

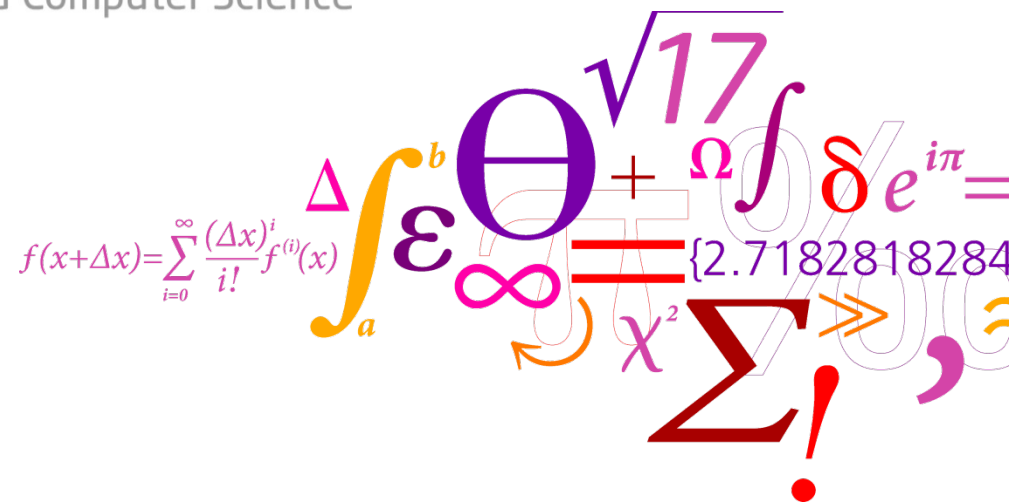
# Software Engineering 2

## A practical course in software engineering

Ekkart Kindler

DTU Compute

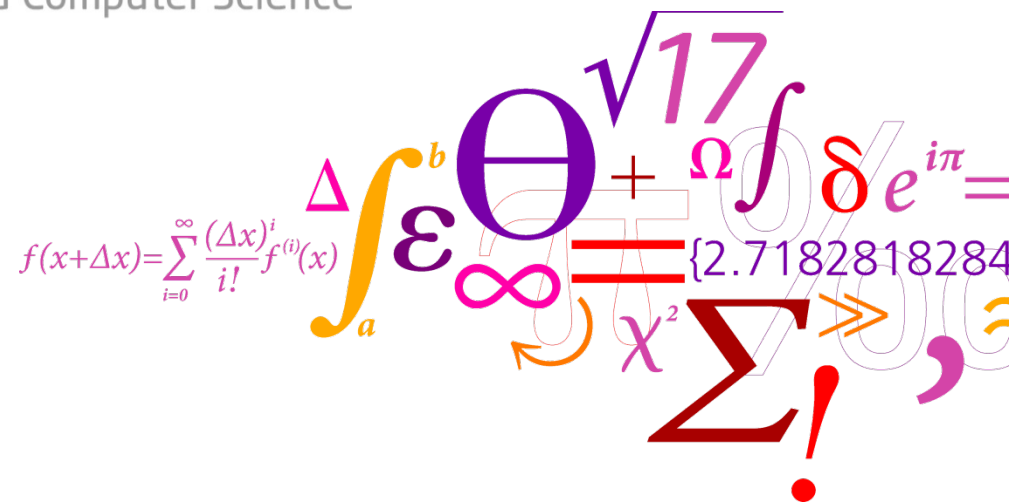
Department of Applied Mathematics and Computer Science



# V. Modelling Software

**DTU Compute**

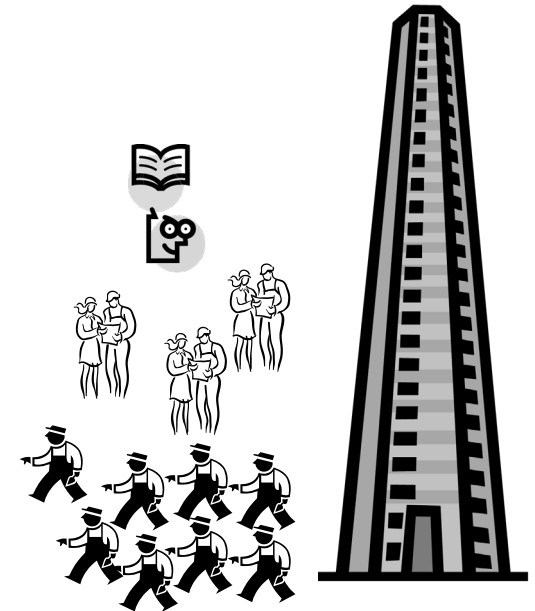
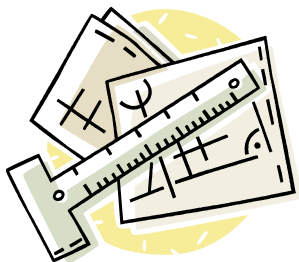
Department of Applied Mathematics and Computer Science



A collage of mathematical symbols and formulas. It includes the Taylor series expansion  $f(x+\Delta x) = \sum_{i=0}^{\infty} \frac{(\Delta x)^i}{i!} f^{(i)}(x)$ , the definite integral  $\int_a^b \epsilon$ , the Greek letter  $\Theta$ , the square root  $\sqrt{17}$ , the Greek letter  $\Omega$ , the Dirac delta  $\delta$ , the exponential function  $e^{i\pi}$ , the infinity symbol  $\infty$ , the chi-squared symbol  $\chi^2$ , the summation symbol  $\Sigma$ , and the number  $\{2.7182818284\}$ .

# 1. Motivation

- Which models are there?
  - What are “software models”?
  - What are they good for?
  - Why do WE need them?
- 
- What is software?
  - What is a model?



## **Modell** [*lat.-vulgärlat.-it.*] *das; -s, -e*:

...

7. die vereinfachte Darstellung der Funktion eines Gegenstands od. des Ablaufs eines Sachverhalts, die eine Untersuchung od. Erforschung erleichtert od. erst möglich macht.

...

[nach Duden: Das Fremdwörterbuch, 1990].

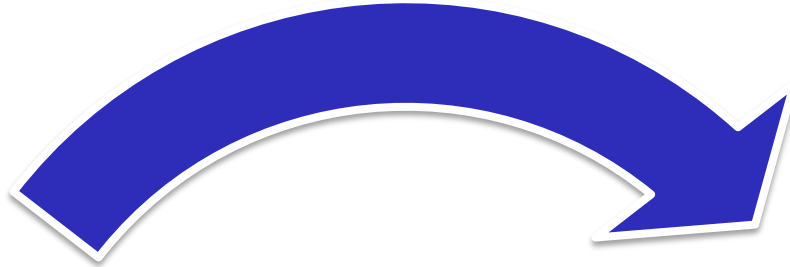
## **Modell** [*lat.-vulgärlat.-it.*] *das; -s, -e*:

...

7. the simplified description of the function, purpose, or process of something; it enables us investigating and analysing this thing.

...

[nach Duden: Das Fremdwörterbuch, 1990].



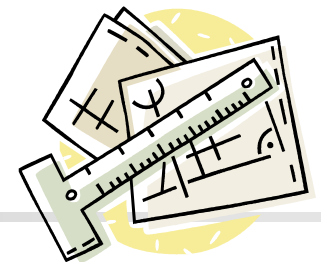
## WHAT

## HOW

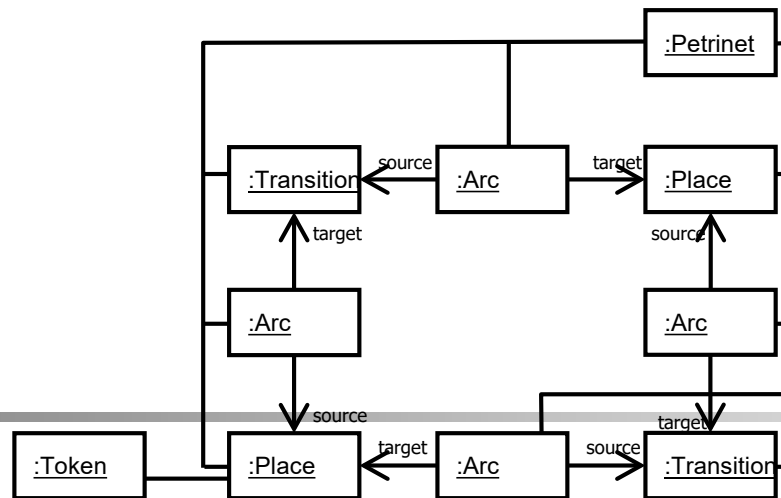
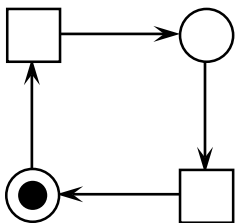
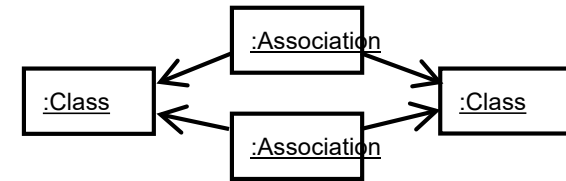
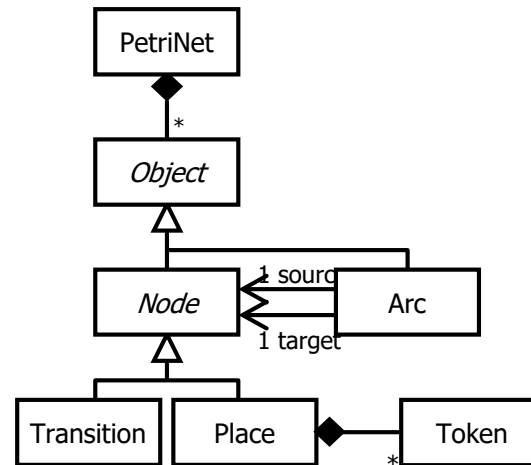
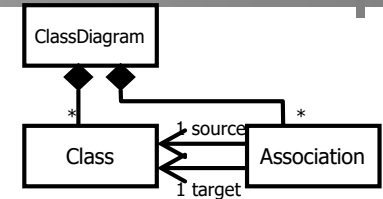


In software engineering, we use models for both, the WHAT and the HOW. We even use the same notation (UML)! But, it is important to keep the different purposes of models apart.

- better understanding the „thing“ under investigation (or development) → help building “your mental model” of the “thing”
- communication
  - on the appropriate level of abstraction
  - with different kinds of people
  - from different angles
- abstraction / composition
- analysis and verification
  - consistency, completeness, correctness, performance, risks, effort, ...
- code generation (cf. L01)



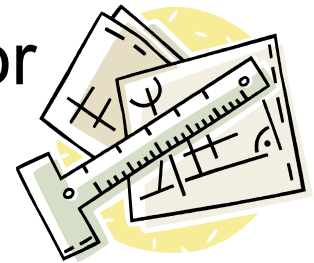
# Reminder (cf. L01)





- „traditionally“: More or less automatic:
  - Sketches of ideas
  - Forward engineering
  - Reverse engineering
  - Reengineering
- Model Driven Architecture (MDA)
  - Generating (at least part of) the software from models
  - Models ARE the software (or a part of it)
- Models encoded in the software (e.g. JPA)

**Initially:** Informal sketches of software for discussion, for better understanding or for communicating an idea



**Later:** Standardized (graphical) notations (UML)

From these diagrams, the program code was produced (mostly) manually!

Forward engineering

- Since software is often not well-documented, it became necessary to retrieve or to extract the essential idea of the software from its code

Reverse engineering

- These models are used to better understand the existing **software**, and to change the software based on this understanding

Reengineering = Reverse +  
Forward engineering

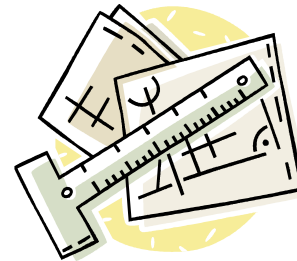
- Some reverse and forward engineering tasks could be automated (mainly structural parts)
- Changes made in the models obtained by reverse engineering can (sometimes) be automatically transferred back into the original code

Roundtrip engineering

## Starting point:

- Software cannot be used in isolation
- It interacts with other software
- In most cases, developers must extend existing software or integrate their software to existing one
- Existing software is often not documented (or at least not documented well)

- Before you can (use,) change or extend software, we need to understand it



- **Reverse engineering** is the process that, for an existing software system, tracks down and retrieves (“mines”) its underlying ideas and concepts and documents them in form of models
- The development process is run in the reverse direction (reverse engineering)

- In the ideal case, the result of **reverse engineering** would be a specification of the software system
- Very important: abstraction and focus on the essentials

Is it possible to “mine” the ideas and to capture them in models at all?



- Tools can support reverse engineering
- But, they cannot fully relieve an engineer of the burden of abstraction and focus!

