Week 7 - Monday

COMP 3100

Last time

- What did we talk about last time?
- Graphic design
- Software engineering design
- Architectural styles

Questions?

Quick Notes on Project Scheduling

Project scheduling

- Project scheduling is organizing the work
  - Into separate tasks
  - When the tasks will be done
  - Who will do them
- Both waterfall and agile approaches benefit from scheduling
  - For waterfall, all tasks in the project are scheduled
  - For agile, there might be an overall schedule for when major phases of the project will be completed
- Tasks should last at least a week but not more than two months
  - A task taking more than two months should be broken into subtasks
- It's helpful to have visualizations of these tasks

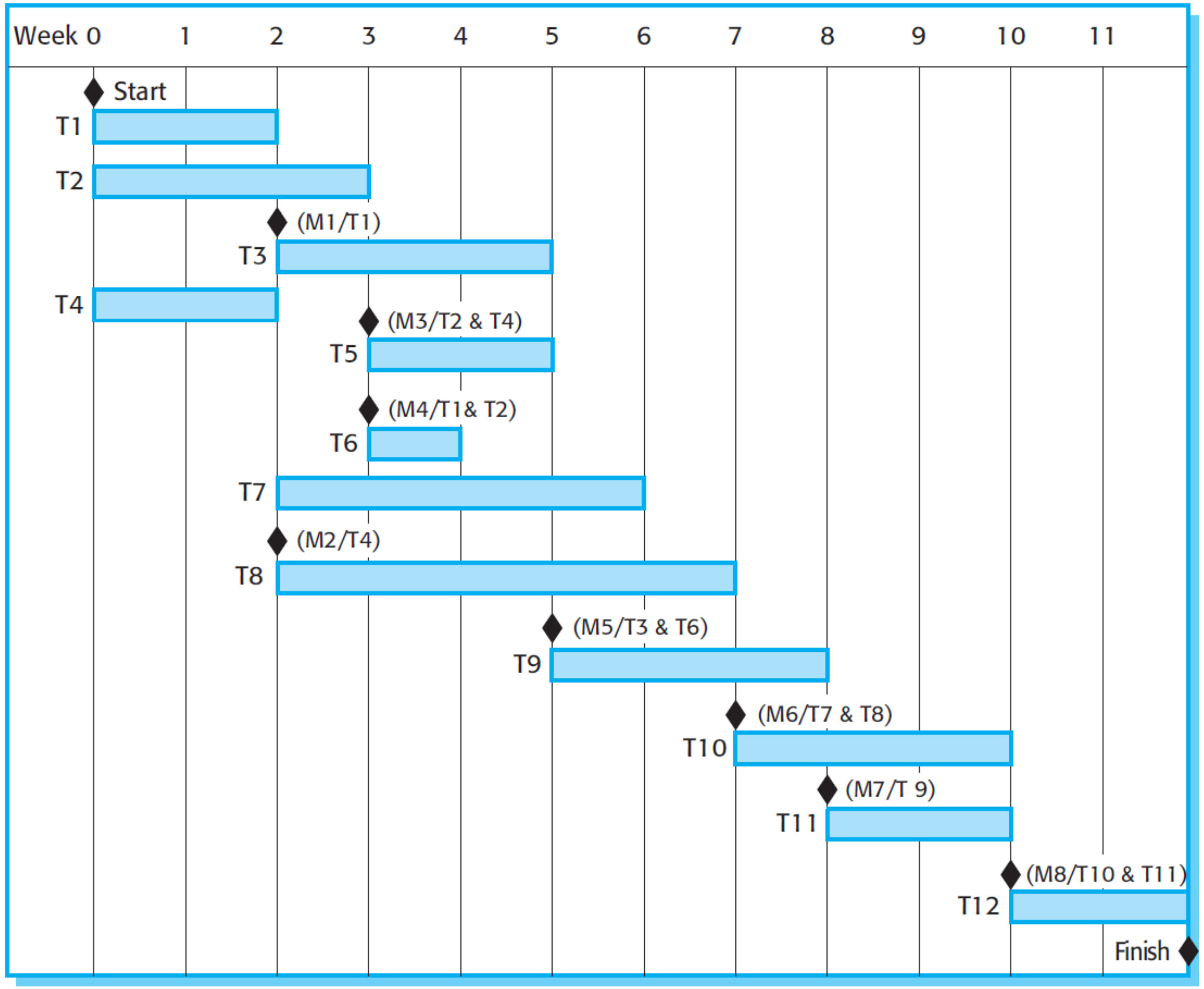
Example of tasks

- This table shows all the task information, but it's hard to visualize
- M is used to label milestones

Task	Effort (person-days)	Duration (days)	Dependencies
T1	15	10	
T2	8	15	
T3	20	15	T1 (M1)
T4	5	10	
T5	5	10	T2, T4 (M3)
T6	10	5	T1, T2 (M4)
T7	25	20	T1 (M1)
T8	75	25	T4 (M2)
T9	10	15	T3, T6 (M5)
T10	20	15	T7, T8 (M6)
T11	10	10	T9 (M7)
T12	20	10	T10, T11 (M8)

