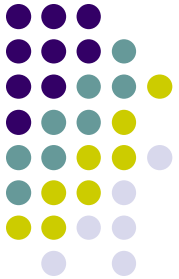
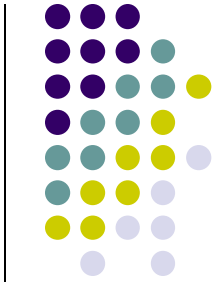


Software Architecture

Design Pattern

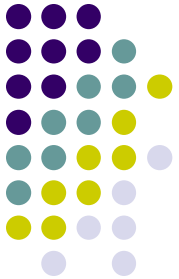
Behavioral Patterns I

Behavioral Patterns



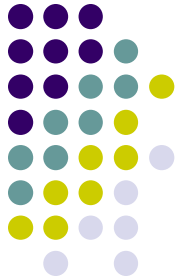
- Behavioral patterns are most specifically concerned with communication between objects.

• ***Chain of Responsibility***	• ***Observer Pattern***
• ***Command Pattern***	• ***State Pattern***
• ***Interpreter Pattern***	• ***Strategy Pattern***
• ***Iterator Pattern***	• ***Template Pattern***
• ***Mediator Pattern***	• ***Visitor Pattern***
• ***Memento Pattern***	



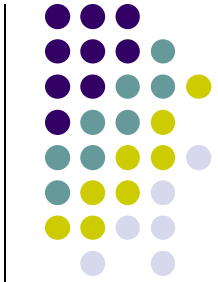
The Iterator Pattern

Iterator Pattern



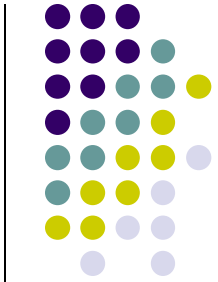
- The Iterator is one of the simplest and most frequently used of the design patterns.
- The Iterator pattern allows you to move through a list or collection of data using a standard interface without having to know the details of the internal representations of that data.

Iterator Pattern



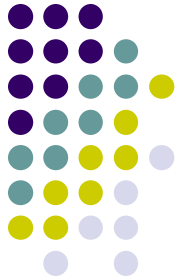
- It is possible to define special iterators that perform some special processing and return only specified elements of the data collection.

Iterator Pattern



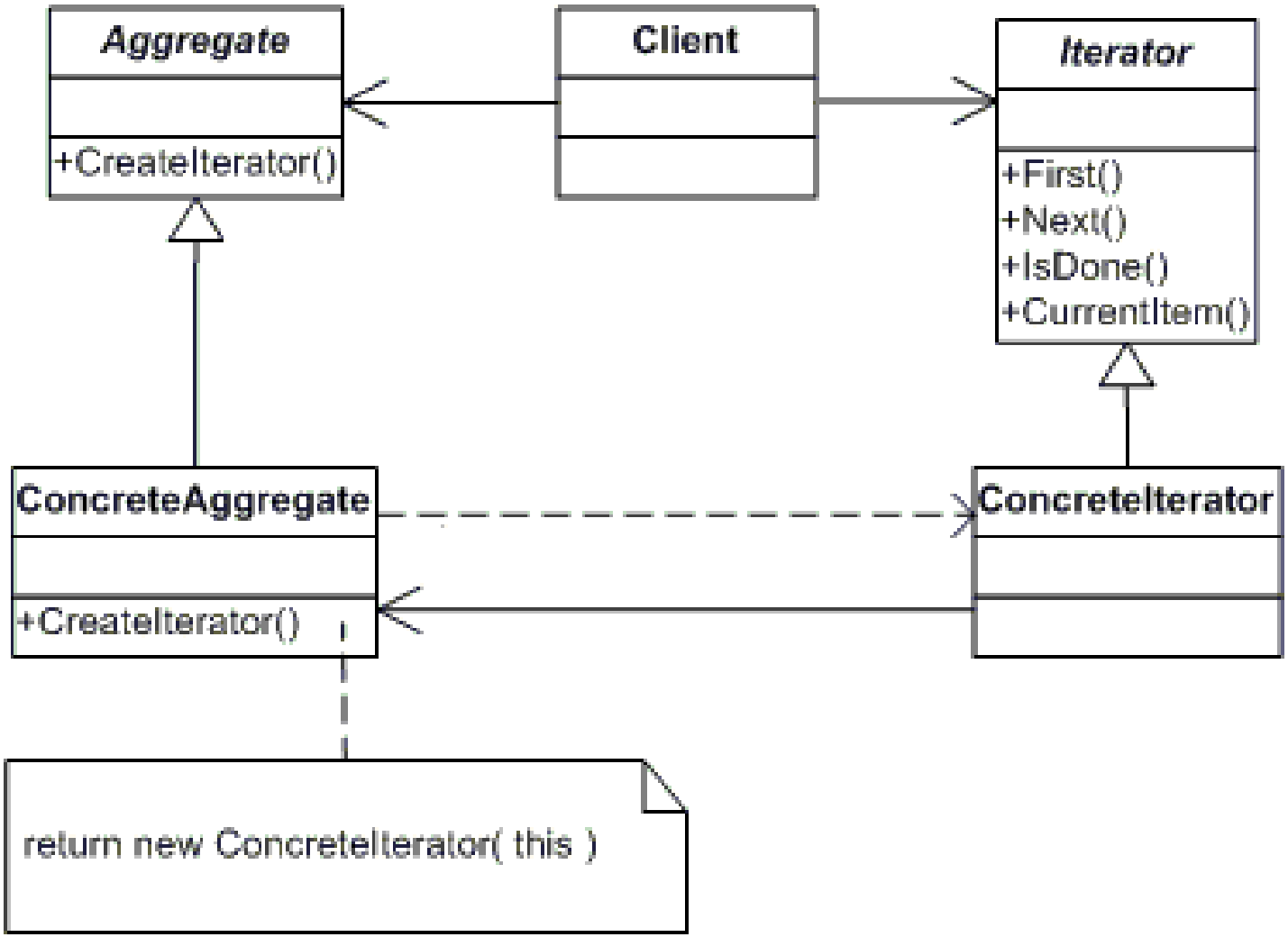
- Intent:
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Motivation:
  - access contents of a collection without exposing its internal structure.

Iterator Pattern

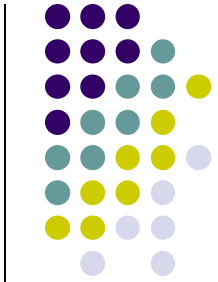


- support multiple simultaneous traversals of a collection.
- provide a uniform interface for traversing different collection.
- It should allow different traversal methods
- It should not add all these methods to the interface for the aggregate

Structure

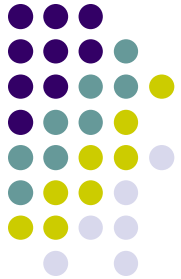

Aggregate
+CreateIterator()
Client
Iterator
+First()
+Next()
+IsDone()
+CurrentItem()
ConcreteAggregate
+CreateIterator()
ConcreteIterator
return new ConcreteIterator(this)

Iterator Pattern



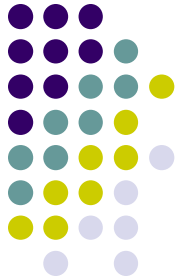
- Participants:
  - Iterator
    - Defines an interface for accessing and traversing elements
  - ConcreteIterator
    - Implements the Iterator interface
    - Keeps track of the current position in the traversal
  - Aggregate
    - Defines an interface for creating an Iterator object (a factory method!)

Iterator Pattern



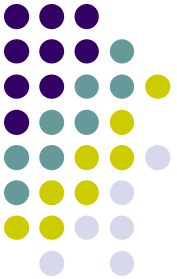
- ConcreteAggregate
  - Implements the Iterator creation interface to return an instance of the proper ConcreteIterator

Iterator Pattern



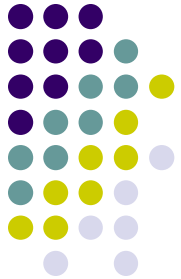
- Related Patterns
  - Factory Method
    - Polymorphic iterators use factory methods to instantiate the appropriate iterator subclass
  - Composite
    - Iterators are often used to recursively traverse composite structures

Java Implementation of Iterator



- Do we have to implement the Iterator pattern “from scratch” in Java?
- Java provides built-in support for the Iterator pattern
  - The java.util.Enumeration and java.util.Iterator interface acts as the Iterator interface
  - Java provides many aggregate classes, such as sets, lists, maps and vector.