**This is the task of an engineer!**

- Moreover, many of today's tools come up with wrong or incomplete results, which need to be corrected or amended by hand.

```
public interface Moveable {
    public void move();
}

public abstract class Element {
    ...
}

public class Track extends Element {
    private Track next;
    private Track prev;
    public Track getNext() {
        return this.next;
    }
    public void setNext(Track value) {
        if (this.next != value) {
            if (this.next != null) {
                Track oldValue = this.next;
                this.next = null;
                oldValue.setPrev (null);
            }
            this.next = value;
            if (value != null) {
                value.setPrev (this);
            }
        }
    }
    public Track getPrev() {
        return this.prev;
    }
    public void setPrev(Track value) {
        if (this.prev != value) {
            if (this.prev != null) {
                Track oldValue = this.prev;
                this.prev = null;
                oldValue.setNext (null);
            }
            this.prev = value;
            if (value != null) {
                value.setNext (this);
            }
        }
    }
}
```

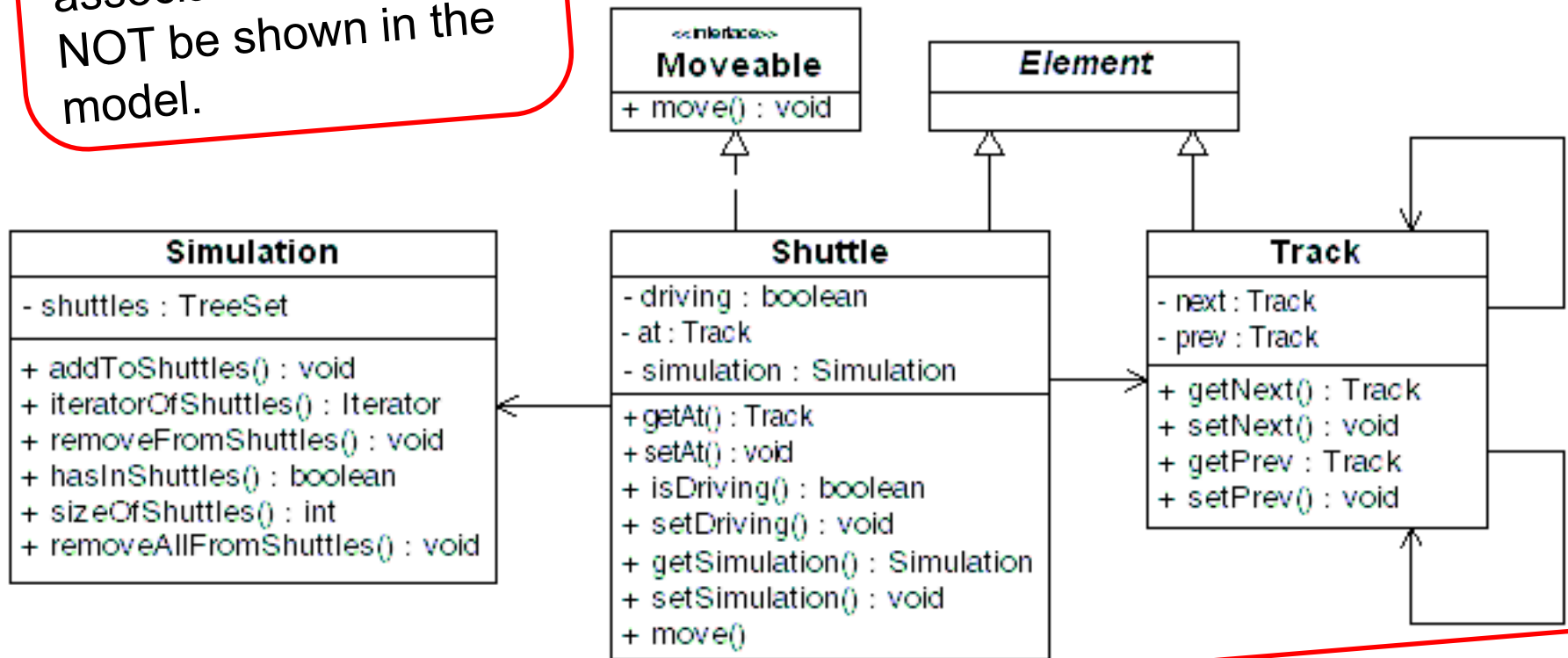
```
public class Shuttle extends Element implements Moveable {
    private boolean driving;
    private Track at;
    private Simulation simulation;
    public Track getAt() {
        return this.at;
    }
    public void setAt(Track value) {
        if ((this.at == null && value != null) ||
            (this.at != null && !this.at.equals(value))) {
            this.at = value;
        }
    }
    public boolean isDriving() {
        return this.driving;
    }
    public void setDriving(boolean value) {
        this.driving = value;
    }
    public Simulation getSimulation() {
        return this.simulation;
    }
    public void setSimulation(Simulation value) {
        if (this.simulation != value) {
            if (this.simulation != null) {
                Simulation oldValue = this.simulation;
                this.simulation = null;
                oldValue.removeFromShuttles (this);
            }
            this.simulation = value;
            if (value != null) {
                value.addToShuttles (this);
            }
        }
    }
    public void move() {
        ...
    }
}
```

```
public class Simulation {

    private TreeSet shuttles = new TreeSet();
    public void addToShuttles(Shuttle value) {
        if (value != null) {
            boolean changed = this.shuttles.add (value);
            if (changed) {
                value.setSimulation (this);
            }
        }
    }
    public Iterator iteratorOfShuttles() {
        return this.shuttles.iterator ();
    }
    public void removeFromShuttles(Shuttle value) {
        if (value != null) {
            boolean changed = this.shuttles.remove
(value);
            if (changed) {
                value.setSimulation (null);
            }
        }
    }
    public boolean hasInShuttles(Shuttle value) {
...
    }
    public int sizeofShuttles() {
        ...
    }
    public void removeAllFromShuttles() {
        ...
    }
}
```

# Example: Result (tool)

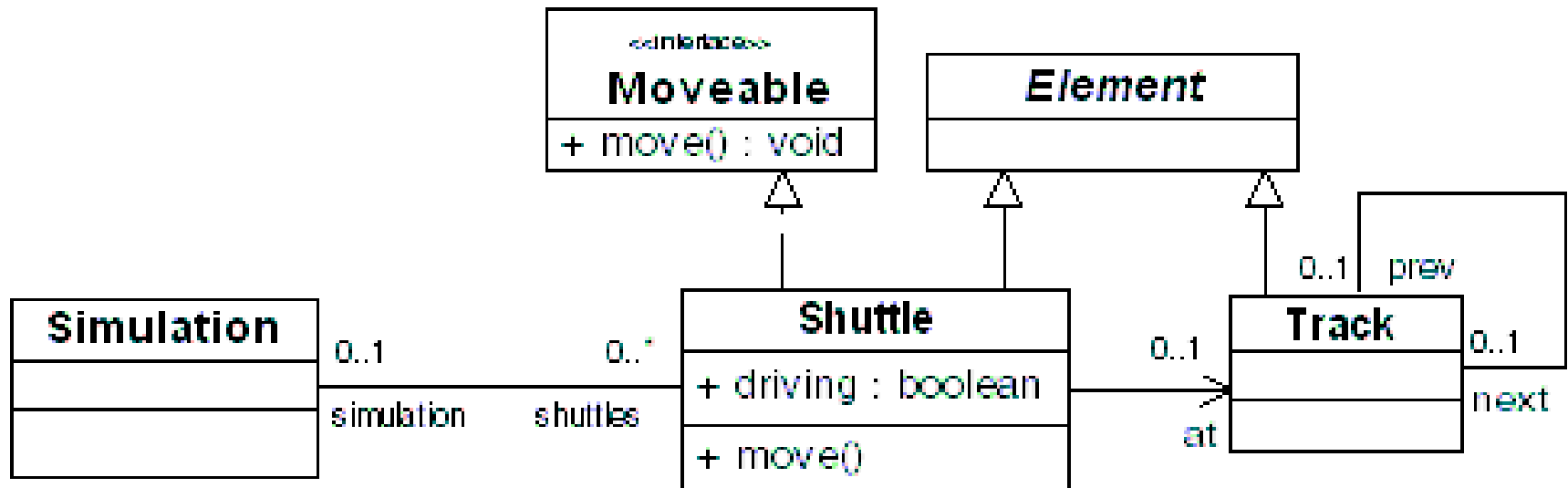
**NB:** “Getters and setter methods” for class attributes and associations should NOT be shown in the model.



**NB:** Important information missing:

- Cardinalities
- Names for roles

# Example: Result (manual)



**Abstraction**

- Much information missing (wrong)
- Redundant information
- Typically, the models cover the structure only; behaviour models missing
- The results that tools come up with are on a very low level of abstraction (class diagrams or very basic design patterns)
- → Still very helpful (and current research improves the situation)

- Initially, we use models for discussion:
  - domain
  - architecture (see project discussions)
- Later, we will use models for written documentation
  - e.g. use case diagrams or activity diagrams (what)
  - e.g. component diagrams and sequence diagrams (how)

- In some technologies (e.g. JPA), the models are represented as Java code with some tags
- The Java classes represent the entities of a domain
- By annotations, they can automatically be mapped to a database
- Even though this is code, they should be considered as "WHAT"

Some tags of JPA, however, define how entities are mapped to a database schema: this is "HOW".



**Today:** We can generate parts of the code from the UML class diagrams automatically (MDA, MDE, **EMF**, EMFT/GMF)

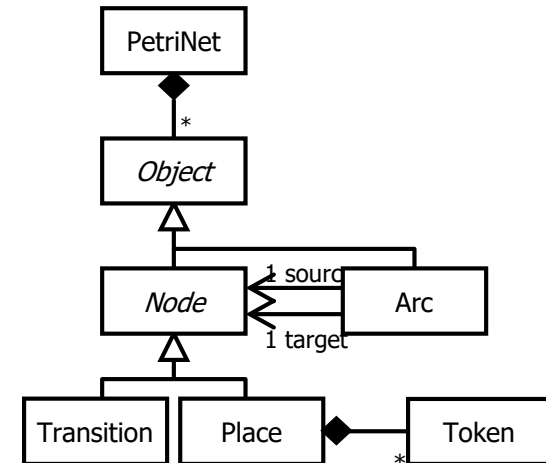
- Class diagrams → Java class stubs with standard access methods (see RE example)
- Implementation of standard behaviour:
  - Loading and saving models
  - Accessing and modifying the models
  - Editors and graphical user interfaces
- The actual functions is implemented by hand

In Code First or JPA, models are code; anyway these are model from which other code is generated.

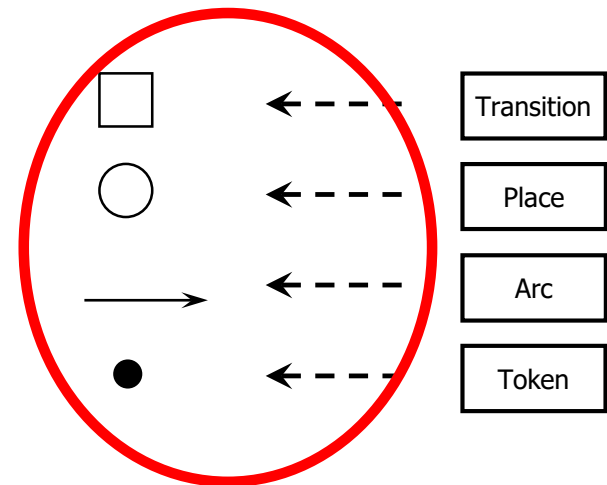
**Future:** Actual functions also „modelled“ and code generated

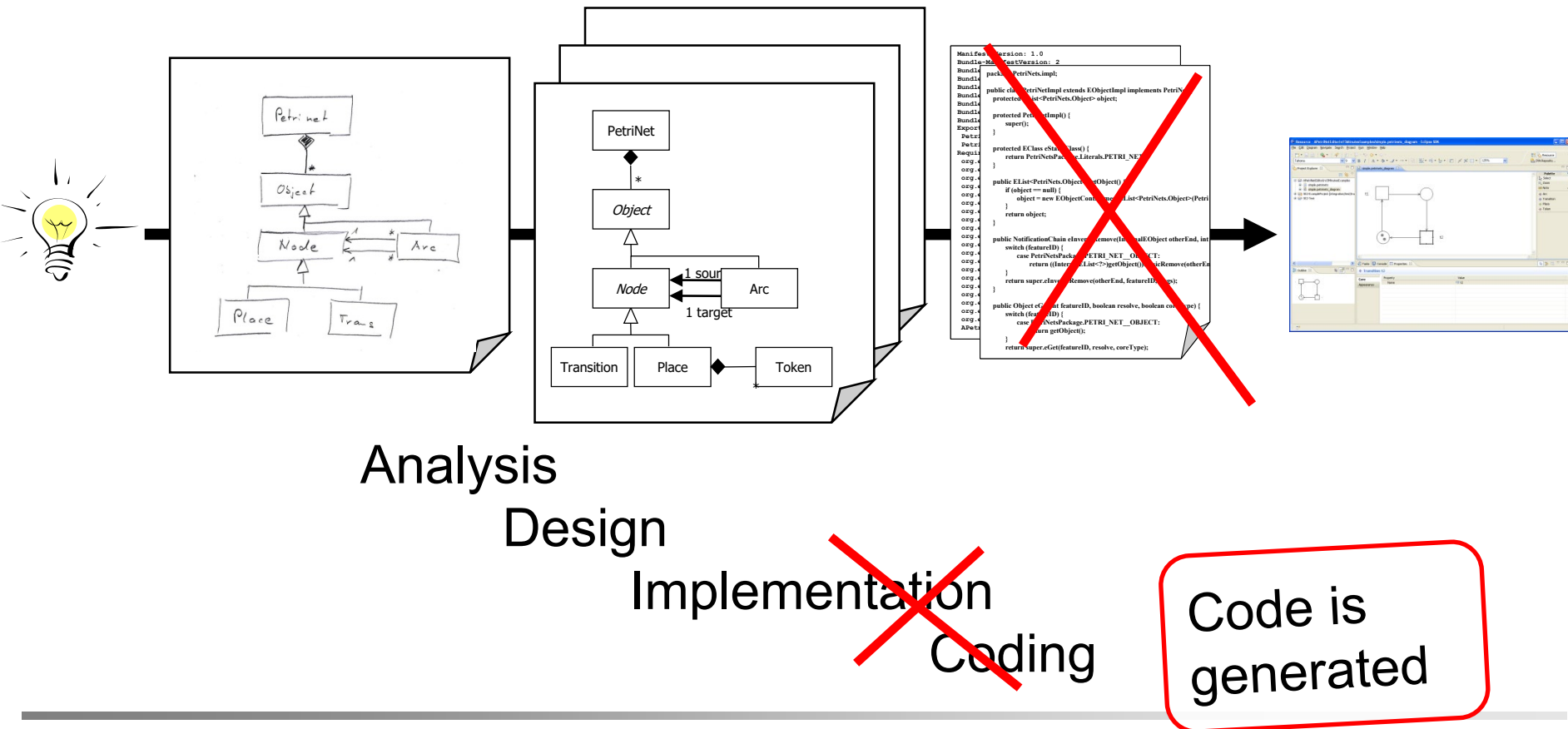
From this (EMF) model for Petri nets:  
Generation of (Java) code for

- all classes
- methods for changing the Petri net
- loading and saving the Petri net as XML files (→XMI)



With this and some more GMF information:  
Generation of the Java code of a graphical complete editor (with many fancy features). No programming at all (to start with).