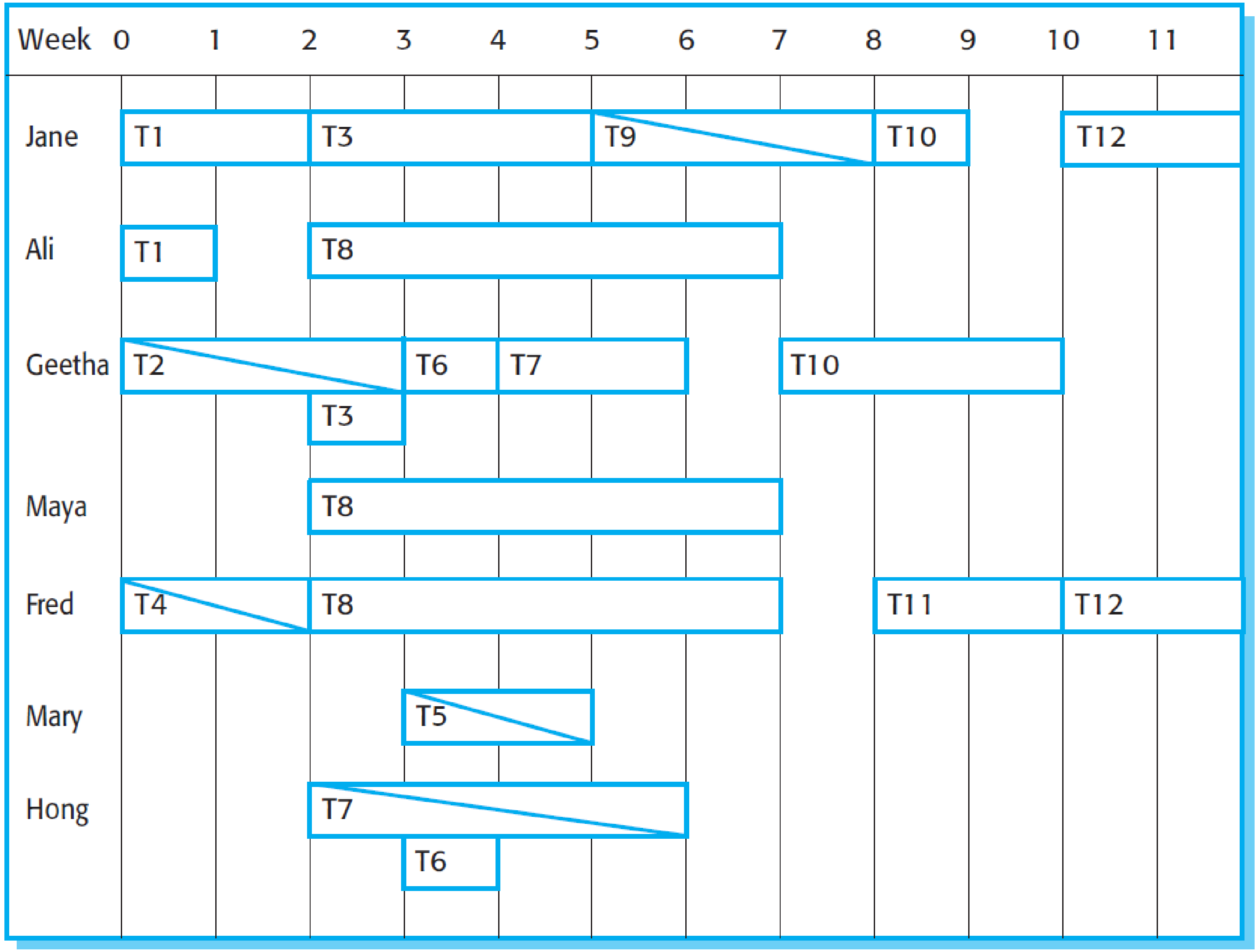
Gantt charts

- Gantt charts show the same information, but in a much clearer way
  - Bars shows the length of each task
  - Dependencies are shown by the starting point of each task
- Recall that you have to make a Gantt chart for Project 2
- Thus, you need to break down your product into tasks and figure out which tasks are dependent on which



Staff allocation

- It's also possible to visualize which staff members are working on which task (and when)
- Doing so might be helpful but is not required for Project 2



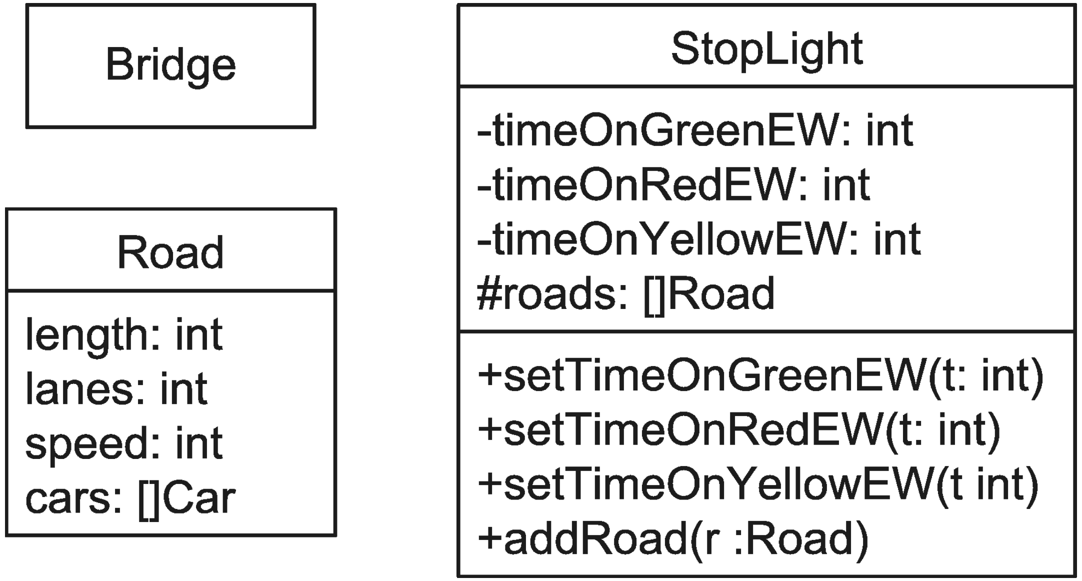
Detailed Design

Detailed design

- **Detailed design** is specifying the internals of the major components
- Sequence diagrams and state diagrams can be useful for this kind of design
- However, class diagrams are indispensable

More depth on class diagrams

- **Class diagrams** are made up of **class symbols** (rectangles)
- These class symbols contain one or more **compartments**
- The top compartment has the class name
- A second, optional compartment often contains attributes (called member variables in Java classes)
  - Often followed by a colon with the type
- A third, optional compartment often contains operations (called methods in Java classes)
  - Sometimes followed by parameter and return types
- Visibility modifiers can be marked:
  - **+** for public
  - **#** for protected
  - **~** for package
  - **-** for private
- Only important attributes and operations need to be specified
  - Classes might contain others that aren't shown



Bridge

Road

length: int
lanes: int
speed: int
cars: []Car

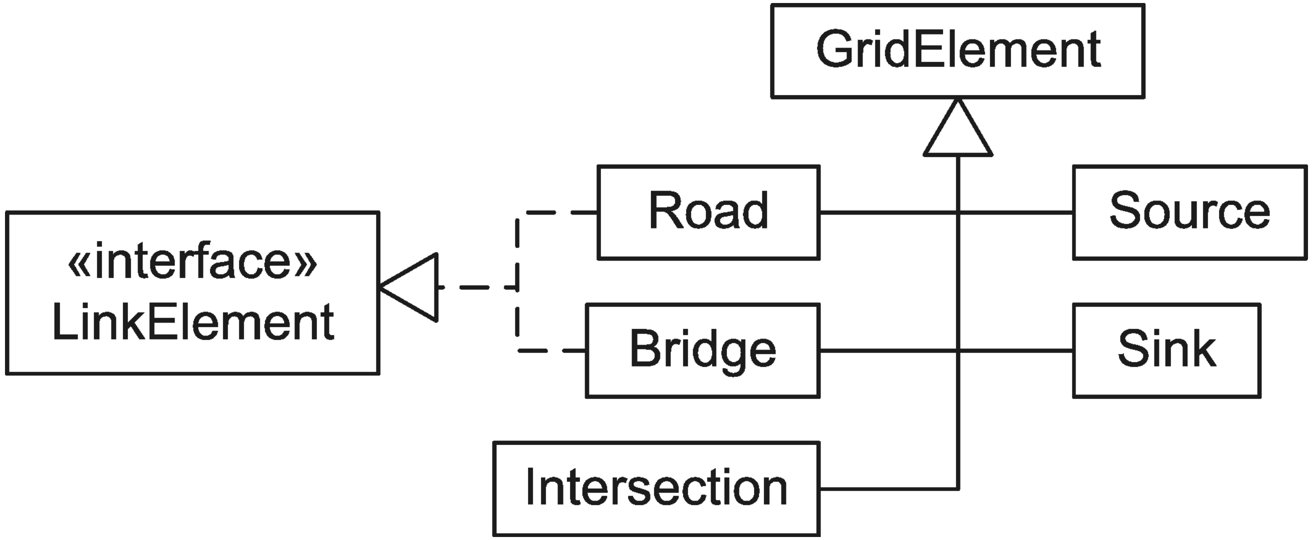
StopLight

-timeOnGreenEW: int
-timeOnRedEW: int
-timeOnYellowEW: int
#roads: []Road

+setTimeOnGreenEW(t: int)
+setTimeOnRedEW(t: int)
+setTimeOnYellowEW(t int)
+addRoad(r :Road)

