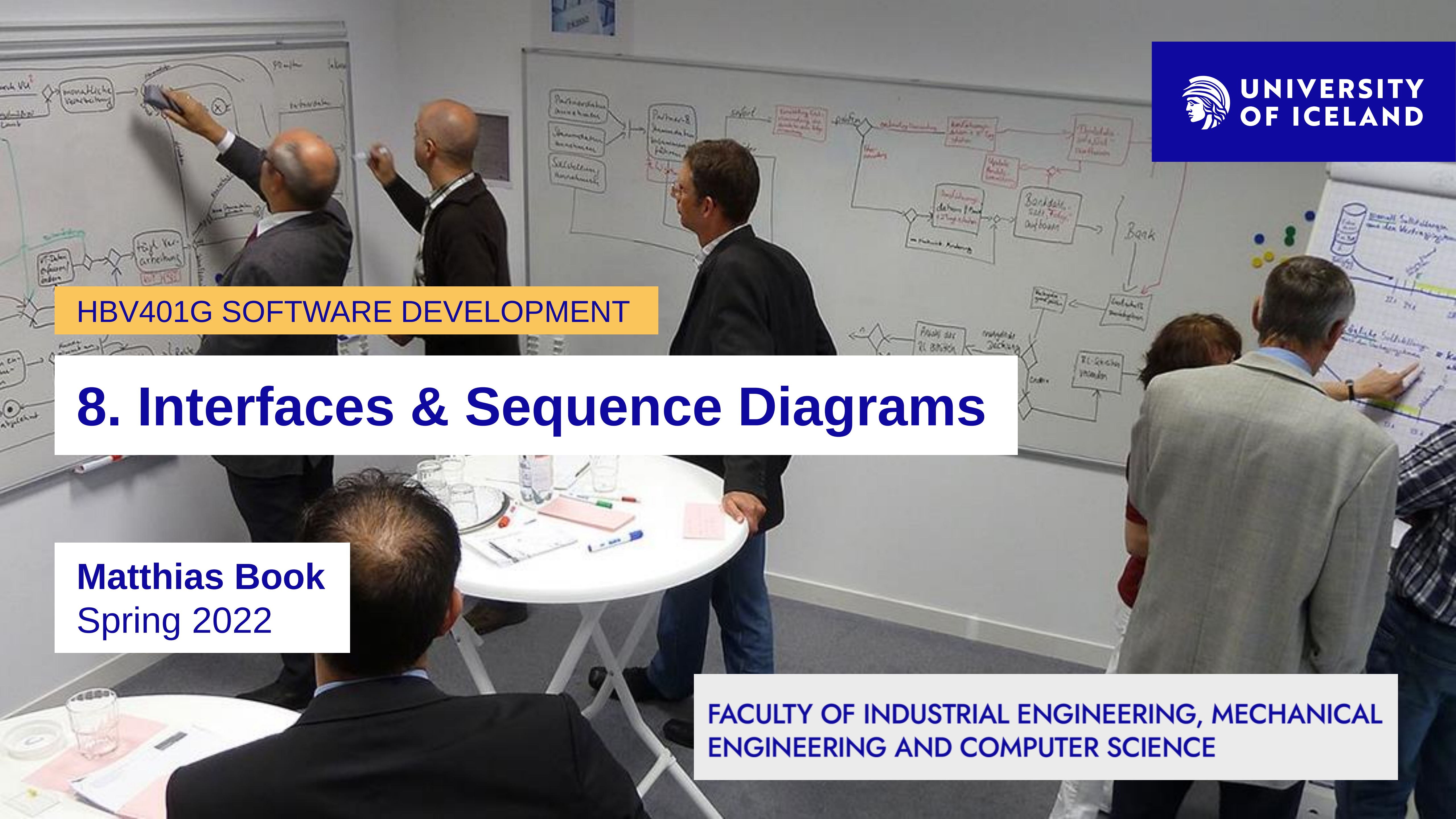


UNIVERSITY OF ICELAND

HBV401G SOFTWARE DEVELOPMENT

8. Interfaces & Sequence Diagrams

Matthias Book
Spring 2022

FACULTY OF INDUSTRIAL ENGINEERING, MECHANICAL ENGINEERING AND COMPUTER SCIENCE

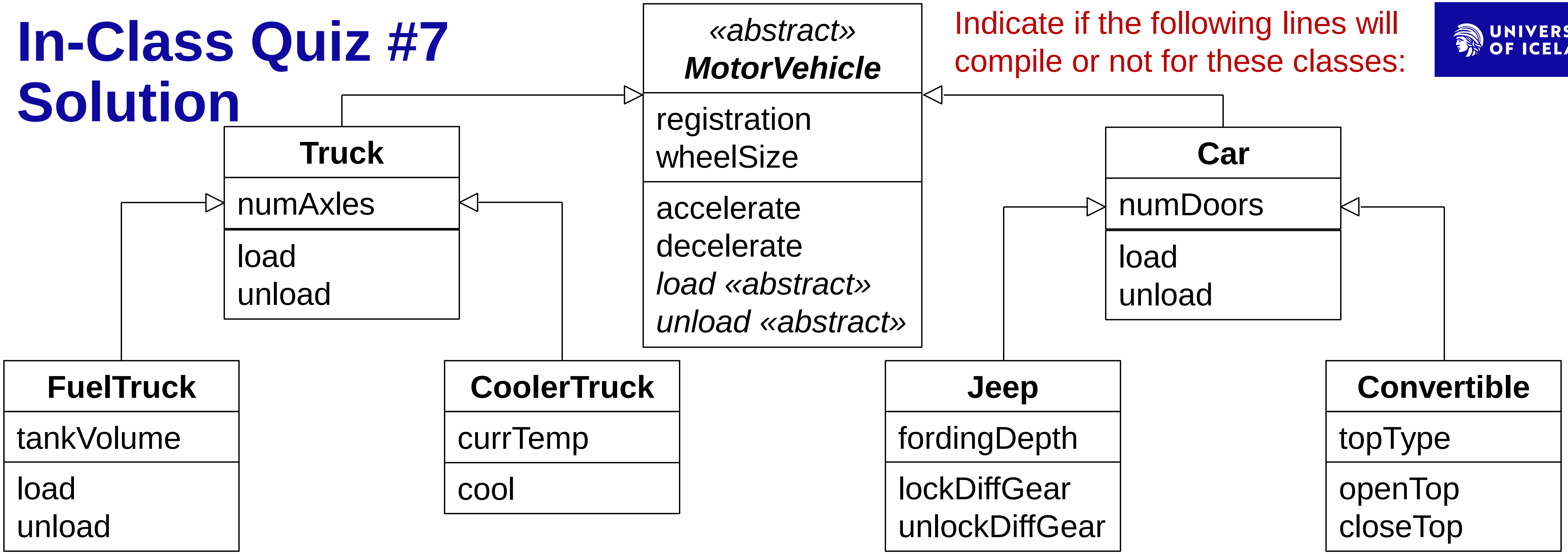
Recap: Polymorphism

- Overridden methods exist in several forms – they are “polymorphic”.
- We have the choice of treating objects in their specific or generic form:
  - If we need to work with an object’s specific features, we refer to it through the interface provided by its own class definition (e.g. FuelTruck).
  - If we do not care about an object’s specific features, but only its general characteristics, we can refer to it through the interface provided by a superclass that is appropriate for our purpose (e.g. Truck or even MotorVehicle).
    - In that case, we do not even have to care about which specific class our object is an instance of – we simply refer to it as if it was an instance of the appropriate superclass.
    - Helpful e.g. when storing objects of related types, cycling through lists of related objects, etc.

➢ This gives us great flexibility in working with objects in a “natural” way, and in adding or changing specific implementations without having to touch most other code that only relies on the generic aspects.
  - One of the key benefits of object-oriented design and programming

In-Class Quiz #7 Solution

Indicate if the following lines will compile or not for these classes:



a) `Jeep j = new Jeep(); j.lockDiffGear();`	YES
b) `MotorVehicle mv = new MotorVehicle(); mv.accelerate();`	NO
c) `Convertible conv = new Convertible(); conv.load();`	YES
d) `Car car = new Convertible(); car.openTop();`	NO
e) `CoolerTruck ct = new Truck(); ct.currTemp = -18;`	NO
f) `Truck t = new FuelTruck(); t.numAxles = 3;`	YES
g) `CoolerTruck cool = new CoolerTruck(); cool.unload();`	YES

Recap: Abstract Classes and Operations

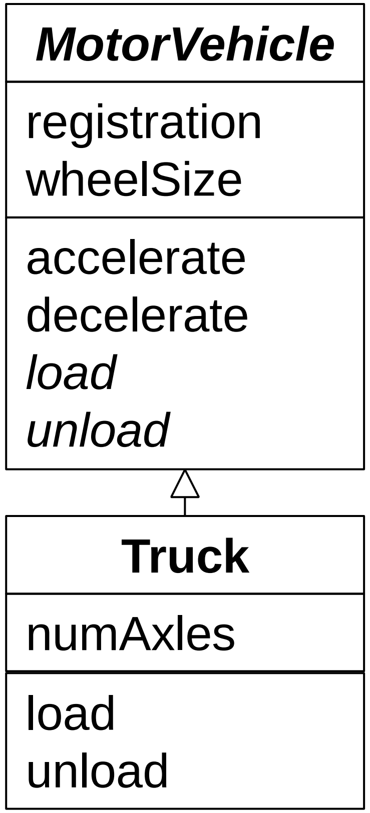
- Occasionally, we introduce a superclass only for structural purposes
  - i.e. we don't want to create instances of it, but bundle some general characteristics that will be inherited and extended by specialized subclasses.
- And/or we might not be able to provide general implementations for some of a superclass' methods, but still want to ensure that those methods will be provided in specialized form by the subclasses
  - i.e. we want to enforce an interface that all subclasses must conform to
- In these cases, we can mark the method and/or class as "abstract"
  - Meaning: No concrete instances of the class can exist
- A class must be abstract if at least one of its methods is abstract
  - But it can also be declared abstract even if all its methods are implemented
- Subclasses must provide implementations for their superclass' abstract methods or be declared as "abstract" themselves

Recap: Abstract Classes in Java

```java
public abstract class MotorVehicle {
   ...
   public abstract void load(...);
   public abstract void unload(...);
}

public class Truck extends MotorVehicle {
   ...
   public void load(...) {
      /* specialized implementation */
   }
   public void unload(...) {
      /* specialized implementation */
   }
}
```

Note:
No method body!

