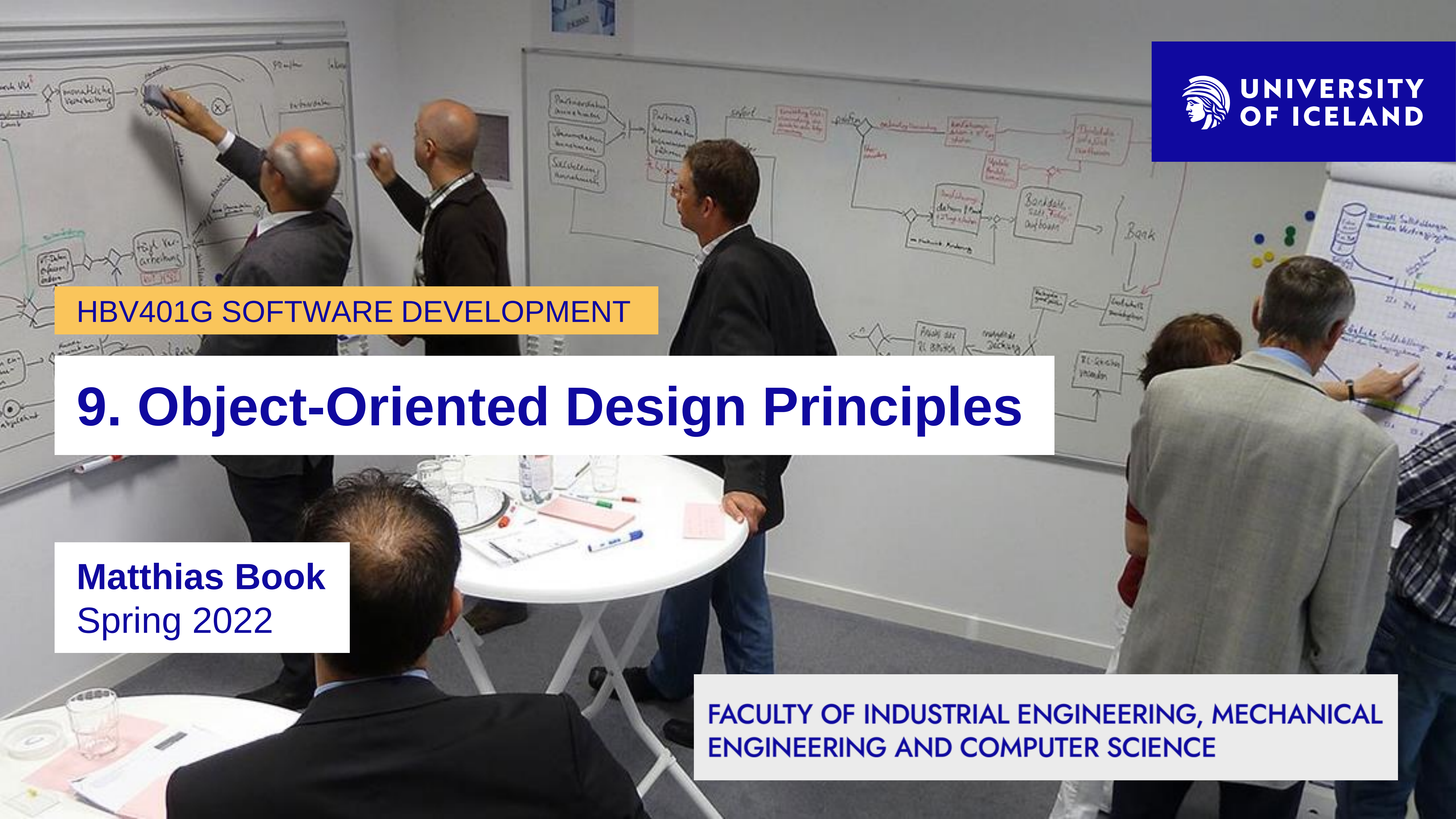


UNIVERSITY OF ICELAND

HBV401G SOFTWARE DEVELOPMENT

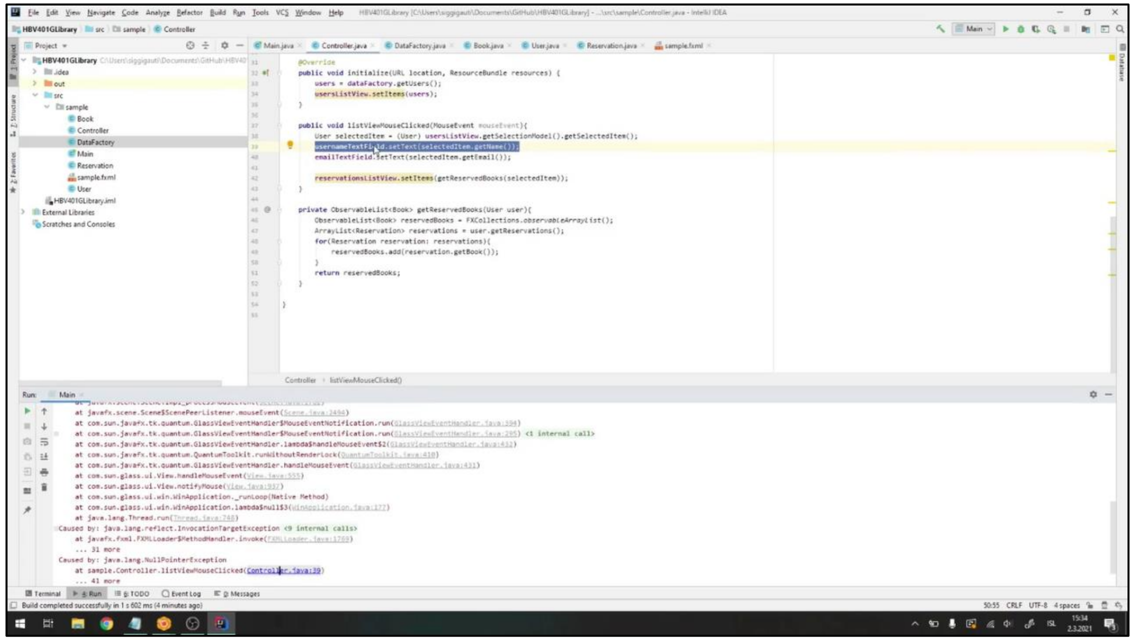
9. Object-Oriented Design Principles

Matthias Book
Spring 2022

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

Coding Kickstart Material

- Prototype of a library information system
  - Illustrates how to build a simple desktop application with
    - graphical user interface
    - in-memory object-oriented data storage
- Code-along video available on *Panopto Video* page in Canvas
- Source code available at https://github.com/siggigauti/HBV401GLibrary
- Instructions and links to tools in video description
  - Please post implementation-related questions in the *#coding-help* channel on Discord
  - Peer support encouraged – if you can answer a question, please do ☺



```java
    @Override
    public void initialize(URL location, ResourceBundle resources) {
        users = dataFactory.getUsers();
        usersListView.setItems(users);
    }

    public void listViewMouseClicked(MouseEvent mouseEvent){
        User selectedItem = (User) usersListView.getSelectionModel().getSelectedItem();
        usernameTextField.setText(selectedItem.getName());
        emailTextField.setText(selectedItem.getEmail());

        reservationsListView.setItems(getReservedBooks(selectedItem));
    }

    private ObservableList<Book> getReservedBooks(User user){
        ObservableList<Book> reservedBooks = FXCollections.observableArrayList();
        ArrayList<Reservation> reservations = user.getReservations();
        for(Reservation reservation: reservations){
            reservedBooks.add(reservation.getBook());
        }
        return reservedBooks;
    }

}
```

```
    at javafx.scene.Scene$ScenePeerListener.mouseEvent(Scene.java:3494)
    at com.sun.javafx.tk.quantum.GlassViewEventHandler$MouseEventNotification.run(GlassViewEventHandler.java:394)
    at com.sun.javafx.tk.quantum.GlassViewEventHandler$MouseEventNotification.run(GlassViewEventHandler.java:295) <1 internal call>
    at com.sun.javafx.tk.quantum.GlassViewEventHandler.lambda$handleMouseEvent$2(GlassViewEventHandler.java:432)
    at com.sun.javafx.tk.quantum.QuantumToolkit.runWithoutRenderLock(QuantumToolkit.java:410)
    at com.sun.javafx.tk.quantum.GlassViewEventHandler.handleMouseEvent(GlassViewEventHandler.java:431)
    at com.sun.glass.ui.View.handleMouseEvent(View.java:555)
    at com.sun.glass.ui.View.notifyMouse(View.java:937)
    at com.sun.glass.ui.win.WinApplication._runLoop(Native Method)
    at com.sun.glass.ui.win.WinApplication.lambda$null$3(WinApplication.java:177)
    at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.reflect.InvocationTargetException <9 internal calls>
    at javafx.fxml.FXMLLoader$MethodHandler.invoke(FXMLLoader.java:1769)
    ... 31 more
Caused by: java.lang.NullPointerException
    at sample.Controller.listViewMouseClicked(Controller.java:39)
    ... 41 more
```