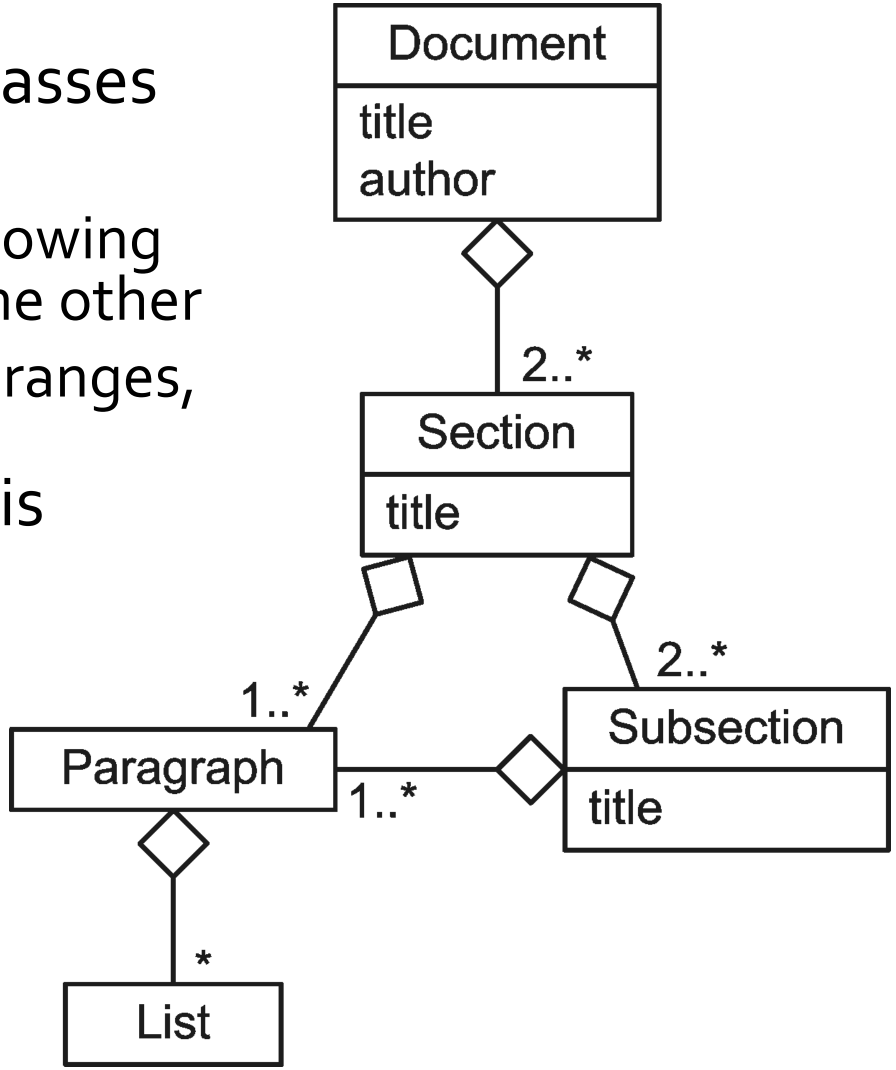
Inheritance and interfaces in class diagrams

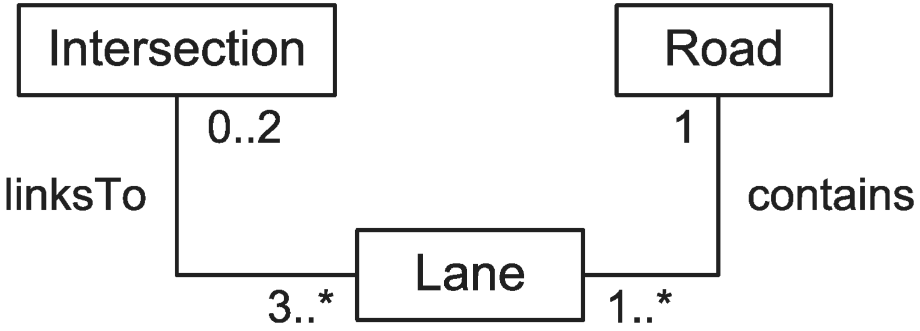
- Inheritance is shown with the **generalization** connector
  - A solid line from the child class to a solid triangle connected to the parent class
  - Confusingly, this means that children classes point at their parent classes
- Interfaces look like classes but are marked with **«interface»** above the class name
  - This kind of marking is called a **stereotype**
  - Stereotypes show extra information that wasn't part of the original UML class diagram specification
- Classes that implement interfaces have dashed lines leading to a solid triangle connected to the interface



