Principles of Software Construction: Objects, Design, and Concurrency

Part 1: Design for reuse

Introduction to design patterns

Michael Hilton **Bogdan Vasilescu**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- HW 2 graded soon
- HW 3 deadline Tue, Feb 12 instead of Thu, Feb 7
- Recitation 3 solutions on Piazza
- Midterm 1 on Thu, Feb 14
  - Review meeting: Wed, Feb 13, 6-8pm, Scaife Hall 125
- Reading due today:
  - UML and Patterns Ch 9 (Domain Models) and Ch 10 (System Sequence Diagrams)
- Inheritance: "is a"
  - Is a `SudokuXVerifier` usable anywhere a `SudokuVerifier` is used?
  - Are `SudokuXVerifier` and `SudokuVerifier` both `Verifiers`?

Key concepts from Tuesday

Delegation

- *Delegation* is simply when one object relies on another object for some subset of its functionality
  - e.g. here, the `Sorter` is delegating functionality to some `Order`
- Judicious delegation enables code reuse

- `Sorter` can be reused with arbitrary sort orders

- `Orders` can be reused with arbitrary client code that needs to compare integers

```
interface Order {
  boolean lessThan(int i, int j);
}

final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

public class Sorter {
  static void sort(int[] list, Order cmp) {
    ...
    boolean mustSwap =
      cmp.lessThan(list[i], list[j]);
    ...
  }
}
```

Using delegation to extend functionality

- One solution:

The LoggingList *is composed of* a List, and delegates (the non-logging) functionality to that List

```
public class LoggingList<E> implements List<E> {
  private final List<E> list;

  public LoggingList<E>(List<E> list) { this.list = list; }

  public boolean add(E e) {
      System.out.println("Adding " + e);
      return list.add(e);
  }

  public E remove(int index) {
      System.out.println("Removing at " + index);
      return list.remove(index);
  }

  …
```

Reuse abstract account code via inheritance

Review

an abstract class is missing the implementation of one or more methods

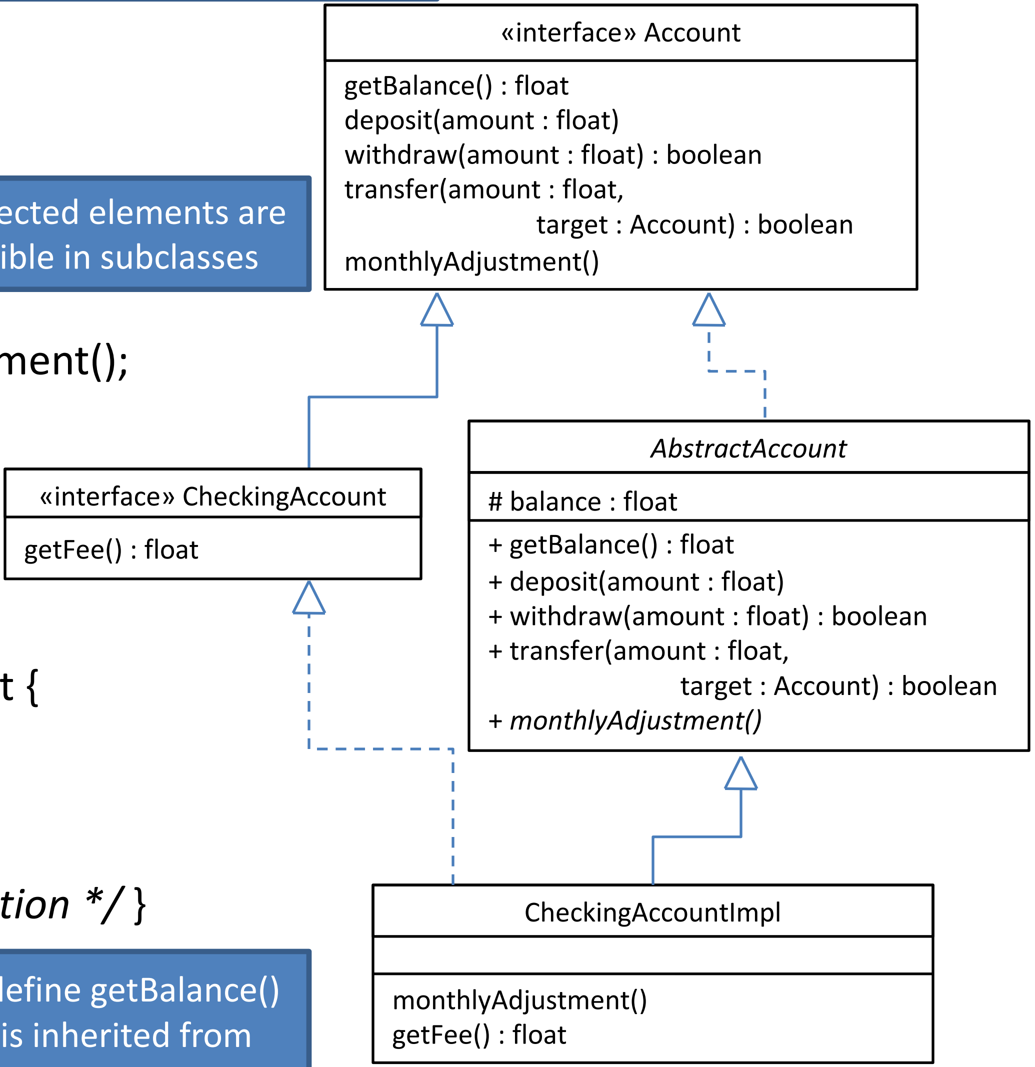
```
public abstract class AbstractAccount
            implements Account {
    protected float balance = 0.0;
    public float getBalance() {
            return balance;
    }
    abstract public void monthlyAdjustment();
    // other me
}
```

protected elements are visible in subclasses

an abstract method is left to be implemented in a subclass

```
public class CheckingAccountImpl
            extends AbstractAcount
            implements CheckingAccount {
    public void monthlyAdjustment() {
            balance -= getFee();
    }
    public float getFee() { /* fee calculation */ }
}
```

