CSC207H: Software Design
Lecture 6

Wael Aboelsaadat

wael@cs.toronto.edu
http://ccnet.utoronto.ca/20075/csc207h1y/
Office: BA 4261
Office hours: R 5-7

Acknowledgement: These slides are based on material by Prof. Karen Reid

Software house: what happens inside?

Understand the requirements

Design the software

Write the program

Test the program

Write Documentation

Package/Sell/Market

How to produce a design?

- Identify classes
- Identify relations between classes

How to identify a class

- nouns → class
- verbs → methods
- adjectives → attribute/member-variables

How to identify relations ?

- Owns/has it?
- Uses it?
- Is type of ?

How good is a class definition?

- Completeness
  - class should captures all meaningful characteristics of an abstraction
- Sufficiency
  - Class provides enough characteristics of an abstraction to allow meaningful and efficient interaction
- Coupling
  - If an individual class is hard to understand, correct or change without referring to other classes, it said to be strongly coupled to another class, which is BAAAAD....

How good is a class definition?

- Cohesion
  - Class methods work together to provide well-defined behaviour, no unrelated elements or "coincidental cohesion"
- Primitiveness
  - Class should only provide primitive operations (clean+tidy or KISS)

Design Example

How to represent a design?

- We need a standard notation
- Unified Modelling Language (UML)

UML

- Unified Modeling Language
  - A way to draw information about program design

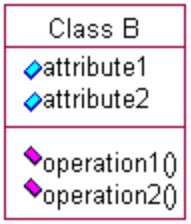
- The pictures we show here come from

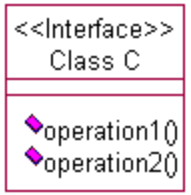
  http://www.dofactory.com/Patterns/Patterns.aspx

UML: class diagram

- *Types*
  - *Class*
  - *Interface*

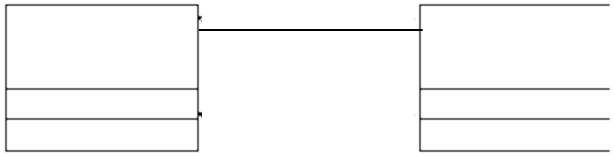
- *What is a valid class?*
  - *Type*
  - *Propertiers*
  - *Methods*
  - *Visibility (public, protected, private)*





UML: class diagram - relations

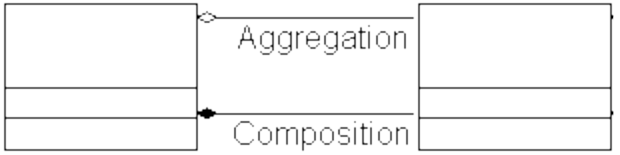
- *Association*



UML: class diagram - relations

- *Association*

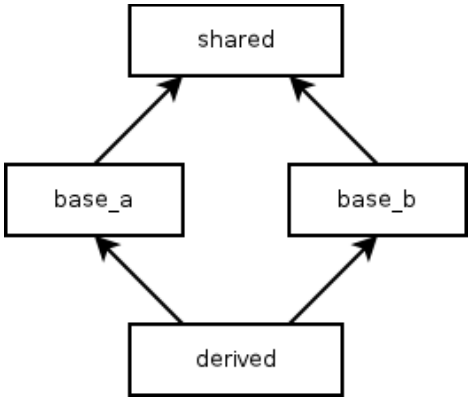
- *Aggregation*
  - *is a "part of" relationship*
  - *Lifetime* responsibility
  - *Contains-a*

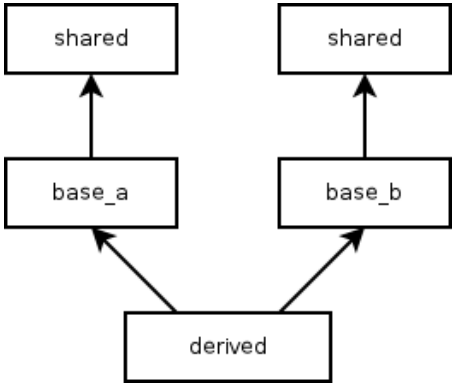


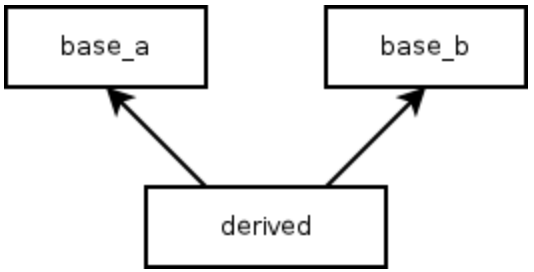
- *Composition*
  - No *lifetime* responsibility
  - *Has-a*

