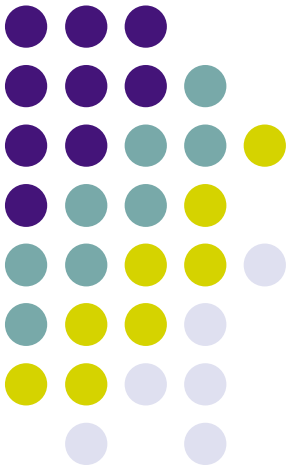
More on Design

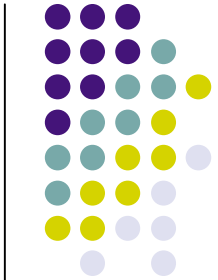
CSCI 5828: Foundations of Software Engineering

Lecture 23

Kenneth M. Anderson

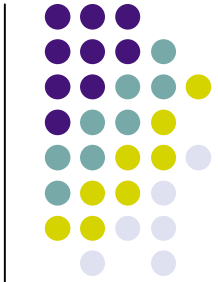


Outline



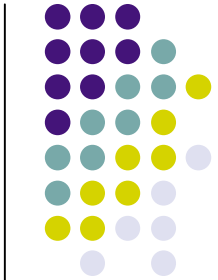
- Additional Design-Related Topics
  - **Design Patterns**
    - **Singleton**
    - **Strategy**
    - **Model View Controller**
  - Design by Convention
    - Inversion of Control (also, Dependency Injection)
  - Refactoring (high level overview)
    - A graphical example (details in a later lecture)

Design Patterns



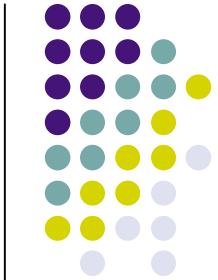
- Addison-Wesley book published in 1995
  - ISBN 0-201-63361-2
- Authors
  - Erich Gamma
  - Richard Helm
  - Ralph Johnson
  - John Vlissides
- Known as “The Gang of Four”
- Presents 23 Design Patterns

What are Patterns?



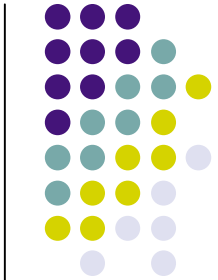
- Christopher Alexander talking about buildings and towns
  - “Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”
  - Alexander, et al., A Pattern Language. Oxford University Press, 1977

Patterns, continued



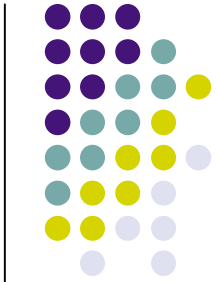
- Patterns can have different levels of abstraction
- In Design Patterns (the book),
  - Patterns are not classes
  - Patterns are not frameworks
  - Instead, Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

Patterns, continued



- So, patterns are formalized solutions to design problems
  - They describe techniques for maximizing flexibility, extensibility, abstraction, etc.
- These solutions can typically be translated to code in a straightforward manner

Elements of a Pattern



- Pattern Name
  - More than just a handle for referring to the pattern
  - Each name adds to a designer's vocabulary
    - Enables the discussion of design at a higher abstraction
- The Problem
  - Gives a detailed description of the problem addressed by the pattern
  - Describes when to apply a pattern
    - Often with a list of preconditions

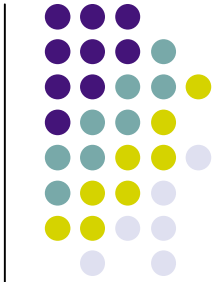
Elements of a Pattern, continued

- The Solution
  - Describes the elements that make up the design, their relationships, responsibilities, and collaborations
  - Does not describe a concrete solution
    - Instead a template to be applied in many situations

Elements of a Pattern, continued

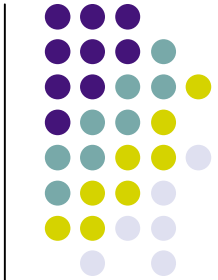
- The consequences
  - Describes the results and tradeoffs of applying the pattern
    - Critical for evaluating design alternatives
  - Typically include
    - Impact on flexibility, extensibility, or portability
    - Space and Time tradeoffs
    - Language and Implementation issues

Design Pattern Template



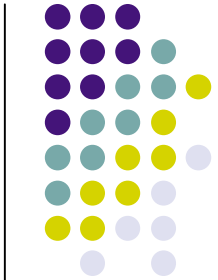
- Pattern Name and Classification
  - Creational
  - Structural
  - Behavioral
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Examples



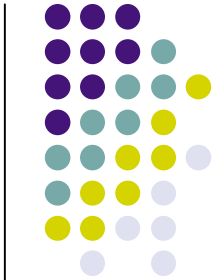
- Singleton
- Strategy
- Model View Controller

Singleton



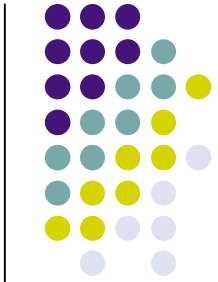
- Intent
  - Ensure a class has only one instance, and provide a global point of access to it
- Motivation
  - Some classes represent objects where multiple instances do not make sense or can lead to a security risk (e.g. Java security managers)

