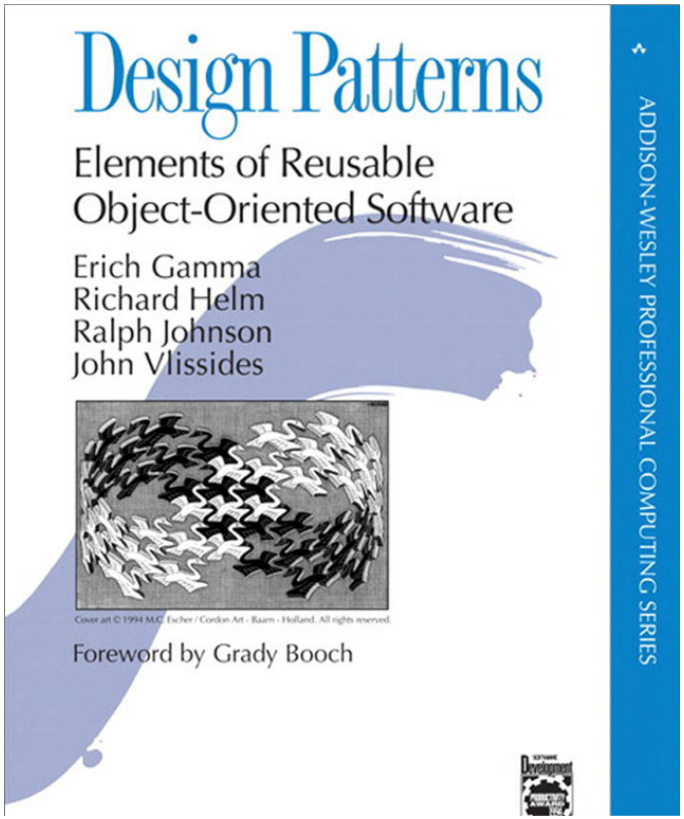
Object-Oriented Design Patterns

Georgia Tech

Design Patterns



Recurring object-oriented designs.

- Make proven techniques more accessible to developers of new systems – don't have to study other systems.
- Helps in choosing designs that make the system more reusable.
- Facilitate documenentation and communication with other developers.

Design pattern catalog: descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

Georgia Tech

Elements of Design Patterns

- The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two.
- The **problem** describes when to apply the pattern.
- The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The pattern provides an abstract description of a design problem and how a general arrangement of classes and objects solves it.
- The consequences are the results and trade-offs of applying the pattern.

Georgia Tech

How Design Patterns Solve Design Problems

- Finding appropriate objects
- Determining object granularity
- Specifying object interfaces
- Specifying object implementations

Georgia Tech

Types versus Classes

Class versus interface inheritance

- An object's class defines how the object is implemented.
  - Class inheritance defines an object's implementation in terms of another object's implementation.
- An object's type refers to its interface – the set of methods it can respond to.
  - Interface inheritance is subtyping. It describes when an object can be used in place of another.

Program to an interface, not an implementation.

Georgia Tech

Reuse Mechanisms

Inheritance versus composition

- Inheritance: "White box reuse" – subclass reuses details of superclass and extends with new functionality
  - Defined at compile-time.
  - Straightforward to use
  - "Inheritance breaks encapsulation" (superclass implementation exposed to subclasses)
  - Reuse can be difficult in new contexts – may require rewriting superclasses or carrying baggage.
- Composition: "Black-box reuse" – new functionality obtained by composing objects of other objects
  - Defined at run-time by objects acquiring references to other objects.
  - Must program to interfaces, so interfaces must be well thought-out and stable.
  - Emphasis on interface stability encourages granular objects with single responsiblities.

Favor object composition over class inheritance.

Georgia Tech

Designing for Change

Common causes of redesign (and design patterns that address them):

- Creating an object by specifying a class explicitly. (Factory)
- Dependence on specific operations. (Command)
- Dependence on hardware and software platform. (Factory, Bridge).
- Dependence on object representations or implementations (Factory, Bridge, Proxy).
- Algorithmic dependencies. (Visitor, Iterator, Strategy, Template Method)
- Tight coupling. Design patterns: (Factory, Bridge, Command, Facade, Mediator, Observer).
- Extending functionality by subclassing. (Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy)
- Inability to alter classes conveniently. (Adapter, Decorator, Visitor)

Georgia Tech

Selecting a Design Pattern

- Consider how design patterns solve design problems.
- Scan Intent sections. Read through each pattern's intent to find one or more that sound relevant to your problem.
- Study how patterns interrelate. Studying these relationships can help direct you to the right pattern or group of patterns.
- Study patterns of like purpose.
- Examine a cause of redesign, look at the patterns that help you avoid the causes of redesign.
- Consider what should be variable in your design.
  - This approach is the opposite of focusing on the causes of redesign.
  - Instead of considering what might force a change to a design, consider what you want to be able to change without redesign.
  - The focus here is on encapsulating the concept that varies, a theme of many design patterns.