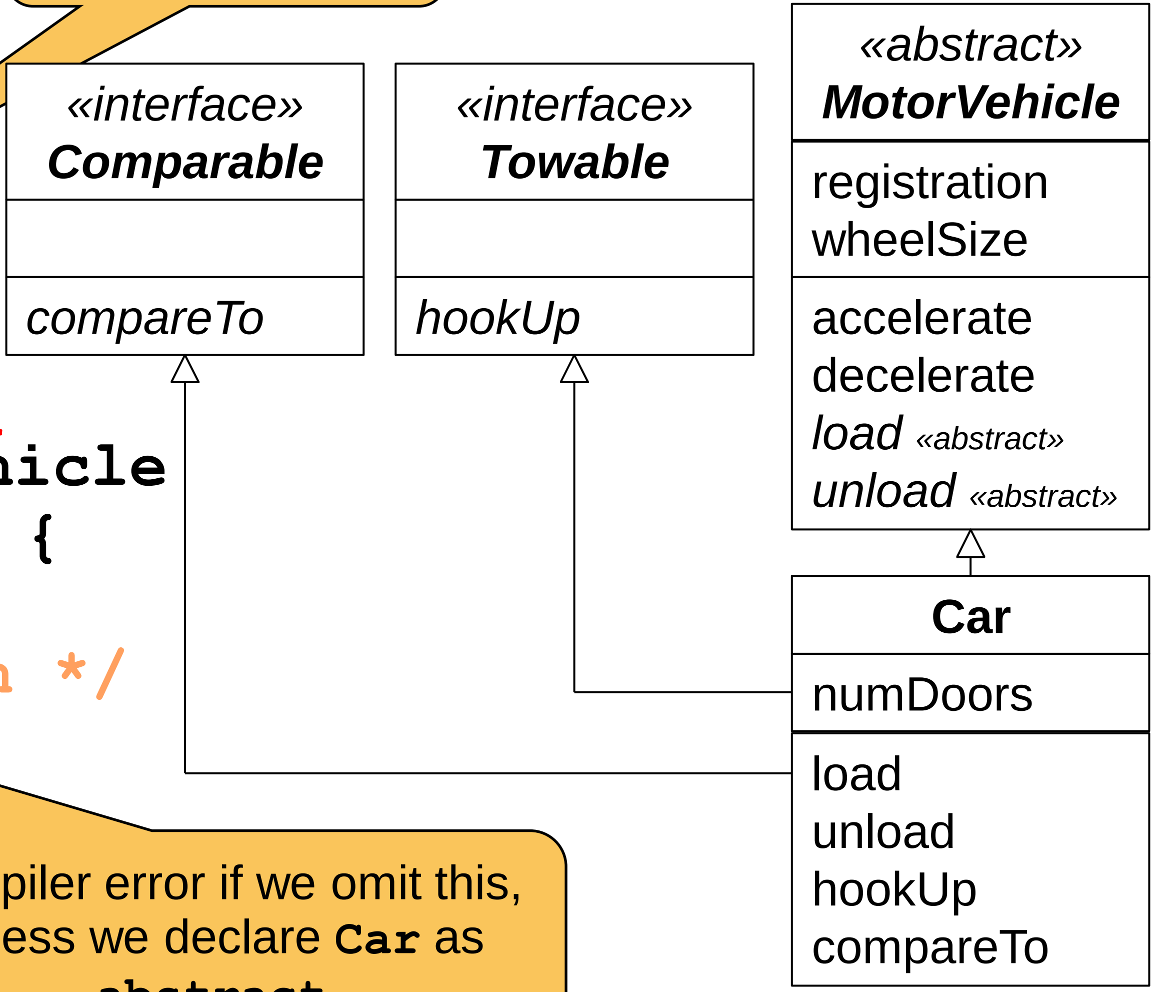
Build completed successfully in 1 s 602 ms (4 minutes ago)

Recap: Interfaces in Java

```
public interface Towable {
  ...
  public void hookUp(...);
}

public class Car extends MotorVehicle
  implements Towable, Comparable {
  public void hookUp(...) {
    /* specialized implementation */
  }
  ...
}
```

Note:
No method body!

Compiler error if we omit this, unless we declare `Car` as `abstract`



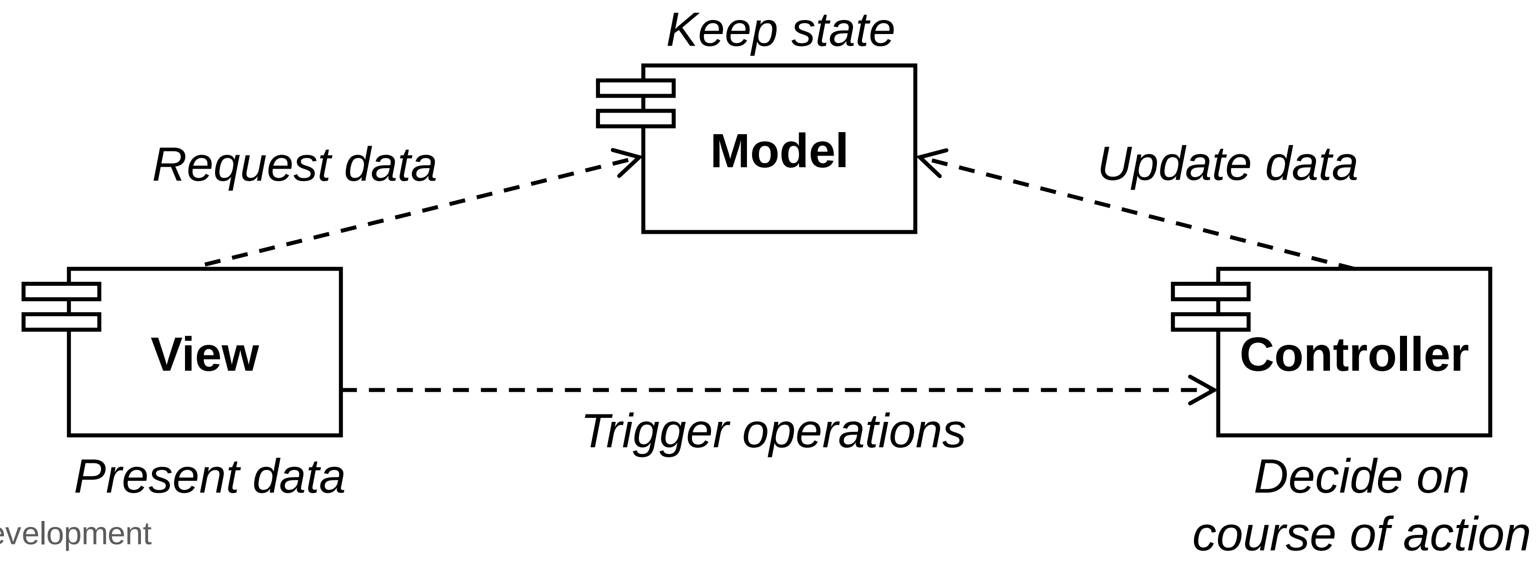
In-Class Quiz #8 Solution

- **Indicate if the following properties apply to *classes* or *interfaces*:**

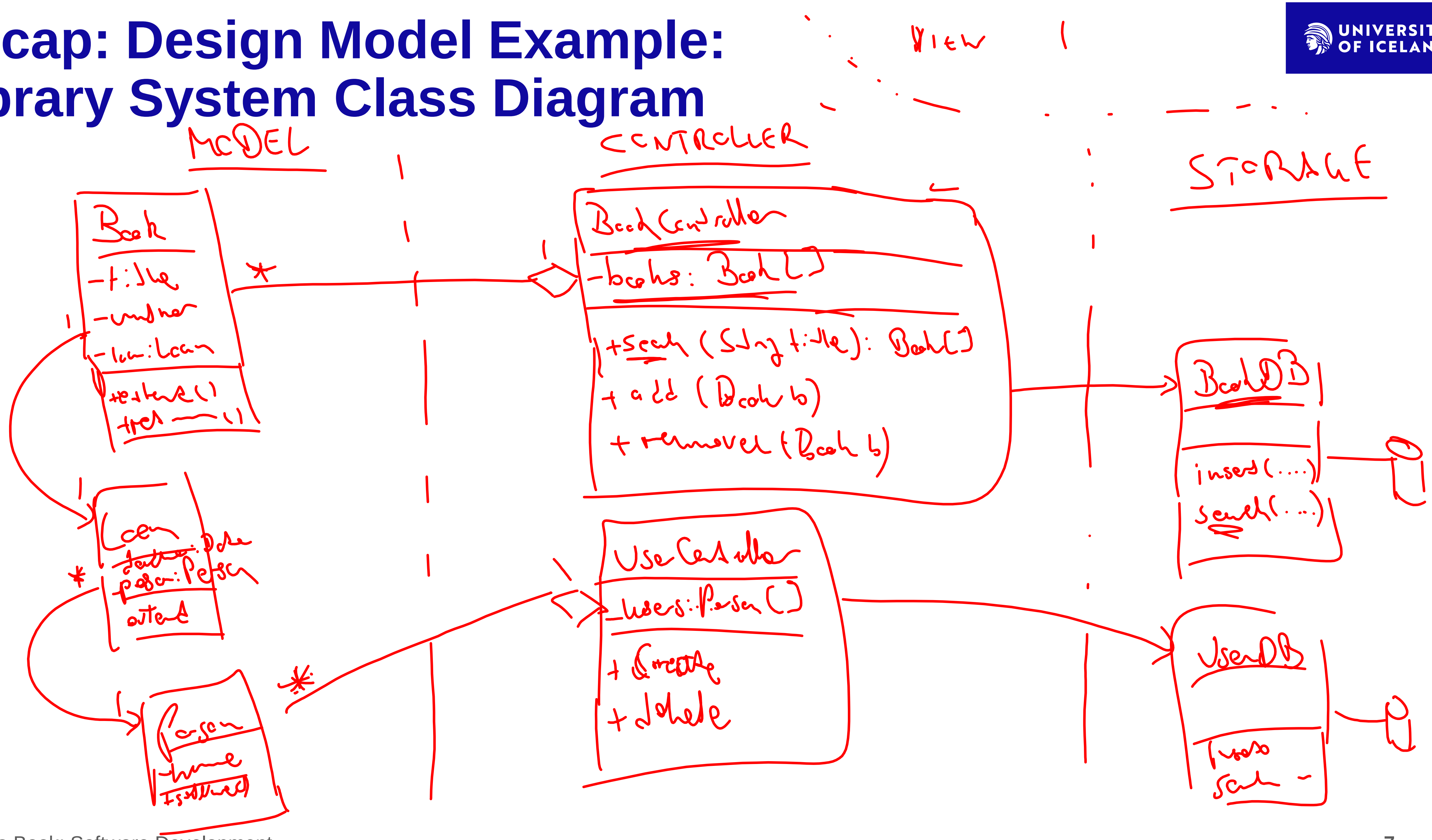
a) A Java class can extend only one CLASS.
b) A Java class can implement multiple INTERFACES.
c) INTERFACES contain only abstract methods.
d) A hierarchy of CLASSES describes commonalities and differences of entities within the same “family”.
e) INTERFACES describe additional capabilities independent of entities’ “heritage”.
f) INTERFACES cannot inherit declarations from CLASSES.

