Object Orientation with Design Patterns

Lecture 2:

Describing patterns

Abstract Factory

Describing Design Patterns

- In general a Pattern has **four** essential elements:
  - **Pattern Name**
    - This is a handle we can use to **describe a design problem**, its solutions, and consequences in a word or two. Naming a pattern immediately increases our design vocabulary.
  - **Problem**
    - The problem that the pattern is trying to solve.
  - **Solution**
    - How the pattern **provides a solution** to the problem in the context in which it shows up.
  - **Consequences**
    - This describes the **results and trade-offs** in using the pattern, i.e., if you implement this pattern how might it affect and be affected by the forces present.
    - This is critical for **evaluating design alternatives** and for understanding the **cost** and **benefits** of applying the pattern.
- One persons pattern can be another's building block !

Describing Design Patterns

- How can we describe design patterns?
  Graphical representations of design are **very useful (UML)** but often these graphical representations just show us an end product of the design process.

- In order to reuse the design pattern we must also **record** the **decisions, alternatives, and trade-offs** that led to it.

- **Concrete examples are important too because they help us see the design in action !**

- We can describe design patterns using a consistent format.

Describing Design Patterns

- Each pattern is divided into sections according to the following template:

  - **Intent – purpose of the pattern**
    - A short statement that answers the following questions:
      - What does the design pattern do?
      - What is its rational and intent?
      - What particular design issue or problem does it address?
  - **Also known as (AKA)**
    - Other well known names for the pattern, if any.
  - **Motivation**
    - A scenario that illustrates a design problem and how the class and objects in the pattern solve the problem.

Describing Design Patterns

- **Structure**
  - A graphical representation of the classes in the pattern – UML
- **Participants**
  - The classes and/or objects participating in the design pattern and their responsibilities
- **Implementation**
  - What pitfalls, hints, or techniques should you be aware of when implementing the pattern? (Language specific details)
- **Sample code**
  - Code fragments that illustrate how you might implement the pattern.

Describing Design Patterns

- **Known uses**
  - Examples of the pattern found in real systems.

- **Related patterns**
  - What design patterns are closely related to this one?
  - What are the important differences?

How to select a Design Pattern

- **Scan intent sections of design patterns**
  - Read through **design pattern intent sections** (potential value of applying pattern) to find one or more that sound relevant to your problem
- **Study how patterns interrelate**
  - By studying how patterns interrelate it can lead you to the **right pattern** or right group of patterns

How to select a Design Pattern

- **Study patterns of like purpose**
  - Creational, Behavioral, Structural
- **Examine a cause of redesign**
  - Look at patterns that help you avoid causes of redesign
- **Consider what should be variable in your design**
  - This is the opposite to the above, consider what **might force a change to the design**. Consider what you would **like to be able to change without redesign**

ABSTRACT FACTORY PATTERN

Abstract Factory

- The abstract factory pattern is **one level higher** then the **Factory pattern**.

- **INTENT –**
  You can use this pattern to return one of several related classes of objects, each of which can return several different objects on request.

- In other words, the Abstract Factory is a factory object that **returns one of several groups of classes.**