Java Implementation of Iterator

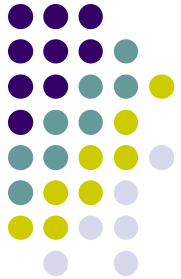


- Aggregate classes that want to support iteration provide methods that return a reference to an Enumeration or Iterator object.
- The Enumeration or Iterator object allows a client to traverse the aggregate object

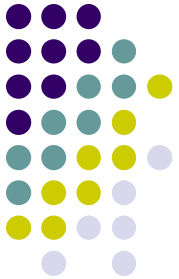
The Iterator Pattern

- Applicability
  - Use the Iterator pattern:
    - To support traversals of aggregate objects without exposing their internal representation
    - To support multiple, concurrent traversals of aggregate objects
    - To provide a uniform interface for traversing different aggregate structures (i.e., to support polymorphic iteration)

Consequence of the Iterator Pattern

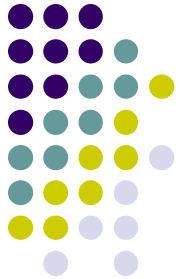


- Benefits
  - Simplifies the interface of the Aggregate by not polluting it with traversal methods
  - Supports multiple, concurrent traversals
  - Supports variant traversal techniques



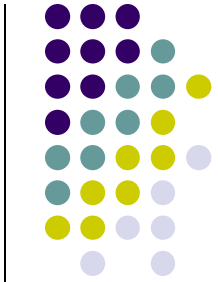
Observer Pattern

Observer Pattern



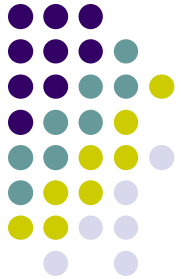
- Observer Design Pattern can be used whenever a subject has to be observed by one or more observers.
- Why do we need the observer pattern?
  - To display data in more than one form at the same time and have all of the displays reflect any changes in that data.

Observer Pattern



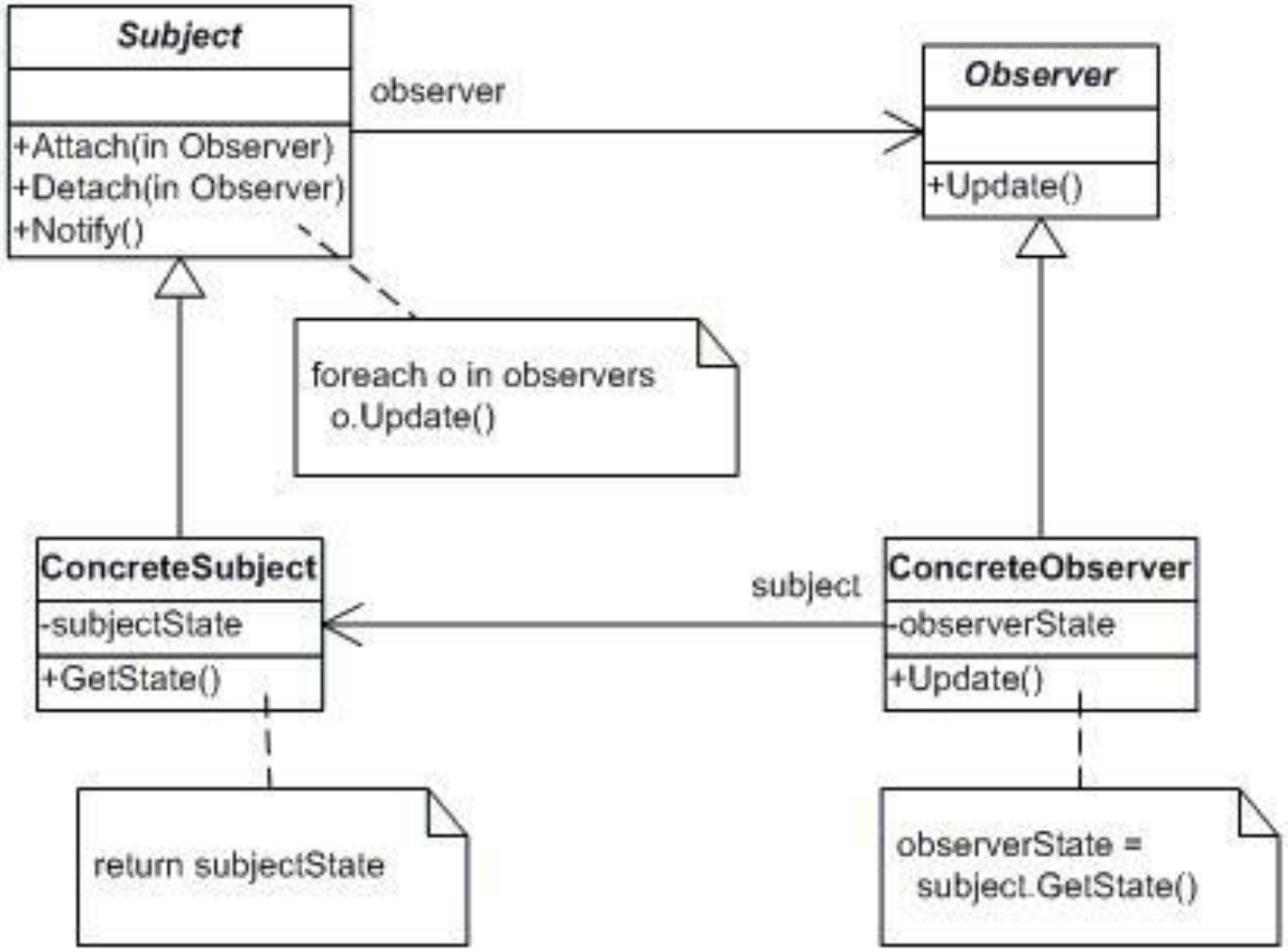
- Intent
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Motivation
  - The need to maintain consistency between related objects without making classes tightly coupled

Observer Pattern

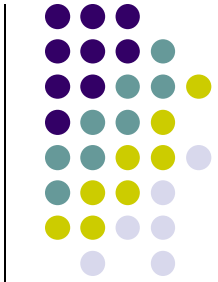


- The Observer pattern separate the object containing the data from the objects that display the data, and that these display objects *observe* changes in that data.

Observer Pattern

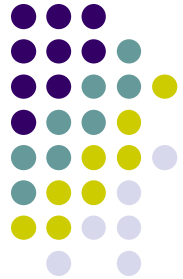


Observer Pattern: Participants



- Subject
  - Keeps track of its observers
  - Provides an interface for attaching and detaching Observer objects
- Observer
  - Defines an interface for update notification
- ConcreteSubject
  - The object being observed
  - Stores state of interest to ConcreteObserver objects

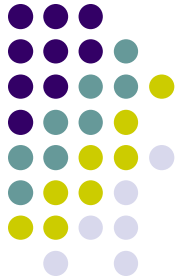
Observer Pattern: Participants



- Sends a notification to its observers when its state changes

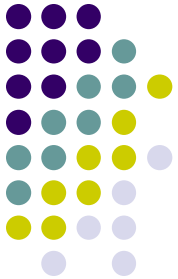
- ConcreteObserver
  - The observing object
  - Stores state that should stay consistent with the subject's
  - Implements the Observer update interface to keep its state consistent with the subject's

Observer Pattern



- We could implement the Observer pattern “from scratch” in Java
- Java provides the java.util.Observer and java.util.Observable classes as built-in support for the Observer pattern
- The java.util.Observable class is the base Subject class. Any class that wants to be observed extends this class.

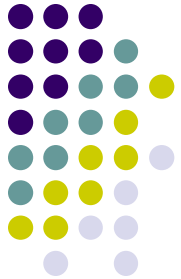
Observer Pattern



- Provides methods to add/delete observers
- Provides methods to notify all observers
- A subclass only needs to ensure that its observers are notified in the appropriate mutations
- Uses a Vector for storing the observer references

- The java.util.Observer interface is the Observer interface. It must be implemented by any observer class.

