Software Design (20VT-2DV608)

Requirements Engineering

Francis Palma
francis.palma@lnu.se

Department of Computer Science and Media Technology
Linnaeus University

Course Outline

- Why Software Engineering - Design (Mauro Caporuscio)
  - w4
- Requirement Engineering (Francis Palma)
  - w5.1 (Jan 29th): Understanding and Elicitation of Software Requirements
  - w5.2 (Jan 31st): Requirements Validation and Management
  - **w6.1 (Feb 5th): Modeling with UML**
  - w6.2 (Feb 7th): Requirements Modelling and Management with Tools
  - W7: ASSIGNMENT RE
- Performance Engineering (Diego Perez)
  - w8.1 to w10
- Design and Refactoring (Mauro Caporuscio)
  - w11.1 to w13

Unified Modeling Language (UML)

- The Unified Modeling Language (UML) is a standard graphical language for modeling object-oriented software.

- UML diagrams show the classes, their attributes, and operations as well as the various types of relationships that exist among the classes.

Object Management Group (OMG)

- The Object Management Group (OMG) is an international and open membership technology standards consortium.
- OMG standards are driven by vendors, end-users, academic institutions, and government agencies.
- www.omg.org, www.uml.org

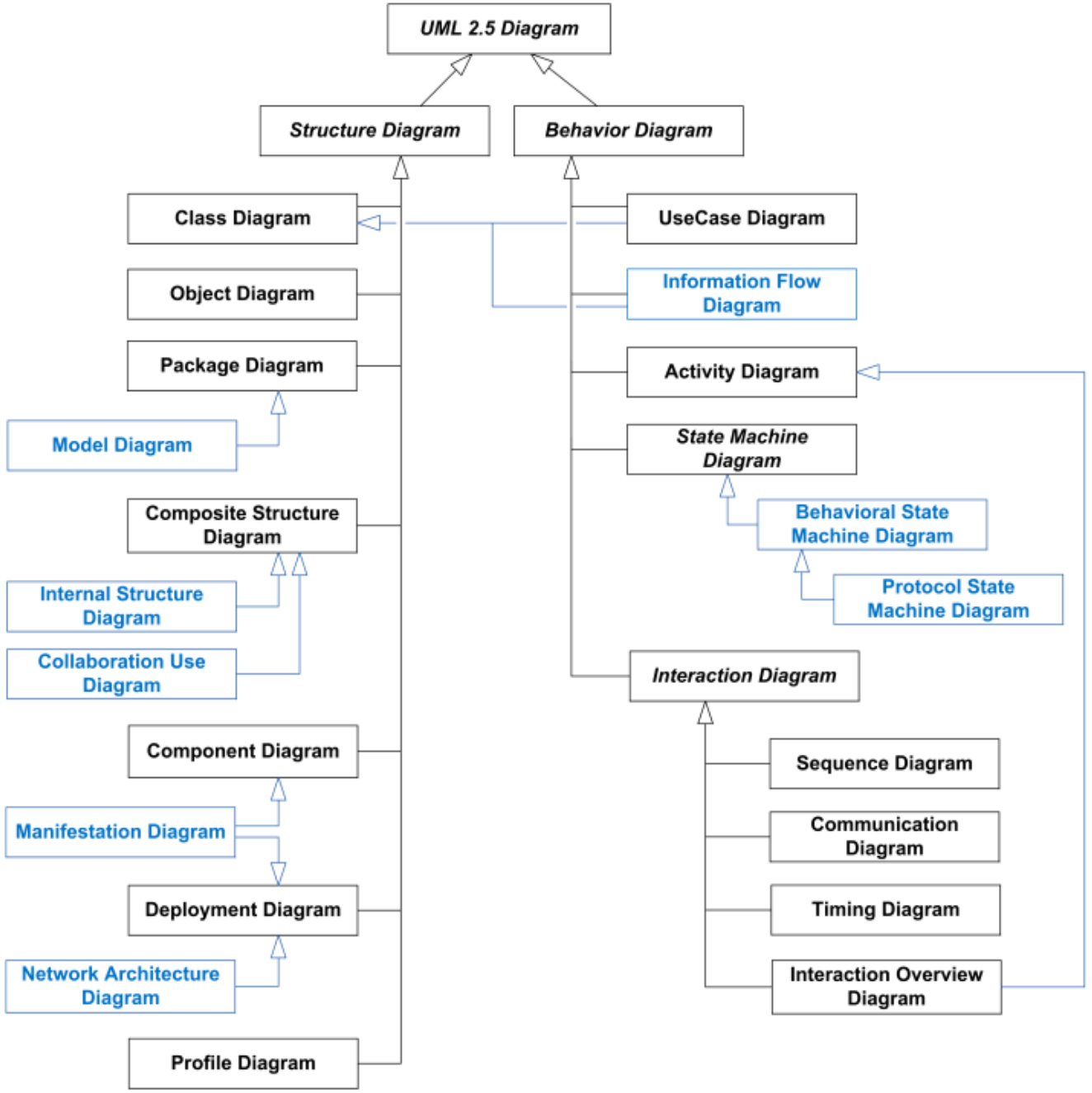
Version	Adoption Date	URL
2.5.1	December 2017	omg.org/spec/UML/2.5.1
2.4.1	July 2011	omg.org/spec/UML/2.4.1
2.3	May 2010	omg.org/spec/UML/2.3
2.2	January 2009	omg.org/spec/UML/2.2
2.1.2	October 2007	omg.org/spec/UML/2.1.2
2.0	July 2005	omg.org/spec/UML/2.0
1.5	March 2003	omg.org/spec/UML/1.5
1.4	September 2001	omg.org/spec/UML/1.4
1.3	February 2000	omg.org/spec/UML/1.3
1.2	July 1999	omg.org/spec/UML/1.2
1.1	December 1997	omg.org/spec/UML/1.1

Diagrams in UML

- Class Diagram
- Activity Diagram
- Communication Diagram
- Component Diagram
- Deployment diagram
- Interaction Overview Diagram
- Object Diagram
- Package Diagram
- Profile Diagram
- State Machine Diagram
- Sequence Diagram
- Timing Diagram
- Use Case Diagram

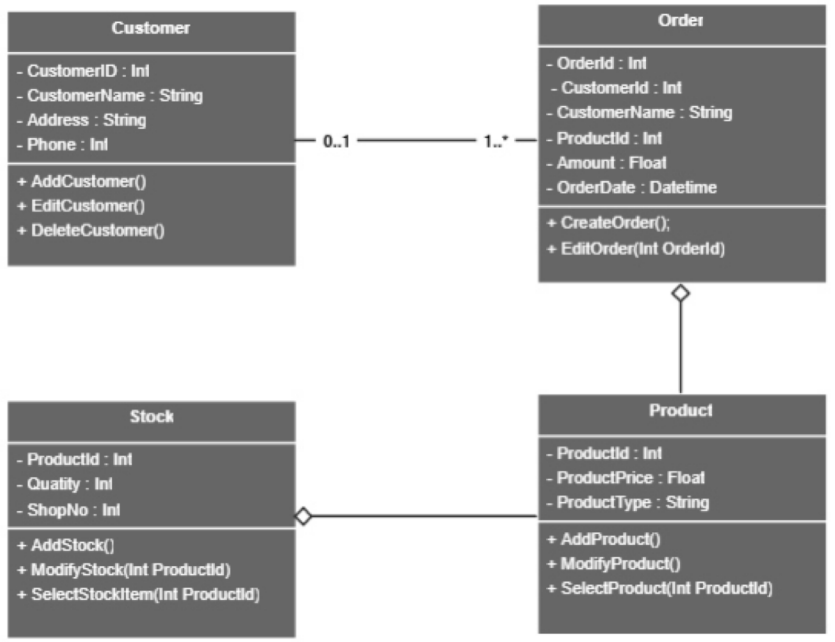
Linnéuniversitetet

Taxonomy of Diagrams in UML



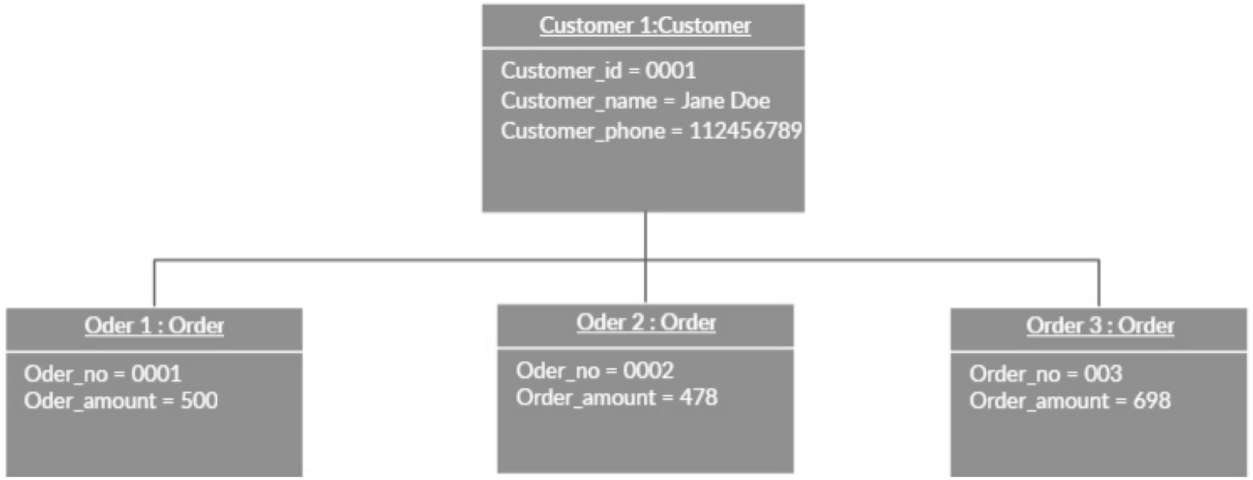
Class Diagram

- A type of structure diagram that describe the entities or elements that must be present in the system being modelled.



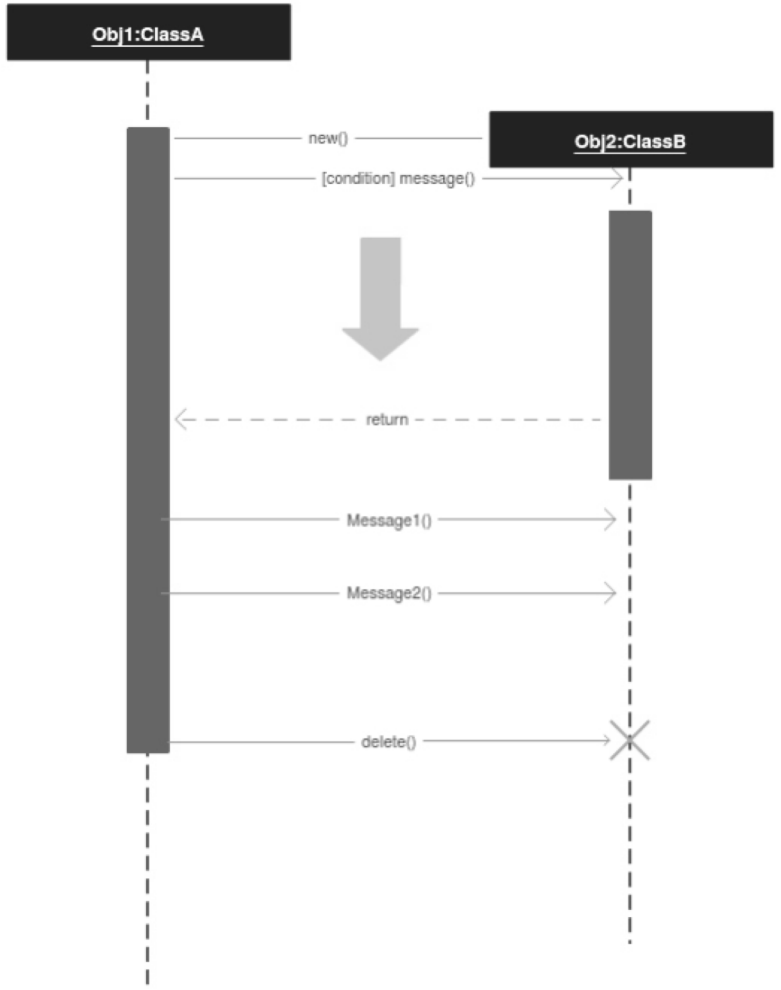
Object Diagram

- Represents a specific instance of a class diagram at a certain moment in time.
  - Focuses on the attributes of objects and how those objects relate to each other.



