CSE 331

The Strategy and State Patterns

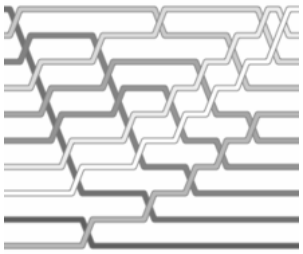
slides created by Marty Stepp
based on materials by M. Ernst, S. Reges, D. Notkin, R. Mercer, Wikipedia
http://www.cs.washington.edu/331/

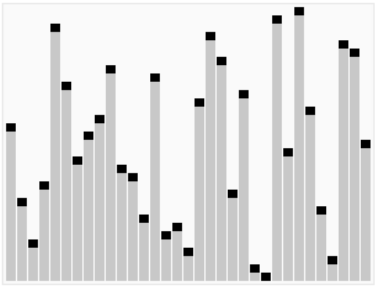
Gang of Four (GoF) patterns

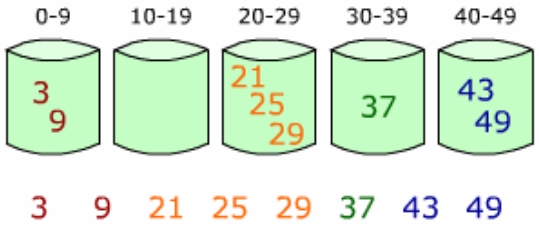
- **Creational Patterns** *(abstracting the object-instantiation process)*
  - **Factory Method**
  - Builder
  - Abstract Factory
  - **Prototype**
  - **Singleton**
- **Structural Patterns** *(how objects/classes can be combined)*
  - **Adapter**
  - **Decorator**
  - Proxy
  - Bridge
  - Facade
  - **Composite**
  - **Flyweight**
- **Behavioral Patterns** *(communication between objects)*
  - Command
  - Mediator
  - Strategy
  - Template Method
  - Interpreter
  - **Observer**
  - Chain of Responsibility
  - **Iterator**
  - State
  - Visitor

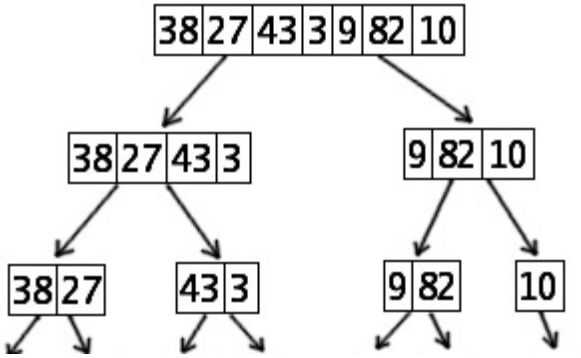
Pattern: Strategy

objects that hold different algorithms to solve a problem









The problem situation

- *Problem:* We want to generalize behavior of one part of our app.
  - Example: Layout of components within containers.
  - Example: Ways of sorting to arrange data.
  - Example: Computer game player AI algorithms.

- Poor solutions to the problem:
  - Boolean flags or many `set` methods to enable various algorithms.
    - myContainer.useFlow(); game.playerDifficulty(3);
  - Lots of `if` statements in our app to choose between algorithms.
    - if (abc) { mergeSort(data); } else if (xyz) { bubbleSort(data); }
  - Rewriting entire model classes just to change the algorithm.
    - FlowContainer, BorderContainer, ..., EasyPlayer, HardPlayer

Strategy pattern

- **strategy**: An algorithm separated from the object that uses it, and encapsulated as its own object.
  - A behavioral pattern.
  - Each strategy implements one specific behavior;
    one implementation of how to solve the same problem.
  - Separates algorithm for behavior from object that wants to act.
  - Allows changing an object's behavior dynamically without extending or changing the object itself.

- examples:
  - file saving; file compression; sorting; `Comparators`
  - layout managers on GUI containers
  - AI algorithms for computer game players

Implementing strategies

- Write an *interface* representing the general behavior / algorithm.

```
public interface CardStrategy {...}
```

- Provide a way to supply an object that meets this interface into the larger overall model (sometimes called *dependency injection*).

```
public class CardGame {
    public void setStrategy(CardStrategy strat) {...}
}
```

- Write classes that implement the interface w/ specific algorithms.

```
public class TimidStrategy implements CardStrategy {...}
public class RandomStrategy implements CardStrategy {...}
public class CleverStrategy implements CardStrategy {...}
```

LayoutManager strategies

- Layout managers in Java implement the Strategy pattern.
  - Each LayoutManager object has an algorithm to position components.

```
public interface LayoutManager {
    void addLayoutComponent(String name, Component comp);
    void layoutContainer(Container container);
    Dimension minimumLayoutSize(Container parent);
    Dimension preferredLayoutSize(Container parent);
    void removeLayoutComponent(Component comp);
}

public class BorderLayout implements LayoutManager {...}
public class FlowLayout implements LayoutManager {...}
public class GridLayout implements LayoutManager {...}
```

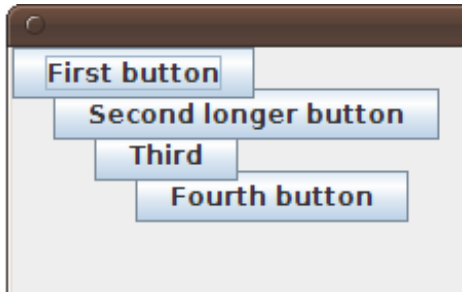
Custom layout example

```
import java.awt.*;

// Lays out components at preferred sizes in a stack that
// cascades from top/left down with 20px between each.
public class CascadingLayout implements LayoutManager {
    private static final int GAP = 20;

    public void layoutContainer(Container container) {
        int xy = 0;
        for (Component comp : container.getComponents()) {
            comp.setSize(comp.getPreferredSize());
            comp.setLocation(xy, xy);
            xy += GAP;
        }
    }

    public Dimension minimumLayoutSize(Container c) {
        return new Dimension(0, 0);
    }
    public Dimension preferredLayoutSize(Container c) {
        return new Dimension(500, 500);
    }
    public void addLayoutComponent(String n, Component c) {}
    public void removeLayoutComponent(Component c) {}
}
```



Strategies as observers

- Sometimes strategies must react to changes in the state of a model.
  - Example: Game player strategies must play when it is their turn.