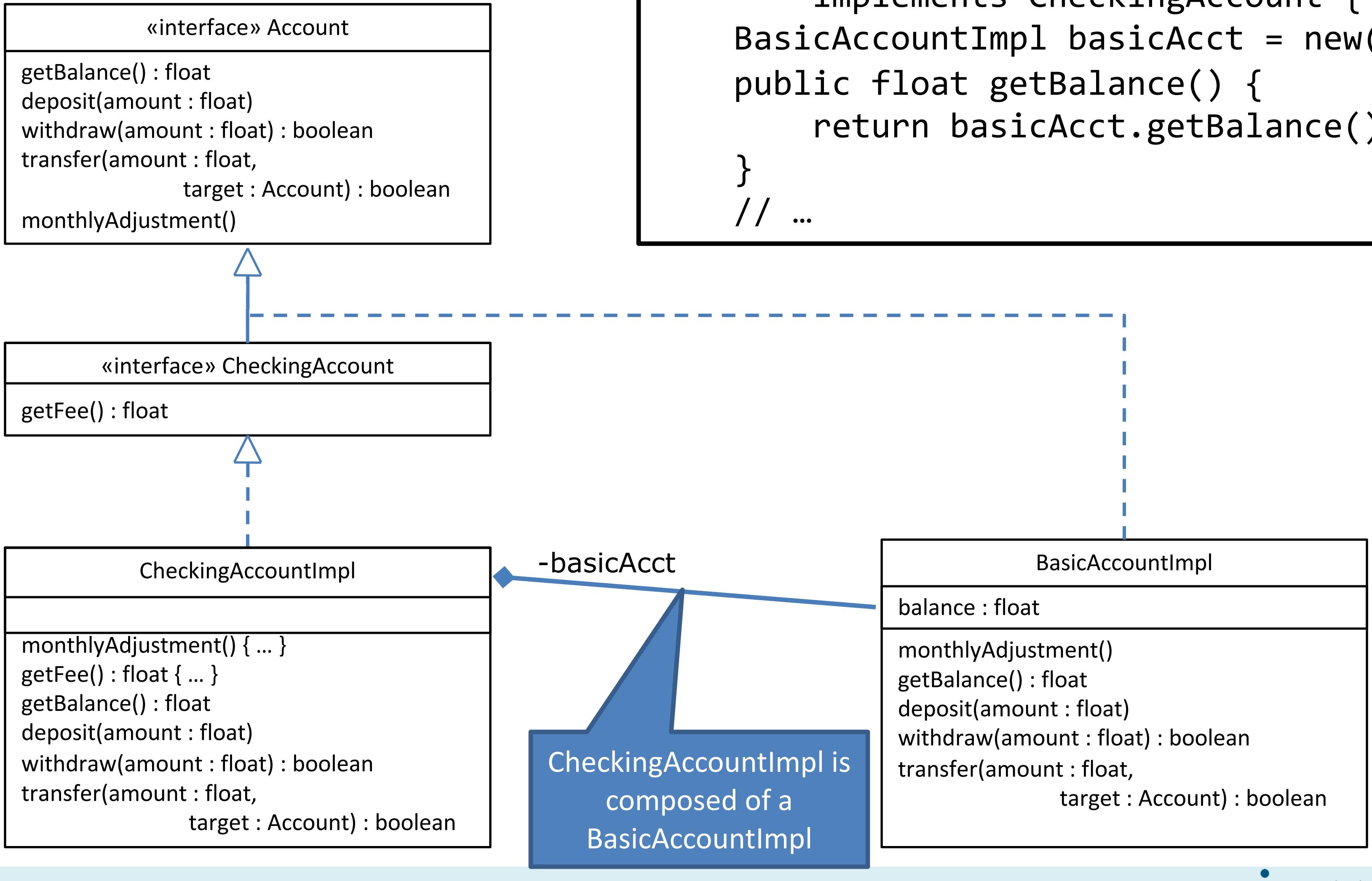
no need to define getBalance() – the code is inherited from AbstractAccount



«interface» Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
monthlyAdjustment()
«interface» CheckingAccount
getFee() : float
AbstractAccount
# balance : float
+ getBalance() : float
+ deposit(amount : float)
+ withdraw(amount : float) : boolean
+ transfer(amount : float, target : Account) : boolean
+ monthlyAdjustment()
CheckingAccountImpl
monthlyAdjustment()
getFee() : float

Alternatively: Reuse via composition and delegation

```
public class CheckingAccountImpl
        implements CheckingAccount {
    BasicAccountImpl basicAcct = new(…);
    public float getBalance() {
        return basicAcct.getBalance();
    }
    // …
```


«interface» Account
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
monthlyAdjustment()
«interface» CheckingAccount
getFee() : float
CheckingAccountImpl
monthlyAdjustment() { … }
getFee() : float { … }
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
-basicAcct
BasicAccountImpl
balance : float
monthlyAdjustment()
getBalance() : float
deposit(amount : float)
withdraw(amount : float) : boolean
transfer(amount : float, target : Account) : boolean
CheckingAccountImpl is composed of a BasicAccountImpl

(Almost always) Prefer composition over inheritance

- Tight coupling:
  - Changes to superclass → changes to subclass implementation
- Base class breaks encapsulation:
  - Exposes implementation details to subclasses (protected members)
- Inherited implementation can't change at runtime
- Inheritance stack may get very deep and confusing

- Inheritance: IS-A
  - Can use subclass where superclass is expected
  - E.g., Cessna biplane "is a" Airplane
- Composition: HAS-A
  - Only want some of the behavior of the superclass
  - E.g., Bird could (but shouldn't) inherit from Airplane, they both fly()

JAVA ASIDE: SUPER, THIS, FINAL, INSTANCEOF

Java details: extended reuse with super

```
public abstract class AbstractAccount implements Account {
   protected long balance = 0;
   public boolean withdraw(long amount) {
       // withdraws money from account (code not shown)
   }
}

public class ExpensiveCheckingAccountImpl
      extends AbstractAccount implements CheckingAccount {
   public boolean withdraw(long amount) {
      balance -= HUGE_ATM_FEE;
      boolean success = super.withdraw(amount)
      if (!success)
         balance += HUGE_ATM_FEE;
      return success;
   }
}
```

Overrides `withdraw` but also uses the superclass `withdraw` method

Java details: constructors with `this` and `super`

```
public class CheckingAccountImpl
      extends AbstractAccount implements CheckingAccount {

   private long fee;

   public CheckingAccountImpl(long initialBalance, long fee) {
      super(initialBalance);
      this.fee = fee;
   }

   public CheckingAccountImpl(long initialBalance) {
      this(initialBalance, 500);
   }
   /* other methods… */ }
```

Invokes a constructor of the superclass. Must be the first statement of the constructor.

Invokes another constructor in this same class

Java details: `final`

- A final ***field***: prevents reassignment to the field after initialization

- A final ***method***: prevents overriding the method

- A final ***class***: prevents extending the class
  - e.g., `public final class CheckingAccountImpl {…}`

Note: type-casting in Java

- Sometimes you want a different type than you have
  - e.g.,
    ```
    double pi = 3.14;
    int indianaPi = (int) pi;
    ```
- Useful if you know you have a more specific subtype:
  - e.g.,
    ```
    Account acct = …;
    CheckingAccount checkingAcct =
                        (CheckingAccount) acct;
    long fee = checkingAcct.getFee();
    ```
  - Will get a `ClassCastException` if types are incompatible
- Advice: avoid downcasting types
  - Never(?) downcast within superclass to a subclass

isr institute for SOFTWARE RESEARCH

An aside: `instanceof`

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
         adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
         adj = savingsAcct.getInterest();
    }
    …
}
```

Do not do this. This code is bad.

- Advice: avoid `instanceof` if possible
  - Never(?) use `instanceof` in a superclass to check type against subclass

An aside: `instanceof`

- Operator that tests whether an object is of a given class

```
public void doSomething(Account acct) {
    long adj = 0;
    if (acct instanceof CheckingAccount) {
        checkingAcct = (CheckingAccount) acct;
         adj = checkingAcct.getFee();
    } else if (acct instanceof SavingsAccount) {
        savingsAcct = (SavingsAccount) acct;
         adj = savingsAcct.getInterest();
    } else if (acct instanceof InterestCheckingAccount) {
        icAccount = (InterestCheckingAccount) acct;
        adj = icAccount.getInterest();
        adj -= icAccount.getFee();
    }
    …
}
```

Do not do this. This code is bad.

Use polymorphism to avoid `instanceof`