Abstract Factory

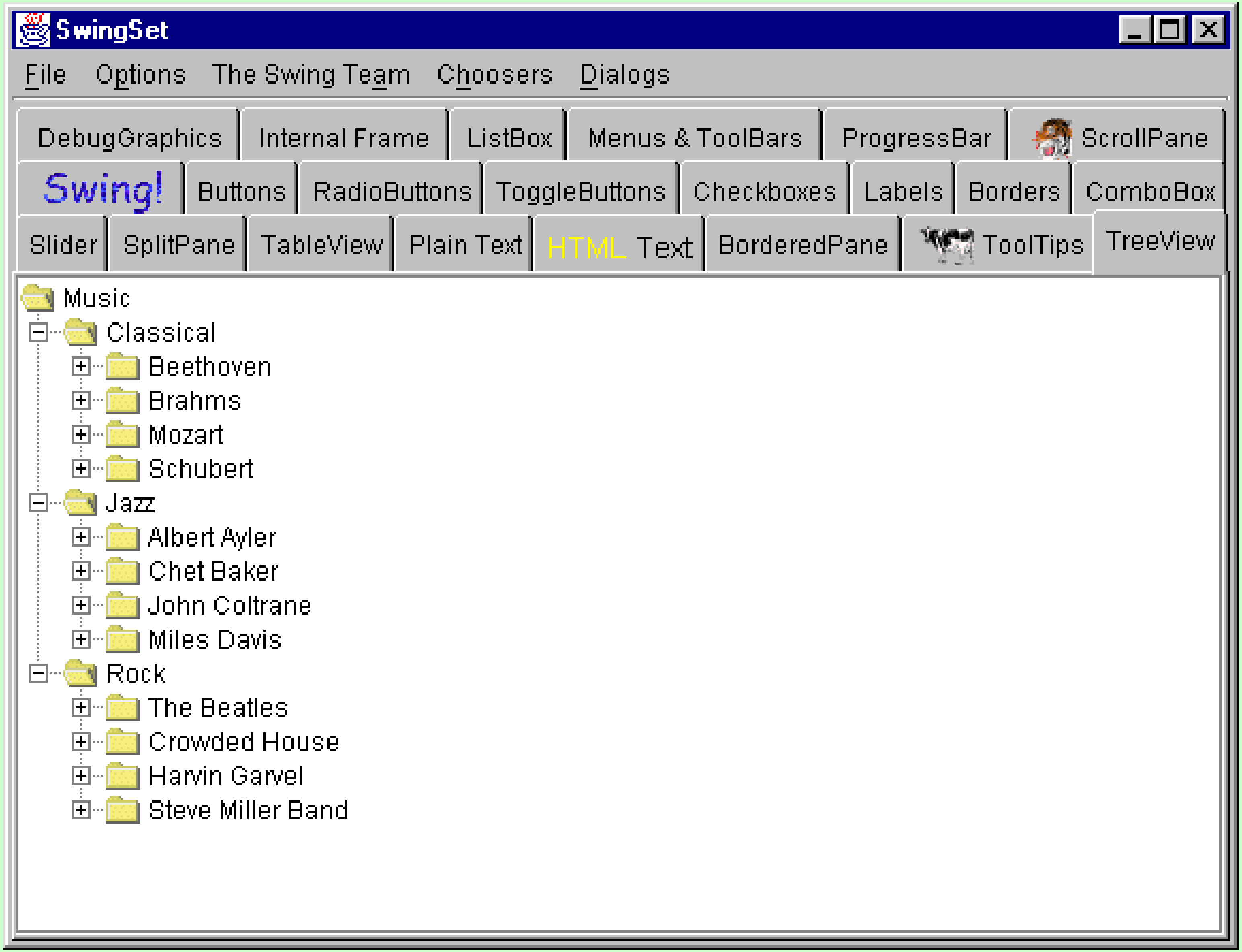
- One classic application of the abstract factory is when your application needs to support **multiple look-and-feel user interfaces**, such Windows x, Motif, and Macintosh. You tell the factory that you want your program to look like Windows, and it returns a GUI factory that returns Windows-like objects.

- Then you can request specific objects like buttons and scrollbars.

