- **CSCI 361**
- Course run by Prof. Sterling and Prof. Boranbayev
- *Teaching Assistant*: Asset Berdibek, Adil Sarsenov
- *Contact Info*: Mark Sterling, Office 7E440

Objectives

- Review some concepts from Object-Oriented programming
- Review relationships in UML class diagrams
- Introduce the notion of a *design pattern*
- Begin to show examples of patterns: *composite*, *singleton*

Required Readings

NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

- Readings accompanying these topics are:
  - Booch 1986[1]
  - Chapters 1 (OOA/D and Design), 9 (Domain Models), 16 (UML Class Diagrams), and 17 (GRASP) of the text[2]
  - *Software Engineering Radio Ep. 215*. Podcast. 2014[3]

---

[1] Grady Booch. "Object-oriented development". In: *IEEE transactions on Software Engineering* 2 (1986), pp. 211–221.

[2] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

[3] http://www.se-radio.net/2014/11/episode-215-gang-of-four-20-years-later/

Classes and Objects

- *Classes* are the basic unit of OO programming
- A class is like a blueprint for the creation of *objects*: particular instances of classes
- An object is defined by three pieces of data
  - Class to which it belongs (the *type*)
  - A state that is captured by a set of attributes or fields
  - A collection of methods (or functions) that operate on the state of an object

Low Representational Gap

- Object-orientation is about mental models
- Objects are supposed to be "like" the representations human beings use to understand the world
- *Low Representational Gap* is the idea that *conceptual* categories and *programming* categories can converge

Grady Booch, 1986[4]

Abstraction and information hiding are actually quite natural activities. We employ abstraction daily and tend to develop models of reality by identifying the objects and operations that exist at each level of interaction.

[4] Grady Booch. "Object-oriented development". In: *IEEE transactions on Software Engineering* 2 (1986), pp. 211–221.

- The **Unified Modeling Language** (UML)[5][6][7] is a modeling language that can help you to document and communicate designs in Software Engineering
- Different types of diagrams to represent different views of the design: *Behavior*, *Structure*, *Communication*
- UML is a huge specification, in this course we will mostly look at Class, Sequence, and Use Case diagrams

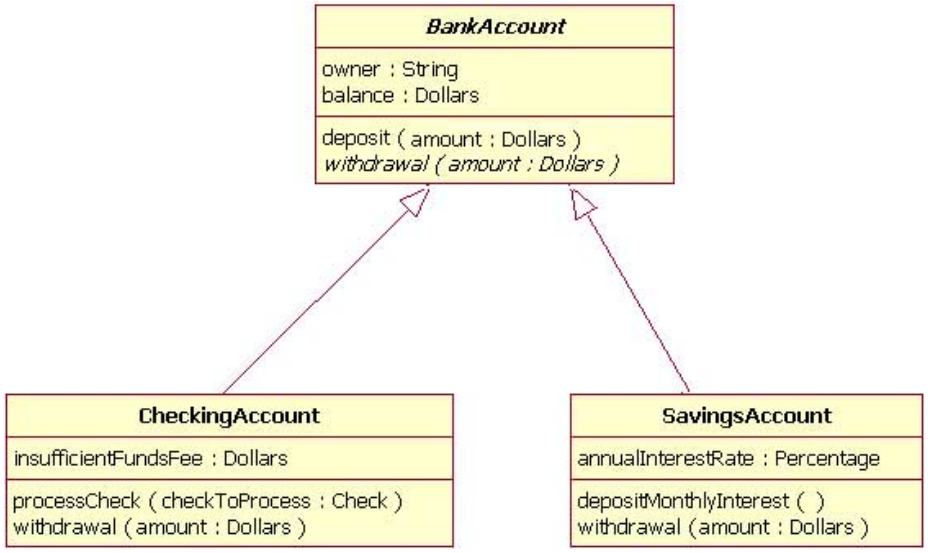
---

[5] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

[6] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[7] Paul Gestwicki. *All the UML you need to know*.

Class Diagrams in UML



- Provides the static relationships between different classes
- Each class is represented by a box with three sections giving: Class names, fields, and methods
- Arrows between the classes indicate various kinds of relationships between the classes

HAS-A and IS-A relationships

- Two of the most important kinds of relationships between classes are *inheritance* or *composition* (aggregation)
- Inheritance: one class is a specialization or generalization of another class (IS-A) Cat $\rightarrow$ Animal
- Composition: one class contains or uses another class (HAS-A) Car $\rightarrow$ Engine
- In UML relationships are represented by different types of lines and arrowheads
- "Composition over Inheritance" is a phrase that you will hear sometimes in software engineering