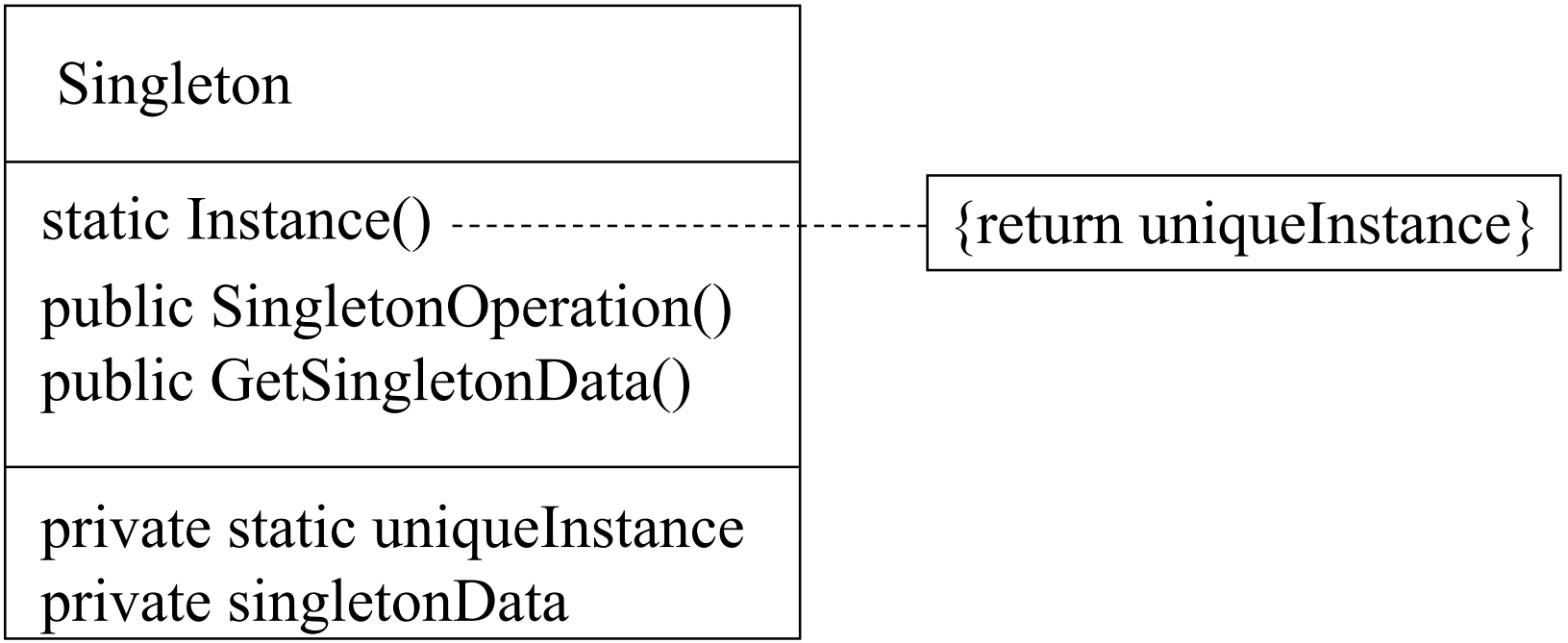
Singleton, continued



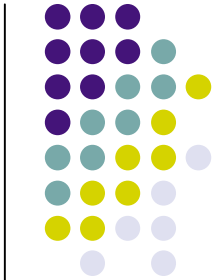
- Applicability
  - Use the Singleton pattern when
    - there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point
    - when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Singleton Structure



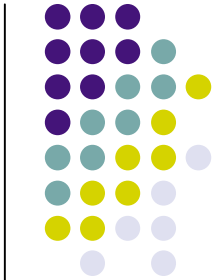


Singleton, continued



- Participants
  - Just the Singleton class
- Collaborations
  - Clients access a Singleton instance solely through Singleton's Instance operation
- Consequences
  - Controlled access to sole instance
  - Reduced name space (versus global variables)
  - Permits a variable number of instances (if desired)

Implementation

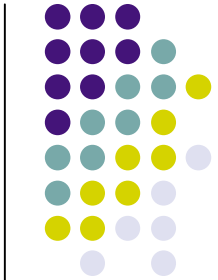


```
import java.util.Date;

public class Singleton {
    private static Singleton theOnlyOne;
    private Date d = new Date();

    private Singleton() {
    }
    public synchronized static Singleton instance() {
        if (theOnlyOne == null) {
            theOnlyOne = new Singleton();
        }
        return theOnlyOne;
    }
    public Date getDate() {
        return d;
    }
}
```

Using our Singleton Class

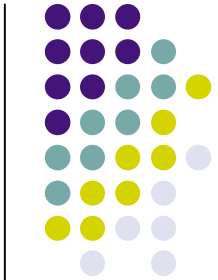


```
public class useSingleton {
    public static void main(String[] args) {
        Singleton a = Singleton.instance();
        Singleton b = Singleton.instance();
        System.out.println("" + a.getDate());
        System.out.println("" + b.getDate());
        System.out.println("" + a);
        System.out.println("" + b);
    }
}
Output:
Sun Apr 07 13:03:34 MDT 2002
Sun Apr 07 13:03:34 MDT 2002
Singleton@136646
Singleton@136646
```

Names of Classes in Patterns

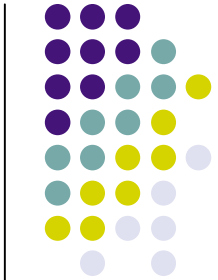
- Are the class names specified in a pattern required?
  - No!
    - Consider an environment where a system has access to only one printer
    - Would you want to name the class that provides access to the printer “Singleton”??!!
      - No, you would want to name it something like “Printer”!
  - On the other hand
    - Incorporating the name of the classes of the pattern can help to communicate their use to designers
      - “Oh, I see you have a “PrinterObserver” class, are you using the Observable design pattern?

Names, continued



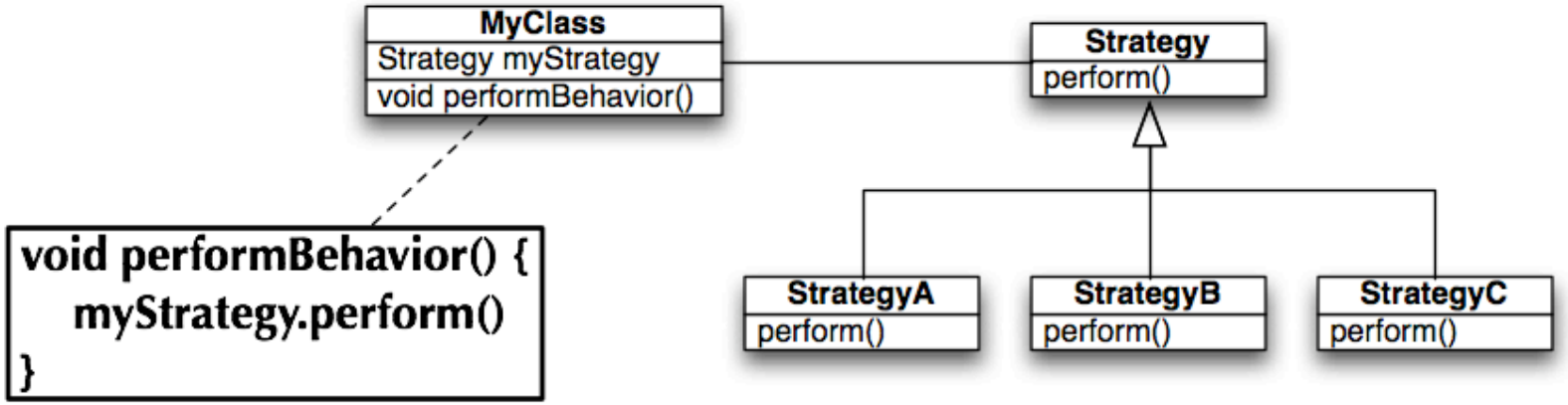
- So, if names are unimportant, what is?
  - Structure!
- We can name our Singleton class anything so long as it
  - has a private or protected constructor
    - need a protected constructor to allow subclasses
  - has a static “instance” operation to retrieve the single instance