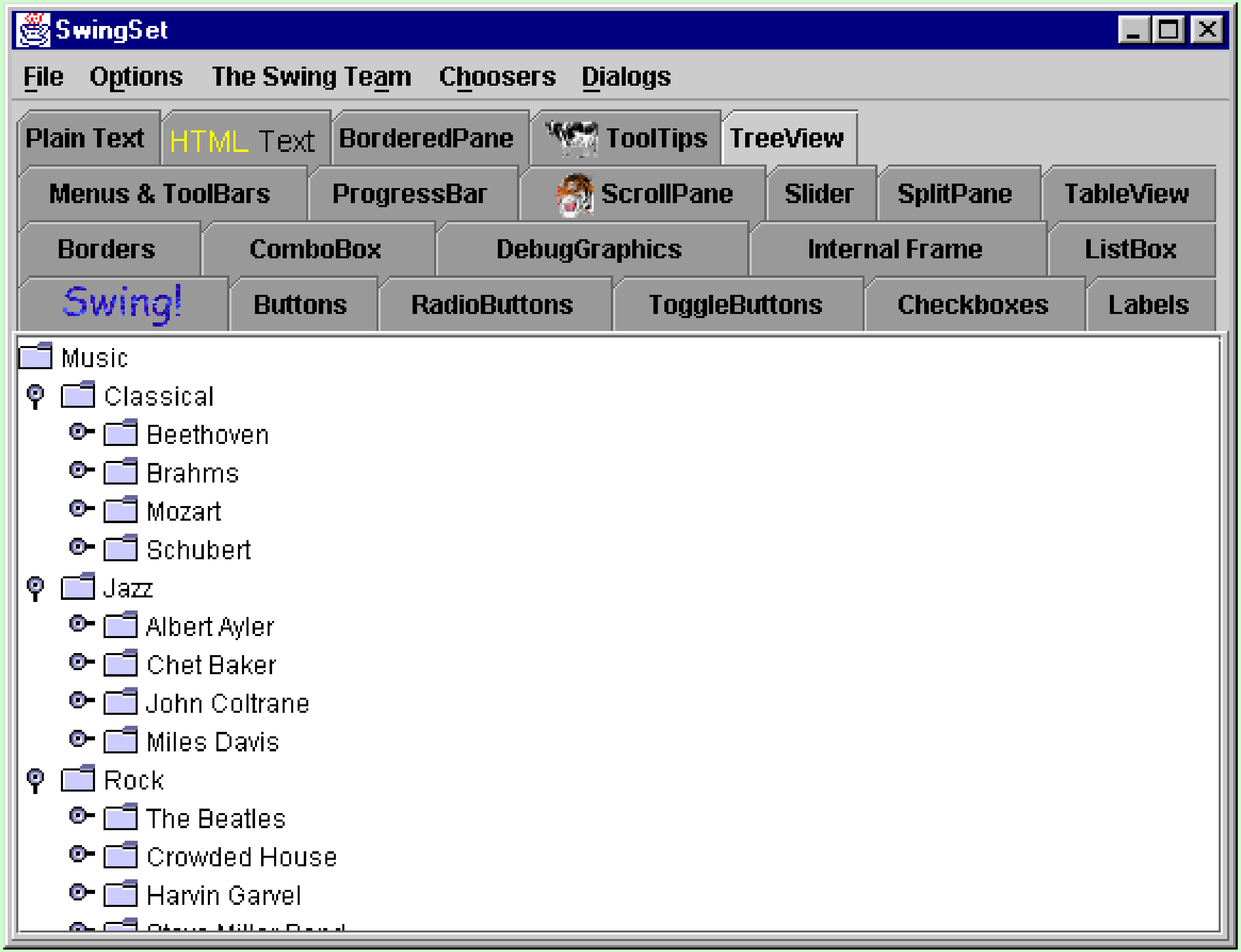
Abstract Factory

- In Java, the pluggable look-and-feel classes accomplish this at the system level so that **instances of the visual interface components are returned** correctly once the program selects the type of look-and-feel.
- For example the following slides show the **Java Swing demo interface** using **Windows**, **Java**, and **Motif look-and-feel.**