```
public interface Account {
    …
    public long getMonthlyAdjustment();
}

public class CheckingAccount implements Account {
    …
    public long getMonthlyAdjustment() {
        return getFee();
    }
}

public class SavingsAccount implements Account {
    …
    public long getMonthlyAdjustment() {
        return getInterest();
    }
}
```

Use polymorphism to avoid `instanceof`

```
~~public void doSomething(Account acct) {
   long adj = 0;
   if (acct instanceof CheckingAccount) {
      checkingAcct = (CheckingAccount) acct;
      adj = checkingAcct.getFee();
   } else if (acct instanceof SavingsAccount) {
      savingsAcct = (SavingsAccount) acct;
      adj = savingsAcct.getInterest();
   }
   …
}~~
```

Instead:

```
public void doSomething(Account acct) {
   long adj = acct.getMonthlyAdjustment();
   …
}
```

Today

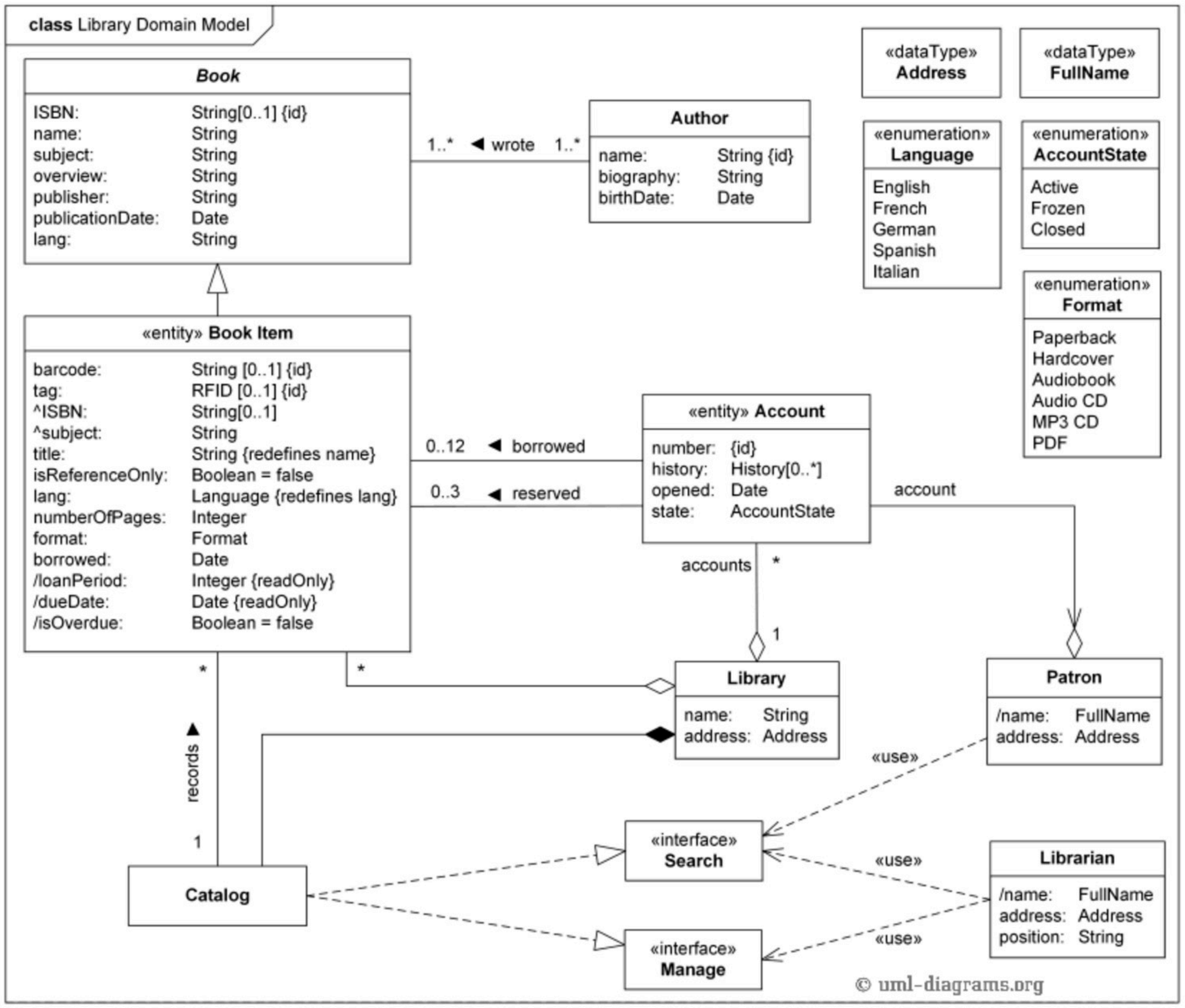
- UML diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern
  - Decorator pattern (next lecture)

Religious debates…

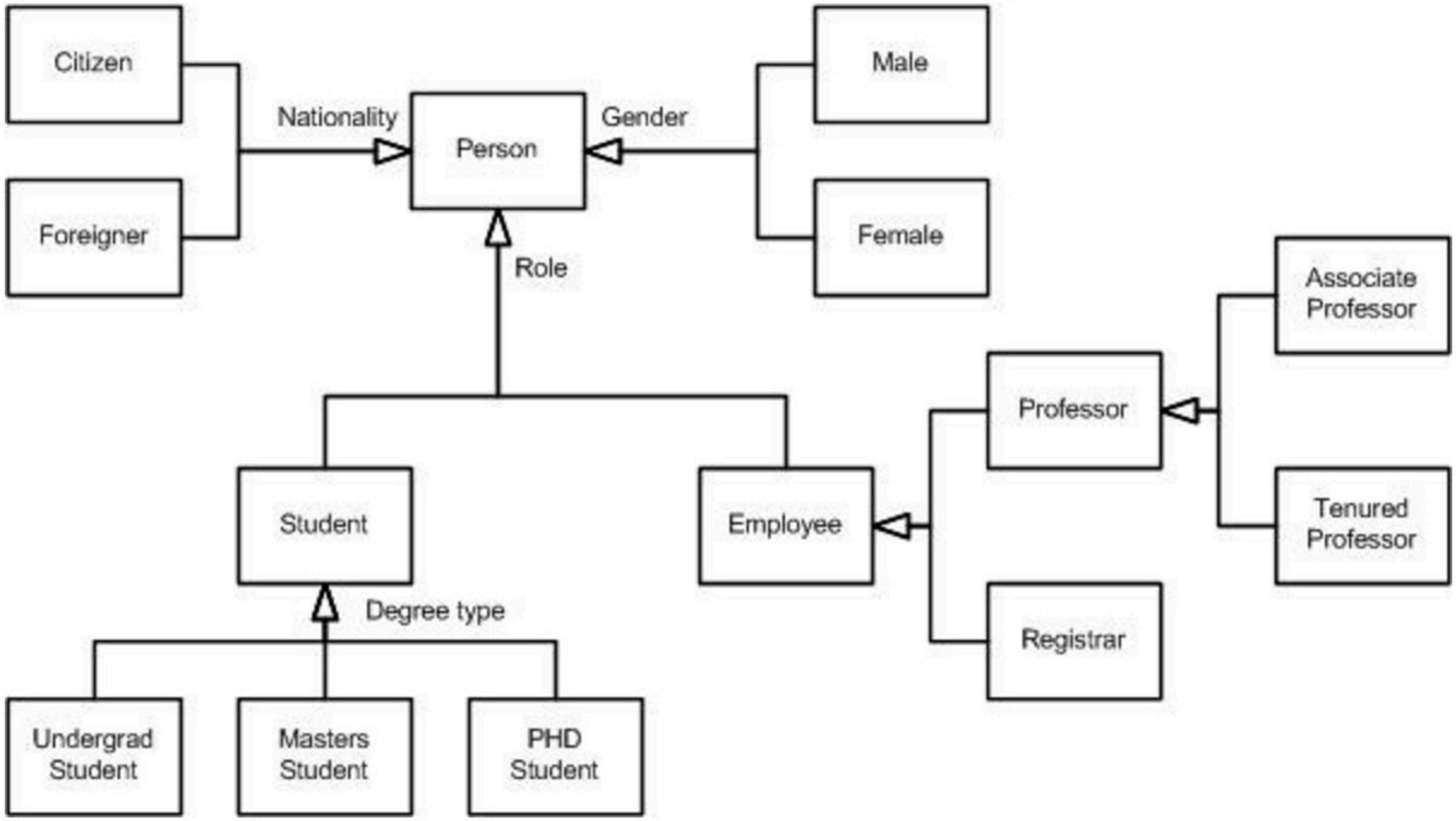