Strategy



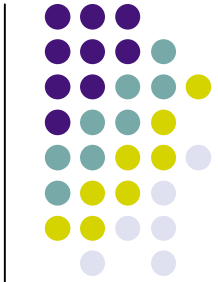
- Separate an object and its behavior by encapsulating the behavior in a separate class
  - This allows you to change an object's behavior dynamically by switching from one behavior implementation to another

Strategy, continued



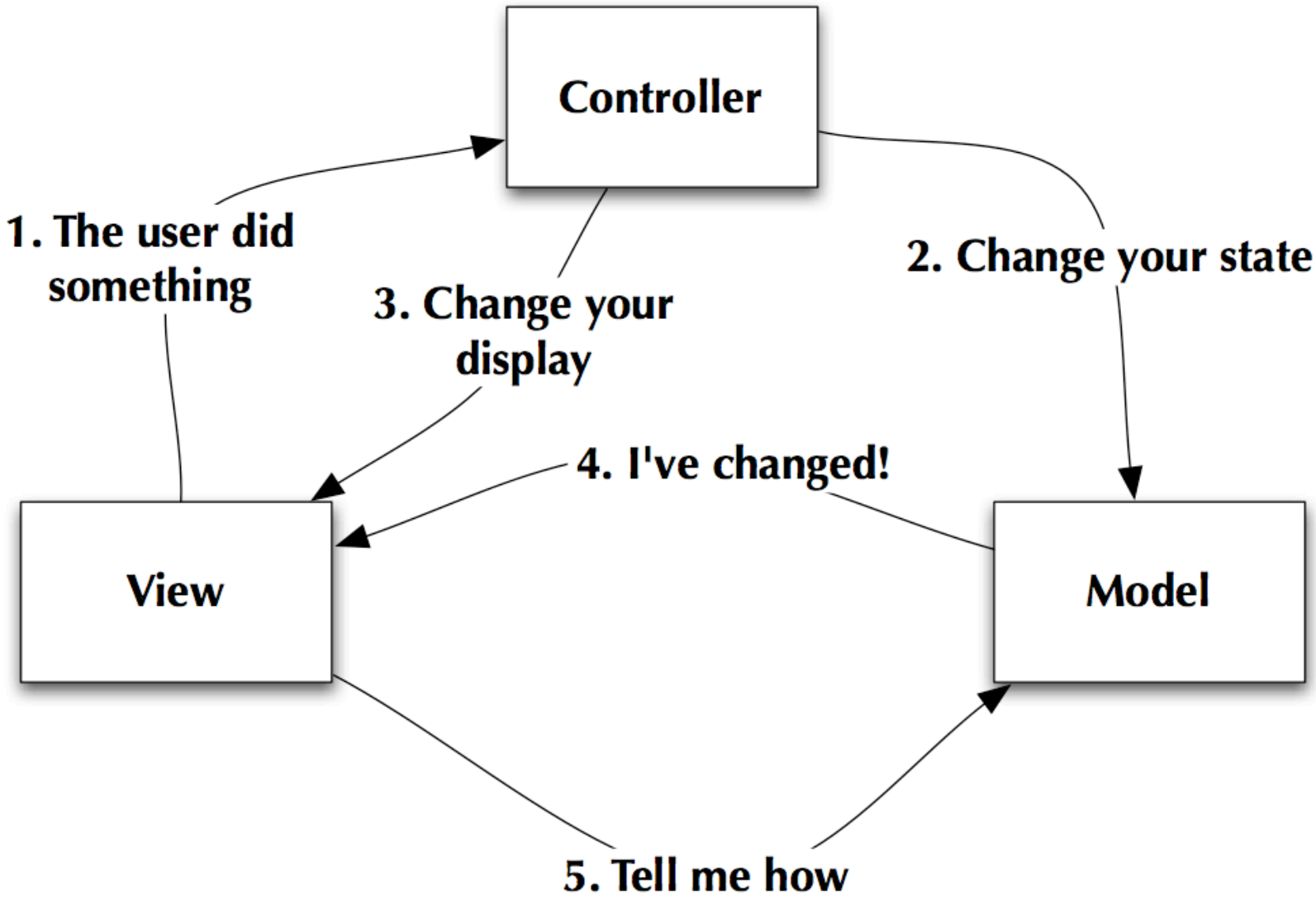
MyClass can exhibit different behaviors, simply by pointing at different instances of Strategy subclasses. (A dependency injection pattern could be used to wire these classes together!)

Model View Controller

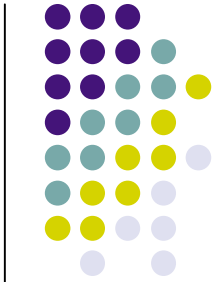


- A pattern for manipulating information that may be displayed in more than one view
  - Model: data structure(s) being manipulated
    - may be capable of notifying observers of state changes
  - View: a visualization of the data structure
    - having more than one view is fine
    - MVC keeps all views in sync as the model changes
  - Controller: handle user input on views
    - make changes to model as appropriate
    - more than one controller means more than one “interaction style” is available

MVC Architecture

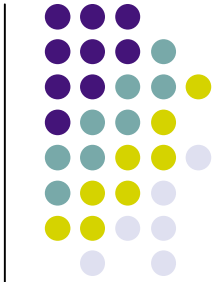


Outline



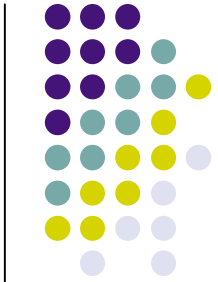
- Additional Design-Related Topics
  - Design Patterns
    - Singleton
    - Strategy
    - Model View Controller
  - **Design by Convention**
    - **Inversion of Control (also, Dependency Injection)**
  - Refactoring (high level overview)
    - A graphical example (details in a later lecture)

Design by Convention



- There has always been a lot of interest in how frameworks for application development are designed
  - Frameworks are typically a set of classes that aid a developer in quickly creating an application for a particular application domain
    - Web frameworks → Web applications
    - GUI frameworks → GUI applications
    - etc.

