A Design Pattern Tutorial

Design Patterns document recurring solutions to recurring problems in object-oriented software design. They are mined from successful object oriented design.

Three Categories of Design Patterns have been found to recur:

- **Creational Patterns**: Are concerned with object creation (how to organize code that creates objects)
- **Structural Patterns**: Deal with composition of classes and objects (how to organize objects)
- **Behavioral Patterns**: Characterize ways in which classes and objects interact and distribute responsibility (how to organize code)

Creational Patterns	**Structural Patterns**	**Behavioral Patterns**
• Abstract Factory • Builder • Factory Method • Prototype • **Singleton**	• Adapter • Bridge • **Composite** • Decorator • Facade • Flyweight • Proxy	• Chain of Responsibility • Command • Interpreter • Iterator • Mediator • Memento • **Observer** • State • **Strategy** • Template Method • Visitor

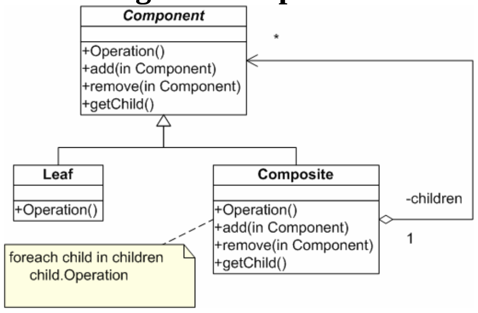
The next few pages will describe and refresh your knowledge of four patterns relevant to this study. They tell you the intent, purpose, participants, give an analogy and a mnemonic to each pattern.

- The intent tells you what problem the pattern is trying to address.
- The purpose is why we do it.
- Participants tell you classes and their roles
- Analogy to a something known
- Mnemonic is a short description of the pattern to remember it easily.

Description of the Composite Pattern

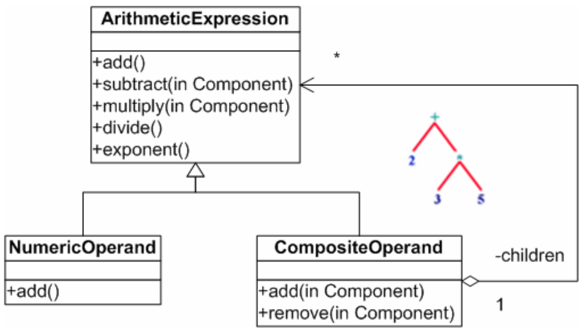
Category	Structural Pattern
Intent	• Enable to assemble complex objects out of primitive objects. • Compose objects into tree structures to represent part-whole hierarchies.
Purpose	Recursive composition by coupling the aggregate class to a common abstraction
Participants / Roles	Component, Composite, Leaf
Analogy	Arithmetic expression
Mnemonic	Recursive composition

UML Diagram Template



- Child management methods should normally be defined in the Composite class.

An Example Using the Composite Pattern

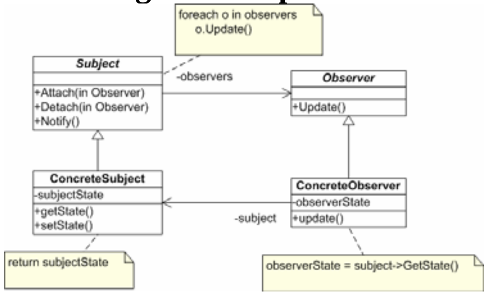


- The role of ArithmeticExpression is that of Component.
- The role of CompositeOperand is that of Composite
- The role of NumericOperand is that of a Leaf (since it has no children)

Description of the Observer Pattern

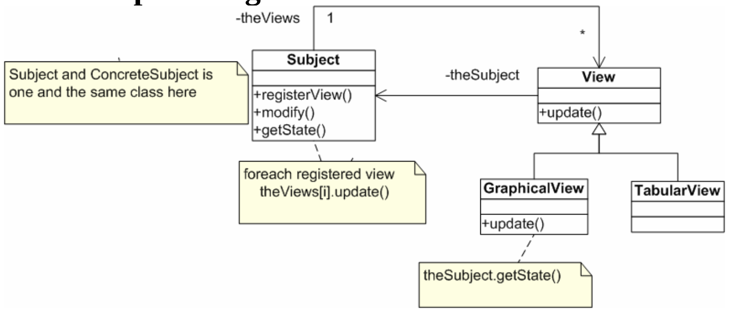
Category	Behavioral Pattern
Intent	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Purpose	Decouple independent sender(model) from dependent receivers (views)
Participants / Roles	Subject, Observer, Concrete Subject, Concrete Observer
Analogy	Auctioneer and bidders
Mnemonic	Decouple dependent “views” from the independent subject

UML Diagram Template



- Define an object that is the "keeper" of the data model and/or business logic (the Subject).
- Delegate all "view" functionality to decoupled and distinct Observer objects.
- Observers register themselves with the Subject as they are created.
- Whenever the Subject changes, it broadcasts to all registered Observers that it has changed, and each Observer queries the Subject for that subset of the Subject's state that it is responsible for monitoring.