UML: class diagram - relations

- *Inheritance*
  - *Why?*
  - *Is-A relationship & exchangeable types*
  - *Types of inheritance*
    - *Single inheritance vs. multiple Inheritance*
    - *Single level vs. multi-level*
  - *Examples*
    - *Apple **is a** fruit*
    - *Door & Window?*
    - *What about manager, secretary, programmer & executive classes?*

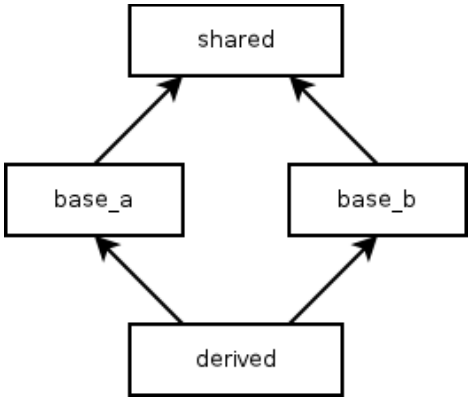


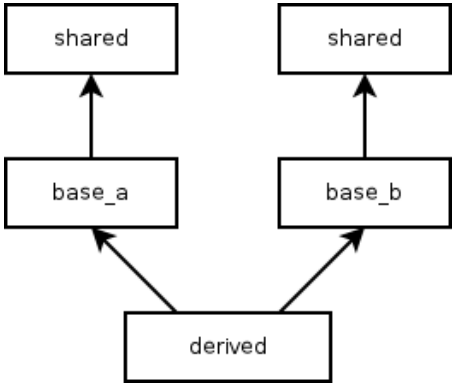


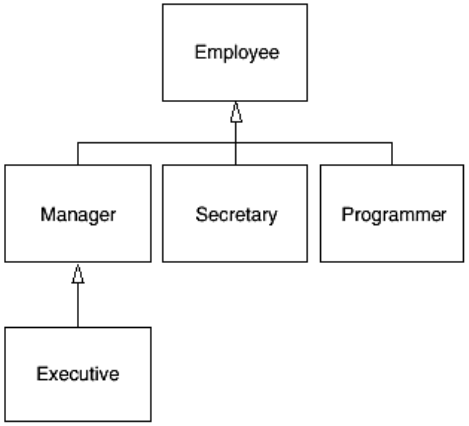


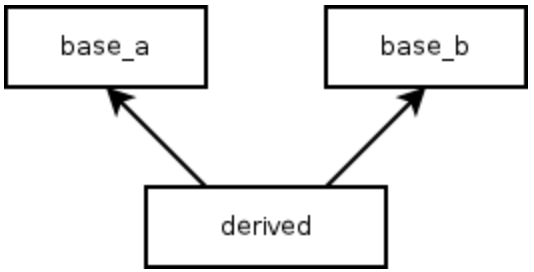
UML: class diagram - relations

- *Inheritance*
  - *Why?*
  - *Is-A relationship & exchangeable types*
  - *Types of inheritance*
    - *Single inheritance vs. multiple Inheritance*
    - *Single level vs. multi-level*
  - *Examples*
    - *Apple **is a** fruit*
    - *Door & Window?*
    - *What about manager, secretary, programmer & executive classes?*

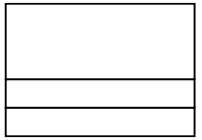
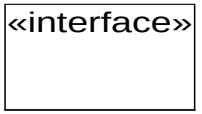
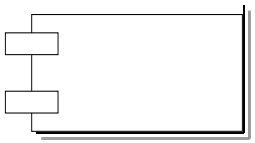
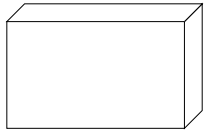




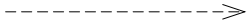




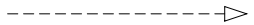
UML Notation

Construct	Description	Syntax
class	a description of a set of objects that share the same attributes, operations, methods, relationships and semantics.	
interface	a named set of operations that characterize the behavior of an element.	
component	a modular, replaceable and significant part of a system that packages implementation and exposes a set of interfaces.	
node	a run-time physical object that represents a computational resource.	