Design by Convention, cont.

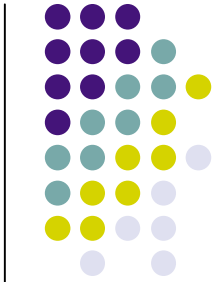


- Typical Use of Frameworks
  - Create subclasses
  - Define plug-ins
  - Create application-specific data and/or config files
- Frameworks typically try to stay out of a developer's way
  - The framework provides only minimal functionality
  - A developer does most of the work of creating an application and/or can override pretty much any framework-defined behavior

Design by conventions, cont..

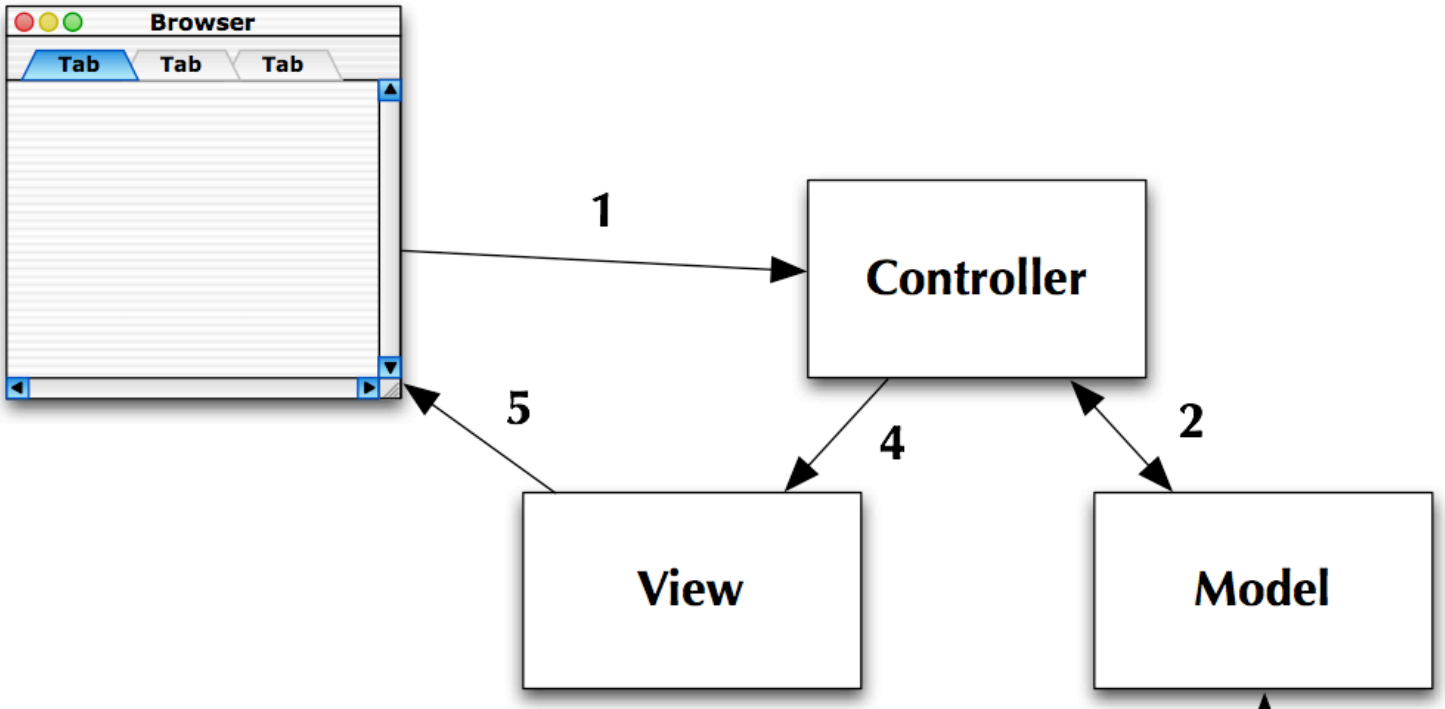
- Recently, there has been interest in creating frameworks that follow a “design by convention” approach
  - Framework authors specify a set of conventions
  - Follow the conventions, get a lot of functionality “for free”
  - Wander from the conventions and risk having your application not work

Ruby on Rails



- Ruby on Rails is a web application framework that is based on the MVC design pattern and makes use of “design by convention”
  - http://www.rubyonrails.org/
- I won’t be able to do Rails justice in this presentation, be sure to watch the videos on this page:
  - http://www.rubyonrails.org/screencasts

Rails Web Architecture

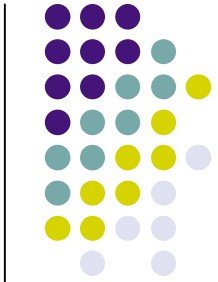


1. Browser sends request
2. Controller interacts with Model
3. Model may access/update Database
4. Controller selects next View
5. View generates response

Design by Convention in Rails

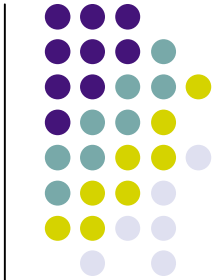
- Use of Design by Convention in Rails
  - scripts are provided to create controllers, views, and models (they produce files in standard locations that are then edited by the developer)
  - Model/Database naming convention
    - Model classes in Rails are mainly empty
    - instead, you define a database table first
    - you call the table using a plural noun (Fishes)
    - you call the model class a singular noun (Fish)
    - Rails populates the Model class dynamically with attributes and methods based on the information in the table

DoC in Rails, continued



- DoC examples in Rails
  - Structure of database described in files called “database migrations”
    - Migrations can be chained together to evolve an old version of a Rails application to a newer version, step by step, while maintaining as much data as possible
  - Names used by controllers are significant
    - if a view layout is created with the same name as a controller, then all views rendered by that controller will use that layout by default