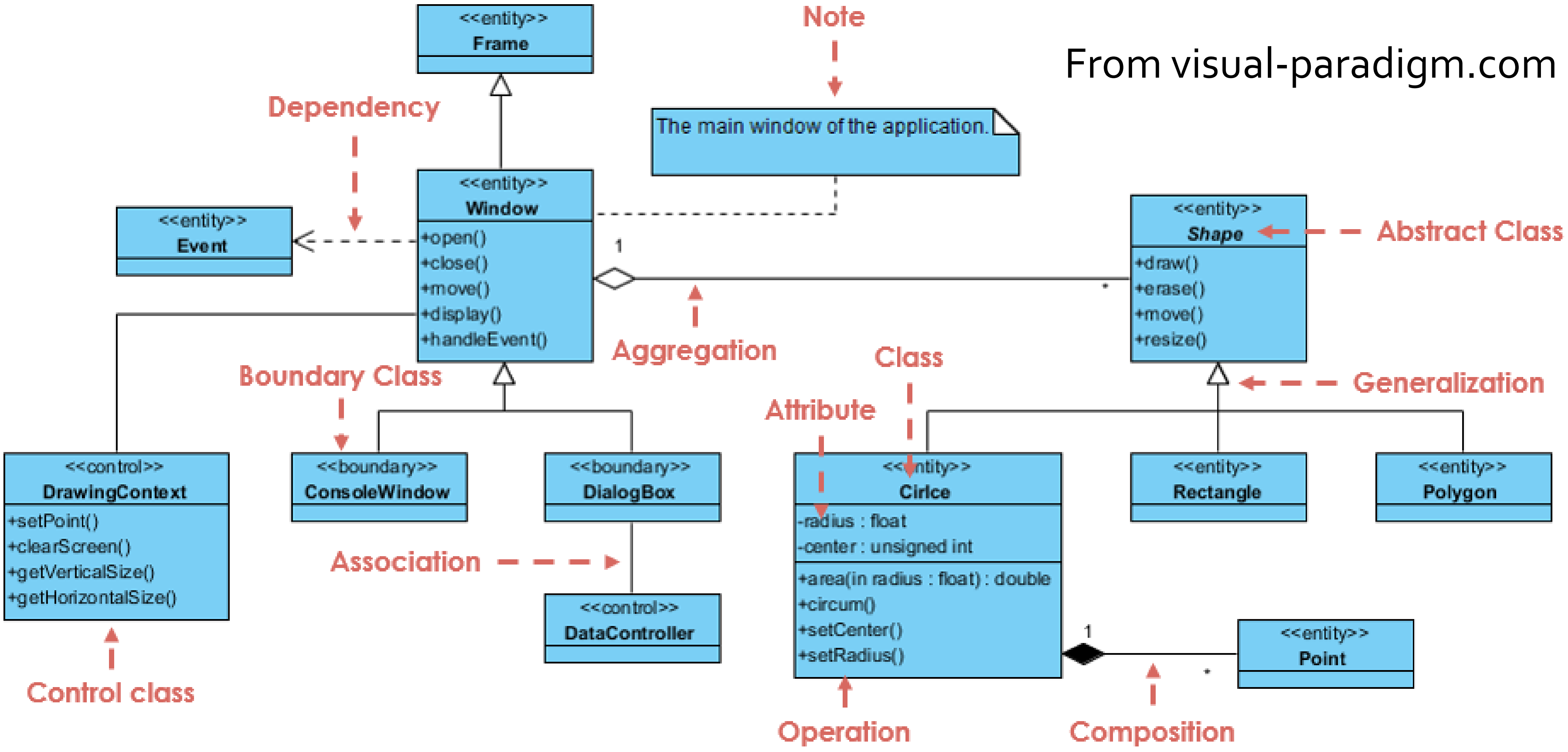
Other associations

- **Associations** are shown with lines between classes
  - Associations can be labeled to explain them
  - The lines can be marked with the **multiplicity**, showing how many of each class can be associated with the other
  - The multiplicity can be comma separated lists or ranges, and * means zero or more
- When a class is part of another class, the part is connected by a line and a diamond (the **aggregation** connection) to the whole





Complex example



From visual-paradigm.com

Design Patterns

Object class identification

- With an architecture designed, you can break down its components into the actual software objects you will need
  - There's no cookbook way to do this
  - It requires thinking long and hard about how best to break functionality into small pieces
- Possible approaches:
  - Look at the written description of your system. Nouns map to objects and members. Verbs map to operations, services, and methods.
  - Tangible entities map to objects and members. For example, aircraft, managers, events, and locations might all be objects.
  - Analyze different use cases and try to find objects that use cases have in common.

Design patterns

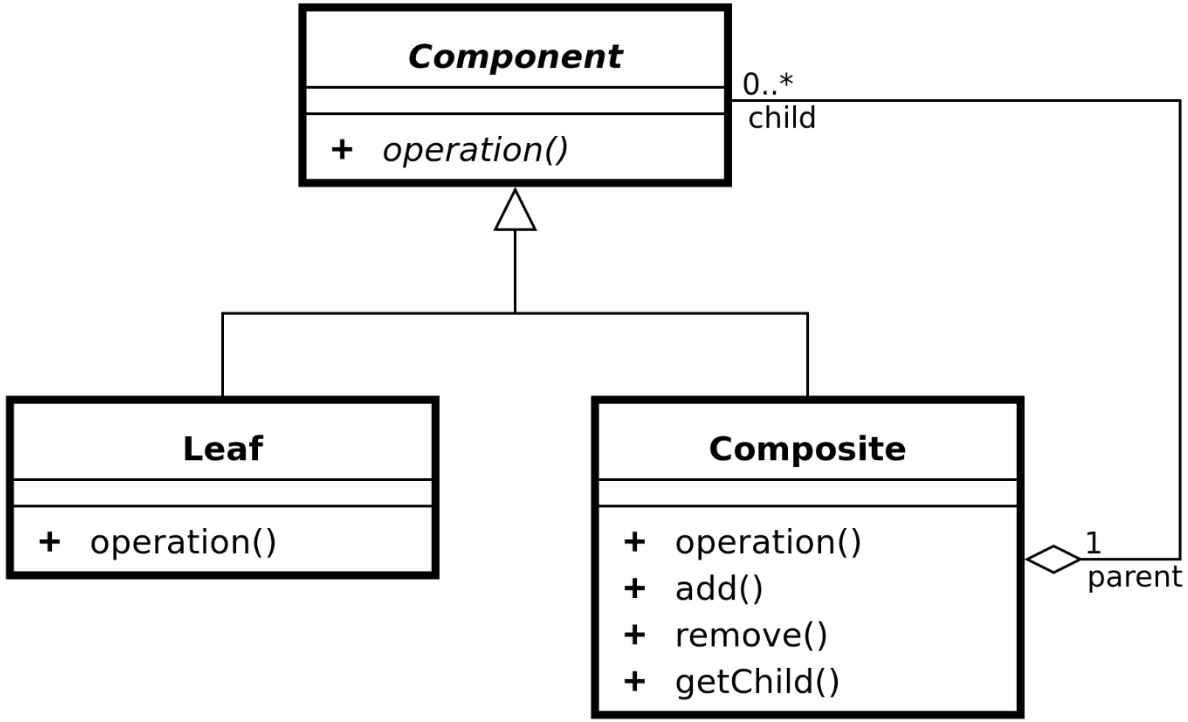
- **Software design patterns** are ways of designing objects that have been used successfully in the past
  - Think of them as rough blueprints or guidelines
- The idea emerged in the late 70s and is best known from the 1994 book *Design Patterns: Elements of Reusable Object-Oriented Software*
  - 23 different patterns, written by the Gang of Four: Gamma, Helm, Johnson, and Vlissides
- 10 years ago, job interviews routinely asked questions about design patterns
- The software engineering community is not as focused on design patterns now, though they are still useful

Elements of design patterns

- Design patterns have four essential elements:
  - A meaningful name
  - A description of the problem area that explains when the pattern may be applied
  - A solution description of the parts of the design, their relationships, and their responsibilities
  - A statement of the consequences of using the design pattern
- Patterns are more abstract than code