"Democracy is the worst form of government, except for all the others…"

-- (allegedly) Winston Churchill

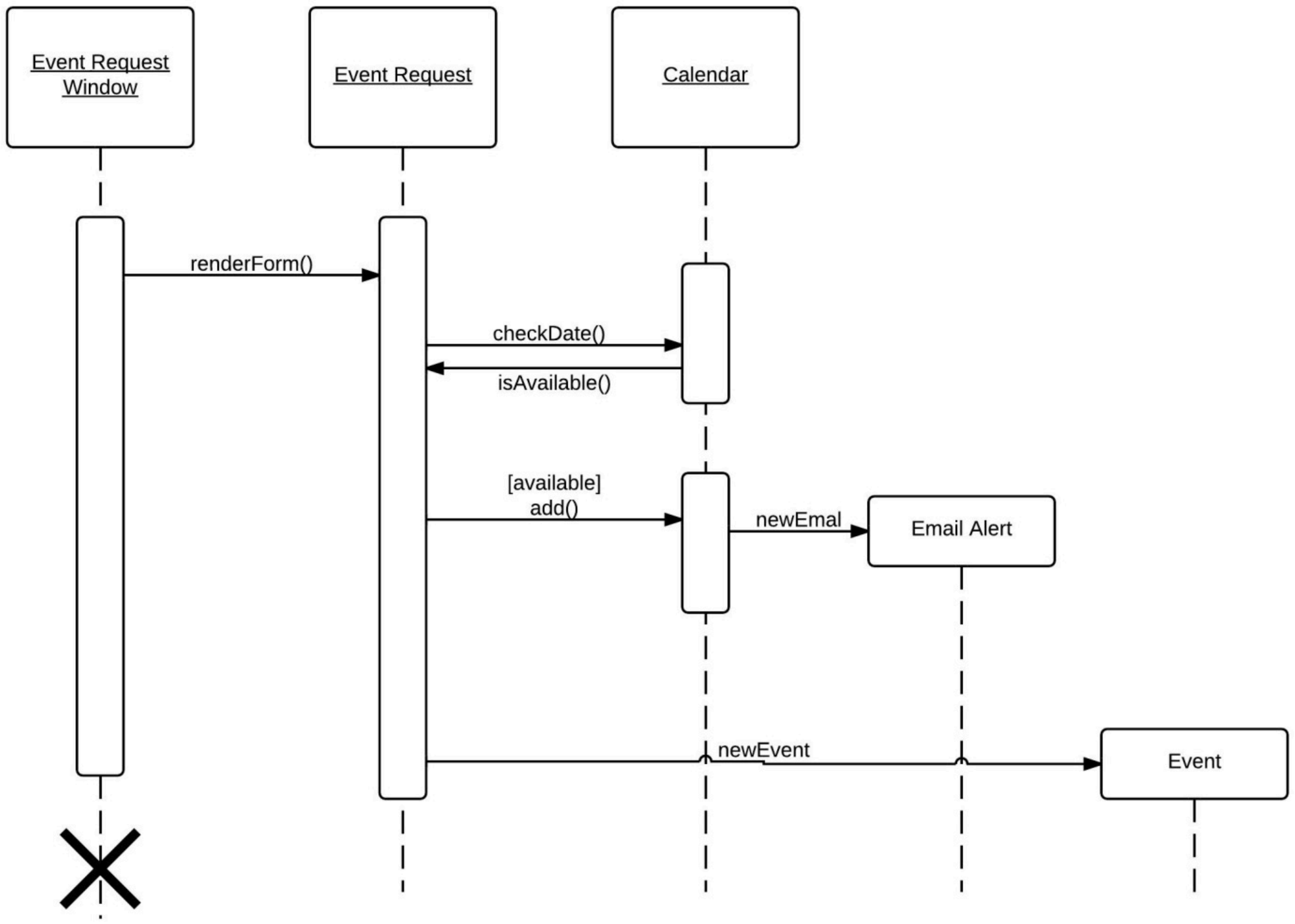
UML: Unified Modeling Language



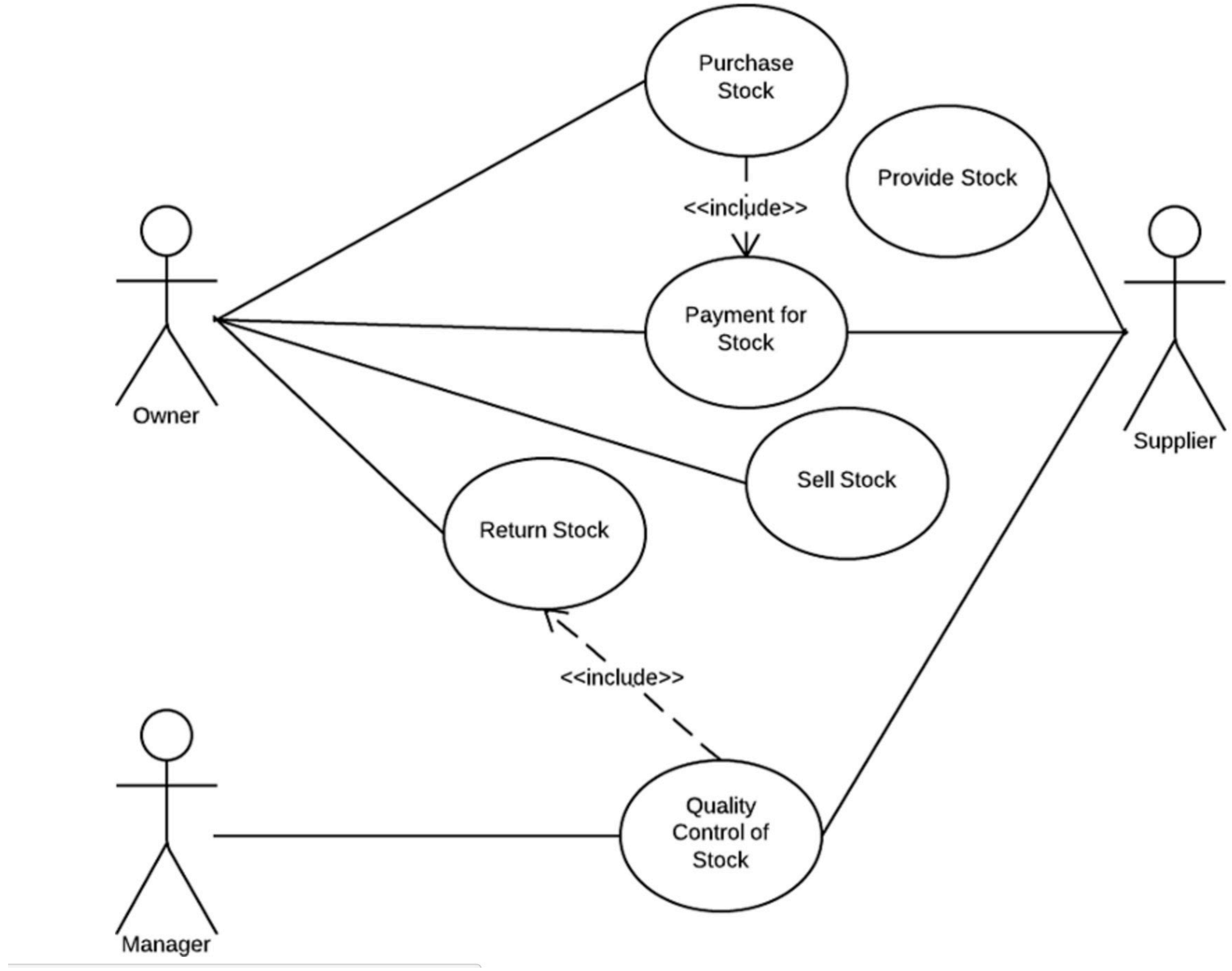
UML: Unified Modeling Language



UML: Unified Modeling Language



UML: Unified Modeling Language



UML in this course

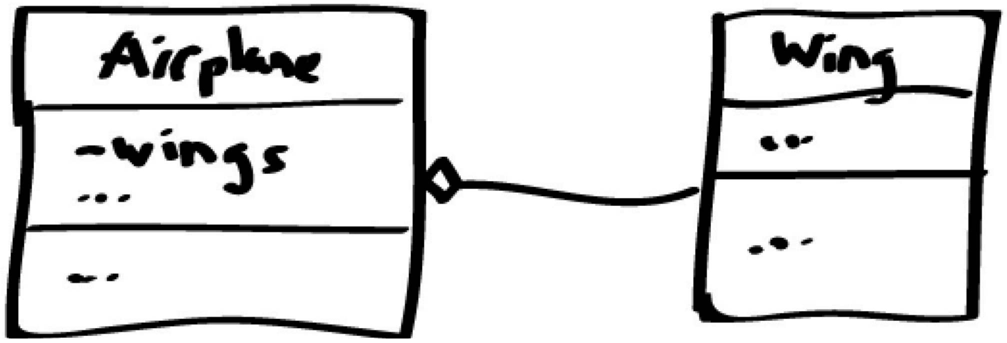
- Mostly:
  - UML class diagrams (domain models, object models)
  - UML interaction diagrams (sequence diagrams)

UML you should know

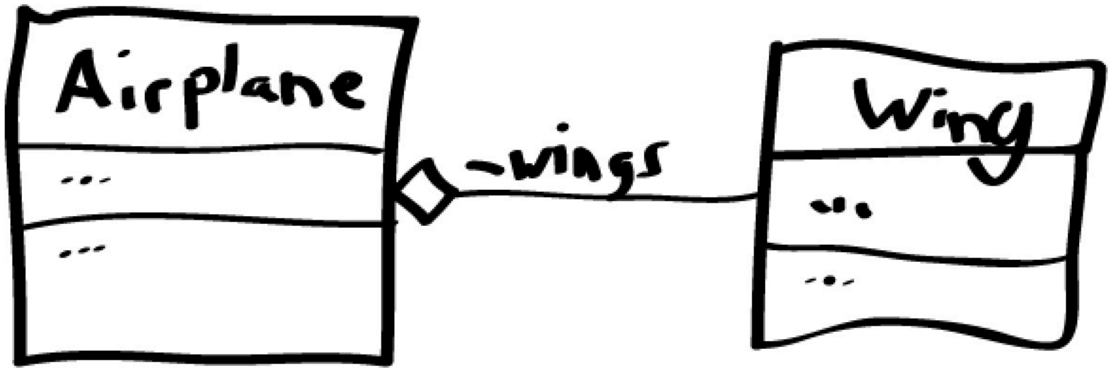
- Interfaces vs. classes
- Fields vs. methods
- Relationships:
  - "extends" (inheritance)
  - "implements" (realization)
  - "has a" (aggregation)
  - non-specific association
- Visibility: + (public) - (private) # (protected)
- Basic best practices…

UML advice

- Best used to show the big picture
  - Omit unimportant details
    - But show they are there: …
- Avoid redundancy
  - e.g., bad:



good:



Today

