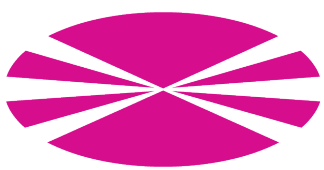
Unit 4: UML (Unified Modeling Language)

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science



UNIVERSIDADE DA CORUÑA

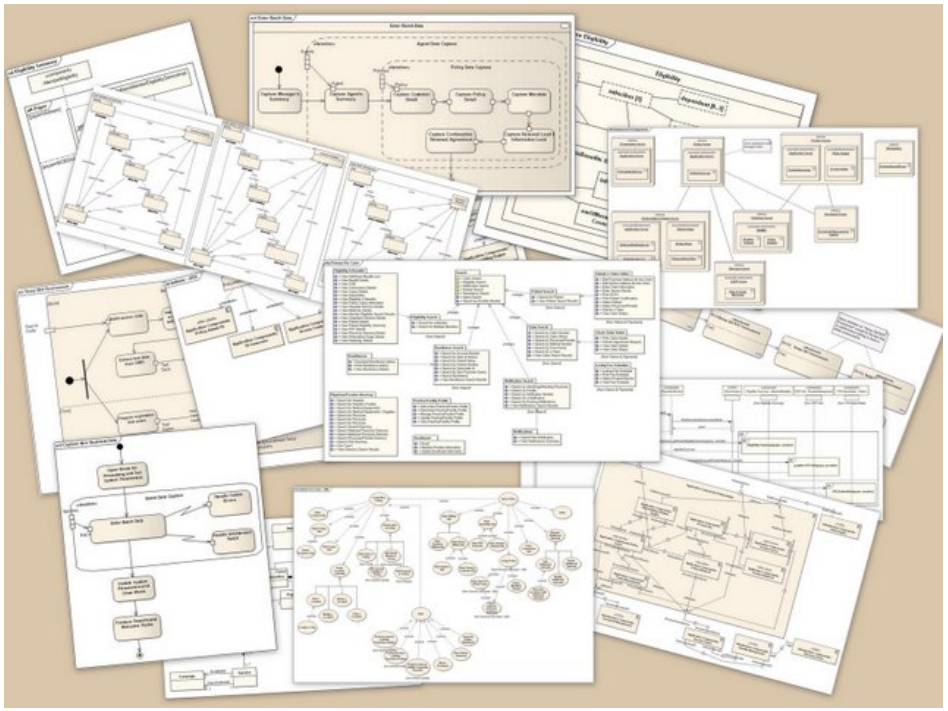
Table of Contents

Table of Contents

UML Objectives

Unified Modeling Language (UML)

UML is a graphical language for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system.



UML Objectives

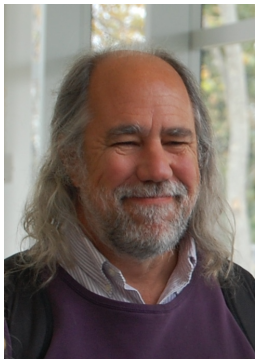
UML Objectives

UML is a standardized general-purpose modeling language in the field of object-oriented software engineering but also expressive enough to model non-software systems.

- It is a standard way of modeling (see Unit 1).
- **UML is not a methodology**
  - UML is only a language. It is not a methodology but just one part of a software development method.
  - UML is process independent, although its authors are also the authors of the Unified Process.
  - If the artifacts of the different methodologies are written in a common language like UML, communication between developers would be fluid.

UML History

- **Three Amigos**
  - Three well-known authors of object oriented methodologies create a unified and non-proprietary modeling language.
  - The new language is supported by professionals and by enterprises and becomes a *de facto* standard.



Grady Booch
(OOD)

James Rumbaugh
(OMT)

Ivar Jacobson
(OOSE)

UML History

- **Standardization: OMG and ISO**
  - Version 1.1 of UML is accepted and maintained since 1997 by the OMG (Object Management Group - `https://www.omg.org/`) a consortium originally aimed at standardizing object-oriented systems.
  - Version 2.4.1 is recognized as a standard by ISO: *ISO/IEC 19505-1:2012 Information technology – Object Management Group Unified Modeling Language (OMG UML)*"
  - Current UML specificaton is version 2.5.1 (2017) - `https://www.omg.org/spec/UML/`

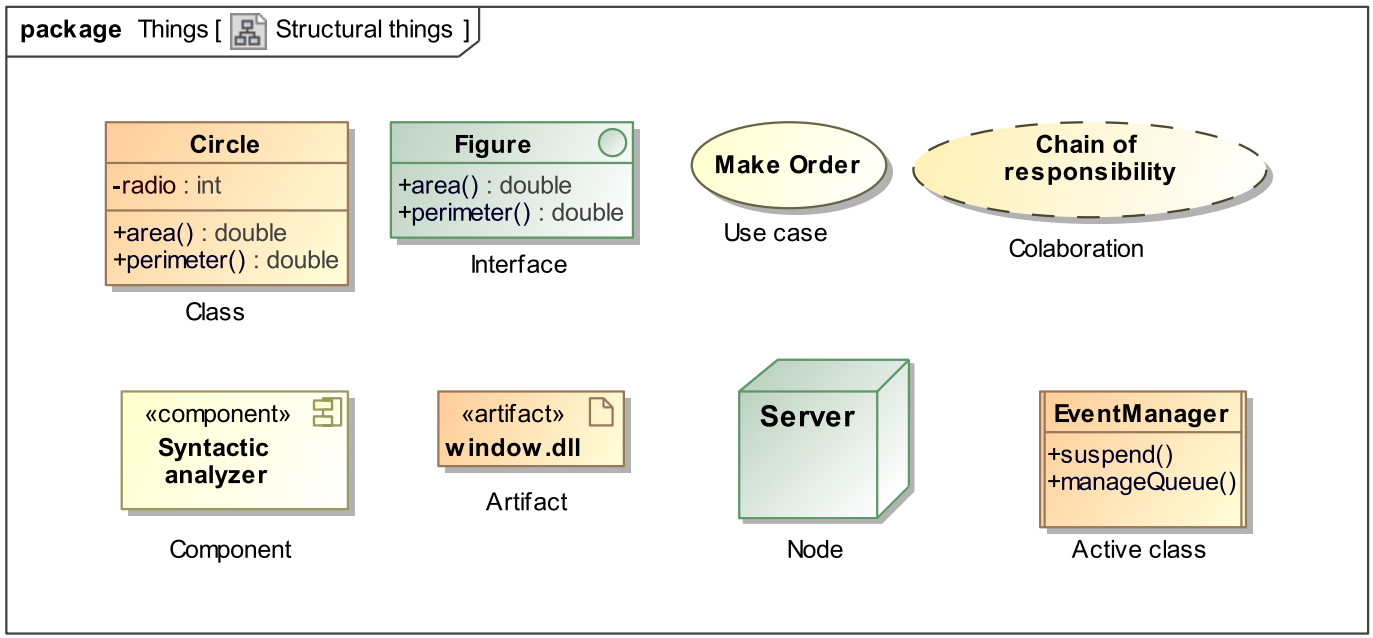
Table of Contents

Building Blocks of UML

- **Things**:
  - Abstractions that are first-class citizens in a model.
- **Relationships**:
  - Basic relational building blocks.
- **Diagrams**:
  - Graphical representation of a set of elements, most often rendered as a connected graph of vertices (things) and paths (relationships).

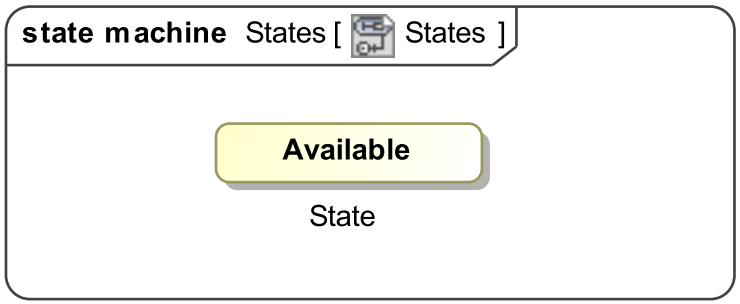
Things

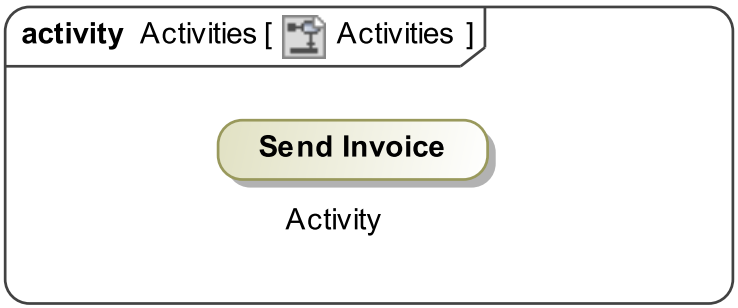
- **Structural things**: These are mostly static parts of a model, representing elements that are either conceptual or physical.

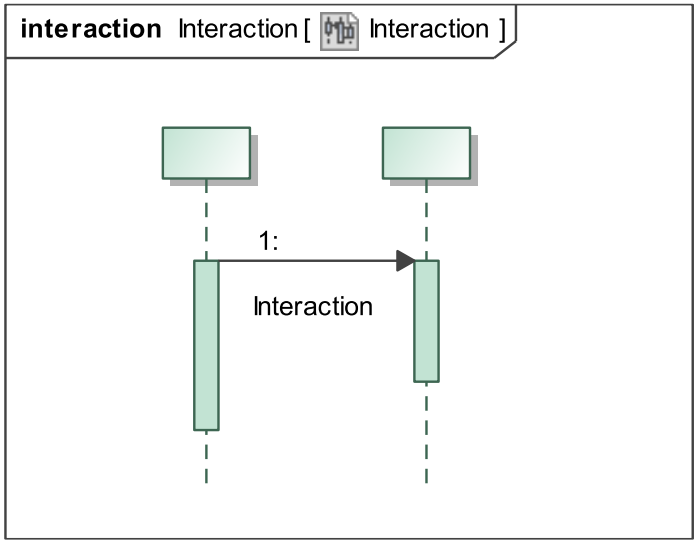


Things

- **Behavioral things**: They are the dynamic parts of UML models (the verbs) and represent behavior over time and space.