Polymorphism

- Any particular instance of an class may be treated as having a different type depending on its subtyping relationships
- A subtype automatically has the type of its superclasses
- In certain situations, objects of classes that share a common superclass can be treated as if they were all of the same type
- Polymorphism and typing relationships are commonly used to create code that is more flexible, behaviors can change at run-time
- (Also applies to interfaces)

Using Polymorphism

- Typing relationships can be exploited by writing programs against the more general classes that appear in our design
- With polymorphism we can *Program to an Interface*

```
// Hard coded
Horse h = new Horse();
h.gallops();
Person p = new Person();
p.walks();

// Polymorphic
Actor a1 = new Horse();
Actor a2 = new Person();
a1.move();
a2.move();
```

Delegating Responsibilities[8]

- One class will typically contain references to other classes for the purpose of achieving more complex behavior
- Class delegates its responsibilities to other objects

```
public class Person {
      private String name;
      private House h;
      public Person(String s, House h) {
            this.name = s;
            this.h = h;
      }
      public Object someMethod() {
            return h.anotherMethod();
      }
}
```

[8] *delegation pattern*

Design Patterns

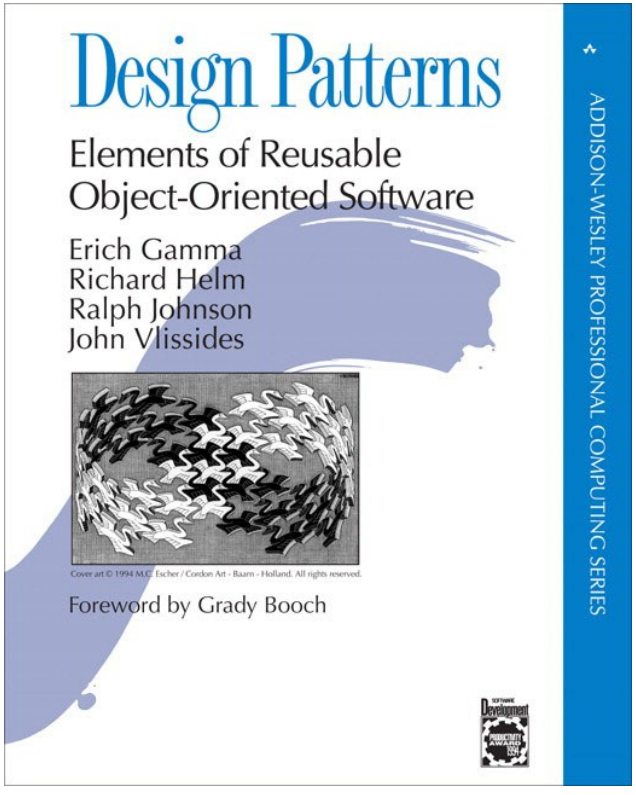
NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

- Polymorphism, inheritance, composition, etc. are the basic tools that we, as programmers, can use
- *Design Patterns*[9]: an idea from architecture that was applied to software development
- Think of patterns as the time-tested solutions that are *discovered* rather than *invented*
- Design patterns indicate how classes should be structured to solve a particular problem (don't provide implementation)
- Heavily exploit polymorphic code and delegation to make behaviors reconfigurable at runtime

[9] Eric Freeman et al. *Head first design patterns*. O'Reilly Media, Inc., 2004; Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

The *Gang of Four* Book

- The Design Patterns book by Gamma, Helm, Johnson, Vlissides (the so-called gang-of-four) is one of the most influential OO programming texts
- Originally published in 1994, has since been reprinted many times and translated to a variety of languages



Construction

- Software Engineering is also primarily concerned with *construction*

Design Patterns (Definition)[10]

NAZARBAYEV UNIVERSITY
SCHOOL OF SCIENCE AND TECHNOLOGY

Design Pattern

Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

- Consist of: *name*, *Problem Description*, *solution* (no implementation)
- Types of Patterns
  - *Creational*
  - *Behavioral*
  - *Structural*
    - Example: *Composite*

[10] Erich Gamma. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

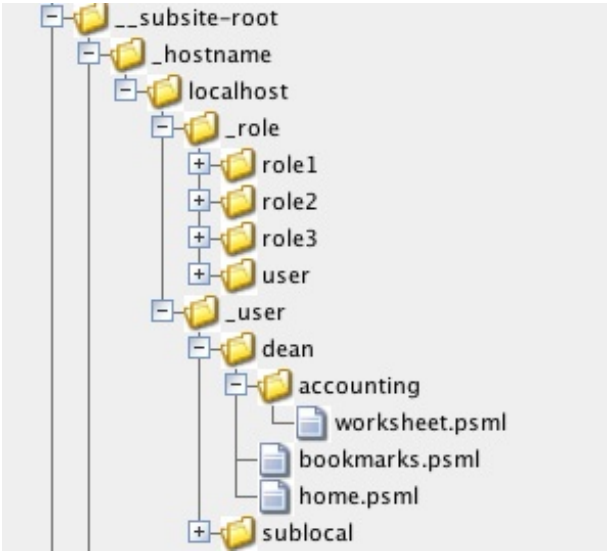
Expertise

- Patterns Catalog the accumulated knowledge of a group of experts working on OO software design
- *Simula* (1967): first object-oriented design language
- *C++*, *Smalltalk* (1980-1990s): dominant programming paradigm due to influence of graphical user interfaces