- UML diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern
  - Decorator pattern (next lecture)

One design scenario

- Amazon.com processes millions of orders each year, selling in 75 countries, all 50 states, and thousands of cities worldwide. These countries, states, and cities have hundreds of distinct sales tax policies and, for any order and destination, Amazon.com must be able to compute the correct sales tax for the order and destination.

Another design scenario

- A computer vision system must detect lines in an image. For different applications the line detection requirements vary. E.g., for a vision system in a driverless car the system must process 30 images per second, but it's OK to miss some lines in some images. A face recognition system can spend 3-5 seconds analyzing an image, but requires accurate detection of subtle lines on a face.

A third design scenario

- Suppose we need to sort a list in different orders…

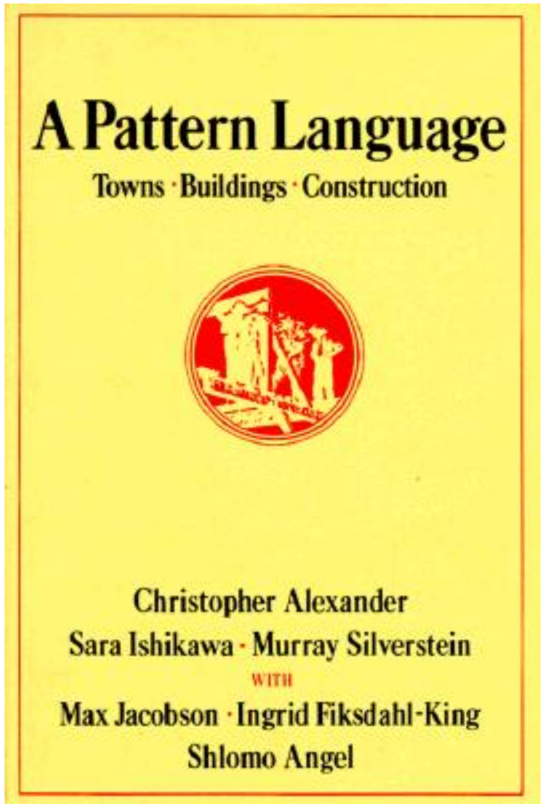
```
interface Order {
  boolean lessThan(int i, int j);
}