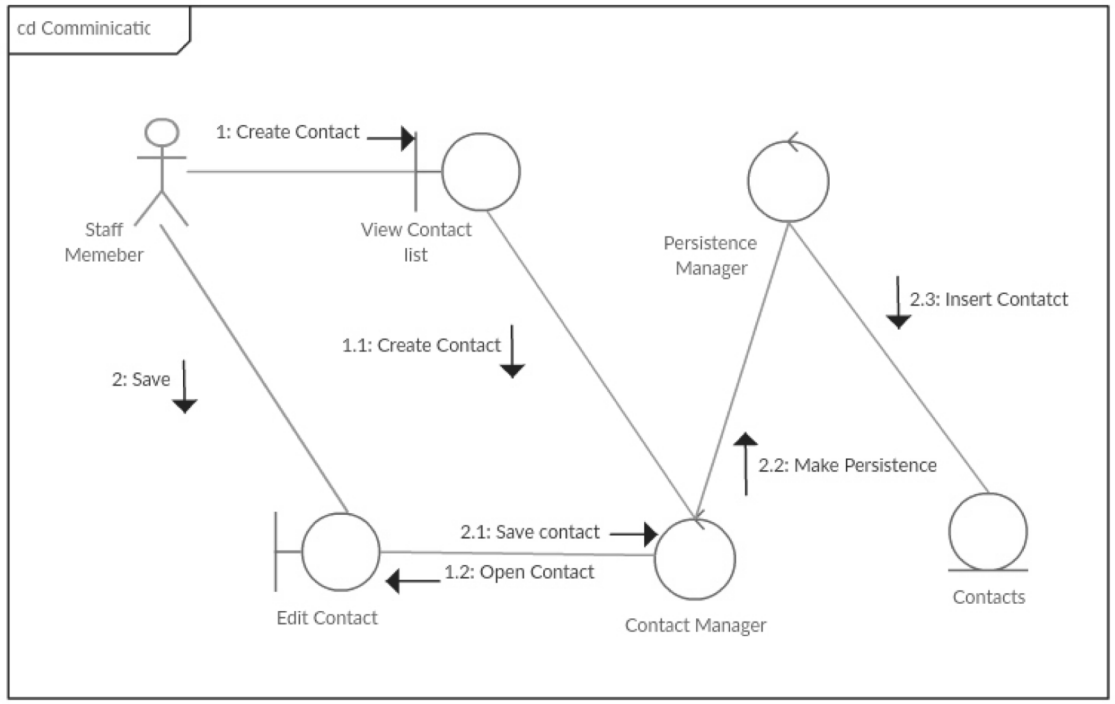
Sequence Diagram

- A *dynamic* modeling that focuses on lifelines, or the processes and objects that live simultaneously, and the messages exchanged between them to perform a function before the lifeline ends.



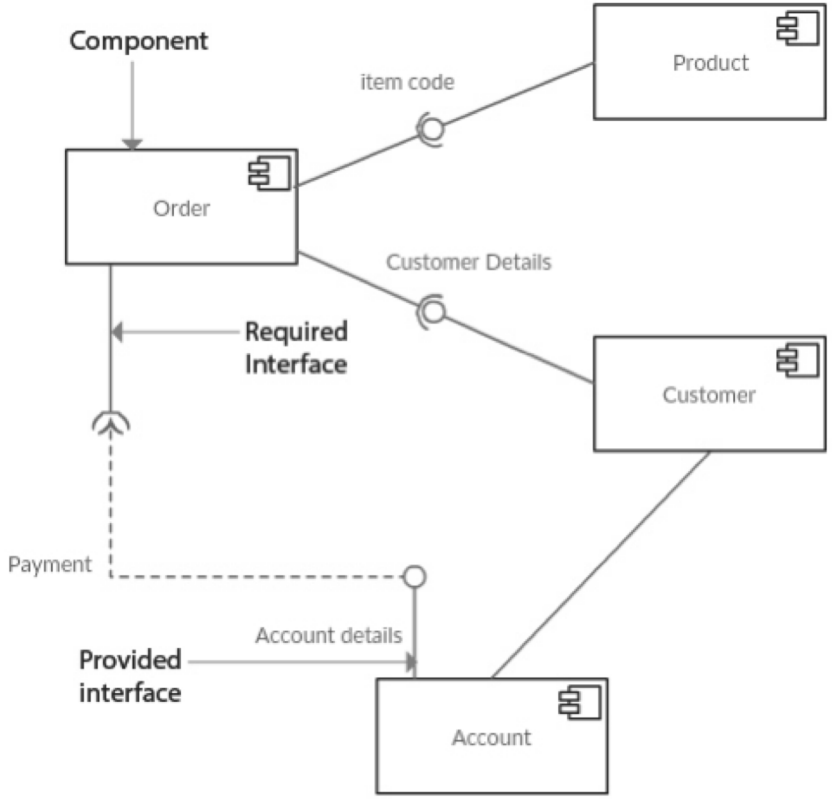
Communication Diagram

- Offers the same information as a sequence diagram, *but while a sequence diagram emphasizes the time and order of events*, a communication diagram emphasizes the messages exchanged between objects in an application.



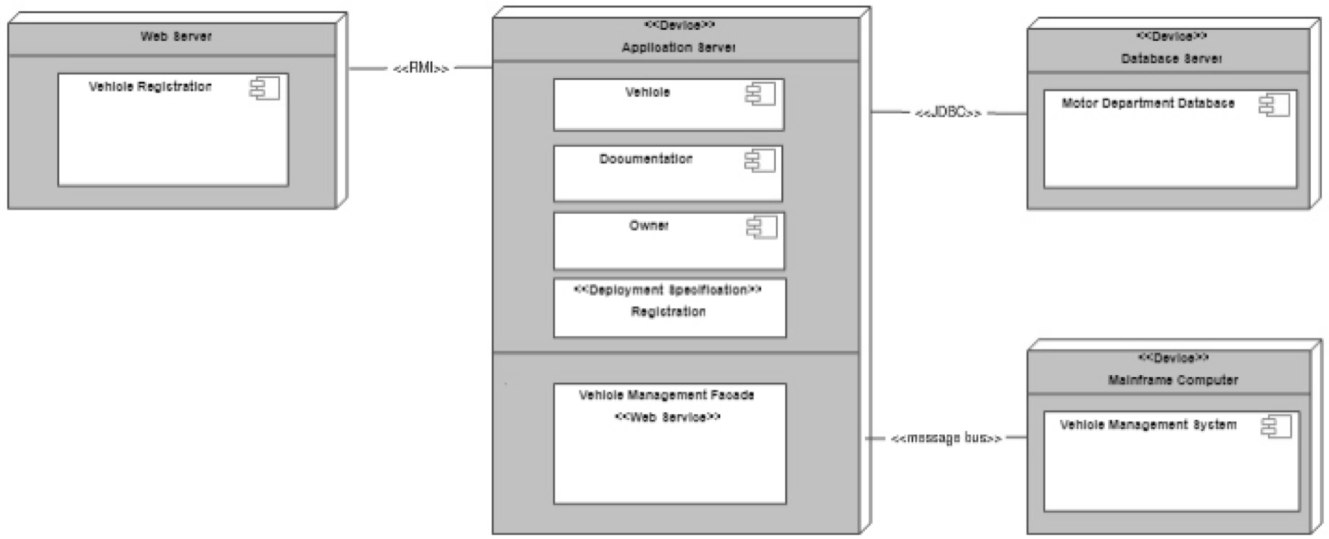
Component Diagram

- The purpose of a component diagram is to show the relationship between different components in a system.
  - In UML 2.0, the term "component" refers to a module of classes that represent independent systems or subsystems with the ability to interface with the rest of the system.



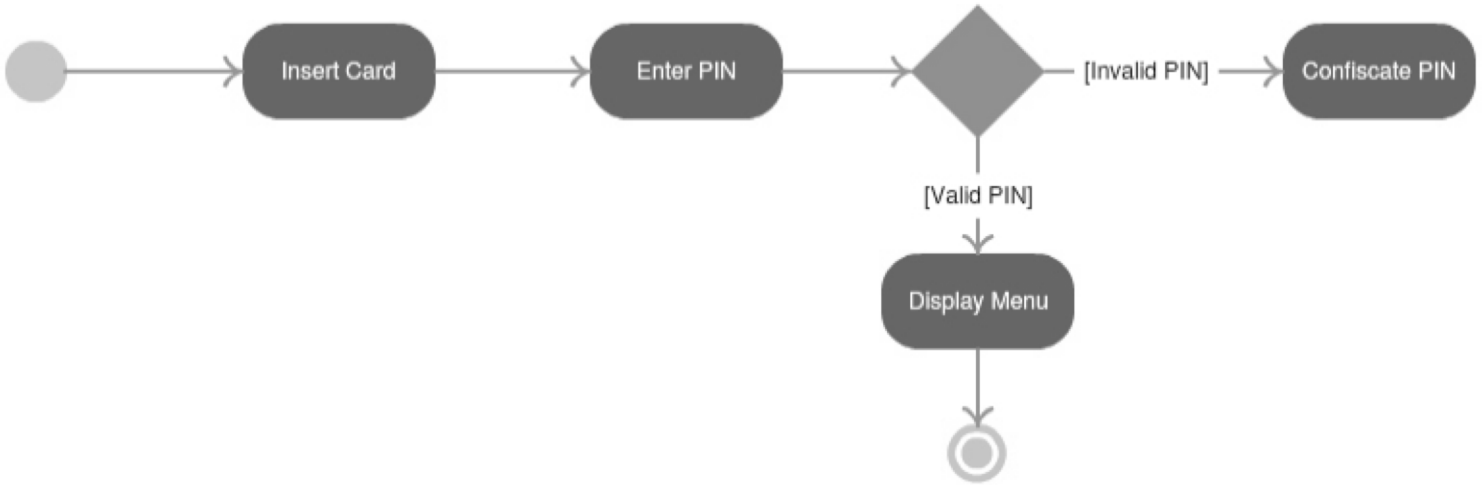
Deployment Diagram

- Describes the physical deployment of information generated by the software program on hardware components.
  - For example, for a web site:
    - hardware components or ‘nodes’ (e.g., a web server, an application server, and a database server)
    - software components or ‘artifacts’ to run on each node (e.g., web application, database)
    - how the different pieces are ‘connected’ (e.g. JDBC, REST, RMI).



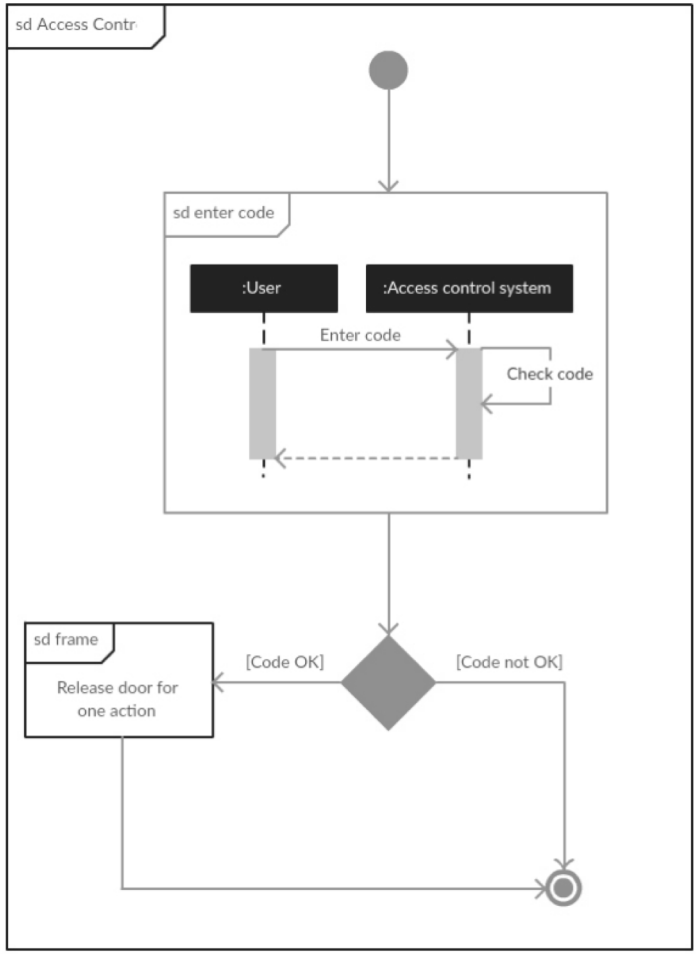
Activity Diagram

- Describes what must happen in the system being modelled.



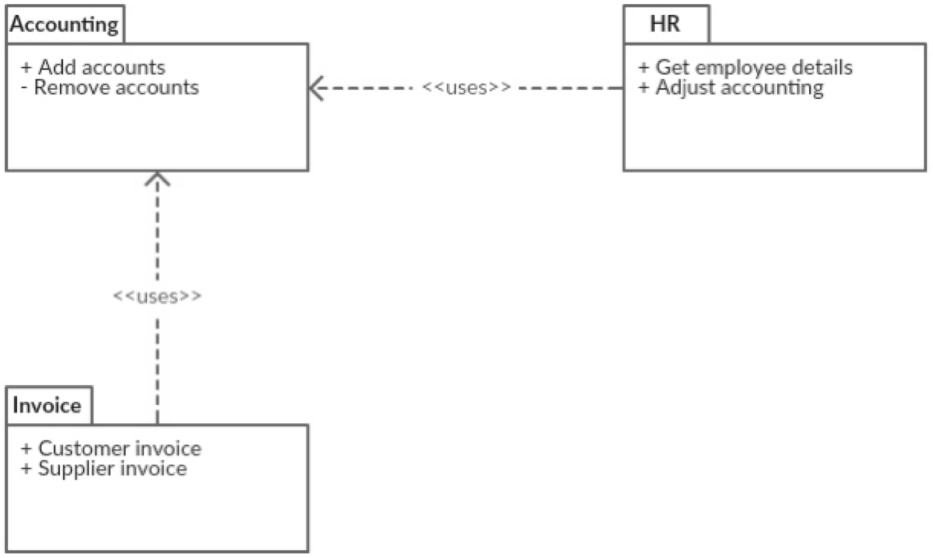
Interaction Overview Diagram

- Used to depict a control flow with nodes that can contain interaction diagrams.
  - It is similar to the activity diagram, i.e., both visualize a sequence of activities, however, in an interaction overview diagram, each individual activity is pictured as a frame which can contain a nested interaction diagram.