UML Notation

Construct	Description	Syntax
association	a relationship between two or more classifiers that involves connections among their instances.	
aggregation	A special form of association that specifies a whole-part relationship between the aggregate (whole) and the component part.	
generalization	a taxonomic relationship between a more general and a more specific element.	
dependency	a relationship between two modeling elements, in which a change to one modeling element (the independent element) will affect the other modeling element (the dependent element).	

UML Notation

Construct	Description	Syntax
realization	a relationship between a specification and its implementation.	

Tools in a Software House

- ✓ Programming Languages
- ✓ Scripting Languages
- • Integrated Development Environment (IDE) App
- • Profiling Tools
- ✓ Version Control App
- ✓ Quality Assurance Framework
- ✓ Software Build Management Framework
- • Requirements/Feature Tracking App
- • Variance Tracking App
- → Architecture Tools

Design & Architecture Tool

- Violet UML modelling app
  http://horstmann.com/violet/

Design Patterns

What is a Design Pattern

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“

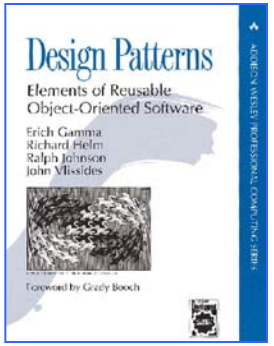
Elements of Design Patterns

- Pattern Name
  - Increases design vocabulary, higher level of abstraction
- Problem
  - When to apply the pattern
  - Problem and context, conditions for applicability of pattern
- Solution
  - Relationships, responsibilities, and collaborations of design elements
  - Not any concrete design or implementation, rather a template
- Consequences
  - Results and trade-offs of applying the pattern
  - Space and time trade-offs, reusability, extensibility, portability

What is a Design Pattern (II)

- Description of communicating objects and classes that are customized to solve a general design problem in a particular context.
- Each pattern focuses in a particular object-oriented design problem or issue
- Patterns describe the shape of code rather than the details

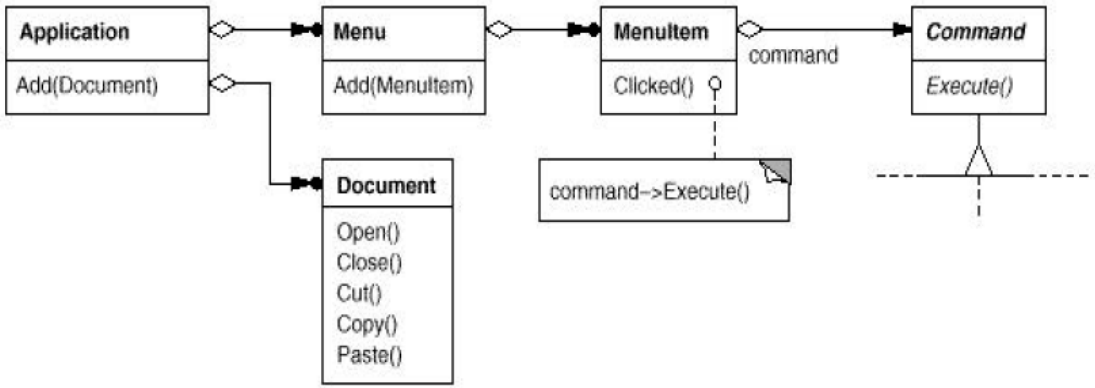
Design Pattern Space



Creational	**Structural**	**Behavioral**
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