Patterns: Pros and Cons

- There are pros and cons to learning and working with patterns
- *Pros*
  - Understanding how patterns work forces you to come to grips (i.e. internalize) aspects of object-orientation
  - Useful shared vocabulary when communicating with other developers
  - Patterns are used in many standard libraries and, in certain cases, have graduated to language features (*iterator* and *decorator* in Java)
- *Cons*
  - Temptation to apply them indiscriminately, overdesign a system
  - Can make code more difficult to understand, for example, adding layers of indirection
  - Usefulness of certain patterns is debated: *singleton*

Example the Composite Pattern



- *Composite*: way of building tree-like data-structures
- Example: file-system, directory tree

Designing a Class for Arithmetic Expressions

- How can we use the composite pattern to build an object to represent simple arithmetic expressions like $(1 + 3) * (5 - 2)$
- Where is the tree-like structure in this example?

Classes to Define

- Overall supertype: `Term`
- Atomic value type: `Vals`
- Operation type: `Opers`
  - Subtypes to represent different types of binary operators

The `Term` or expression type

- The supertype defines an *interface* by which we can access both of the concrete classes in our implementation

```
abstract class Term {
  public abstract double getValue();
}
```

The Vals or leaf type

- The leaf type provides a concrete implementation of the method to return a value and has a field to store a value

```
class Vals extends Term {
  private double myValue;

  public Vals(double m) {
    this.myValue = m;
  }

  public double getValue() {
    return myValue;
  }
}
```

The Opers or operation type

- The leaf type provides a concrete implementation of the method to return a value and has a field to store a value

```
abstract class Opers extends Term {
  protected Term leftVal;
  protected Term rightVal;

  public Opers(Term l, Term r) {
    this.leftVal = l;
    this.rightVal = r;
  }
}
```

A concrete operation

- Create a new operation to do addition

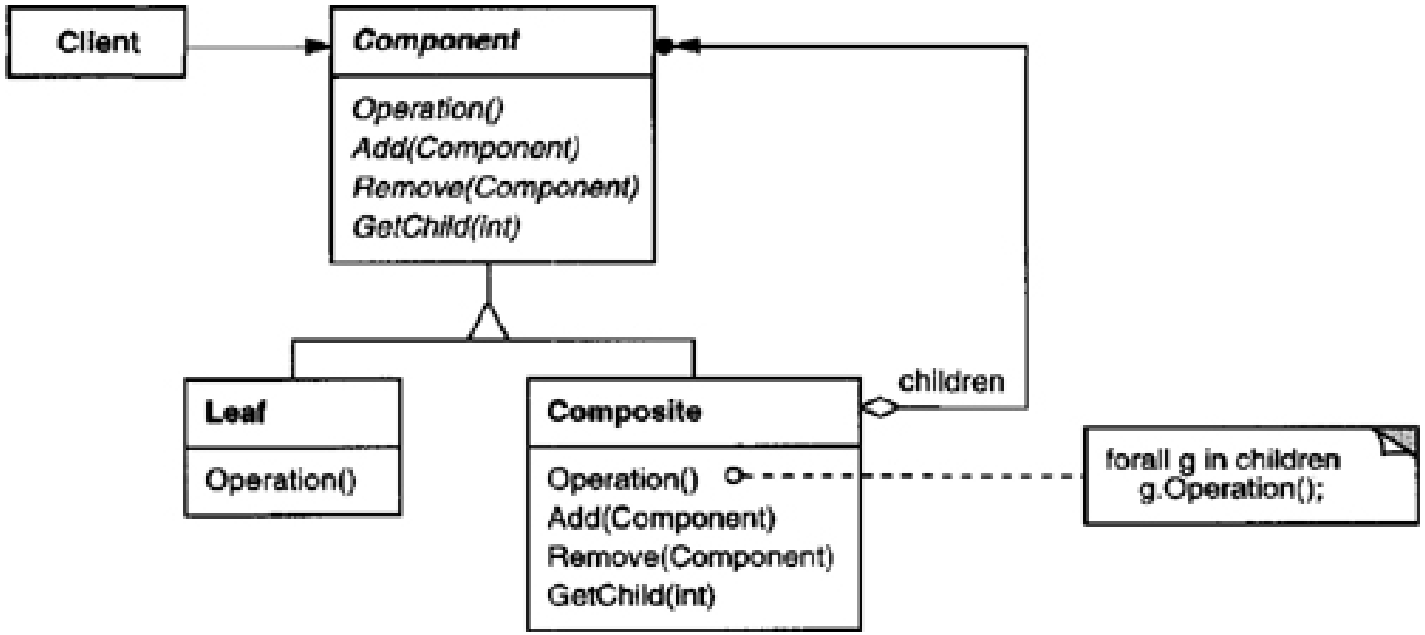
```
class Adding extends Opers {

  public Adding(Term l, Term r) {
    super(l, r);
  }

  @Override
  public double getValue() {
    return this.leftVal.getValue() +  ...
    ...  this.rightVal.getValue();
  }
}
```