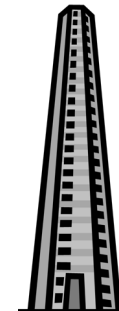
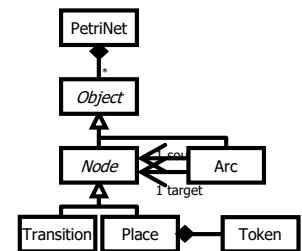
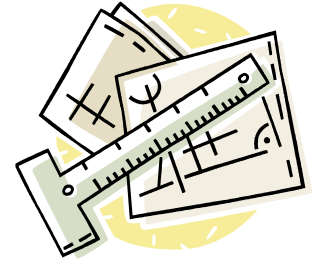


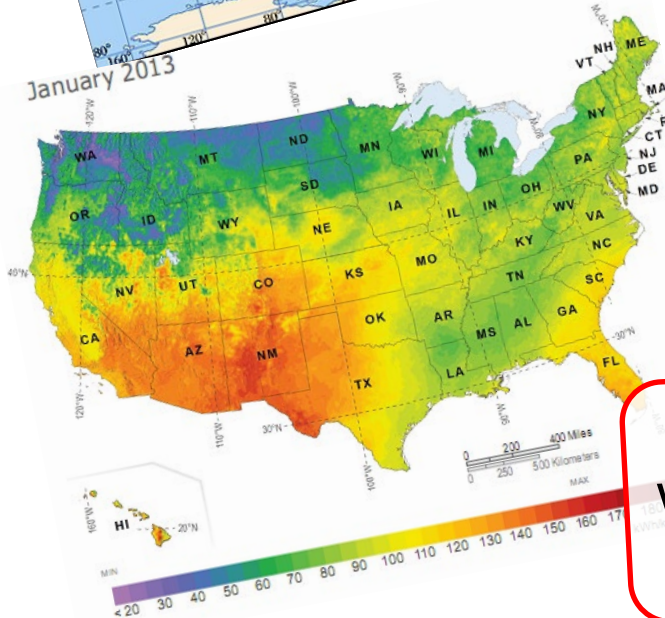
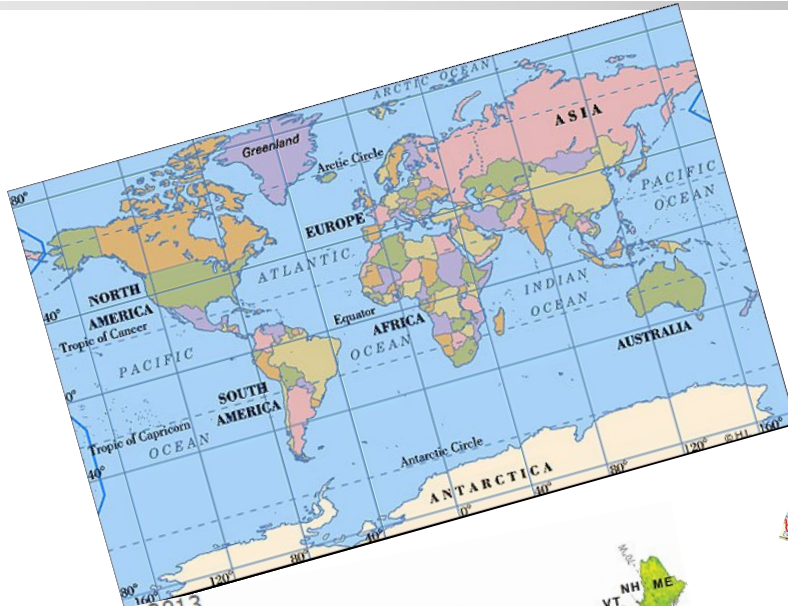


- Better understanding
- Mapping of instances to XML syntax (XMI)
- Automatic code generation
  - API for creating, deleting and modifying model
  - Methods for loading and saving models (in XMI)
  - Standard mechanisms for keeping track of changes (observers)
  - Editors and GUIs

### Analogies:

- Models as floor plans (see earlier slides)
  - Architects and construction engineers use quite different kind of plans – driven by the purpose
  - They even use models (miniatures)
- Models as maps
  - Understand the world (→ domain)
  - Find your way round in the software





Which of them is the best?



- Different level of abstraction and detail
- Different focus
- Different aspects

One map/model  
is not enough

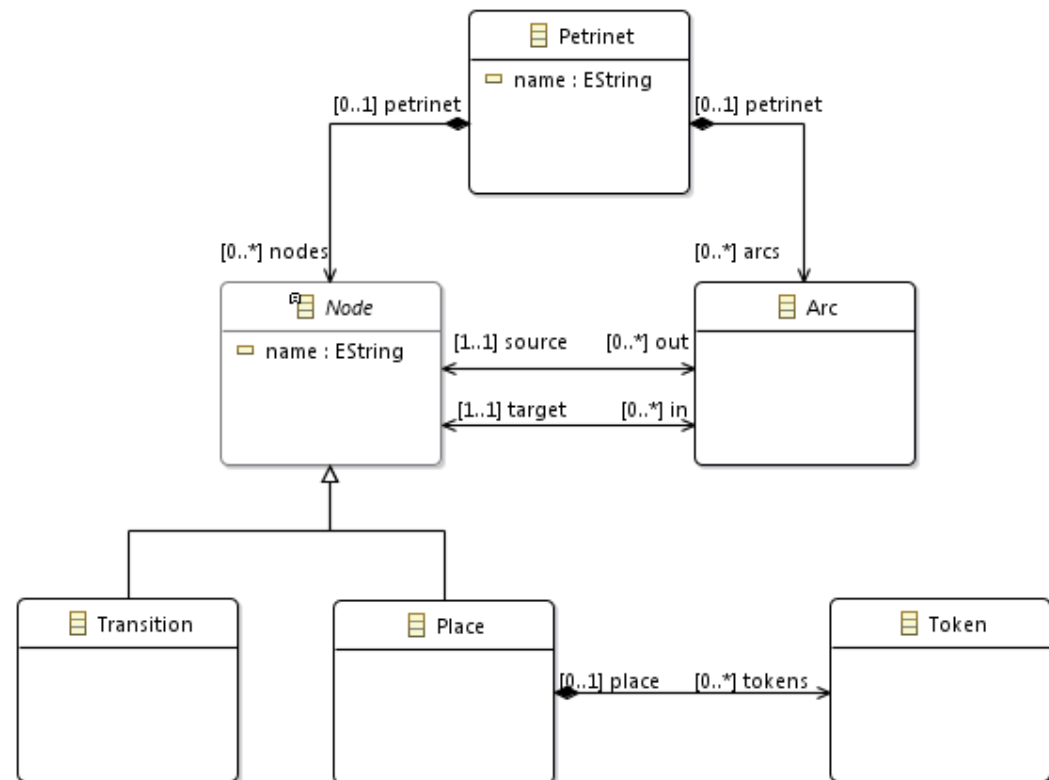
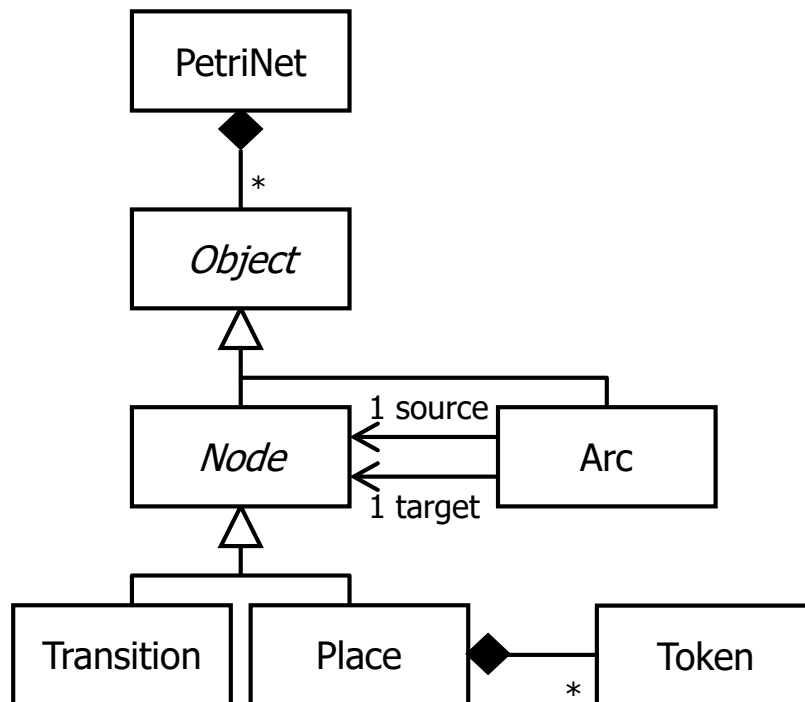
→ **Different purpose**

- For programs (small software), models are often not needed, and making them might be a waste of time
- For software, they are essential for building something which works out and the different pieces fit to each other



- Two models: Which is better?

Always ask first:  
Better for what?



- Blackboard Discussion (BBD):

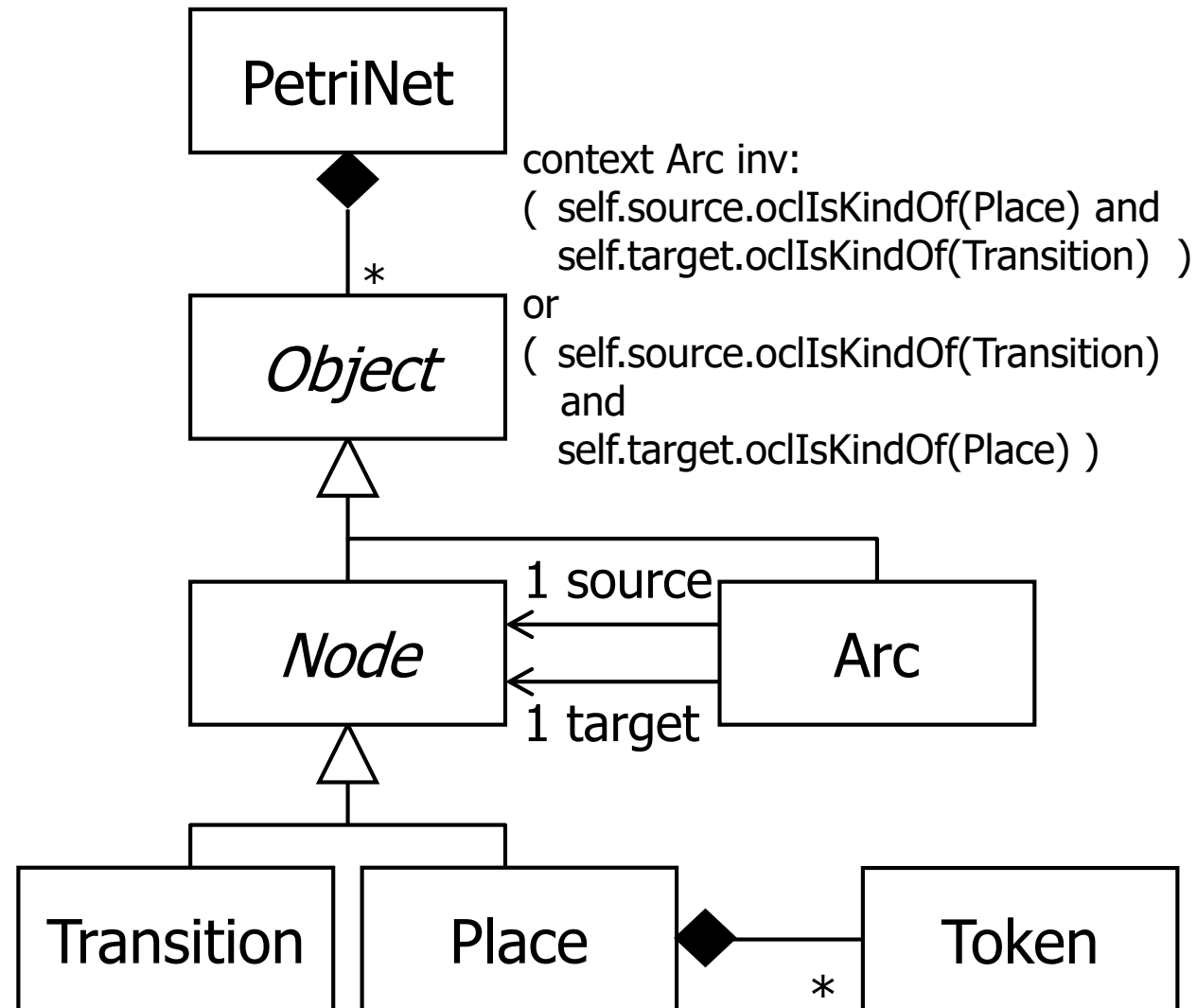
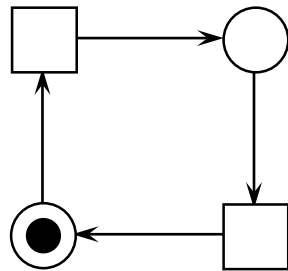
Purpose

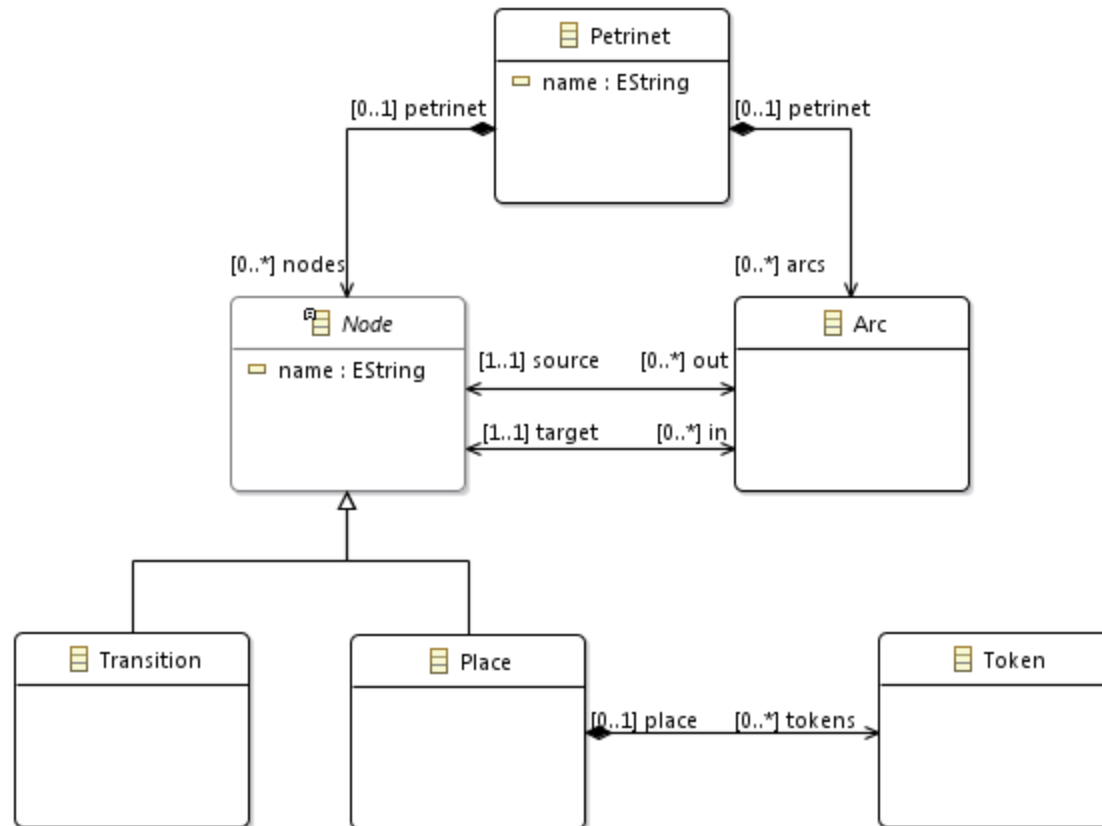
Kind of model

Petri net example revisited (see next two slides)

## Discussion:

- Should in/out (opposites of target and source) be in domain model?
- What makes them a domain model?
- What is the difference to a data model or data base schema?