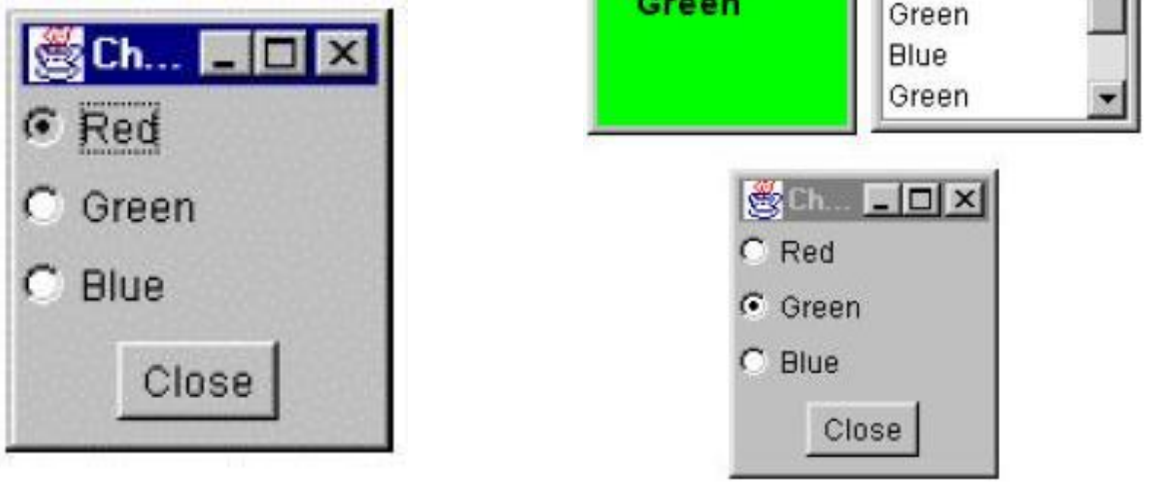
Observer Pattern



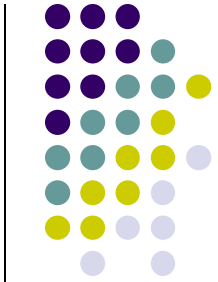
- Java's GUI event model is based on the Observer Pattern
- Comparison to the Observer Pattern:
  - ConcreteSubject => event source
  - ConcreteObserver => event listener
- For an event listener to be notified of an event, it must first register with the event source
- Java AWT event model has 18 different listener interfaces

Observer Pattern:Example

- two observers:
  - An observer that displays the color (and its name)
  - An observer that adds the current color to a list box.

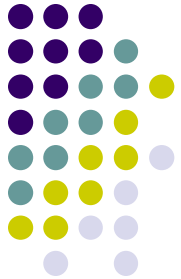


OBSERVER PATTERN



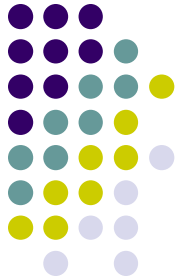
- Applicability:
- The observer pattern is used when:
  - the change of a state in one object must be reflected in another object without keeping the objects tight coupled.
  - the framework we are writing needs to be enhanced in future with new observers with minimal changes.
- Classical Examples:
  - Model View Controller Pattern
  - Event management

Consequences



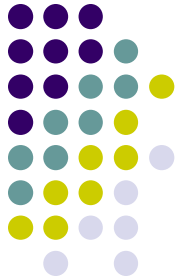
- Benefits
  - Minimal coupling between the Subject and the Observer
    - Can reuse subjects without reusing their observers and vice versa
    - Observers can be added without modifying the subject
    - All subject knows its list of observers

Consequences

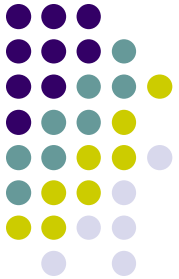


- Subject does not need to know the concrete class of an observer, just that each observer implements the update interface
- Subject and observer can belong to different abstraction layers

OBSERVER PATTERN

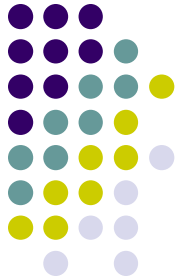


- Support for event broadcasting
  - Subject sends notification to all subscribed observers
  - Observers can be added/removed at any time



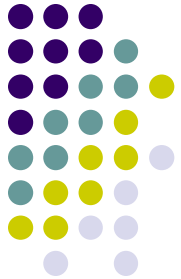
State Pattern

State Pattern



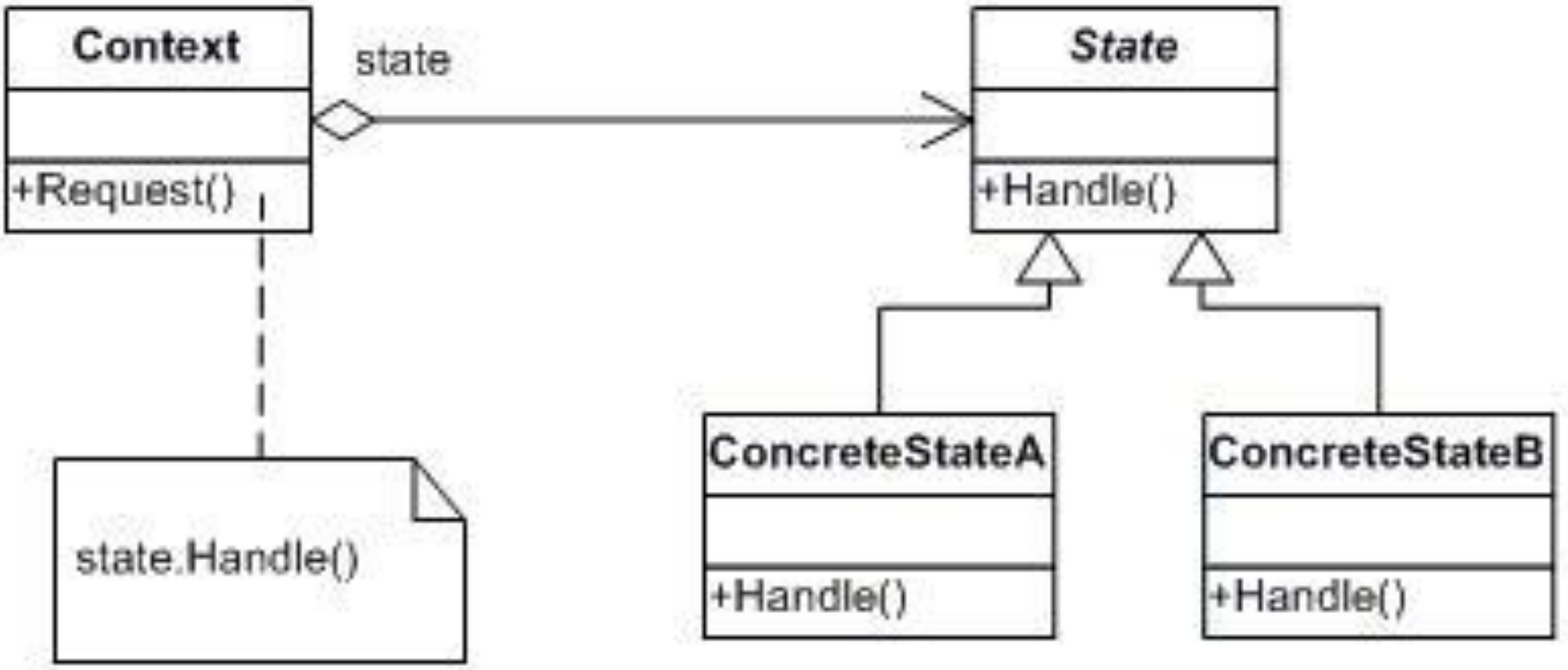
- Why do we need the state pattern?
  - Creating a class, which performs slightly different computations or displays different information based on the arguments passed into the class, frequently leads to some sort of *switch* or *if-else* statements inside the class that determine which behavior to carry out. It is this inelegance that the State pattern seeks to replace.