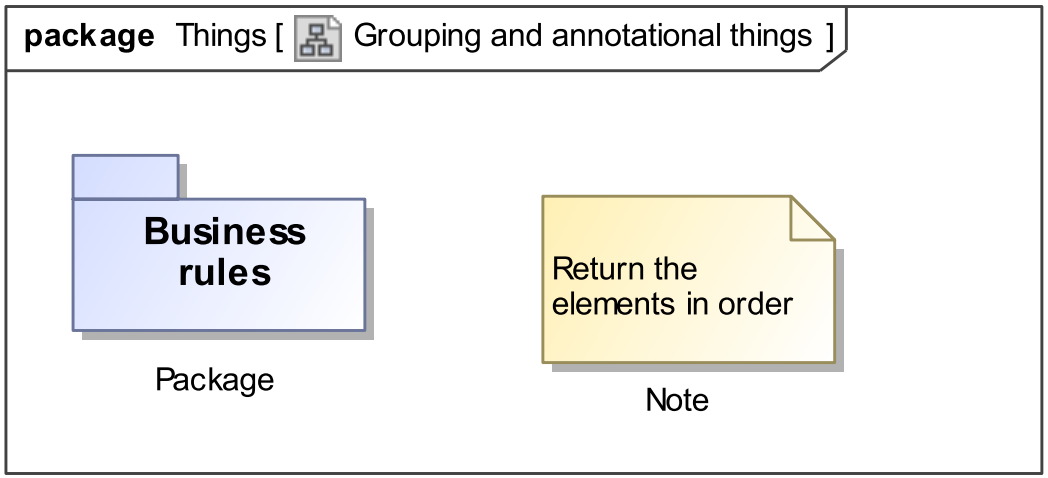






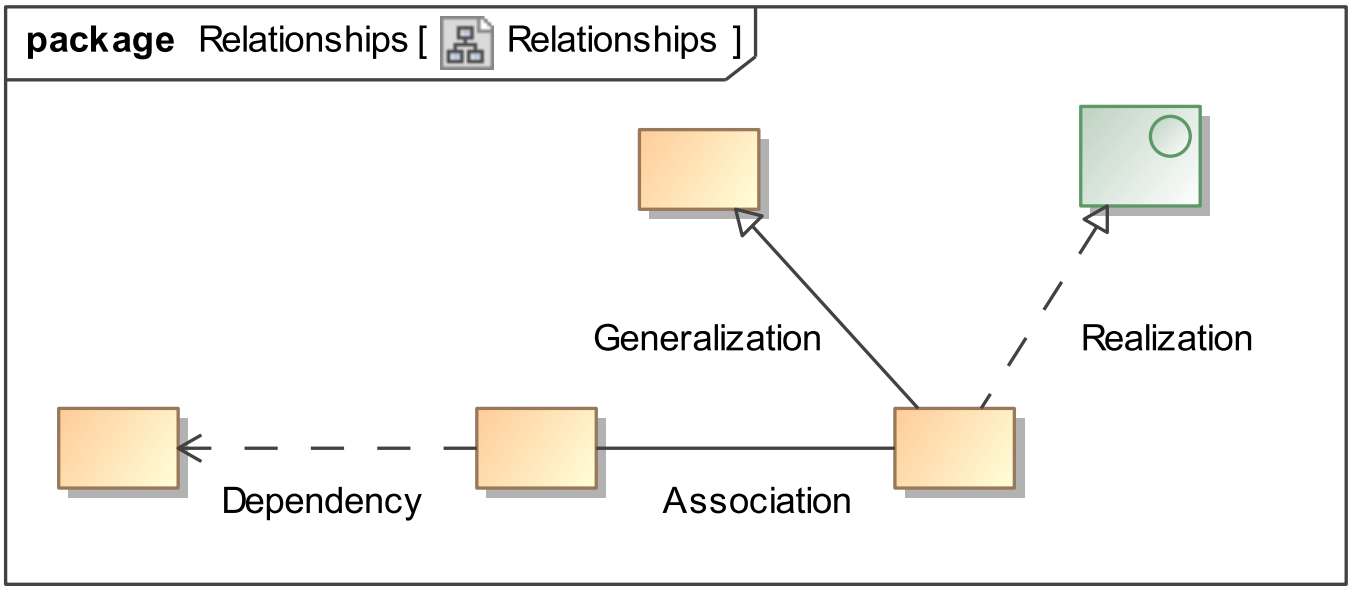
Things

- **Grouping things**: Organizational part of UML models. There is one primary kind of grouping thing, namely, *packages*.
- **Annotational things**: Explanatory part of UML models. There is one primary kind of annotational thing, namely, *notes*, that can be applied to describe, illuminate, and remark about any element in a model.



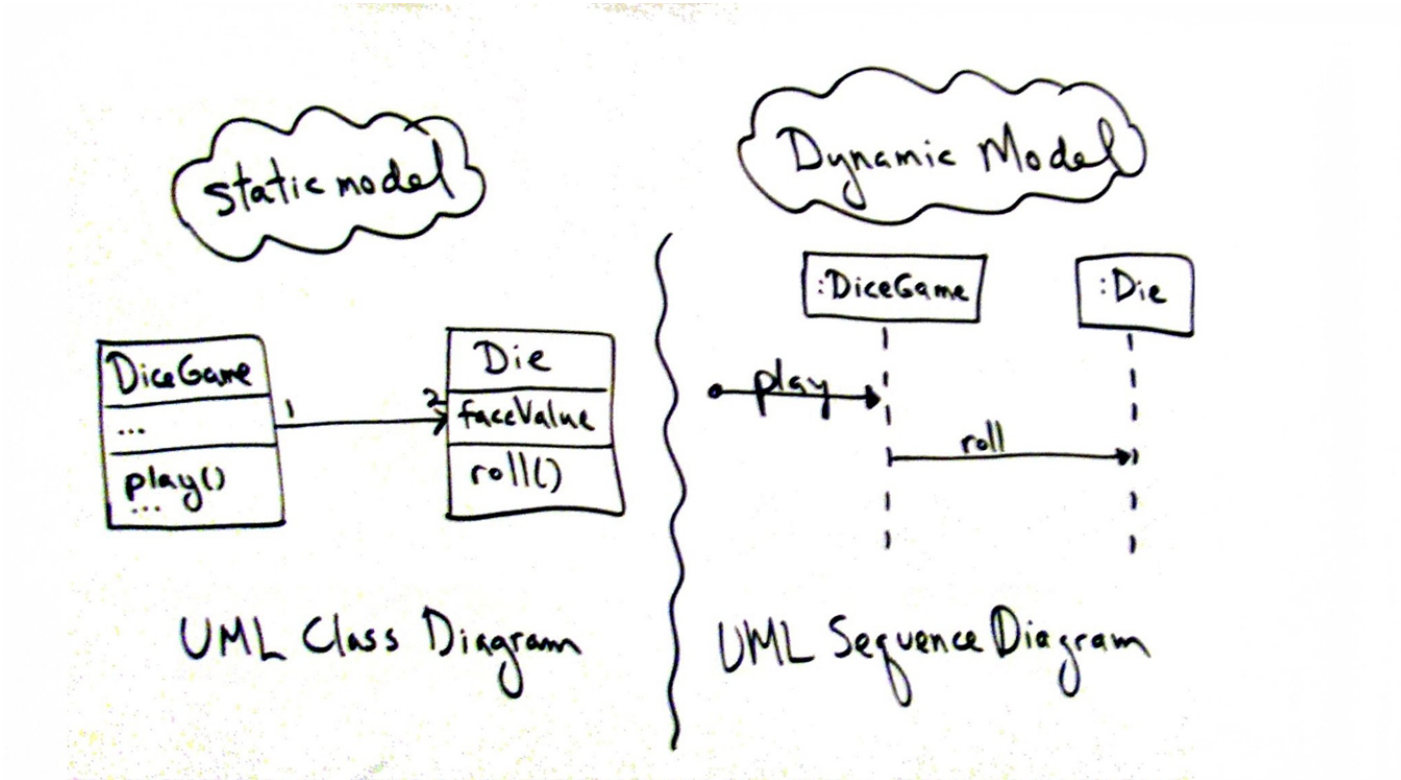
Relationships

- There are four kinds of relationships in UML: **Generalization**, **Realization**, **Association** and **Dependency**.
- All of them appear in class diagrams.



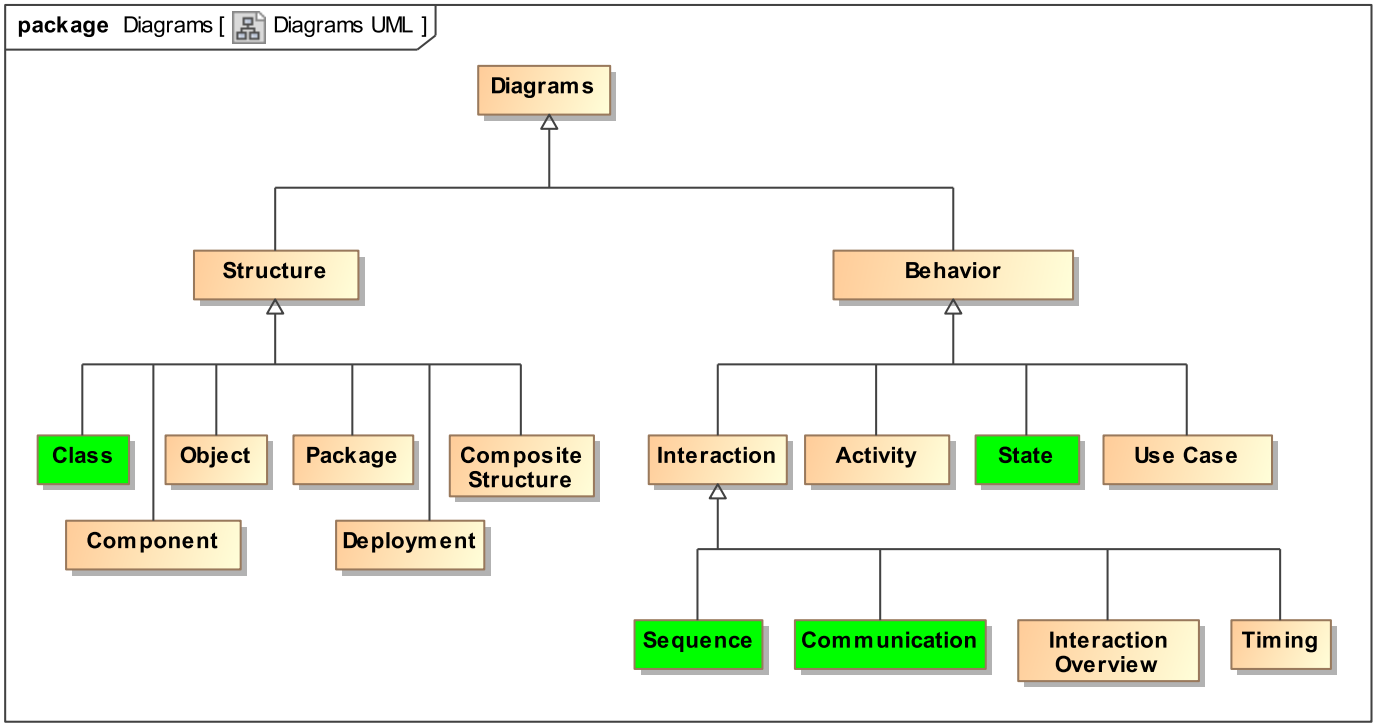
Diagrams

- Diagrams are drawn to visualize a system from different perspectives, so a diagram is a projection into a system.
- There are two main groups: **Static Diagrams** and **Dynamic Diagrams**.