Same model can have different representations:

- Graphical / tree
- Java
- Ecore
- XML Schema (XSD)

Actually, in our EMF technology, Ecore models can be imported from XML Schema and from annotated Java classes (see Java example on the next slides).

Different representation might serve different purposes and have a different focus!

What would the focus for XSDs, Java and Ecore be?

Also Code First / JPA can be considered a model represented in Java (with annotations mapping it to a database schema).

```
/** @model */  
public interface Petrinet {  
  
    /** @model opposite="petrinet" containment="true" */  
    List<Node> getNodes();  
  
    /** @model opposite="petrinet" containment="true" */  
    List<Arc> getArcs();  
  
    /** @model */  
    String getName();  
  
}
```

```
/** @model */  
public interface Arc {  
  
    /** @model opposite="out" required="true" */  
    Node getSource();  
  
    /** @model opposite="in" required="true" */  
    Node getTarget();  
  
    /** @model opposite="arcs" transient="false" */  
    Petrinet getPetrinet();  
  
}
```



```
/** @model abstract="true" */  
public interface Node {  
  
    /** @model opposite="nodes" transient="false" */  
    Petrinet getPetrinet();  
  
    /** @model opposite="target" */  
    List<Arc> getIn();  
  
    /** @model opposite="source" */  
    List<Arc> getOut();  
  
    /** @model */  
    String getName();  
}
```

```
/**  
 * @model  
 */  
public interface Transition extends Node {  
  
}
```

```
/**
 * @model
 */
public interface Place extends Node {

    /**
     * @model opposite="place" containment="true"
     */
    List<Token> getTokens();

}
```

```
/**
 * @model
 */
public interface Token {

    /**
     * @model opposite="tokens" transient="false"
     */
    Place getPlace();

}
```

```
/** @model */  
public interface Petrinet {  
  
    /** @model opposite="petrinet" containment="true" */  
    List<Node> getNodes();  
  
    /** @model opposite="petrinet" containment="true" */  
    List<Arc> getArcs();  
  
    /** @model */  
    String getName();  
  
}
```

Independently of the representation,

- a domain models solely serves the purpose of getting a grip on the concepts of a domain
- they are not for programming (even though, they might later be used for that)

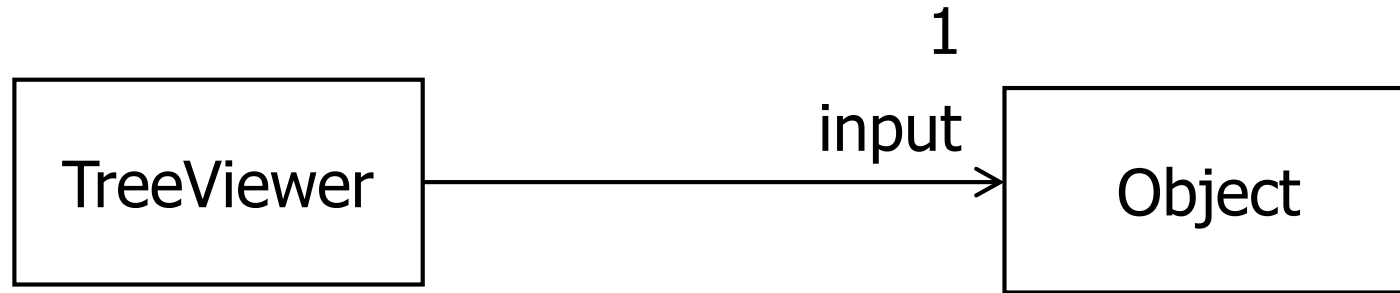
A domain model might also model behaviour. Which UML models are made for modelling a domains behaviour (→ discussion; see also section 7)

Whereas domain models are on the "what" only, software models give an abstraction of the "how" of the software (architecture and design).

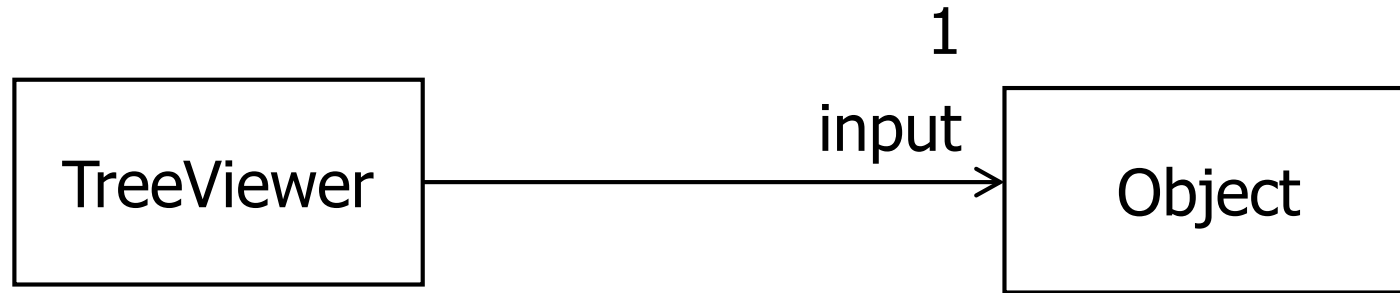
In the following, we use an example from the Eclipse architecture to demonstrate this.

- “JFace is a UI toolkit with classes for handling many common UI programming tasks.”  
[<https://wiki.eclipse.org/JFace>]
- Viewers are a core part of editors (there are different kinds of viewers), which are generic.
- Here, we discuss the TreeViewer, which is the basis for the automatically generated tree editor for Petri nets.





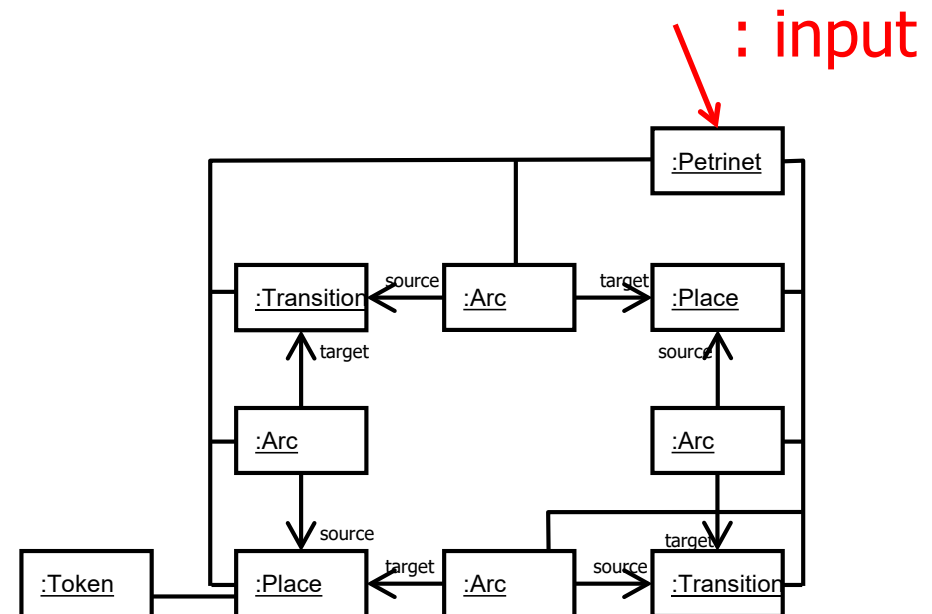
Assuming that the input object (model) is a Petri net (→ slides 8 & 36)

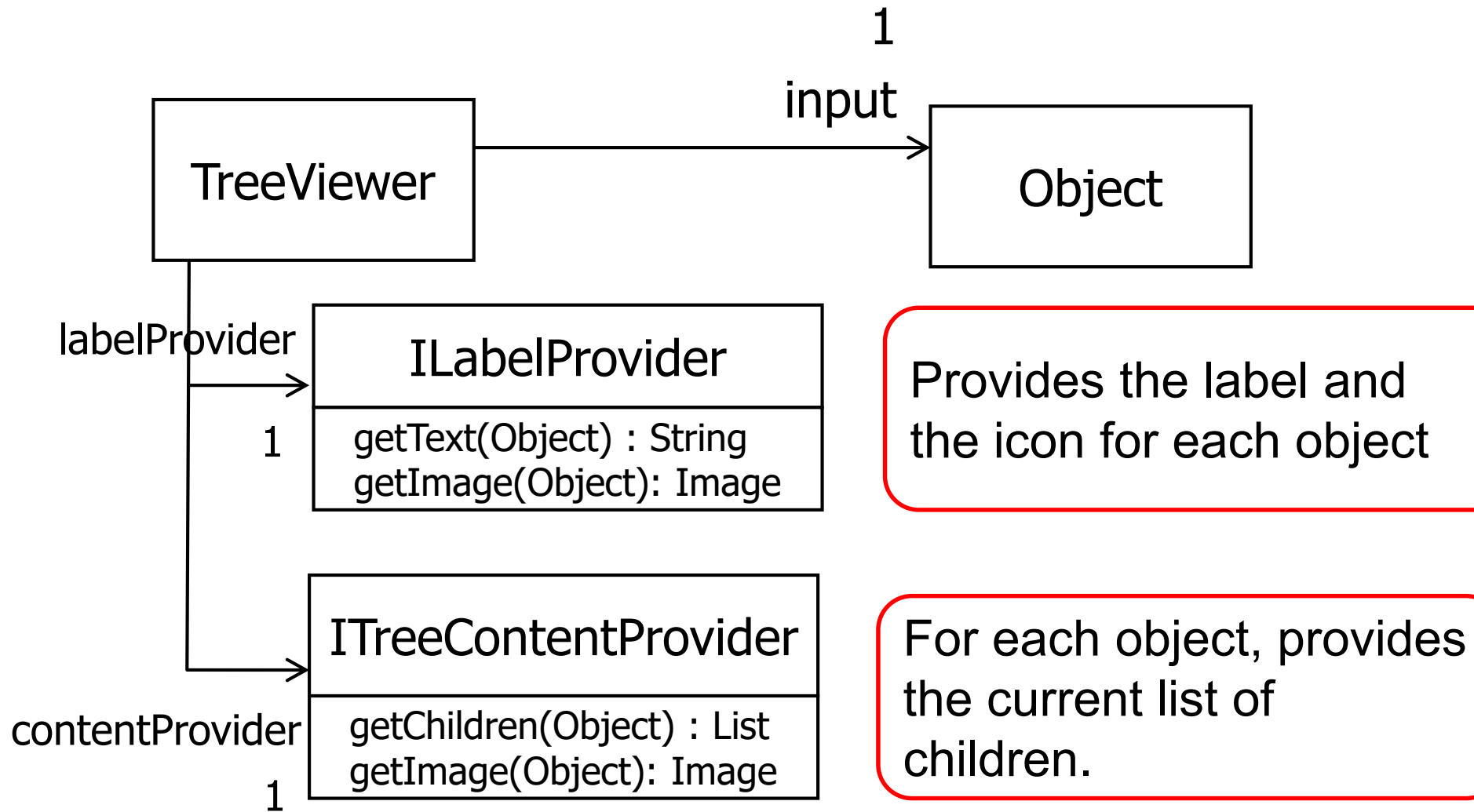


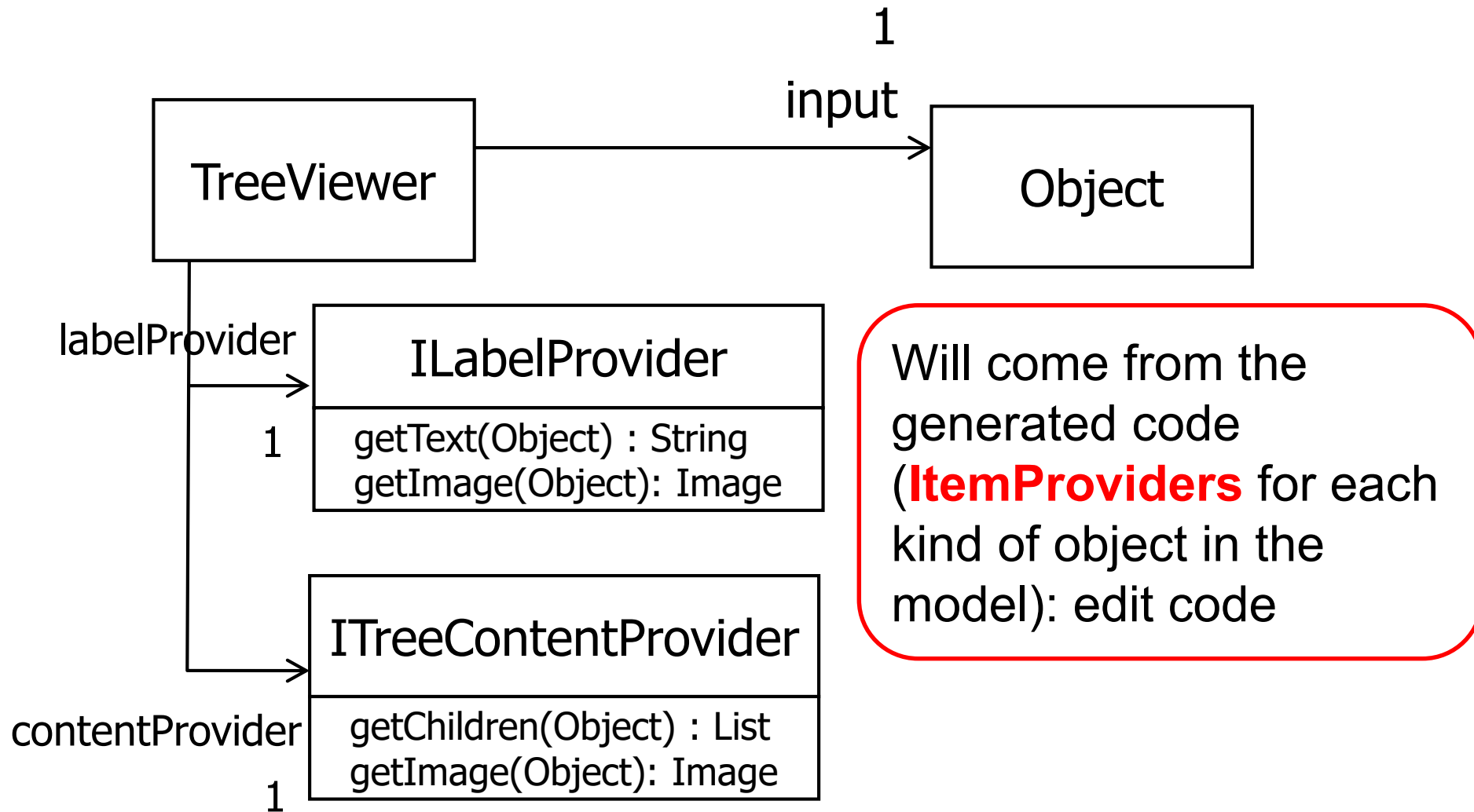
Shows the input as a tree (with all the features of a tree view like opening and closing sub-trees, etc)