Compiler error if we omit any of these, unless we declare **`Truck`** as **`abstract`** as well



Interfaces

see also:

Learning UML 2.0, Chapters 4 and 5

Interfaces
(as modelling/programming language constructs)

a) The term “interface” is usually understood as the set of methods that a class (or a component) makes available to other classes or components (subject to visibility restrictions).

b) In OOA/OOD/OOP, the term “interface” can also refer to an actual modeling/programming language construct
   - In UML, written as «*interface*», and in Java, written as **`interface`**

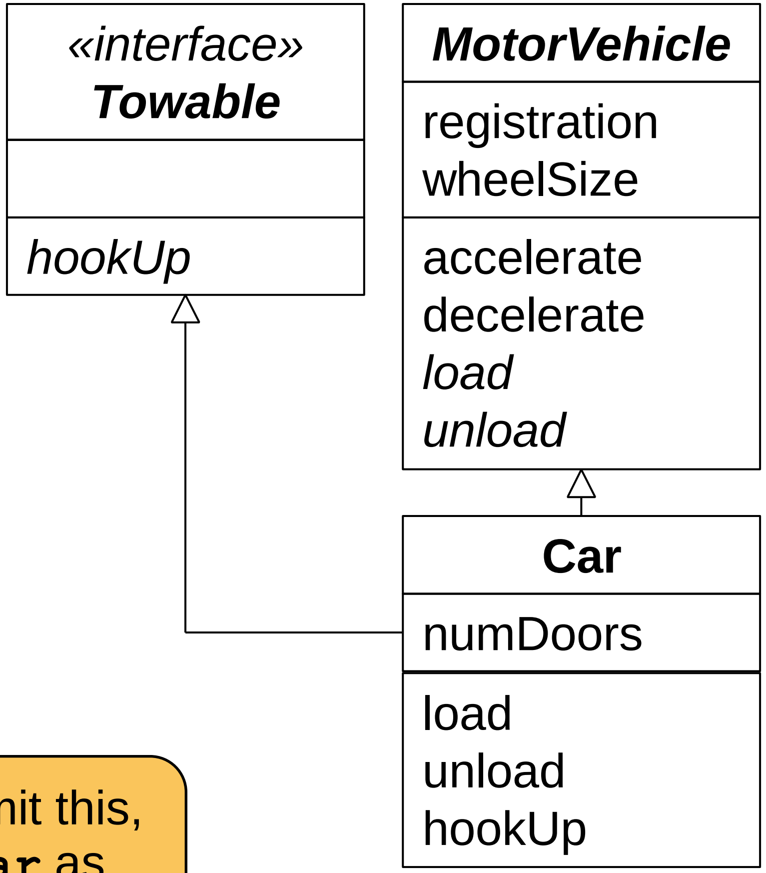
- Similar to an abstract class that **contains only abstract methods**, an «*interface*»/**`interface`** (as in b) prescribes an interface (as in a) that subclasses need to satisfy (by “implementing” it).

Interfaces in Java

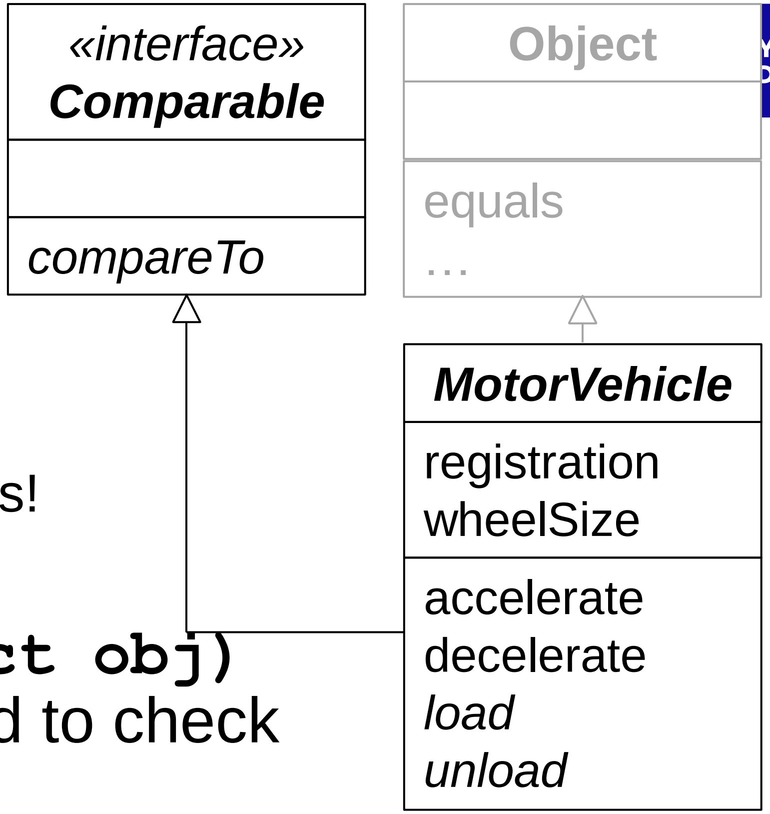
```
public interface Towable {
   ...
   public void hookUp(...);
}

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

Note:
No method body!

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



Example: Ordering Objects



- Comparison with ==, **<** and **>** does not work on objects
  - Operators “do not know” which attributes need to be compared in which way in order to establish equality or order of objects
  - ➢ Developers must implement equality and comparison operations!
- All objects inherit the method `boolean equals(Object obj)` from their implicit `Object` superclass, which is supposed to check if `this` object and the given `obj` are equal.
- Objects that should additionally have a natural order must implement the method `int compareTo(T o)` of the interface `Comparable<T>`
  - Return value < 0: this object is smaller than object `o`
  - Return value = 0: this object is equal to object `o`
  - Return value > 0: this object is greater than object `o`

Declaration of `equals` and `compareTo` Methods

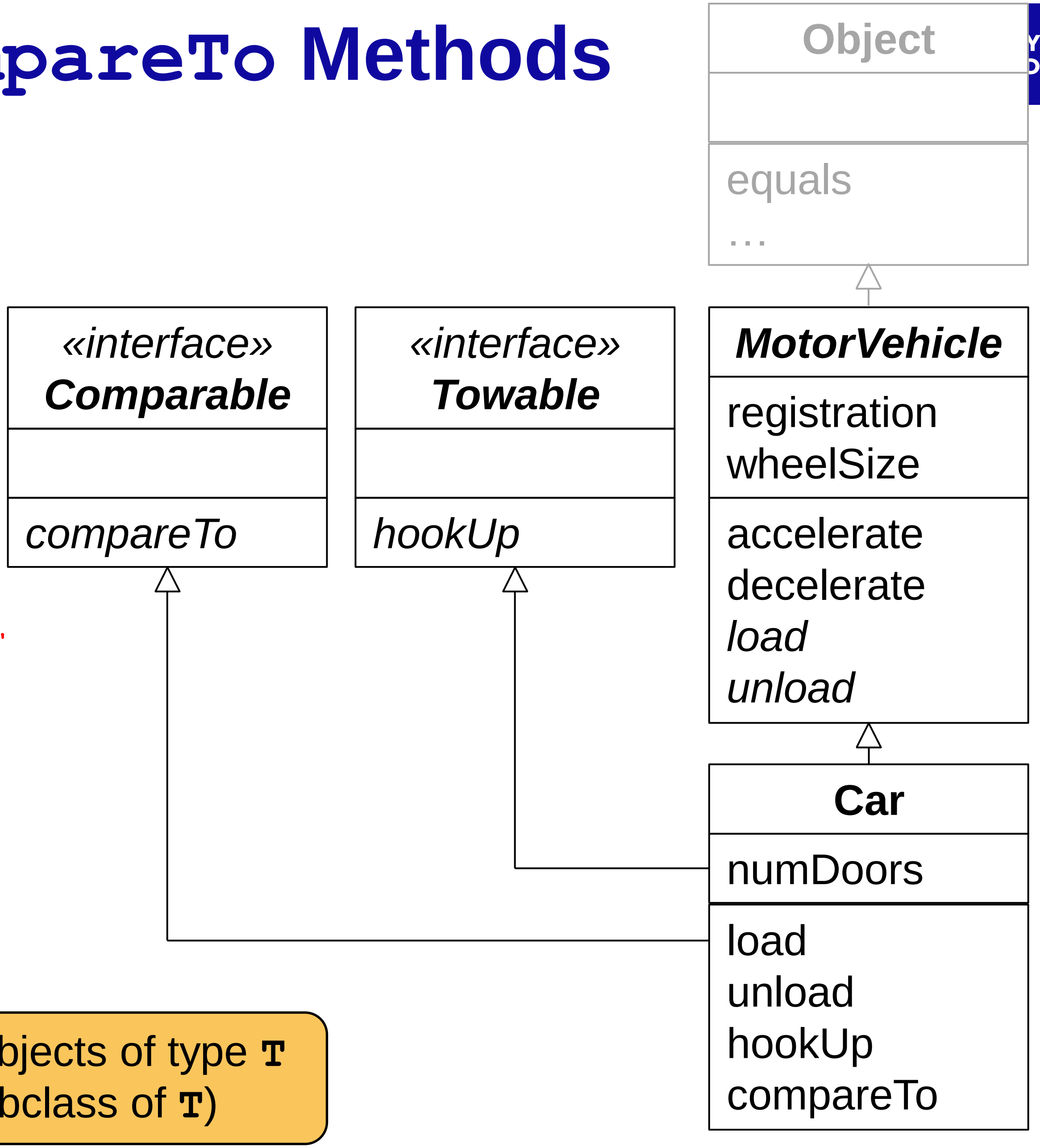
- The **`equals`** method of **`Object`** is implemented as:

```
public class Object {
  ...
  public boolean equals(Object obj) {
    return this == obj;
  }
}
```

- The **`Comparable`** interface is declared as:

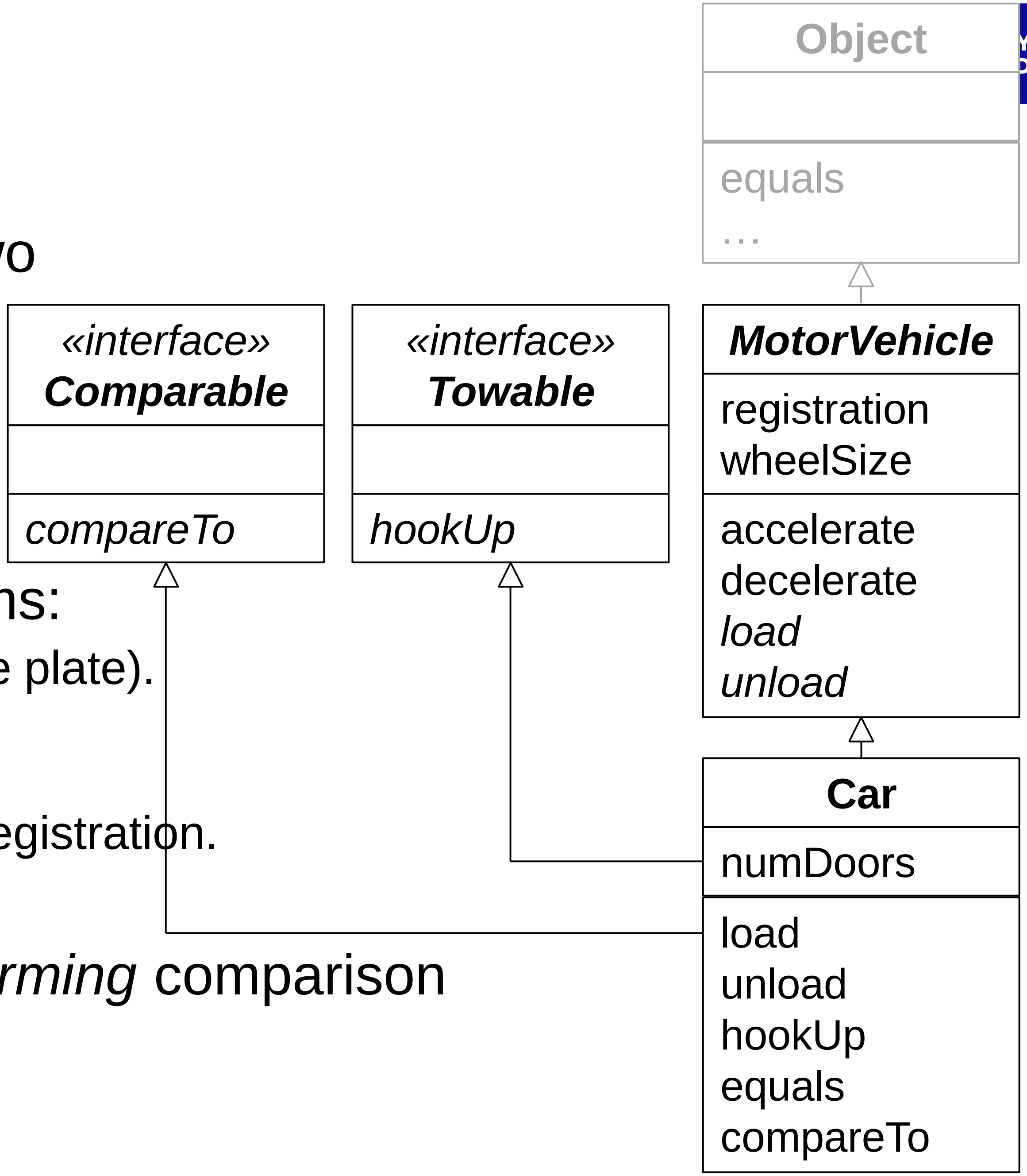
```
public interface Comparable<T> {
  public int compareTo(T o);
}
```

Compare objects of type **`T`** (or a subclass of **`T`**)



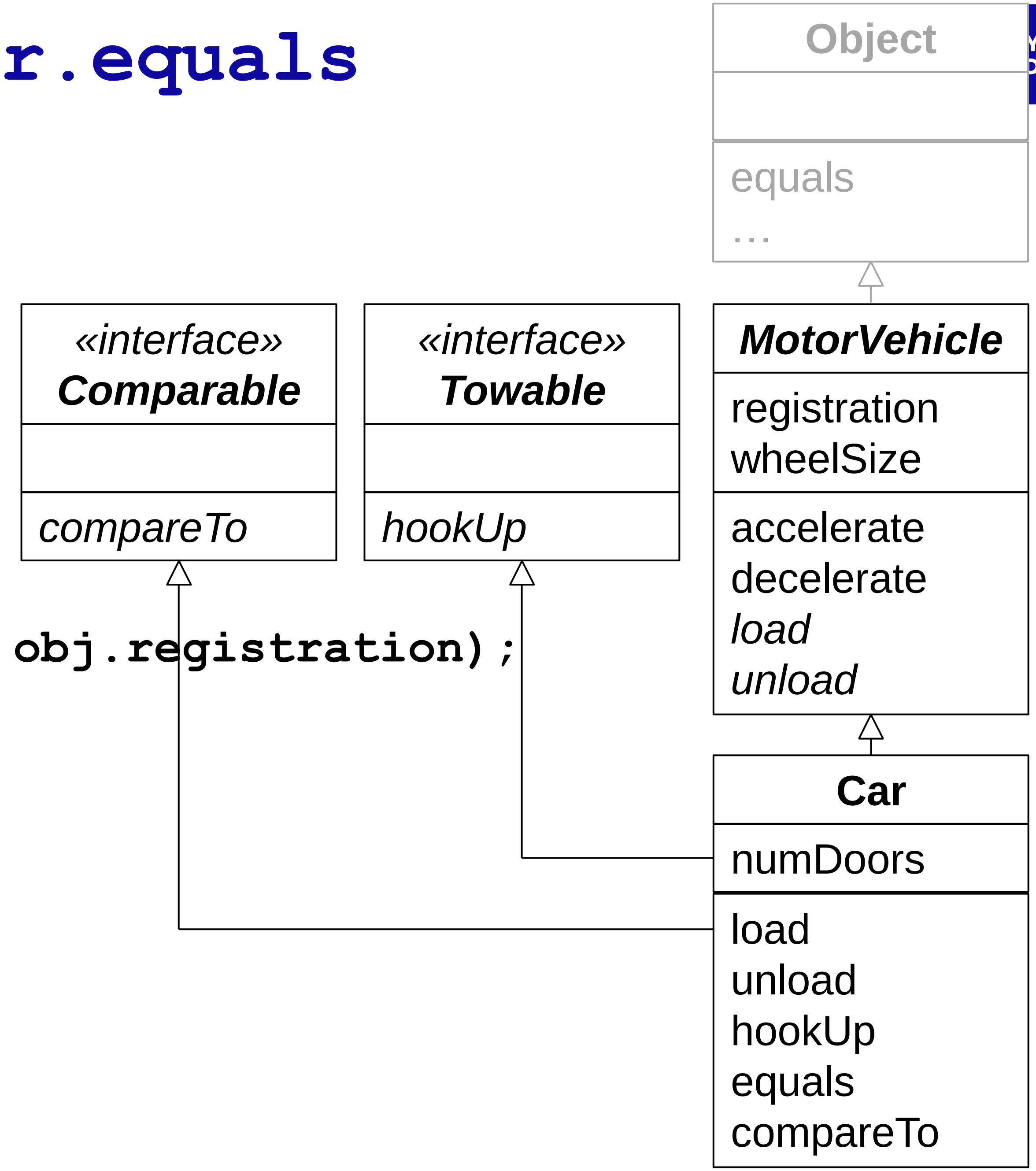
Example: Comparing Cars

- The method **`Object.equals`** checks if two references point to the same instance
  - But we could also define two separate Car objects as equal if their registrations match.
- We can think of different ordering algorithms:
  a) Use alphabetical order of registration (license plate).
  b) Use wheel size.
     If that is identical, use number of doors.
     If that is identical, use alphabetical order of registration.
  c) …
- Goal either way: Decouple algorithm *performing* comparison from algorithms *using* comparison



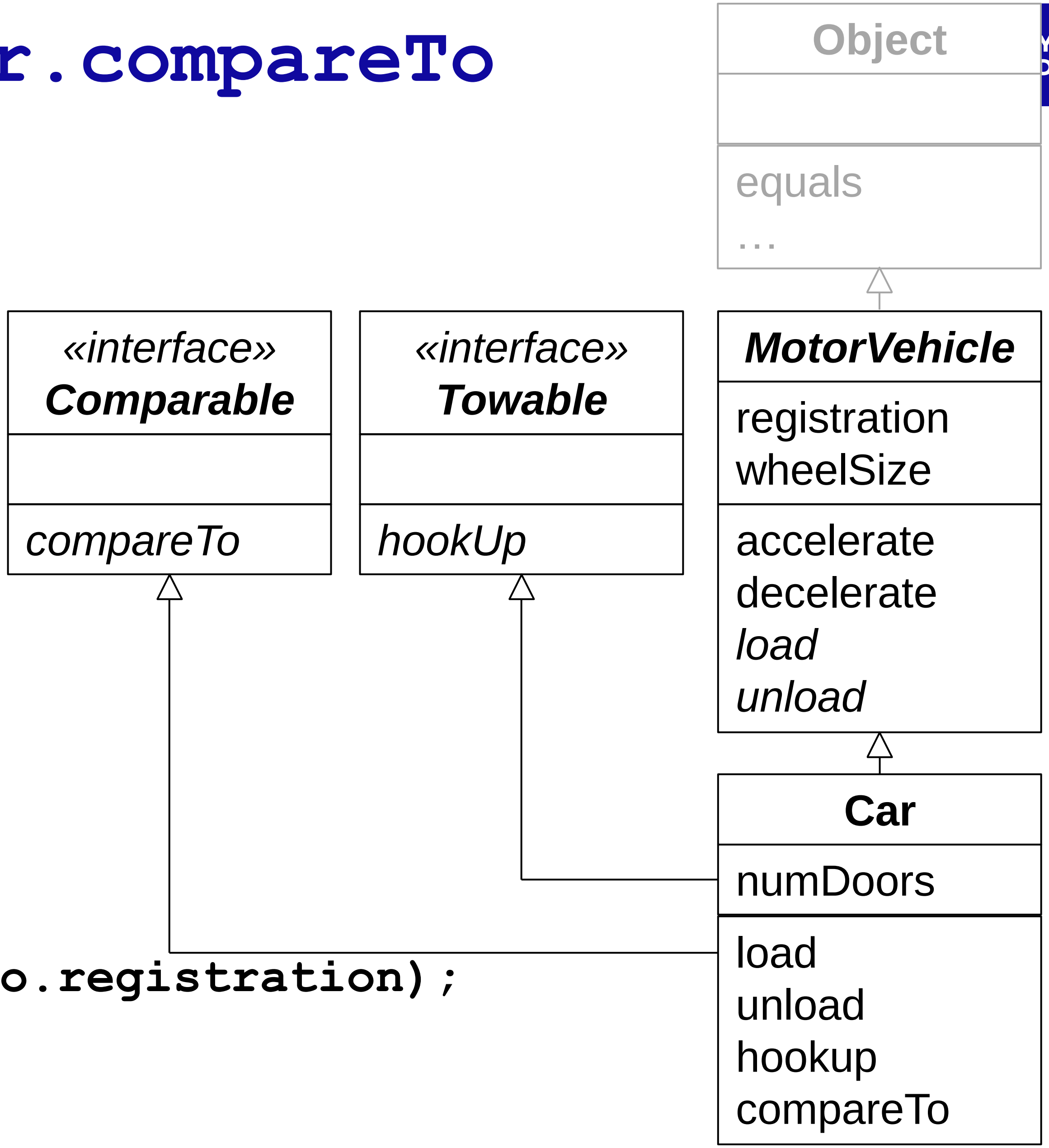
Possible Implementation of `Car.equals`