Recap: Model-View-Controller (MVC) Design Pattern

- Divide the system's classes into three large components / layers:
  - **Model:** Application state (data and operations working on them); usually subdivided into
    - working data (objects in memory)
    - persistence layer (database access logic)
    - Classes can usually be derived from domain model to some degree (some adaptation required)
  - **View:** View(s) of a state; usually the user interface, but possibly also other access services
    - Classes representing screens, dialogs, widgets etc. – sometimes left out for simplicity
  - **Controller:** Input interface; invokes operations on Model, based on events from View
    - Classes with methods implementing/controlling the main business logic of the system

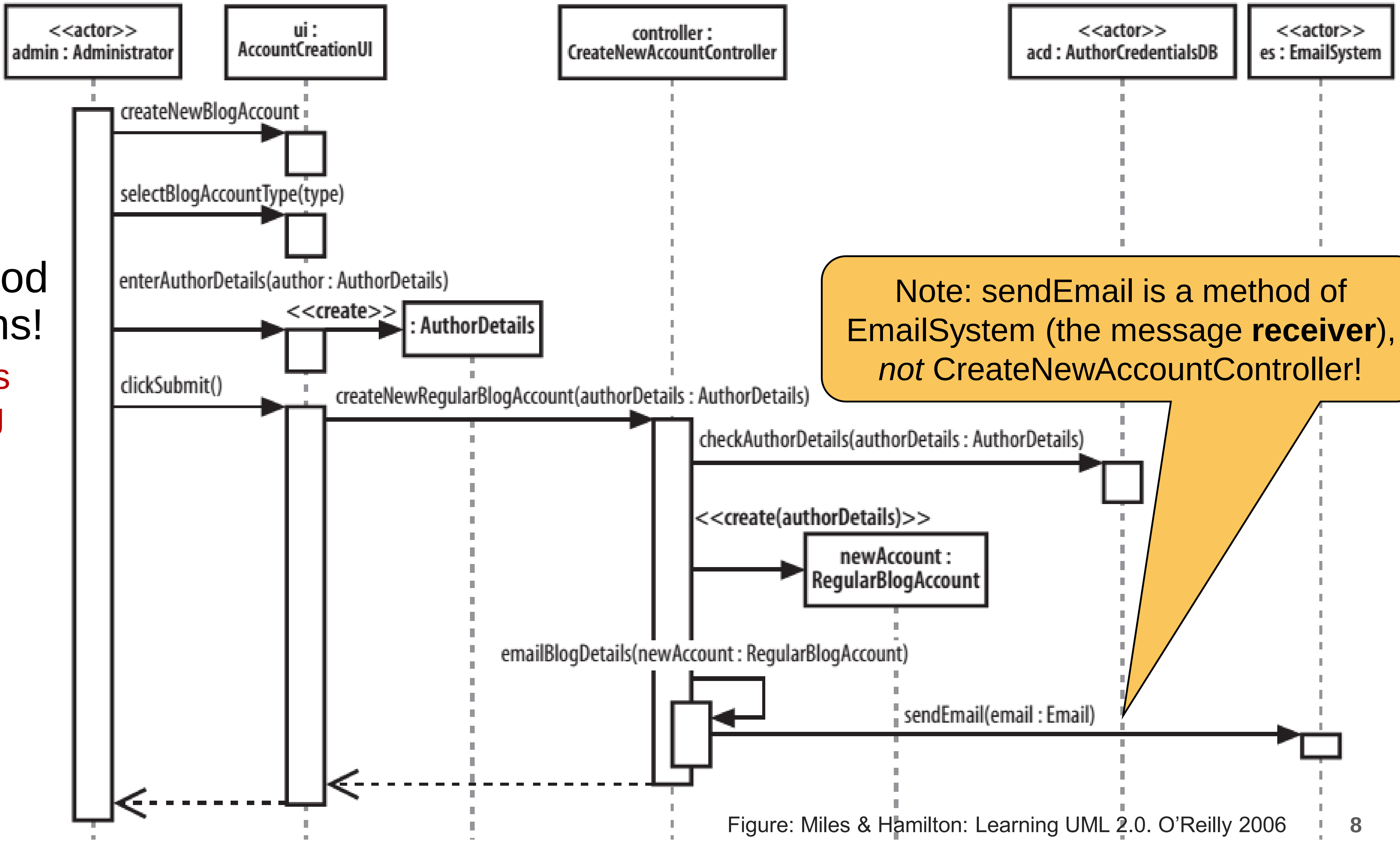


Recap: Design Model Example: Library System Class Diagram



Recap: UML Sequence Diagrams

- Note: Arrows labelled with method invocations!
  - Use this labeling style in your assign-ment!

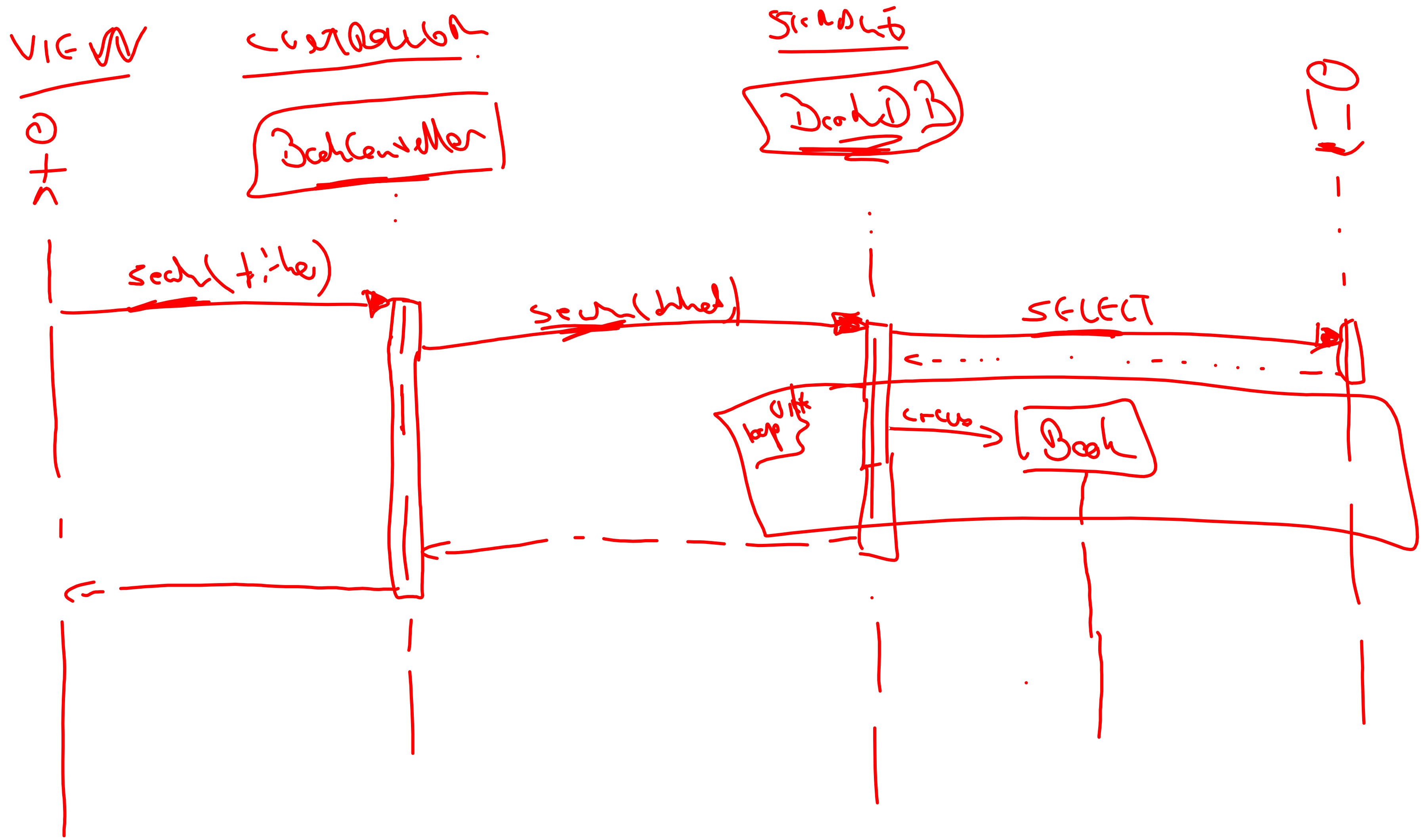


<<actor>> admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>> acd : AuthorCredentialsDB
<<actor>> es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails(author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails : AuthorDetails)
checkAuthorDetails(authorDetails : AuthorDetails)
<<create(authorDetails)>>
newAccount : RegularBlogAccount
emailBlogDetails(newAccount : RegularBlogAccount)
sendEmail(email : Email)

Note: sendEmail is a method of EmailSystem (the message **receiver**), *not* CreateNewAccountController!

Figure: Miles & Hamilton: Learning UML 2.0. O'Reilly 2006

Recap: Design Model Example: Library System Sequence Diagram



Recap: Team Assignment 3: Design Model

- By **Sun 13 Mar**, submit in Ugla:
  - A **UML class diagram** showing:
    - All classes your component will be composed of
      - Model layer: Classes representing your domain entities (incorporating feedback from last assignment)
      - Controller layer: Classes driving your business processes (incorporating feedback from last assignment)
      - ~~View layer: User interface logic~~ (not required for this assignment)
      - Storage layer: Database / data access logic
  - A **UML sequence diagram** showing:
    - How your component responds to an incoming user request, using its data sources
    - How you are interfacing with other teams' components
- On **Wed 16 Mar**, present and **explain** your model to your tutor:
  - Why did you structure the classes and their relationships the way you did?
  - What considerations influenced your way of accessing your data source(s)?
  - How does the scenario in your sequence diagram work? What other scenarios could occur?

Recap: Team Assignment #3: Design Model

- **Grading criteria**
  - Class diagram is complete design model (with regard to classes, methods, attributes, Model/Control/Storage layers) (25%)
  - Relationships between classes, and placement of attributes and methods in classes, are plausible (25%)
  - Sequence diagrams show plausible roundtrip through the MVC layers / communication across teams, consistent with class diagram (40%)
  - UML diagrams are syntactically correct (10%)