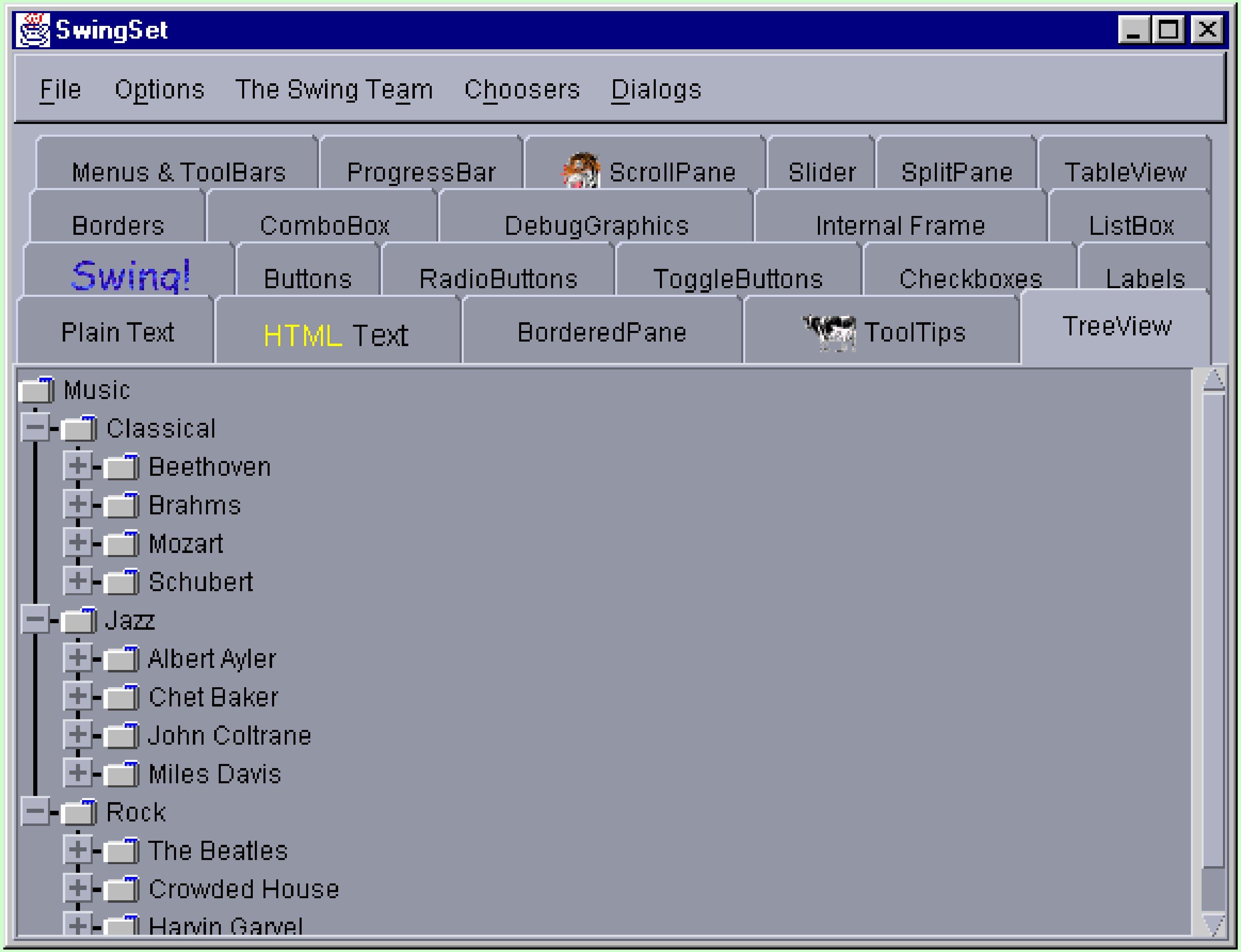
Abstract Factory



Abstract Factory



Abstract Factory



Abstract Factory

- In the following code sample we find the name of the current windowing system and then tell the pluggable look-and-feel (PLAF) Abstract Factory to generate the correct objects.

```java
// Getting and setting the current PLAF

String laf = UIManager.getSystemLookAndFeelClassName();

try
{
    UIManager.setLookAndFeel(laf);
}
catch(UnsupportedLookAndFeelException e)
{
    System.err.println("Unsupported L&F: " + laf);
}
catch(Exception e)
{
    System.err.println("Error loading " + laf);
}
```

A GardenMaker Factory

- Lets consider a simple example where you might want to use the Abstract factory in your application.

- Suppose you are writing a program to **plan the layout of gardens**. These could be **annual gardens, vegetable gardens, or perennial gardens**. No matter what the type of garden we still want to ask the same questions:
  - What are good center plants?
  - What are good border plants?
  - What plants do well in partial shade?

- And probably many more plant questions that we will ignore in this simple example.

