What are Design Patterns and Why are they useful?

What - Design Patterns are abstractions of commonly occurring structures and behaviors in the real world.

Why - Object-oriented development generates software systems which model the "real world."

- Design Patterns are template for units of development which are larger than individual objects.

- Recognition of a Design Pattern creates a substantial unit of design which has been proven to work in similar circumstances in the past.

Example - Relationship among clients and professional service providers:

doctor - patient

account - client

lawyer - client, etc.

Definition - A "pattern" describes a commonly occurring problem and a generic solution which can be reused or modified and reused.
- **Typically an object model or a fragment of an object model**
- **Specifies objects and interactions among objects.**

Motivation - Most application programs can be designed as a composition of instances of frequently occurring patterns.

What you need to learn - How to recognize the occurrence of an instance of a pattern so that you will know to look for an existing solution.

Relationship between "patterns" and "frameworks" - Frameworks are usually implementations of patterns or a set of patterns in some language. For example, he java collections framework for storing and accessing objects or the Java AWT framework for creating GUIs.

Topic Coverage

1. Techniques for Design
    - Factorization and Generalization
2. Mechanisms for Design
    - Inheritance and Delegation
3. Design Patterns
    - What?
    - Types:
        - Creational Patterns
            - Singleton
            - Factory
        - Structural Patterns
            - Iterator
            - Associations
        - Behavioral Patterns
            - Strategy
            - Template
4. Sorting Example

Techniques for Design

Factorization

Identification of code segments which implement a common logical pattern in one or more classes or programs.

Capture this code logic in a generic representation

Reuse this generic representation of this code logic to replace the multiple occurrences in the programs and/r classes.

Generalization

Analyze solutions to one or more problems and create a single solution which solves multiple problems or multiple variations of the original problems.

Factorization

```
Class Computation{
  void method1 () {
     ComputeStep1();
     ComputeStep2();
     ComputeStep3();
      …………..
  }
void method2 () {
     ComputeStep1();
     ComputeStep2();
     ComputeStep3();
      …………..
  }
………….
}
```

```
Class FactorizedComputation{
  void computeAll () {
     ComputeStep1();
     ComputeStep2();
     ComputeStep3();
      …………..
  }
void method1 () {
     ComputeAll();
      …………..
  }
void method2 () {
     ComputeAll();
      …………..
  }
………….
}
```


Mechanisms for Factorization and Generalization

Inheritance - Put common methods in superclass and have multiple subclasses inheriting the common behavior.

```
class Common{
  void computeAll() {
     ComputeStep1();
     ComputeStep2();
     ComputeStep 3();
      …………..
  }
}
```

```
Class ComputationA extends
Common{
  void method1 () {
     …………….
     computeAll();
       …………..
   }
}
```

```
Class ComputationB extends
Common{
  void method1 () {
     …………….
     computeAll();
       …………..
   }
}
```

Mechanisms for Factorization and Generalization - Continued

Delegation - Implement the common behavior in a separate class and create an instance of the separate class in each class which needs to utilize the common behavior.

```
class Helper{
  void computeAll() {
    ComputeStep1();
    ComputeStep2();
    ComputeStep3();
     …………..
  }
}
```

```
class ComputationA{
  Helper helper = new Helper()
   void compute () {
     …………….
     helper.computeAll();
       …………..
   }
}
```

```
class ComputationB{
  Helper helper = new Helper()
   void compute () {
     …………….
     helper.computeAll();
       …………..
   }
}
```


Template for Specification of a Pattern

Name - What is it.

Category - Creational, Structural or Behavioral

Intent - Describe the problem to which a solution is sought.

Applicability - Situations when the pattern will be used.

Structure - Usually an object diagram.

Implementation - Mechanisms used in defining and implementation of pattern.

Design Patterns and Reusable Components

Singleton - Example Creational Pattern

Name - Singleton

Category - Creation Pattern

Intent - Guarantee that only a single instance of a class is instantiated and used.

Applicability - Used when only a single instance of a class is to be instantiated and used.

Example - performance measurement - java.lang.Runtime

public class Runtime extends Object

Every Java application has a single instance of class Runtime that allows the application to interface with the environment in which the application is running. The current runtime can be obtained from the getRuntime method.

Design Patterns and Reusable Components

Singleton
static getInstance() operation() getData()
Static theInstance; **data;**

The participant is a single class which declares a single instance of the class as a static variable and defines a static method, getinstance to provide access to the single instance.

```
Public class Singleton
{   public static void Singleton getInstance()
    {if (theInstance==null)
        { theInstance = new Singleton(); }
     return theInstance;
    }

   protected Singleton()
     { /* initialize instance variables. */}

   /* Implementation of methods of class Singleton */

   private static Singleton theInstance = null;
}
```

Design Pattern - Iterator

Category - Behavioral (Structural) design pattern

Intent - Sequential access to elements of a collection of object instances.

Applicability - Is used :

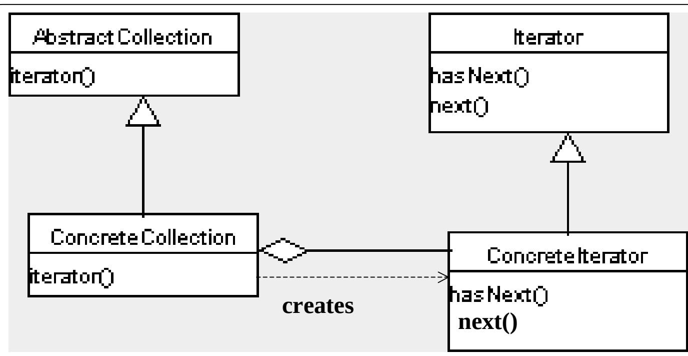
To access the contents of a collection without knowledge of how the object instances are stored.