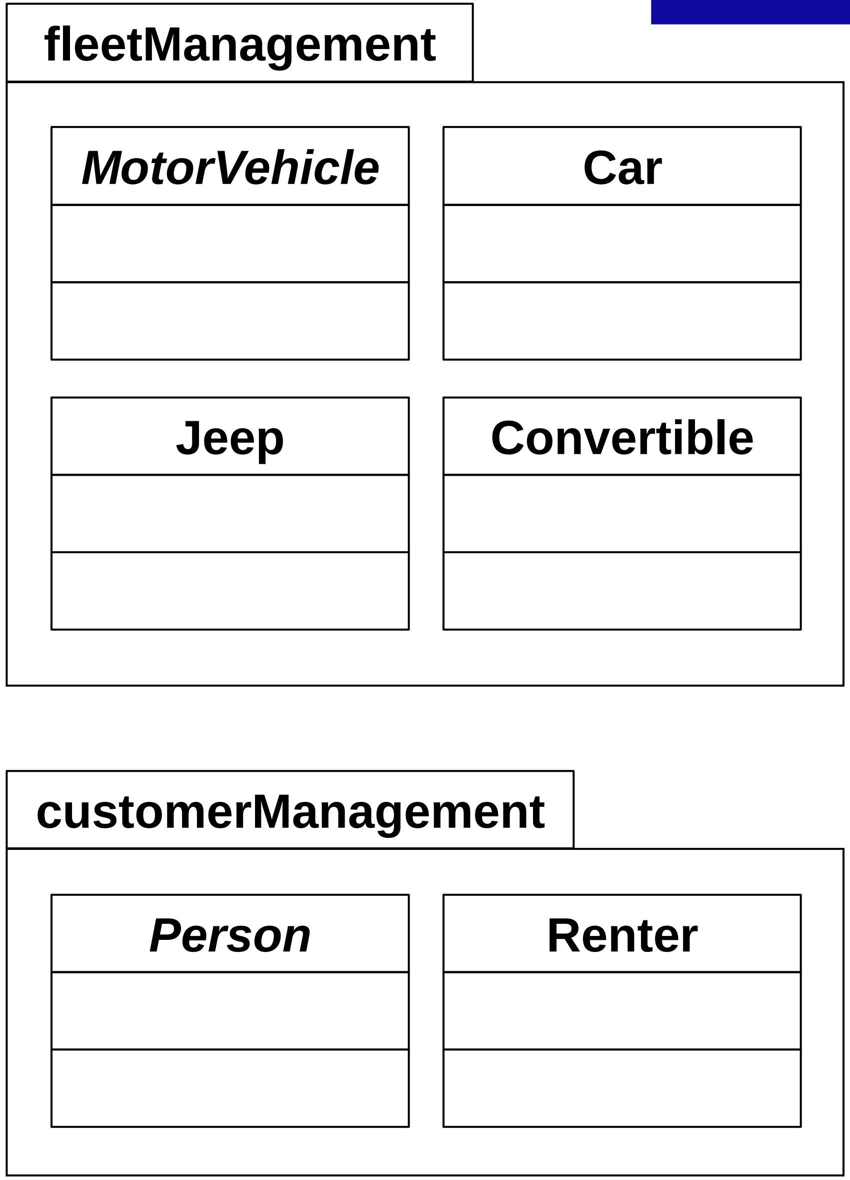
- **Deadlines (mandatory for assignment grade / optional for bonus grade)**
  - by **Wed 9 Mar 12:00**, submit your anonymized draft to the Peer Feedback Assignment #3
    - the earlier you submit, the more time your reviewers have to give you feedback
  - on **Wed 9 Mar 15:00-18:15**, discuss your draft with your tutor
  - by **Fri 11 Mar 23:59**, submit your peer feedback on the drafts assigned to you
    - the earlier you submit, the more time the other students have to incorporate your feedback
  - by **Sun 13 Mar 23:59**, submit your final PDF document to the Team Assignment #3
  - by **Wed 16 Mar 12:00**, rate the quality / usefulness of the peer feedback you received
  - on **Wed 16 Mar 15:00-18:15**, present and explain your model to your tutor

Packages and Visibility

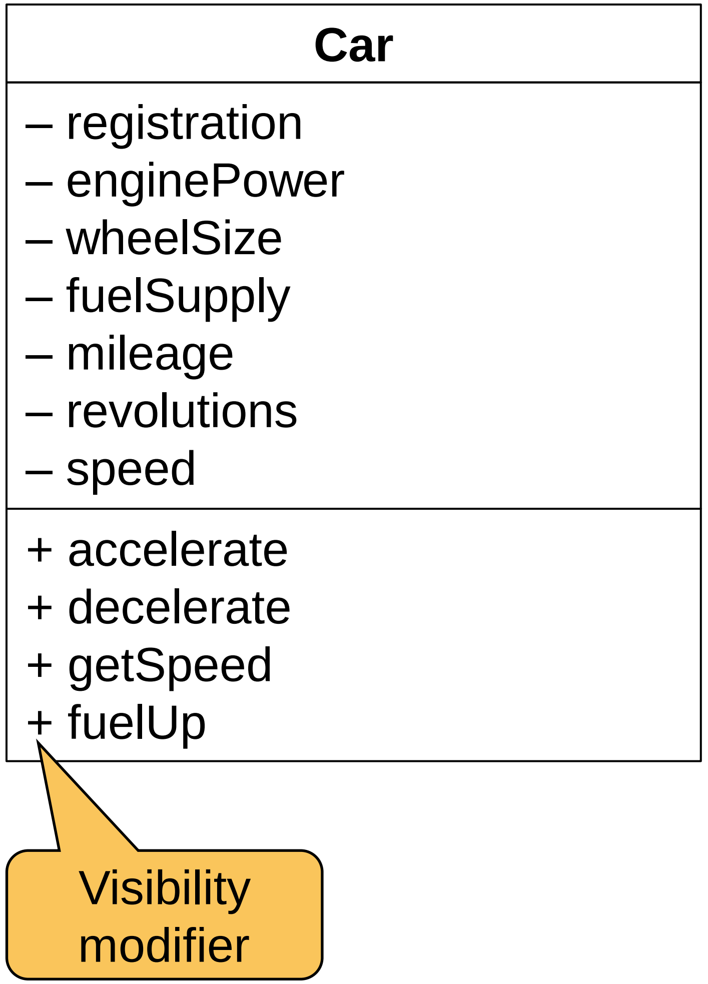
Packages

- Classes can be organized in packages
  - To help structure our system
  - To enforce visibility rules *(discussed next)*
- Modeling in UML: Package Diagram
- Declaration in Java:
  - State package at beginning of class source
  - Place `.java` files in matching subfolders

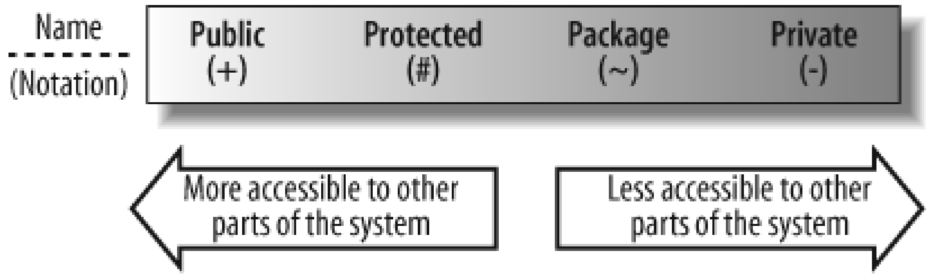
```
package fleetManagement;

public class Car {
   ...
}
```



Recap: Visibility



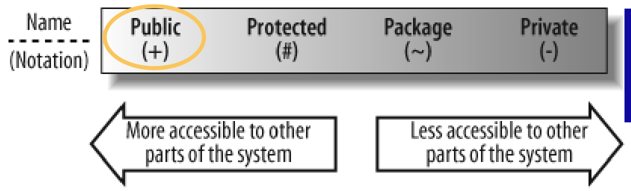
- To facilitate encapsulation, the visibility of classes, attributes and methods can be restricted with visibility modifiers.
- Attributes almost always have private visibility.
  - Instead of being directly manipulated by others, they should only be changed by methods in the same class.
  - Ensures that you have control over the attributes' values and can prevent any invalid values that may lead to errors.
- Methods may have any of the visibilities, depending on who they are providing functionality to:



Name
(Notation)
Public (+)
Protected (#)
Package (~)
Private (-)
More accessible to other parts of the system
Less accessible to other parts of the system

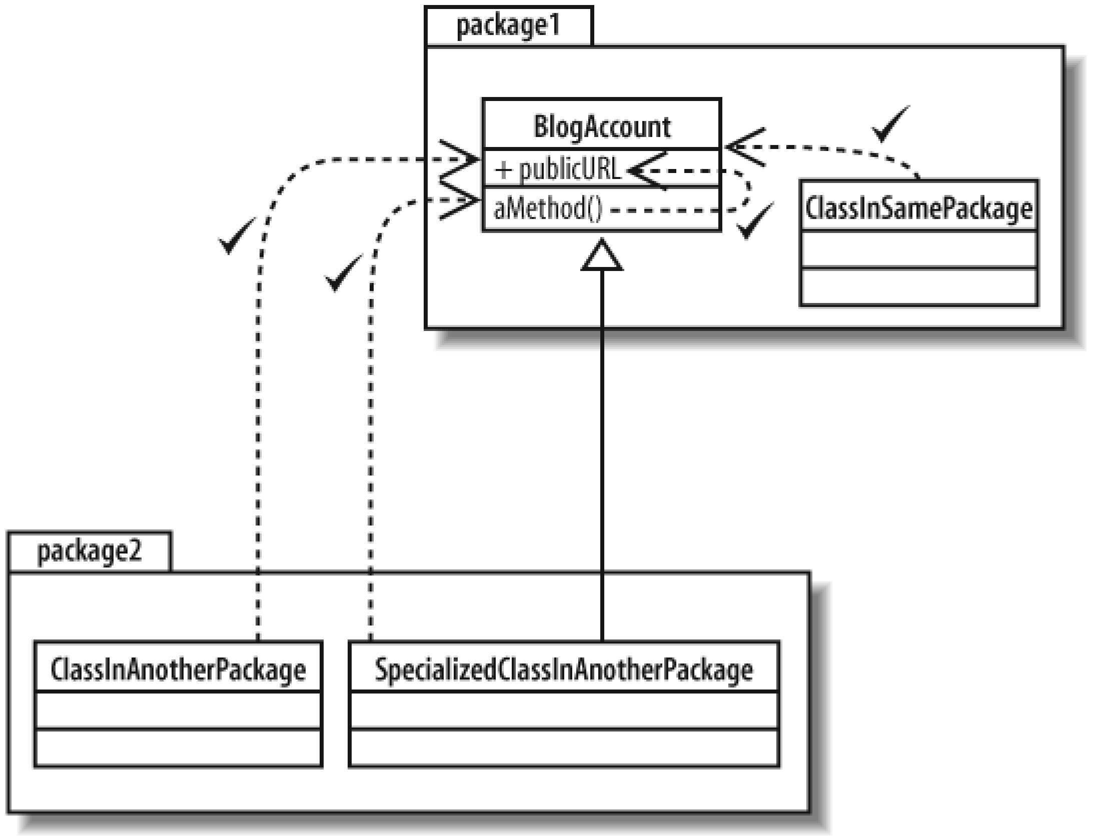
Miles, Hamilton: Learning UML 2.0. O'Reilly, 2006

Public Visibility

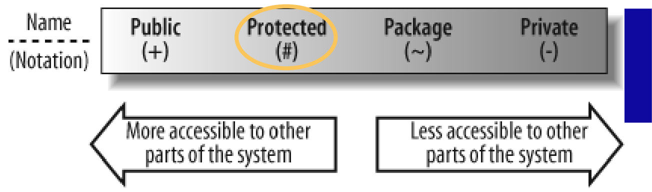


Name
(Notation)
Public (+)
Protected (#)
Package (~)
Private (-)
More accessible to other parts of the system
Less accessible to other parts of the system