```
public class Car extends MotorVehicle
                 implements Towable,
                     Comparable<Car> {
  ...
  public boolean equals(Object obj) {
    if (obj instanceof Car) {
      return registration.equals((Car) obj.registration);
    } else {
      return false;
    }
  }
}
```



Possible Implementation of `Car.compareTo`

```
public class Car extends MotorVehicle
                 implements Towable,
                     Comparable<Car> {
  ...
  public int compareTo(Car o) {
    if (wheelSize != o.wheelSize) {
      return wheelSize - o.wheelSize;
    }
    else if (numDoors != o.numDoors) {
      return numDoors - o.numDoors;
    }
    else return registration.compareTo(o.registration);
  }
}
```



Usage of `Comparable` Interface

- Using the **`Comparable`** interface, we can now make comparisons as part of all kinds of algorithms – without making them dependent on the type of object:
  - Example: Sort an array of cars:

    ```
    Car[] cars = ...; Arrays.sort(cars);
    ```

  - Example: Determine the largest element of an array:

    ```
    public Comparable max(Comparable[] set) {
      Comparable largest = set[0];
      for (int i = 1; i < set.size(); i++) {
        if (largest.compareTo(set[i]) < 0)
          largest = set[i];
      }
      return largest;
    }
    ```

➢ The algorithms using **`Comparable`** are independent of the type of compared object (and the algorithm for making the comparison)!

Why is `equals` in a (Mandatory) Superclass but `compareTo` in an (Optional) Interface?

- Wouldn't it make sense to have **`compareTo`** declared in **`Object`** and thereby make any objects comparable?

No, because:

1. For **`equals`**, there is a default implementation that works sensibly in most cases (two object references are equal if they point to the same instance)
   - Only needs to be overridden if distinct instances of a class should be considered equal based on certain attribute values
2. For **`compareTo`**, there is no default rule for establishing any objects' natural order, so this method would have to be abstract in **`Object`**.
   - This would force every single class to provide its own implementation of **`compareTo`**, even if the definition of a natural order is not possible or not necessary for it.