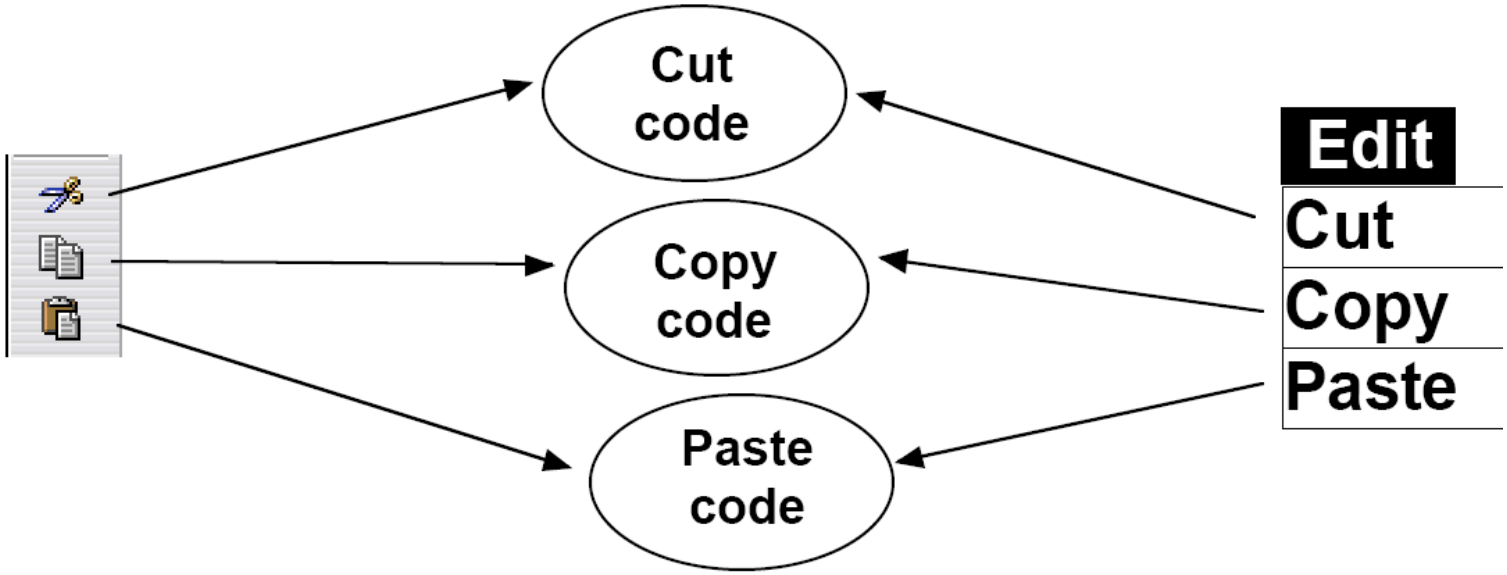
Pattern: Command

objects that represent actions



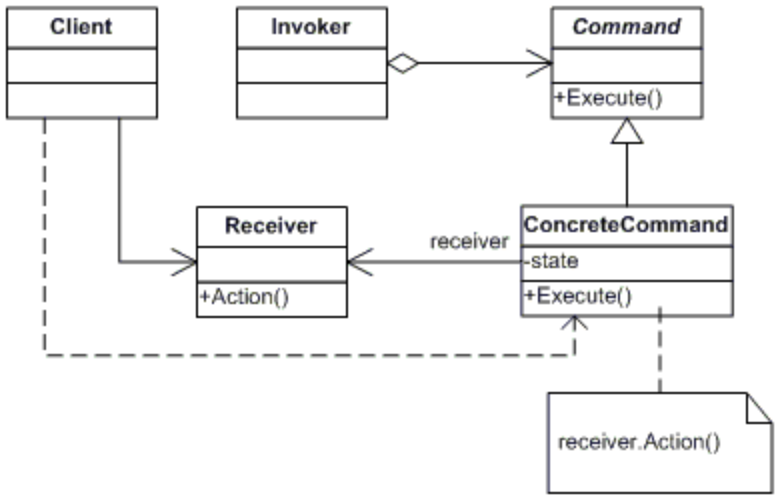
Common UI commands

- it is common in a GUI to have several ways to activate the same behavior
  - example: toolbar "Cut" button and "Edit / Cut" menu
  - this is *good* ; it makes the program flexible for the user
  - we'd like to make sure the code implementing these common commands is not duplicated