final Order ASCENDING =  (i, j) -> i < j;
final Order DESCENDING = (i, j) -> i > j;

static void sort(int[] list, Order cmp) {
  …
  boolean mustSwap =
    cmp.lessThan(list[i], list[j]);
  …
}
```

institute for SOFTWARE RESEARCH

Design patterns

"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"

– Christopher Alexander, Architect (1977)


A Pattern Language
Towns · Buildings · Construction
Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel

How not to discuss design (from Shalloway and Trott)

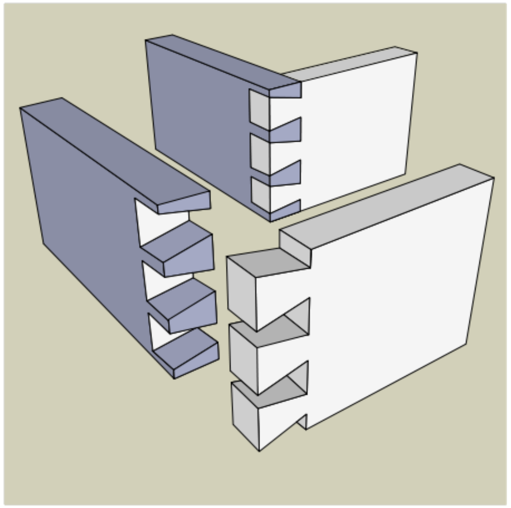
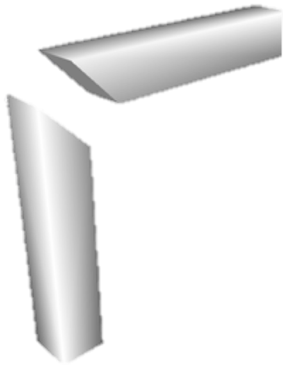
- Carpentry:
  - How do you think we should build these drawers?
  - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…

How not to discuss design (from Shalloway and Trott)

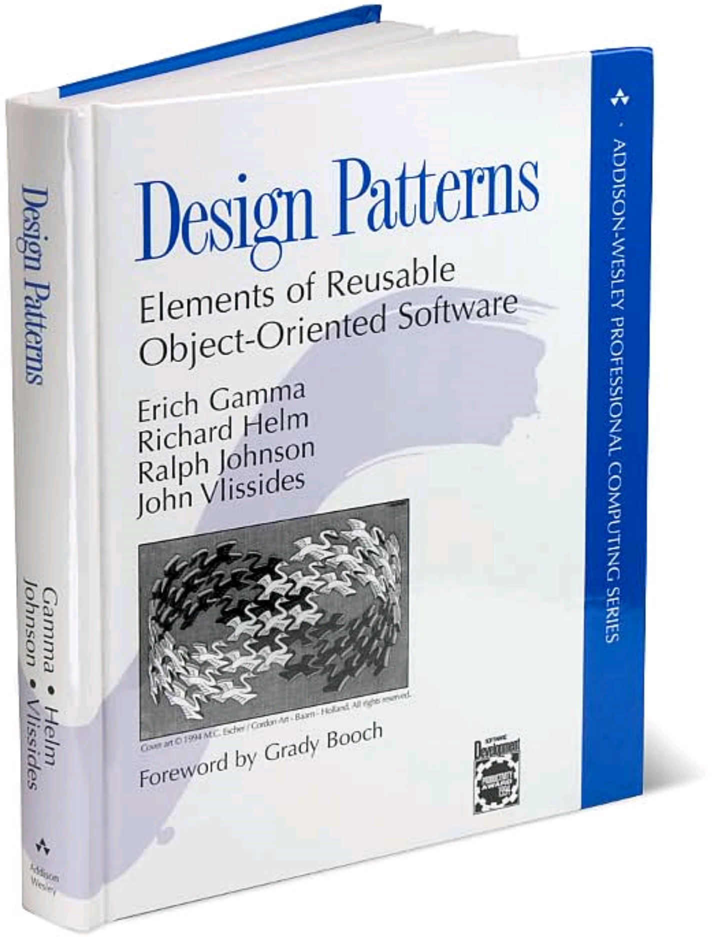
- Carpentry:
  - How do you think we should build these drawers?
  - Well, I think we should make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight down, and repeating…
- Software Engineering:
  - How do you think we should write this method?
  - I think we should write this if statement to handle … followed by a while loop … with a break statement so that…

Discussion with design patterns