Static design vs. dynamic design

- In this course we will focus on the following diagrams:

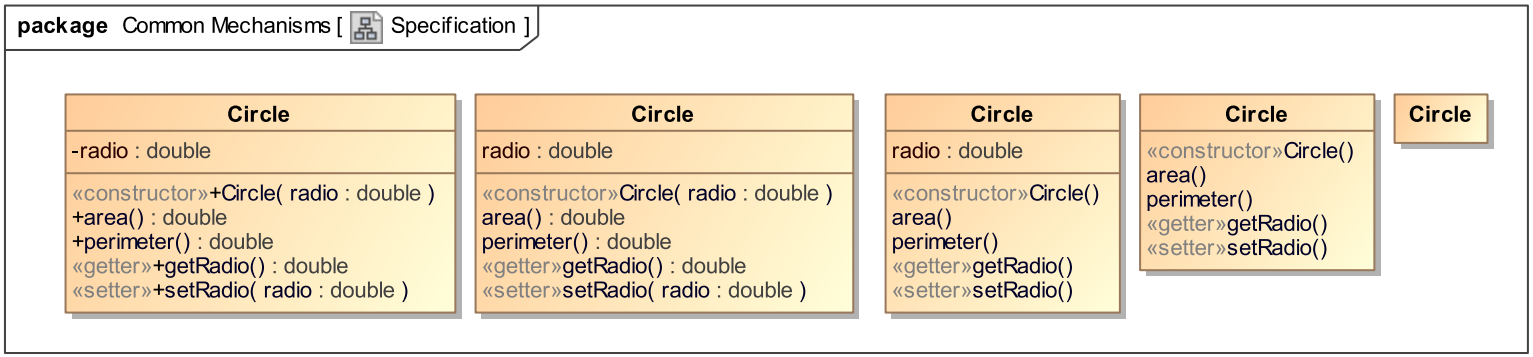


Common Mechanisms

- In UML there are four *common mechanisms* that apply consistently throughout the language.
- They are: **Specifications**, **Adornments**, **Common divisions** and **Extensibility mechanisms**.
- **Specifications**
  - Behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and semantics of that building block.
  - For example, behind a class icon is a specification that provides the full set of attributes, operations, and behaviors that the class embodies.
  - Visually, a class icon might only show a small part of this specification.

Common Mechanisms

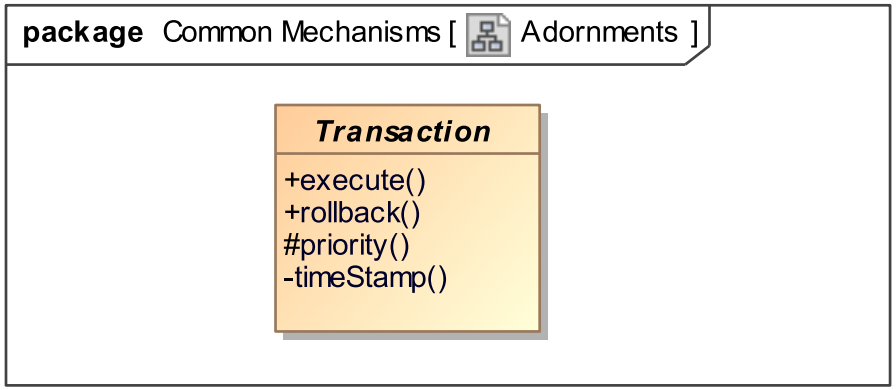
- **Specification example**
  - Class `Circle` can be represented in complete form or hiding part of the specification to make the diagram more readable.



- Dedicated UML tools can do this very easily ⇒ advantage over general drawing tools.
- In contrast, UML tools are less flexible as they require that the model we are creating adheres to UML rules.

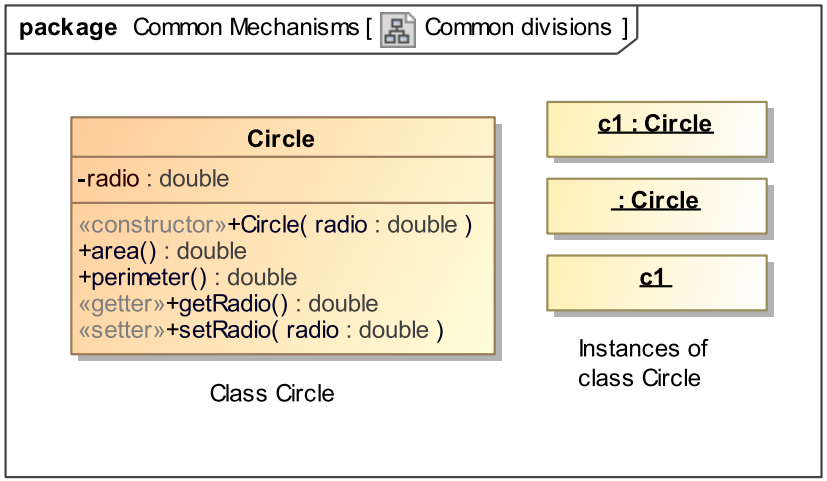
Common Mechanisms

- **Adornments**
  - Every element in UML's notation starts with a basic symbol, and you can add a variety of adornments specific to that symbol.
  - For example, a class specification may include details, such as whether it is abstract (the name is in italics) or the visibility (i.e., the access specifiers) for its attributes and operations.



Common Mechanisms

- **Common Divisions**
  - Almost every building block in UML has a class/object dichotomy.
  - For example, you can have classes and objects; use cases and use case executions; components and component instances; nodes and node instances; and so on.
  - Graphically, UML distinguishes an object by using the same symbol as its class and then simply underlying the object's name. The name of the object and the name of the class are separated by a colon.

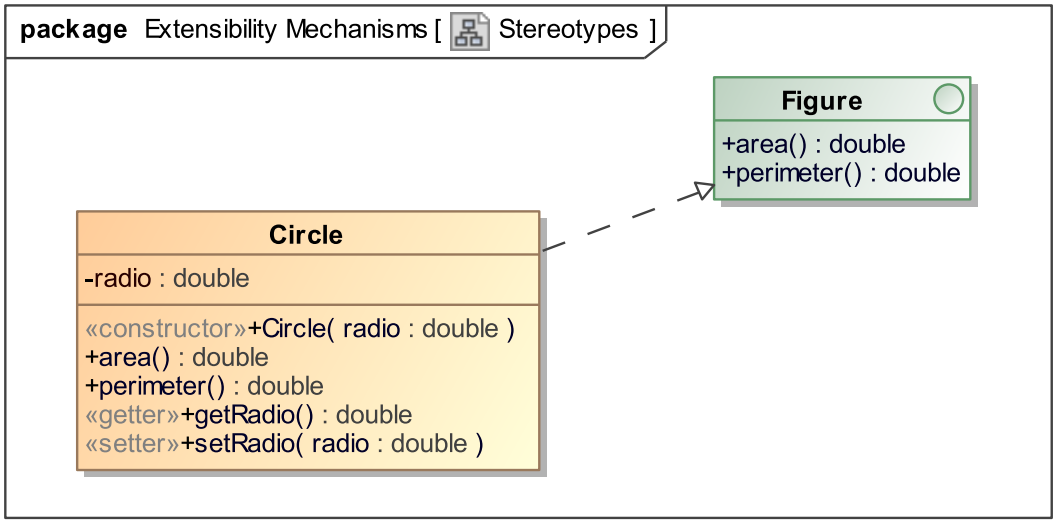


Common Mechanisms

- **Extensibility Mechanisms**
  - UML provides a standard language for writing software blueprints, but a closed language can never express all the possible nuances of all models across all domains.
  - For this reason, UML is opened-ended, allowing the extension of the language in controlled ways.
  - Many of these extensibility mechanisms are in UML tools in what is called *profiles*: UML extensions tailored for a given technology (Java, C#, etc.).
  - UML's extensibility mechanisms include: **Stereotypes**, **Tagged values** and **Constraints**.

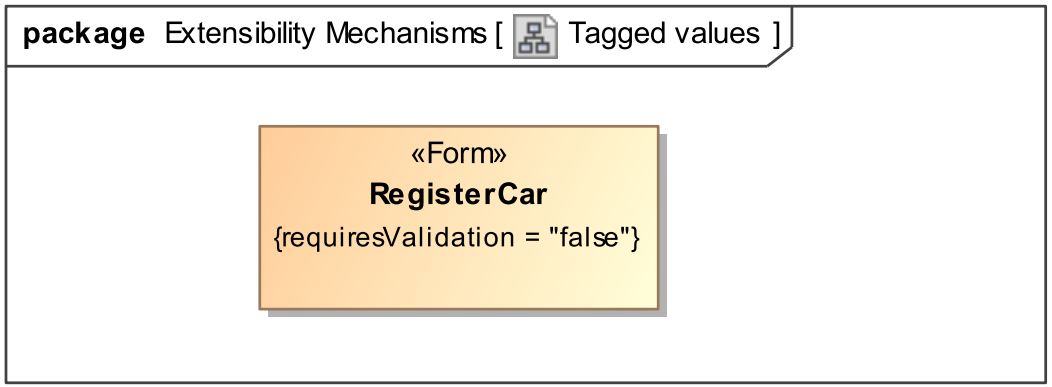
Extensibility Mechanisms

- **Stereotypes**
  - A stereotype extends UML's vocabulary, letting you create new kinds of building blocks that are derived from existing ones but specific to your problem.
  - Stereotype notation uses *guillemets* or *latin quotation marks*: «. . . »
  - On occasions the new stereotyped elements can be rendered with a new graphic notation (e.g. the «interface» stereotype is represented in MagicDraw as a class with a circle in the upper right corner).



Extensibility Mechanisms

- **Tagged values**
  - A tagged value is a keyword-value pair that may be attached to any kind of model element.
  - For example: {`author="Joe Smith", deadline=31-March-2016, status=analysis`}
  - Since UML 2.0, a tagged value can only be represented as an attribute defined on a stereotype.



Extensibility Mechanisms

- **Constraints**
  - A constraint extends the semantics of a UML building block, allowing you to add new rules or modify existing ones.
  - For example, you might want to add a constraint to the queue class so that all elements are inserted in order.