➢ Better to have natural ordering has an optional capability, i.e. an interface.

Usage of (Abstract) Superclasses vs. *«Interface»*s

(Abstract) Superclass

- **Extending a class** is usually understood as an expression of **type** (“a car is a motor vehicle”)
- In Java/C#, subclasses can extend only one (direct) superclass
- Part of the regular generalization / specialization hierarchy of classes
- Helpful for modeling the commonalities and individualities of classes in the same “family” (e.g. **`MotorVehicle`**, **`Car`**, **`Jeep`**…)

«Interface»

- **Implementing an *«interface»*** is often understood as an expression of **capability** (“a car is towable”)
- Classes can implement several *«interface»*s
- Part of separate *«interface»* hierarchy
- Helpful for equipping classes with interfaces for additional capabilities that are independent of their class “heritage” (e.g. **`Comparable`**, **`Serializable`**, **`Cloneable`**…)

Summary: Usage of «*Interface*»s

- Declaration similar to abstract classes, **BUT:**
  - Contain only abstract methods, no implementations at all

- Usage similar to abstract classes, **BUT:** independent of class hierarchy, i.e...
  - Unrelated classes can implement the same «*interface*»
  - A class can implement several «*interface*»s

- Typical semantics:
  - **Superclasses** indicate what a class **primarily is**
  - **«*Interface*»s** indicate what a class **can also do**

In-Class Quiz #8: Classes vs. Interfaces

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

a) A Java class can extend only one _____.

b) A Java class can implement multiple _____.

c) _____ contain only abstract methods.

d) A hierarchy of _____ describes commonalities and differences of entities within the same “family”.

e) _____ describe additional capabilities independent of entities’ “heritage”.

f) _____ cannot inherit declarations from _____.

UNIVERSITY OF ICELAND

Break

Recap: Object-Oriented Analysis, Design & Programming

The object-oriented view permeates all phases of software construction:

- ✓ **Object-Oriented Analysis (OOA) → yields the Domain Model**