Root object of the tree which is to be shown in the TreeViewer

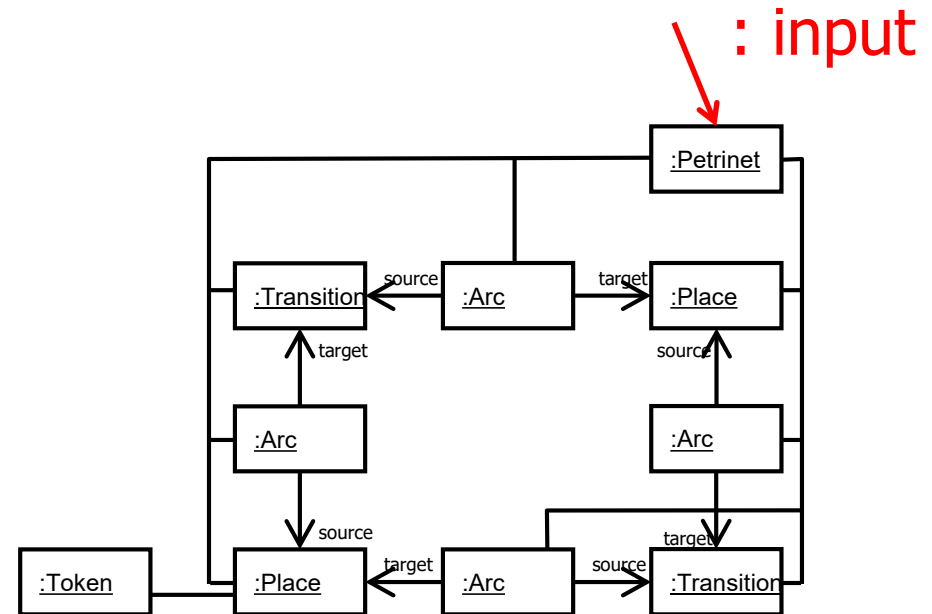
- How could the TreeViewer, which does not know anything about Petri nets (and the classes representing the concepts of Petri nets), know how this tree should be shown?







# Similarly for Properties



| Property | Value          |
|----------|----------------|
| In       | ↗ Arc t2 -> p1 |
| Name     | ≡ p1           |
| Out      | ↗ Arc p1 -> t1 |
|          |                |
|          |                |
|          |                |
|          |                |
|          |                |
|          |                |

ICollectionSourceProvider  
(not discussed here)

- In order to make sure that the viewer properly updates, whenever changes occur, it registers itself as listener to the respective elements (actually to their ItemProviders).

See observer pattern  
later in Sect. 5.

*Mid way  
summary!*

- Understanding the world (conceptual models, domain models)
- Understanding what the software is supposed to do (requirements)
- Understanding and finding your way round in existing software (→ Map)
- Outline the idea of how to realize the software (architecture)
- Overview of components and their interplay
- Detailed design and realization of the software



- Generate parts of the software automatically
- Define data representations (XML, database schemas, ...)
- Define interfaces between different parts of the software
- ...

# 5. Design Patterns

Originally, the term was introduced in architecture: Alexander et al. 1977.

Design patterns (in software engineering) are the distilled experience of software engineering experts on **how** to solve standard problems in software design.

Freeman & Freeman call this “experience reuse”!

Often called the “Gang of Four” (GoF / Go4).

- Gamma, Helm, Johnson, Vlissides:  
Design Patterns. Addison-Wesley 1995.
- Eric Freeman, Elisabeth Freeman:  
Head First Design Patterns. O'Reilly  
2004 [FF]
- ...

- Design patterns are a topic of their own, worth being taught as a separate course (e.g. seminar/special course)
- This lecture gives just a glimpse of the general idea and some patterns, which are important to understand and use EMF

## **Name and classification**

Observer, object, behavioural

## **Intent**

”Define a one-to-many dependency between objects so that an object changes all its dependents are notified and updated automatically” [GoF].

## **Also know as**

Dependents, Publish-Subscribe, Listener

## Motivation

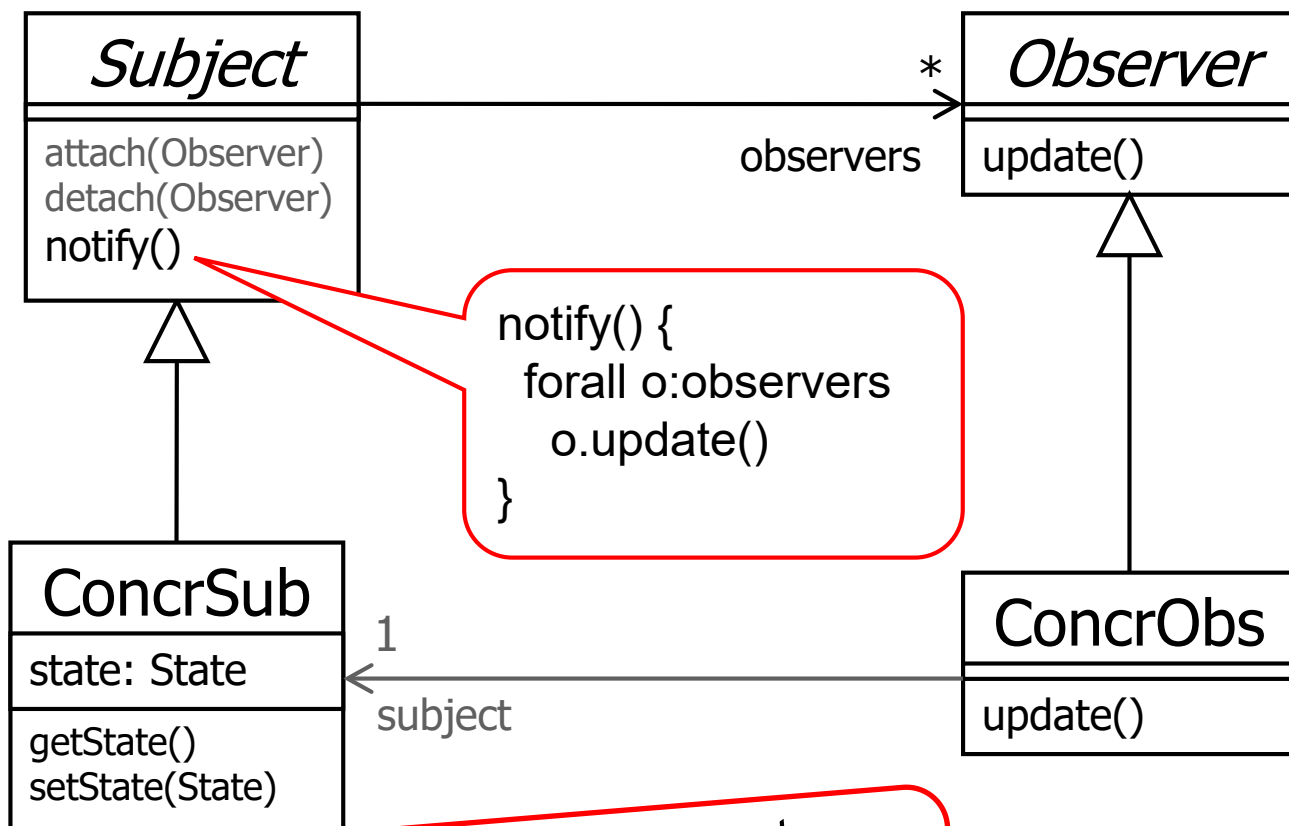
[...] maintain consistency between related objects without introducing tight coupling (which increases reusability) [...]

## Typical Example

... update views when the underlying model changes ...

Roughly following  
GoF

## Structure



The concrete observer does not necessarily need to "know" the concrete subject; then the subject is passed as a parameter to `update()`

## Participants (see structure)



- **Subject**

- knows its observers
- provides an interface for attaching and detaching Observer objects

- **Observer**

- defines the updating interface for being notified

- **ConcreteSubject**

- stores the state (of interest)
- sends notifications

- **ConcreteObserver**

- Implements the Observer's updating interface to keep its state consistent



## Collaboration



Black board discussion

- Name
- Classification
- Intent
- Also known as (aka)
- Motivation
- Application
- Structure
- Participants
- Collaboration
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

Sometimes there is more:  
Variants, “Counter indications”,  
...

What we called  
Factory up to now.

## Name and classification

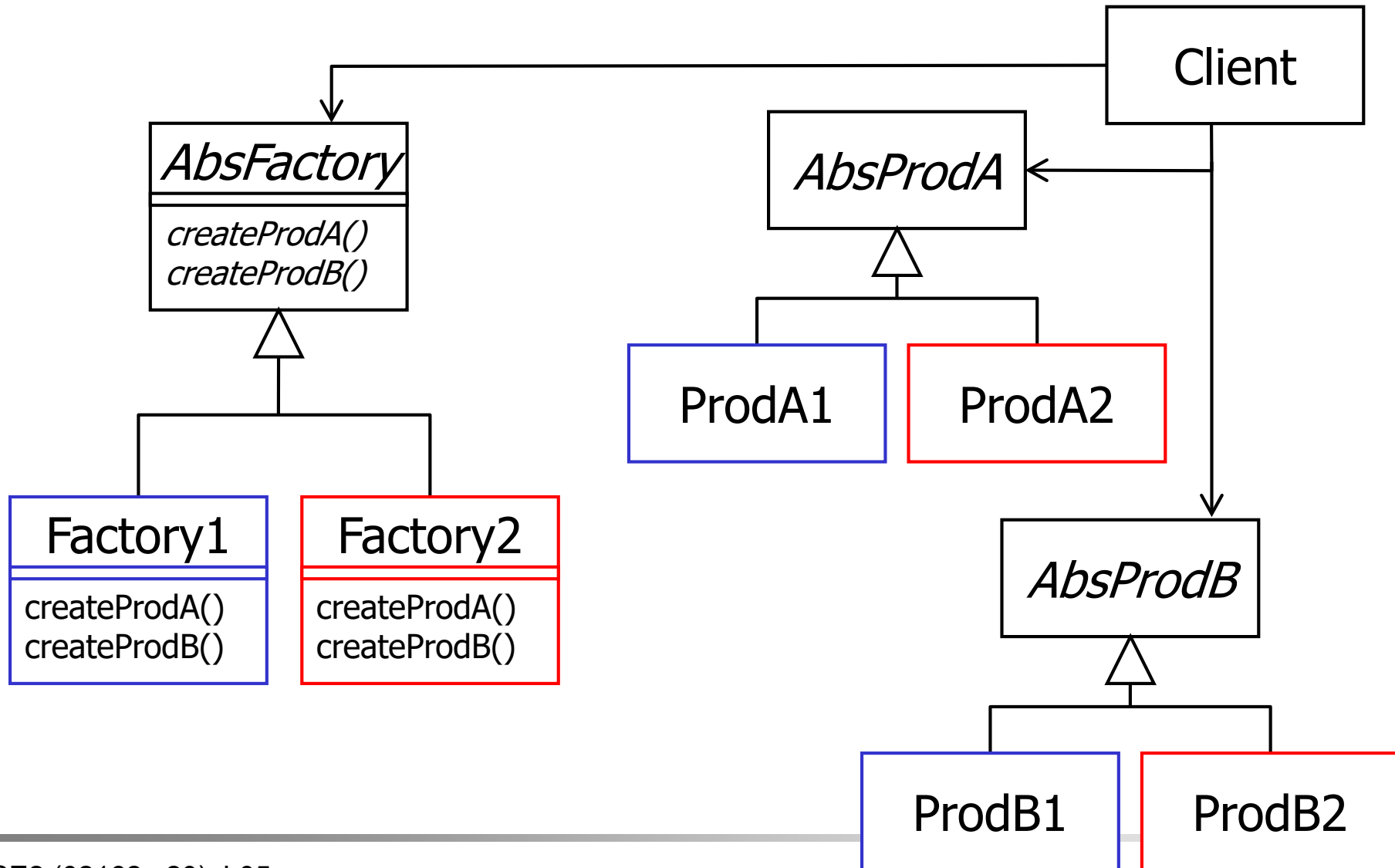
Abstract factory, object, creational

## Intent

Provide an interface for creating families of related or dependent objects without specifying their concrete classes [GoF]

## Motivation

Use of different implementations in different contexts with easy portability ...



## Name and classification

Singleton, object-based, creational

## Intent

Ensure that a class has only one instance, and provide a global point of access to it [GoF]

## Motivation

...

See [GoF] or [FF] for details.

- Factory Method
- Command
- Adapter

The Factory Method pattern is different from the Abstract Factory.