To provide multiple types of traversals of a collection.

To provide a uniform interface for traversing different types of collections of different types of objects.

(make iteration polymorphic.)

An Iterator is a specialization of an Adapter design pattern. An adapter provides an interface known to its clients and provides access to instances of a class not necessarily known to its clients. An adaptor provides the functionality promised by an interface without having to assume what class is used to implement the interface.



Iterator - defines an interface for accessing and traversing the elements of a collection.
ConcreteIterator - (eg. LinkedListInterator) which implements the Iterator interface.
AbstractCollection - (e.g. List) which defines an interface for creating a concrete iterator.
ConcreteCollection - implements the iterator() method to return an instance of a proper Iterator.

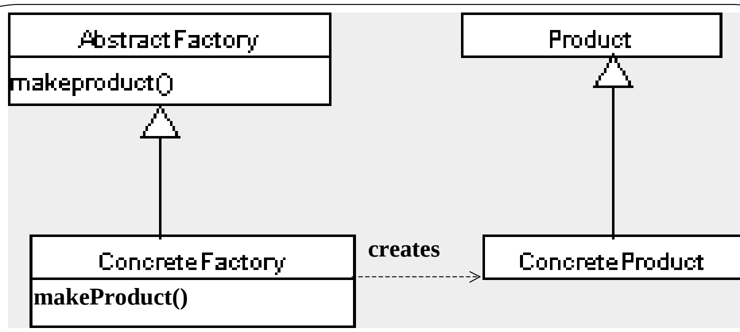
Design Pattern - Factory

Category - Creational Design Pattern

Intent - Define an interface for creating objects but let subclasses determine which class to instantiate and how.

Application - When the same process is to be applied to creation of objects of different classes.

Example: create a family of concrete sorting algorithms



Product - (e.g. SortAlgorithm) defines an interface for the objects the factory will create.

ConcreteProduct - (e.g. QuickSortAlgorithm) implements the Product interface.

AbstractFactory - (e.g. AlgorithmFactory) defines a factory method (makeSortAlgorithm) which returns an object of type Product.

ConcreteFactory - (e.g. StaticSortAlgorithm) implements and overrides the makeProduct method of AbstractFactory to create an instance of concreteProduct.

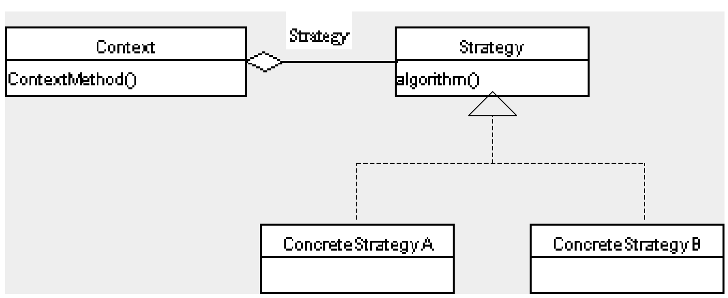
Design Pattern - Strategy

Category - Behavioral design pattern

Intent - Define a family of algorithm, encapsulate each one and make them interchangeable.

Applicability - The Strategy pattern should be used when:

- **Many related classes differ only in their behavior (e.g. plot different functions.)**
- **Different variants of an algorithm are needed (e.g. the sort algorithm in Section 5.3.)**
- **An algorithm uses data that the using clients should not know about. (The layout manager in AWT.)**
- **A class defines many behaviors with selection among the behaviors appearing as condition statements in its methods.**



Strategy - (Funtion) defines an interface common to all supported algorithms.

ConcreteStrategy - (Sine or Cosine) - implements the algorithm through the Strategy interface.

Context - (MultiPlotter) maintains references to one or more Strategy objects (e.g. functions in MultiPlotter)

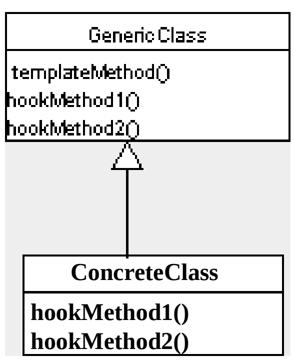
Design Pattern - Template

Category -Behavioral Design Pattern

Intent - Define the skeleton of an algorithm in a methodbut defer some steps of the algorithm to subclasses thus allowing for customization of the implementation of the algorithm.

Applicability -

- **Implement invariant parts of an algorithm once to minimize coding and maintenance efforts across a family of algorithms.**
- **Factorize common behaviors to avoid code duplication across subclasses.**



GenericClass (e.g. DBAnimationApplet) defines abstract hook methods (e.g. paitFrame()) that concrete classes (e.g. BouncingBall2) override to implement steps of an algorithm and implements a template method (e.g. update())that defines the skeleton of an algorithm by calling hook methods

ConcreteClass - (e.g. BouncingBall2) implements the hok methods (e.g. paintFrame()) to call out subclass specific steps of the algorithm defined in the template method.