  - Identifying classes, their relationships and behavior in the reality of the <u>application domain</u>
    - Exploring the application domain, identifying business processes, identifying the objects handled by those processes, deriving classes from those objects, defining collaboration between classes…
- ➢ **Object-Oriented Design (OOD) → yields the Design Model**
  - Refining the models created during analysis to reflect <u>technical (implementation) needs</u>
    - Solutions for user interaction, data storage, component distribution, parallel/asynchronous execution; choice of suitable algorithms; optimization of data structures…
- **Object-Oriented Programming (OOP) → yields the Implementation**
  - Expressing the models created during design in a programming language (incl. refinement)
    - Adaptation to language specifics, use of libraries, additional low-level technical objects (e.g. for exception handling), conversion of types, implementation of algorithms…

UML Sequence Diagrams

see also: Learning UML 2.0, Chapter 7

Modeling Class Interactions with Sequence Diagrams

- **UML class diagrams** show the static structure of a system
  - Internal structure of classes (attributes and methods)
  - Overall structure of the system (class relationships)

- **UML sequence diagrams** show the dynamic behavior of a system
  - Communication between objects (via method calls)
  - Creation and destruction of objects

- Focus of sequence diagrams:
  - Model who exchanges messages with (i.e. calls methods of) whom in which order
  - Especially helpful to illustrate individual steps in sequential / parallel / nested communication

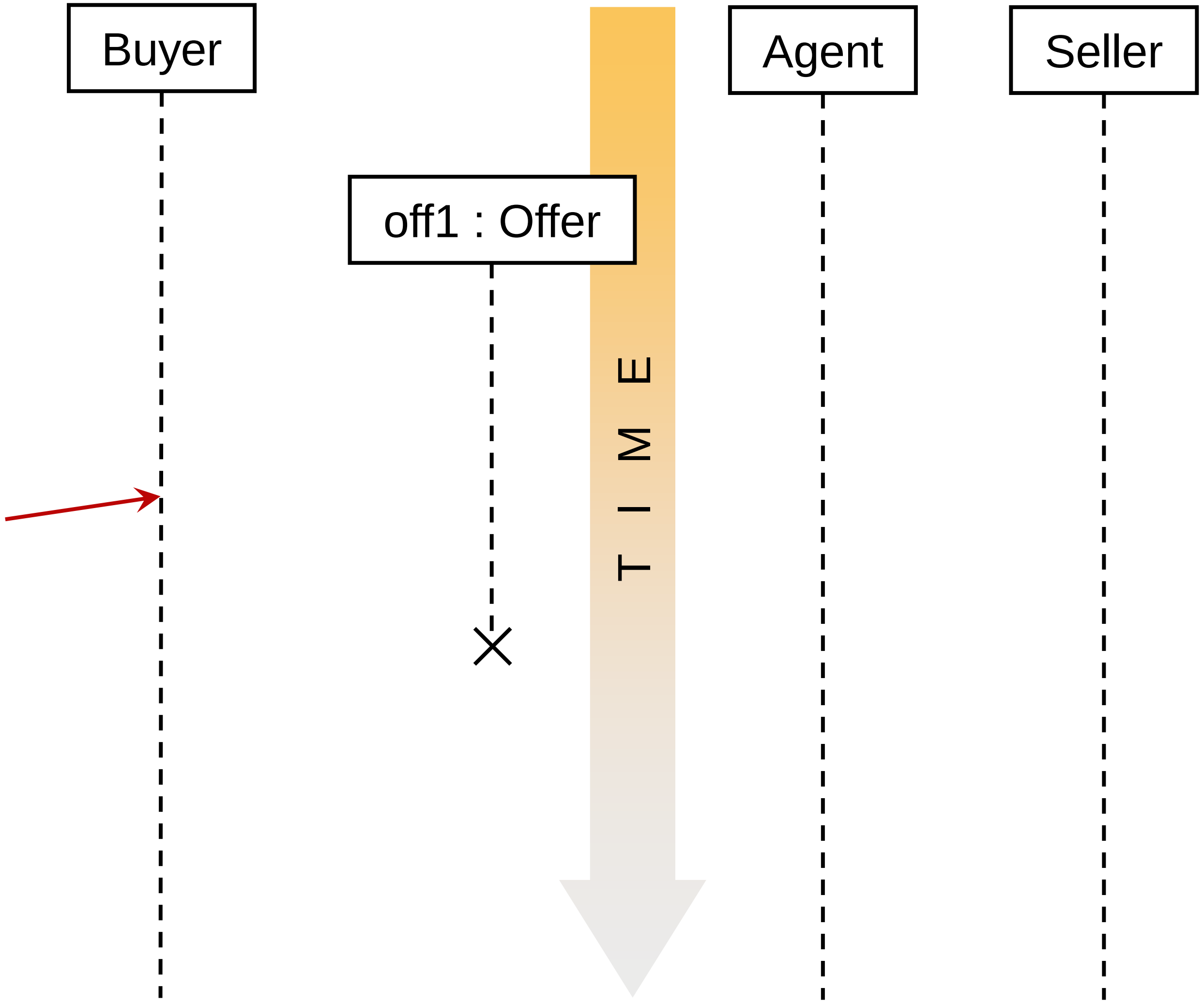
Participants

- Participants can be objects, larger system parts, or other actors
  - Always arranged horizontally
- Naming convention: “Name : Type”
  - Name designates an actor, object, or system part (component)
  - Type is a class (usually also to be found in an accompanying class diagram)
  - Either part is optional
    - In a business-oriented diagram, we may just use names (roles) of actors
    - In a more technical diagram, we may just use labels of classes

Lifelines

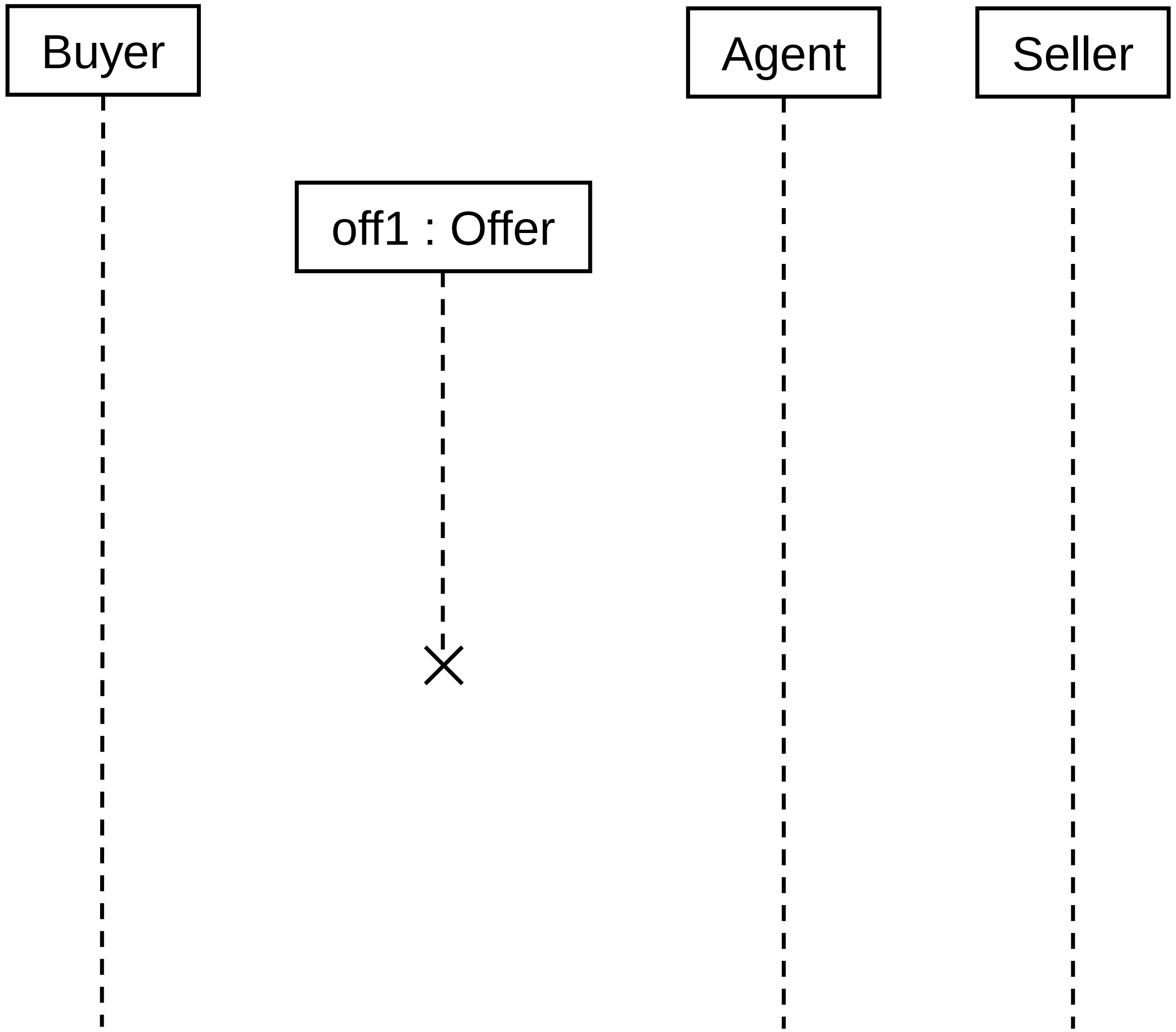
- Time proceeds downward from top
  - Only the order of elements matters