An Example Using the Observer Pattern

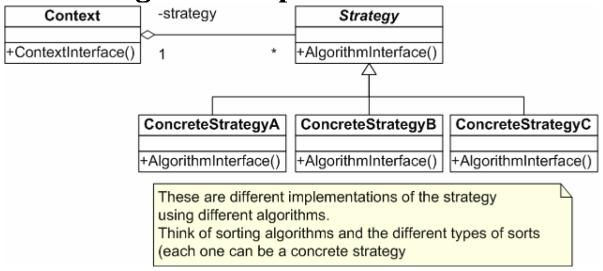


- The role of Subject is that of Subject (they just happen to have the same name)
- The role of View is that of Observer
- GraphicalView and TabularView are Concrete Observers

Description of the Strategy Pattern

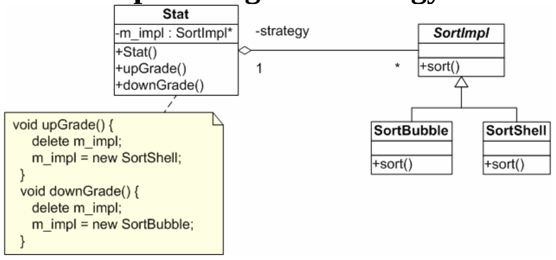
Category	Behavioral Pattern
Intent	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Purpose	publish interface in a base class, bury implementation in derived classes
Participants / Roles	Strategy, Context, ConcreteStrategy
Analogy	Number sorting algorithms
Mnemonic	Plug-compatible algorithms

UML Diagram Template



- This pattern minimizes coupling between context and the different strategies.
- Why? Because the context is coupled only to the strategy (abstraction)
- The abstraction is captured in the interface (Strategy) but the implementation is buried in the concrete classes.

An Example Using the Strategy Pattern



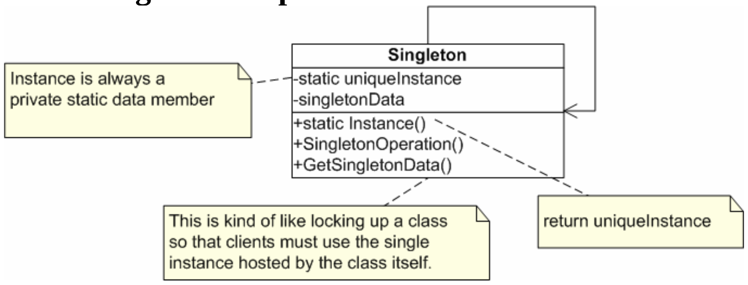
```
void upGrade() {
    delete m_impl;
    m_impl = new SortShell;
 }
 void downGrade() {
    delete m_impl;
    m_impl = new SortBubble;
 }
```

- The role of SortImpl is that of Strategy
- The role of Stat is that of Context
- SortBubble and SortShell are Concrete Strategies

Description of the Singleton Pattern

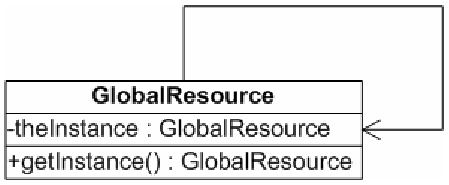
Category	Creational Pattern
Intent	Ensure a class only has one instance, and provide a global point of access to it.
Purpose	Single instance enforcement, lazy initialization, global access
Participants / Roles	Singleton
Analogy	President's office
Mnemonic	Guardian of the single instance

UML Diagram Template


Singleton
-static uniqueInstance
-singletonData
+static Instance()
+SingletonOperation()
+GetSingletonData()
Instance is always a private static data member
This is kind of like locking up a class so that clients must use the single instance hosted by the class itself.
return uniqueInstance

- Make the class of the single instance object responsible for creation, initialization and access.
- Declare the instance as a private static data member
- Provide a public static member function that encapsulates all initialization code and provides access to the instance
- Do not use singletons to replace global variables. A singleton is a global variable.

An Example Using the Singleton Pattern


GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource

- The role of GlobalResource is that of Singleton