Using the Design

- What happens when we call `getValue()` on the root node of some complicated expression?

```
...
Multiplying x = new Multiplying(
  new Adding(new Vals(2.0), new Vals(2.0)),
    new Adding(new Vals(2.0),
      new Multiplying(new Vals(2.0), new Vals(2.0))
    )
  );
...
```

Composite Design Pattern (Full Description)[11]



- The composite pattern allows us to easily create and reconfigure tree-like structures at run-time
- In example: the tree was the syntax tree of simple arithmetic expressions

[11] Class diagram from ibid.

Achieving Composite with Java Collections

- The Expression example did not have the `add()` and `remove()` methods and only allowed binary trees
- Java Collections allow us to add these behaviors and allow for an arbitrary number of children

```
class TreeRoot extends TreeComponent {
  ArrayList<TreeComponent> treeComponents = new ArrayList<TreeComponent>();

  public TreeRoot(String name) {
    this.name = name;
  }

  // ...

  public void add(TreeComponent treeComponent) {
    treeComponents.add(treeComponent);
  }

  public void remove(TreeComponent treeComponent) {
    treeComponents.remove(treeComponent);
  }

  public TreeComponent getChild(int i) {
    return (TreeComponent)treeComponents.get(i);
  }
}
```

Summarizing Arithmetic Expression Class

- Our example composite implementation contained 3 classes: an atomic value type, a binary operation type, and an overall super-type (`Vals`,`Opers`,`Term`)
- Build up complicated expressions by nesting constructors
- We have a `getValue()` method that calculated an expression recursively
- We might also create a method `getDepth()` that returns the depth of an expression

The maven project structure

- The structure of a typical maven project is shown below
- `pom.xml` is the *project object model*, the main configuration for your project

```
/Project Root/
  |--/src/
  |     |--/main/
  |     |     |--/java/
  |     |     |--/webapp/
  |     |--/test/java/
  |--README.txt
  |--LICENSE.txt
  |--NOTICE.txt
  |--pom.xml
```

The Singleton Pattern

- The singleton pattern is used when we want to have a unique instance of a class and provide a global point of access to it.
- The pattern itself only consists of a single class
- Classic implementation of the singleton uses a private constructor and a static variable

Singleton

"Ensure a class has only one instance, and provide a global point of access to it." *GoF*

Restricting Object Creation


```
public class Singleton {
    //

    private Singleton() {}

    //
}
```

- The classic singleton implementation uses a `private` constructor
- What are the implications of this?

Providing access to the singleton instance


```
public class Singleton {

    private Singleton() {}

    public static Singleton getInstance() {}
}
```

- What should be the contents of the `getInstance()` method?

The getInstance() method


```
public class Singleton {

    private Singleton() {}

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

- What *type* of static variable do we need, and how can we ensure that only one object can be created.

The Complete Singleton Implementation


```
public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
            }

        return uniqueInstance;
        }
}
```

- Here we have all of the basic elements of the singleton pattern

Issues with Singleton

- Uses
  - Can be useful if we need some kind of global state in an application
  - Factory patterns often employ a singleton
- Issues
  - Usefulness or appropriateness of singletons are debated
  - Sometimes called an *antipattern*
  - Issue with multi-threading

Dealing with multi-threading


```
public static synchronized Singleton getInstance() {
   if (uniqueInstance == null) {
      uniqueInstance = new Singleton();
      }

   return uniqueInstance;
   }
```

- One option to deal with multi-threading problems is to declare the `getInstance()` method `synchronized`
- The synchronized keyword basically ensures that method calls are atomic

Homework, Due in lab Aug, 17

- Complete the implementation of the arithmetic expression composite pattern described in this slide set
  - `getValue()`: return the value of a particular term (computed recursively)
  - `getDepth()`: same as `getValue()` but returns the depth of the syntax tree instead of its value (as a double)
- Download the required software: Java JDK, Eclipse, maven, and git
- Make a github profile
- Decide about forming teams with your peers (required size of teams will be 5-6 people)