- GoF present 23 patterns
- There are many more (and more complex combinations of patterns, e.g. MVC --)
- “Pattern terminology” can be used to communicate design!
- Patterns should **not** be used to schematically
- Generated code, typically, makes use of many patterns. Automatic code generation “saves us making some design decisions” (observer, singleton, factory, and adapters are part of the EMF-generated code)

The **domain models** are an (the) essential part of the software

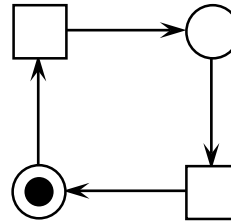
In addition to that, we need

- Information about the presentation of the model to the user
- The coordination with the user

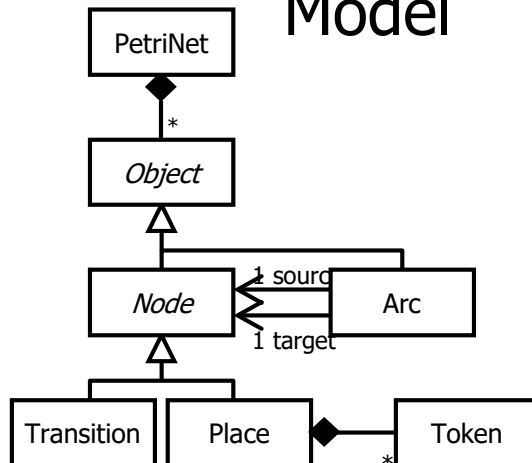
Note: These parts of the software can be modelled too (don't get confused: „models are everywhere“); domain model vs. software model



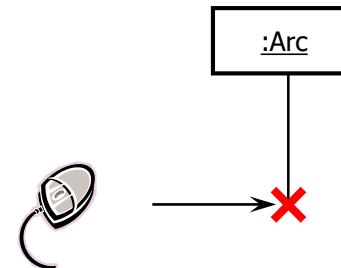
## View



## Model

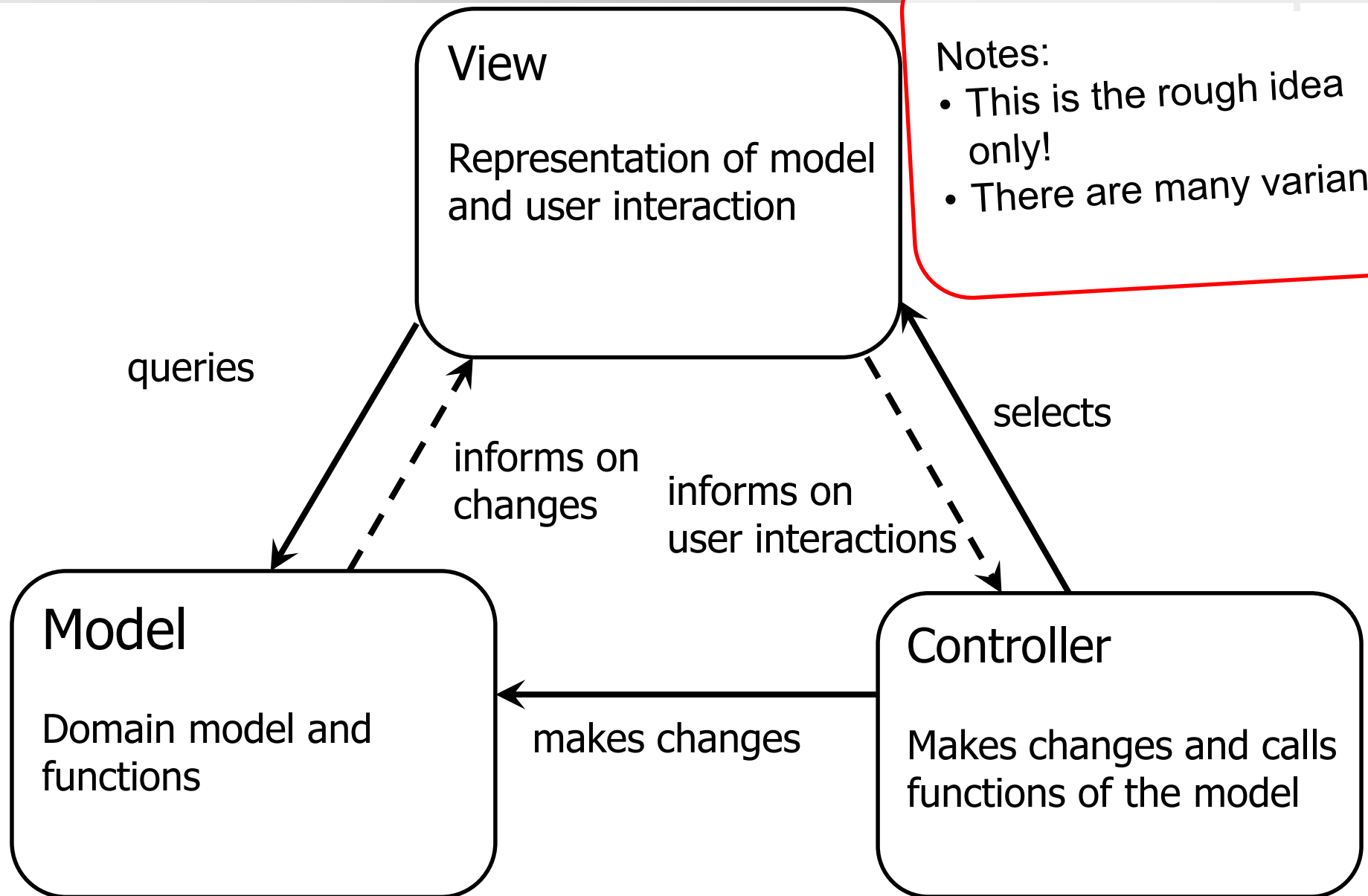


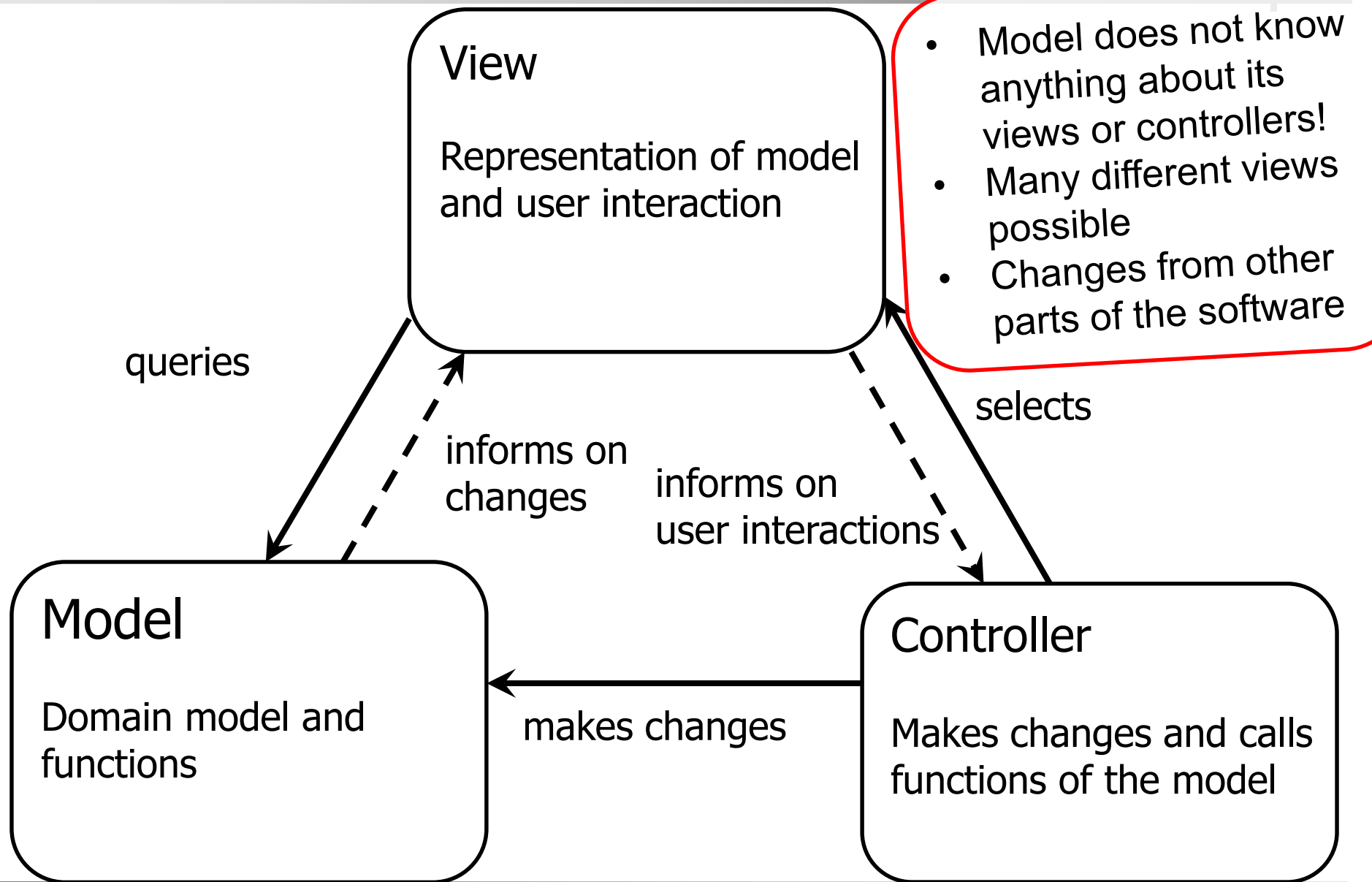
## Controller



Notes:

- This is the rough idea only!
- There are many variants





MVC is a principle (pattern / architecture)  
according to which software should be structured

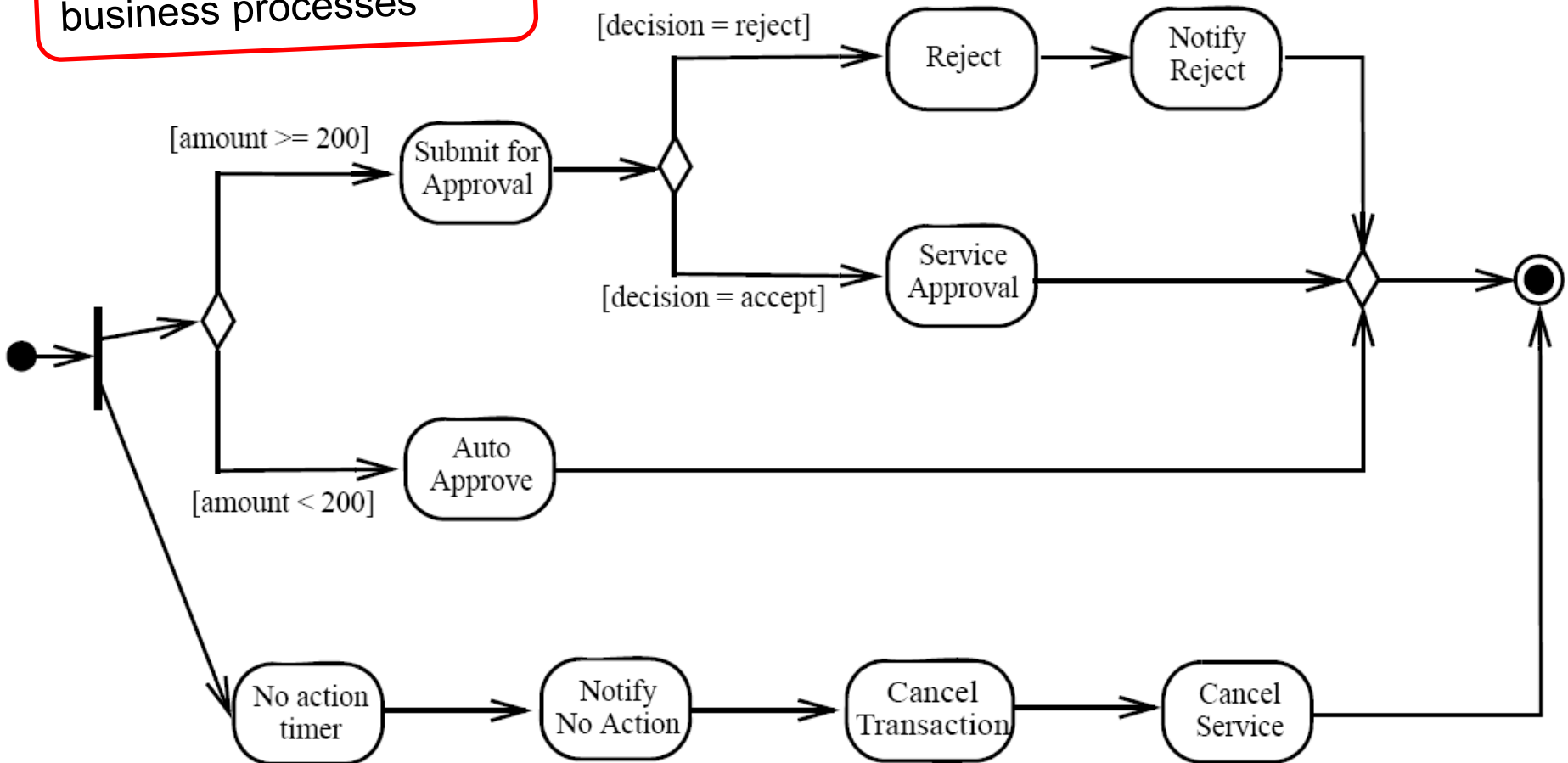
Eclipse and GEF (as well as GMF) are based on  
this principle and guide (force) you in properly  
using it

In UML, there are different concepts and diagrams that concern behaviour modelling

- Use case diagrams
- Activity diagrams
- Interaction diagrams
  - Sequence diagrams
  - Communication diagrams
- State machine diagrams (State Charts)
- Methods of classes (MOF: Operation)  
(in combination with OCL, the input/output relation of a method can be specified)

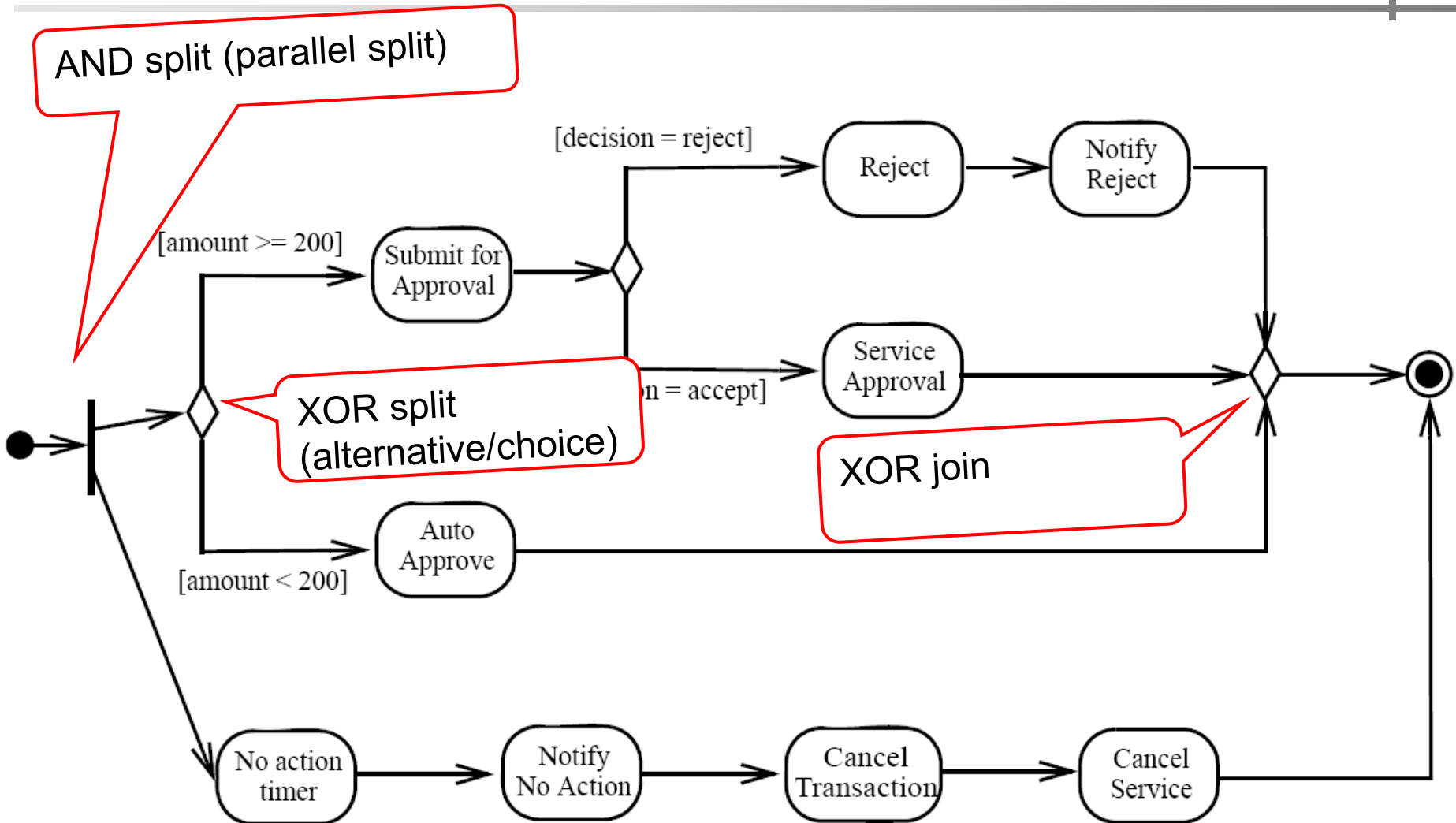
Use cases talk about functionality, which is “behaviour on a high level of abstraction”; they are not very concrete; but a use case can be associated with other behaviour diagrams with a more detailed description of the behaviour.