A GardenMaker Factory

- We can create an abstract class Garden which can answer these questions.

```java
public abstract class Garden {
    public abstract Plant getShade();
    public abstract Plant getCenter();
    public abstract Plant getBorder();
}
```

- In this case the Plant class will just contain and return a plant name.

```java
public class Plant {
    private String name;
    public Plant(String pname) {
        name = pname;
    }
    public String getName() {
        return name;
    }
}
```

A GardenMaker Factory

- In design pattern terms the **Garden class** is the **Abstract Factory**.
  It defines the methods of a concrete class that can **return one of several classes, in this case one each for center, border, and shade-loving plants.**

- In a real system, for each type of garden we would probably consult a database of plant information. In this example, we'll return one kind of plant from each category. So for example, for the vegetable garden we write the following:

A GardenMaker Factory

```
public class VeggieGarden extends Garden {
    public Plant getShade() {
        return new Plant("Broccoli");
    }
    public Plant getCenter() {
        return new Plant("Corn");
    }
    public Plant getBorder() {
        return new Plant("Peas");
    }

}
```

- Similarly we can create garden classes for PerennialGarden and AnnualGarden. **Each of these concrete classes is a *Concrete Factory***, since it implements the methods outlined in the parent abstract class.

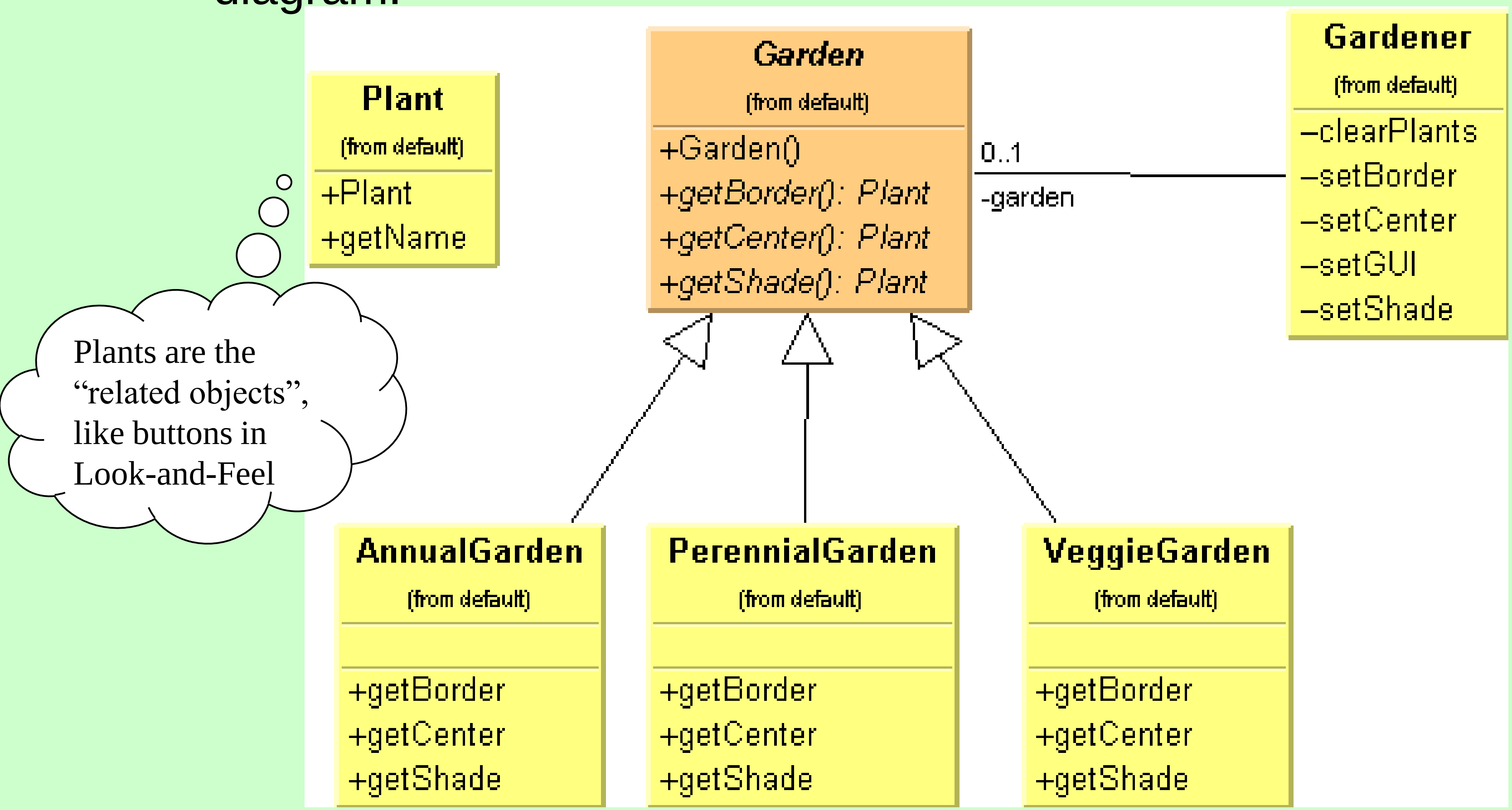
A GardenMaker Factory

```
public class AnnualGarden extends Garden {
    public Plant getShade() {
        return new Plant("Coleus");
    }
    public Plant getCenter() {
        return new Plant("Marigold");
    }
    public Plant getBorder() {
        return new Plant("Alyssum");
    }
}

public class PerennialGarden extends Garden {
    public Plant getShade() {
        return new Plant("Astilbe");
    }
    public Plant getCenter() {
        return new Plant("Dicentrum");
    }
    public Plant getBorder() {
        return new Plant("Sedum");
    }
}
```