Georgia Tech

Using Design Patterns (1 of 2)

- Read the pattern once through for an overview.
- Go back and study the Structure, Participants, and Collaborations sections. Make sure you understand the classes and objects in the pattern and how they relate to one another.
- Look at Sample Code to see a concrete example of the pattern in code.
- Choose names for pattern participants that are meaningful in the application context. OK to use abstract participant names from design pattern. For example, if you use the Strategy pattern for a text compositing algorithm, then you might have classes SimpleLayoutStrategy or TeXLayoutStrategy.

Georgia Tech

Using Design Patterns (2 of 2)

- Define the classes. Declare their interfaces, establish their inheritance relationships, and define the instance variables that represent data and object references. Identify existing classes in your application that the pattern will affect, and modify them accordingly.
- Define application-specific names for operations in the pattern. Use the responsibilities and collaborations associated with each operation as a guide. Be consistent in your naming conventions. For example, you might use the "Create-" prefix consistently to denote a factory method.
- Implement the operations to carry out the responsibilities and collaborations in the pattern. The Implementation section from a pattern catalog and sample code offers hints to guide you in the implementation.

Georgia Tech

Common Design Patterns

- **Abstract Factory** Provide an interface for creating families of related or dependent objects without specifying their concrete classes. (GoF, 87)
- **Factory Method** Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses. (GoF, 107)
- **Adapter** Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompat ible interfaces. (GoF, 139)
- **Composite** Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly. (GoF, 163)

Georgia Tech

Common Design Patterns

- **Decorator** Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality. (GoF, 175)
- **Observer** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. (GoF, 293)
- **Strategy** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it. (GoF, 315)
- **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. (GoF, 325)

Georgia Tech

Creational Design Patterns

Abstracts the instantiation process.

- Encapsulate knowledge about which concrete classes the system uses.
- Hide how instances of these classes are created and put together.

Georgia Tech

Abstract Factory

Intent: Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Structure:

Participants

- **AbstractFactory** declares an interface for operations that create abstract product objects.
- **ConcreteFactory** implements the operations to create concrete product objects.
- **AbstractProduct** declares an interface for a type of product.
- **ConcreteProduct** defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface.

Georgia Tech

Abstract Factory Example: `java.sql.Connection`

```
1
2  public interface Connection ... {
3      public Blob createBlob();
4      public Statement createStatement();
5      public PreparedStatement prepareStatement();
6      ...
7  }
```

- The `Connection` interface has factory methods for a family of related classes.
- A particular `Connection` instance would return database-specific implementations of `Statement`, etc.

```
1  String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
2  Connection conn = DriverManager.getConnection(URL);
```

Georgia Tech

Factory Method (a.k.a. Virtual Constructor)

Intent: Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Structure:

Participants:

- **Product** defines the interface of objects the factory creates. ConcreteProduct implements the Product interface.
- **Creator** declares the factory method, which returns an object of type Product.
- **ConcreteCreator** overrides factory method to return a ConcreteProduct object.

Georgia Tech

Factory Method Example: Active Records (1 of 4)

Say we have a solution domain object that represents a problem domain entity:

```
1  public class Person {
2      protected final int id;
3      protected String name;
4      public Person(int id, String name) {
5          this.id = id;
6          this.name = name;
7      }
8      public int getId() { return id; }
9      public String getName() { return name; }
10     public void setName(String name) { this.name = name; }
11 }
```

How can we add persistence capability in an abstract way so that we can swap out different persistence implementations (database, etc.)?