UML way of modelling  
business processes



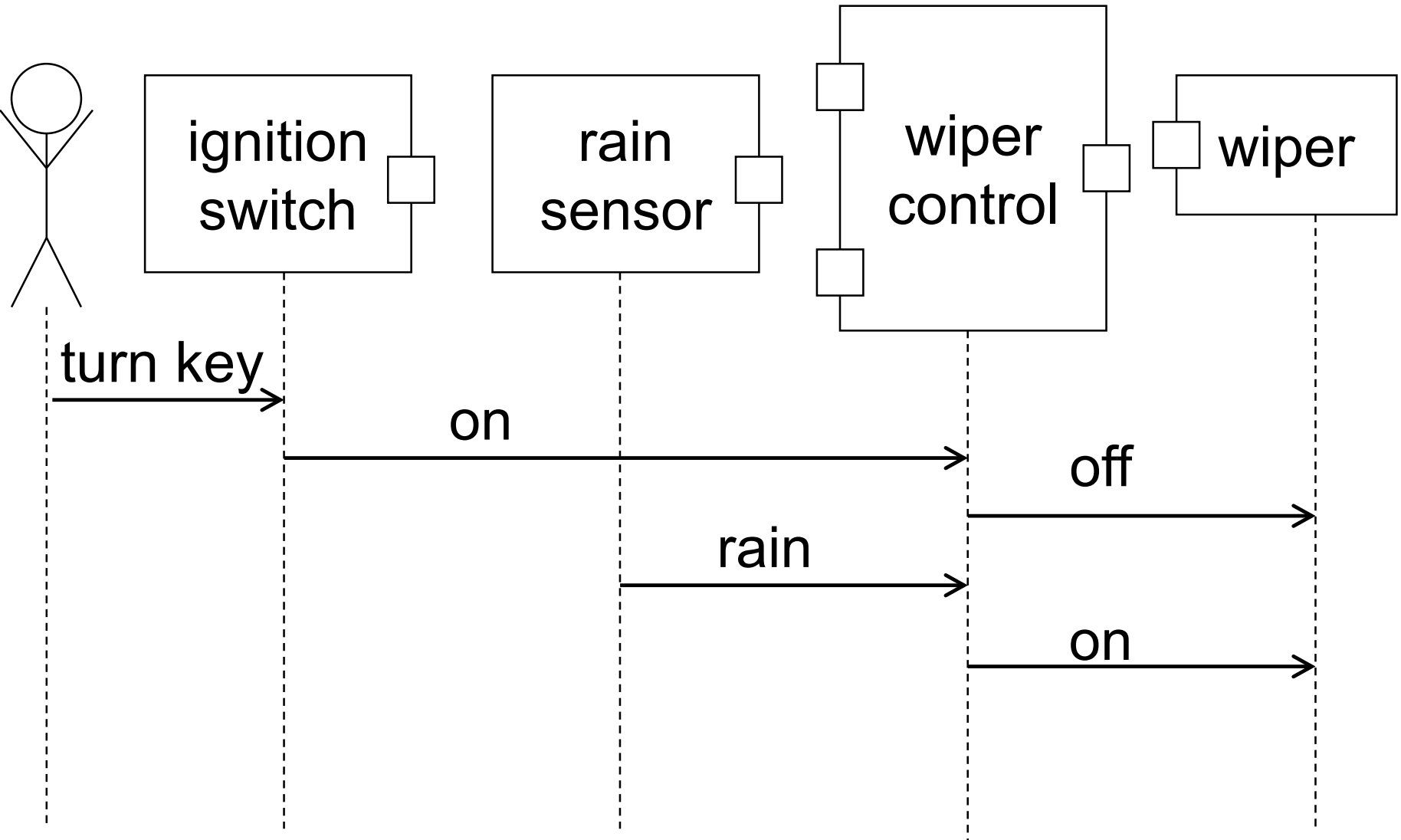
From: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2,  
November 2007, p. 331

# Activity diagrams

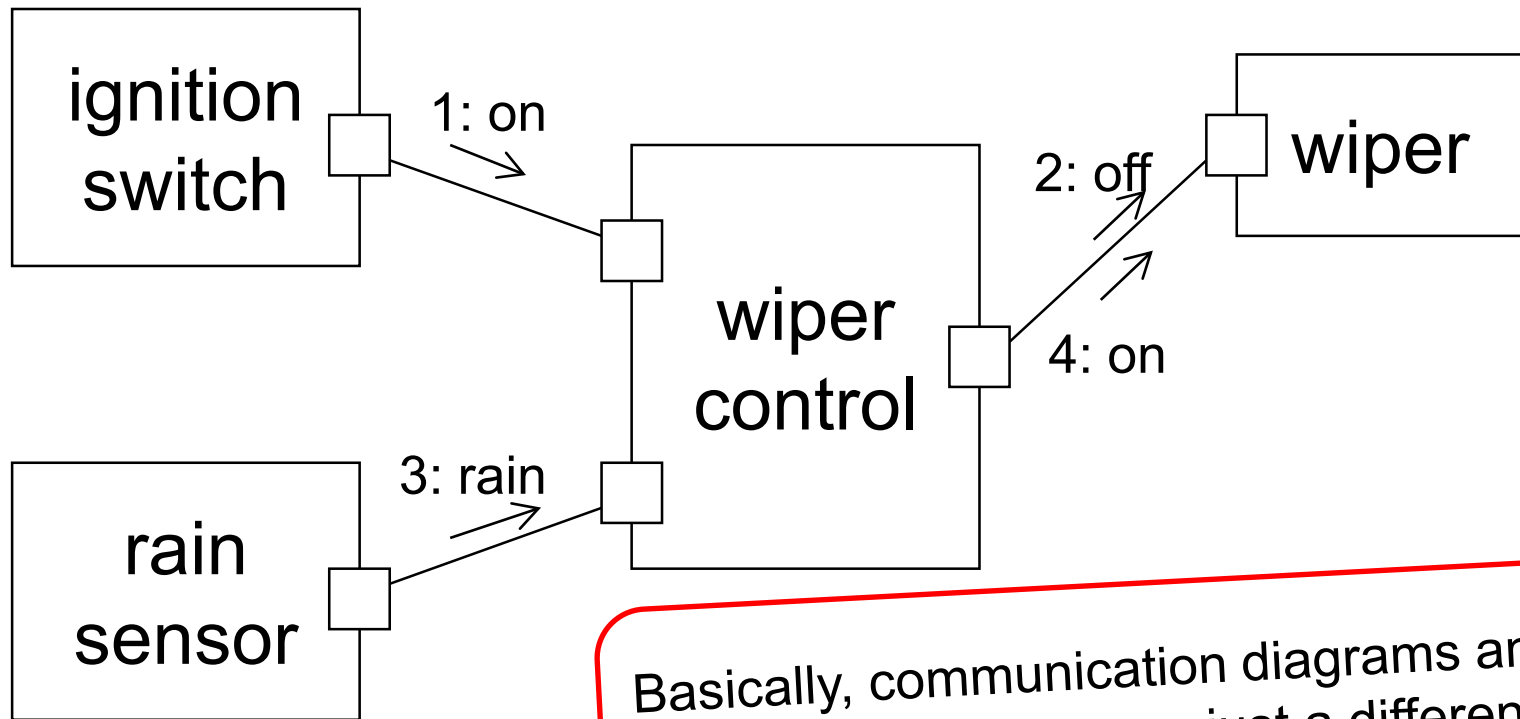


From: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2, November 2007, p. 331

# “Sequence Diagram”

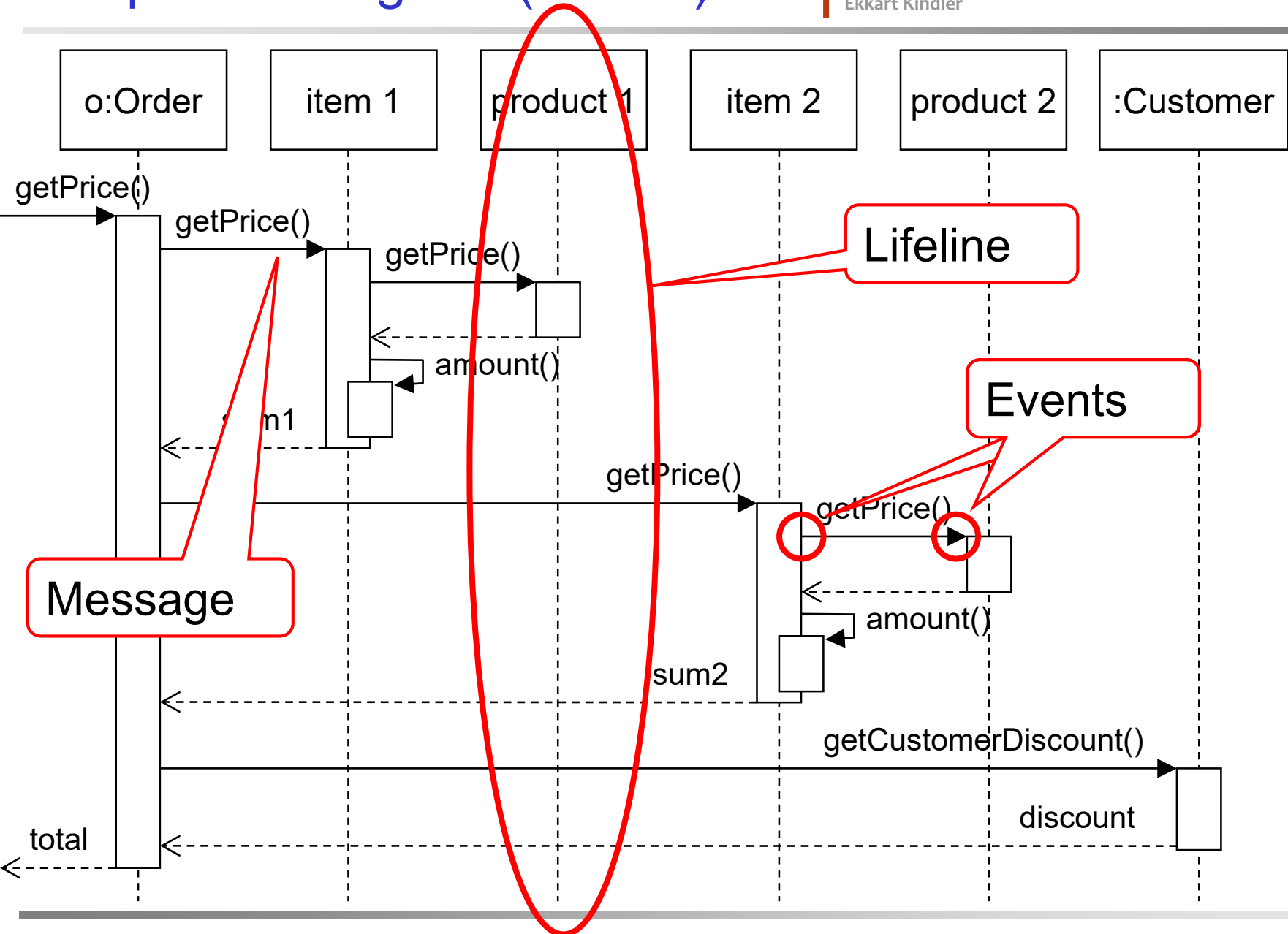




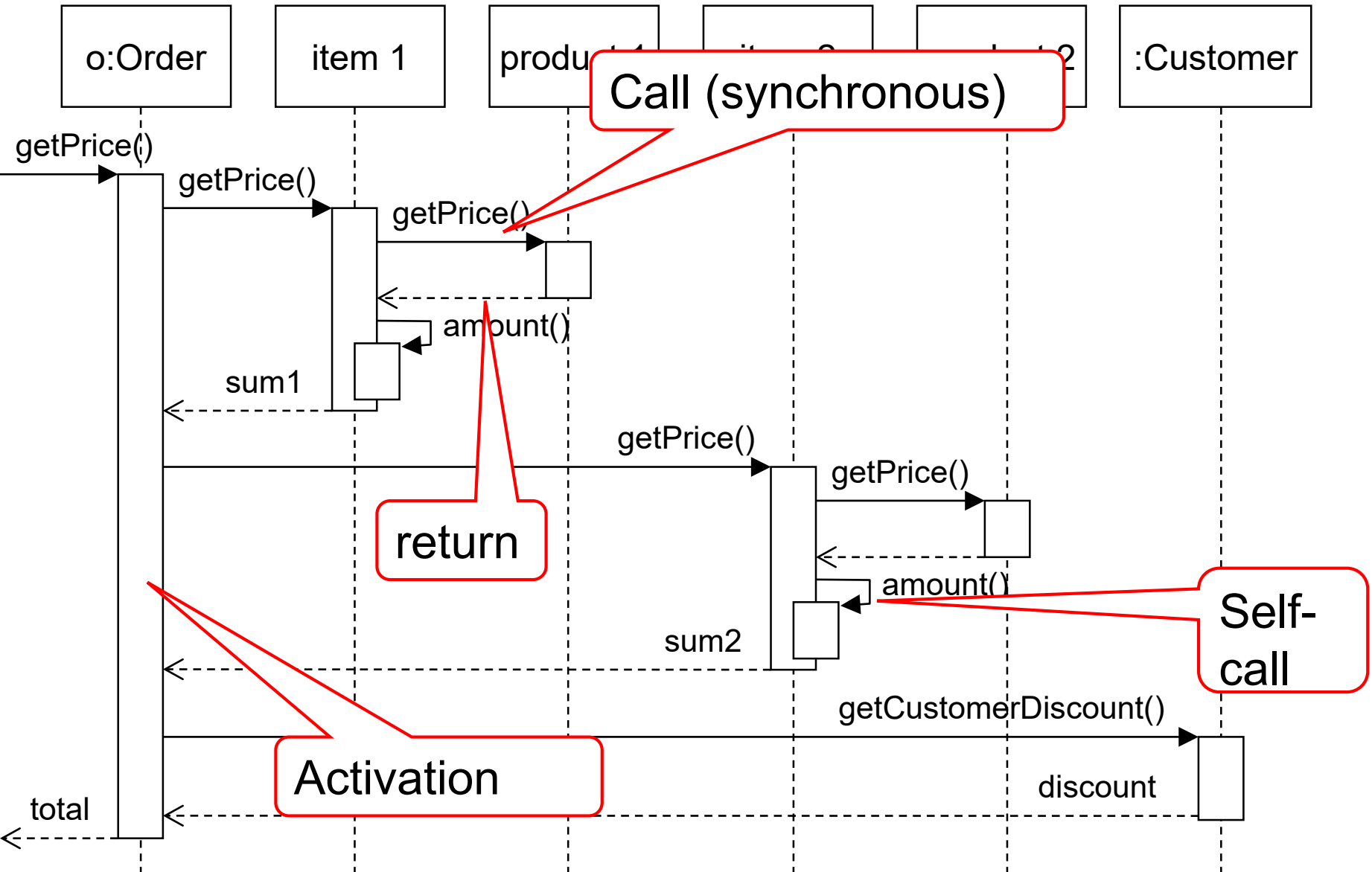


Basically, communication diagrams and sequence diagrams are just a different representation of the same concepts.