Composite pattern

- The **composite pattern** is useful for part-whole hierarchies of objects
- A group of objects somewhere in the hierarchy can be treated like a single object
- The Swing library uses the composite pattern for its graphical components
- Problems the composite pattern solves:
  - Representing a part-whole hierarchy so that clients can treat parts and wholes the same
  - Representing a part-whole hierarchy as a tree

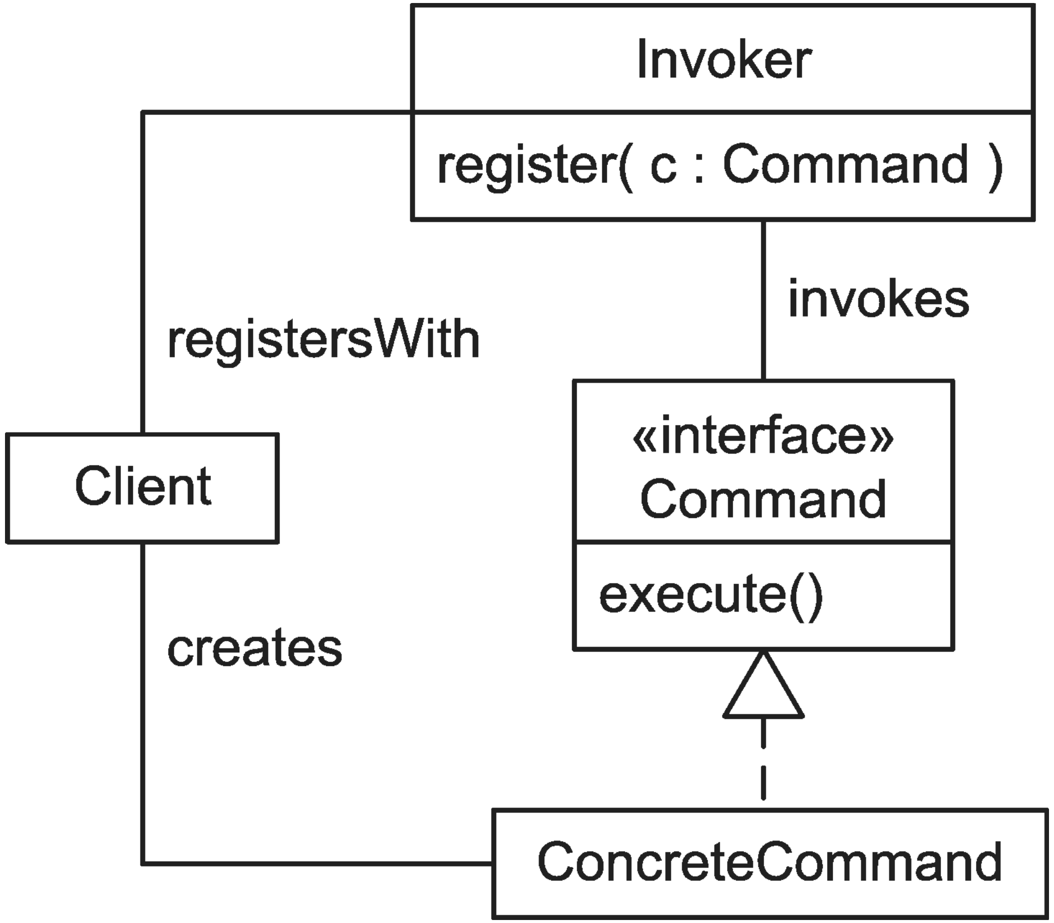


Composite pattern in code

```
interface Component {
     public void doAction(); // Draw, print, etc.
}
public class Composite implements Component {
     private List<Component> children = new ArrayList<>();
     public void add(Component component) {
          children.add(component);
     }
     public void doAction() {
          for (Component component : children)
               component.doAction();
     }
}
```

Command pattern

- The **command pattern** is useful for encapsulating an action in an object
- The action is independent from the objects that used it and can be stored for later
- The Swing library uses the command pattern for events
- Problems the command pattern solves:
  - Decoupling the requester from a request

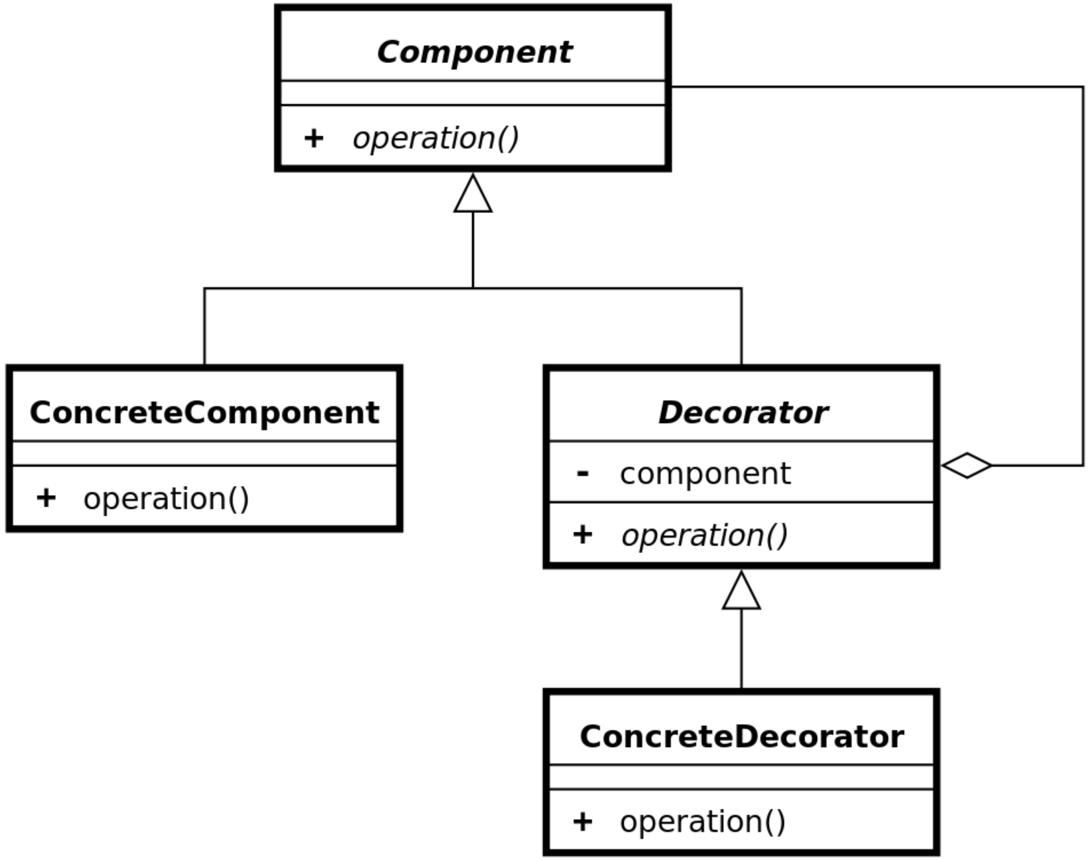


Command pattern in code

```java
interface Command {
    public void execute(); // Do something
}
public class Invoker {
    private Map<String, Command> commands = new HashMap<>();
    public void register(String name, Command command) {
        commands.put(name, command);
    }
    public void execute(String name) {
        Command command = commands.get(name);
        if (command == null)
            throw new IllegalStateException("No command!");
        command.execute();
    }
}
```