State Pattern

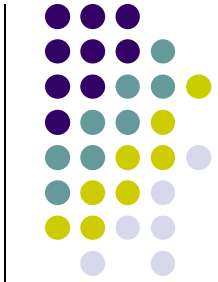


- Intent
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

State Pattern

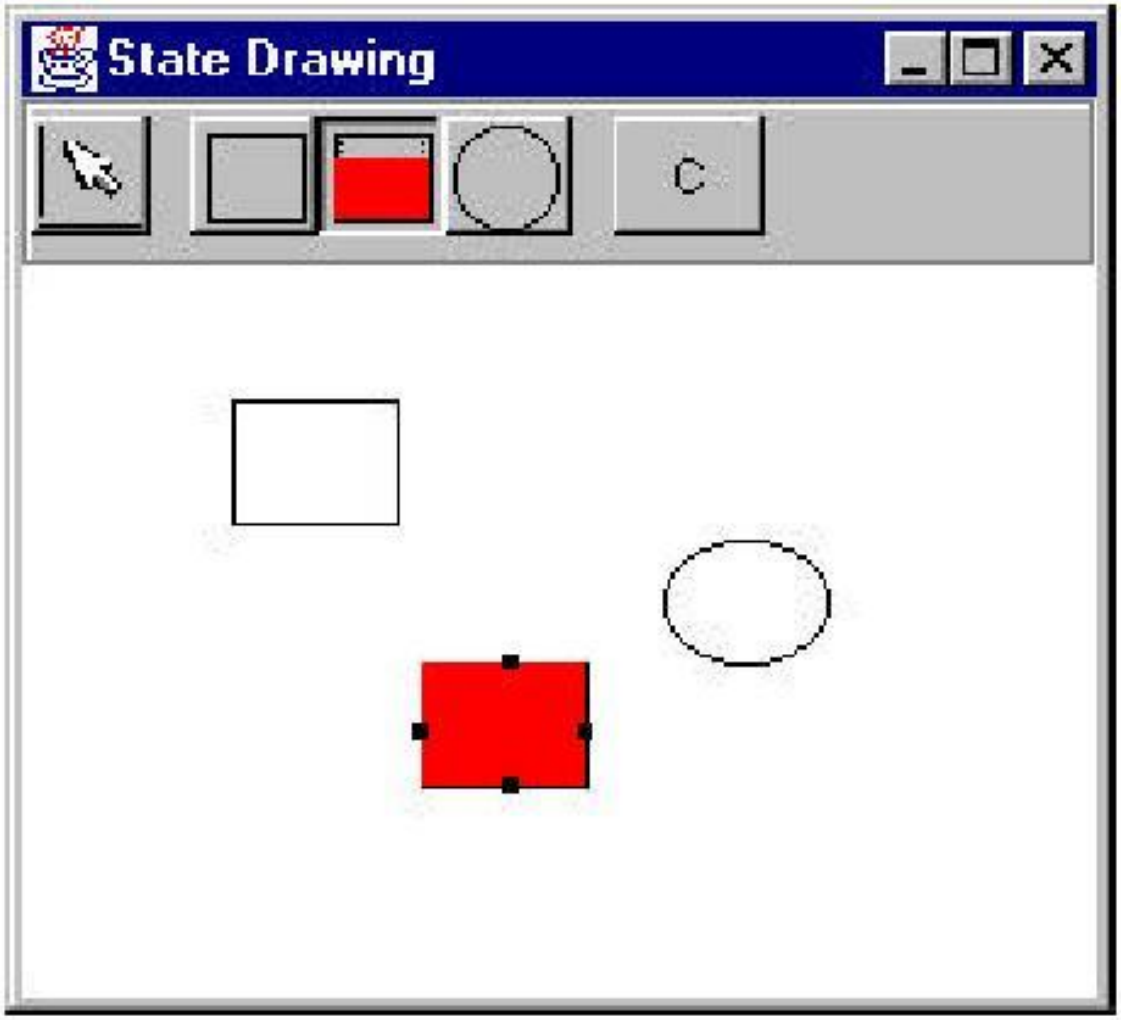


State Pattern:participants

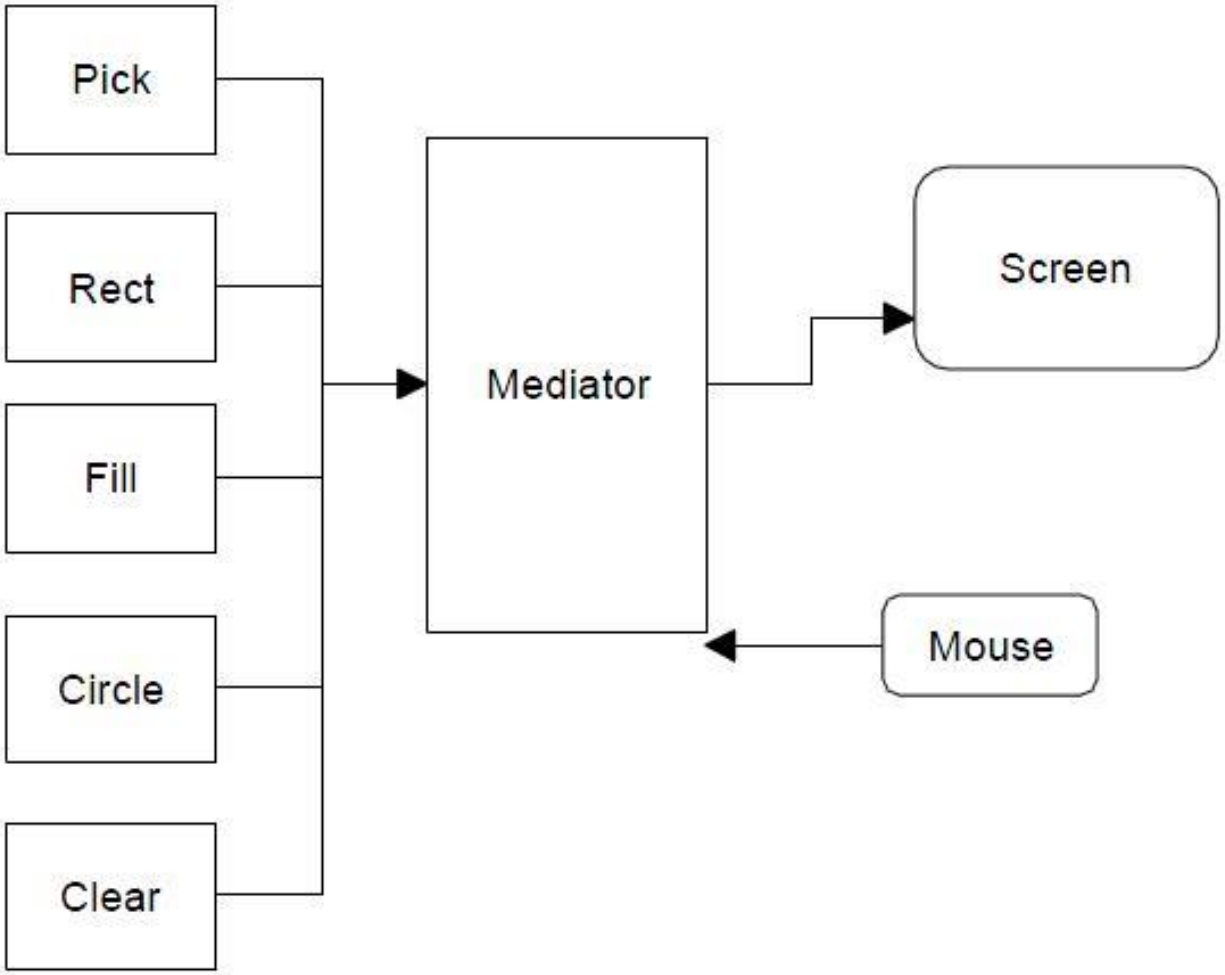


- **Context**
  - defines the interface of interest to clients
  - maintains an instance of a ConcreteState subclass that defines the current state.
- **State**
  - defines an interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete State**
  - each subclass implements a behavior associated with a state of Context