Inversion of Control

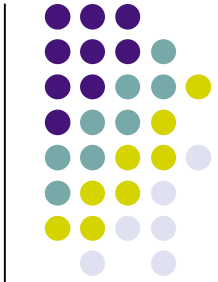


- All application frameworks make use of a design pattern known as “inversion of control”
  - It occurs whenever we define code that will be called by the framework to handle application specific behavior
    - Indeed, inversion of control is what distinguishes a framework from a library
  - For instance an application framework may require a developer to create a subclass of a class called Document
    - When the framework wants to save a document it calls Document.save() which via polymorphism calls the save() method of the subclass provided by the developer

Inversion of Control, continued

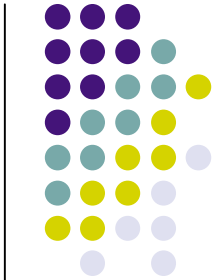
- Inversion of control requires a different style of programming, Contrast
  - puts “What is your name?”
  - name = gets
- with
  - JButton ok = new JButton(“Ok”)
  - ok.addActionListener(this)
  - panel.add(ok)
  - panel.show()
- With the latter, we have no way of knowing when the button will be clicked. We have to wait until our “action listener” is notified, and then respond

IoC Frameworks



- In J2EE, there is a notion of a “container” that is used to provide an “environment” within which a J2EE application can run
  - The application defines a bunch of code and dependencies statically—following certain conventions—and the container uses this information to
    - wire up the application at run-time
    - call its code at specific times in the life cycle

IoC Frameworks, continued



- With respect to wiring an application at run-time, there is a variant of inversion of control known as “dependency injection”
- Martin Fowler provides details of different types of dependency injection at:
  - http://www.martinfowler.com/articles/injection.html
- I will cover the basics in the next few slides (my example is inspired from an example contained in the article above)

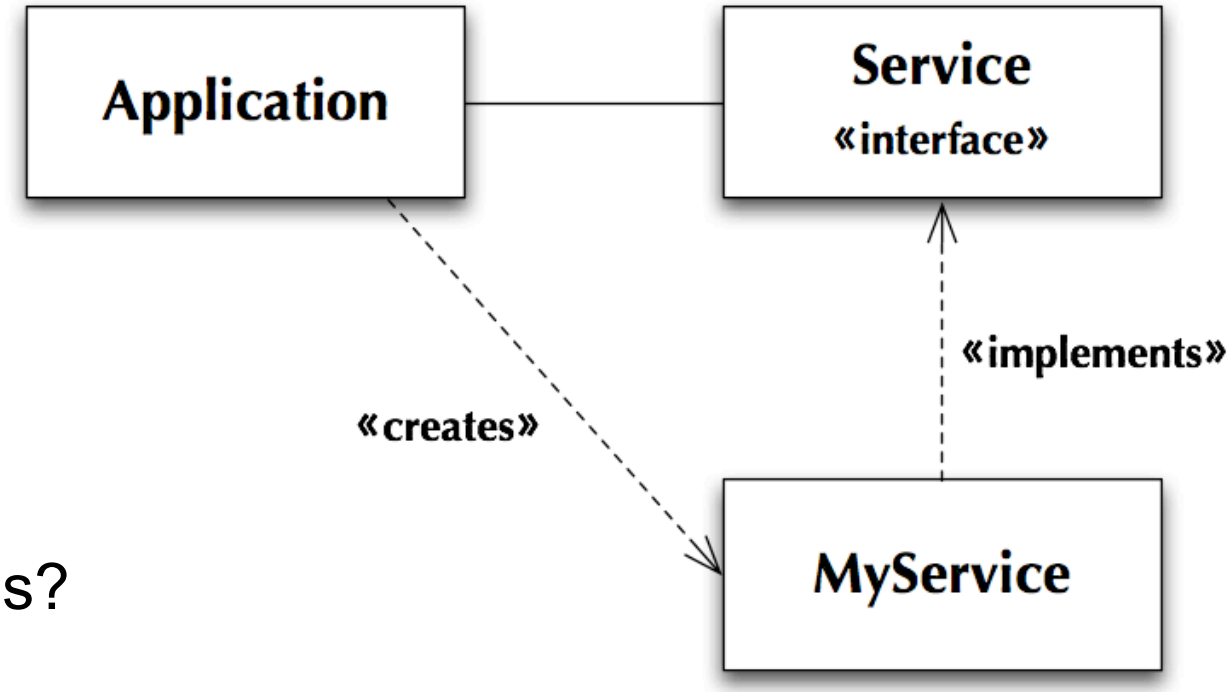
Dependency Injection Example

- Application makes use of a specific service

- Con: Application can not easily switch between different variations of the same service without being modified

