Objectives

- Briefly review the ACM/IEEE Software Engineering Code of Ethics and Professional Practice
- Review some concepts from Object-Oriented programming
- Review relationships in UML class diagrams
- Introduce the notion of a *design pattern*
- Show an example of a pattern: *composite*

Required Readings

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- 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/

Code of Ethics in Software Engineering

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- A joint task force of the ACM and IEEE created a guide to ethics consisting of 8 principles
- Think of the code as a *tool* rather than a *proscription*

ACM/IEEE Code of Ethics (Paraphrased)[4]

1. PUBLIC act in the public interest
2. CLIENT AND EMPLOYER act in the best interests of the client
3. PRODUCT maintain high quality standards in software products
4. JUDGEMENT integrity in professional judgments
5. MANAGEMENT ethical approach to management of software projects
6. PROFESSION maintain reputation of the profession
7. COLLEAGUES treat your peers and colleagues with respect
8. SELF continual learning over the lifetime in the profession

[4] Don Gotterbarn, Keith Miller, and Simon Rogerson. "Software engineering code of ethics". In: *Communications of the ACM* 40.11 (1997), pp. 110–118.

Other Ideas about Ethics

- Plato: "No one does wrong willingly."
- Consequentialism
  - "The greatest good for the greatest number."
- Non-Consequentialism
  - Kant's *Categorical Imperative*: "I ought never to act except in such a way that I could also will that my maxim should become a universal law."

Current Issues in Software

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

- 737 Max Failures
- Privacy and Free Speech on social media
- 2016 US Elections
  - Secure systems (Clinton/DNC email hacking)
  - Influence of questionable news sources
- General Data Protection Regulation (GDPR)
  - *right to be forgotten*
  - *right to explanation*
- Recent headline: "HUD complaint accuses Facebook ads of violating Fair Housing Act"[5]

[5] https://techcrunch.com/2018/08/19/hud-complaint-accuses-facebook-ads-of-violating-fair-housing-act/

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[6]

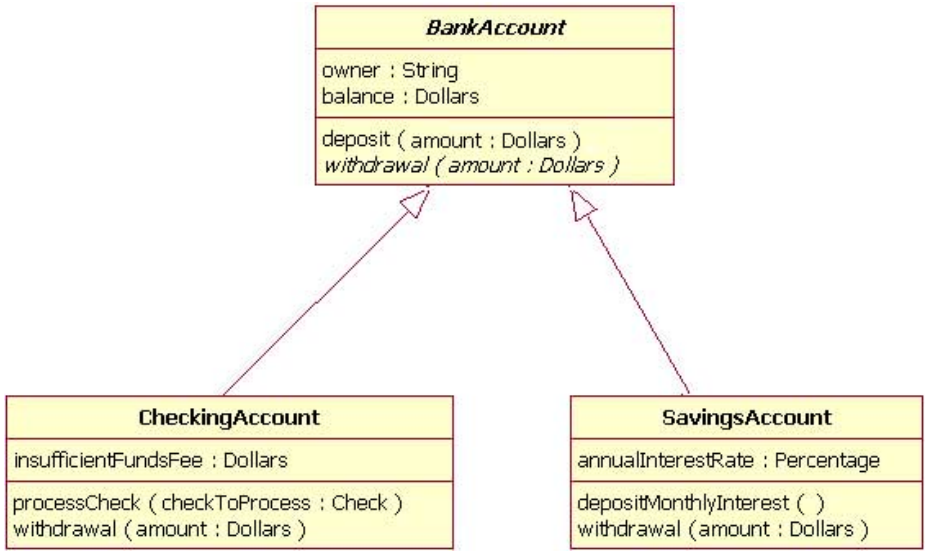
Abstraction and information hiding are actually quite natural activities. We employ abstraction daily and tend to develop models of reality by identifying the objects and operations that exist at each level of interaction.

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

- The **Unified Modeling Language** (UML)[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 examine a subset of the kinds of UML diagrams

[7] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004; Paul Gestwicki. *All the UML you need to know*. Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design and Interative Development*. Prentice Hall PTR, 2012.