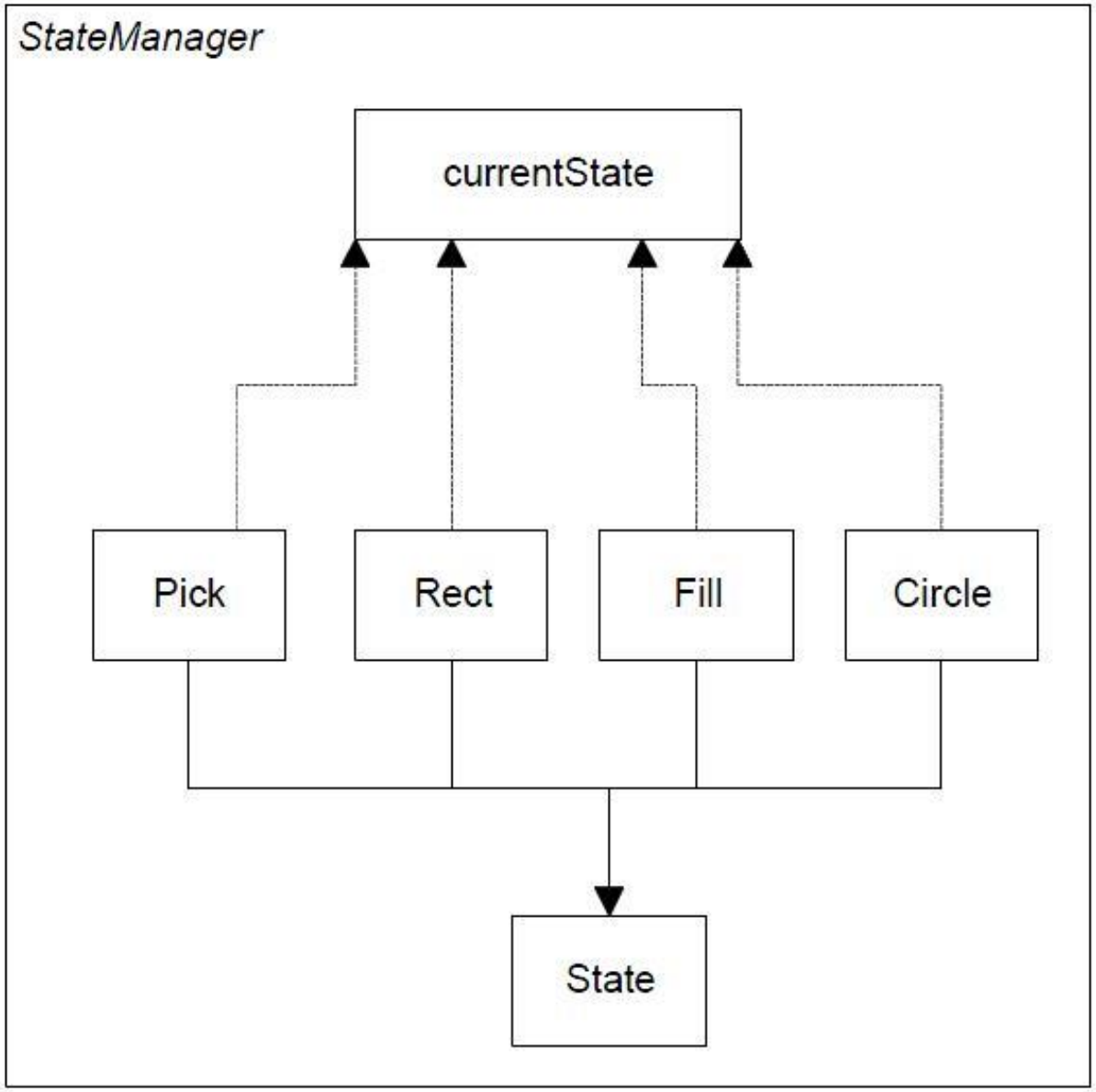
State Pattern: Example



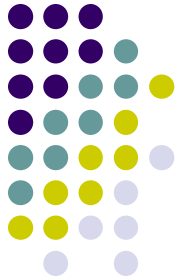
State Pattern: Example



The State Pattern: Example 2

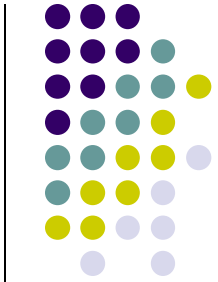


State Pattern: Applicability

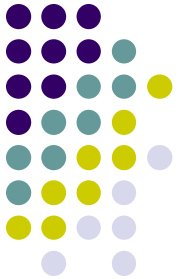


- Use the State pattern whenever:
  - An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
  - Operations have large, multipart conditional statements that depend on the object's state. The State pattern puts each branch of the conditional in a separate class.

Consequences

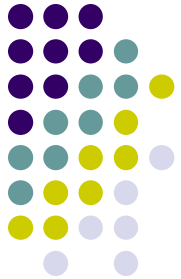


- pros
  - Puts all behavior associated with a state into one object
  - Allows state transition logic to be incorporated into a state object rather than in a monolithic if or switch statement
- cons
  - Increased number of objects



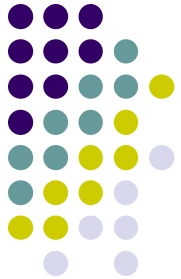
Strategy pattern

Strategy pattern



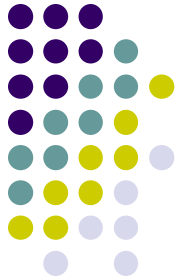
- Why do we need the strategy pattern?
  - There are a number of related algorithms encapsulated in a Context. The client program can select one of these differing algorithms or in some cases the Context might select the best one for the client.

Strategy pattern



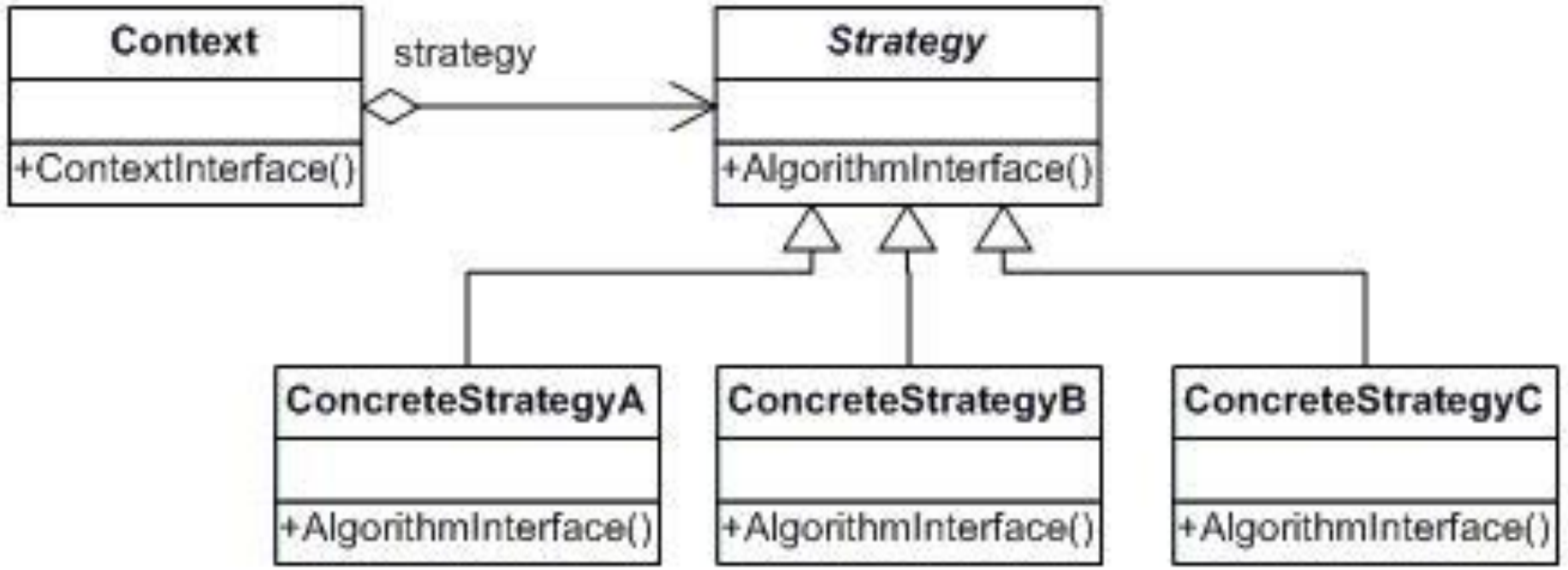
- Save files in different formats.
- Compress files using different algorithms
- Capture video data using different compression schemes
- Use different line-breaking strategies to display text data.
- Plot the same data in different formats: line graph, bar chart or pie chart.

Strategy pattern

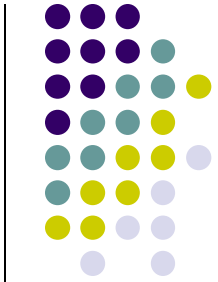


- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy pattern

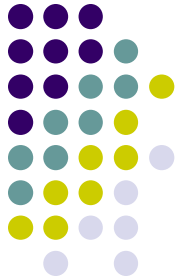


Strategy pattern



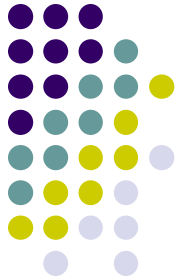
- **Strategy**
  - declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy
- **ConcreteStrategy**
  - implements the algorithm using the Strategy interface