Georgia Tech

Factory Method Example: Active Records (2 of 4)

Active Records are objects that know how to store and retrieve themselves from a data store. The simplest implementation of an ActiveRecord uses an abstract class:

```
1  public abstract class ActivePerson extends Person {
2      public ActivePerson(int id, String name) {
3          super(id, name);
4      }
5      public abstract Person createNew(String name);
6      public abstract Person findById(int id);
7      public abstract void save();
8  }
```

`ActivePerson` extends `Person` with persistence capabilities. Now applications that use a particular data store can sublcass `ActivePerson` and implement data store-specific versions of these persistence methods.

Georgia Tech

Factory Method Example: Active Records (3 of 4)

Here's a subclass of `ActivePerson` that uses a HashMap:

```
1  public class HashMapPerson extends ActivePerson {
2      private static HashMap<Integer, Person> persons = new HashMap<>();
3      private static int lastUsedId = 0;
4      protected HashMapPerson(int id, String name) {
5          super(id, name);
6      }
7      public Person createNew(String name) {
8          Person newPerson = new HashMapPerson(lastUsedId++, name);
9          persons.put(newId, newPerson);
10         return newPerson;
11     }
12     public Person findById(int id) {
13         return persons.get(id);
14     }
15     public void save() {
16         // nothing to do - client has alias to object in HashMap
17     }
18 }
```

Factory Method Example: Active Records (4 of 4)

Benefits of using `ActivePerson`:

- A `MySQLPerson` would implement MySQL-specific code that maps relational database reperesentations of objects to their Java object counterparts.
- Application is coded to `ActivePerson` interface so versions of `ActivePerson` that use different data stores can be swapped out by changing only the client code that instantiates the `ActivePerson` objects.
- You could put all of your active record-instantiating code in an Abstract Factory or a registry (which could be a singleton) so there's only one place to make this change for all kinds of peristed objects.

There are other ways of doing this, but active records are easy to understand. Object-relational mapping and data store frameworks use these concepts.

Georgia Tech

Implementing Factories with Reflection

Reflection is an advanced Java programming technique often used to implement factories. Consider:

```
1 MyClass instance = new MyClass();
```

You can also do this with reflection:

```
1 MyClass instance = (MyClass) Class.forName("MyClass").newInstance();
```

You can store the string "MyClass" in a properties file, which could be changed without changing any code. Take a look at greeter for a simple but complete example of this technique.

Georgia Tech

Singleton

Intent: Ensure a class only has one instance, and provide a global point of access to it.

Structure

Participants

- **Singleton** defines an Instance operation that lets clients access its unique instance.
- **Instance** is a class operation (that is, a class method in Smalltalk and a static member function in C++, or static method in Java). May be responsible for creating its own unique instance.

Georgia Tech

Singleton Example: java.text.NumberFormat

Remember NumberFormat from CS 1331?

```
1  public abstract class NumberFormat extends Format {
2      protected NumberFormat() {}
3      public final static NumberFormat getInstance() { ... }
4      public static NumberFormat getInstance(Locale inLocale) { ... }
5      ...
6  }
```

- `NumberFormat` instance is instantiated once; this instance is shared by all users of NumberFormat
- `getInstance()` is also a factory method: creates a NumberFormat instance for a particular Locale

Georgia Tech

Implementing a Singleton

Three things to make a singleton:

- hide constructor,
- store singleton instance in some cache,
- provide public access to singleton instance.

A minimum example:

```
1  public class MySingleton {
2      protected static instance;
3      // Hidden with private visibility - can only instantiate inside
4      private MySingleton() {}
5
6      public static MySingleton getInstance() {
7          if (instance == null) {
8              instance = new MySingleton();
9          }
10         return instance;
11     }
12
13 }
```

Georgia Tech

Structural Design Patterns

Concerned with how classes and objects are composed to form larger structures.