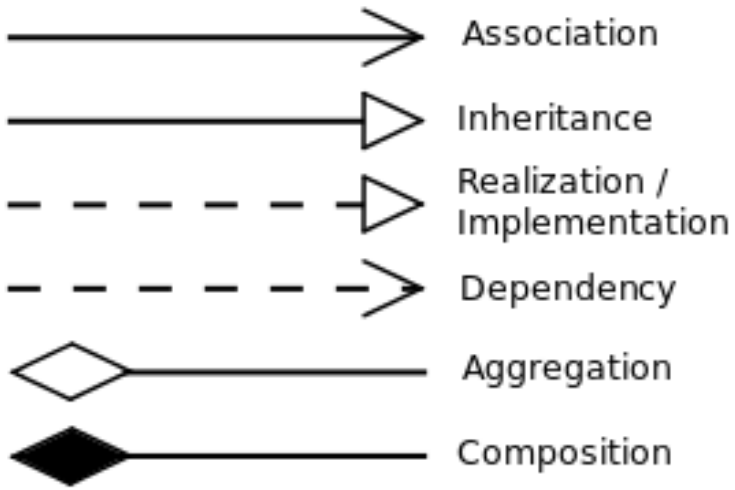
Class Diagrams in UML[8]



- Provides the static relationships between different classes
- Each class is represented by a box with three sections giving: Class names, fields, and methods
  - Member visibility can be shown by the symbols $+$, $-$, $\#$, $\sim$
- Arrows between the classes indicate various kinds of relationships between the classes

[8] image: `https://developer.ibm.com/articles/the-class-diagram/`

Class Diagrams Associations[9]



- Lines and arrowheads have specific meaning in UML class diagrams
- A diamond usually indicates a member variable or a property
- A triangular arrowhead indicates subtype (generalization)
  - Dashed lines usually point to interfaces

[9] image: https://upload.wikimedia.org/wikipedia/commons/9/93/Uml_classes_en.svg

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 often considered helpful design advice

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, and to allow behavior that can change at run-time
- (Also applies to interfaces)

Using Polymorphism (1)

- 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.gallop();
Person p = new Person();
p.walk();

// Polymorphic
Creature c1 = new Horse();
Creature c2 = new Person();
c1.move();
c2.move();
```

Using Polymorphism (2)


```
class Horse implements Creature {}
class Person implements Creature {}

public static void direct(Creature c) {
    c.move();
}
```

- When we take the time to program to an interface our code becomes more flexible
- With our `direct` method, we can direct any object the implements the `Creature` interface

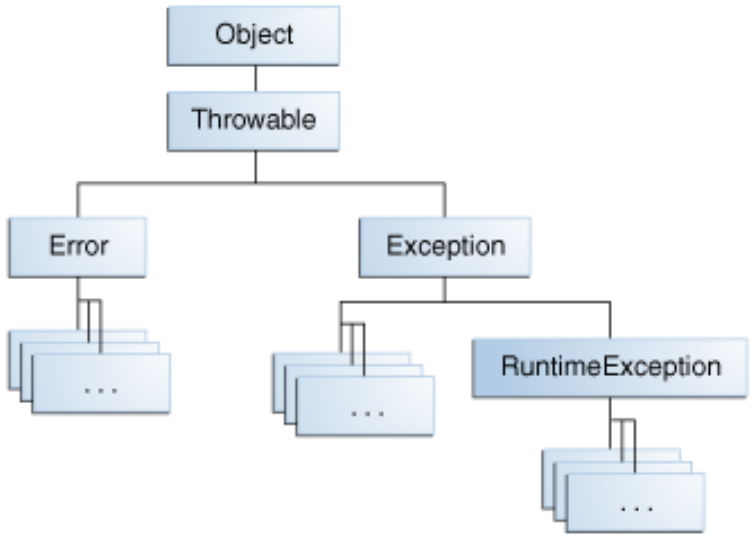
Delegating Responsibilities[10]

- 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();
      }
}
```

[10] *delegation pattern*

Throwable[11]



- The `Throwable` classes in Java
- Errors
  - Serious problems thrown by the VM
- Checked Exceptions
  - Subclasses of `Exception` other than `RuntimeException`
- Unchecked Exceptions
  - Subclasses of `RuntimeException`

[11] https://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html

Using Exceptions (1)


```
try {
  throw new MyException("Message");
} catch (MyException e) {
  // do something to handle exceptional case
}

// Method throws a Checked Exception
void myMethod() throws Exception {
  throw new Exception();
}
```