Strategy pattern



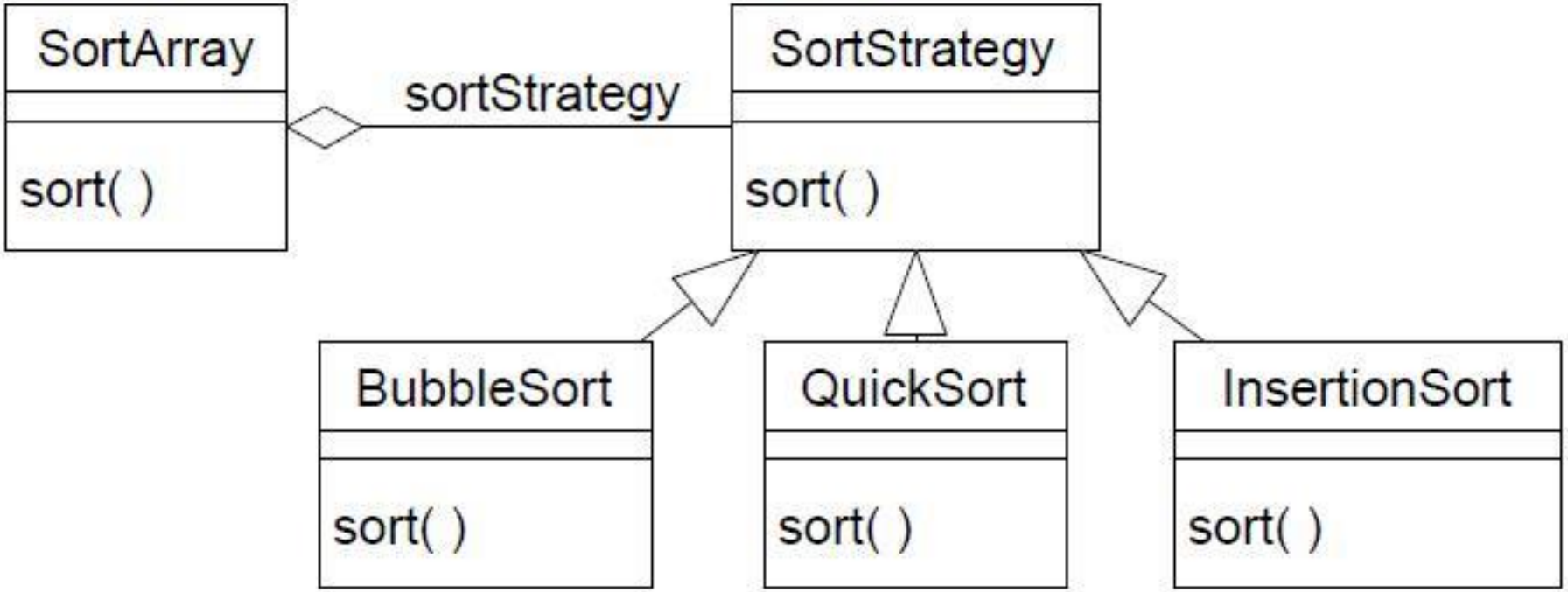
- **Context**
  - is configured with a ConcreteStrategy object
  - maintains a reference to a Strategy object
  - may define an interface that lets Strategy access its data.

Strategy Pattern Example 1

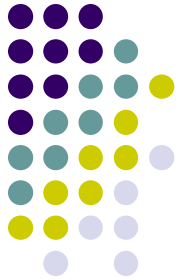


- Situation: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available.
- Solution: Encapsulate the different sort algorithms using the Strategy pattern.

Strategy Pattern Example 1

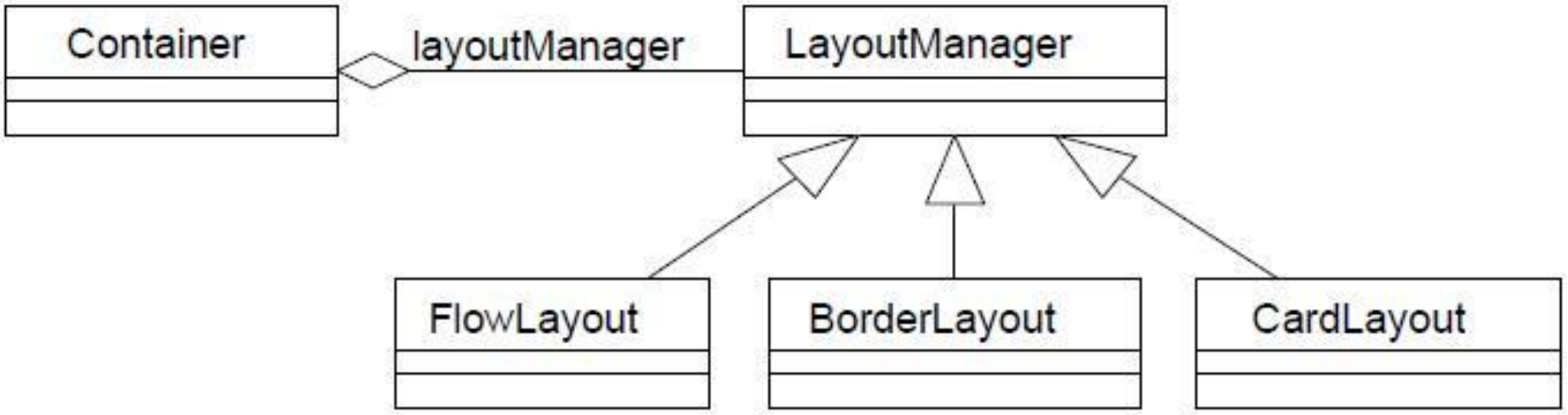


Strategy Pattern Example 2

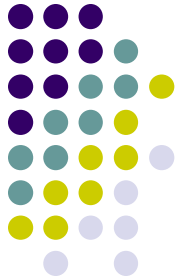


- Situation: A GUI container object wants to decide at run-time what strategy it should use to layout the GUI components it contains. Many different layout strategies are already available.
- Solution: Encapsulate the different layout strategies using the Strategy pattern! This is what the Java AWT does with its LayoutManagers!