  - Distance between elements has no meaning in a sequence diagram
    - ➢ No precise indication of elapsed time
    - ➢ Use UML timing diagram if needed

- Each participant has a vertical lifeline that indicates its duration of existence
- Participants can be placed lower in the diagram if they are created later
- An X marks the destruction of a participant



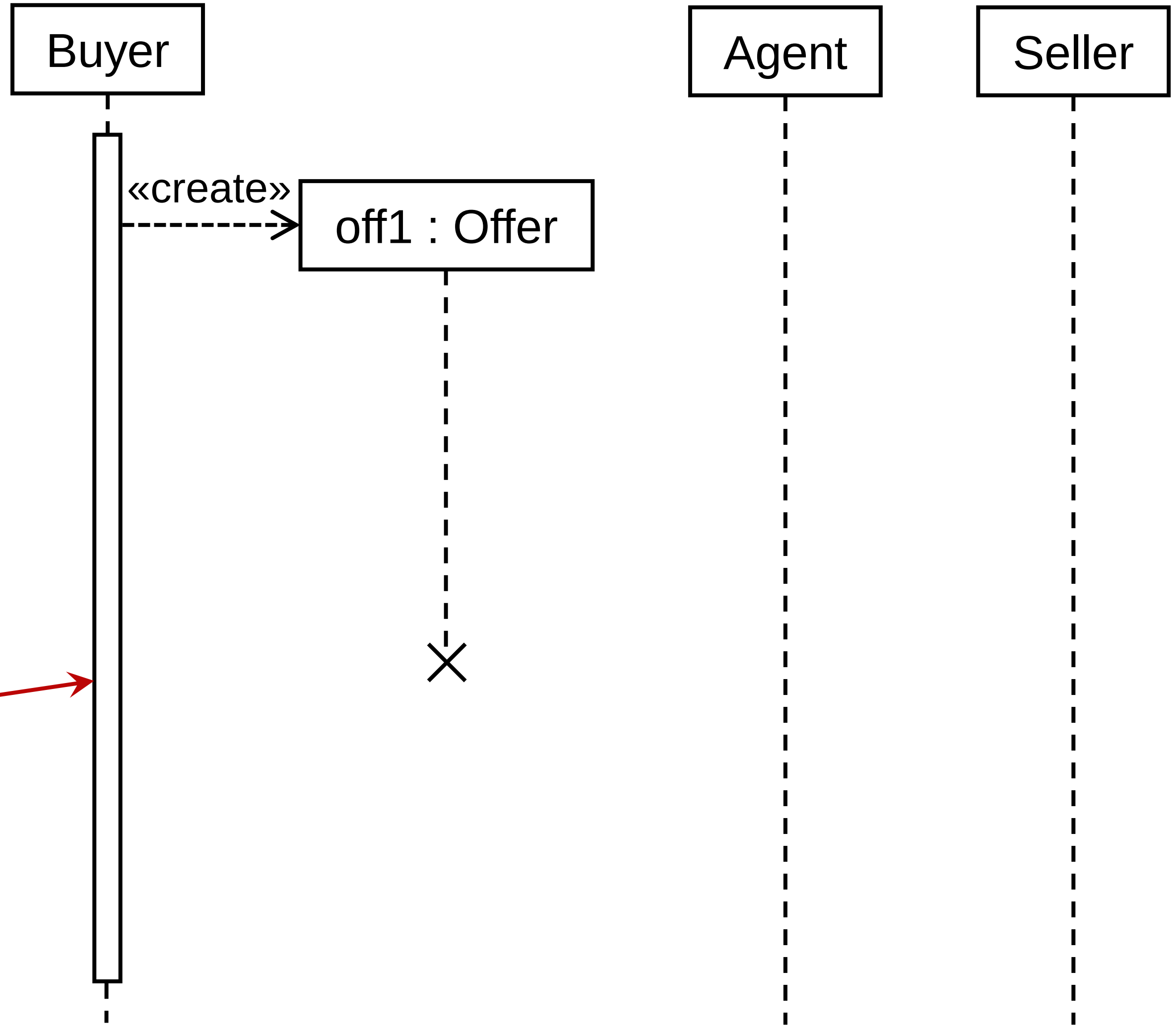
Messages

- Participants exchange messages
- Messages are symbolized as arrows
  - In a domain model, usually labeled according to content of message
    - Note: This is often some activity written from the message *sender's* perspective
  - **In a design model, always labeled with the called method's name**
    - **Note: This is necessarily written from the message *receiver's* perspective**
  - ➢ Caution when interpreting/implementing a domain model in a design context
- Parameters and return types may be specified (as in class diagrams)



Participant Creation and Destruction Messages; Activation Bars

- A «create» message to a participant's box indicates its creation in the world described by the diagram.
  - Java interpretation: Class instantiation
- A «destroy» message to a participant's lifeline renders it unavailable from that point onwards.
  - Java interpretation: Not necessary, as garbage collector does this automatically
- An activation bar indicates the period in which a participant is working (or waiting to continue to work)
  - Java interpretation: Indicates that the method is currently being executed

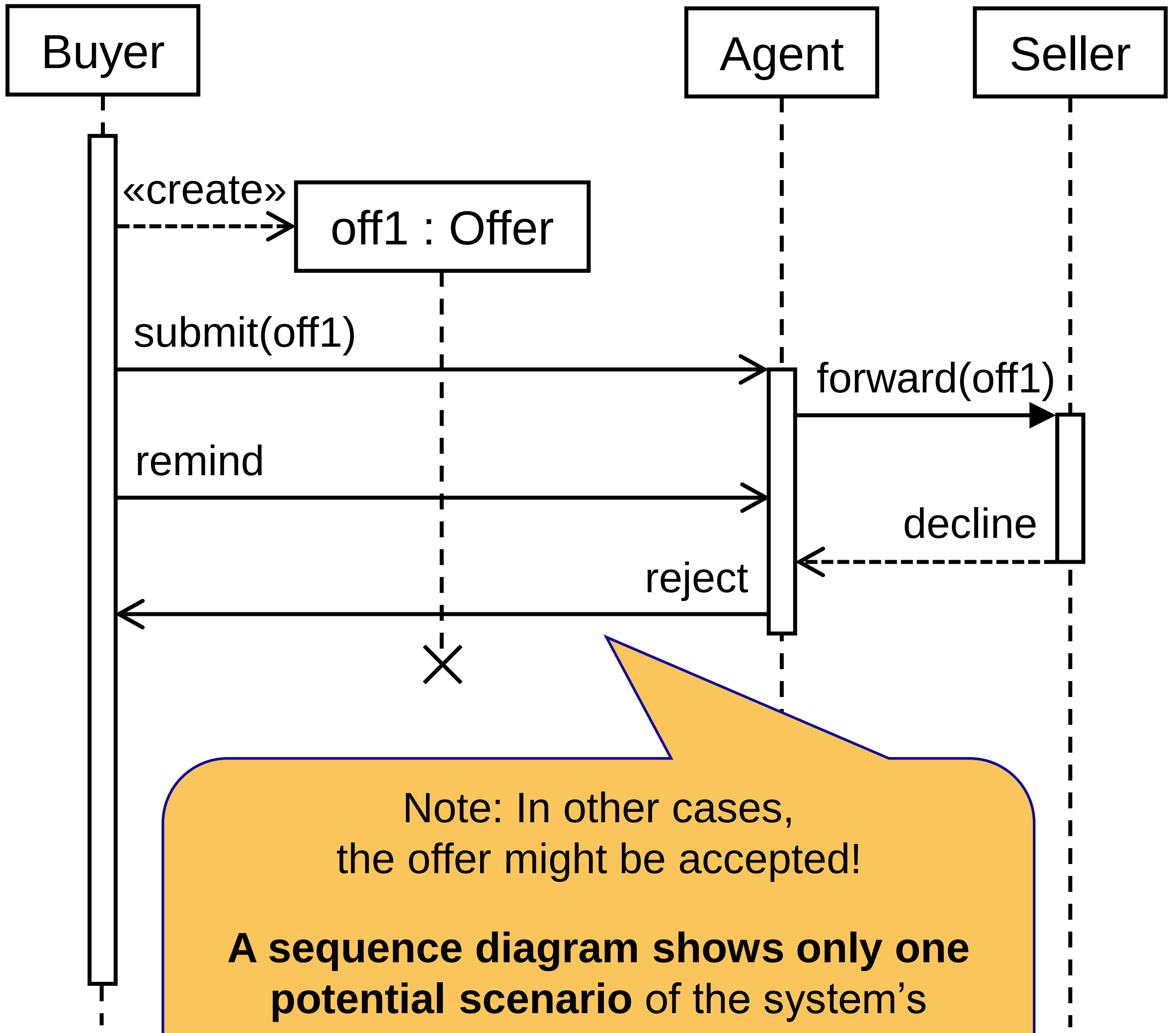


Message Types

Note: Different arrow symbols have different meanings – make sure you use the right one!

Example: Domain model sequence diagram

- **Synchronous message**
  - The message sender waits until the message receiver responds to the invocation.
    - Java interpretation: Calling a method and waiting for it to terminate before control returns to the sender
- **Return message**
  - Control flow returns after invocation of synchronous message
- **Asynchronous message**
  - The message sender does not wait for the receiver's response but remains in control after sending.
    - Java interpretation: Calling a method in a parallel thread