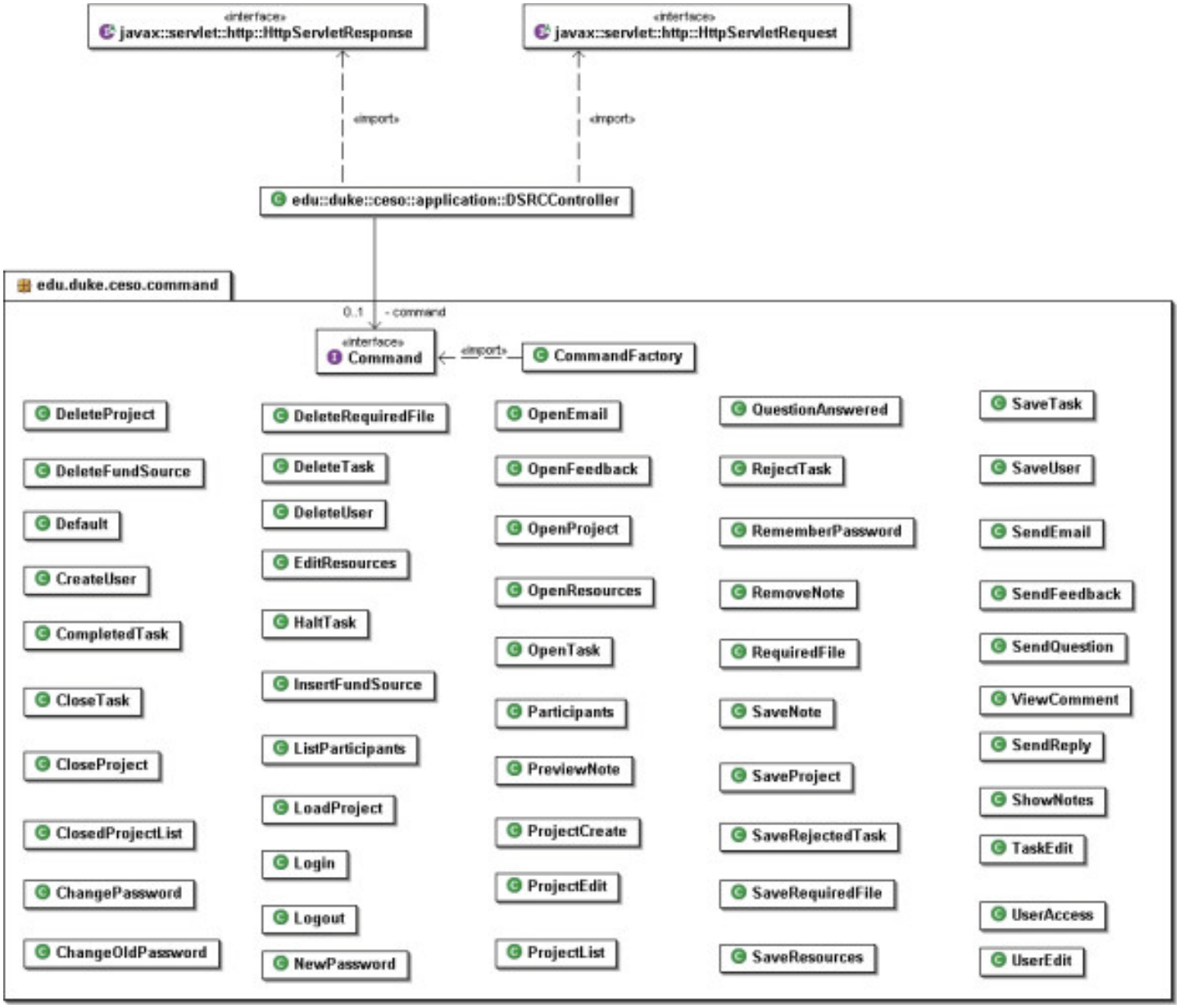


Command pattern

- **Command**: an object that represents an action
  - sometimes called a "functor" to represent an object whose sole goal is to encapsulate one function



Command pattern - example



Pattern: Singleton

At max One Instance of a class!

Singleton Pattern

- Used to ensure that a class has only one instance. For example, one printer spooler object, one file system, one window manager, etc.