Strategy Pattern Example 2

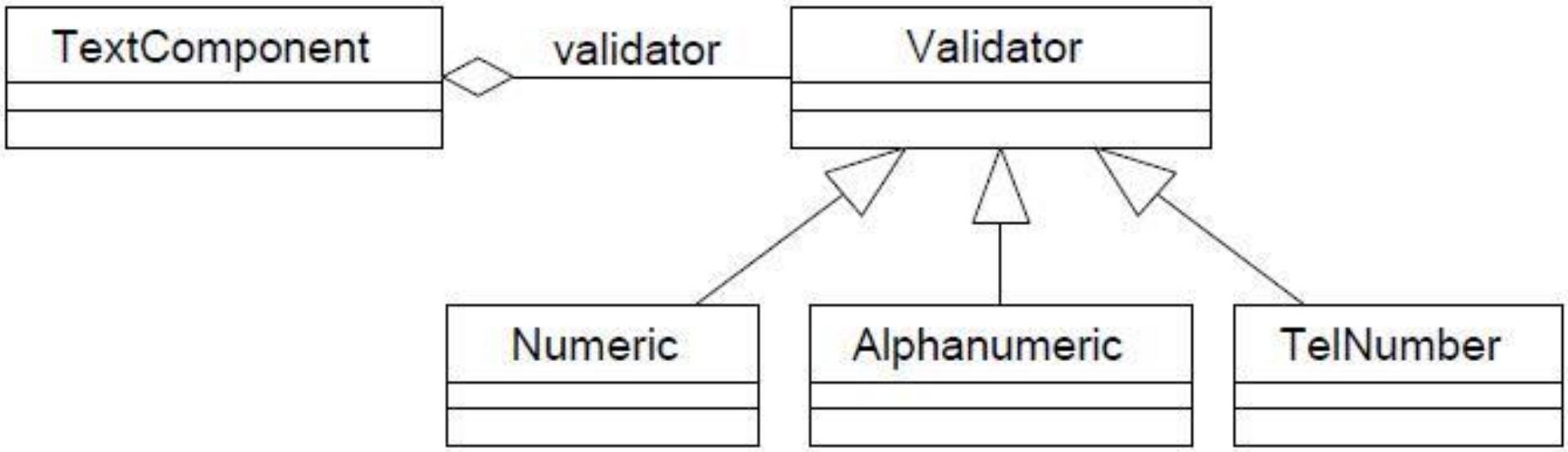


Strategy Pattern Example 3

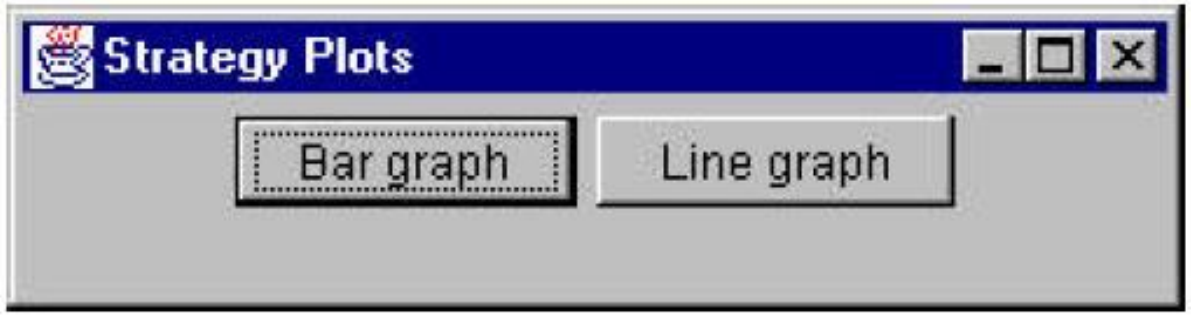


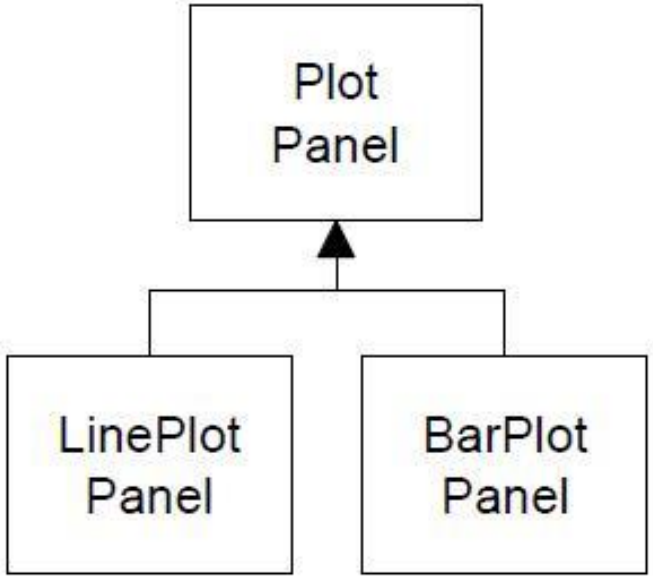
- Situation: A GUI text component object wants to decide at runtime what strategy it should use to validate user input. Many different validation strategies are possible: numeric fields, alphanumeric fields, telephone-number fields, etc.
- Solution: Encapsulate the different input validation strategies using the Strategy pattern!

Strategy Pattern Example 3

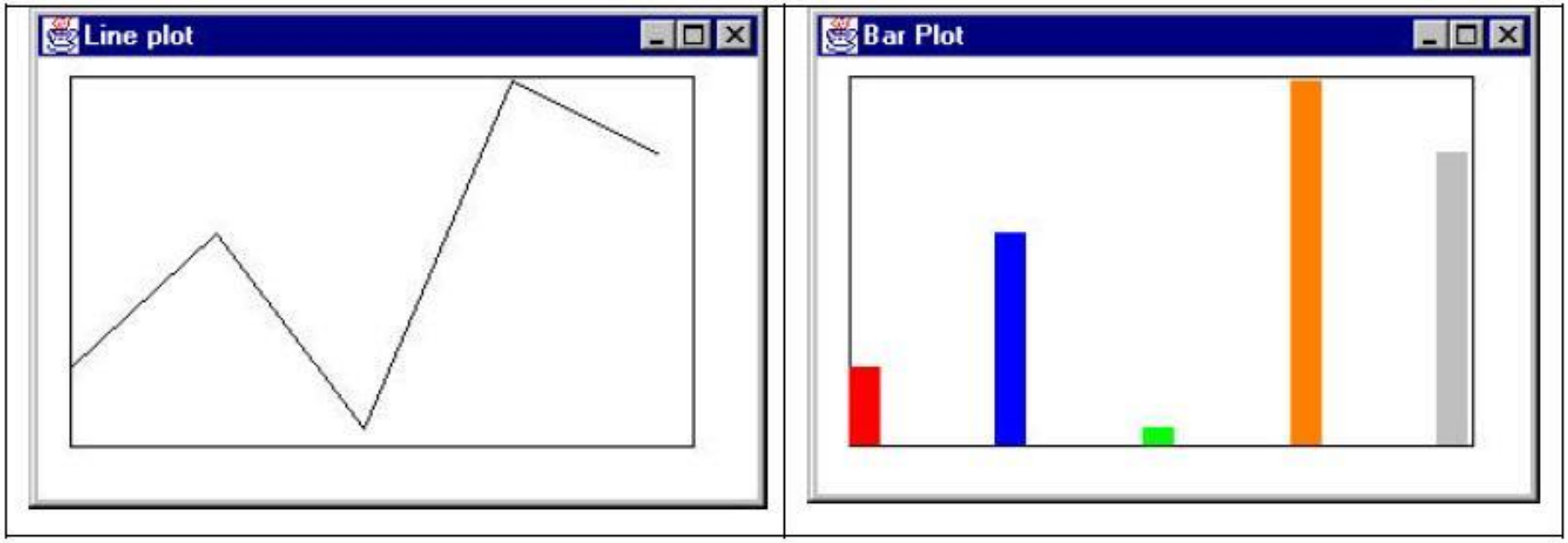


Strategy Pattern Example 4

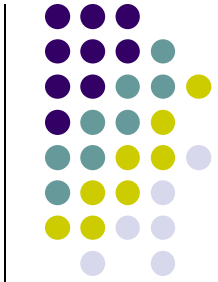




Strategy Pattern Example 4

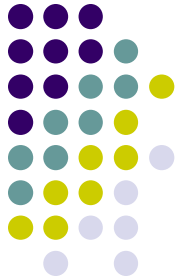


Strategy Pattern & State Pattern



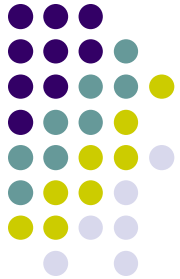
- The difference:
  - A Strategy object encapsulates an algorithm
  - A State object encapsulates a state-dependent behavior (and possibly state transitions).
  - Strategy: the user generally chooses which of several strategies to apply and that only one strategy at a time is likely to be instantiated and active within the Context class.
  - State: it is likely that all of the different States will be active at once and switching may occur frequently between them.

Strategy Pattern & State Pattern



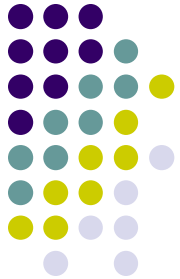
- Strategy encapsulates several algorithms that do more or less the same thing, while State encapsulates related classes that each do something somewhat different.
- the concept of transition between different states is completely missing in the Strategy pattern.

Strategy pattern: Applicability



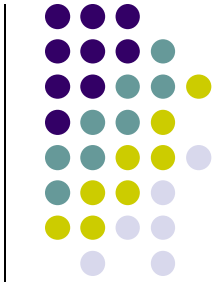
- Use the Strategy pattern whenever:
  - Many related classes differ only in their behavior
  - You need different variants of an algorithm
  - An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.

Strategy pattern: Applicability

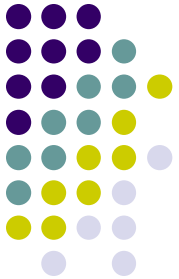


- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

Strategy pattern

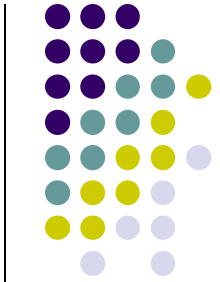


- pros
  - Provides an alternative to subclass the Context class to get a variety of algorithms or behaviors
  - Eliminates large conditional statements
  - Provides a choice of implementations for the same behavior
- cons
  - Increases the number of objects
  - All algorithms must use the same Strategy interface



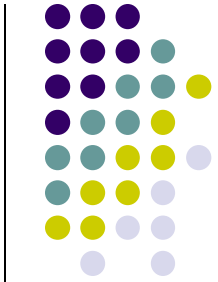
Template Pattern

Template Pattern



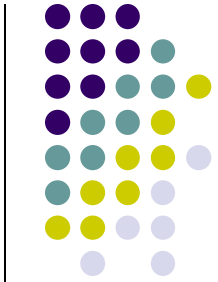
- The idea behind the Template pattern is that some parts of an algorithm are well defined and can be implemented in the base class, while other parts may have several implementations and are best left to derived classes.