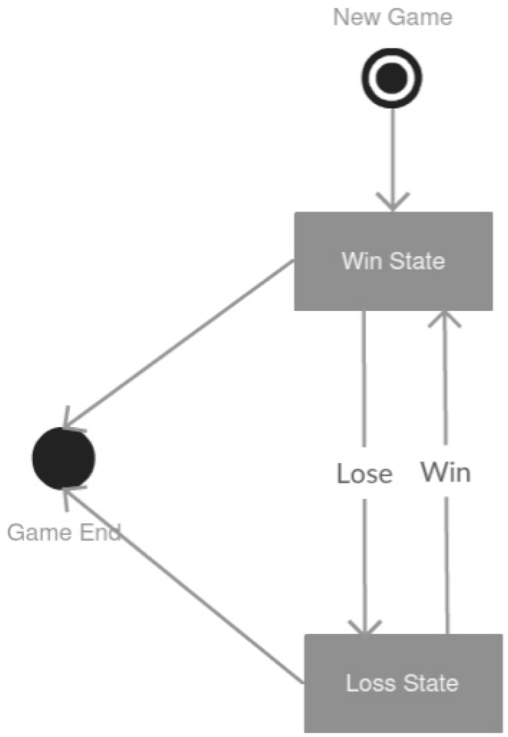
Package Diagram

- Shows packages and dependencies between the packages.



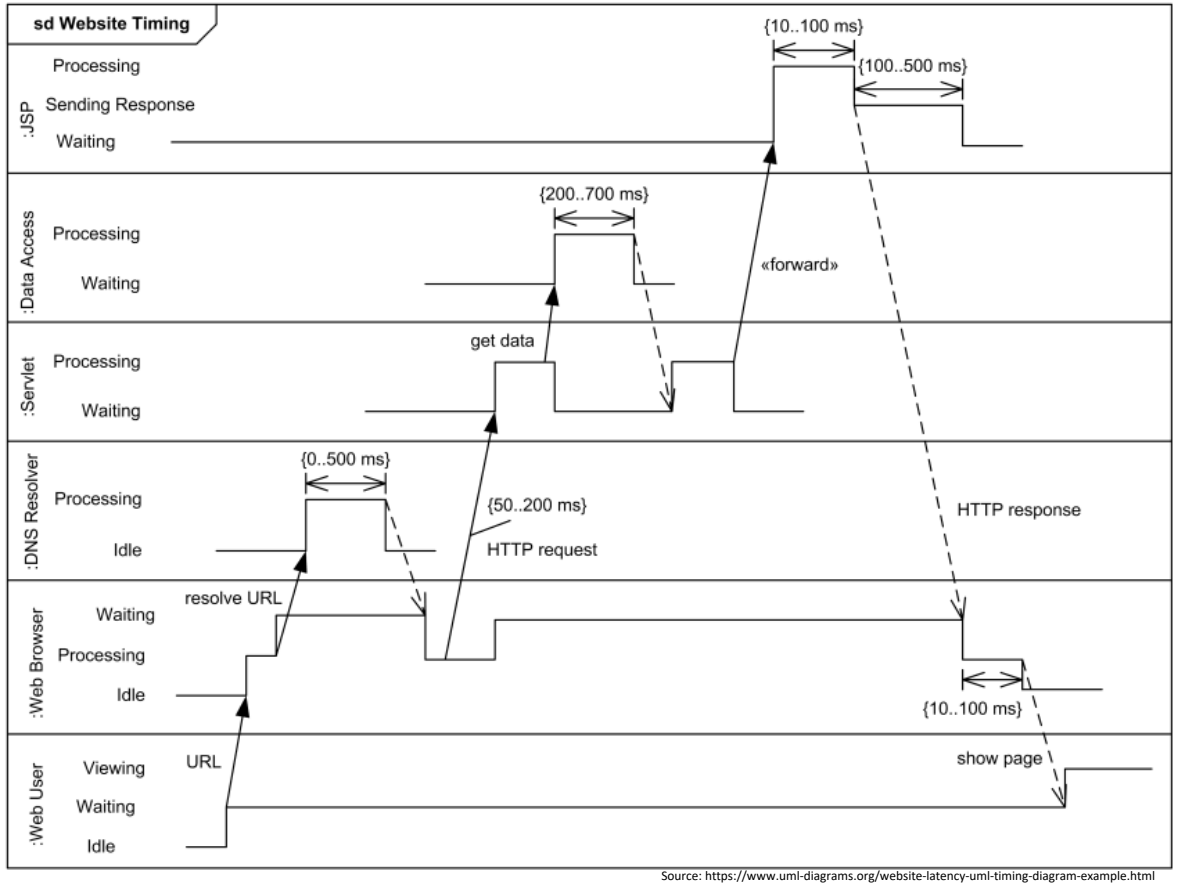
State Machine Diagram

- Shows transitions between various objects.
  - describe the behavior of objects that act according to the state they are in at the moment.



Timing Diagram

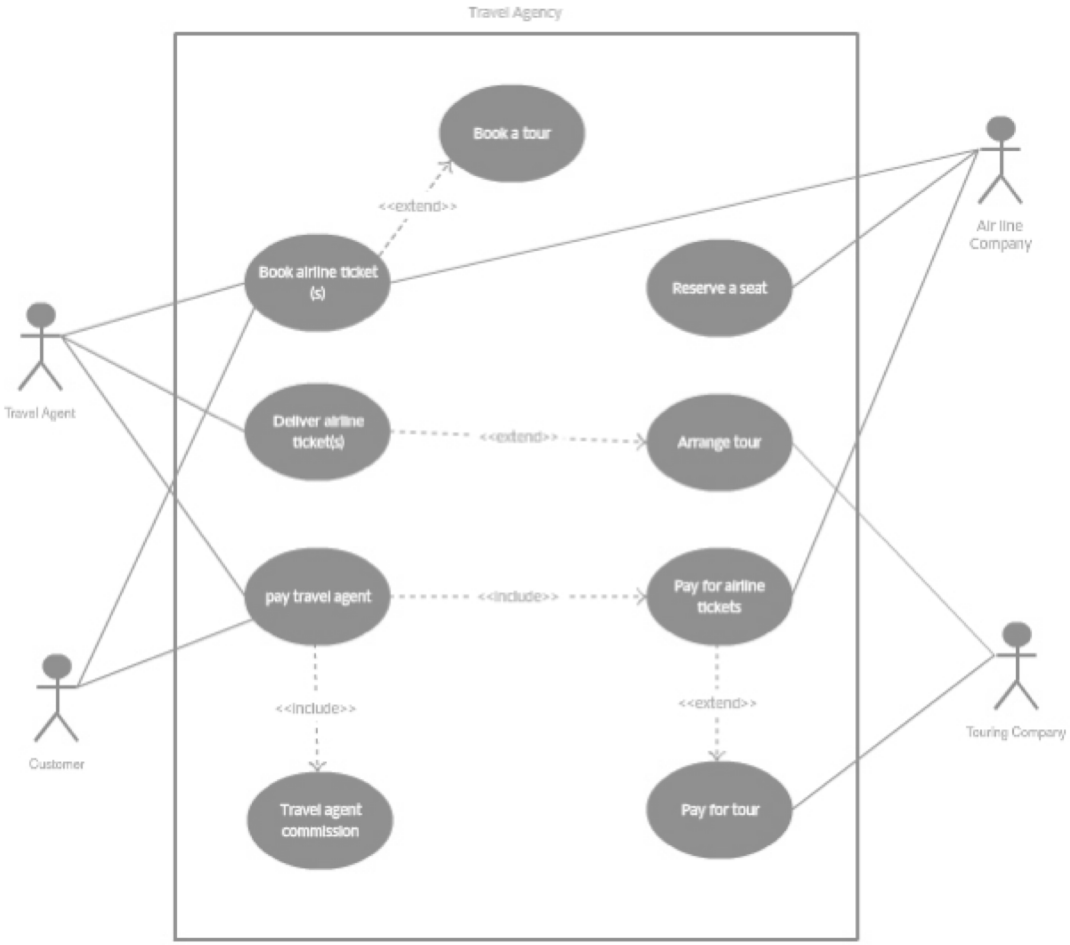
- Focuses on timing constraints. It is useful to know how objects interact with each other during a certain period of time.



Source: https://www.uml-diagrams.org/website-latency-uml-timing-diagram-example.html

Use Case Diagram

- Summarizes the details of your system's users (i.e., actors) and their interactions with the system.



UML Basics

- A *model* captures an interrelated set of information about the system
  - *A diagram is simply one view of that information.*
- **Model *vs.* Diagram?**
- Class diagrams describe the data found in a software system.
- Essentials of UML class diagrams:
  - Classes represent the types of data themselves.
  - Associations show how instances of classes reference instances of other classes.
  - Attributes are simple data found in instances.
  - Operations represent the functions performed by the instances.
  - Generalizations are used to arrange classes into inheritance hierarchies.

Types of Models

- **Exploratory domain models** represent what you have learned about the various entities and relationships in the domain; they help in understanding that domain.

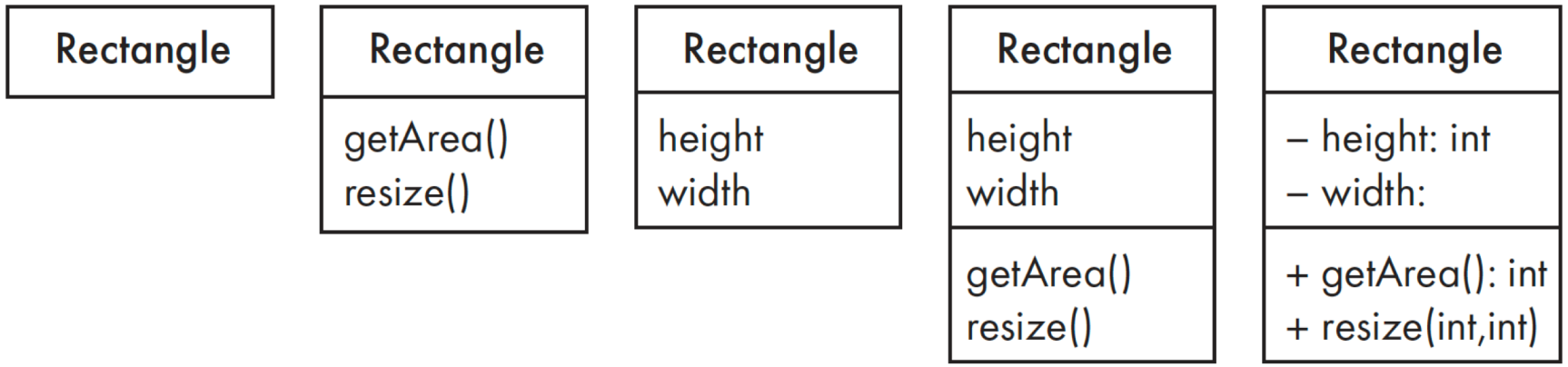
- **System domain models** are developed during requirements analysis or the early stages of design that also contains domain classes, associations and attributes.
  - Represent data that will actually be manipulated and stored by the system.

Suggested Sequence of Activities for Modelling

1. Identify a first set of candidate classes.
2. Starting with the most important classes, add any associations and attributes.
3. Work out the clearest generalizations.
4. List the main responsibilities of each class.
5. Based on responsibilities, decide on specific operations that are needed.
6. Review the entire process to examine the model and see if it needs to add/delete classes, associations, attributes, generalizations, responsibilities or operations. Identify Design Patterns and Interfaces.
7. Repeat Step 6 as needed until the model is satisfactory.

UML Basics: Classes

- A class can be at different levels of detail depending on the phase of development and on what you wish to communicate.

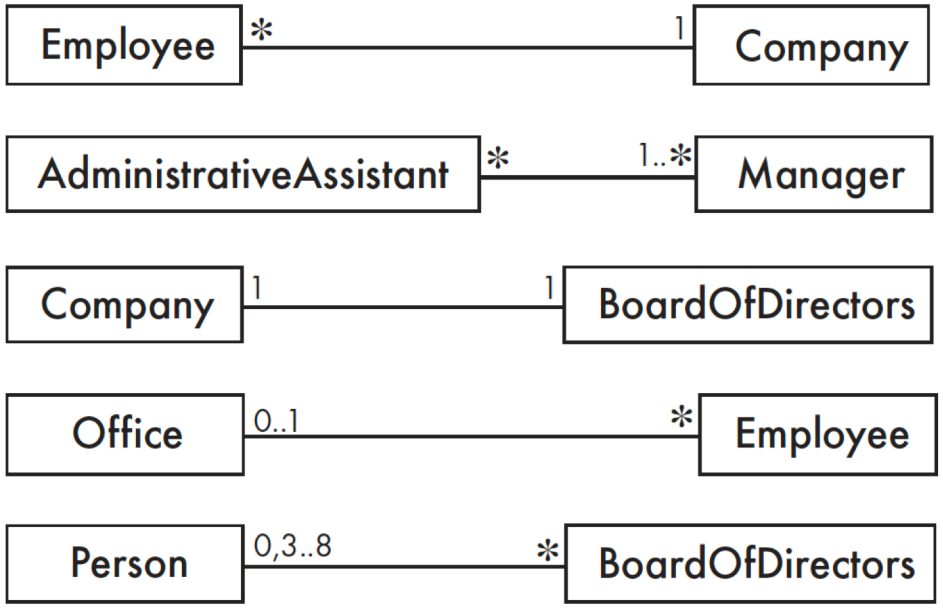


- A detailed operation's signature:

accessModifier operationName(parameterName: parameterType,...): returnType

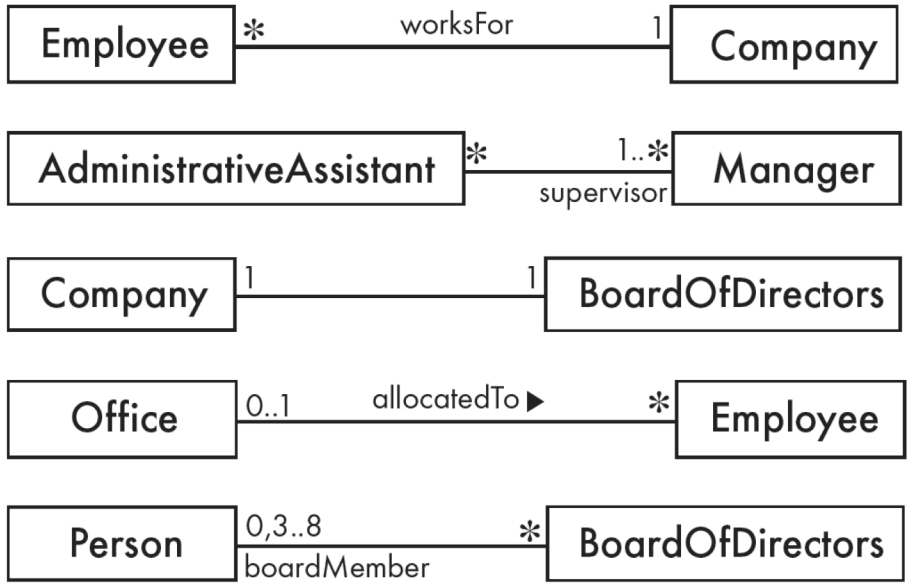
UML Basics: Associations and Multiplicity

- An association, drawn as a line between the classes, shows how instances of two classes will reference each other.
- Symbols indicating multiplicity are shown at each end of the association.