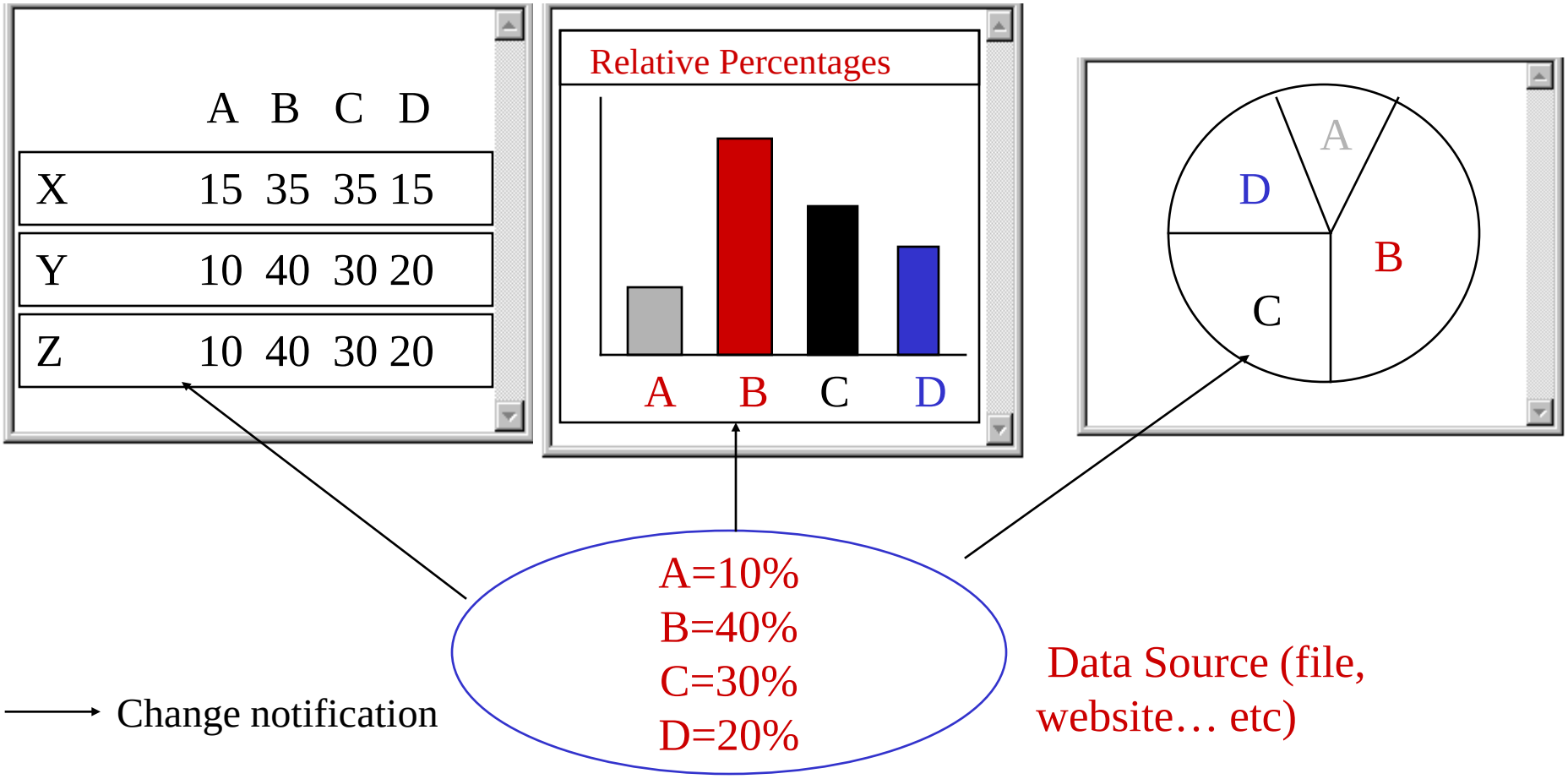
- Instead the class itself is made responsible for keeping track of its instance. It can thus ensure that no more than one instance is created. *This is the singleton pattern.*