- "Throw" and exception with the `throw` keyword and handle exceptions with *try-catch* blocks
- Declare that a method might throw an exception with the keyword `throws`
- Checked exceptions either need to be handled or declared

Using Exceptions (2)

- Use each of the three types of exceptional cases appropriately, there are typical cases
- **Errors**: As a general rule do not try to handle or declare an `Error`, they are usually a sign that something has gone seriously wrong
  - `StackOverflowError`, you wrote an infinite recursion
- **Checked Exceptions**: Exceptions that can be reasonably expected even if there are no programming errors and that could be recovered from (don't overdo these)
  - `IOException`
  - `FileNotFoundException`
- **Unchecked Exceptions**: Exceptions that are caused by programming erros
  - `IllegalArgumentException`
  - `ArrayIndexOutOfBoundsException`

Passing Data to Methods


```
public static void main(String[] args) {
    String a = "Alice";
    myMethod(a);
    System.out.println(a);
}

public static void myMethod(String name) {
    name = "Bob";
}
```

- Make sure you understand the pass-by-value semantics of the Java language (see the example above)
- Briefly, assignments won't affect the caller but calling methods on a passed object will affect the caller

Variable argument parameter


```
public static void main(String[] args) {
    find("Alice");
    find("Alice", "Bob");
    find(new String[] {"Alice", "Bob"});
    find();
}

public static void find(String... names) {
    // names : String[]
    System.out.println(names.length);
}
```

- A variable argument parameter is indicated by three periods
- You can either pass an array or list the elements

Best Practices

- Always use packages to organize your code (don't use the default package)
- Try to keep one class per file
- Avoid `public` members where possible, provide access through well-named (e.g. Javabeans) getters and setters
- Program to an interface as often as you can
  - Only use `new` when essential
- Pass the necessary data needed to build instances in the constructor (dependency injection)

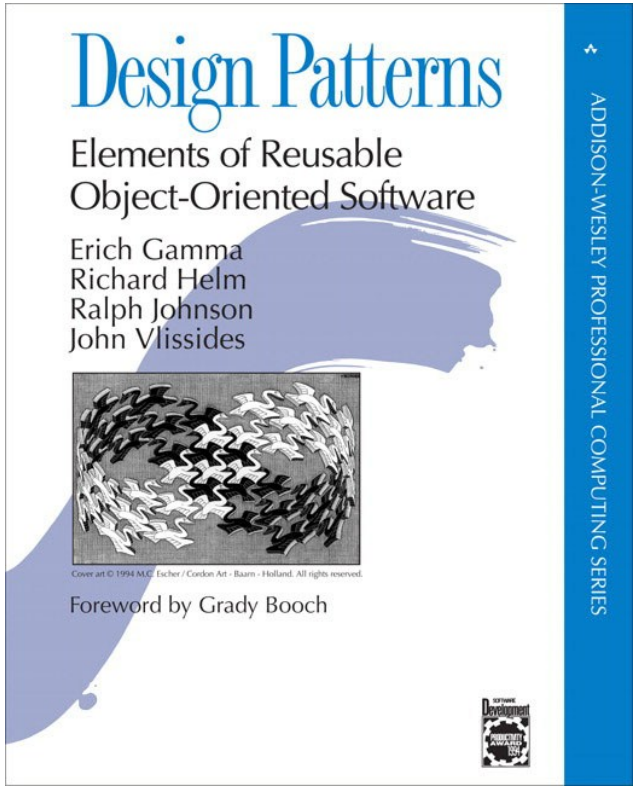
Design Patterns

- Polymorphism, inheritance, composition, etc. are the basic tools that we, as programmers, can use
- *Design Patterns*[12]: 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)
- Leverage polymorphic code and delegation to make behaviors reconfigurable at runtime

[12] 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



Design Patterns (Definition)[13]