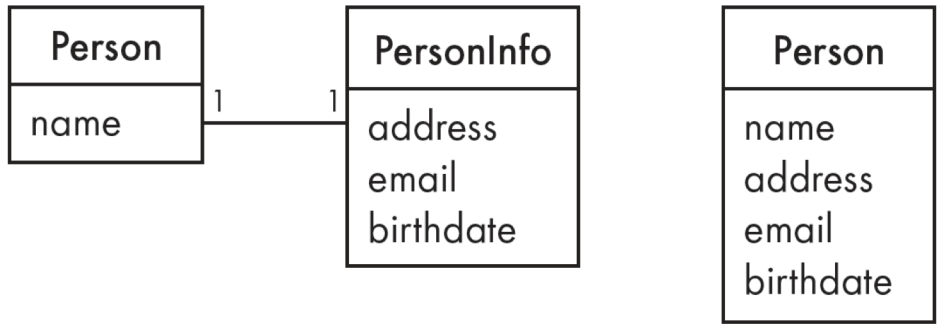
UML Basics: Labelling Associations

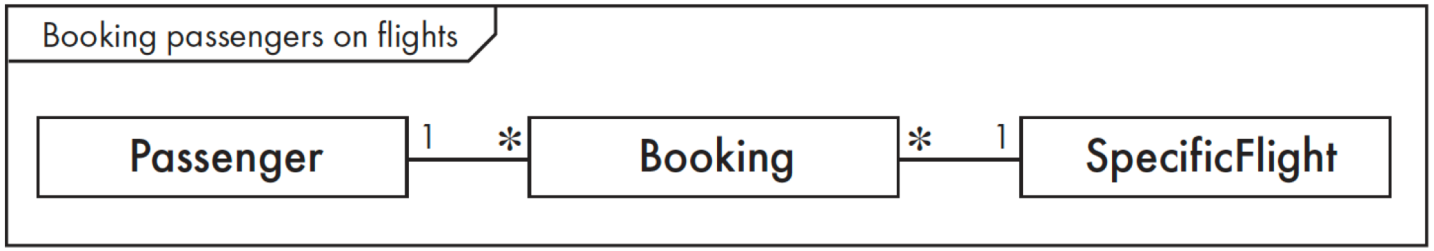
- Mark each association with the nature of the association.
  - Two types of labels: association names and role names.
  - An association name should be a verb or verb phrase.
  - If omitted, an association's name is simply ***has,*** by default.



UML Basics: Analyzing and Validating Associations

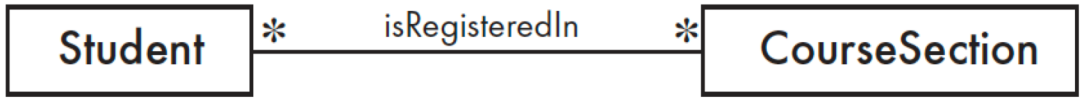
- It is very common to make errors when creating associations.
- Three of the most common patterns of multiplicity: (i) One-to-many, (ii) Many-to-many, and (iii) One-to-one



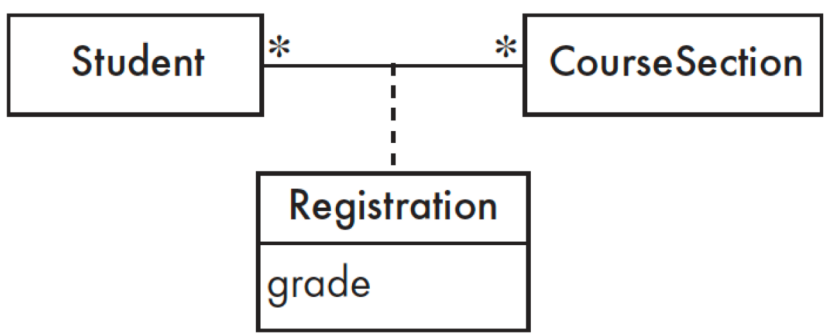


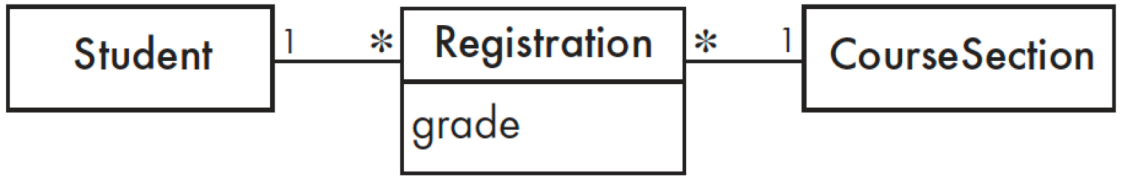
UML Basics: Association Classes

- In some cases, an attribute that concerns two associated classes cannot be placed in either of the classes.



- **The solution: create an association class.**

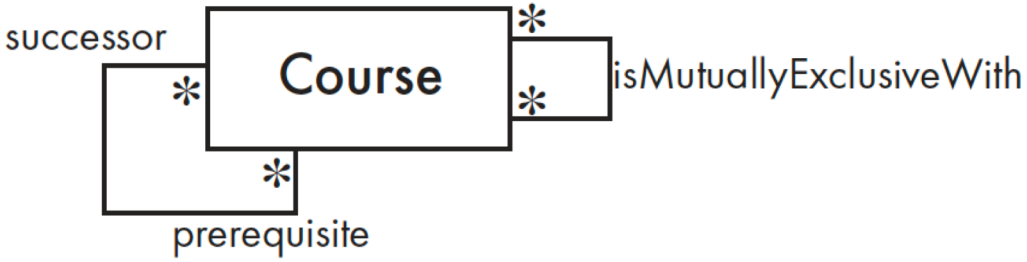




- An association class can be named using a noun that reflects the meaning of the association.

- *Tips: Check for many-to-many associations if you can replace with an association class.*

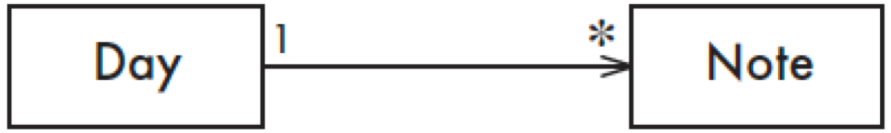
UML Basics: Reflexive Associations



- It is possible for an association to connect a class to itself.
- The *asymmetric* and *symmetric* associations.
- *Tips: Label an asymmetric reflexive association using role names instead of an association name.*

UML Basics: Directionality in Associations

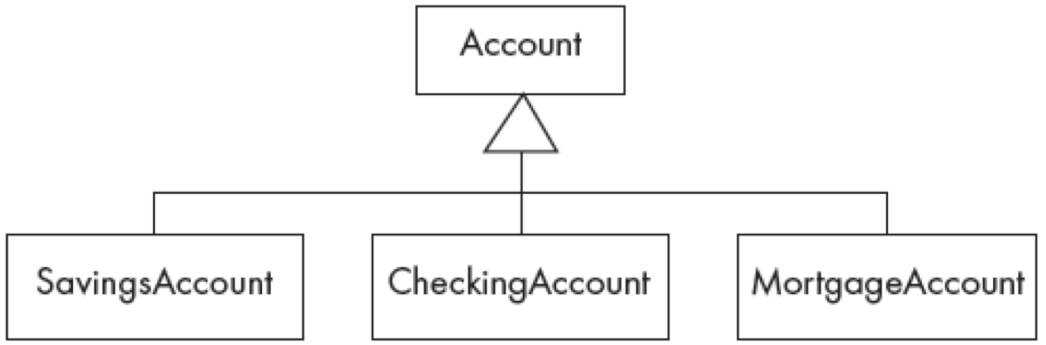
- Associations and links are by default bi-directional.



- *Note: Making associations uni-directional can improve efficiency and reduce complexity, but limit the flexibility of the system.*

UML Basics: Generalizations

- **Generalization**: The relationship between a subclass (derived class) and an immediate superclass (base class).



- A hierarchy with one or more generalizations is called an *inheritance hierarchy* or a *generalization hierarchy*.

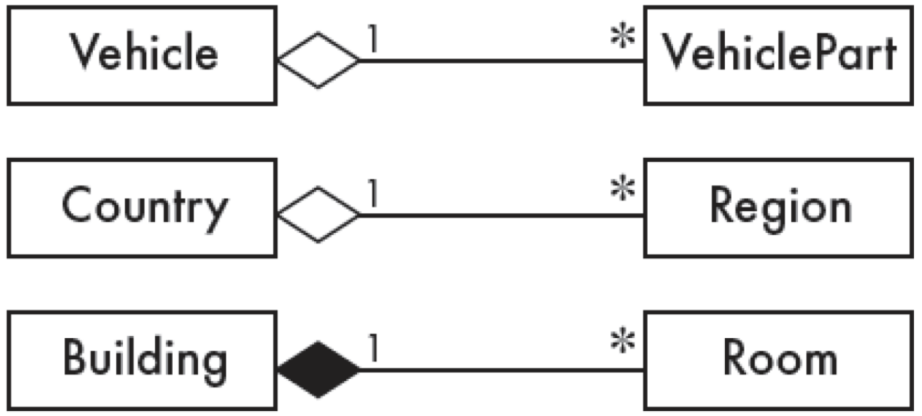
UML Basics: Associations *vs.* Generalizations

An **association** describes a relationship that will exist between instances at runtime.	A **generalization** describes a relationship between classes in a class diagram.

- An object diagram can *never* contain a generalization.

UML Basics: Aggregation

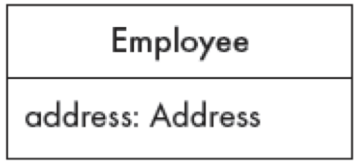
- Aggregations are special associations that represent '*part–whole*' relationships.
- The 'whole' side of the relationship is often called the *aggregate*.
- Aggregations are specified using a *diamond* symbol, which is placed next to the aggregate.
- In most cases, aggregations are *one-to-many*.

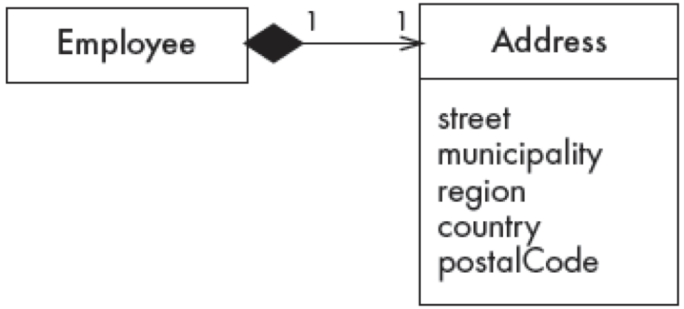


UML Basics: Composition

- Composition = **(Strong)** Aggregation
- If the aggregate is destroyed, then the parts are destroyed as well.
  - The parts of a composition can never have a life of their own; they exist only to serve the aggregate.

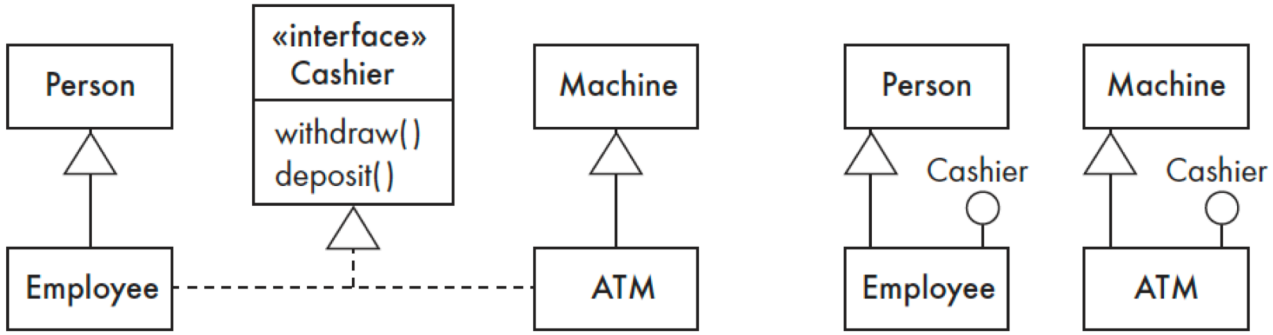
- A one-to-one composition often corresponds to a complex attribute.