- Carpentry:
  - "Is a dovetail joint or a miter joint better here?"
- Software Engineering:
  - "Is a strategy pattern or a template method better here?"

History: *Design Patterns* (1994)



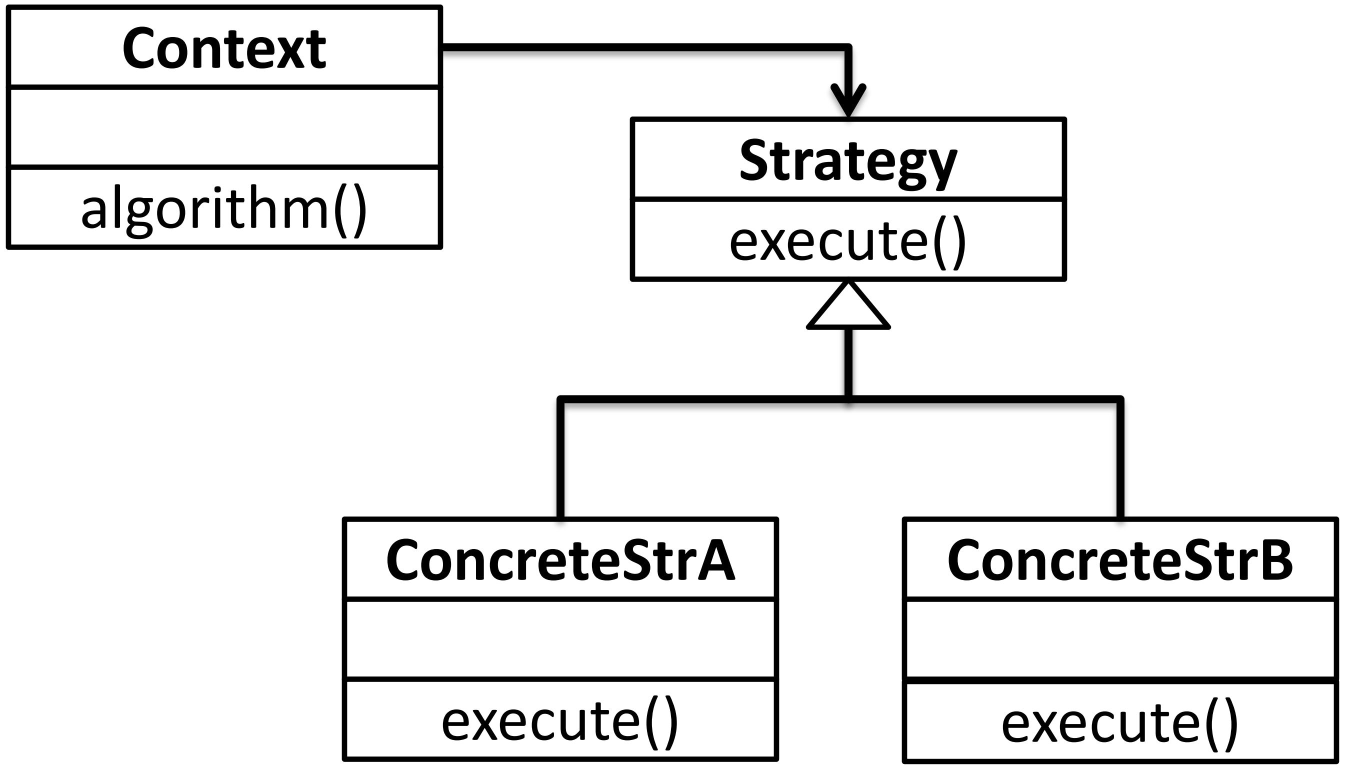
Elements of a design pattern

- Name
- Abstract description of problem
- Abstract description of solution
- Analysis of consequences

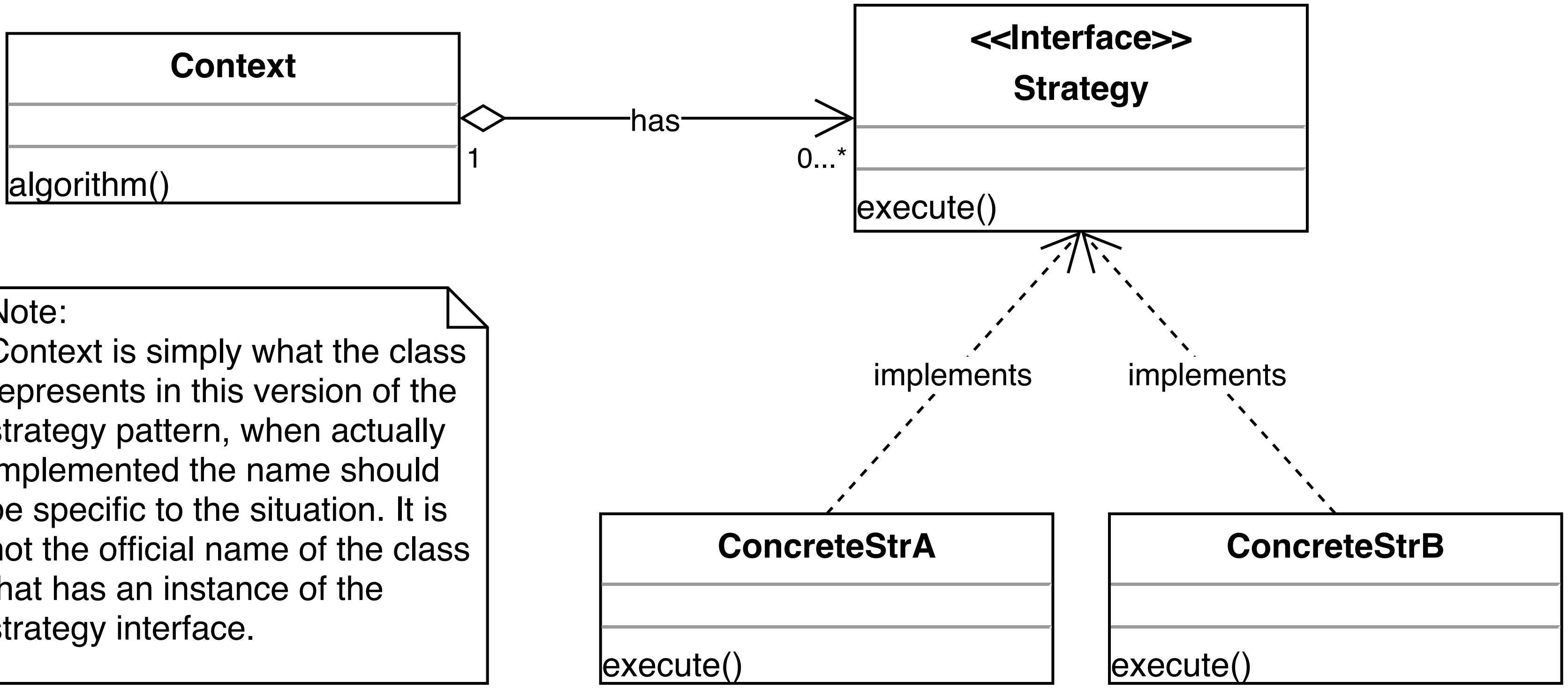
Strategy pattern

- Problem: Clients need different variants of an algorithm
- Solution: Create an interface for the algorithm, with an implementing class for each variant of the algorithm
- Consequences:
  - Easily extensible for new algorithm implementations
  - Separates algorithm from client context
  - Introduces an extra interface and many classes:
    - Code can be harder to understand
    - Lots of overhead if the strategies are simple

UML for strategy pattern



Update 2/26: Better UML for strategy pattern



Note:
Context is simply what the class represents in this version of the strategy pattern, when actually implemented the name should be specific to the situation. It is not the official name of the class that has an instance of the strategy interface.

Strategy pattern example