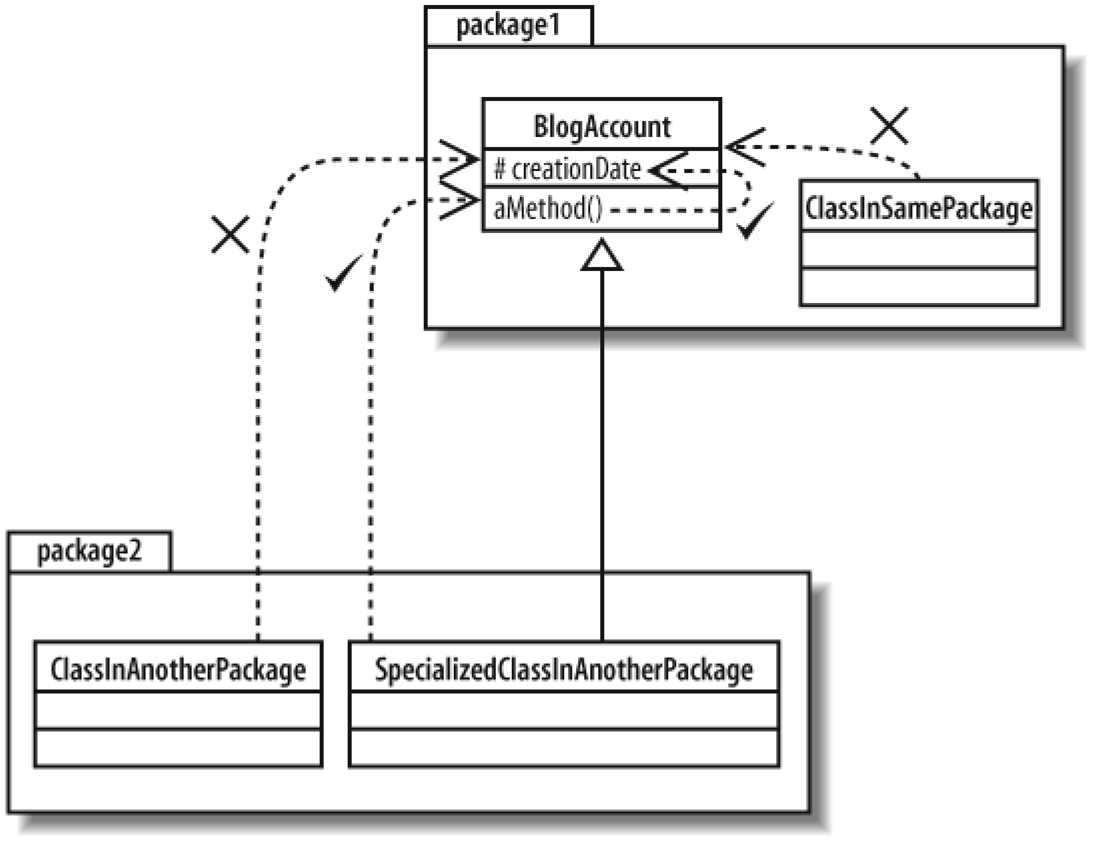
- **The attribute or operation is accessible by any other class.**
- Typically never used for attributes
  - They should be encapsulated by their class.
- Used for methods that are part of the class' public interface
  - i.e. features the class provides to "the world"
    - This means other people will be relying on the functionality, so you can't simply change it!
  - For most classes, a public interface is not desirable though
    - It is sufficient if a component is represented to the outside world by just a few components
- in Java: Visibility modifier **`public`**



Protected Visibility

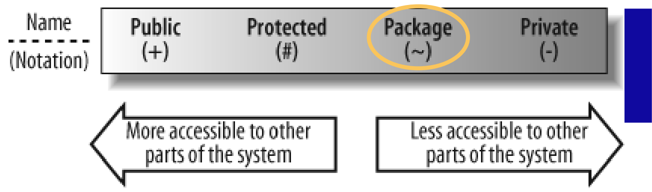

Name
(Notation)
Public (+)
Protected (#)
Package (~)
Private (-)
More accessible to other parts of the system
Less accessible to other parts of the system

- **The attribute or operation is accessible by any subclass (i.e. inheriting class).**
- Chosen for attributes or methods that are not part of the class' public interface, but that need to be accessed by methods of subclasses in order to extend or change your class' functionality
  - Caution: In a way, this *is* opening up another kind of public interface, because you have no control over who is inheriting from your class, and how they are using your attributes and methods. You still need to always provide the functionality they are expecting from your methods though.
- in Java: Visibility modifier `protected`

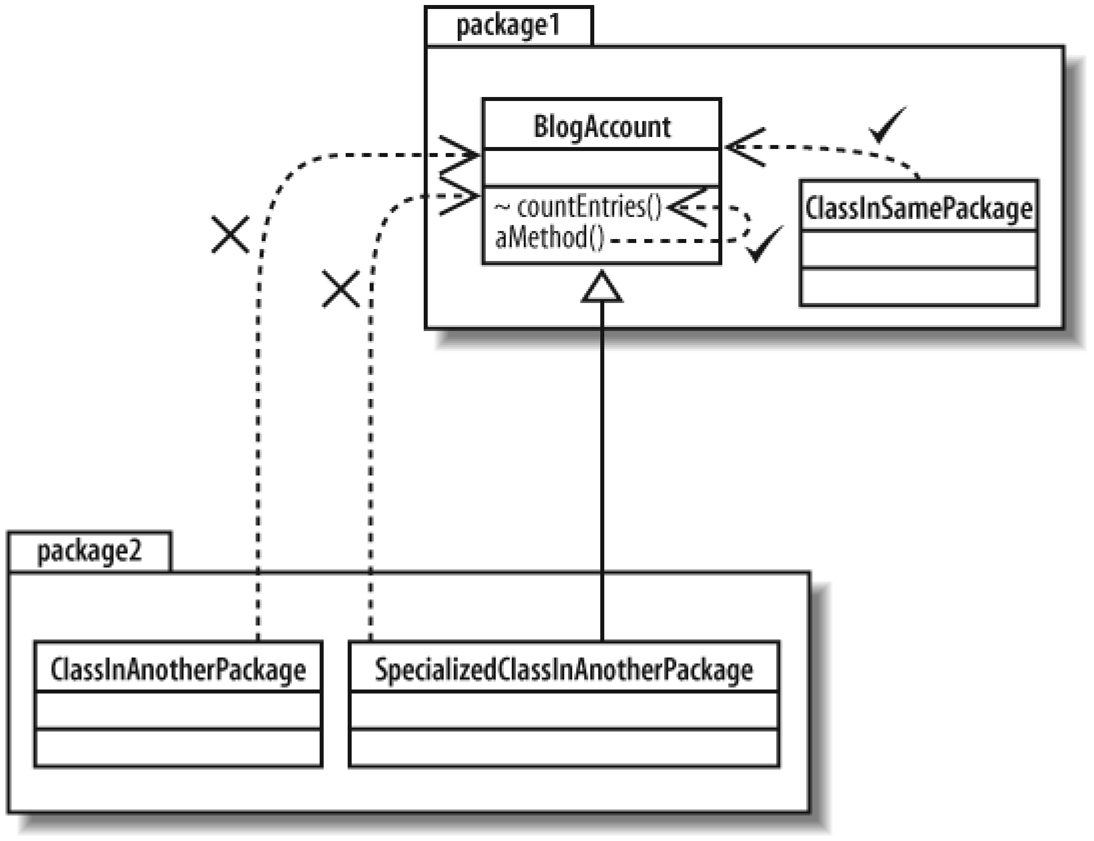


Miles, Hamilton: Learning UML 2.0. O'Reilly, 2006

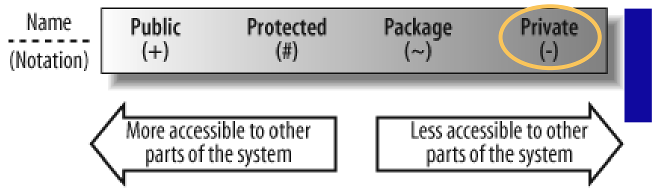
Package Visibility


Name
(Notation)
Public
(+)
Protected
(#)
Package
(~)
Private
(-)
More accessible to other parts of the system
Less accessible to other parts of the system

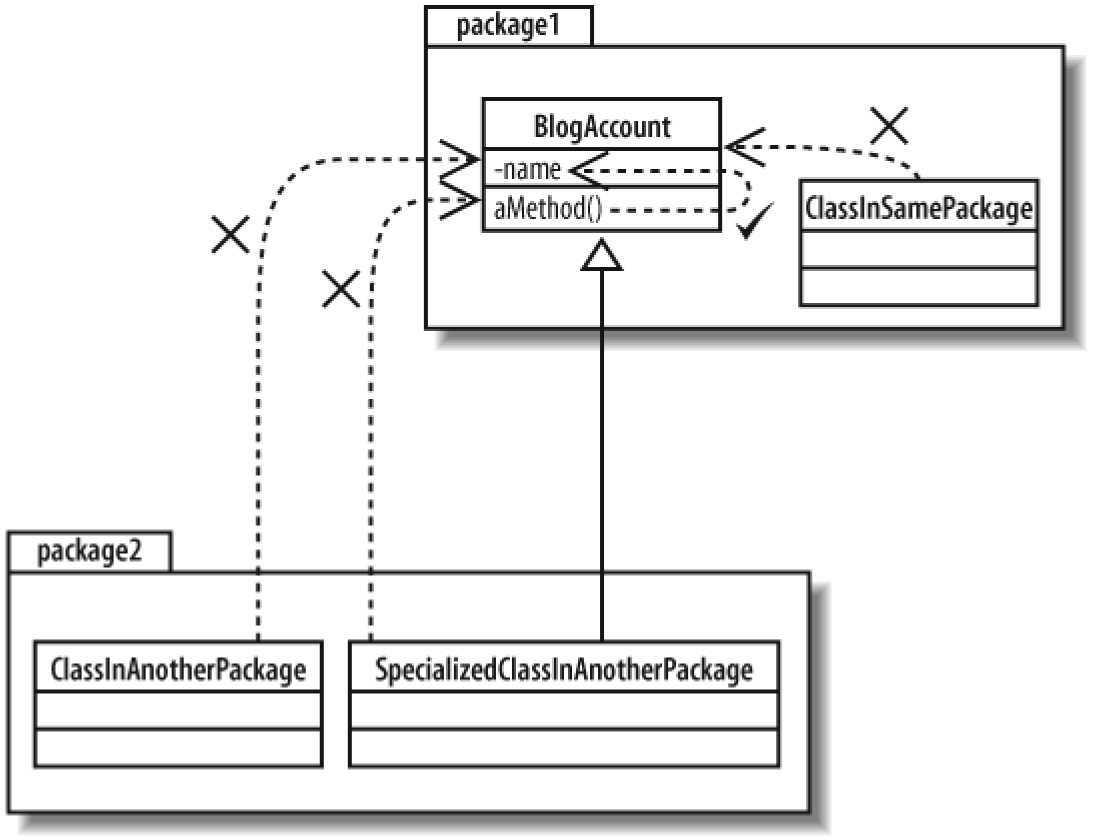
- **The attribute or operation is accessible by any class in the same package.**
- Typically never used for attributes
  - They should be encapsulated by their class.
- Used for methods that are part of the class' interface for use in your component
  - i.e. features the class provides to its "family"
  - Do not use lightly even if you know who is building the other classes in your component.
  - The less interfaces you open up to anyone, the less other classes can depend on your internals, and the more flexible you remain to change them.
- in Java: no ("default") visibility modifier