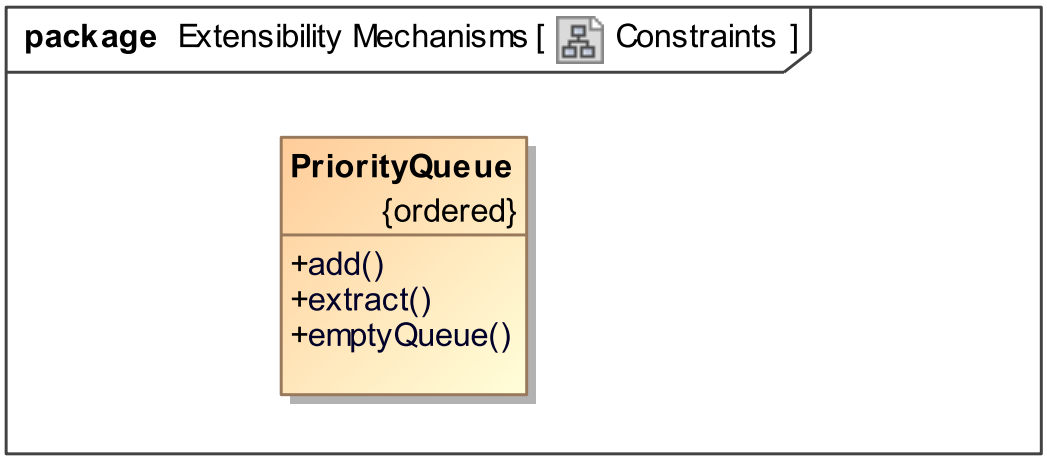


Table of Contents

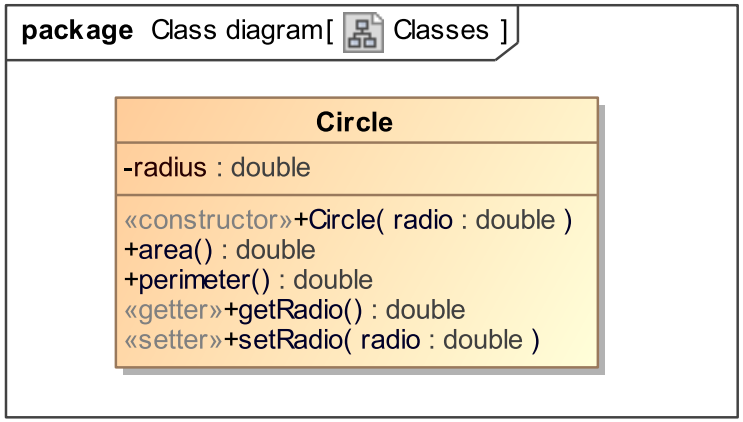
UML Classes

- A class definition in Java is composed of three elements: a name, a list of attributes and a list of methods or operations.

Java class definition

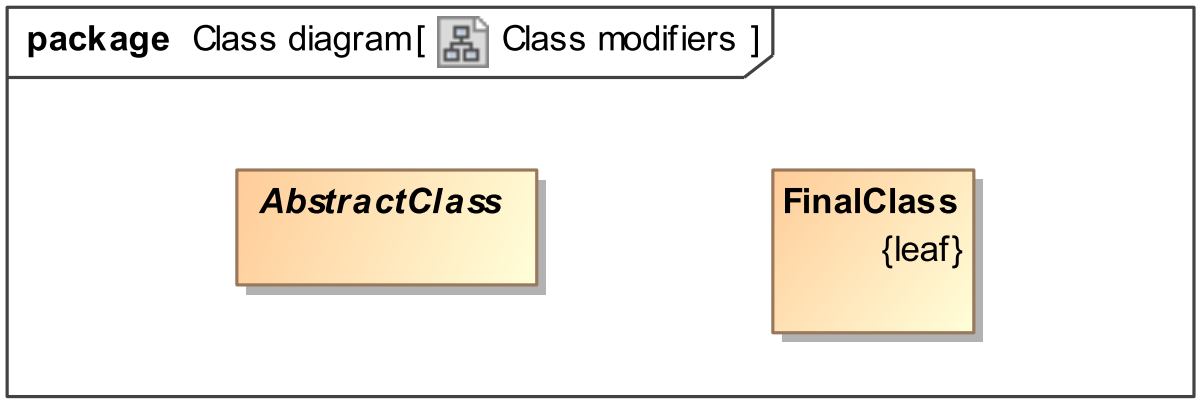
```
[Modifiers] class Name [extends Class]
[implements interface, ...] { // Attributes // Methods }
```

- A UML class is rendered as a solid-outline rectangle divided in three compartments separated by horizontal lines for storing the name, the attributes and the operations respectively.



UML class modifiers

Java modifiers	UML Representation
`public`	Exists in the specification but normally is not rendered graphically (it can be done adding a "+" before the name of the class)
`abstract`	The name of the class is in *italics*
`final`	Restriction {`leaf`} or property `isLeaf`[1]



[1] Since UML 2.3 `isLeaf` is a property of the specification of classes or methods but it is not rendered graphically.

UML attributes

- UML follows a style similar to Pascal for representing attributes and methods.

Java attributes

```
[Visibility][Modifiers] type[] name [= value];
```

UML attributes

```
[Visibility] name [:type] [Multiplicity] [=value]
[{Modifiers}]
```

UML attributes

Java visibility	UML representation
`public`	+
`protected`	#
`[package]`	~
`private`	–

Java attribute modifier	UML representation
`static`	The attribute is underlined
`final`	Constraint {`readOnly`}. If it is a *blank final*, the {`frozen`} constraint can be used.

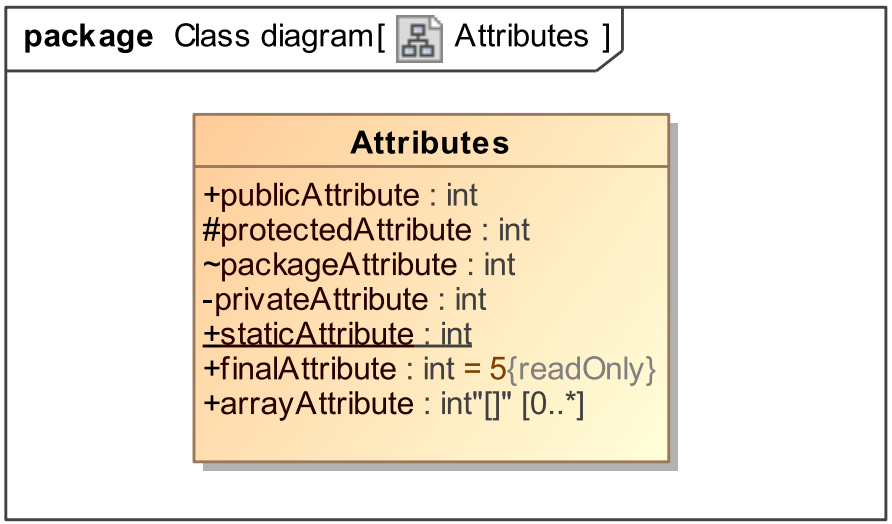
UML attributes

- **UML multiplicity**
  - Represented as an interval between a lower limit and an upper limit: `[lower_limit..upper_limit]`.
  - The upper limit can be rendered as "`*`" indicating that it is not bounded.
  - Some cases can be simplified to just one value, e.g. `[5..5]` is equivalent to `[5]`, or `[0..*]` is equivalent to `[*]`.
  - Default multiplicity, if not specified otherwise, is `[1]`.
  - In attributes a multiplicity higher than one is used to represent arrays.

UML attributes

Attributes example

```
public class Attributes {
    public int publicAttribute;
    protected int protectedAttribute;
    int packageAttribute;
    private int privateAttribute;
    //
    public static int staticAttribute;
    public final int finalAttribute = 5;
    //
    public int[] arrayAttribute;
}
```



UML methods

Java methods

```
[Visibility][Modifiers] returnType name
([parametersList]) [throws ExceptionList] {bodyMethod};
```

UML methods

```
[Visibility] name ([parametersList]) [:returnType]
[{Modifiers}]
```

UML methods

- **Visibility**: Similar to attributes.

Java method modifier	UML representation
`static`	Method is underlined
`abstract`	Method is in *italics*
`final`	Restriction {`leaf`} or `isLeaf` property

UML methods

UML parameters

```
[Direction] name : type [Multiplicity] [=defaultValue] [properties]
```

- Parameters can be `in`, `inout`, `out` and `return`.
  - `in`: Indicates that parameter values are passed in by the caller.
  - `inout`: Indicates that parameter values are passed in by the caller and then back out to the caller.
  - `out`: Indicates that parameter values are passed out to the caller.
  - `return`: Indicates that parameter values are passed as return values back to the caller.
- Defaults to `in` if omitted (all Java parameters are `in`).

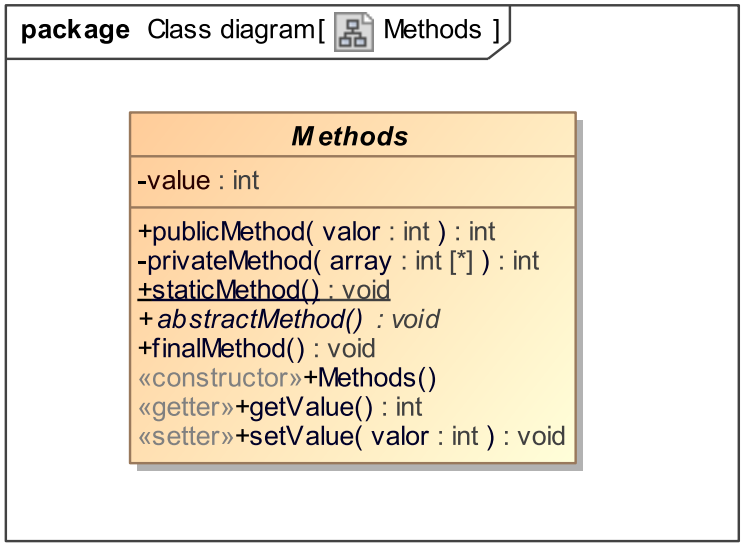
UML method modifiers

- A UML method can include several constraints, for example:
  - {query}: Operation does not change the state of the object.
  - {ordered}: Operation returns ordered values.
  - {unique}: Operation returns a set of values without duplicates.
- Sometimes UML tools use stereotypes to indicate particular language artifacts. For example, for Java:
  - «constructor»: It is a constructor method.
  - «getter»: It is a getter method for a property.
  - «setter»: It is a setter method for a property.

UML methods

Methods example

```
public abstract class Methods {
    private int value;

    public int publicMethod(int valor) { ... }
    private int privateMethod(int[] array) {...}
    public static void staticMethod() {}
    public abstract void abstractMethod();
    public final void finalMethod() {};
    public Methods() {}
    public int getValue() { ... }
    public void setValue(int value) { ... }
}
```

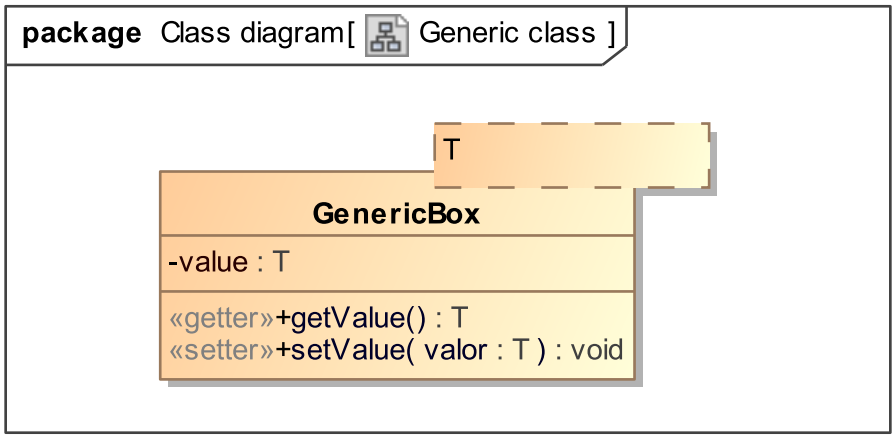


UML generic classes

- Generic classes are rendered with a dashed rectangle superimposed in the upper right-hand corner of the class in which the generic parameter is included.

