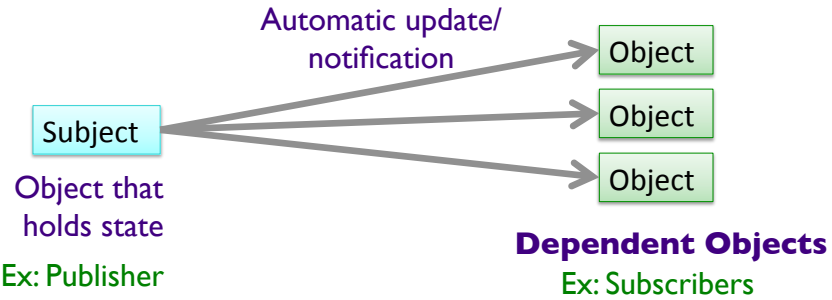
Objectives

- Design Patterns
  - Observer
  - MVC
- Dependency Inversion Principle
  - Factory design pattern
  - Screensavers

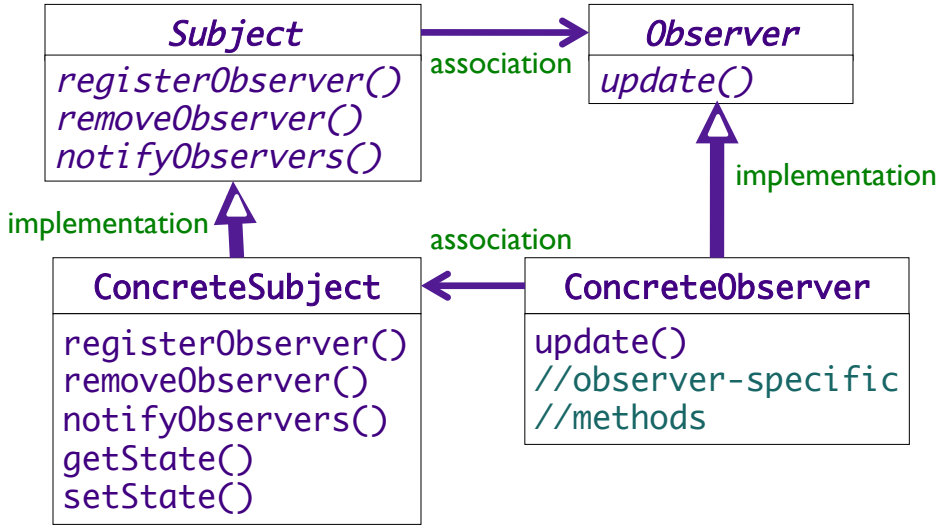
Design Pattern: **Observer**

- Defines a 1-to-many dependency between objects
- When one object changes state, all of its dependents are notified and updated automatically



Observer Pattern

Have we seen this pattern?



Design Principle: **Loose Coupling**

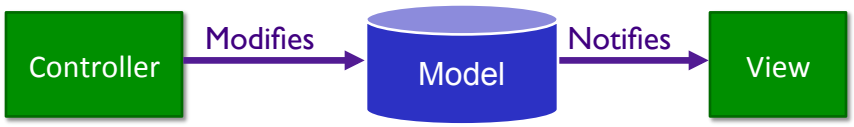
- A principle behind **Observer** pattern

> Goal: loosely coupled designs between objects that interact

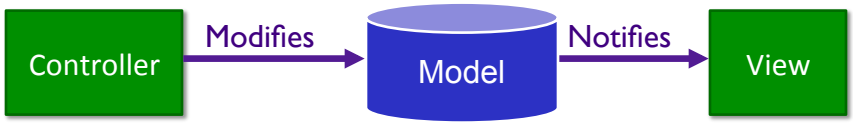
- Loosely coupled objects can interact but have very little knowledge of each other
  - Minimize dependency between objects
  - More flexible systems
  - Handle change

Model - Viewer - Controller (MVC)

- A common **design pattern** for GUIs
- Separate
  - Model: application data
  - View: graphical representation
  - Controller: input processing

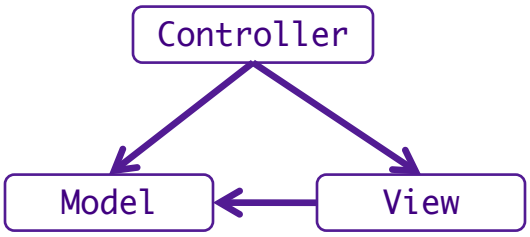


Model-Viewer-Controller

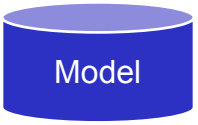


- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations

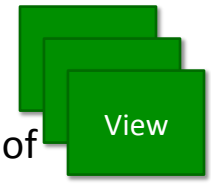


Model



- Code that carries out some task
- Nothing about how view presented to user
- Purely **functional**
- Must be able to register views and notify views of changes

Multiple Views



- Provides GUI interface components of model
  - Look & Feel of the application
- User manipulates view
  - Informs **controller** of change
- Example of multiple views: spreadsheet data
  - Rows/columns in spreadsheet
  - Pie chart, bar chart, …

Controller(s)

- Takes user input and figures out what it means to the model
  - Makes decisions about behavior of model based on UI
- Update **model** as user interacts with **view**
  - Calls model's mutator methods
- Views are associated with controllers

Dependency Inversion Principle

Depend upon abstractions

Dependency Inversion Principle

Depend upon abstractions.
Do not depend upon concrete classes.