Private Visibility


Name
(Notation)
Public
(+)
Protected
(#)
Package
(~)
Private
(-)
More accessible to other parts of the system
Less accessible to other parts of the system

- **The attribute or operation is accessible only within the same class.**
- Used for almost all attributes
  - Ensures encapsulation in their class:
    - No-one else can modify them and potentially set them to invalid / inconsistent values
    - No-one else can read them and rely on implementation details of your object's state that you may want to change sometime
- Used for methods that are part of a class' internal mechanisms
  - i.e. functions that are not part of any interface, but just structuring a class' inner workings
- in Java: Visibility modifier `private`



Miles, Hamilton: Learning UML 2.0. O'Reilly, 2006

Break

UNIVERSITY OF ICELAND

Object-Oriented Design Principles

see also:

Head First Object-Oriented Analysis and Design, Chapter 8

Object-Oriented Design Challenges

- Object-oriented thinking is relatively easy when dealing with real-life objects
- Applying the same principles to more abstract / technical objects is harder:
  - Who are the “actors”?
  - What are their responsibilities?
  - How can we encapsulate information and functionality?
  - How can we support re-use, extension and change without high refactoring efforts?
  - How can we ensure our class structure is easy to understand?

- To help us create clean and efficient designs, we can rely on
  - OO design principles: Rules we should follow in designing class structures *(following slides)*
  - OO design patterns: Proven ways of arranging classes to solve certain problems *(later class)*

1. Single Responsibility Principle (High Cohesion)

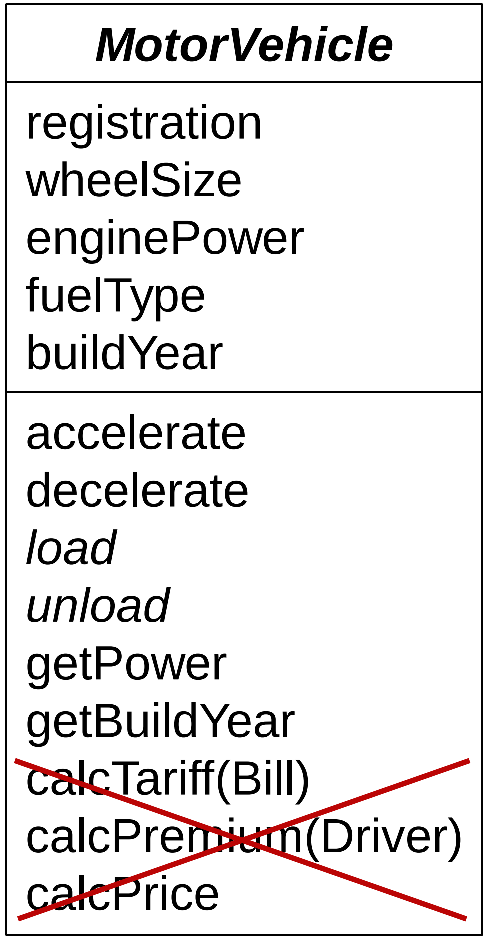
- **Each module (i.e. package/component/class) in your system should have a single responsibility, and all of the module's implementation should be focused on carrying out that responsibility.**

- Goal: Each module should have a clearly defined role in the system

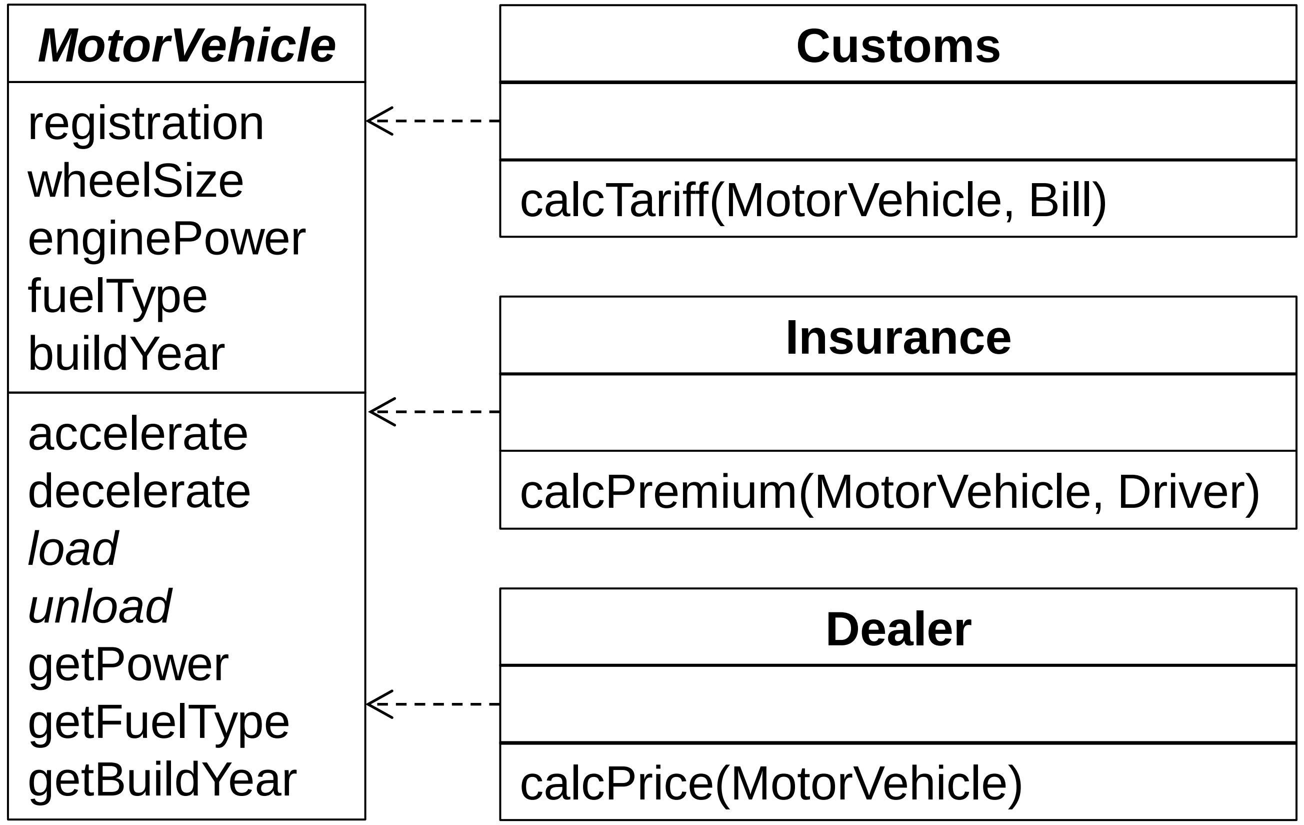
- A module should only need to be changed when its own requirements change
  - It should not need to be changed because of changing requirements for other modules!

- Supports teamwork, fault localization, maintainability, re-usability

Single Responsibility Principle
Example



- **Test:** Does “The <*class*> <*method*> <*parameters*> itself” make sense?
  - ✗ “The motor vehicle calculates premiums for the driver itself.”
  - ✓ “The insurance calculates premiums for the motor vehicle and driver itself.”
- Use common sense!



Single Responsibility Principle
Example

- The `Car` class' main responsibility should be executing all functions of a car
  - e.g. accelerating, decelerating, loading, unloading, gauging fuel, etc.

- Other functions may depend on attributes of the car,
  but are not core responsibilities of the car itself
  - e.g. calculating the customs tariff, insurance value, resale price etc.

- These functions are the responsibility of separate classes
  - which can refer to the `Car` class' methods for obtaining all necessary information,
    but need to implement the calculation themselves

➢ When customs tariffs change, we change the `Customs` class, not the `Car` class!

2. Separation of Concerns (Low Coupling)

- **Any requirement we have for the system should be represented in a single module (package/component/class), and not be spread across several modules.**

➢ We need to identify which module should be responsible for a task

- And how to implement it with minimal exposure to the other classes' properties

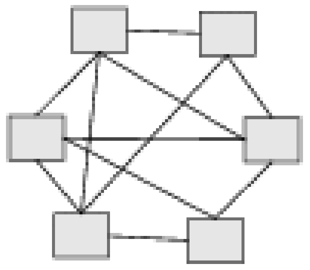
- Distinction:
  - **Single responsibility principle:** Each class implements only one requirement
  - **Separation of concerns:** Each requirement is implemented by only one class

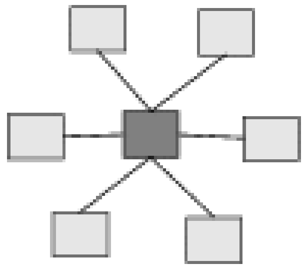
High Cohesion, Low Coupling

- The previous principles can also be formulated in two rules:

- **Maximize cohesion within a class.**
  - All parts of a class should be highly related and dependent on each other.
  - If there are parts without relation to other parts, it may make sense to move them into a class of their own.

- **Minimize coupling between classes.**
  - The number of classes that need to work together to solve a task should be minimized.
  - If several modules share responsibility for a certain task, it may make sense to move control of that task to a class of its own (which can then rely on the original classes for subtasks)
    - Caveat: A new class might not eliminate dependencies, but just hide them!





3. Don’t Repeat Yourself

- **Any functionality should be implemented only once within the system.**

➢ Avoid duplicate code by moving common functionality into a single location.