Generic class `GenericBox`

```
class GenericBox<T> {
    private T value;

    public T getValue() {
        return value;
    }

    public void setValue(T value) {
        this.value = value;
    }
}
```



Exercise

UML representation of class Box

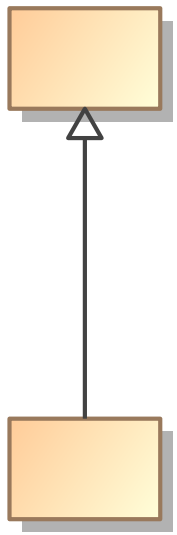
```
public class Box {
    private int side;
    private int value;
    private static int numBoxes = 0;
    public static final double PI = 3.1416;

    public Box() { ... }
    public Box(int value) { ... }
    public Box(int value, int side) { ... }
    public void setValue(int value) { ... }
    public int getValue() { ... }
    public int getSide() { ... }
    public void setSide(int side) { ... }
    public static int getNumBoxes() { ...  }
    public boolean equals(Object obj) { ... }
    public int hashCode() { ... }
    public int areaSide() { ... }
    public double perimeterCircle() { ... }
    public String toString() { ... }
}
```

Generalization relationship

Generalization relationship

Relationship between a general classifier and a more specific classifier that will inherit all the structure and behavior of the parent. Instances of the child may be used anywhere instances of the parent apply.

- **Notation**
  - A solid line representing an arrow from the child to the parent, with a large hollow triangle on the end connected to the parent.
  - Several generalization relationships can be drawn as a tree branching into several lines to the children.

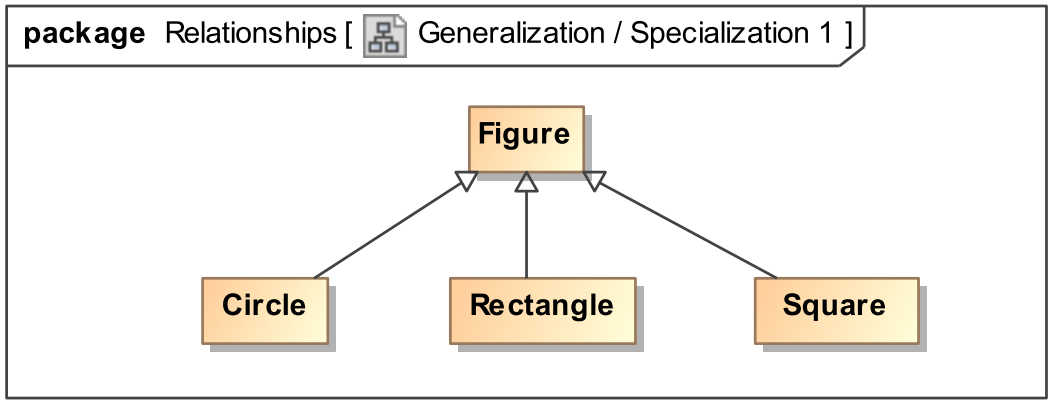


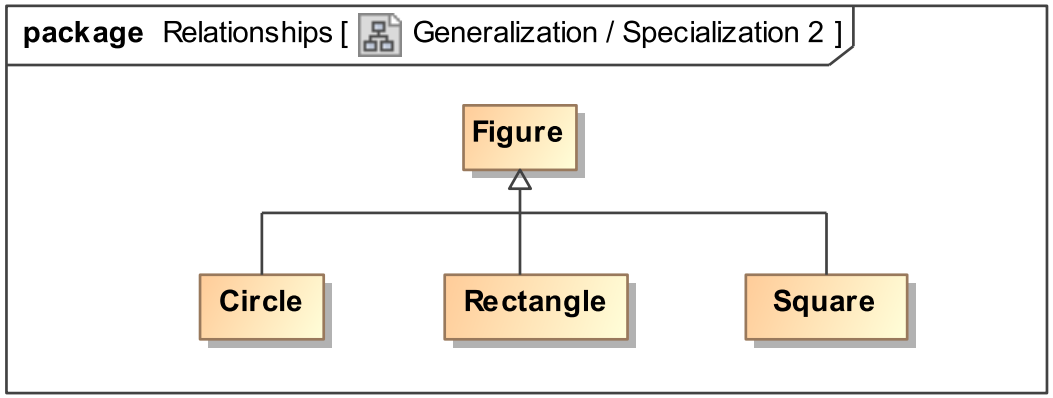
Generalization relationship

- **In code...**
  - This relationship is easy to identify in Java, you only have to identify the `extends` keyword.

`extends` in Java

```
class Figure { ... }

class Circle extends Figure
{ ... }

class Rectangle extends Figure
{ ... }

class Square extends Figure
{ ... }
```



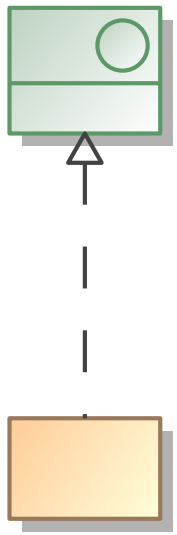


Realization relationship

Realization relationship

A realization is a semantic relationship between classifiers in which one classifier specifies a contract that another classifier guarantees to carry out.

- **Notation**
  - Rendered as a dashed directed line with a large hollow arrowhead pointing to the classifier that specifies the contract.
  - There is an alternative elided notation known as the *lollipop* notation.

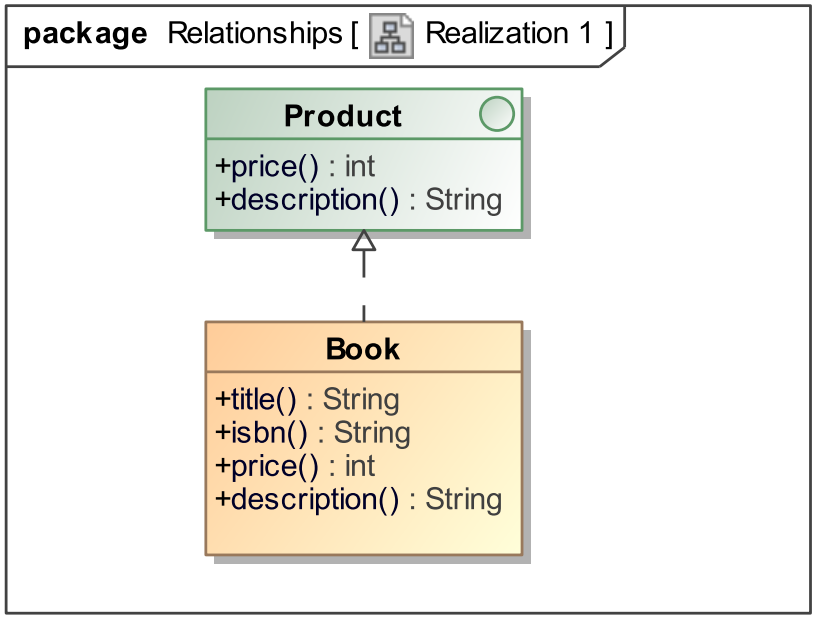


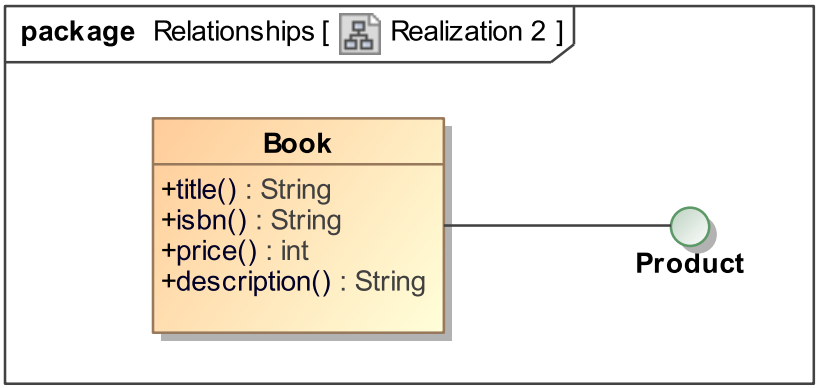
Realization relationship

- **In code...**
  - This relationship is easy to identify in Java, you only have to identify the `implements` keyword.

`implements` in Java

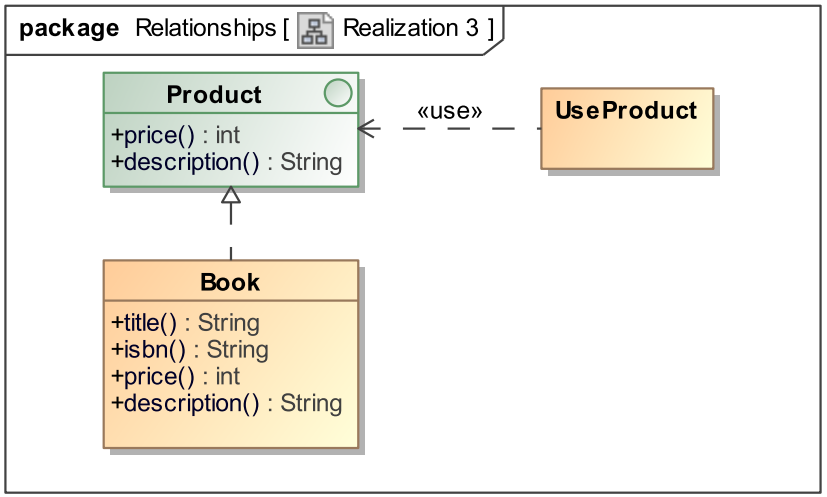
```
interface Product {
  int price();
  String description();
}

class Book implements Product {
  public String title() {...};
  public String isbn() {...};

  public int price() {...}
  public String description() {...}
}
```

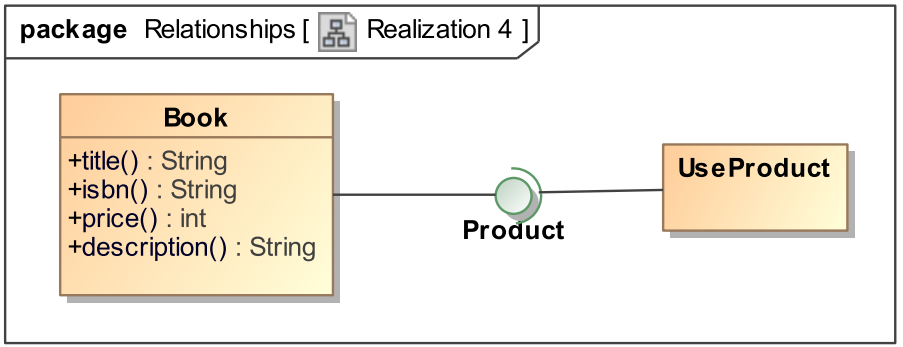




Realization relationship

- It is important to distinguish between the class that realizes (implements) the interface and the class that is using it.
- The latter uses an association or a depencency with the interface or, if we are using the *lollipop* notation, a *socket*.