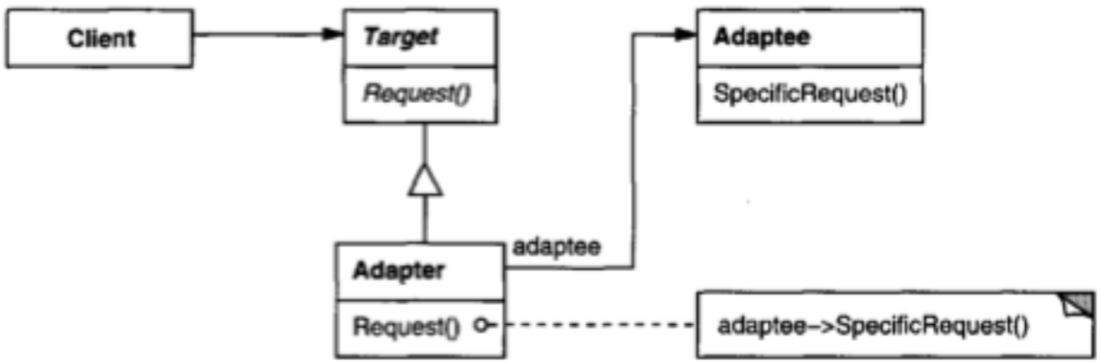
- Structural class patterns use inheritance to compose interfaces or implementations.
- Rather than composing interfaces or implementations, structural object patterns describe ways to compose objects to realize new functionality.
  - The added flexibility of object composition comes from the ability to change the composition at run-time, which is impossible with static class/interface composition.

Georgia Tech

Adapter (A.K.A Wrapper)

Intent: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure



Participants

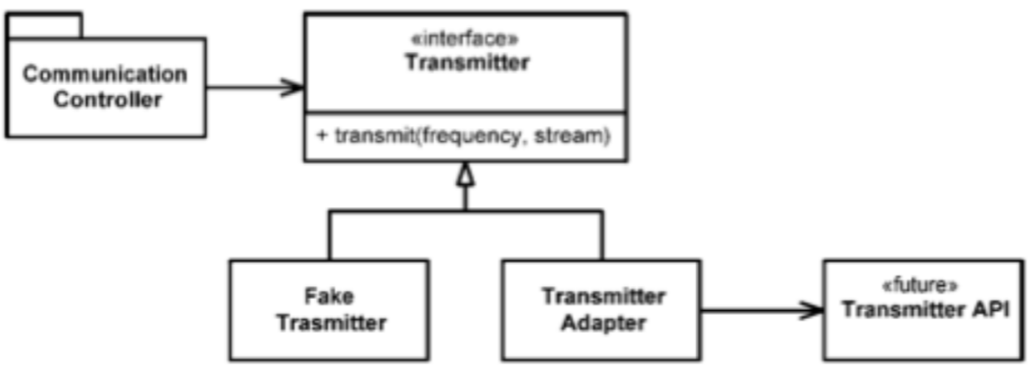
- **Target** defines the domain-specific interface that Client uses.
- **Client** collaborates with objects conforming to the Target interface.
- **Adaptee** defines an existing interface that needs adapting.
- **Adapter** adapts the interface of Adaptec to the Target interface.

Georgia Tech

Adapter Example

Imagine we're a team writing an application that will use a hardware transmitter, but the transmitter's software is handled by another team that hasn't defined their software interface.

- We can define our own interface the way we want it to work.
- While we're waiting for the transmitter team, we create a fake implementation to work with.
- When the transmitter team finally gives us their interface, we can write an adapter to fit it to our interface.
- The rest of our code is unaffected.

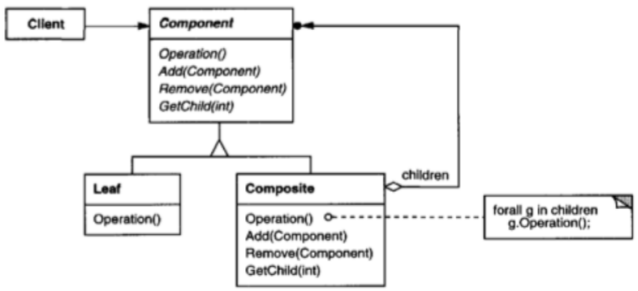


Georgia Tech

Composite

Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure



Participants

- **Component** declares the interface for objects in the composition.
- **Leaf** represents leaf objects (objects with no children).
- **Composite** defines behavior for components having children; stores child components; implements child-related operations.
- **Client** manipulates objects in the composition through the Component interface.

Composite Example: Dive Log (1 of 3)

Say we have a dive log with individual dives. Both represent the concept of dive experience, so we can represent this concept abstractly which allows us to get reports of dive experience in a uniform way whether we have a single dive or a log of several dives:

```
1 public interface DiveExperience {
2     public Date getDateTimeBegin();
3     public Date getDateTimeEnd();
4     public int getMaxDepthFeet();
5     public int getBottomTimeMinutes();
6 }
```

This interface plays the Component role in the composite pattern.