NAZARBAYEV UNIVERSITY
SCHOOL OF ENGINEERING AND DIGITAL SCIENCES

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*

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

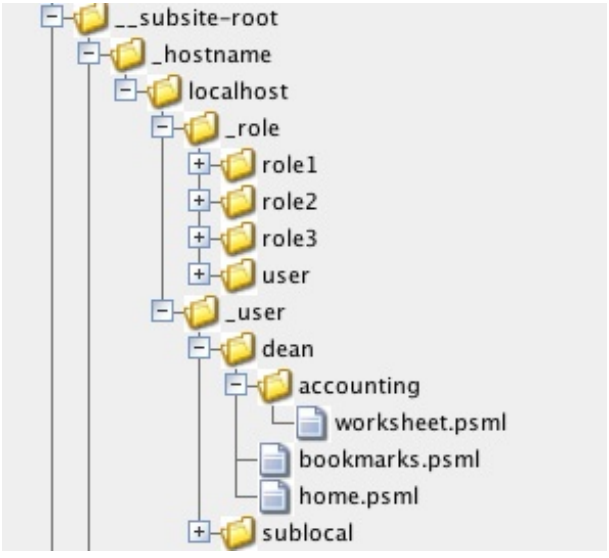
Expertise

- 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

- *Pros*
  - Understanding how patterns work forces you to understand many 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

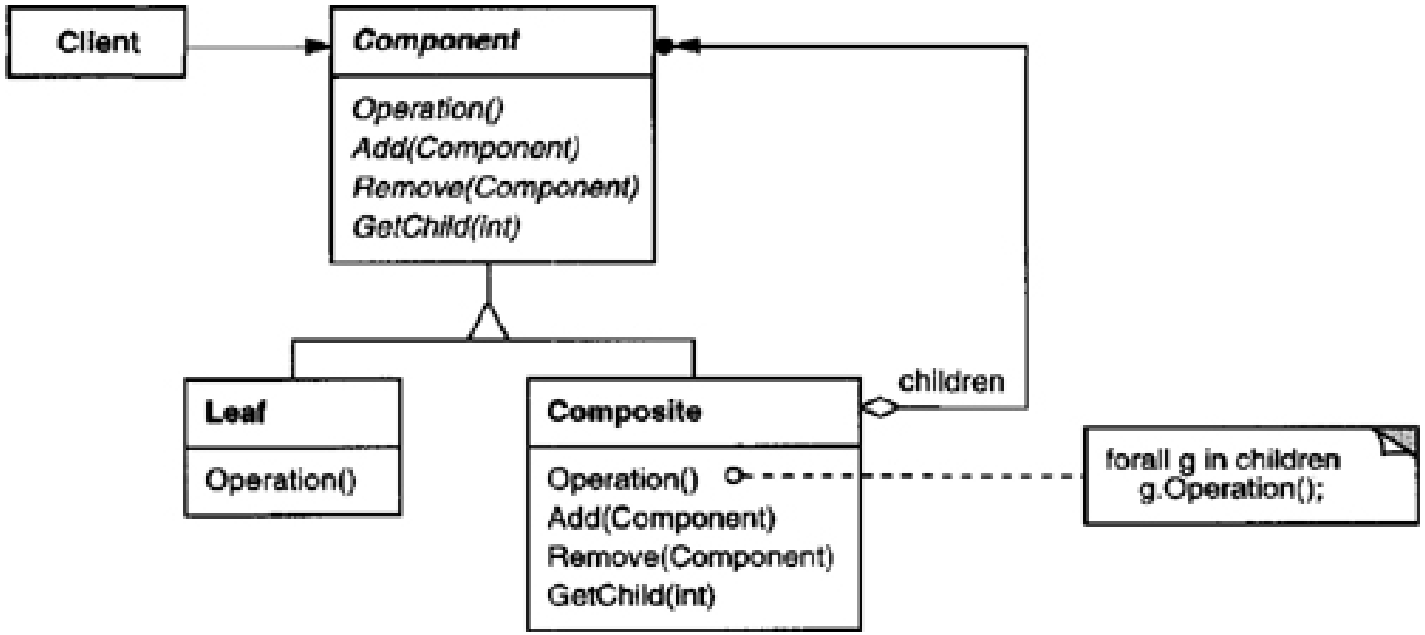
```
class Adding extends Opers {

  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)[14]



- 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

[14] 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 TreeComposite extends TreeComponent {
  List<TreeComponent> children = new ArrayList<TreeComponent>();

  public TreeComposite(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 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

Tasks

- 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
- Part of the lab task for Friday has been posted to moodle, please review before the lab (Maven, JUnit)