- High-level components should not depend on low-level components
  - Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions
- "Inversion" from the way you think

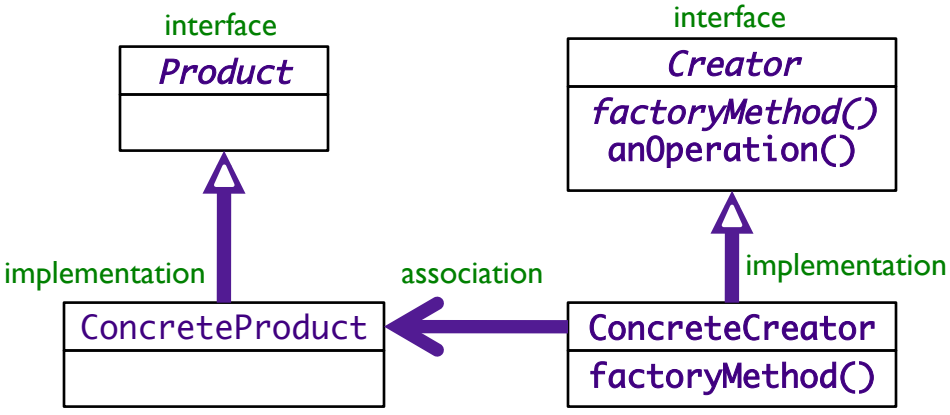
FACTORY DESIGN PATTERN

Design Pattern: **Factory Methods**

- Allows creating objects without specifying exact (concrete) class of created object
- Often used to refer to any method whose main purpose is creating objects

- How it works:
  1. Define a method for creating objects
  2. Child classes override method to specify the derived type of product that will be created

Factory Method Pattern



UML Class Diagram

Guidelines to Follow DIP

- No variable should hold a reference to a concrete class
  - Using new → holding reference to concrete class
  - Use factory instead
- No class should derive from a concrete class
  - Why? Depends on a concrete class
  - Derive from an interface or abstract class instead
- No method should override an implemented method of its base class
  - Base class wasn't an abstraction
  - Those methods are meant to be shared by child classes

What's a problem with following all of these guidelines?

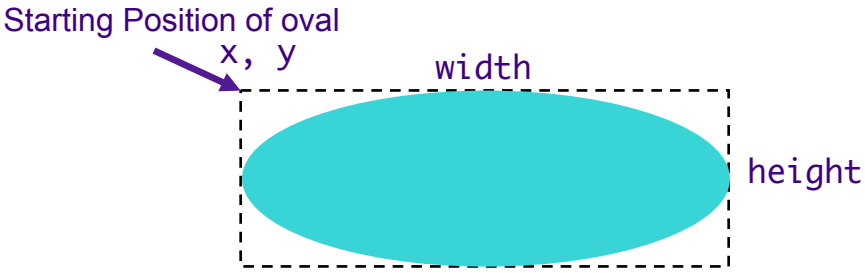
GRAPHICS PROGRAMMING

Graphics Object

- Abstract class
  - Implementation different for each platform
- A collection of settings for drawing images and text, such as colors and fonts
- Where used:
  - `paintComponent(Graphics g)`

Drawing Lines, Rectangles, Ovals

- Draw ovals, rounded rectangles within bounding rectangle



- Filled or outlined (e.g., `fillRect` vs `drawRect`)
- Can also draw arcs, polygons, polylines

Colors

- Colors made up of three components
  - Red, Green, Blue component
  - RGB values
    - Components: either 0 to 255 or 0.0 to 1.0
- `Color` class defines 13 color constants
  - `black`, `blue`, `cyan`, `darkGray`, `gray`, `green`, `lightGray`, `magenta`, `orange`, `pink`, `red`, `white`, and `yellow`
  - Also defined in all caps
  - See API

http://en.wikipedia.org/wiki/List_of_colors

Using `Graphics` object

1. Set the color/font
2. Draw the shape/string

```
public void paintComponent(Graphics g) {

   super.paintComponent(g);
   this.setBackground(Color.WHITE);

   // set new drawing color using integers
   g.setColor(new Color(255, 0, 0));
   g.fillRect(15, 25, 100, 20);
   g.drawString("Current RGB: " + g.getColor(), 130, 40);

   …
}
```

Understanding Code

Import existing Java project:
`/csdept/local/courses/cs209/handouts/screensavers.tar.gz`

- Simple Bouncers
  - How draws
  - How animates
- Screen Savers
  - What represents an object in the screen saver?
  - How generates screen saver objects?
  - How handles animation?
  - How handles events?