Example, continued

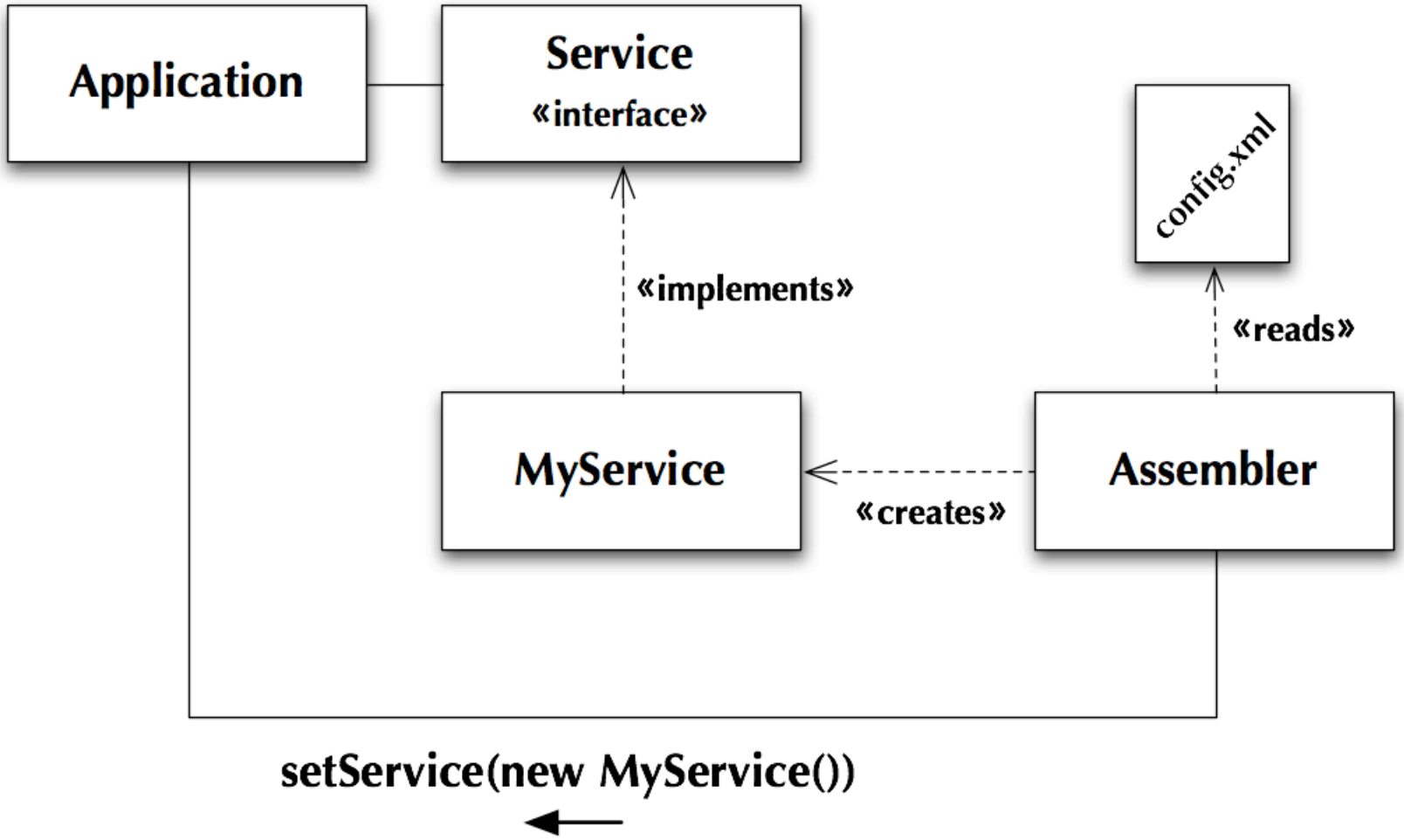
- Create Service interface
- Switch Application to use new interface



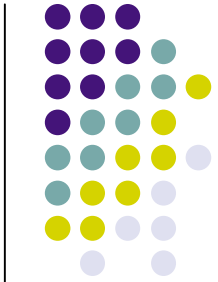
Pros?

Cons?

Example, continued

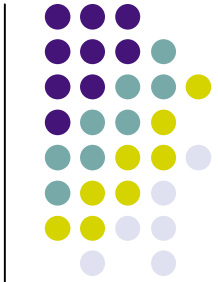


Outline



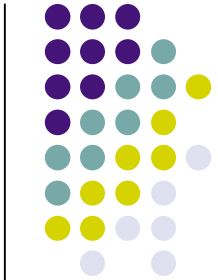
- Additional Design-Related Topics
  - Design Patterns
    - Singleton
    - Strategy
    - Model View Controller
  - Design by Convention
    - Inversion of Control (also, Dependency Injection)
  - **Refactoring (high level overview)**
    - **A graphical example (details in a later lecture)**

Refactoring



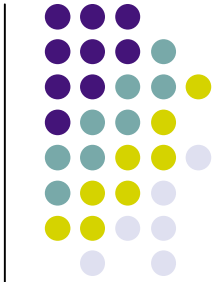
- Reference
  - Refactoring: Improving the Design of Existing Code
  - by Martin Fowler
  - Addison Wesley, 1999

What is Refactoring



- Refactoring is the process of changing a software system such that
  - the external behavior of the system does not change
    - e.g. functional requirements are maintained
  - but the internal structure of the system is improved
- This is sometimes called
  - “Improving the design after it has been written”

(Very) Simple Example



- Consolidate Duplicate Conditional Fragments (page 243); This

```
if (isSpecialDeal()) {
   total = price * 0.95;
   send()
} else {
   total = price * 0.98;
   send()
}
```

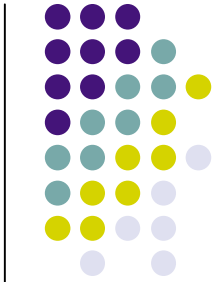
- becomes this

```
if (isSpecialDeal()) {
   total = price * 0.95;
} else {
   total = price * 0.98;
}
send();
```

Design or Code?

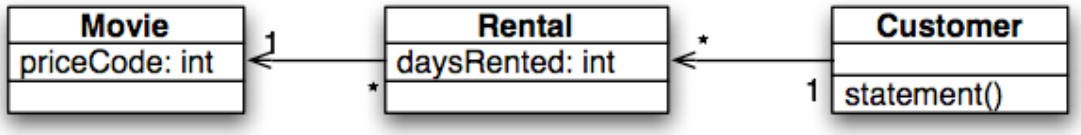
- There is confusion about whether or not refactoring is a design technique or an implementation technique
- It's both!
  - It can only be applied after code has been written
  - However, what it does is help to improve the structure of a software system (i.e. its design)

More detail?



- I intend to cover refactoring in more detail in a future lecture
- Graphical overview
  - Instead, to emphasize my point that refactoring is a design technique, here is a series of slides that shows how a system's structure can be improved via a series of refactorings
    - Each slide (except the first) shows the UML diagram of the system after one or more refactorings has been applied

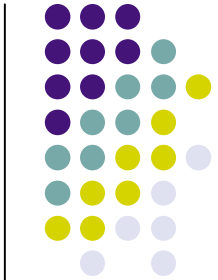
Original System

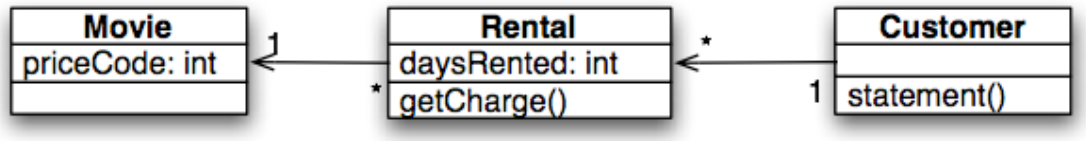


A video rental place uses a system in which a Customer object can generate a statement to determine how much a set of Rentals costs as well as how many “frequent renters points” the customer has earned

Rental and Movie are “data holders”; all logic is contained in statement()