UML Basics: Interfaces

- An interface describes a portion of the visible behavior of a set of objects.
  - The abstract methods and/or class variables.



- The «interface» notation is an example of a *stereotype* in UML.
- **Difference between generalization and interface?**
  - Generalization → 'is-a' relation between subclass and superclass
  - Interface → 'can-be-seen-as' relation between the implementing class and the interface

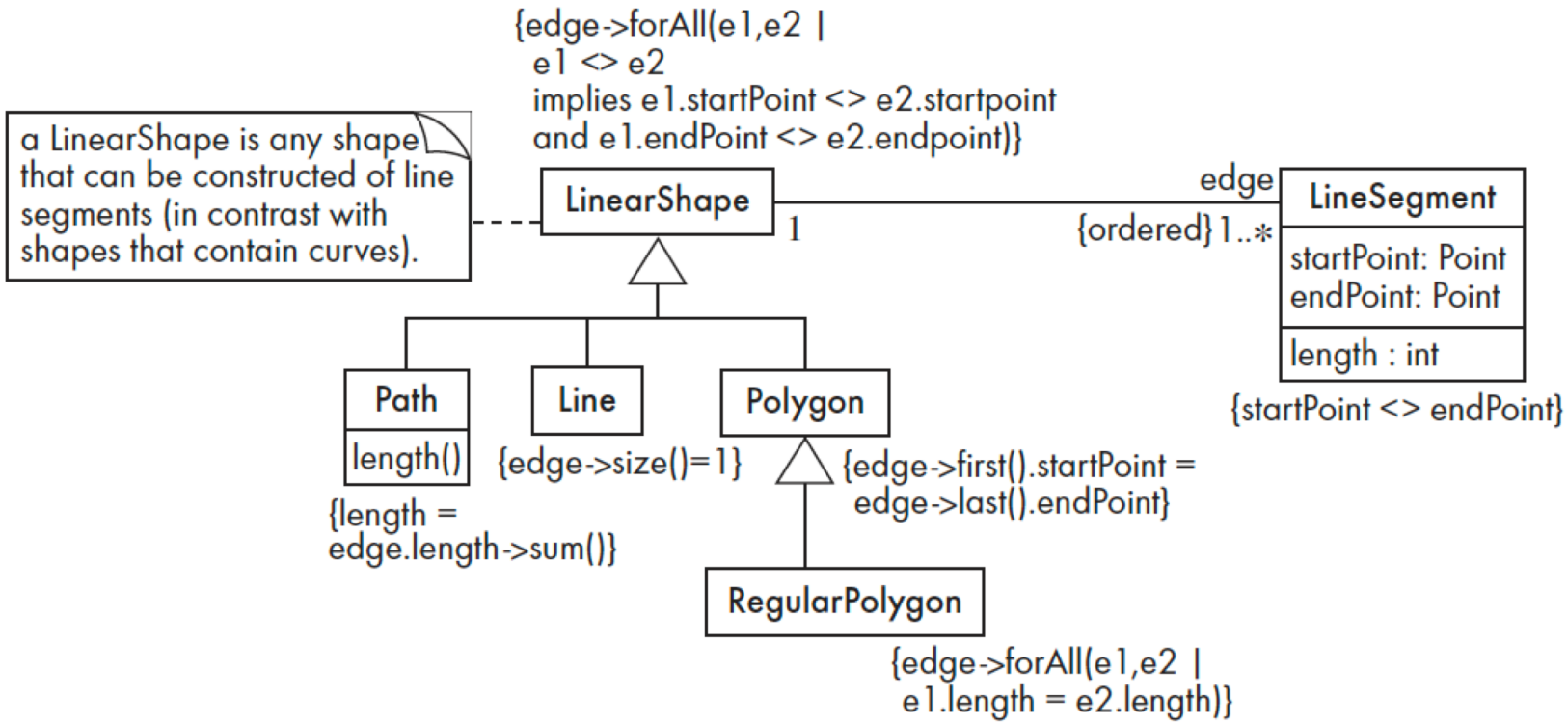
UML Basics: Describing the UML

- **Descriptive text and other diagrams**
  - Explain aspects of the system using any notation
  - Do not repeat what is shown in the UML diagrams
  - Rationale over design decisions

- **Notes**
  - Small block of text embedded in a UML diagram

- **Constraints**
  - A note written in a formal language that can be interpreted by a computer
  - A logical statement that should evaluate to true
  - Recommended language is Object Constraint Language (OCL)

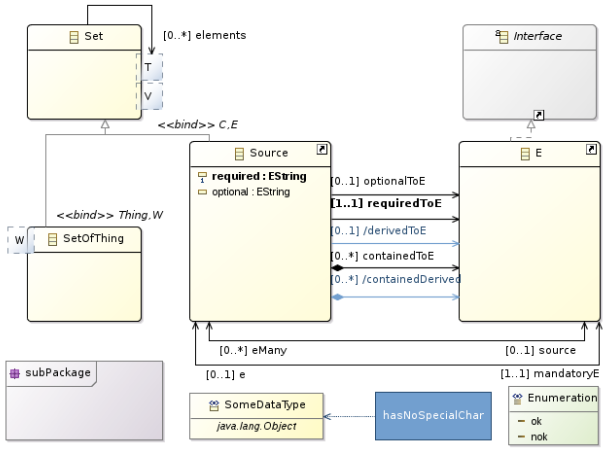
UML Basics: Describing the UML



UML Modelling using Eclipse Ecore

- Ecore: https://wiki.eclipse.org/Ecore

- The core EMF includes a metamodel (Ecore)
  - Describes models and provides runtime support
  - Allows manipulating EMF objects generically



Ecore Model

Java Code

Code generation

EPackage
EClass
EAttribute
EReference

Package
Class
Attribute
Reference

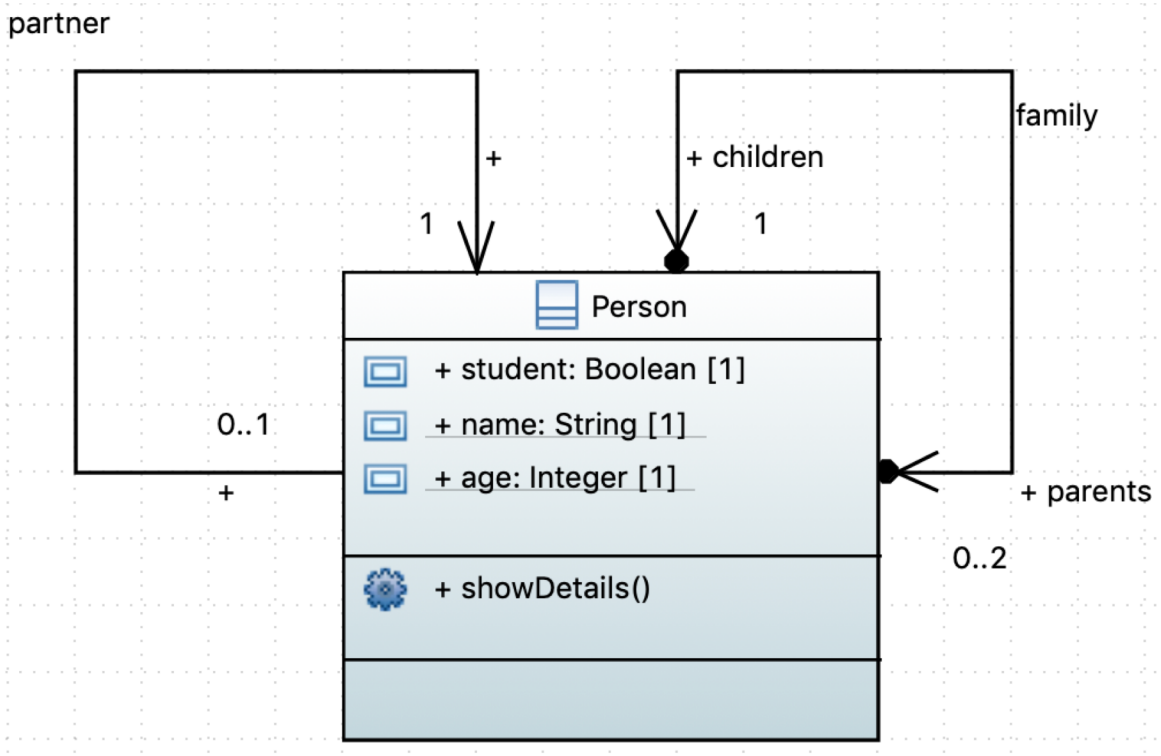
Object Constraint Language (OCL)

- http://www.omg.org/spec/OCL/2.4
- OCL is a formal language developed by IBM to describe expressions on UML models to specify *invariants* that must hold for the system or to *query over objects* in a model
- **Why OCL?**
  - A UML diagram (e.g., class) not refined enough – need to describe additional constraints on objects
  - A pure specification language (no side effects), not a programming language (no logics, control flow)
  - The evaluation of an OCL expression is instantaneous.
- **Use of OCL**
  - A query language
  - Specify invariants on classes and types in the class model
  - Describe pre- and post conditions on Operations and Methods
  - Specify constraints on operations

The Basic OCL Operators

- References to role names, association names, attributes and the results of operations
- The logical values *true* and *false*
- Logical operators → **and**, **or**, **=**, **>**, **<** or **<>**
- String values such as: 'a string'
- Integers and real numbers
- Arithmetic operations → *****, **/**, **+**, **-**

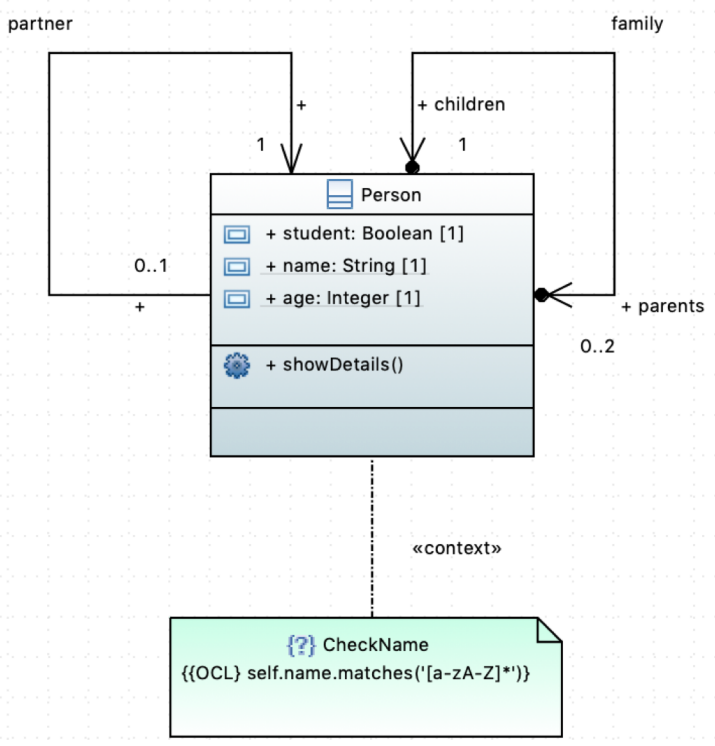
An OCL Example



OCL Example: *Scalar Constraints*

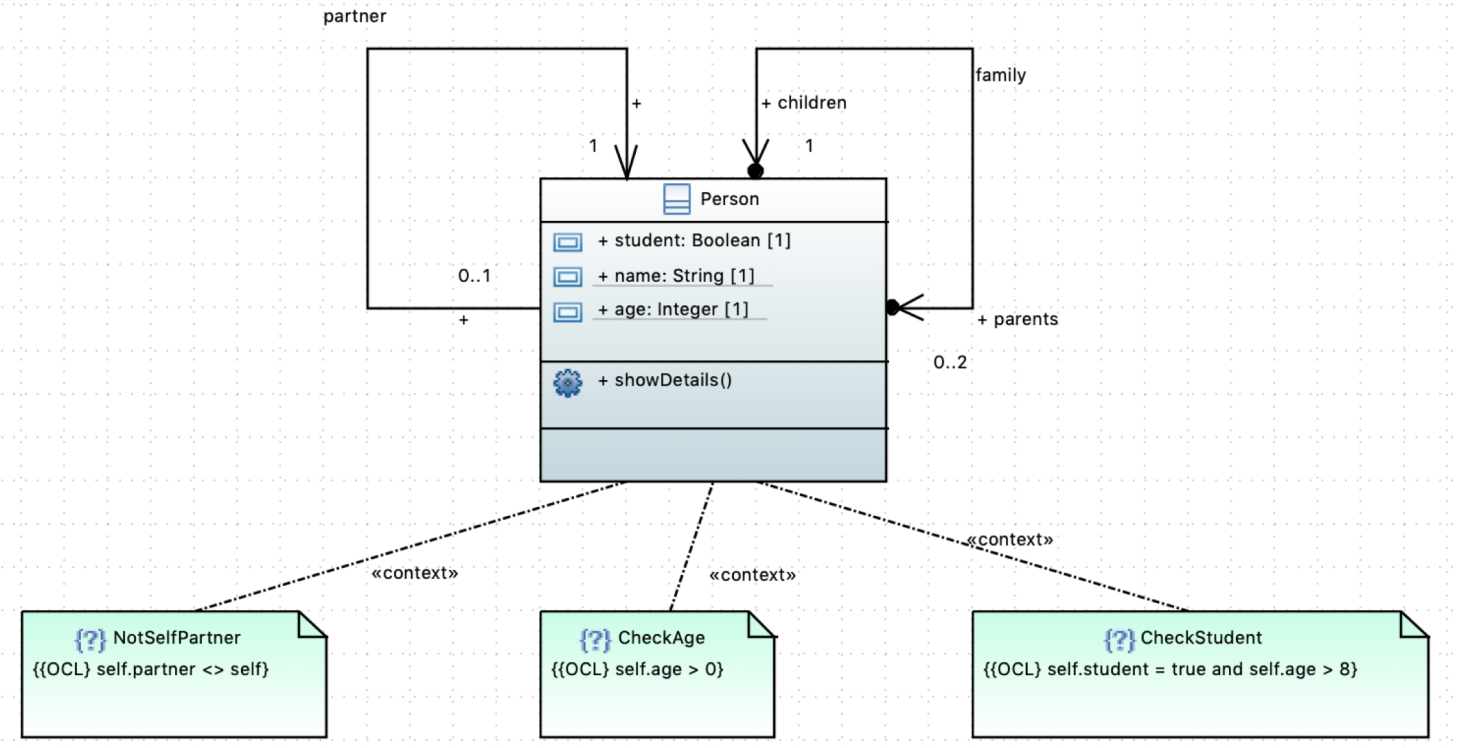
- Use a regular expression to specify that the name must consist of just alphabetic characters.

```
self.name.matches('[a-zA-Z]*')
```