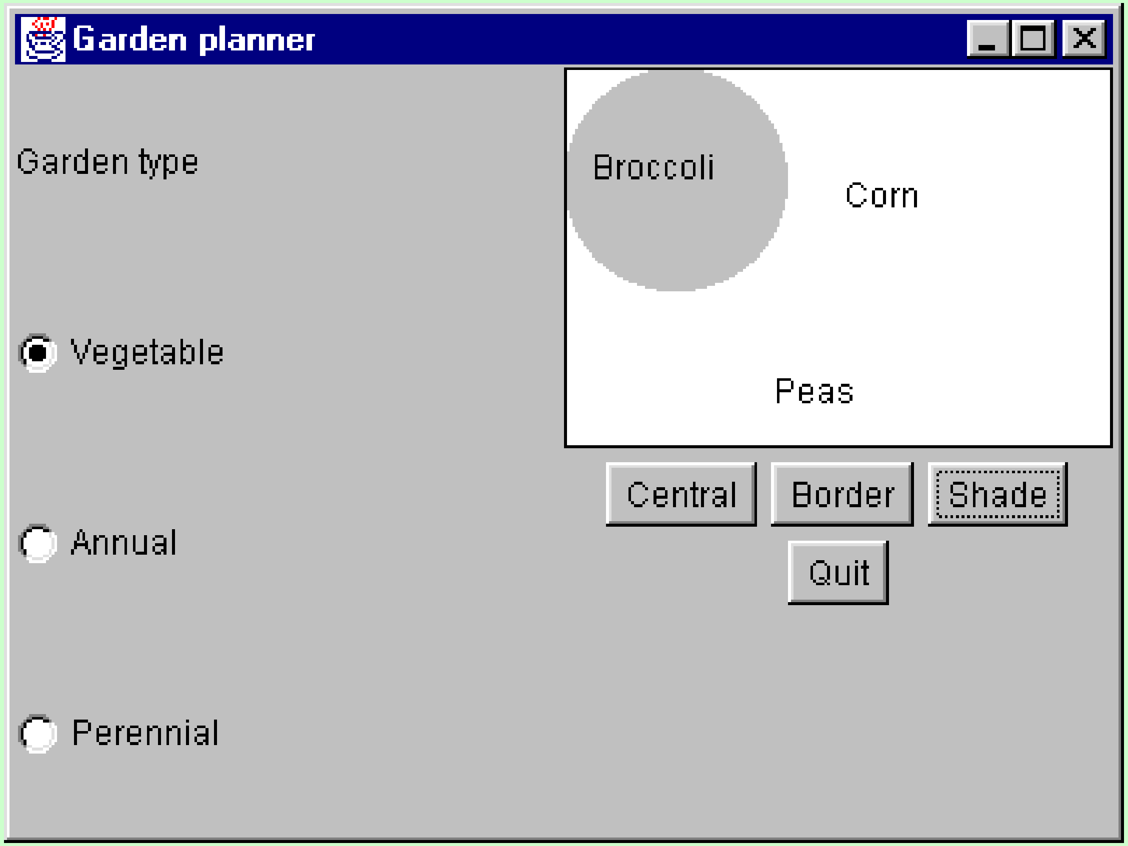
A GardenMaker Factory

- Now we have a series of garden objects, each of which returns one of several Plant objects. This is illustrated in the following class diagram.



A GardenMaker Factory

- We can easily construct a driver program Gardener to return one of these garden objects based on a radio button selection as shown in the following screen shot:



How the user interface works

- The simple interface consists of **two parts**: **the left side, which selects the garden type**, and the **right side, which selects the plant category**. When you click on one of the garden types, this causes the program to return a type of garden that depends on which button you select.

- At first, you might think that we would need to **perform some sort of test to decide which button was selected** and then instantiate the right Concrete Factory class.

- However, a more elegant solution is to **create a different ItemListener for each radio button *as an inner class*** and have each one create a different garden type.

How the user interface works

- First we create the instances of each Listener class.

```
Veggie.addItemListener(new VeggieListener());
Peren.addItemListener(new PerenListener());
Annual.addItemListener(new AnnualListener());
```

- Then we define the actual inner classes, as shown on the next slide.

How the user interface works

```java
//------------------------------
class VeggieListener implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        garden = new VeggieGarden();
        clearPlants();
    }
}
//------------------------------
class PerenListener implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        garden = new PerennialGarden();
        clearPlants();
    }
}
//------------------------------
class AnnualListener implements ItemListener {
    public void itemStateChanged(ItemEvent e) {
        garden = new AnnualGarden();
        clearPlants();
    }
}
```

How the user interface works

- Thus when the user clicks on one of the plant type buttons, the plant type is returned, and the name of the plant is displayed.

```