Singleton example code

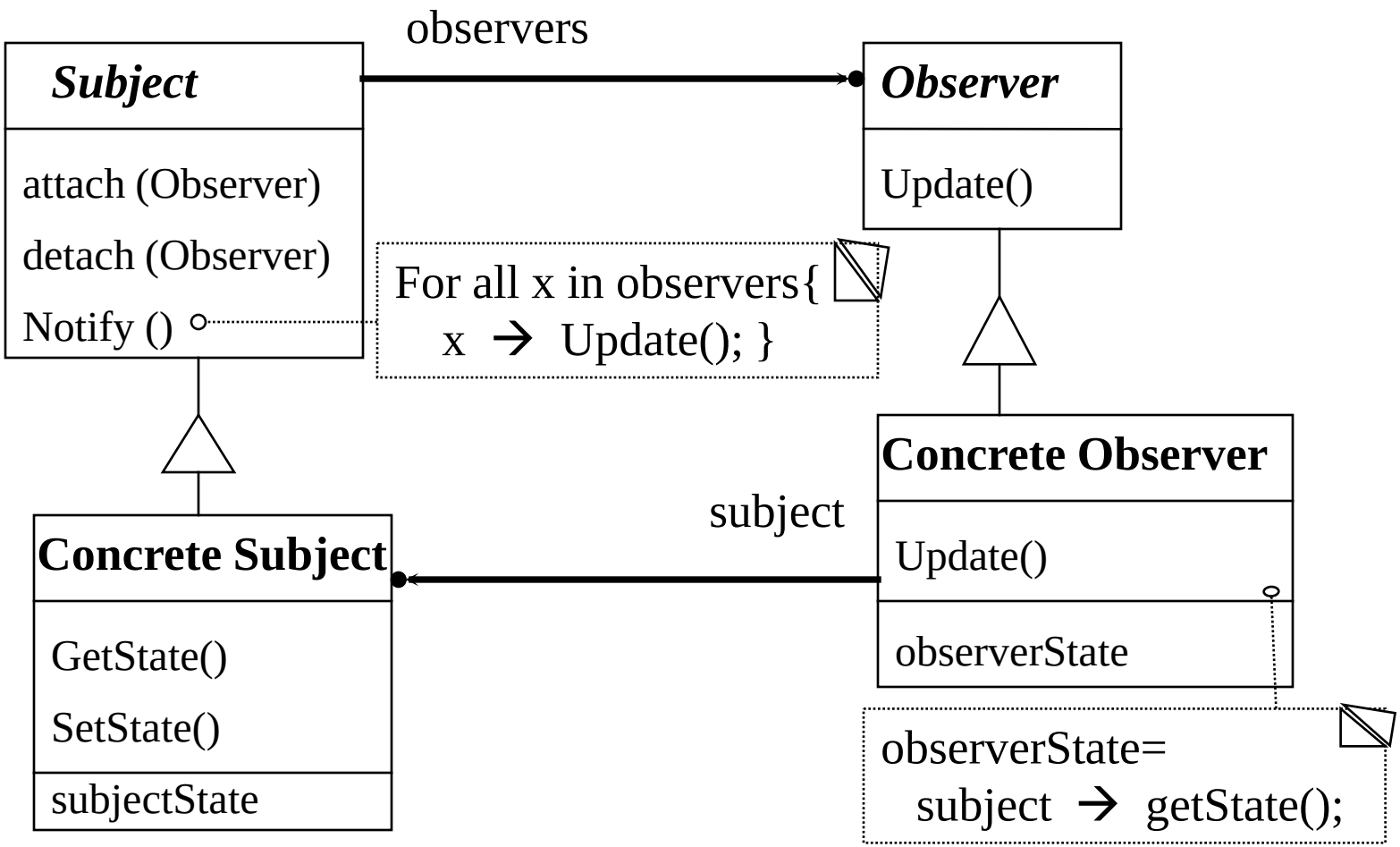
```
public class MySingletonClass {

  private static MySingletonClass instance
    = new MySingletonClass();

  public static MySingletonClass getInstance()
  {
       return instance;
  }

    /** There can be only one. */
  private MySingletonClass() {}

}
```

Pattern: Observer

Observer Pattern



Observer Pattern



Observer Pattern

- Need to separate presentational aspects with the data, i.e. separate views and data.
- Classes defining application data and presentation can be reused.
- Change in one view automatically reflected in other views. Also, change in the application data is reflected in all views.
- Defines one-to-many dependency amongst objects so that when one object changes its state, all its dependents are notified.

Observer Pattern

- GUI programming example

Pattern: Template Method

What’s Wrong With This?

```
public class PizzaMaker {
  public void cookPizzas(List pizzas) {
    for (int i=0; i<pizzas.size(); ++i) {
      Object pizza = pizzas.get(i);
      if (pizza instanceof ThinCrustPizza) {
        ((ThinCrustPizza)pizza).cookInWoodFireOven();
      }
      else if (pizza instanceof PanPizza) {
        ((PanPizza)pizza).cookInGreasyPan();
      }
      else {

      }
    }
  }
}
```

The Open-Closed Principle

- *Classes should be open for extension, but closed for modification*
  - I.e., you should be able to extend a system *without* modifying the existing code
- The type-switch in the example violates this
  - Have to edit the code every time the marketing department comes up with a new kind of pizza

Abstraction is the Solution

- Solve the problem by creating a `Pizza` interface with a `cook` method
  - Or an abstract base class whose cook method must be overridden by every child

- Simple, right?