Composite Example: Dive Log (2 of 3)

Dive plays the Leaf role:

```
1  public class Dive implements DiveExperience, Comparable<Dive> {
2      private Date dateTimeBegin, dateTimeEnd;
3      private int maxDepthFeet, bottomTimeMinutes;
4      public Dive(Date dateTimeBegin, Date dateTimeEnd,
5                  int maxDepthFeet, int bottomTimeMinutes) {
6          this.dateTimeBegin = dateTimeBegin;
7          // ...
8      }
9      public Date getDateTimeBegin() **return dateTimeBegin; **
10     public Date getDateTimeEnd() ** return dateTimeEnd; **
11     public int getMaxDepthFeet() ** return maxDepthFeet; **
12     public int getBottomTimeMinutes() ** return bottomTimeMinutes; **
13     public int compareTo(Dive other) {
14         return this.getDateTimeBegin().
15                 compareTo(other.getDateTimeBegin());
16     }
17     public boolean equals(Object other) ** .. **
18     public int hashCode() ** ... **
19 }
```

Composite Example: Dive Log (3 of 3)

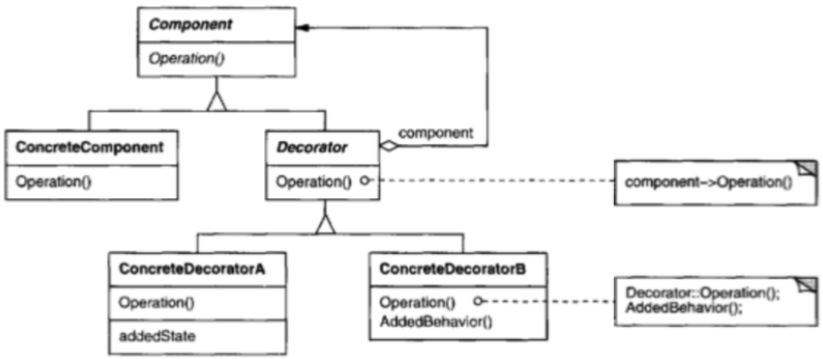
And DiveLog plays the Composite role:

```
1  public class DiveLog implements DiveExperience {
2      private TreeSet<Dive> dives = new TreeSet<Dive>();
3      private int maxDepthFeet = 0;
4      public void add(Dive dive) {
5          dives.add(dive);
6          if (dive.getMaxDepthFeet() > maxDepthFeet)
7              maxDepthFeet = dive.getMaxDepthFeet();
8      }
9      public Date getDateTimeBegin() {
10         return dives.first().getDateTimeBegin();
11     }
12     public Date getDateTimeEnd() {
13         return dives.last().getDateTimeEnd();
14     }
15     public int getMaxDepthFeet() { return maxDepthFeet; }
16     public int getBottomTimeMinutes() {
17         int sum = 0;
18         for (Dive dive: dives) {
19             sum += dive.getBottomTimeMinutes();
20         }
21         return sum;
22     }
23 }
```

Decorator

Intent: Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

Structure



Participants

- **Component** defines the interface for objects that can have responsibilities added to them dynamically.
- **ConcreteComponent** defines an object to which additional responsibilities can be attached.
- **Decorator** maintains a reference to a Component object and defines an interface that conforms to Component's interface.

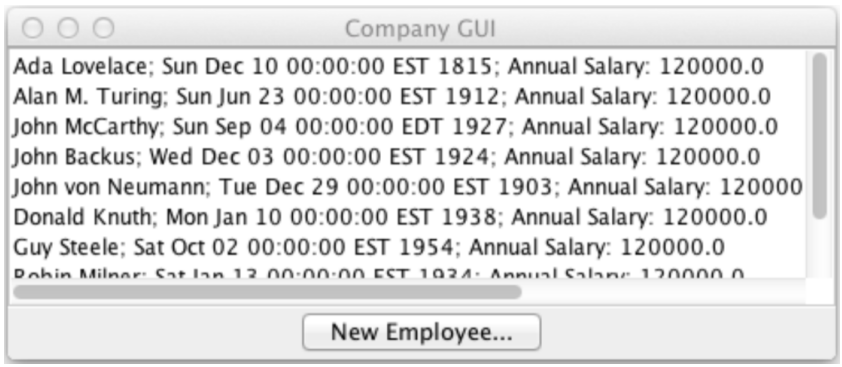
Georgia Tech

Decorator Example: JScrollPane

The Swing library provides a scrollbar decorator called `JScrollPane`. Using it is easy:

```
1  add(new JScrollPane(new JList(...)));
```

By simply wrapping our `JList` in a `JScrollPane` the list will show horizontal and vertical scroll bars as needed.