Using an interface

```
class UseProduct {
  Product myProduct;
  // ...
}
```

Association relationship

Association relationship

Structural relationship that specifies that objects of one thing are connected to objects of another.

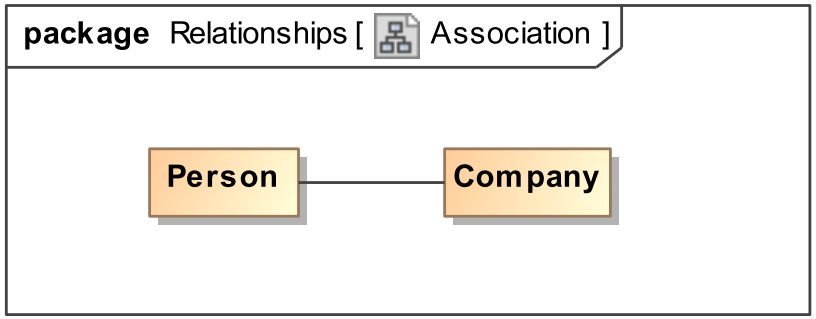
- **Notation**
  - Graphically, an association is rendered as a solid line connecting the associated classes.
  - Over this line you can arrange different adornments that establish the characteristics of that association.

Association relationship

- **In code...**
  - The most common way to establish an association between two classes in code is making one class an attribute of the other, therefore becoming a part of its state.
  - A simple association, without adornments, does not have much information.

Java association

```
class Person {
  Company employer;
  // ...
}

class Company {
  Person[] employee;
  // ...
}
```

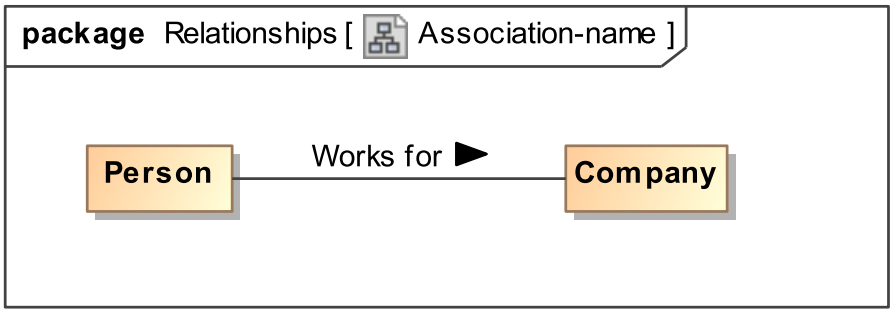


UML association adornments

- **Main adornments**
  - Simple adornments that we should try to include in each association.
  - They include: *name*, *role names*, *multiplicity*, *navigation* and *visibility*.
- **Aggregation/composition adornments**
  - Adornments that specify if an association is an *aggregation* or a *composition*.
- **Other adornments**
  - Adornments used only on specific situations that do not occur often.
  - They include: *qualification*, *association classes* and *n-ary associations*

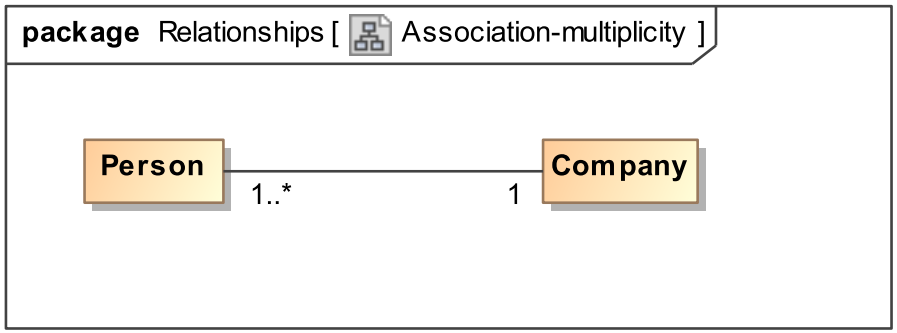
Main adornments

Adornment	**Association name**
Meaning	Used to describe the nature of the relationship so there is no ambiguity about its meaning.
UML	The name is rendered over the association and can be accompanied by a direction triangle that points in the direction you intend to read the name.
Java	It is intended to be used more in domain conceptual diagrams (used in analysis tasks) and not in design diagrams that are in more direct contact with code. Therefore this adornment has no direct representation in code.



Main adornments

Adornment	Multiplicity
Meaning	Establish how many objects may be connected across an instance of an association.
UML	It is written as an expression `minimum..maximum` near each one of the classes they are modifying.
Java	A multiplicity of `1` is simply represented by an attribute. Greater than one is represented by arrays or object collections.

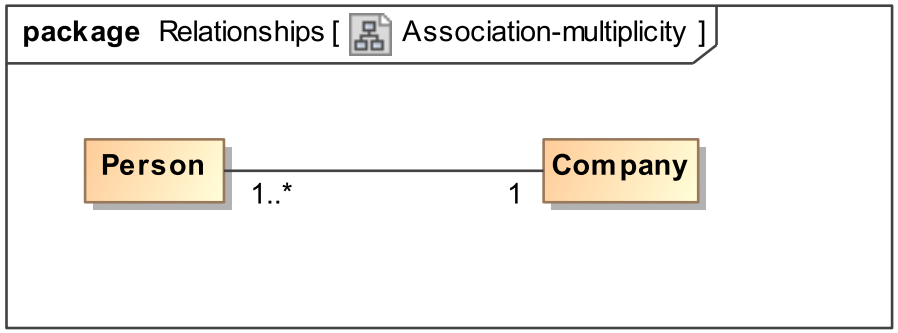


Multiplicity in Java

```
class Person {
  Company employer;
}

class Company {
  List<Person> employee;
}
```

Main adornments

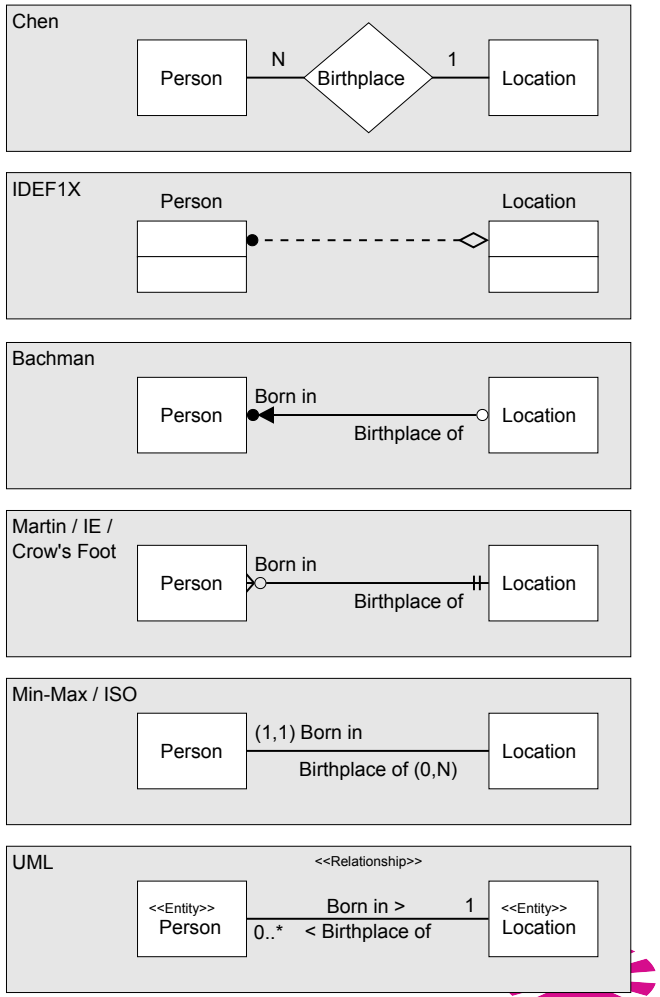
- ***Look-across* cardinality**
  - UML follows a *look-across* notation to represent cardinality.
  - It is called *look-across* because when reading a cardinality you have to look to "the other side" of the relationship.
  - For example: "*A person works for **a** company*".



  - The *1* is near `Company` because that is its cardinality in the relationship, regardless of how it is read.

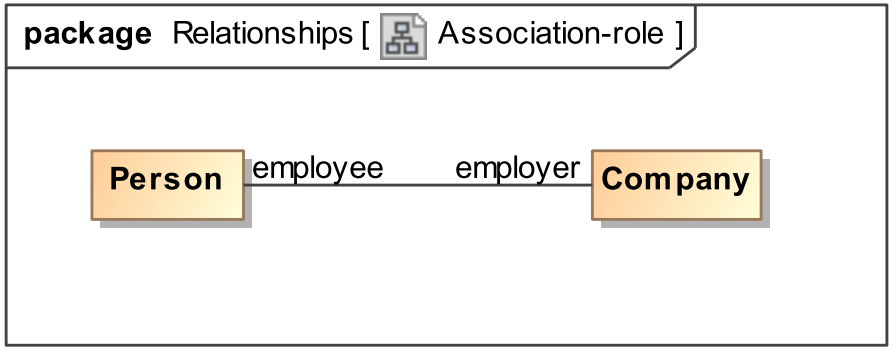
Main adornments

- ***Look-here* cardinality**
  - In contrast with software design and UML, in database design authors follow several schemes for rendering their diagrams.
  - Some schemes, as Chen diagrams, use *look-across* cardinalities.
  - Others, like Min-Max followed by authors like Elmasri and Navathe, follow *look-here* cardinalities.
  - *Look-here* cardinalities do not put the cardinality of the entity next to it, but it is placed following the direction of reading.



Main adornments

Adornment	**Role name**
Meaning	Shows the specific role that a class plays in the relationship.
UML	Rendered as a name placed near the class that it is modifying.
Java	Role names correspond to attribute names in Java.