How Open Should You Be?

```
public abstract class Pizza {
  public final void cook() {
    placeOnCookingSurface();
    placeInCookingDevice();
    int cookTime = getCookTime();
    letItCook(cookTime);
    removeFromCookingDevice();
  }
  protected abstract void placeOnCookingSurface();
  protected abstract void placeInCookingDevice();
  protected abstract int  getCookTime();
  protected abstract void letItCook(int min);
  protected abstract void removeFromCookingDevice();
}
```

Template Method Design Pattern

- The *Template Method* design pattern is used to set up the skeleton of an algorithm
  - Details then filled in by concrete subclasses

- But what if someone wants to do something you didn't anticipate?
  - E.g., wants to add a `PancakePizza` that has to be flipped over halfway through the cooking process

Override the Template Method?

```
public final void cook() {
  placeOnCookingSurface();
  placeInCookingDevice();
  int cookTime = getCookTime();
  letItCook(cookTime/2);
  flip();
  letItCook(cookTime/2);
  removeFromCookingDevice();
}
```

– But `cook` was `final`

– And it's storing up trouble for the future

Squeeze It Somewhere Else?

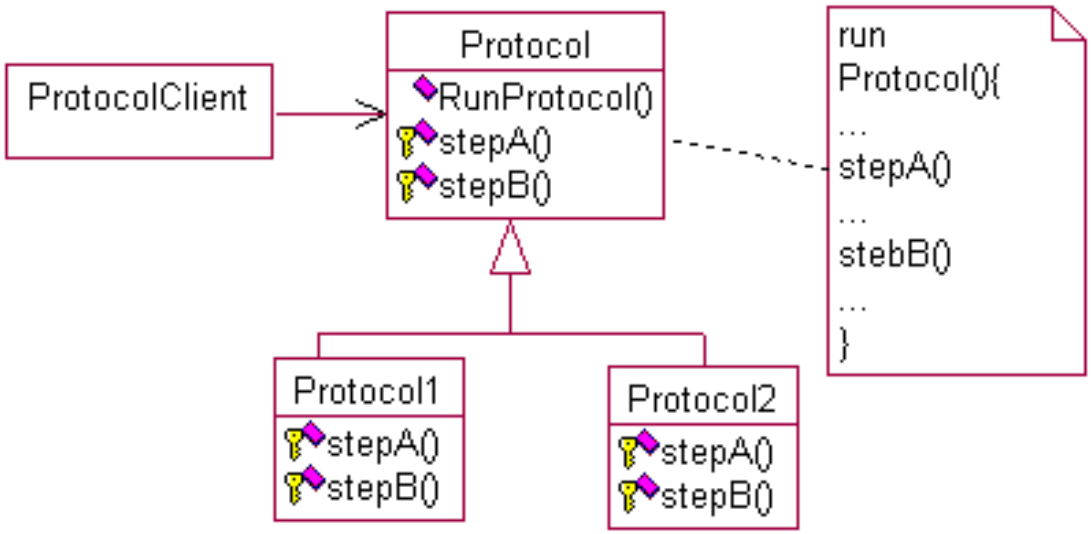
```
protected void removeFromCookingDevice() {
  flip();
  letItCook(cookTime);
  …remove from skillet…
}
```

- `removeFromCookingDevice` shouldn't be doing other things
  - Think about the documentation
- And once again, we're storing up trouble for the future

Leave Space for Future Growth?

```
public final void cook() {
  beforePlacingOnCookingSurface();
  placeOnCookingSurface();
  beforePlacingInCookingDevice();
  placeInCookingDevice();
  beforeCooking();
  for (int i=0; i<getCookingPhases(); i++) {
    letItCook(getCookTime(i));
    afterCookingPhase(i);
  }
  beforeRemovingFromCookingDevice();
  removeFromCookingDevice();
  afterRemovingFromCookingDevice();
}
```

Template Method Pattern



Design Patterns: discussion

- Not just about object-oriented design
  - User interface patterns
  - Business patterns
  - Anti-patterns (things to avoid)

- Be careful, not every coding problem is a design pattern

Serves Two Purposes

- **Communication**: a concise way for designers to communicate with each other
  - And argue out exactly what they mean
  - Often without worrying about specific implementation details

- **Education:** gives them a way to communicate what they know to newcomers
  - Don't expect to connect them all to your own experience the first time
  - But keep them in mind as you work on other courses
  - "Hey, I know how to do this!“

Are We Winning?

- You can’t tell if your designs are any good until you can tell good from bad

- Presume you know how to tell good *code* from bad
  - Indentation, variable naming, documentation, unit tests, etc.
  - From here on in, you can only *lose* marks for doing it badly