Note: In other cases, the offer might be accepted!

A sequence diagram shows only one potential scenario of the system's behavior. Several diagrams may be needed to show all variants.

Example: Design Model Sequence Diagram

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

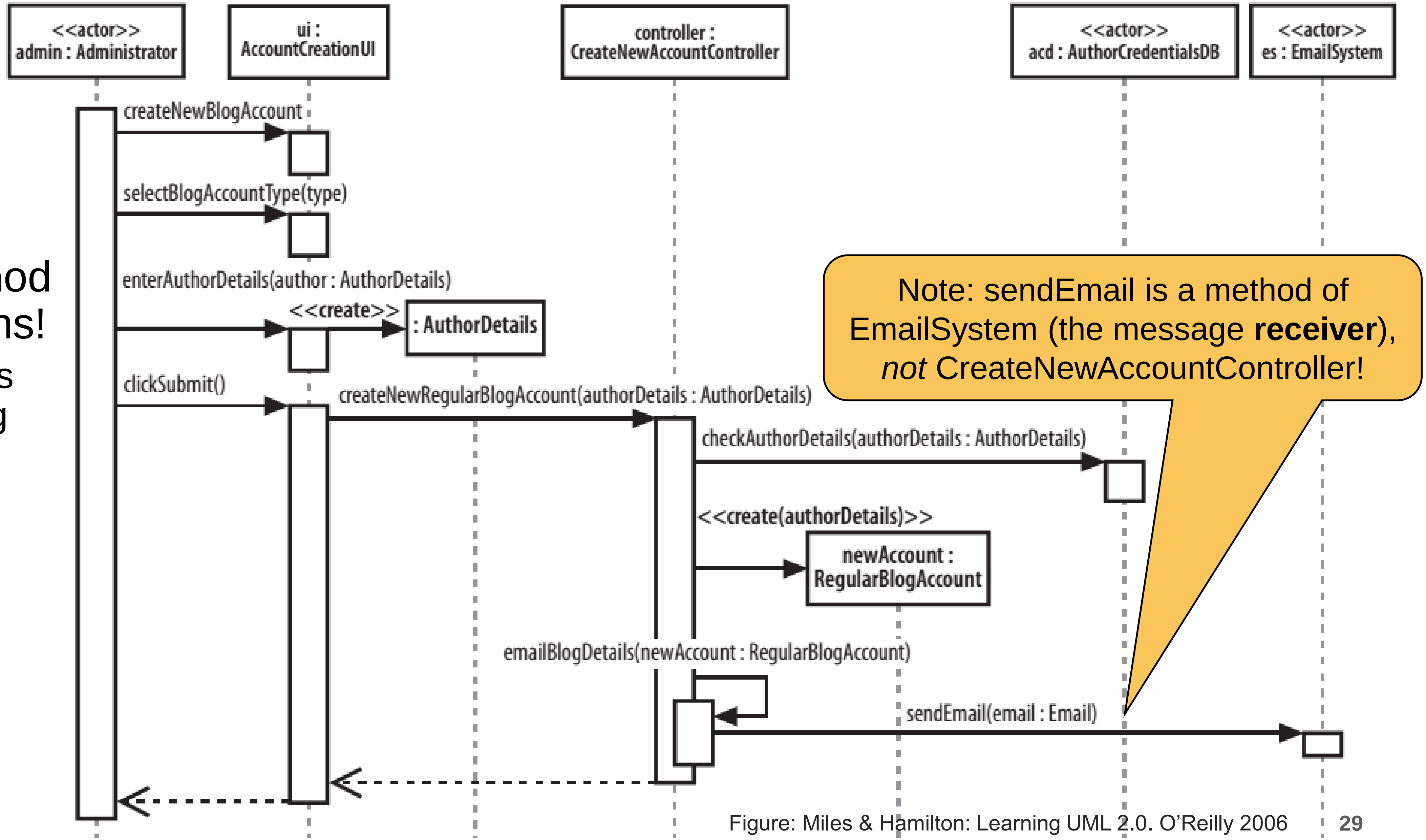
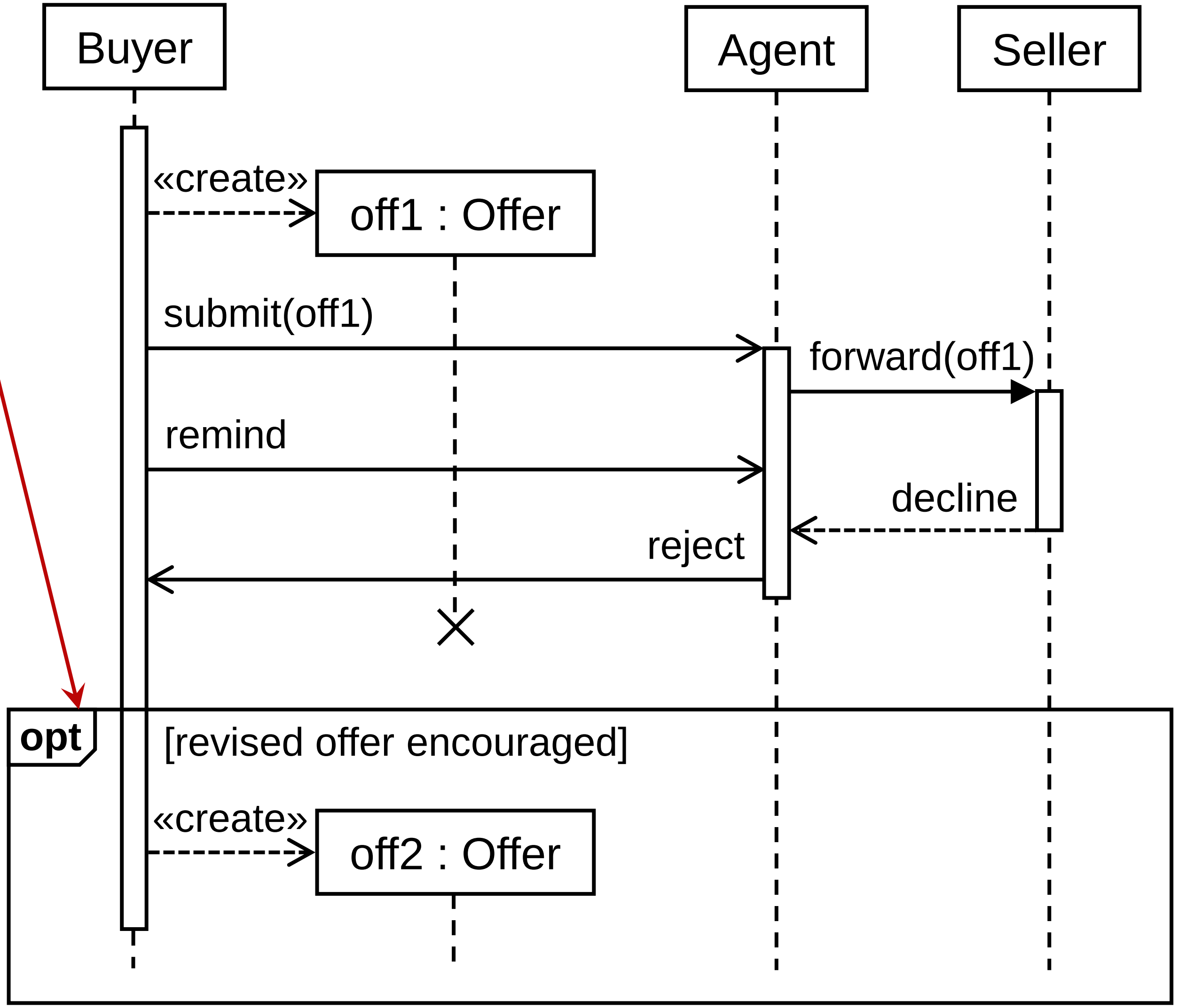


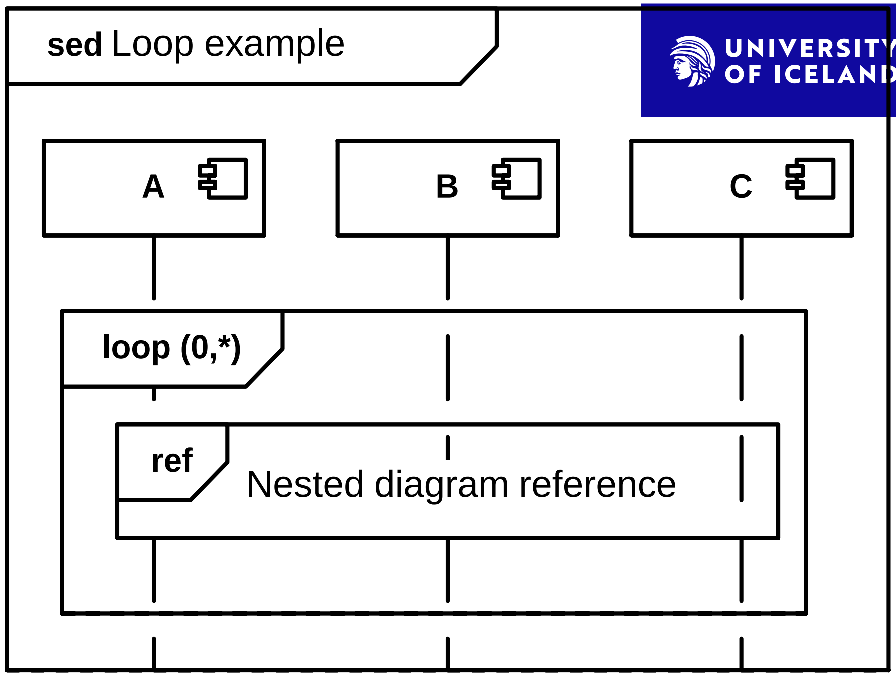
Figure: Miles & Hamilton: Learning UML 2.0. O'Reilly 2006

Sequence Fragments

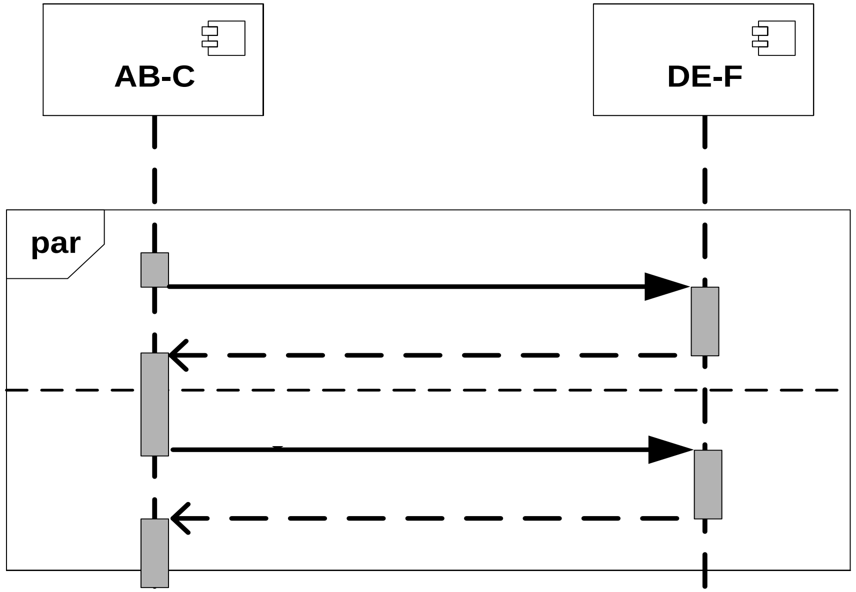
- More complex interactions can be described with sequence fragments
  - Contents of fragment boxes are executed based on certain criteria, depending on fragment type, e.g.
  - **opt** [*guard*]
    - executed only when *guard* condition true
  - **loop** (*min*, *max*) [*guard*]
    - executed at least *min* and at most *max* times while *guard* condition is true
  - **ref**
    - refers to sub-diagram to be included here
- Use sparingly – can make a sequence diagram very hard to read
  - Use only for small, local control flow
  - For distinct scenarios, use indiv. models

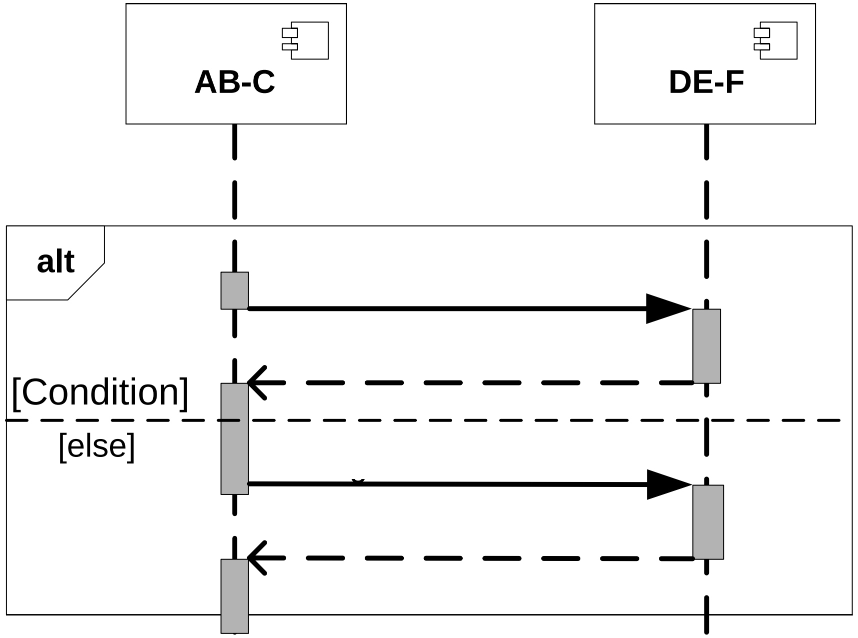


UML Sequence Diagram Fragments

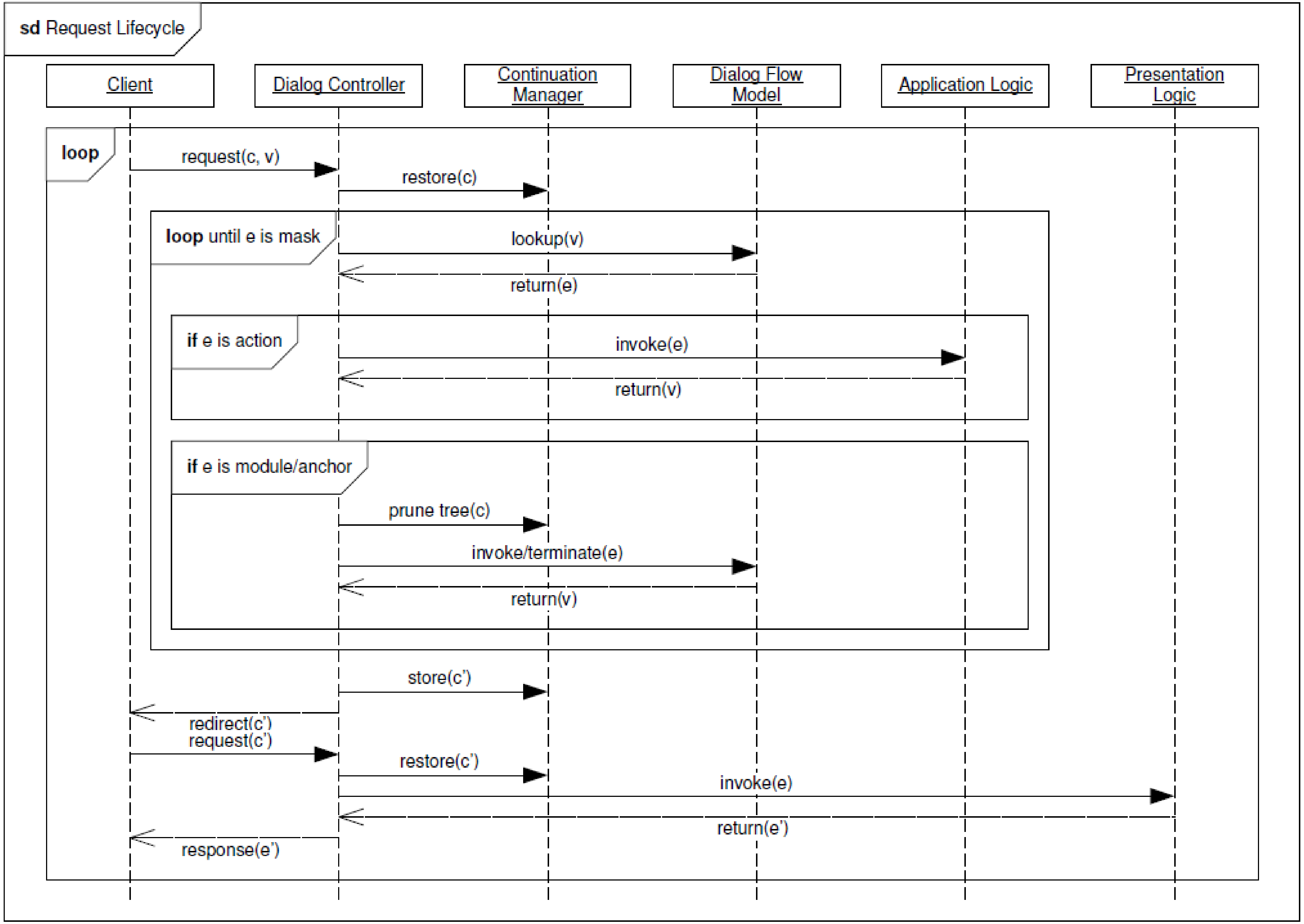


- **par:** Parallel execution
- **alt:** Alternative execution
- **loop:** Looped execution
- **ref:** Nested execution



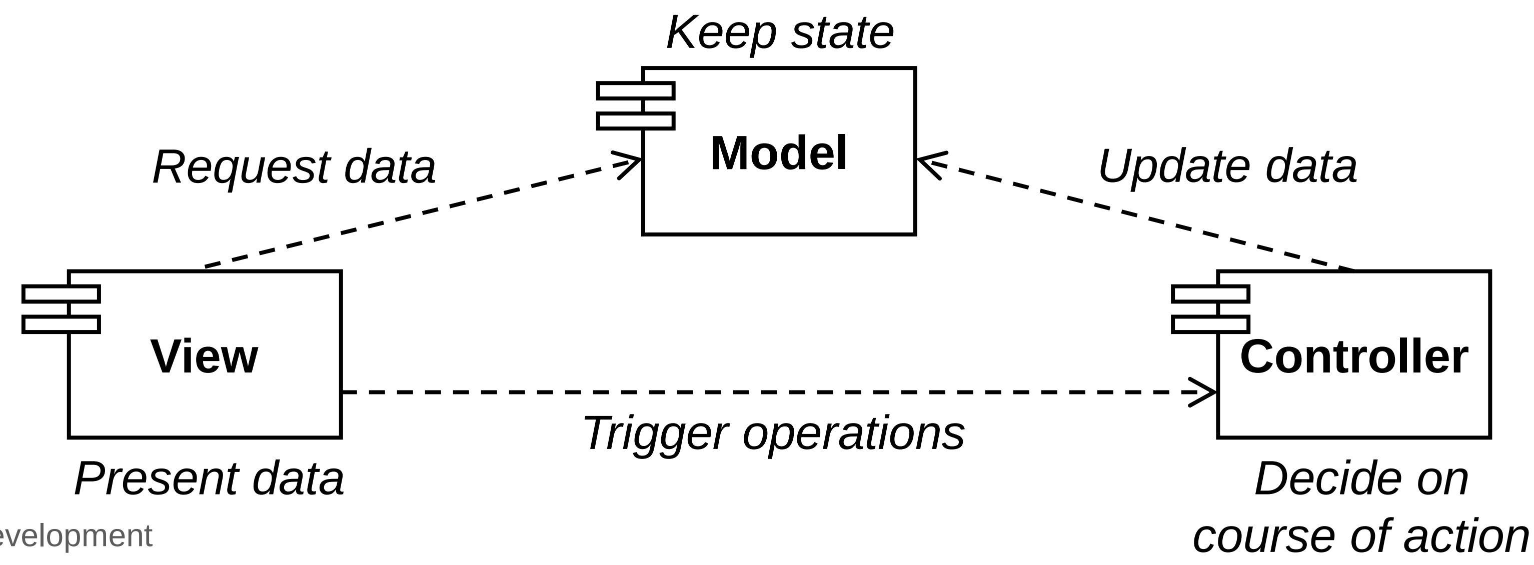


Example: A More Complex Sequence Diagram



Design Pattern: Model-View-Controller (MVC)

- 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



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?

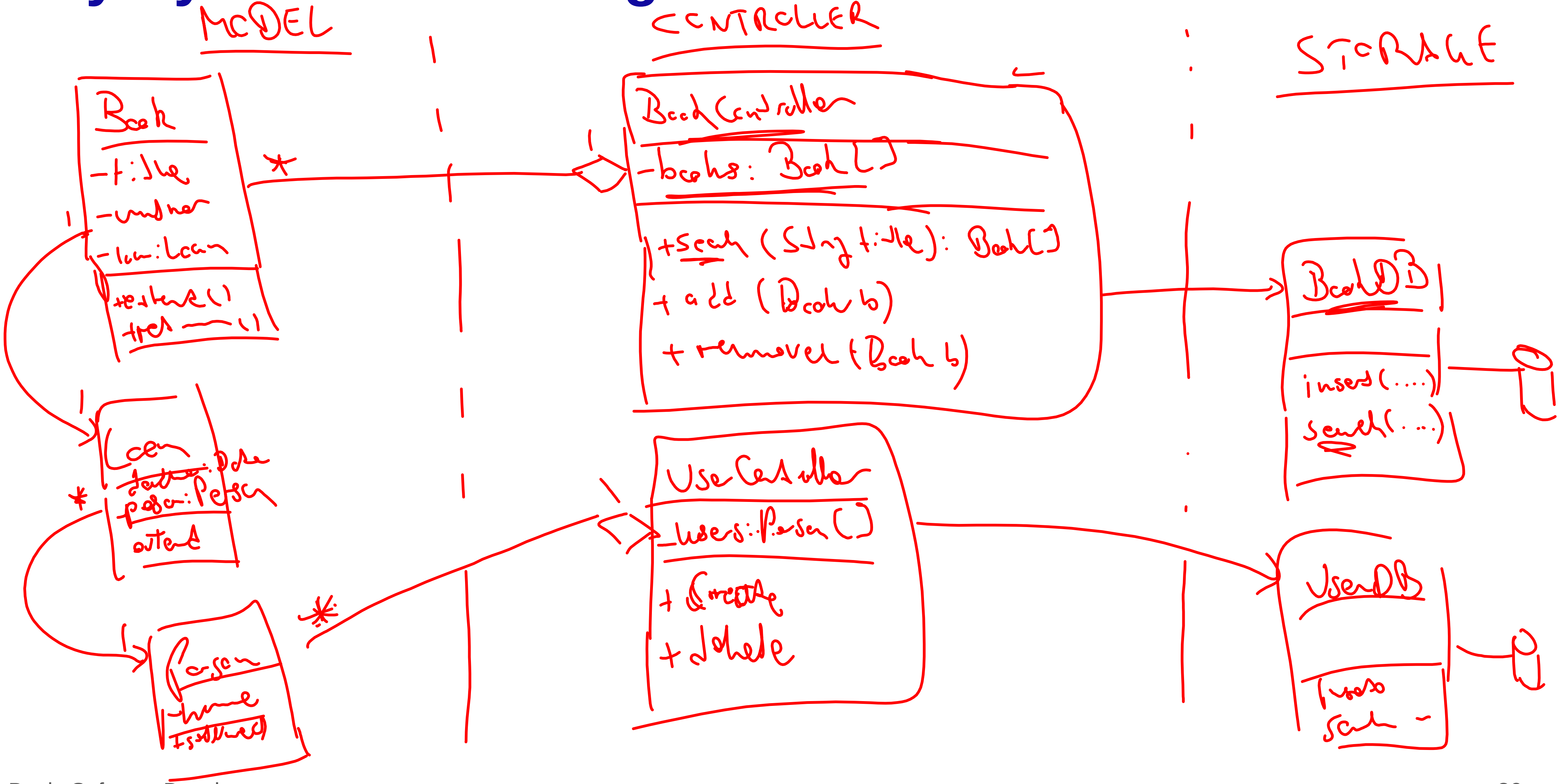
Team Assignment #3: Design Model

UNIVERSITY OF ICELAND

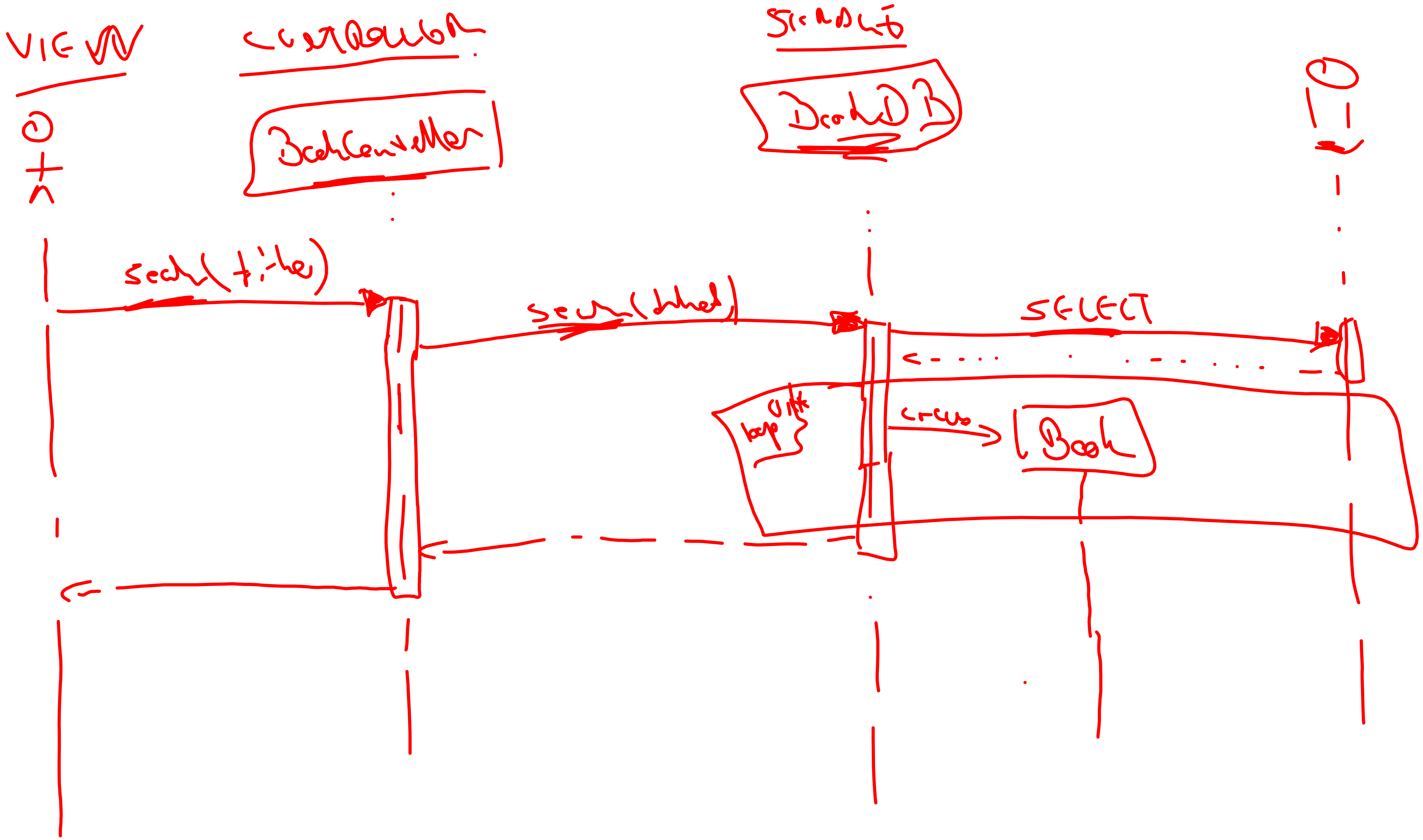
- **Grading criteria**
  - Class diagram is complete design model (with regard to classes, methods, attributes, Model/Control/Storage layers) (25%)
  - Relationships between classes, and placement of attributes and methods in classes, is plausible (25%)
  - Sequence diagrams show plausible roundtrip 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

Design Model Example:
Library System Class Diagram



Design Model Example: Library System Sequence Diagram



Thank you!

book@hi.is