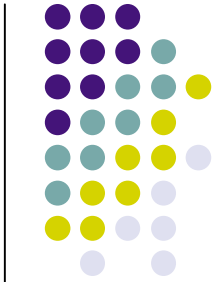
Transform Rental

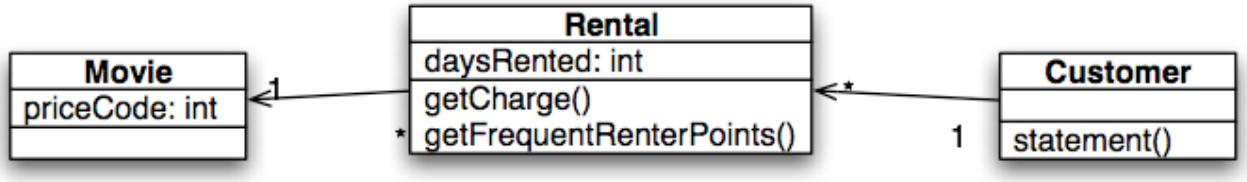




Move some behavior out of statement() and into Rental; Rental now has a getCharge() method which statement() calls when it needs to display the charge of a particular Rental object

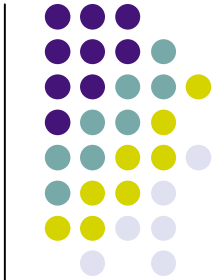
Transform Rental, continued

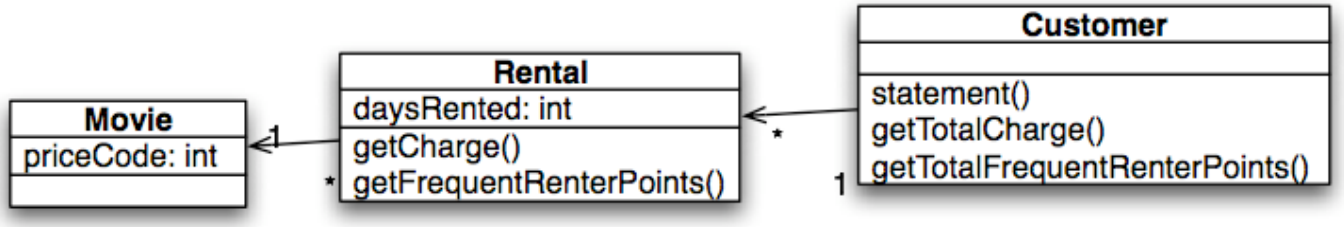




Continue to move behavior into Rental, this time with respect to calculating frequent renter points

Transform Customer

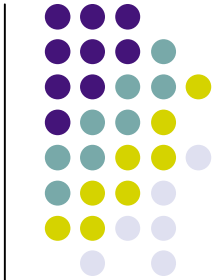


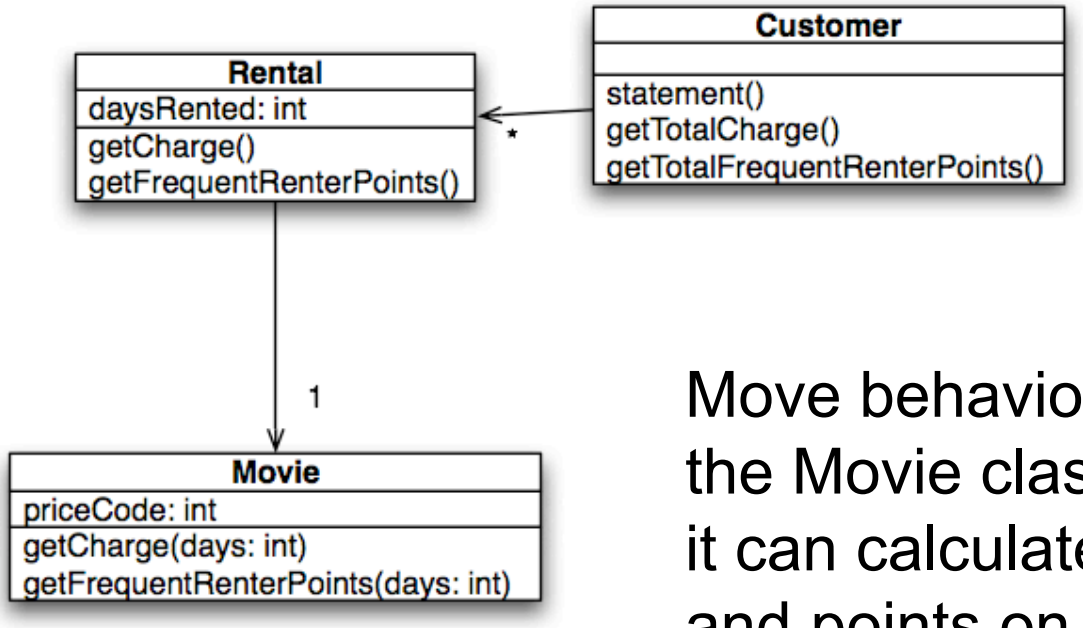


Continue to move code out of statement(); in this step, code for calculating a customer's total charge and points is moved out of statement()

After all, statement() should only care about formatting the statement, not calculating the data that appears within the statement

Transform Movie

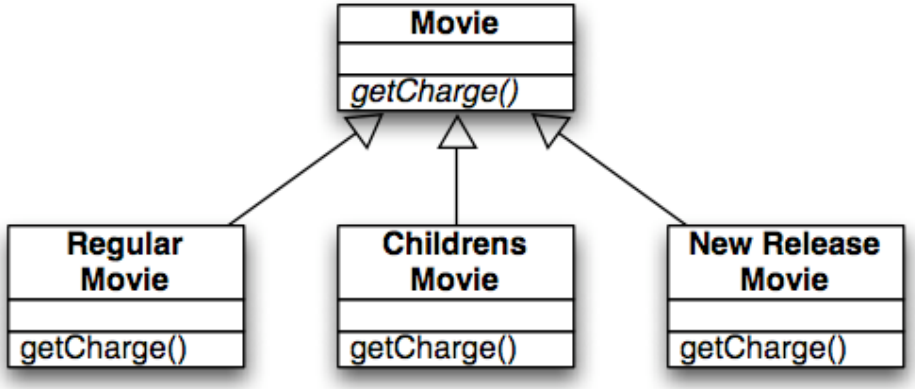




Move behavior into the Movie class; now it can calculate a charge and points on its own