OCL Example: *Scalar Constraints*

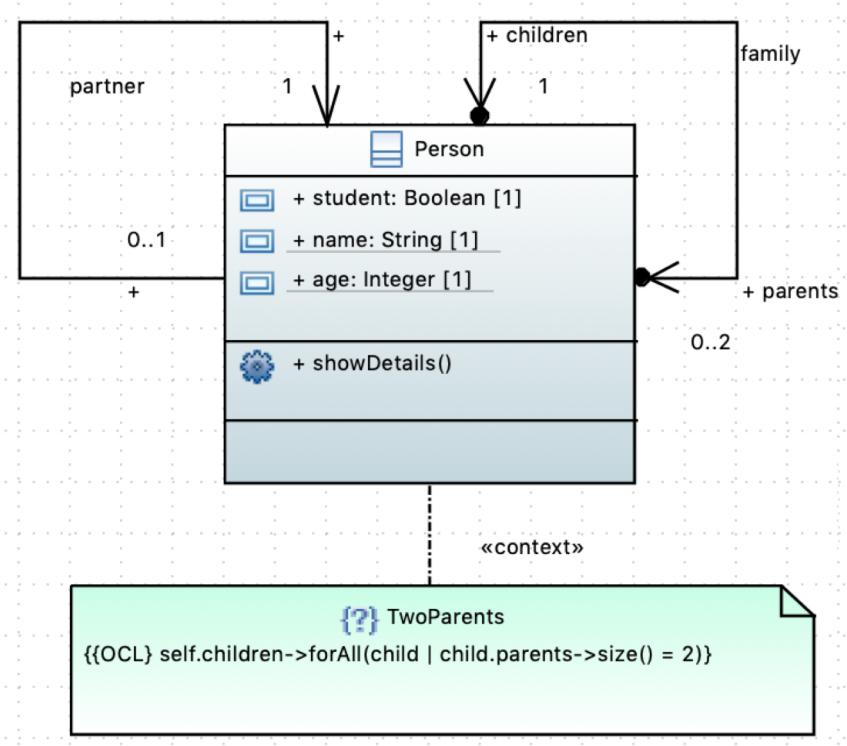
```
self.partner <> self
self.age > 0
self.student = true and self.age > 8
```



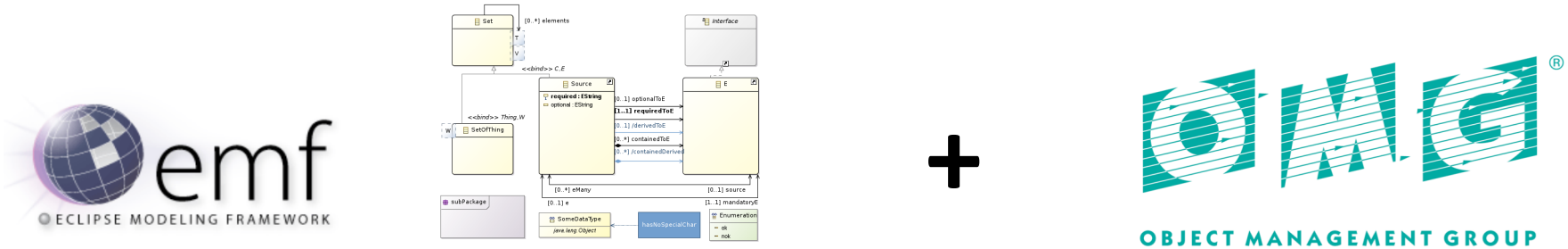
OCL Example: *Collection Constraints*

- Each child should have two parents.

```
self.children->forAll(child | child.parents->size() = 2)
```



Hands on with Ecore and OCL



Ecore + **OCL**

Model Validation
Code Generation

Objectives

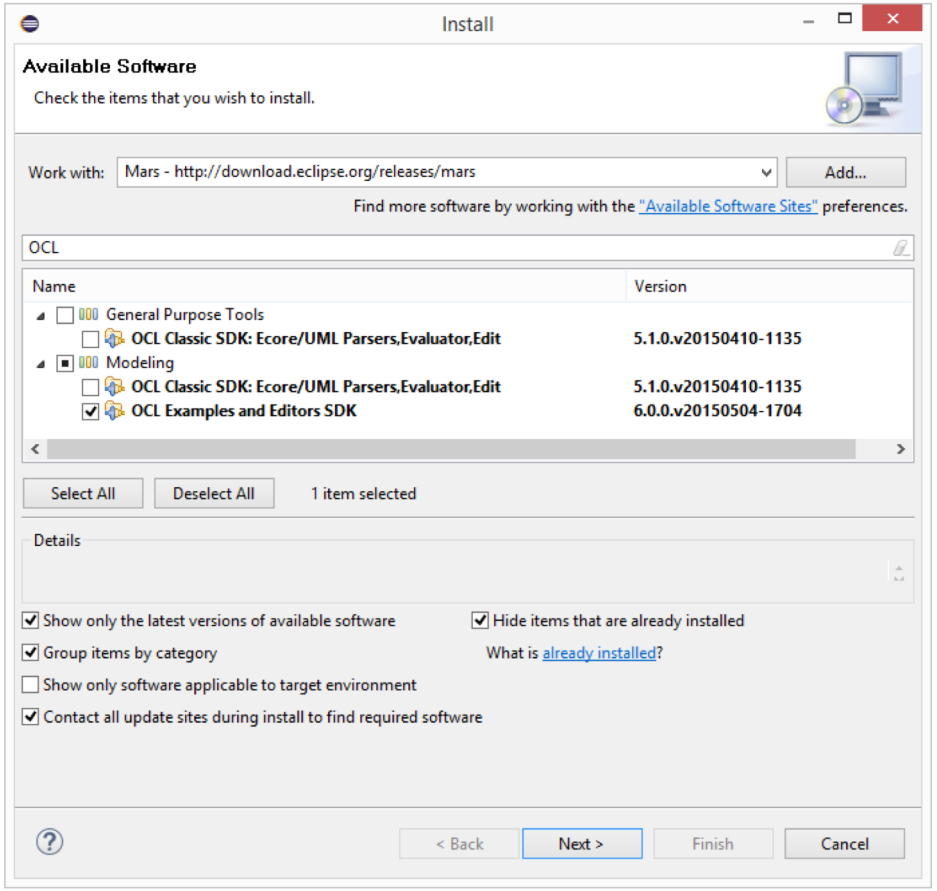
- Install OCL and the additional Editors

- Use the OCLinEcore editor
  - Create an Ecore model using the OCLinEcore text editor
  - Create a dynamic instance of that Ecore model
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

Install OCL and the additional Editors

- **Install OCL and the additional Editors**
- Use the OCLinEcore editor
  - Create an Ecore model using the OCLinEcore text editor
  - Create a dynamic instance of that Ecore model
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

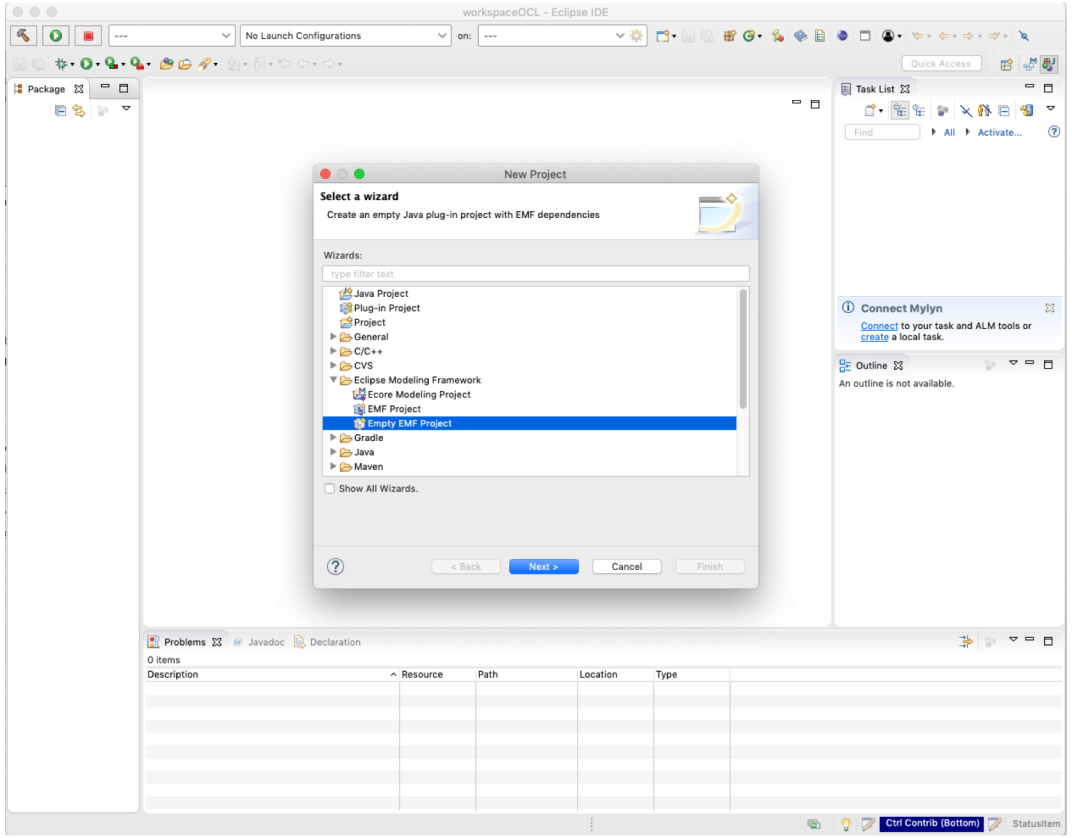
- Eclipse -> Help -> Install New Software…
- From default update site: expand **Modelling** category and select **OCL Examples and Editors SDK**
- Restart Eclipse when finished



Create an Ecore model using the OCLinEcore text editor

- Install OCL and the additional Editors
- **Use the OCLinEcore editor**
  - **Create an Ecore model using the OCLinEcore text editor**
  - Create a dynamic instance of that Ecore model
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

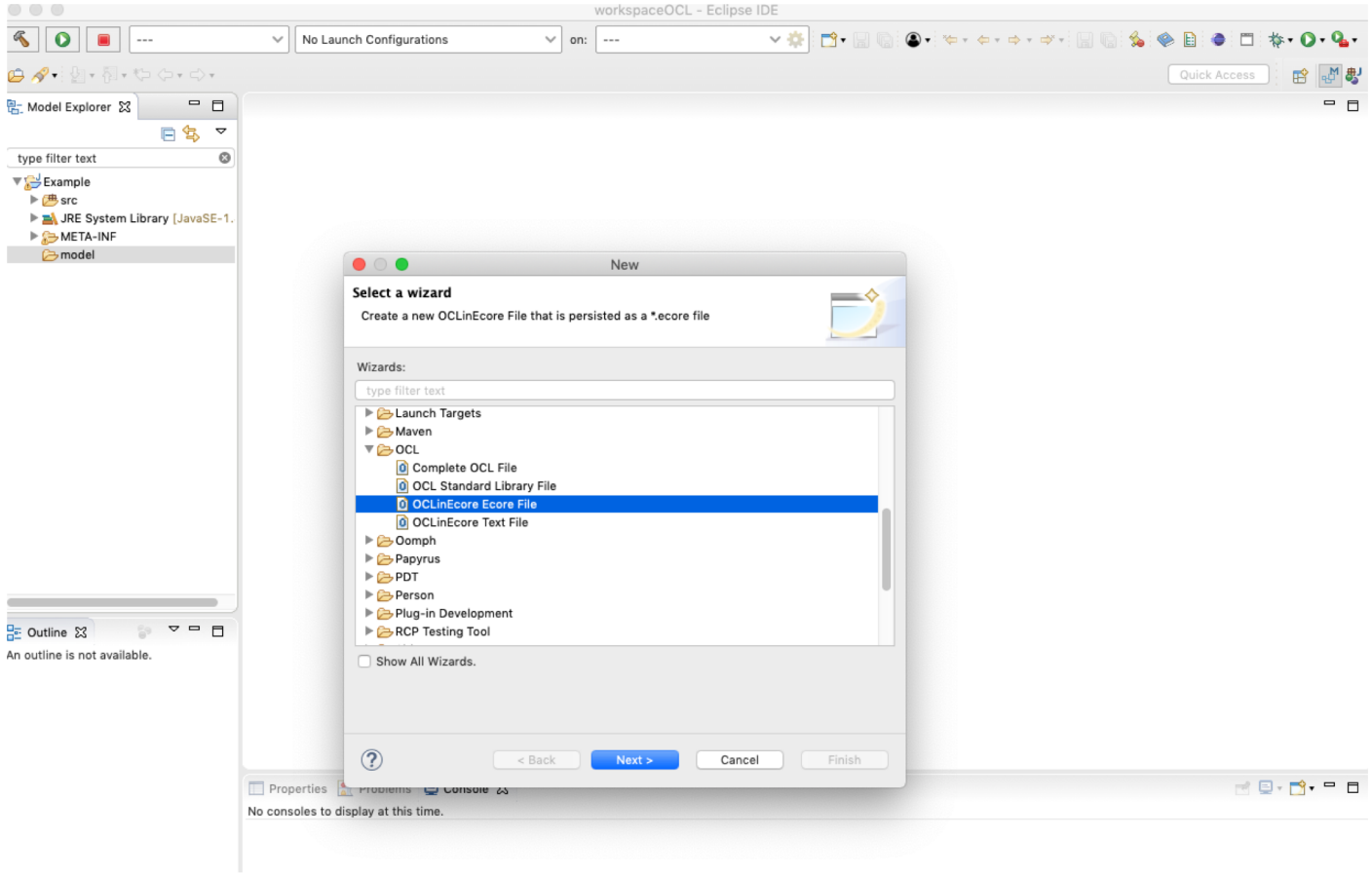
- We use the **OCLinEcore** editor that provides text editing
- From menu, select **File** -> **New** -> **Project** -> **Eclipse Modeling Framework** -> **Empty EMF Project**
- Give a project name (Library) -> **Finish**