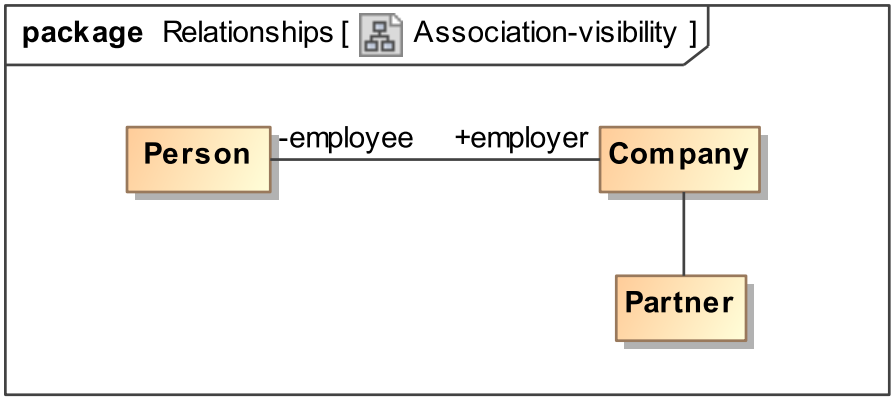


Role names in Java

```
class Person {
  Company employer;
}

class Company {
  List<Person> employee;
}
```

Main adornments

Adornment	Visibility
Meaning	Used to limit the visibility across that association relative to objects outside the association.
UML	Rendered through visibility icons in role names.
Java	Represented using visibility specifiers in attributes.

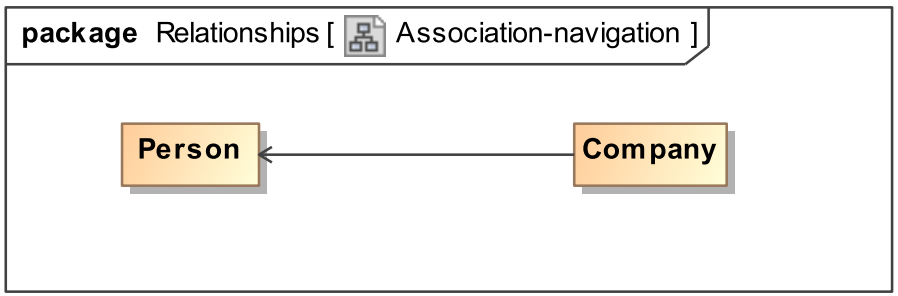


Visibility in Java

```
class Person {
  public Company employer;
}

class Company {
  private List<Person> employee;
}
```

Main adornments

Adornment	Navigation
Meaning	Indicates if it is possible to navigate from objects of one type to objects of the other type.
UML	Adorning an association with an open arrowhead pointing to the direction of traversal, or with a small cross ("x") if it is not navigable.
Java	Removing the attribute of a class limits the navigability to that class through the association.

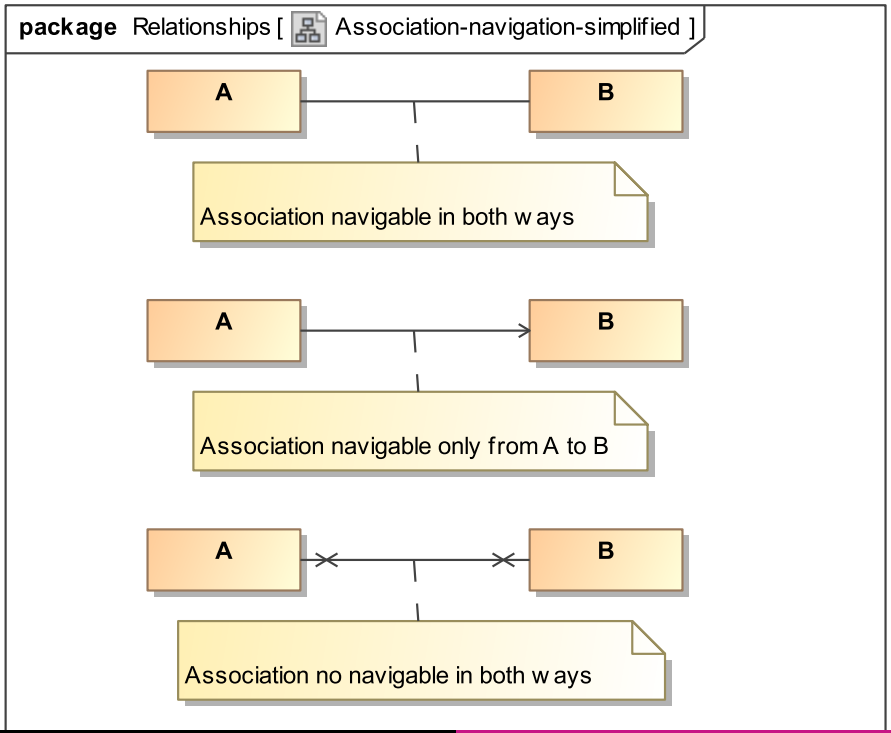


Navigation in Java

```
class Person {
 // We remove the Company attribute
}

class Company {
  List<Person> employee;
}
```

Navigation

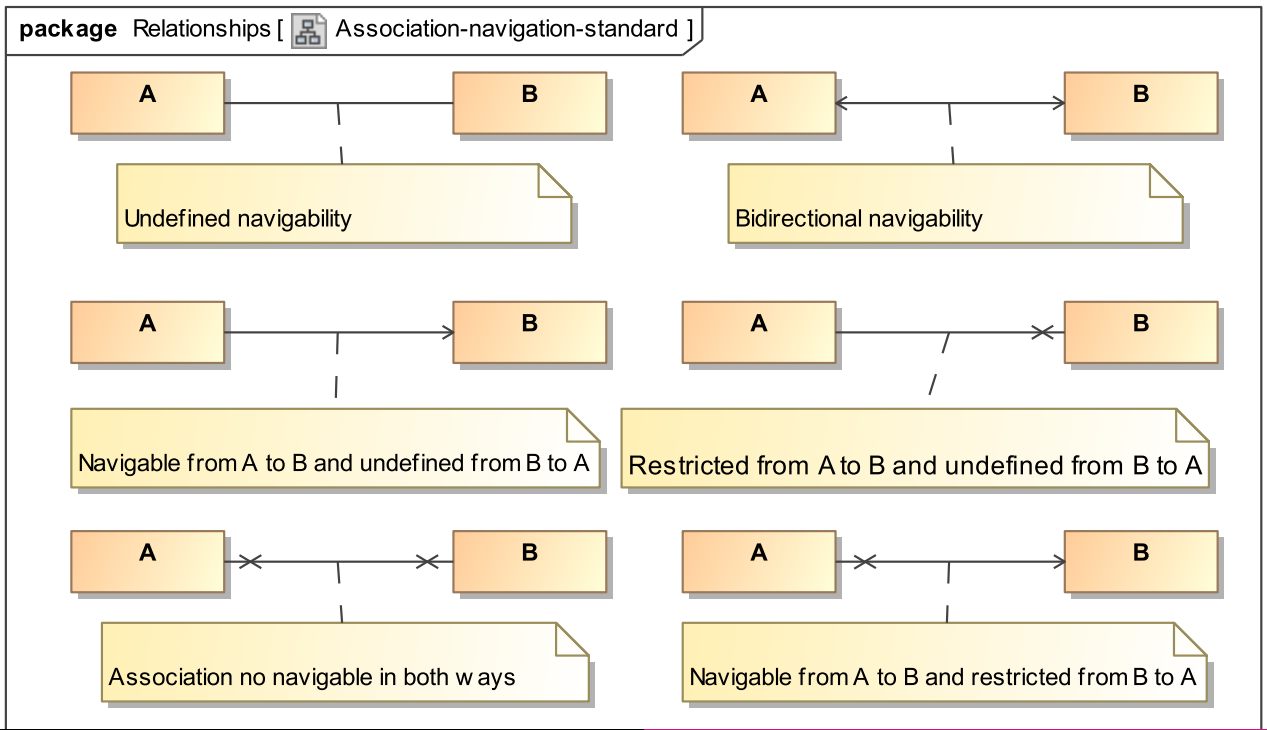
- **Implicit version**
  - By default, navigation across an association is bidirectional.
  - If you include an arrow, navigability is restricted in the opposite direction.
  - A double cross indicates restricted navigability in both directions
  - This implicit version is used in MagicDraw.



Navigation

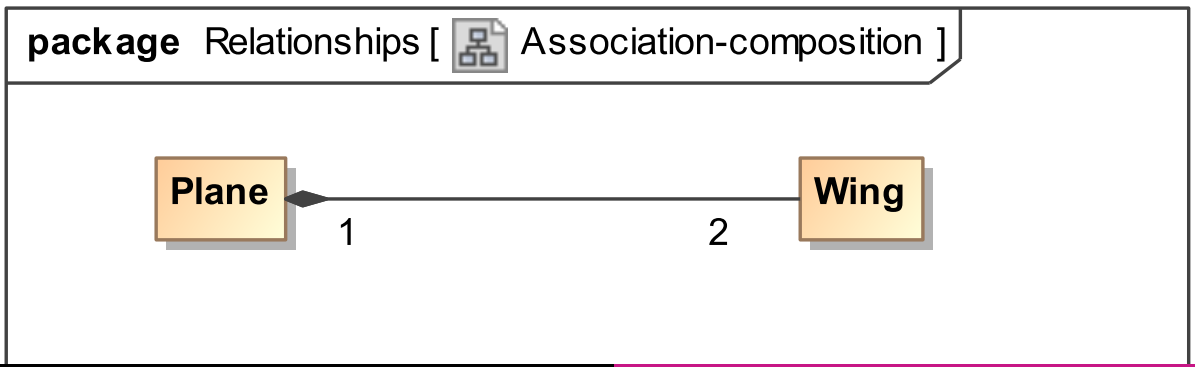
IMPORTANT!!
Not supported
by MagicDraw.

- **Explicit version (rarely used)**
  - If nothing is specified, navigability is undefined.
  - An arrow indicates that the navigability is possible in that direction (does not indicate anything on the other direction).
  - A cross indicates that the navigability is restricted in that direction (does not indicate anything on the other direction).



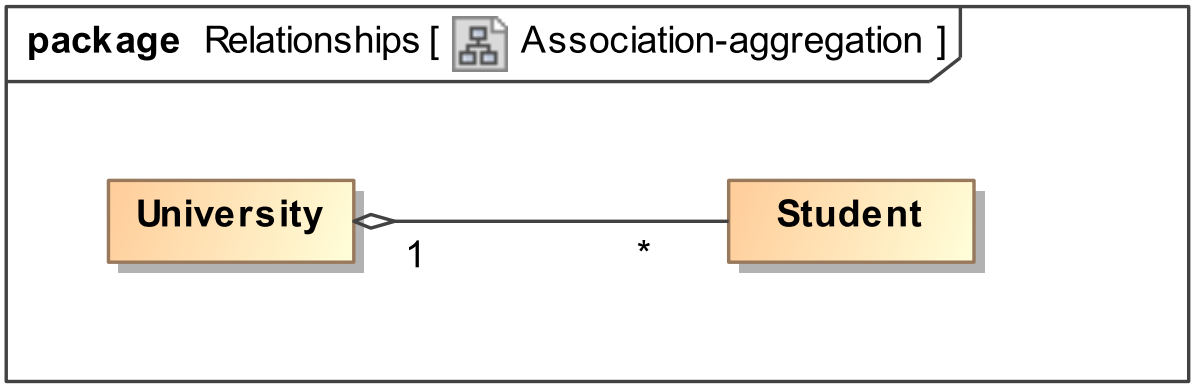
Aggregation vs. composition

- **Composition**
  - It is a "whole-part" relationship.
  - An object can be part of only one composite at a time (multiplicity of "whole" *cannot* be greater than one).
  - Lifetime of the "part" is coincident with lifetime of the "whole".
  - Composition is a stronger relationship between the two classes. For example a "part" can be viewed as as weak entity in a Entity-Relationship diagram.
  - Composite structure diagrams can be used to represent compositions in a more compact way.
  - There are several ways to implement compositions in code: as a normal associaton, using private objects, inner classes, etc.