Decorator pattern

- The **decorator pattern** provides a way to add responsibilities to an object dynamically at run-time
- It is commonly used to customize the appearance of GUI elements
- The Swing library uses the decorator pattern to customize borders
- Problems the decorator pattern solves:
  - Adding responsibilities to an object dynamically at run-time
  - Providing a flexible alternative to inheritance for extending functionality

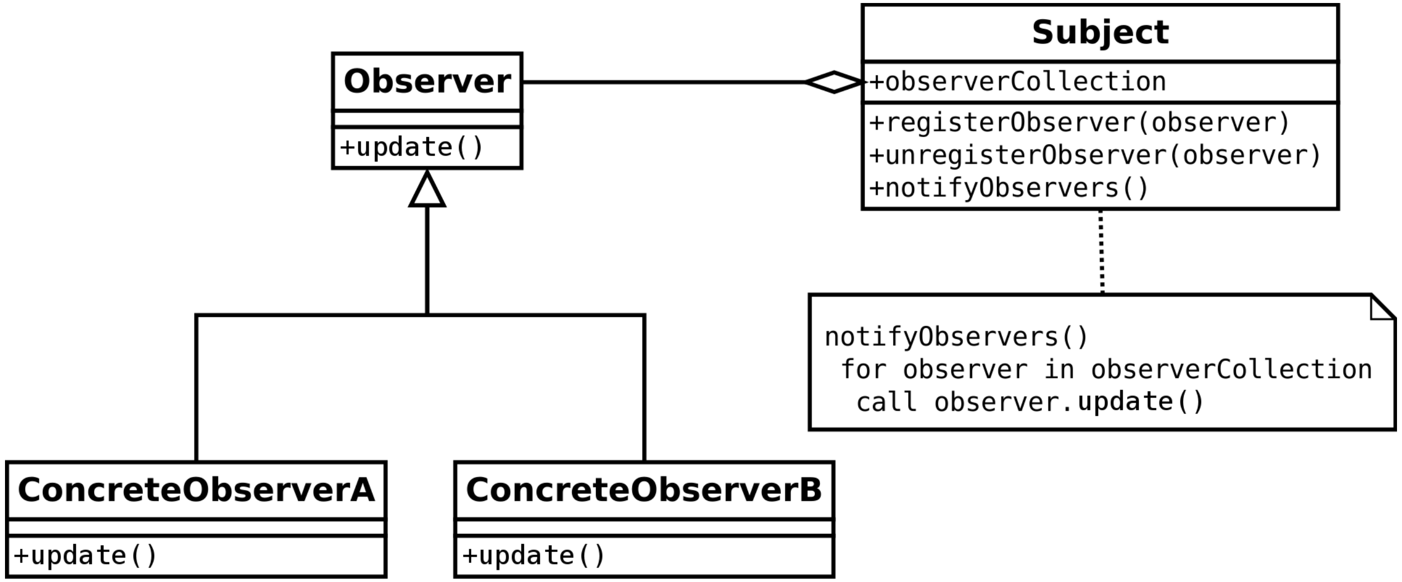


Decorator pattern in code

```
public class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window windowToBeDecorated) {
        super(windowToBeDecorated);
    }

    public void draw() {
        super.draw();
        drawVerticalScrollBar();
    }

    private void drawVerticalScrollBar() {
        // Draw the vertical scrollbar
    }
}
```

Observer pattern

- The **observer pattern** is useful for a one-to-many dependency where one object changing can update many other objects
- An observer pattern defines Subject and Observer objects
- When a subject changes state, registered observers are updated automatically
- Problems the observer pattern solves:
  - Making a one-to-many dependency between objects without tightly coupling the objects
  - Updating an arbitrarily large number of other objects automatically when one object changes state



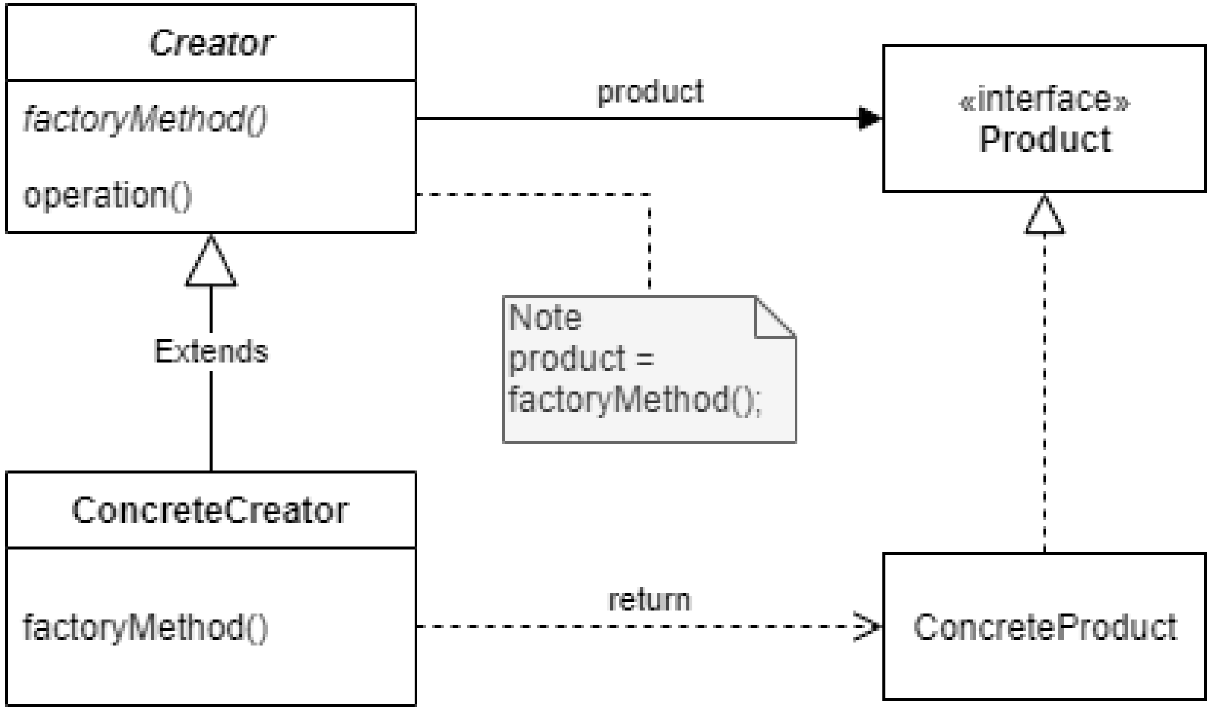
```
notifyObservers()
 for observer in observerCollection
  call observer.update()
```