Create an Ecore model using the OCLinEcore text editor

- Install OCL and the additional Editors
- **Use the OCLinEcore editor**
  - **Create an Ecore model using the OCLinEcore text editor**
  - Create a dynamic instance of that Ecore model
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

- Right-click on **model** folder in the **Library** project, select **New -> Other...** then select the **OCLinEcore Ecore File** from the **OCL** category.



Create an Ecore model using the OCLinEcore text editor

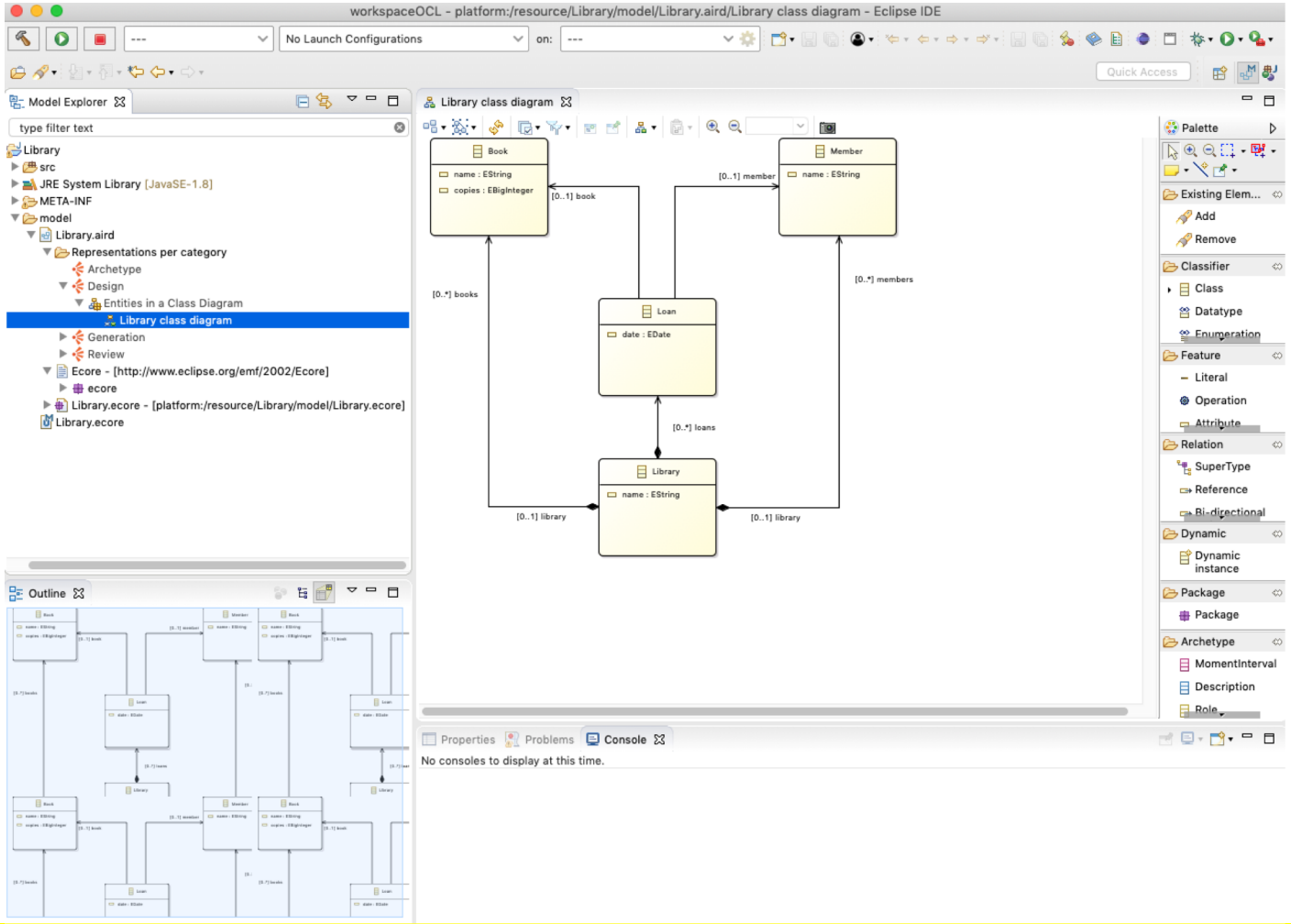
- Now open the Ecore model using the OCLinEcore text editor and provide initial content.
- Right-click on **Library.ecore**, select **Open With -> OCLinEcore Editor** and replace the content with the OCL code:

```
package Library : lib = 'http://www.eclipse.org/mdt/ocl/oclinecore/library'
{
  class Library
  {
    attribute name : String;
    property books#library : Book[*] { composes };
    property loans : Loan[*] { composes };
    property members#library : Member[*] { composes };
  }
  class Book
  {
    attribute name : String;
    attribute copies : Integer;
    property library#books : Library[?];
  }
  class Member
  {
    attribute name : String;
    property library#members : Library[?];
  }
  class Loan
  {
    property book : Book;
    property member : Member;
    attribute date : ecore::EDate[?];
  }
}
```

Create an Ecore model using the OCLinEcore text editor

- Install OCL and the additional Editors
- **Use the OCLinEcore editor**
  - **Create an Ecore model using the OCLinEcore text editor**
  - Create a dynamic instance of that Ecore model
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

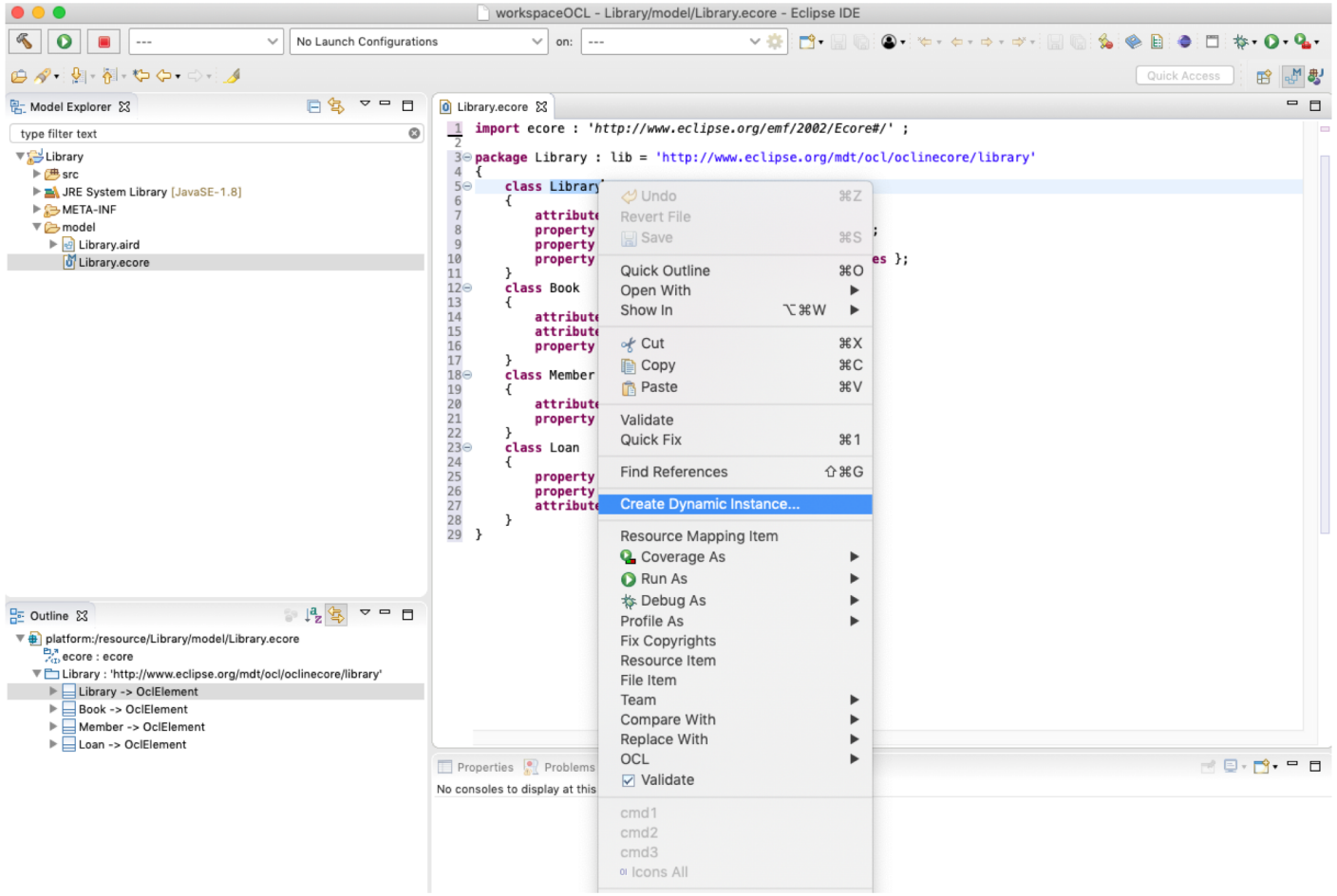
- We generate the visual representation of the model as **Library.aird**



Create a dynamic instance of that Ecore model

- Install OCL and the additional Editors
- **Use the OCLinEcore editor**
  - Create an Ecore model using the OCLinEcore text editor
  - **Create a dynamic instance of that Ecore model**
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

- In the **OCLinEcore** Editor view, select the class **Library** then Right-click on it, select **Create Dynamic Instance…**
- This will create an instance of the Library.ecore metamodel called **Library.xmi** in **model** folder.

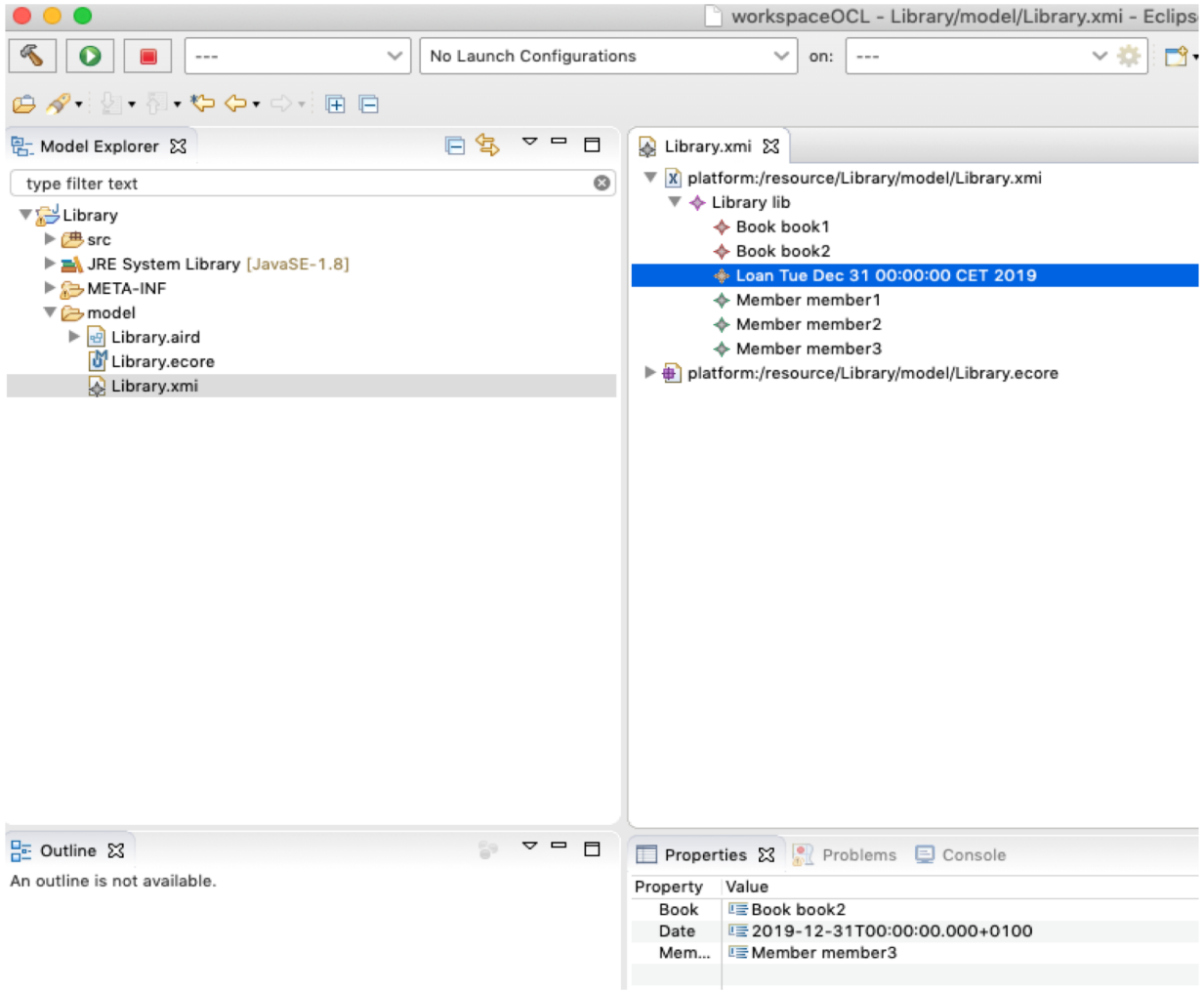


Create a dynamic instance of that Ecore model

- Install OCL and the additional Editors
- **Use the OCLinEcore editor**
  - Create an Ecore model using the OCLinEcore text editor
  - **Create a dynamic instance of that Ecore model**
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - Validate the model and observe the OCL enrichments
  - Use the Interactive OCL Console to execute the OCL enrichments