- This principle is not just about avoiding copy & paste!
  - It’s not just important to have functionality only in one place…
  - …but to make sure this place makes sense (cf. the single responsibility principle)

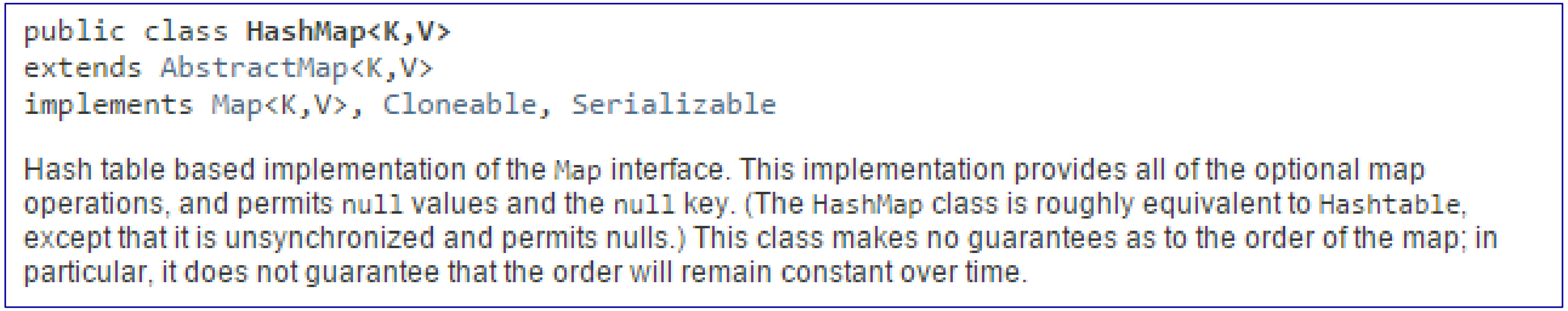
- “Duplicate code” here also means slightly different variations of similar code!
  - e.g. search algorithms for cars, trucks, jeeps….
  - Generalization, specialization and polymorphism can help us to avoid implementing different variants of certain code, and instead have one implementation that works in several contexts

4. Program to Interfaces

- **A module should always depend only on the published interfaces of other modules and not rely on any assumptions about their internal implementation.**

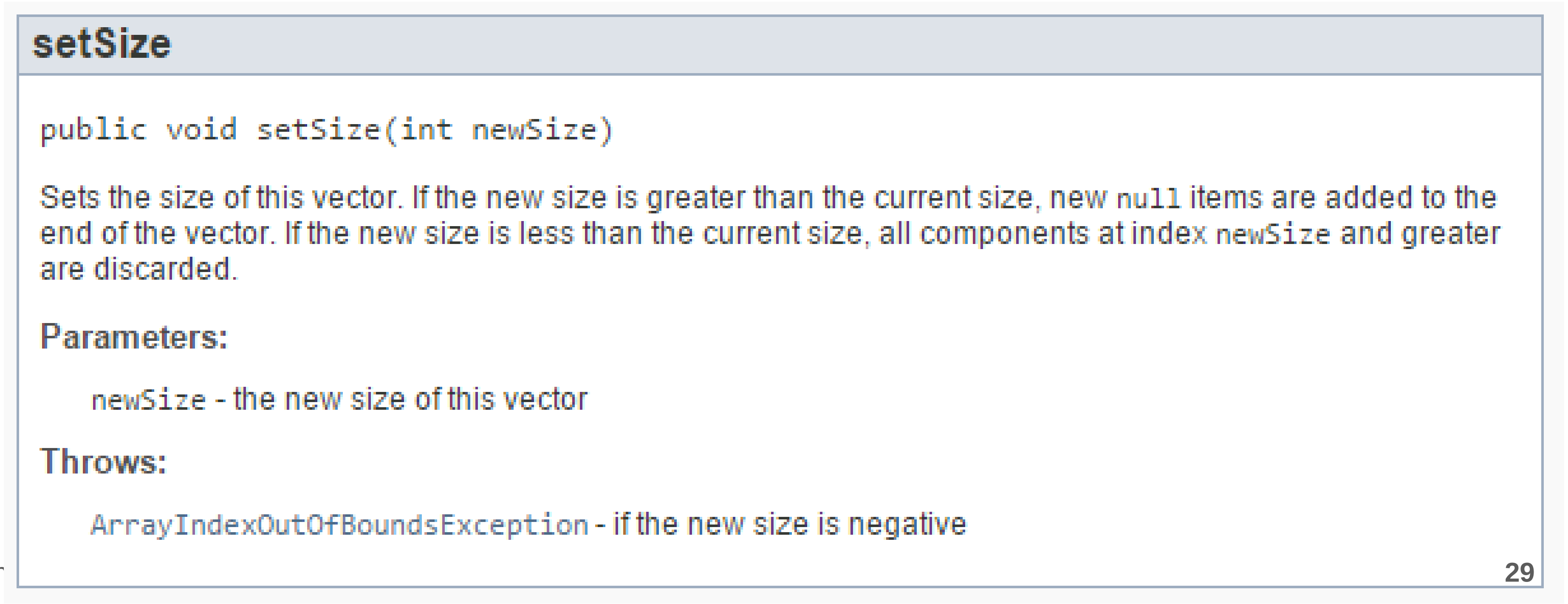
- The interface comprises not just the formal declaration of method signatures, but also published pre- and post-conditions, error signals (exceptions), treatment of special input values, etc.
- This is a contract that goes both ways!
  - Outside modules must not rely on anything but the published interface.
  - The internal implementation may change anytime, but must always satisfy the interface.
- Anything not published can change
  - e.g. stability of a sorting algorithm

Program to Interfaces
Java API Documentation Examples



```
public class HashMap<K,V>
extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

Hash table based implementation of the `Map` interface. This implementation provides all of the optional map operations, and permits `null` values and the `null` key. (The `HashMap` class is roughly equivalent to `Hashtable`, except that it is unsynchronized and permits nulls.) This class makes no guarantees as to the order of the map; in particular, it does not guarantee that the order will remain constant over time.



setSize

```
public void setSize(int newSize)
```

Sets the size of this vector. If the new size is greater than the current size, new `null` items are added to the end of the vector. If the new size is less than the current size, all components at index `newSize` and greater are discarded.

Parameters:

`newSize` - the new size of this vector

Throws:

`ArrayIndexOutOfBoundsException` - if the new size is negative

5. Open/Closed Principle
for Class Implementations

- **Classes should be open for extension, but closed for modification.**

- For a **class' implementation**, the inheritance mechanism already ensures this:
  - The implementation of a method cannot be changed by other classes.
  - But a subclass can extend the class and provide a modified implementation of the method…
    - …which may or may not invoke the original implementation
      - Limitation: It can only *incorporate* the original implementation, but not change individual aspects of it

- Caution: Somewhat risky since we have no idea how the subclass' method will work with the superclass' other methods and attributes!
  - Might introduce bugs, violate invariants etc., even if no malicious intent is present.
  - Any public, protected or package-visible method can be overridden by a subclass!
    - …unless it is declared as `final`, i.e. not overrideable
      (which makes it safe, but violates the Open/Closed Principle)

Open/Closed Principle
Limitations and Risks in Extending Implementations

```
public class Car {
  public void accelerate(
      int kmh) {
    engageGear();
    while (getSpeed() < kmh) {
      increaseCombustion(1);
    }
    neutralEngine();
  }
}
```

```
public class Hybrid extends Car
{
  public void accelerate(
      int kmh) {
    choosePowerSource(kmh);
    super(kmh);
    logPowerConsumption();
  }
}
```

```
public class Buggy extends Car
{
  public void accelerate(
      int kmh) {
    increaseCombustion(100);
  }
}
```

Open/Closed Principle
for Class Interfaces

- **Classes should be open for extension, and closed for modification.**

- For a **class' interface**, this requires developer discipline and foresight:
  - An agreed and published interface should never be modified, since that may break other classes that rely on it!
- So the only way to "safely" modify a published class' interface is to extend it in a subclass or use it as a delegate in a class that provides the new interface.
  - Other classes can then choose which interface to rely on.
  - Note: A subclass' interface must not change the superclass' interface at will, just extend it (Reason: Liskov Substitution Principle)
- Note: Classes offer a different interface at each level of visibility!
  - The more of them the Open/Closed Principle can be applied to, the better dependent classes are insulated from interface changes.

Open/Closed Principle
Limitations in Extending Interfaces

```java
public abstract class
    MotorVehicle {
  public abstract void
        accelerate(int toKmh);
}
```

- Suppose we want to use relative instead of target speed acceleration
- Not possible to
  - Simply overwrite **`accelerate`** with a relative acceleration algorithm
    - Reason: Program to Interfaces Principle
  - Add abstract **`accelerateRel`** method
    - Would break other subclasses who would suddenly be required to implement it

```java
public class Car
    extends MotorVehicle {

  public void accelerate(
        int toKmh) {
    // target speed impl.
  }

  public void accelerateRel(
        int byKmh) {
    // relative speed impl.
  }
}
```

➢Need extra method in subclass

6. Liskov Substitution Principle

- **Instances of superclasses should be replaceable with instances of their subclasses without altering any desirable properties (correctness, behavior etc.) of the program.**

- This is only the case when
  - the subclass is a true extension of the superclass (i.e. it only *expands* the superclass' behavior, but does not *modify* the original behavior)
    - So when we use it in place of the superclass, the extensions are ignored and we just use the still-intact inherited functionality

- This is why we could not simply override the target-speed `accelerate` method of `MotorVehicle` with a relative-speed `accelerate` method in `Car`: The system would then behave differently if we substituted `MotorVehicle` with `Car`
  - So we had to put the extended functionality into a method of its own (`accelerateRel`).

Liskov Substitution Principle
Limitations

```
public class Rectangle {
 private int width, height;