We've extended the functionality of a `JList` without having to subclass it.

Behavioral Design Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects. These patterns characterize complex control flow that's difficult to follow at run-time. They shift your focus away from flow of control to let you concentrate just on the way objects are interconnected.

- Behavioral class patterns use inheritance to distribute behavior between classes. (Template Method)
- The Strategy (315) pattern encapsulates an algorithm in an object. Strategy makes it easy to specify and change the algorithm an object uses.

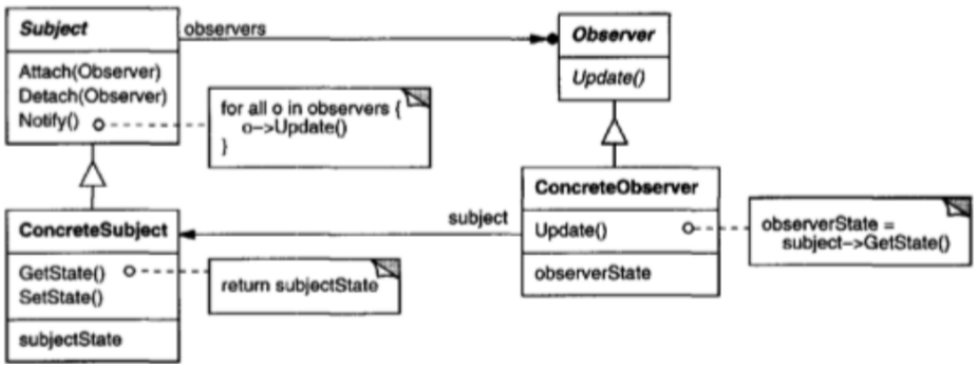
Behavioral object patterns use object composition rather than inheritance.

Georgia Tech

Observer (a.k.a. Dependents, Publish-Subscribe)

Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Structure



Participants

- **Subject** knows its observers.
- **Observer** defines a notification interface for objects that should be notified of changes in a subject.
- **ConcreteSubject** sends a notification to its observers when its state changes.
- **ConcreteObserver** implements Observer notification interface.

Observer Example: Swing Buttons

`javax.swing.AbstractButton` is a Subject, `javax.swing.JButton` is a ConcreteSubject. We set up an exit button like this:

```
1  JButton exitButton = new JButton("Exit");
2  exitButton.addActionListener(new ExitListener());
```

JButton's `addActionListener` method takes an object that implements the java.awt.event.ActionListener interface:

```
1  public interface ActionListener extends EventListener {
2      /**
3       * Invoked when an action occurs.
4      */
5      public void actionPerformed(ActionEvent e);
6  }
```

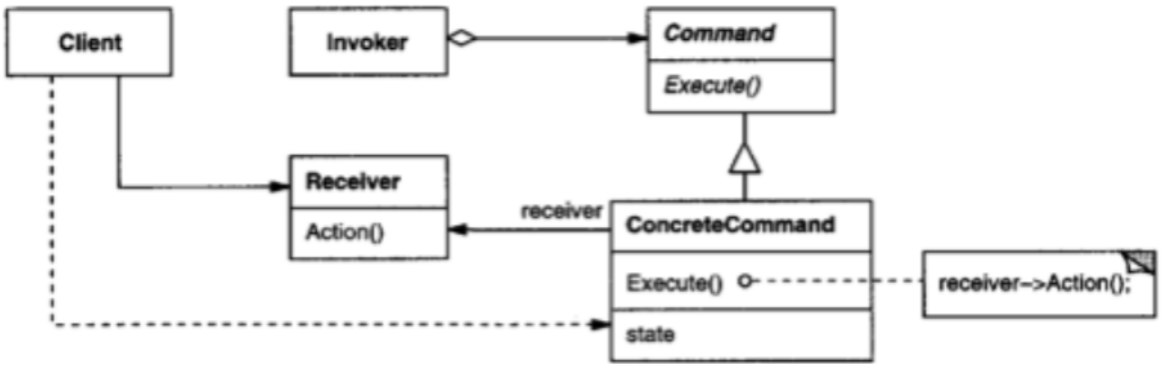
`java.awt.event.ActionListener` is an Observer, and `ExitListener` is a ConcreteObserver.