Observer pattern in code

```
public class Subject {
     private Object data;
     private List<Observer> observers = new ArrayList<>();
     public void registerObserver(Observer observer) {
          observers.add(observer);
     }
     public void setData(Object data) {
          this.data = data;
          for (Observer observer : observers)
               observer.update(data);
     }
}
```

Factory method pattern

- The **factory method design pattern** allows a method to be overridden so that a child class can determine what kind of object to create
- A factory method is defined that is used to create objects
- Problems the factory method pattern solves:
  - Allowing subclasses to define which class to instantiate

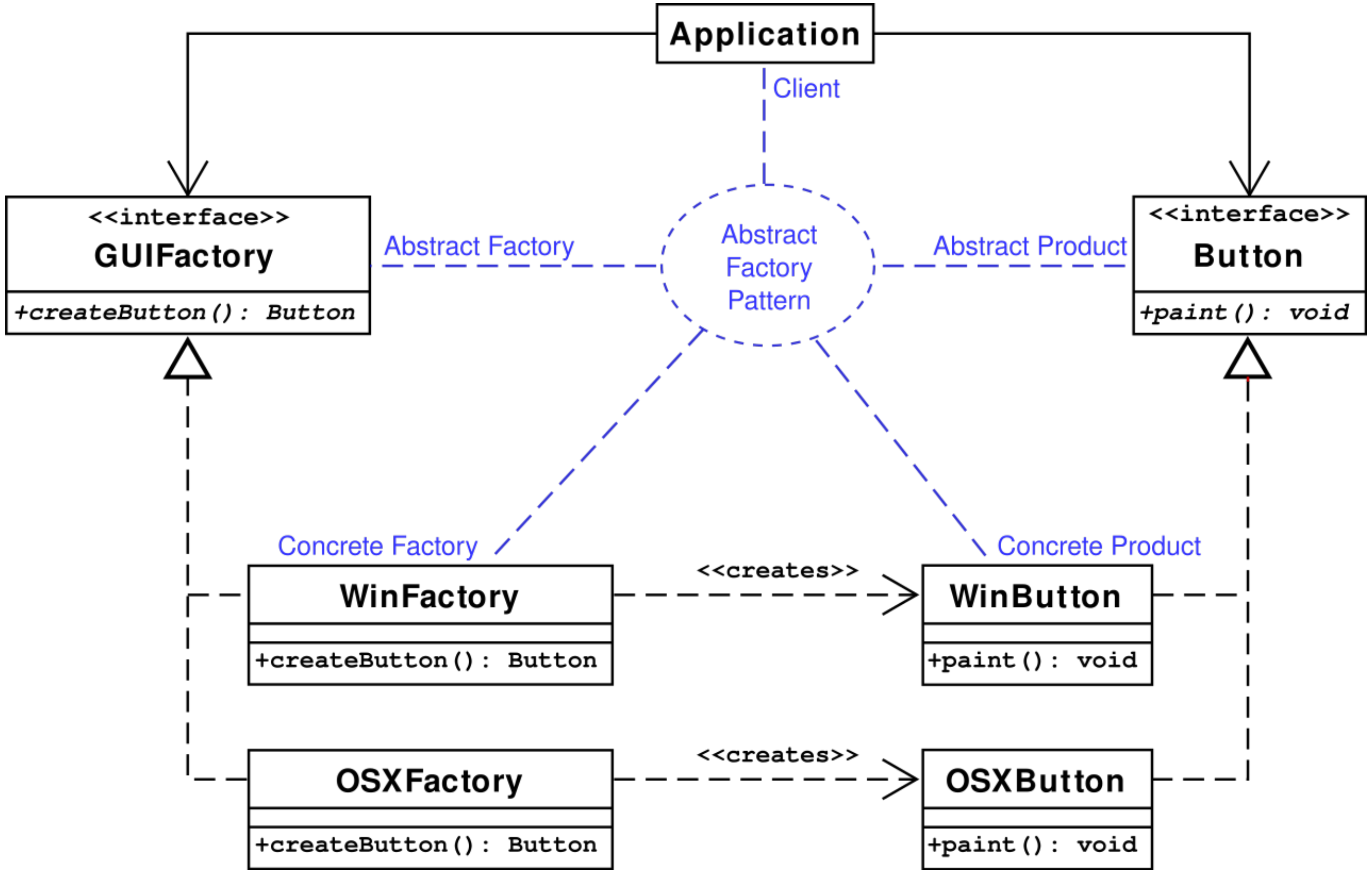


Factory method pattern in code

```
interface Room {
     public void connect(Room room);
}
public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }
    abstract protected Room makeRoom();
}
```

Abstract factory pattern

- The **abstract factory pattern** is similar except that it uses some object as a factory instead of overriding a method
- Problems the abstract factory pattern solves:
  - Making a class be independent of the objects it requires
  - Making a family of related objects

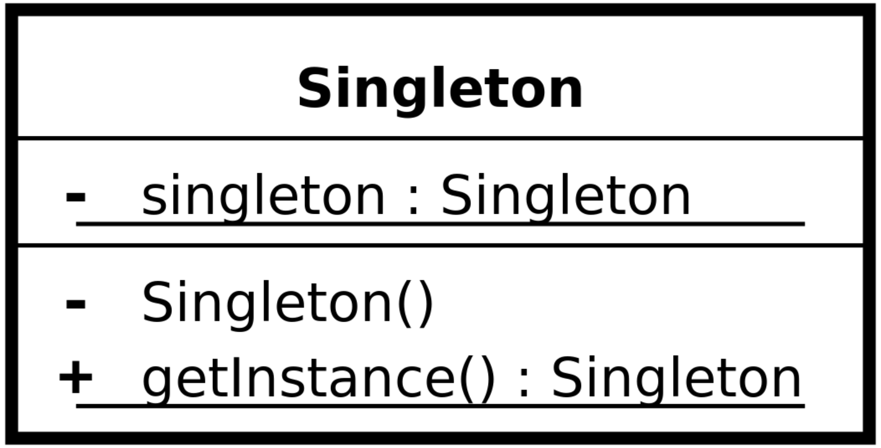


Abstract factory pattern

```
public interface Button {
       void paint();
}

public interface GUIFactory {
       public Button createButton();
}

public class WindowsFactory implements GUIFactory {
       public Button createButton() {
              return new WindowsButton();
       }
}

public class OSXFactory implements GUIFactory {
       public Button createButton() {
              return new OSXButton();
       }
}
```

Singleton pattern

- Sometimes it's useful to have only a single instance of a class
- The **singleton pattern** makes it so that it's possible to make only one object of a class and makes it easy to access
- Problems the singleton pattern solves:
  - Ensuring that there's only one instance of a class
  - Making the instance of a class easy to get