 public void setWidth(int w) {
   width = w;
 }
 public void setHeight(int h) {
   height = h;
 }
 public int getWidth() {
   return width;
 }
 public int getHeight() {
   return height;
 }
}
```

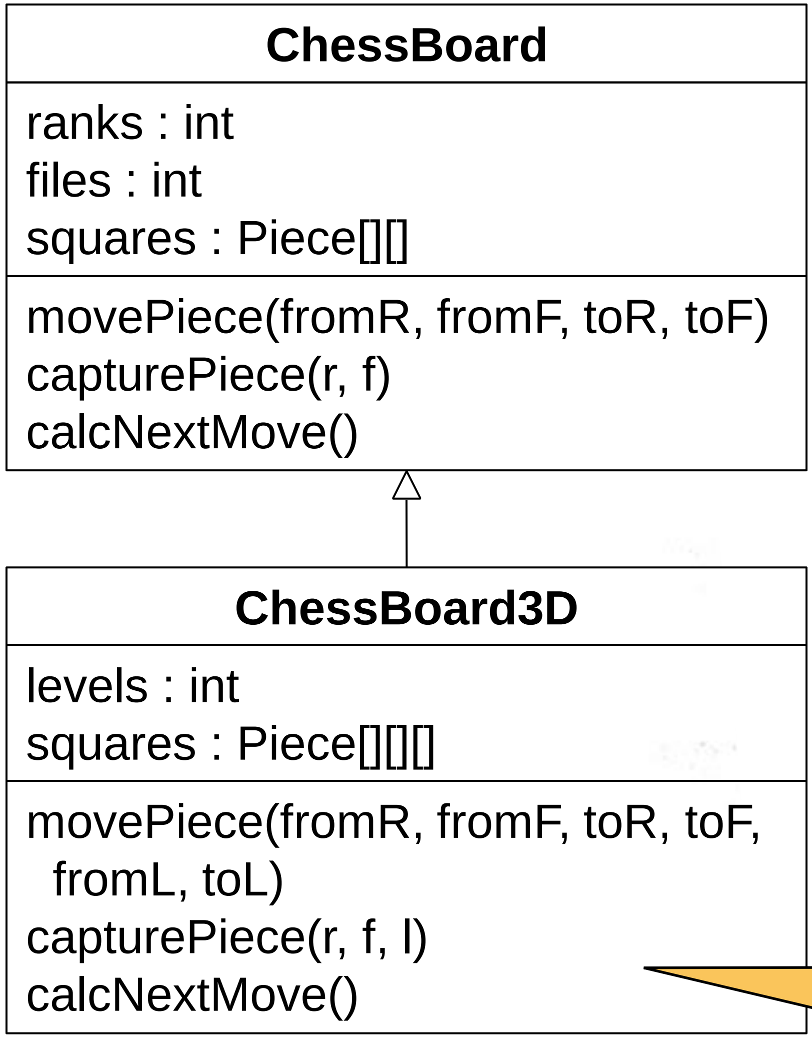
```
public class Square
    extends Rectangle {
 public void setWidth(int w) {
   width = w;
   height = w;
 }
 public void setHeight(int h) {
   width = h;
   height = h;
 }
}
```

- `Rectangle r = new Square()` will behave like a square, not a rectangle!
  - ➢ Violation of Liskov Substitution Principle
    - Whether that is a problem depends on the application and the purpose of inheritance

Using Delegation Instead of Inheritance

Motivation:

- Suppose we want to model a chess game…

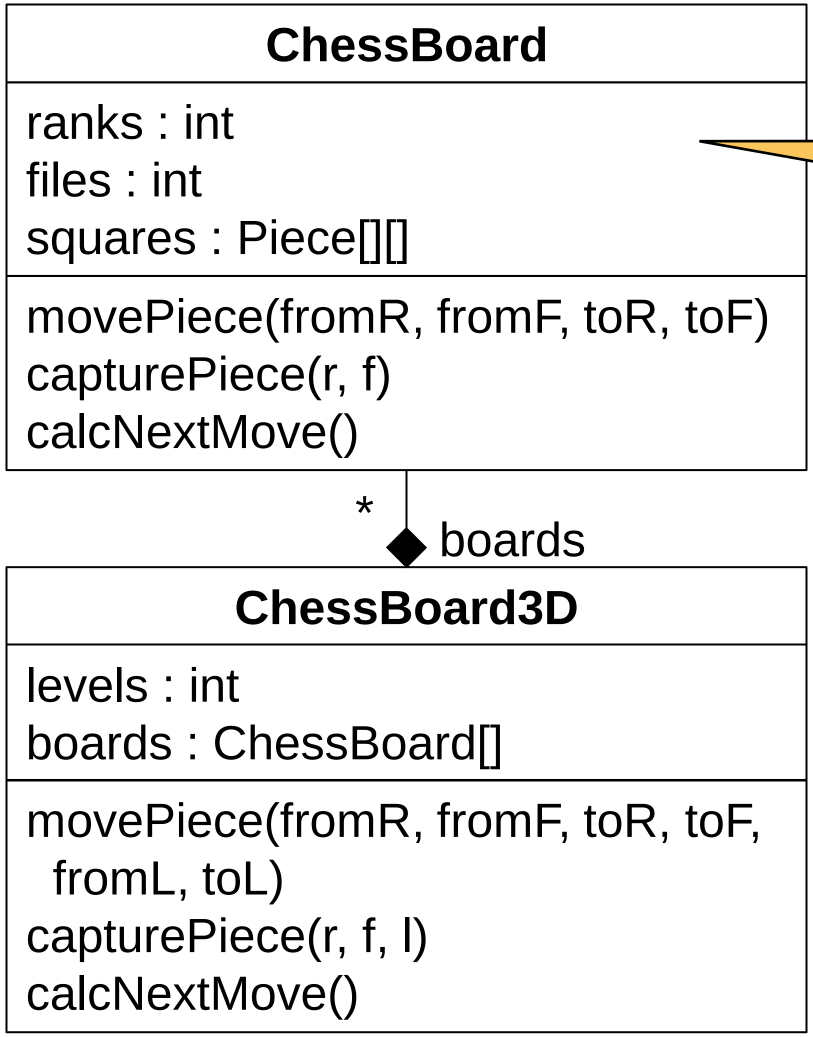


- …in three dimensions!
  - Naïve approach: Extend ChessBoard

- Not that much re-use :(
- and we inherit lots of inapplicable methods :(

Using Delegation Instead of Inheritance

- Better approach: Instead of *extending* the **`ChessBoard`** class, we can *use* it inside the **`ChessBoard3D`** class

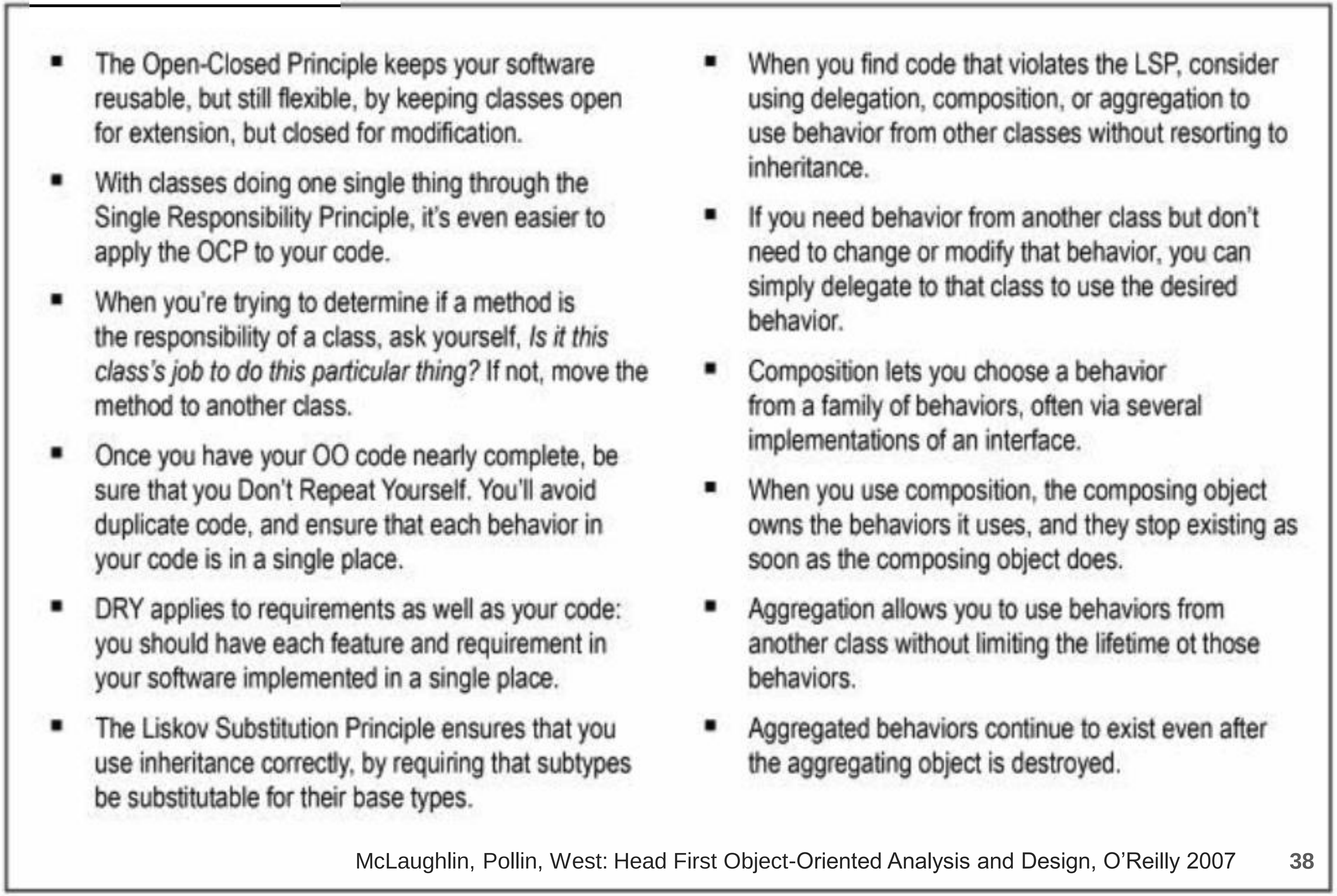


Different levels can even have different sizes now!

- **`ChessBoard3D`** is still responsible for implementing the 3D game, but it *delegates* all the 2D stuff to the original **`ChessBoard`**.

- Guideline: If you need to use functionality of another class, but don't want to change any of that functionality, consider using **delegation** instead of inheritance.

Summary



- The Open-Closed Principle keeps your software reusable, but still flexible, by keeping classes open for extension, but closed for modification.
- With classes doing one single thing through the Single Responsibility Principle, it's even easier to apply the OCP to your code.
- When you're trying to determine if a method is the responsibility of a class, ask yourself, *Is it this class's job to do this particular thing?* If not, move the method to another class.
- Once you have your OO code nearly complete, be sure that you Don't Repeat Yourself. You'll avoid duplicate code, and ensure that each behavior in your code is in a single place.
- DRY applies to requirements as well as your code: you should have each feature and requirement in your software implemented in a single place.
- The Liskov Substitution Principle ensures that you use inheritance correctly, by requiring that subtypes be substitutable for their base types.
- When you find code that violates the LSP, consider using delegation, composition, or aggregation to use behavior from other classes without resorting to inheritance.
- If you need behavior from another class but don't need to change or modify that behavior, you can simply delegate to that class to use the desired behavior.
- Composition lets you choose a behavior from a family of behaviors, often via several implementations of an interface.
- When you use composition, the composing object owns the behaviors it uses, and they stop existing as soon as the composing object does.
- Aggregation allows you to use behaviors from another class without limiting the lifetime ot those behaviors.
- Aggregated behaviors continue to exist even after the aggregating object is destroyed.

McLaughlin, Pollin, West: Head First Object-Oriented Analysis and Design, O'Reilly 2007

In-Class Quiz #9: Object-Oriented Design Principles

Match the names of the following principles to their descriptions:

a) Single Responsibility Principle
b) Separation of Concerns
c) Don’t Repeat Yourself
d) Program to Interfaces
e) Open/Closed Principle
f) Liskov Substitution Principle

1. Abstract out things that are common and place them in a single location.
2. An object’s implementation should focus on carrying out that object’s single responsibility.
3. Classes should be open for extension, but closed for modification.
4. Classes should depend on each other as little as possible.
5. Don’t rely on details of an implementation, but only on what is defined by an interface.
6. Superclasses should be substitutable by subclasses without changing expected behavior.

UNIVERSITY OF ICELAND

Thank you!

book@hi.is