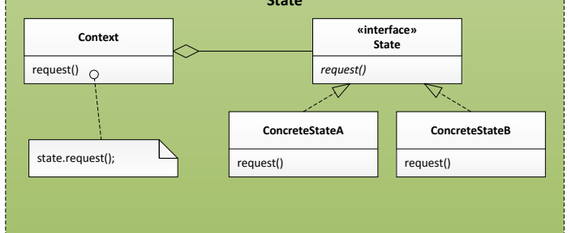
Problem: "Handling of tree data structures"
Example of use: Implementing Folder and File classes. Common methods collected in Component.

Null Object



Problem: "The absence of an object or behvaior, is often represented by a reference being null. However, this leads to numerous checks ensuring that no method is invoked on null."
Example of use: Disable progress indication during e.g. automatic testing

State



Problem: "Your products behaviour varies at run-time depending on some internal state."
Example of use: PayStation AlternatingRate strategy (Context object also implements State). State Machines in general.

Creational

Abstract Factory - *Provide an interface for creating families of related or dependent objects without specifying their concrete classes.* Page 217 [FRS]
Builder - *Separate the construction of a complex object from its representation allowing the same construction process to create various representations.* Page 301 [FRS]

Structural

Adapter - *Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.* Page 295 [FRS]
Composite - *Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.* Page 322 [FRS]
Decorator - *Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.* Page 289 [FRS]
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.* Page 282 [FRS]
Proxy - *Provide a surrogate or placeholder for another object to control access to it.* Page 317 [FRS]
Null Object - *Avoid null references by providing a default object.* Page 325 [FRS]

Behavioral

Command - *Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.* Page 308 [FRS]
Iterator - *Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.* Page 312 [FRS]
Observer - *Define a one-to-many dependency between objects where a state change in one object results with all its dependents being notified and updated automatically.* Page 335 [FRS]
State - *Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.* Page 185 [FRS]
Strategy - *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.* Page 130 [FRS]
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.* Page 366 [FRS]

Principles for Flexible Design

① Program to an interface. not an implementation

② Favor object composition over class inheritiance

③ Consider what should be variable in your design (encapsulate the behavior that varies)

UML

Parameter or local variable
Owns one or more instances
"has a" association specialization
extends interfaces
implements interfaces