```
public interface Strategy {
  public int doOperation(int num1, int num2);
}
```

Strategy pattern example

```
public interface Strategy {
  public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy {
  @Override
  public int doOperation(int num1, int num2) {
    return num1 + num2;
  }
}

public class OperationSubtract implements Strategy { ... }
```

Strategy pattern example

```java
public interface Strategy {
  public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy {
  @Override
  public int doOperation(int num1, int num2) {
    return num1 + num2;
  }
}

public class OperationSubtract implements Strategy { ... }

public class Context {
  private Strategy strategy;
  public Context(Strategy strategy){ this.strategy = strategy; }
  public int algorithm(int num1, int num2){
    return strategy.doOperation(num1, num2);
  }
}
```

Strategy pattern example

```
public interface Strategy {
  public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy {
  @Override
  public int doOperation(int num1, int num2) {
    return num1 + num2;
  }
}

public class OperationSubtract implements Strategy { ... }

public class Context {
  private Strategy strategy;
  public Context(Strategy strategy){ this.strategy = strategy; }
  public int algorithm(int num1, int num2){
    return strategy.doOperation(num1, num2);
  }
}
public static void main(String[] args) {
  Context context = new Context(new OperationAdd());
  System.out.println("10 + 5 = " + context.algorithm(10, 5));
}
```

Patterns are more than just structure

- Consider: A modern car engine is constantly monitored by a software system. The monitoring system must obtain data from many distinct engine sensors, such as an oil temperature sensor, an oxygen sensor, etc. More sensors may be added in the future.

Different patterns can have the same structure

Command pattern:

- Problem: Clients need to execute some (possibly flexible) operation without knowing the details of the operation
- Solution: Create an interface for the operation, with a class (or classes) that actually executes the operation
- Consequences:
  - Separates operation from client context
  - Can specify, queue, and execute commands at different times
  - Introduces an extra interface and classes:
    - Code can be harder to understand
    - Lots of overhead if the commands are simple

Example: Strategy pattern vs Command pattern

- How something should be done (Strategy)

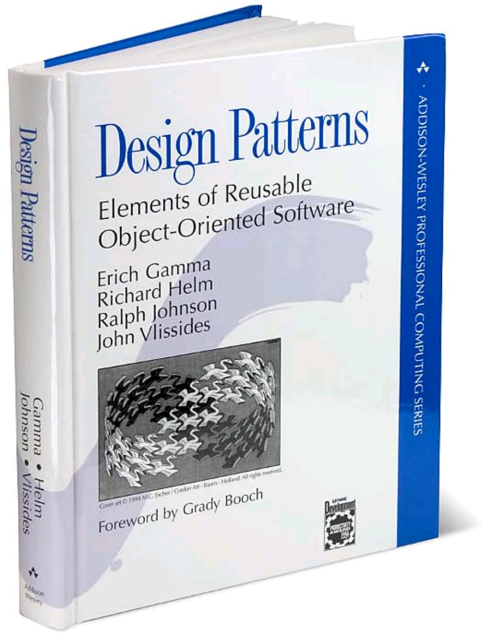
```
public class ConcreteStrategy implements BaseStrategy {
  @Override public void execute(Object argument) {
    // Work with passed-in argument.
  }
}
```

- Versus what needs to be done (Command)

```
public class ConcreteCommand implements BaseCommand {
  private Object argument;
  public ConcreteCommand(Object argument) {
    this.argument = argument;
  }

  @Override public void execute() {
    // Work with own state.
  }
}
```

Design pattern conclusions

- Provide shared language
- Convey shared experience
- Can be system and language specific



Today

- UML diagrams
- Introduction to design patterns
  - Strategy pattern
  - Command pattern
- Design patterns for reuse:
  - Template method pattern
  - Iterator pattern
  - Decorator pattern (next lecture)

One design scenario

- A GUI-based document editor works with multiple document formats. Some parts of the algorithm to load a document (e.g., reading a file, rendering to the screen) are the same for all document formats, and other parts of the algorithm vary from format-to-format (e.g. parsing the file input).

Another design scenario

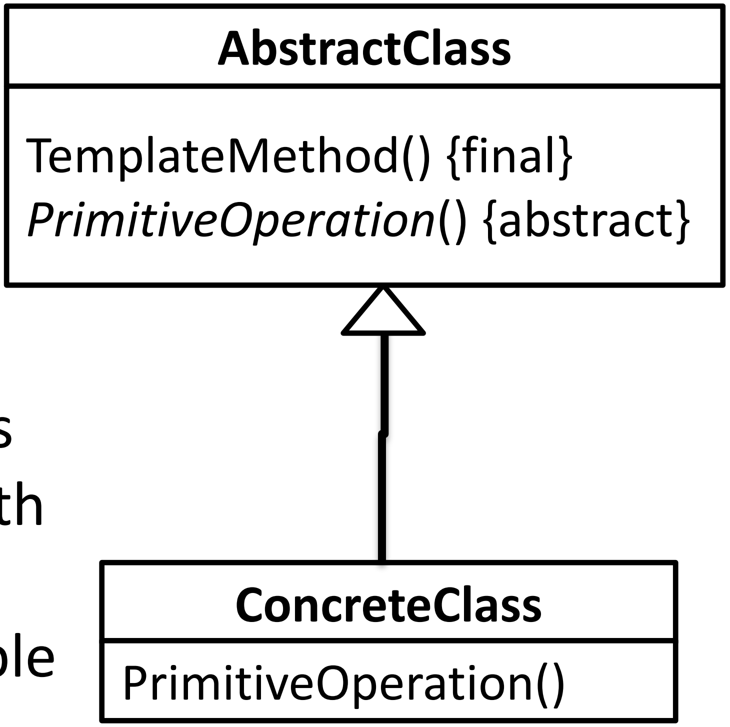
- Several versions of a domain-specific machine learning algorithm are being implemented to use data stored in several different database systems. The basic algorithm for all versions is the same; just the interactions with the database are different from version to version.

The abstract `java.util.AbstractList<E>`

```
abstract T   get(int i);
abstract int size();
boolean      set(int i, E e);          // pseudo-abstract*
boolean      add(E e);                 // pseudo-abstract*
boolean      remove(E e);              // pseudo-abstract*
boolean      addAll(Collection<? extends E> c);
boolean      removeAll(Collection<?> c);
boolean      retainAll(Collection<?> c);
boolean      contains(E e);
boolean      containsAll(Collection<?> c);
void         clear();
boolean      isEmpty();
abstract Iterator<E>  iterator();
Object[]     toArray()
<T> T[]      toArray(T[] a);
…
```

*throws an `UnsupportedOperationException`

Template method pattern



- Problem: An algorithm consists of customizable parts and invariant parts
- Solution: Implement the invariant parts of the algorithm in an abstract class, with abstract (unimplemented) primitive operations representing the customizable parts of the algorithm. Subclasses customize the primitive operations
- Consequences
  - Code reuse for the invariant parts of algorithm
  - Customization is restricted to the primitive operations
  - Inverted (Hollywood-style) control for customization

Template method example

```
public abstract class Game {
  abstract void startPlay();
  abstract void endPlay();
  //template method
  public final void play(){
    startPlay();
    endPlay();
  }
}
```

Template method example

```
public abstract class Game {
  abstract void startPlay();
  abstract void endPlay();
  //template method
  public final void play(){
    startPlay();
    endPlay();
  }
}
```

```
public class Football extends Game {
  @Override
  void startPlay() {
    System.out.println("Football Started!");
  }
  @Override
  void endPlay() {
    System.out.println("Football Finished!");
  }
}

public class Cricket extends Game { ... }
```

Template method example

```
public abstract class Game {
  abstract void startPlay();
  abstract void endPlay();
  //template method
  public final void play(){
    startPlay();
    endPlay();
  }
}
```

```
public class Football extends Game {
  @Override
  void startPlay() {
    System.out.println("Football Started!");
  }
  @Override
  void endPlay() {
    System.out.println("Football Finished!");
  }
}

public class Cricket extends Game { ... }
```

```
public class Demo {
  public static void main(String[] args) {
    Game game = new Cricket();
    game.play();
    game = new Football();
    game.play();
  }
}
```

Template method vs. the strategy pattern

- Template method uses inheritance to vary part of an algorithm
  - Template method implemented in supertype, primitive operations implemented in subtypes
- Strategy pattern uses delegation to vary the entire algorithm
  - Strategy objects are reusable across multiple classes
  - Multiple strategy objects are possible per class

Summary

- Use UML class diagrams to simplify communication
- Design patterns…
  - Convey shared experience, general solutions
  - Facilitate communication
- Specific design patterns for reuse:
  - Strategy
  - Template method
  - Iterator (next lecture)