- So it can be useful to have the strategy *observe* the model:
  - `myGame.addObserver(myStrategy);`

- Possible complication: Can the strategy do something malicious? Can a rogue strategy put the game into an invalid state?
  - How might we avoid or fix this problem?

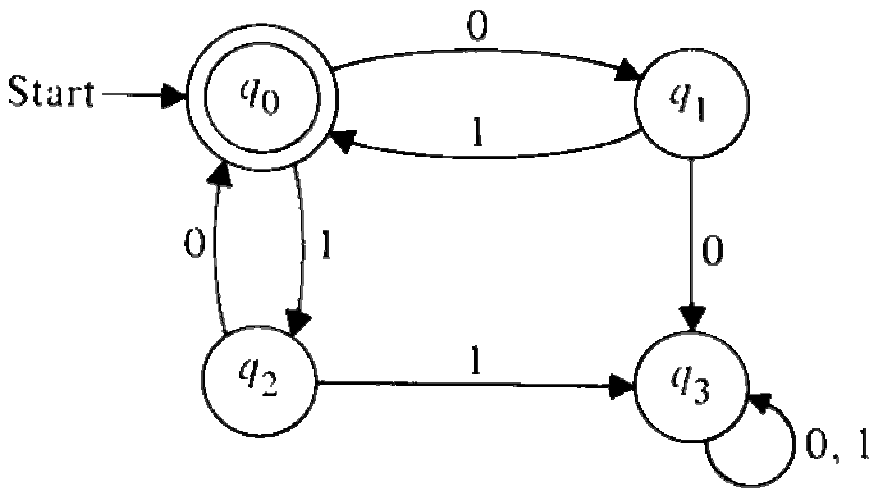
Strategy exercise

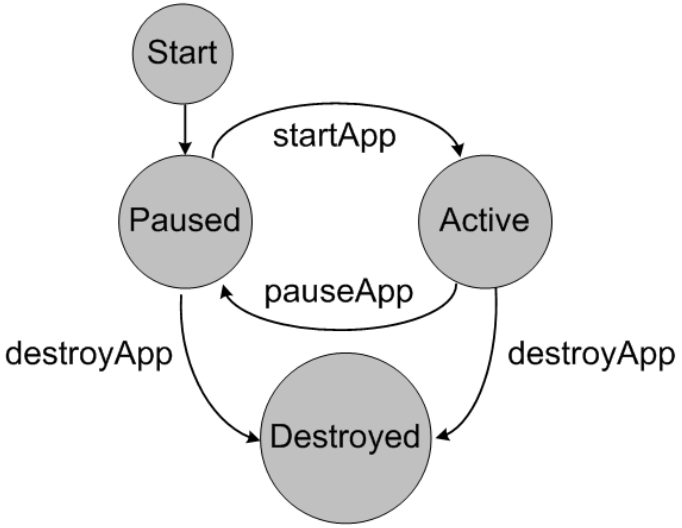
- Modify the Rock-Paper-Scissors game to pit a human player against a computer player.

- Give the computer player the ability to use different strategies:
  - `RockStrategy`: Always chooses rock.
  - `RandomStrategy`: Chooses completely at random.
  - `LearningStrategy`: Chooses the weapon to beat the weapon that was chosen by the human player last game.
  - `StatisticalStrategy`: Chooses the weapon that will beat the weapon being used by the human player the majority if the time. If there is a tie, chooses any weapon randomly.

Pattern: State

representing the state of one object using another object





The problem situation

- *Problem:* We have a model with complex states.
  - Example: A poker game that can be in progress, betting, drawing, ...
  - Example: A network app that can wait for messages, send, ...
  - Various parts of our code (in and out of the model) need to understand and react to that state in different ways.

- Poor solutions to the problem:
  - Trying to deduce the model's state based on complex analysis of various fields within the model.
    - if the winner is null and current player is p2, then ...
    - if my message buffer queue is empty and 0 bytes available, then ...

State pattern

- **state**: An object whose sole purpose is to represent the current "state" or configuration of another larger object.
  - A behavioral pattern.
  - Often implemented with an `enum` type for the states.
  - Each state object represents one specific state for the larger object.
  - The larger object will set its state in response to various mutations.
  - Allows various observers and interested parties to quickly and accurately know what is going on with the larger object's status.

- Analogous to the notion of *finite state machines*.
  - Set of states (nodes)
  - Set of edges (mutations that cause state changes)

State enum example

```
// Represents states for a poker game.
public enum GameState {
    NOT_STARTED, IN_PROGRESS, WAITING_FOR_BETS,
    DEALING, GAME_OVER;
}

// Poker game model class.
public class PokerGame {
    private GameState state;

    public GameState getState() { return state; }

    public void ante(int amount) {
        ...
        state = WAITING_FOR_BETS;    // change state
        setChanged();
        notifyObservers(state);
    }
}
```