Singleton

- singleton : Singleton

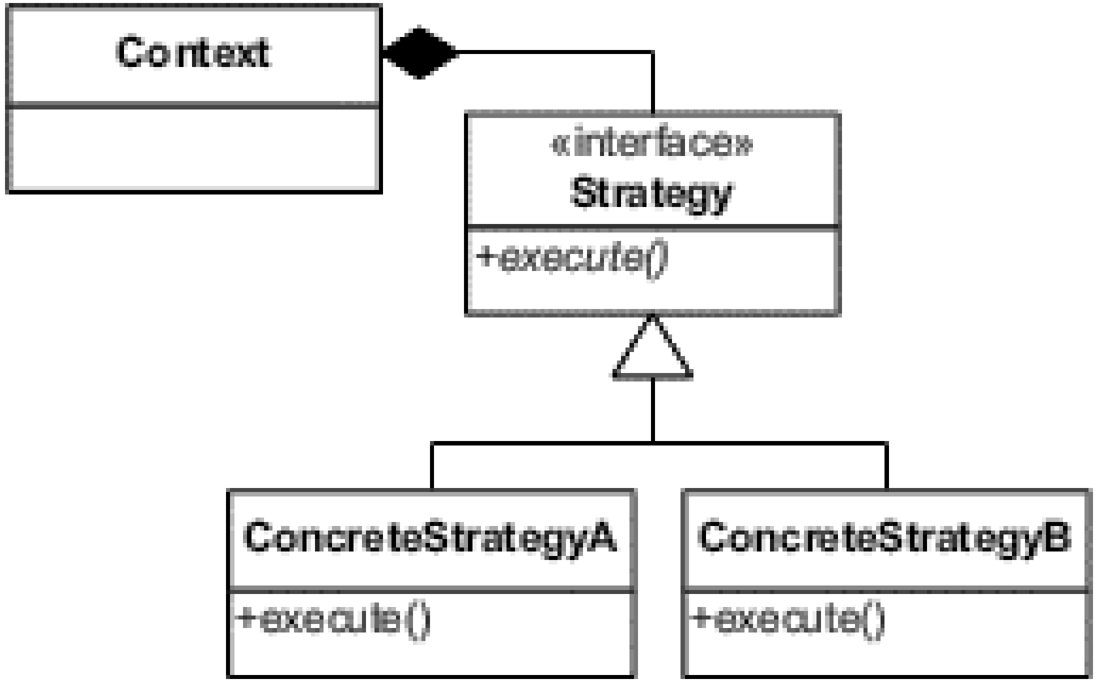
- Singleton()
+ getInstance() : Singleton

Singleton pattern in code

```
public final class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return INSTANCE;
    }
}
```

Strategy pattern

- The **strategy pattern** allows an algorithm to be selected at run-time
- In Java, that algorithm is usually encapsulated in the method of an object
- Problems the strategy pattern solves:
  - Configuring a class with an algorithm at run-time
  - Selecting or exchanging an algorithm at run-time

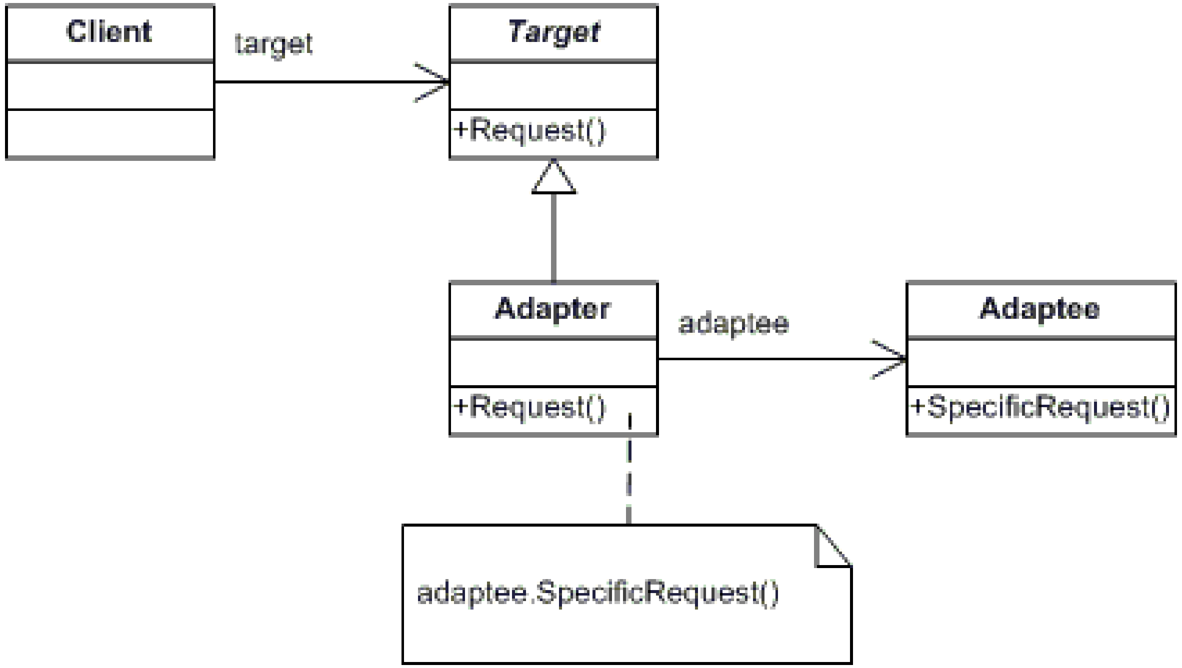


Strategy pattern in code

```
interface BillingStrategy {
    double getPrice(double rawPrice);
}

// Normal billing strategy (unchanged price)
public class NormalStrategy implements BillingStrategy {
    public double getPrice(double rawPrice) {
        return rawPrice;
    }
}

// Strategy for Happy hour (50% discount)
public class HappyHourStrategy implements BillingStrategy {
    public double getPrice(double rawPrice) {
        return rawPrice*0.5;
    }
}
```

Adapter pattern

- Sometimes you have an object that doesn't generate the right kind of output
- The **adapter pattern** allows you to turn the output from something that gives one kind of output into the kind you need
- Problems the adapter pattern solves:
  - Reusing a class that doesn't have an interface the client requires
  - Allowing classes with incompatible interfaces to work together



Adapter pattern

```
public interface IceProvider {
    Ice getIce();
}

public class WaterToIce implements IceProvider {
    private WaterMaker maker = null;

    public WaterToIce(WaterMaker maker) {
        this.maker = maker;
    }

    public Ice getIce() {
        return maker.getWater().freeze();
    }
}
```

Upcoming

Next time…

- Construction techniques

Reminders

- Read Chapter 8: Construction Techniques
- Keep working on the draft of Project 2
  - **Due Friday!**