Previously, Rental took care of that, but that doesn't make sense

Transform Movie

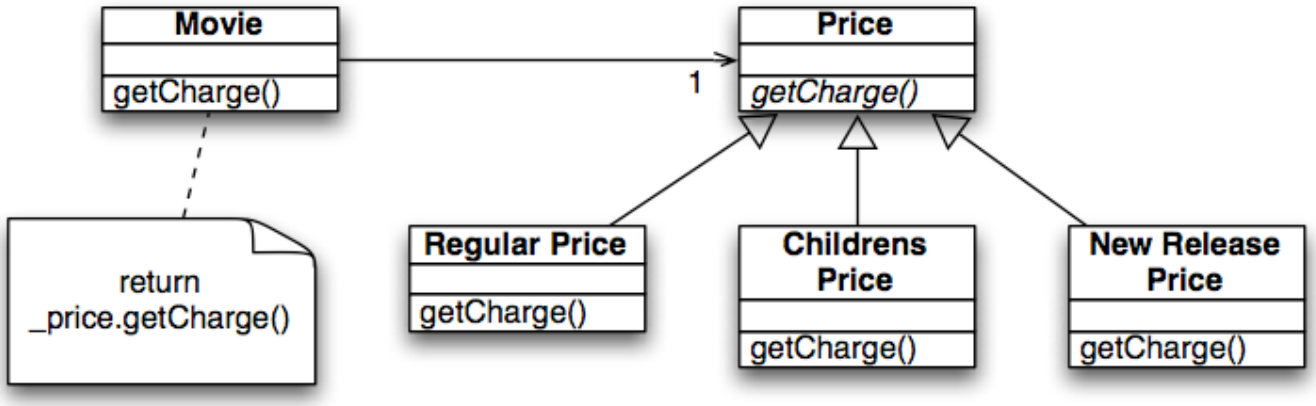


We want to charge different prices for different types of movies; that's what the priceCode variable was for in previous diagrams

priceCode is a throwback to procedural oriented code, however, we should represent different types of movies via subclasses

However, we want a movie's charge to vary according to how recently it was released

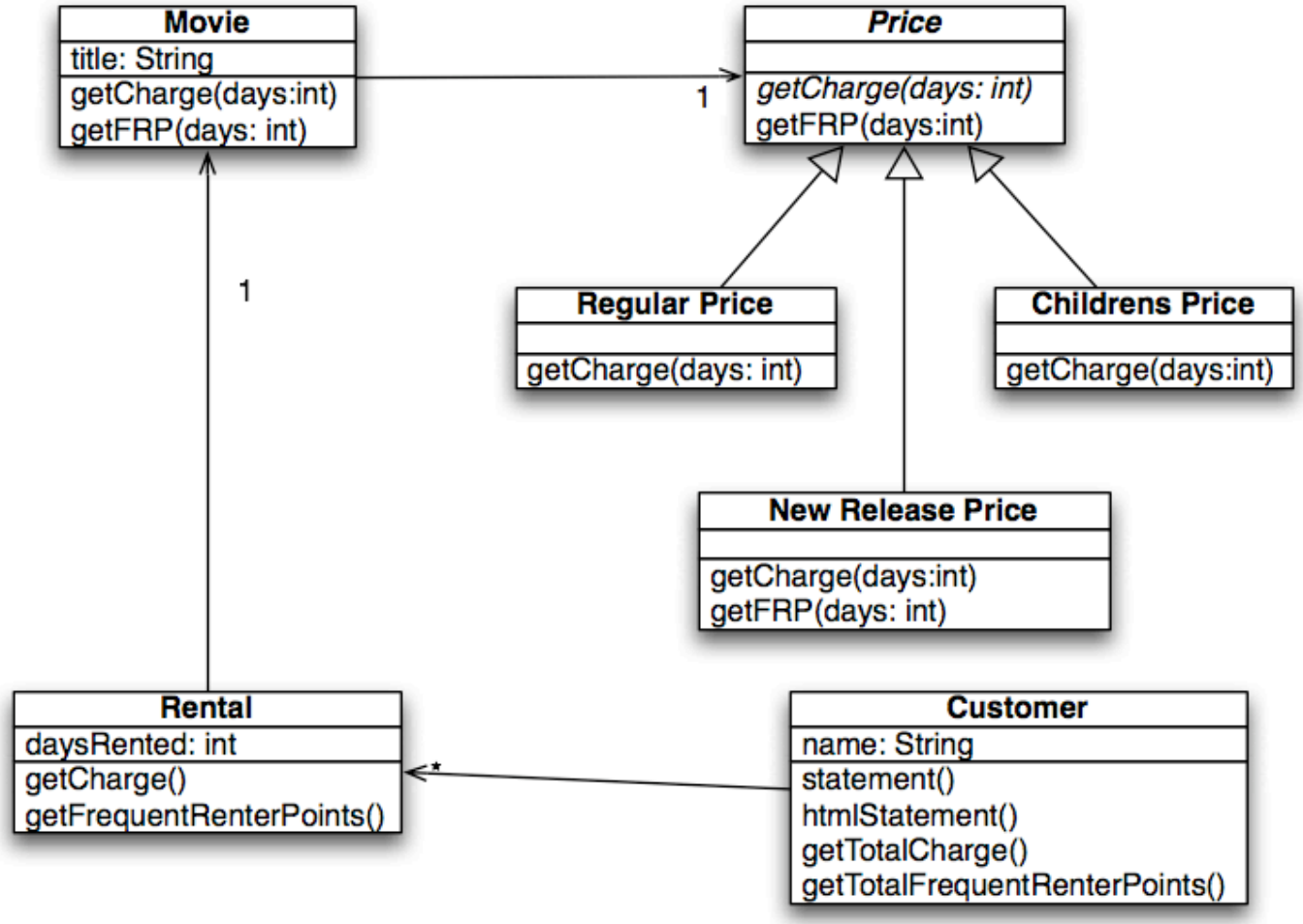
Transform Movie



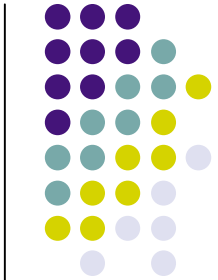
Strategy pattern to the rescue! Avoid the need for Movie subclasses and instead use the strategy pattern to determine the charge for a particular type of movie

Thus, each movie starts out with a “new release” price and then eventually switches to a price based on category

Final System



Summary



- In this lecture, we have reviewed other design-related techniques
  - Design Patterns
  - Design by Convention
  - Refactoring
- Coming Up Next
  - Agile Methods