Georgia Tech

Command (a.k.a. Action, Transaction)

Intent: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Structure



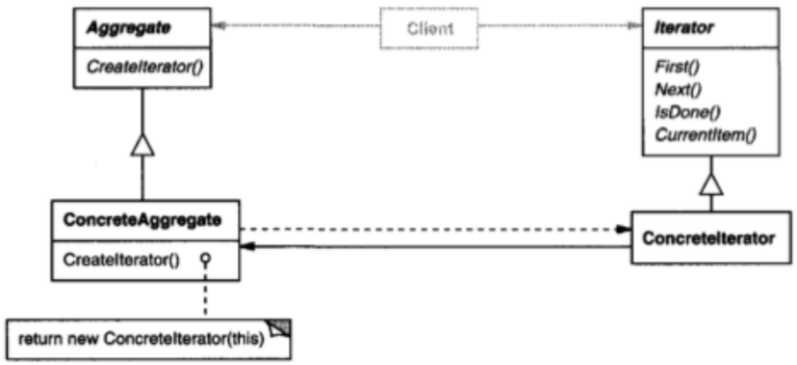
Participants

- **Command** declares an interface for executing an operation.
- **ConcreteCommand** defines a binding between a Receiver object and an action; implements Execute by invoking the corresponding operation(s) on Receiver.
- **Client** creates a ConcreteCommand object and sets its receiver. Invoker asks the command to carry out the request.

Iterator (a.k.a. Cursor)

Intent: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Structure



Participants

- **Iterator** defines an interface for traversing elements.
- **ConcreteIterator** implements the Iterator interface; keeps track of the current position in the traversal of the aggregate.
- **Aggregate** defines an interface for creating an Iterator object.
- **ConcreteAggregate** implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

Georgia Tech

Iterator Example: BST Traversal (1 of 2)

Binary tree implemented as linked nodes:

```
1  public class BinaryTree<E extends Comparable<E>> implements
2      Iterable<E> {
3      private class Node<E> {
4          E item;
5          Node<E> left;
6          Node<E> right;
7          Node(E item, Node<E> left, Node<E> right) {
8              this.item = item;
9              this.left = left;
10             this.right = right;
11     }
12 }
13 ...
14     private Node<E> root;
15 ...
```

We'd like to allow clients to traverse a BST in a uniform way whether traversing in-order, pre-order, or post-order.

Iterator Example: BST Traversal (2 of 2)

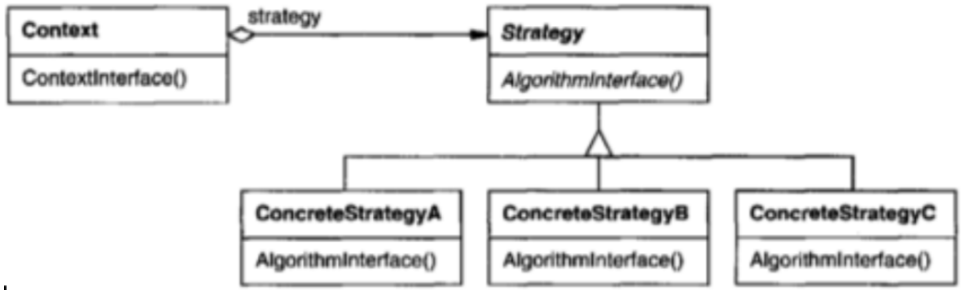
`java.util.Iterator` interface provides a uniform way to traverse all Java collections. Here's an implementation for BST:

```
1  private class InOrder<E> implements Iterator<E> {
2    private Node<E> curNode;
3    private Stack<Node<E>> fringe;
4    public InOrder(Node<E> root) {
5      curNode = root;
6      fringe = new LinkedStack<>();
7    }
8    public boolean hasNext() { ... }
9    public E next() {
10     while (curNode != null) {
11       fringe.push(curNode);
12       curNode = curNode.left;
13     }
14     curNode = fringe.pop();
15     E item = curNode.item;
16     curNode = curNode.right;
17     return item;
18   }
19   public void remove() { throw new UnsupportedOperationException(); }
20 }
```

Strategy (a.k.a. Policy)

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

Structure



Participants

- **Strategy** declares an interface common to all supported algorithms.
- **ConcreteStrategy** implements the algorithm using the Strategy interface.
- **Context** is configured with a ConcreteStrategy object; maintains a reference to a Strategy object; may define an interface that lets Strategy access its data.