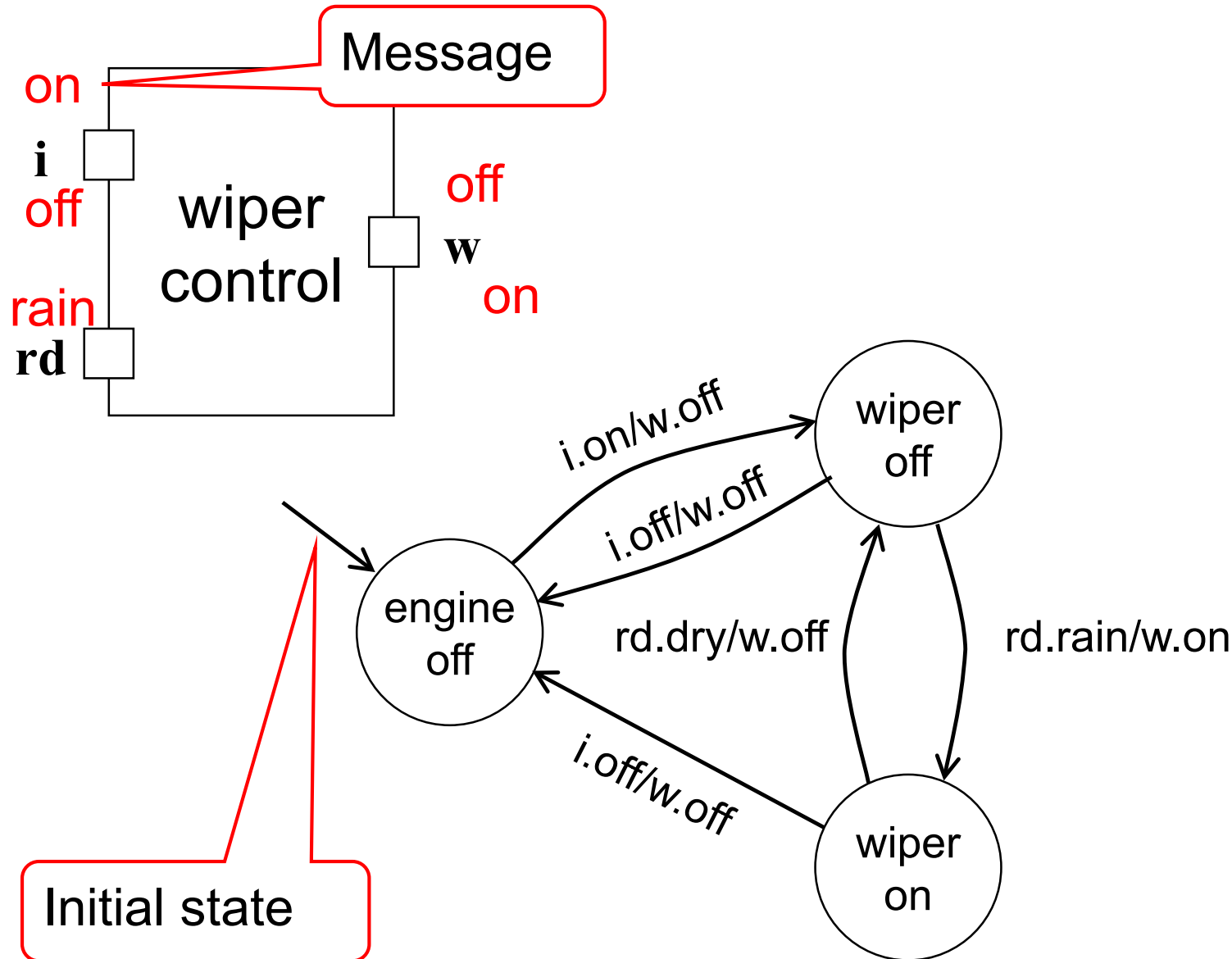
# Sequence diagram (details)

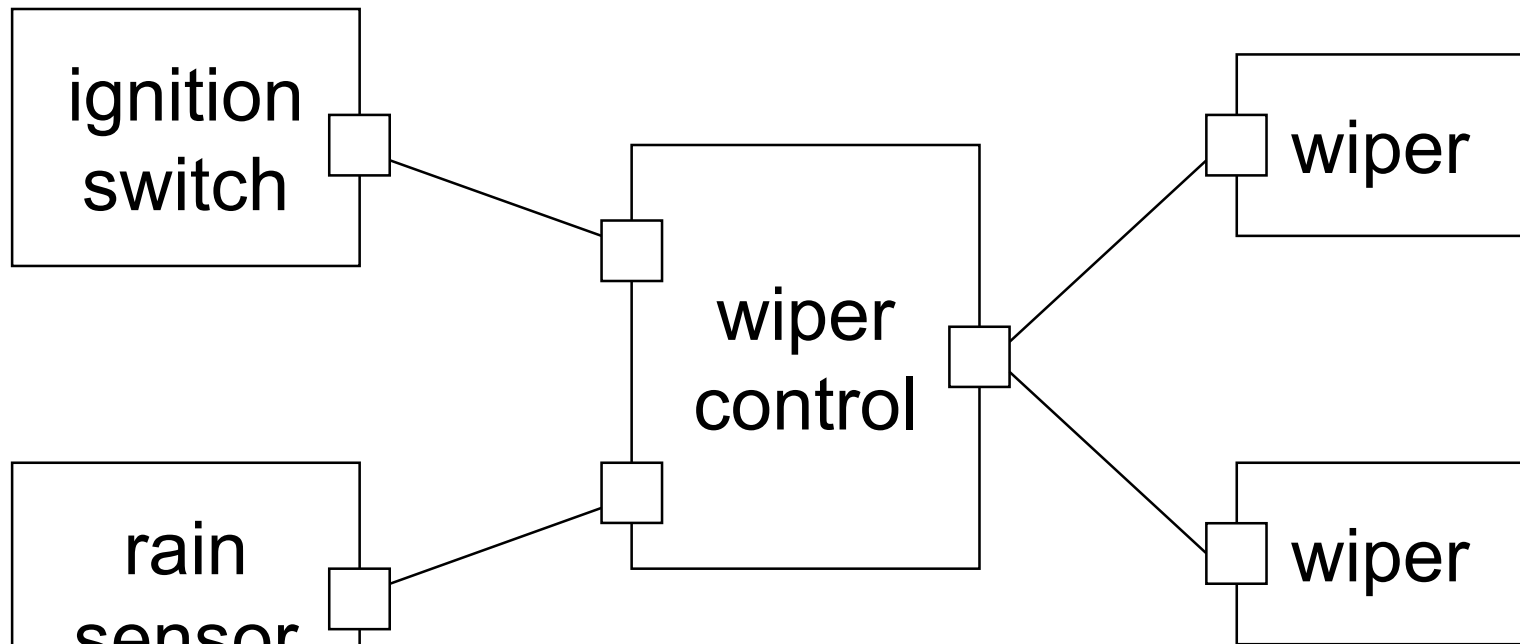


# Sequence diagram (details)



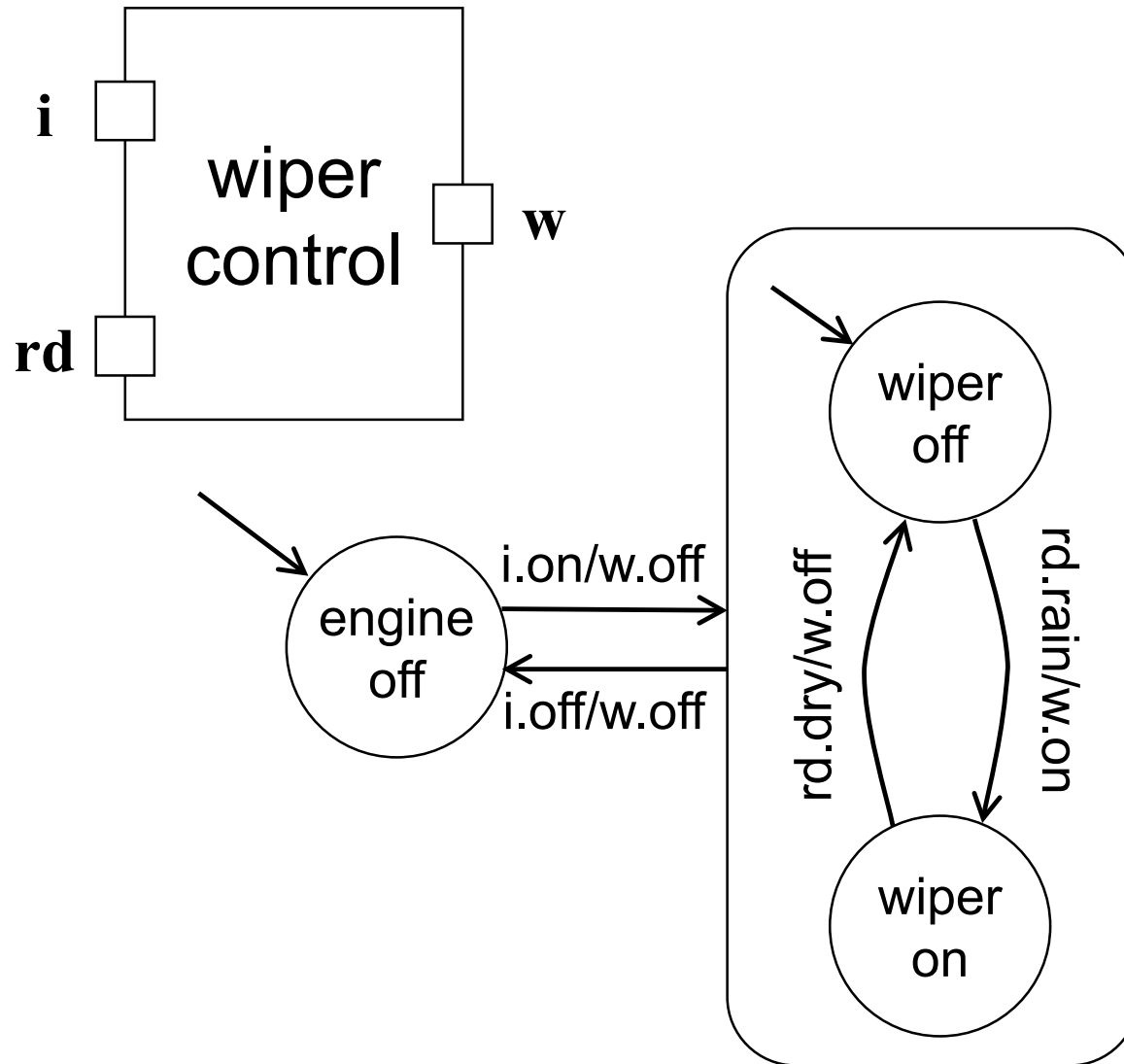
# “State machines”





One automaton for each component (plus this structure) defines the complete behaviour of our “wiper system”.

# “State machines”



Complex  
state

State machines  
(automata) come in  
many different flavours  
and with many  
different semantics.

- Use case diagrams
- Activity diagrams
- Interaction diagrams
  - Sequence diagrams
  - Communication diagrams
- State machine diagrams (State Charts)
- Methods of classes  
(in combination with OCL, the input/output relation of a method can be specified)

Discuss:  
→ Why so many?  
→ What is their purpose?

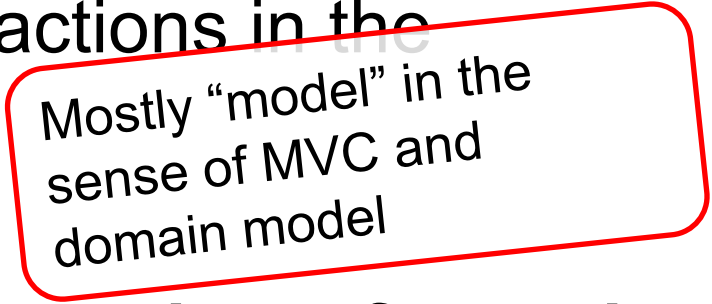
- In general, UML behaviour models are used to analyse, design, and document a system
- It is hard to generate code from that automatically.

There are some exceptions: e.g. Harel and Marelly (Book: "Come Let's Play", 2003) show that sequence diagrams are sufficient for making a system work. But this does not work on the large scale.



# 8. Architectural Views

(after Sommerville)

- Logical view: ... shows the abstractions in the system ...  

- **Development view: ... shows how the software is decomposed for the development ...**
- Process view: ... how the system is composed of interaction processes at runtime ...
- Physical view: shows hardware and how software components are distributed across it ...

- + Use cases



More details later