public void actionPerformed(ActionEvent e) {
    Object obj = e.getSource();
    if (obj == Center)
        setCenter();
    if (obj == Border)
        setBorder();
    if (obj == Shade)
        setShade();
    if (obj == Quit)
        System.exit(0);
}
```

```
private void setCenter() {
    if (garden != null)
        centerPlant = garden.getCenter().getName();

    gardenPlot.repaint();
}
```

The Garden Abstract Factory implemented without inheritance?

- It is possible to implement this pattern in Java without using the keyword **extends** (between the Abstract and concrete factories)

- Java does not implement true multiple inheritance but does allow multiple interface inheritance (i.e. you inherit the behaviours as specified by the interface, but the interface methods are abstract)

- Keeping this in mind, any class which implements an interface has an “is-a” relationship with the interface that has been implemented (think about addActionListener(this) which only accepts ActionListener as a parameter)

- Therefore it is possible to create an **interface** of type **Garden** which specifies the three methods required, and instead of the using **extends** to inherit from the abstract class use **implements**

Abstract Factory Consequences

- **Isolates the concrete classes** that are generated
- **Actual names of classes are hidden in the factory** and need not be known by the client.
- Because of isolation you can **change or interchange** these **product class families freely**.
- While all of the classes that the Abstract Factory pattern generates have the same base class there is nothing stopping you from creating subclasses that have additional methods. For example, a BonsaiGarden class might have a *Height* or *WateringFrequency* method that is not present in the other classes.

QUIZ

- A car factory contains concrete factories such as Honda, BMW. Each concrete factory has different types of cars, eg family car, sports car, estate car.
- The cars have different attributes such as price, engine size, no.of doors, etc.
- Create a UML diagram to implement the abstract factory pattern using the details above.