Template Pattern



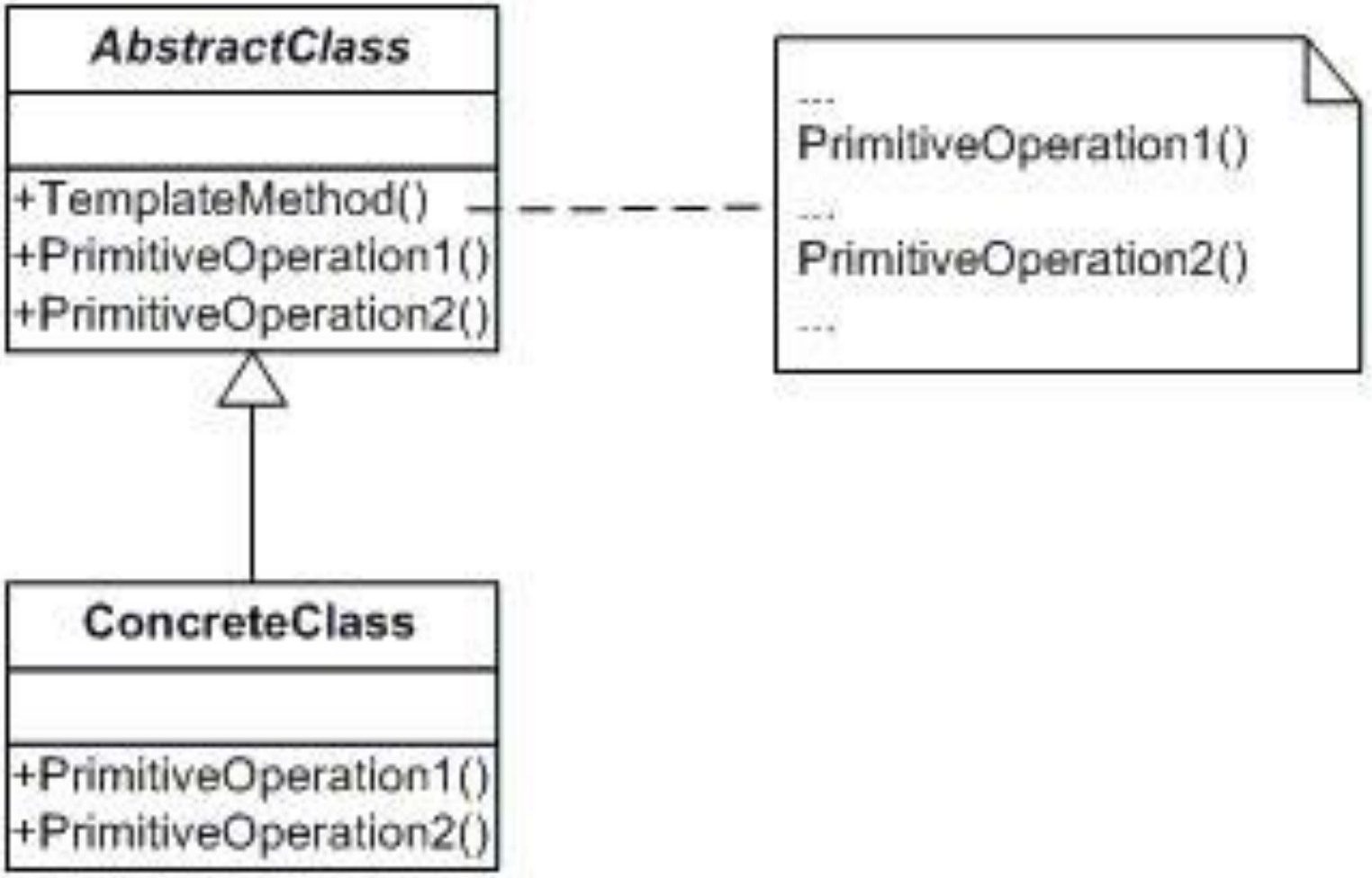
- Intent
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
  - Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

Template Pattern

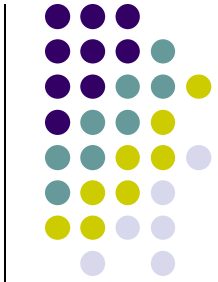


- Motivation
  - Sometimes you want to specify the order of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations

Template Pattern

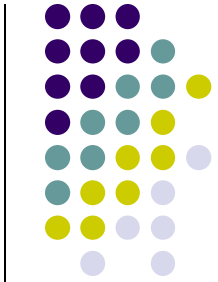


Template Pattern



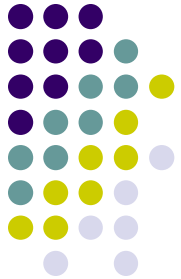
- **AbstractClass**
  - defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.

Template Pattern



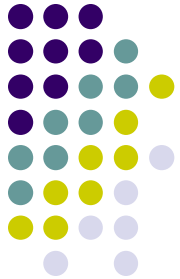
- **ConcreteClass**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm

The Template Pattern



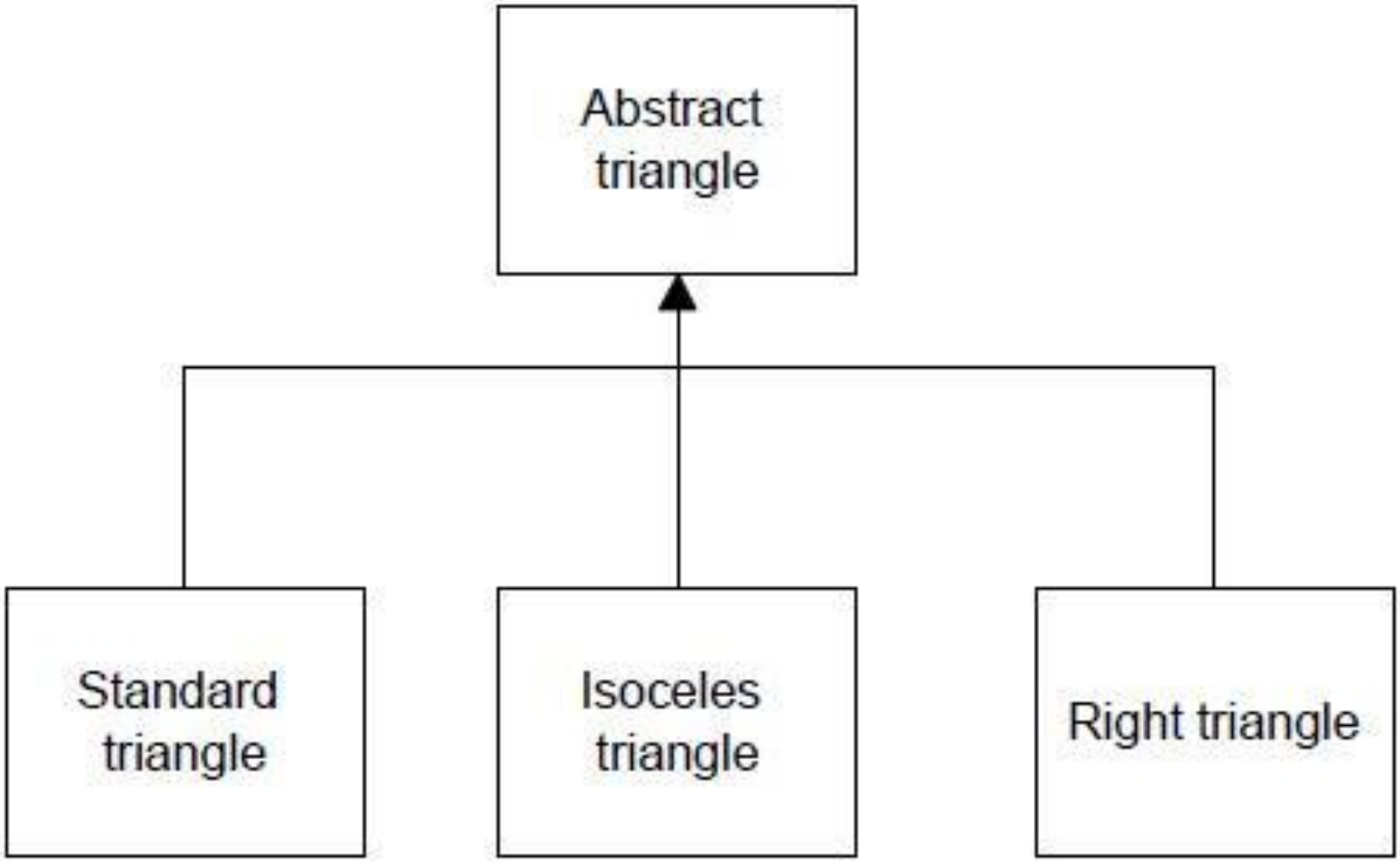
- Kinds of Methods in a Template Class
  - Concrete methods.
    - Complete methods that carry out some basic function that all the subclasses will want to use.
  - abstract methods.
    - Methods that are not filled in at all and must be implemented in derived classes.

The Template Pattern

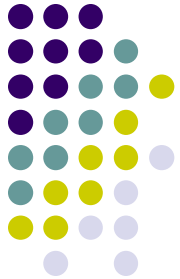


- Hook methods.
  - Methods that contain a default implementation of some operations, but which may be overridden in derived classes.
- Template methods .
  - a Template class may contain methods which themselves call any combination of abstract, hook and concrete methods. These methods are not intended to be overridden, but describe an algorithm without actually implementing its details.

The Template Pattern: Example



Template pattern :Applicability



- Use the Template Method pattern:
  - to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
  - when refactoring is performed and common behavior is identified among classes. A abstract base class containing all the common code (in the template method) should be created to avoid code duplication.