- We populate the model elements

- From the right-click menu of the **Library** model instance select **New Child** -> **Books Book** twice, then **New Child** -> **Loans Loan** once and then **New Child** -> **Members Member** three times.
- Name the elements.
- Specify that the Loan is for book2 by member3.
- Validate the **Library** model instance.



Enrich the Ecore model with OCL using the OCLinEcore text editor

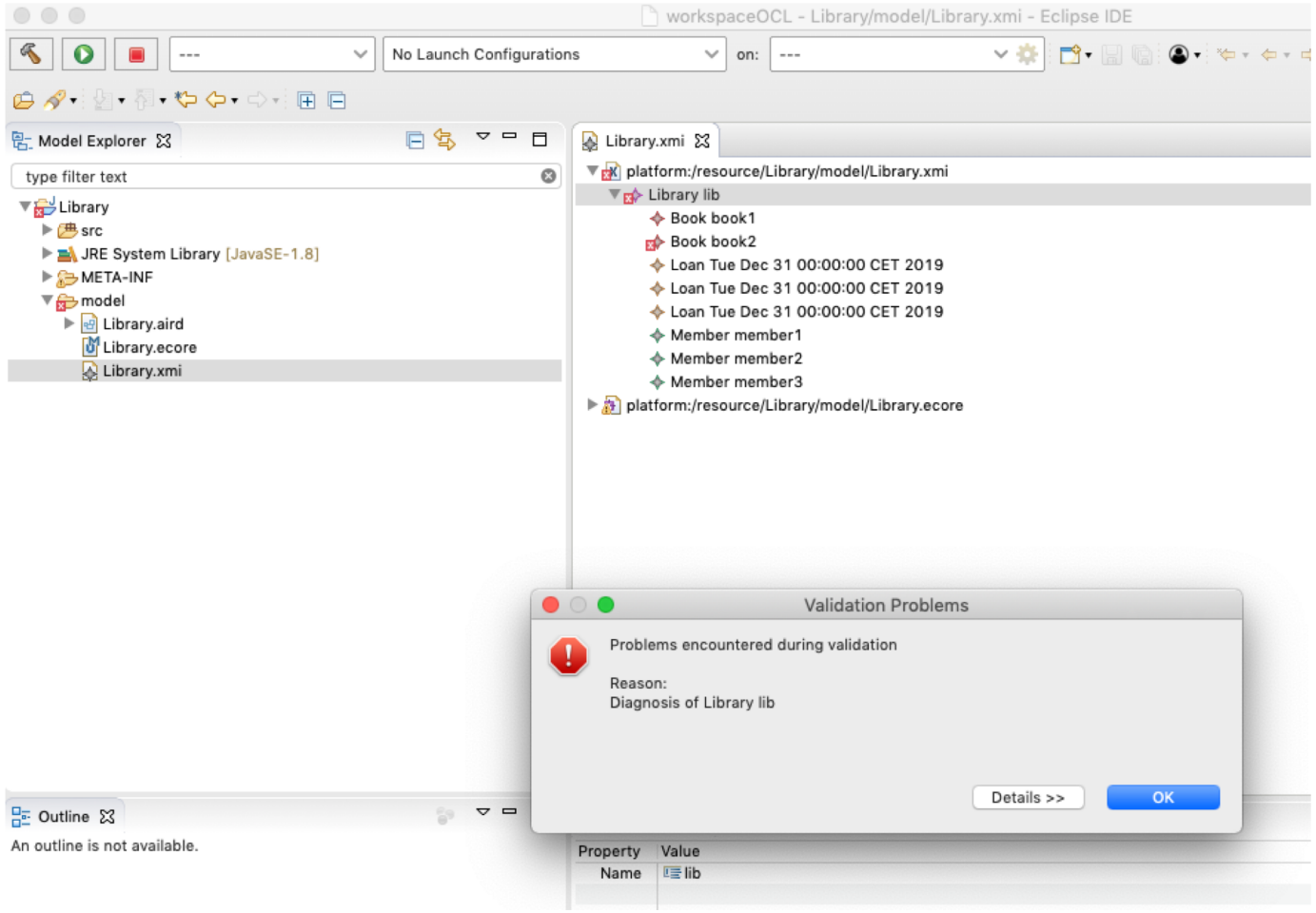
- Modify the Book class in the original meta-model so that the condition holds in the instance models.
- The new Book class in the **Library.ecore** will look like below:

```
class Book
{
        invariant SufficientCopies:
                library.loans->select(book=self)->size() <= self.copies;
        attribute name : String[?];
        attribute copies : Integer[?];
        property library#books : Library[?];
}
```

Validate the model and observe the OCL enrichments

- Install OCL and the additional Editors
- **Use the OCLinEcore editor**
  - Create an Ecore model using the OCLinEcore text editor
  - Create a dynamic instance of that Ecore model
  - Enrich the Ecore model with OCL using the OCLinEcore text editor
  - **Validate the model and observe the OCL enrichments**
  - Use the Interactive OCL Console to execute the OCL enrichments

- Validate the **Library.xmi** instance model and it will **fail**



Use the Interactive OCL Console to execute the OCL enrichments

- Make the OCL Console visible. Go to **Window -> Show View -> Console**, right-click on **Open Console** and choose the **Interactive Xtext OCL**.

- The inputs and outputs are like below:

```
Evaluating:
books
Results:
Library lib::Book book2
Library lib::Book book1

Evaluating:
members
Results:
Library lib::Member member2
Library lib::Member member1
Library lib::Member member3

Evaluating:
loans
Results:
Library lib::Loan Tue Dec 31 00:00:00 CET 2019
Library lib::Loan Tue Dec 31 00:00:00 CET 2019
Library lib::Loan Tue Dec 31 00:00:00 CET 2019
```

We can see the number of loans for each Book by selecting them one by one and executing the following in the OCL Console:

```
library.loans->select(book=self)->size()
```

Helper Features and Operations

- We can use helper attributes and operations to make the OCL clearer and provide a richer meta-model.
- Replace the **Book** class in the **Library.ecore** meta-model by the following:

```
class Book
{
        attribute name : String[?];
        attribute copies : Integer[?];
        property library#books : Library[?];
        property loans : Loan[*] { derived,volatile }
        {
                derivation: library.loans->select(book=self);
        }
        operation isAvailable() : Boolean[?]
        {
                body: loans->size() < copies;
        }
        invariant SufficientCopies:
                library.loans->select(book=self)->size() <= self.copies;
}
```

- If you open the **Library.xmi** model instance, it will be reloaded with the derived property

Use the Interactive OCL Console to execute the OCL enrichments

- Again make the OCL Console visible. Go to **Window -> Show View -> Console**, right-click on **Open Console** and choose the **Interactive Xtext OCL**.

- The helper operation can be evaluated in the **OCL Console** view:
  - Select Book book2 and type **isAvailable()** for execution.
  - This will return **false** since two copies of book2 are already in load, so book2 is not available now.

Helper Features and Operations

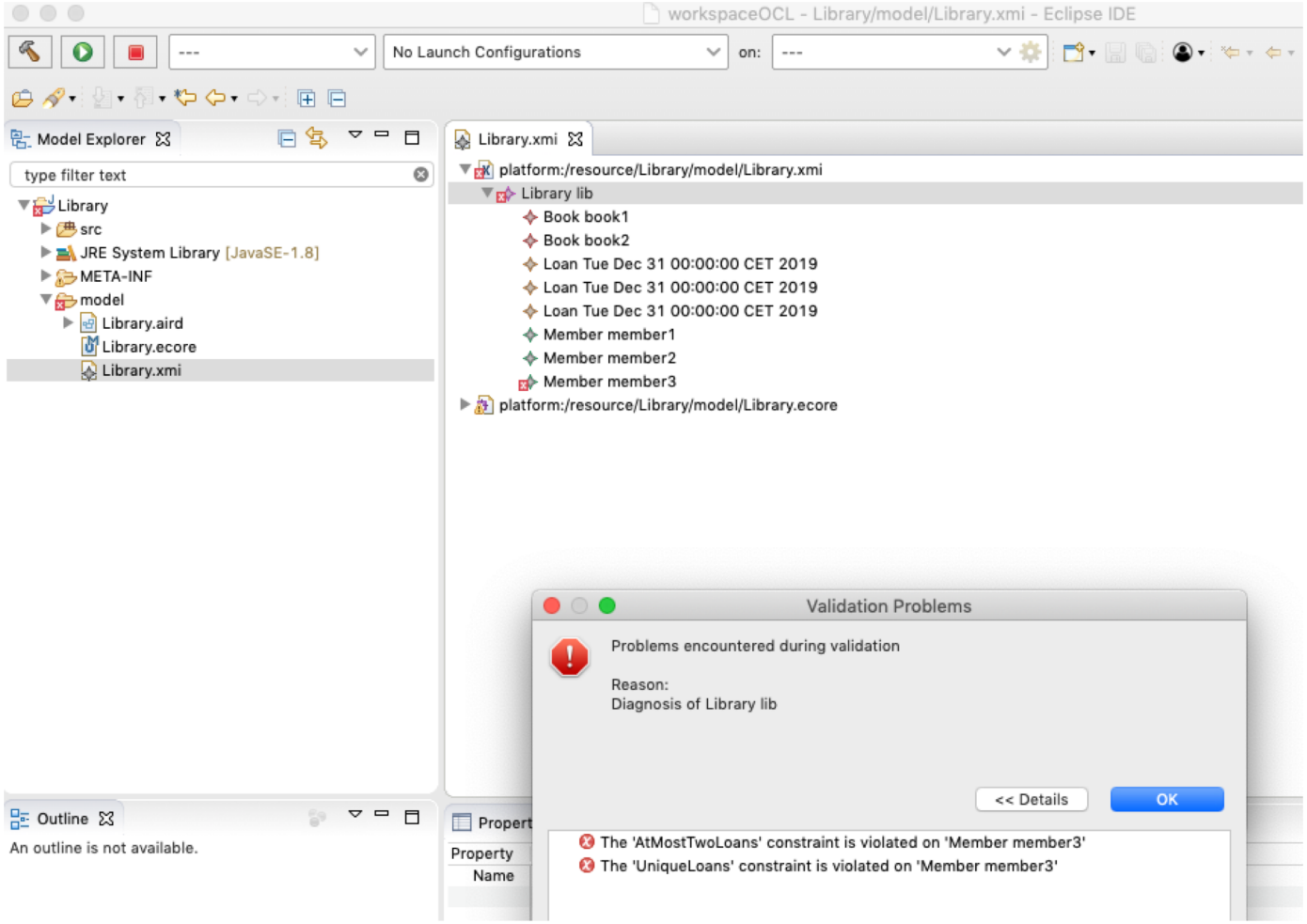
- We now add further helpers and constraints to enforce “at most two loans per member policy” and “to require loans to be unique”.
- Replace the Member class in **Library.ecore** meta-model by:

```
class Member
{
        attribute name : String[?];
        property library#members : Library[?];
        property loans : Loan[*] { derived volatile }
        {
                initial: library.loans->select(member=self);
        }
        property books : Book[*] { !unique derived volatile }
        {
                initial: loans->collect(book);
        }
        invariant AtMostTwoLoans:
                loans->size() <= 2;
        invariant UniqueLoans:
                loans->isUnique(book);
}
```

Helper Features and Operations

- After we validate the **Library.xmi** instance model, two errors appear since **member3** has **more than 2 loans** and he has **multiple copies** of **book2.**

- *How can we correct that?*



Related Online Materials

- https://www.omg.org/spec/UML/2.5.1/
- https://www.omg.org/spec/OCL/2.4/
- http://www.uml.org
- https://www.omg.org/spec/OCL/About-OCL/
- https://help.eclipse.org/oxygen/index.jsp
- https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.ocl.doc%2Fhelp%2FTutorials.html

Questions?