Georgia Tech

Strategy Example: Repetitive Dives (1 of 4)

When we breath air at depth the increased pressure causes nitrogen to dissolve into body tissues. In SCUBA diving one must be mindful of resudual nitrogen in the body absorbed during a dive.

- On repetitive dives residual nitrogen limits the depth and time allowed on subsequent dives before decompression is required.
- The residual nitrogen in a diver's body is represented by a "pressure group" named by a single letter.
- There are many different ways to calcuate this pressure group: PADI's dive tables, NAUI's dive tables, the U.S. Navy dive tables, and so on.
- These tables different strategies for calculating pressure groups.

Georgia Tech

Strategy Example: Repetitive Dives (2 of 3)

We can represent the general **Strategy** for calculating pressure group ofr repetitive dives as an interface:

```
1  public interface DiveTable {
2      public void addDives(SortedSet<Dive> dives);
3      public String calculatePressureGroup();
4  }
```

The PADI table is an example of a **ConcreteStrategy**:

```
1  public class PadiDiveTable implements DiveTable {
2      private SortedSet<Dive> dives;
3      public void addDives(SortedSet<Dive> dives) {
4          this.dives = dives;
5      }
6      public String calculatePressureGroup() {
7        // calculate using 'PADIs dive table.
8      }
9  }
```

Georgia Tech

Strategy Example: Repetitive Dives (3 of 3)

The Context in which a DiveTable strategy is used is RepetitiveDives:

```
1  public class RepetitiveDives {
2      private TreeSet<Dive> dives = new TreeSet<Dive>();
3      public void add(Dive dive) {
4          dives.add(dive);
5      }
6      public String calculatePressureGroup(DiveTable diveTable) {
7          diveTable.addDives(dives);
8          return diveTable.calculatePressureGroup();
9      }
10 }
```

And if we have an instance of RepetitiveDives we can calucate the ending pressure group with any concrete strategy:

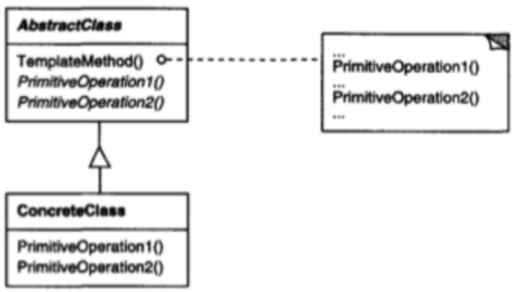
```
1  repetitiveDives.calculatePressureGroup(new PadiDiveTable());
2  // or
3  repetitiveDives.calculatePressureGroup(new NauiDiveTable());
4  // and so on ...
```

Georgia Tech

Template Method

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.

Structure



Participants

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

Georgia Tech

Template Method Example: Q Learning Agent

```
1  class TabularQLearner[WS, MS, A] extends Learner ... {
2    override def observe(worldState: WS, action: A, worldNextState: WS) ={
3      observe(worldState, action, worldNextState)
4      val state: MS = moduleState(worldState)
5      val nextState: MS = moduleState(worldNextState)
6      val r = reward(nextState)
7      val maxAction = calcMaxAction(nextState)
8      val newVal = q((state, action)) + alpha *
9        (r + gamma * q((nextState, maxAction)) - q((state, action)))
10     q += ((state, action) -> newVal)
11     r
12   }
13 }
```

`observe` is a template method, calling the primitive `moduleState` and `reward` methods defined in a subclass.

```
1  class FindGoal extends TabularQLearner[ ... ] {
2    def moduleState(ws: WumpusState) = FindGoalState(ws.wumpus, ws.goal)
3    def reward(ms: FindGoalState) =
4      if (ms.wumpus == ms.goal) 1.0 else -0.4
5  }
```

Georgia Tech

Closing Thoughts

Design patterns

- promote loose coupling and high cohesion
- identify and encapsulate points of change in a system
- promote good general OO design guidance
  - program to interfaces, not implementations
  - favor composition over inheritance
- Creational patterns abstract the instantiation process.
- Structural patterns are concerned with how classes and objects are composed to form larger structures.
- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.

Georgia Tech