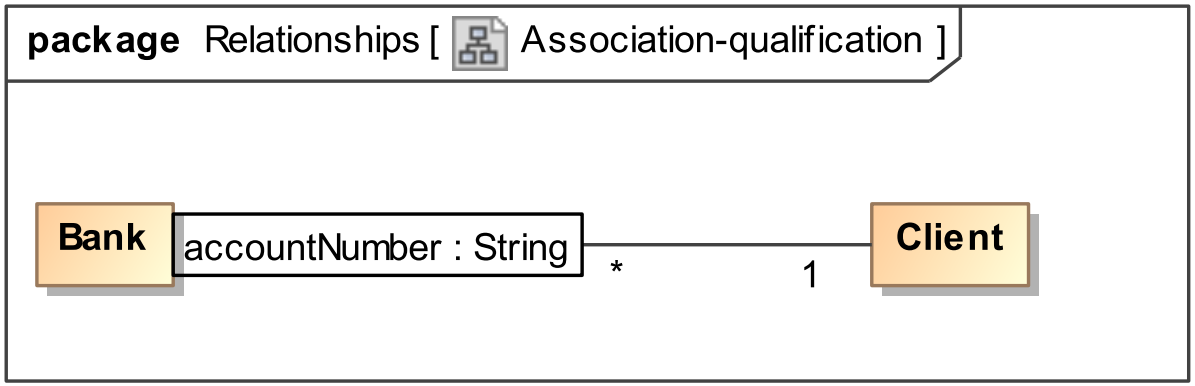
Aggregation vs. composition

- **Aggregation**
  - It is a "whole-part" relationship.
  - A part may be shared by several wholes (multiplicity of "whole" can be greater than one).
  - Lifetime of the "part" is independent of the lifetime of the "whole".
  - Used to distinguish conceptually a "whole" from a "part" so it is very similar to the normal association.
  - James Rumbaugh suggests that if in doubt use the simple association.
  - Represented in code as the normal association.



Advanced adornments

Adornment	**Qualification**
Meaning	It is an association attribute whose values identify a subset of objects (usually a single object) related to an object across an association.
UML	Rendered as a small rectangle attached to the end of an association. Multiplicity is *after* applying the qualifier.
Java	`Map` collections let you implement associations in which the index to navigate through them is another object.



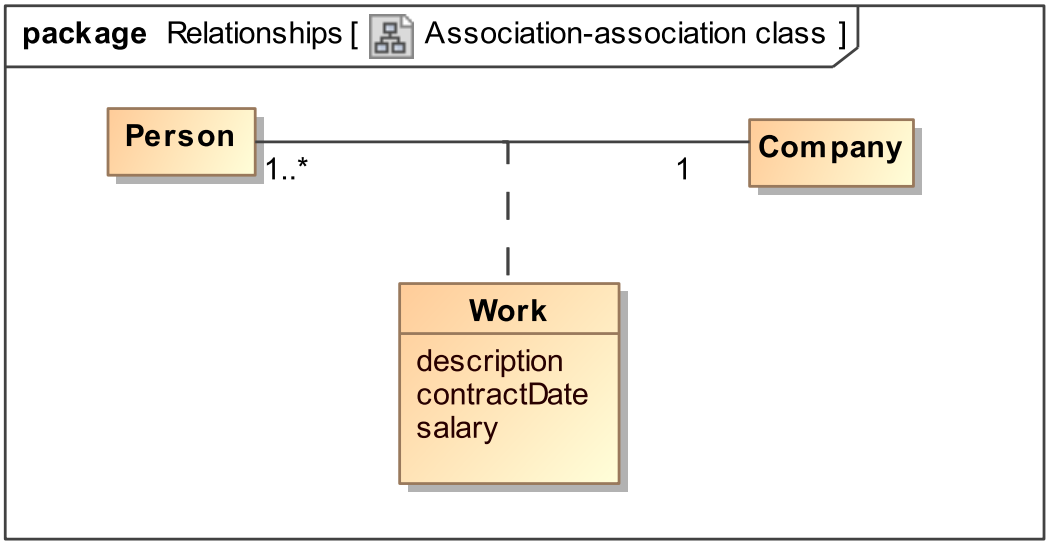
Advanced adornments

Qualification in Java

```
class Bank {
    Map<String, Client> clients = new HashMap<>();

    public void insertClient(String num, Client c) {
        clients.put(num, c);
    }

    public Client getClient(String num) {
        return clients.get(num);
    }
}
```

Advanced adornments

Adornment	**Association classes**
Meaning	In an association between two classes, the association itself might have properties. These properties are modeled as a class connected to the relationship.
UML	Rendered as a class symbol attached by a dashed line to an association line.
Java	Association is implemented through the association class and not through attributes in each class.



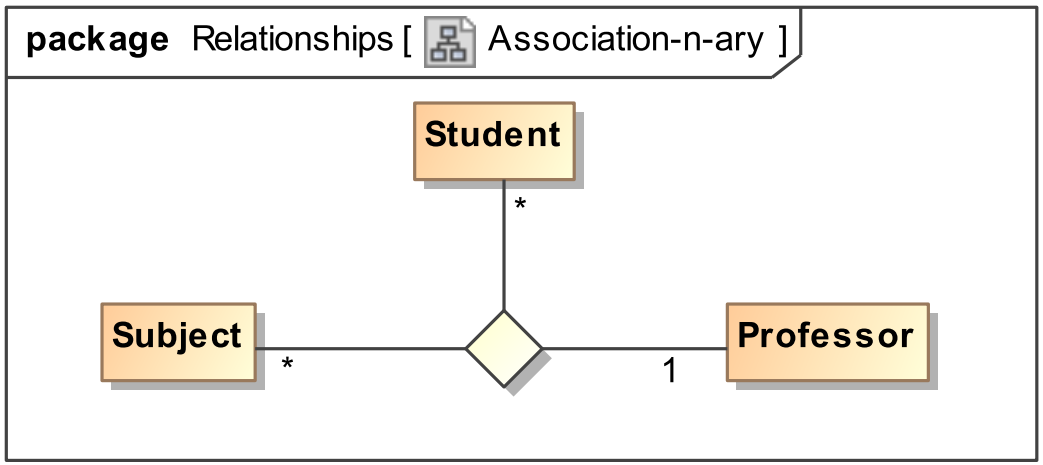
Advanced adornments

Association classes in Java

```
class Person {
    Work work;
}

class Company {
    List<Work> works;
}

class Work {
    String description;
    Date contractDate;
    int salary;
}
```

Advanced adornments

Adornment	**N-ary associations**
Meaning	Associations between three or more classes.
UML	Drawn as a diamond (larger than a terminator on a line) with a solid line for each association end connecting the diamond to the classifier that is the end's type. Multiplicity of an end is referred to be the multiplicity with respect to the tuple consisting of the other two values (for a tuple student-subject there is only one professor).
Java	Using association classes.



Advanced adornments

N-ary associations in Java

```
public class Professor {
    // ...
}

class Student {
    // ...
}

class Subject {
    // ...
}

class Teaching {
    Student student;
    Professor professor;
    Subject subject;
}
```

Dependency relationship

Dependency relationship

A dependency is a use relationship, specifying that a change in the specification of one thing (for example, a provider) may affect another thing that uses it (for example, a client) but not inversely.

- **Notation**
  - Rendered as a dashed line, directed to the thing that is dependent on.
  - If you want to be more specific, UML defines a number of stereotypes that may be applied to dependency relationships: `use`, `create`, etc.

Dependency relationship

- **Meaning**
  - Dependency is called a supplier-client relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the supplier element(s).
- **Dependency vs. other relationships**
  - All relationships, including generalization, association, and realization, are conceptually kinds of dependencies.
  - Generalization, association, and realization have enough important semantics about them that they are treated as distinct kinds of relationships in the UML.
  - A common strategy is to model generalization, association, and realization first, then view all other relationships as kinds of dependencies.

Dependency relationship

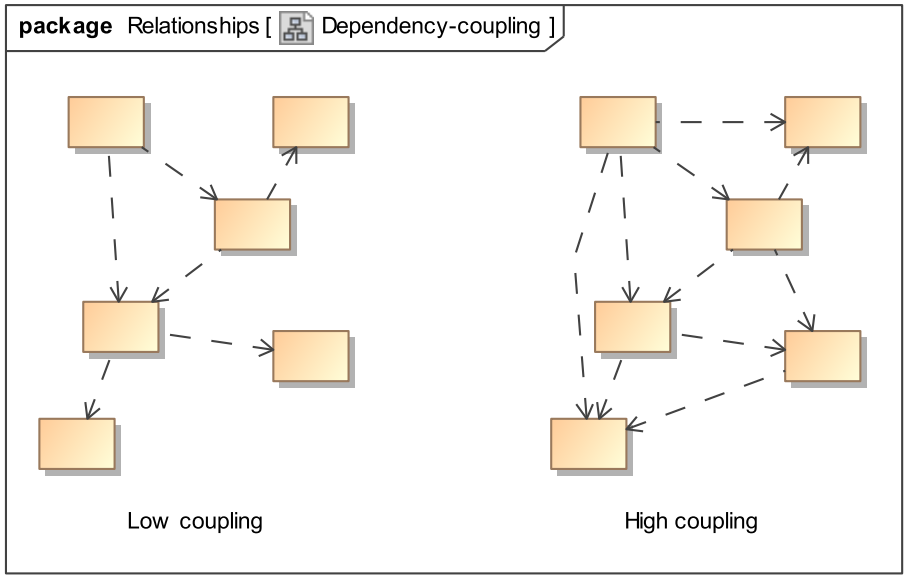
- **In code...**
  - There are different ways you can render a dependency in code.
  - For example: `Provider` class is a parameter or a return type of a method of `Client` class. Or `Client` class calls a static method of `Provider` class, etc.
  - **`Client` does not have an attribute of the `Provider` class $\Rightarrow$ it will be probably *Association***

Dependency in Java

```
class Provider {
    public static void staticMethod(){}
}
class Client {// Without Provider attrib.
    // Passing arguments
    public void methodX(Provider p) { }

    // Returning values
    public Provider methodY() { }

    public void methodZ() {
        // Local variables
        Provider p1 =
            Factory.getProvider();

        // Calling static methods
        Provider.staticMethod();

        // Creating instances
        Provider p2 = new Provider();
    }
}
```

Dependency relationship

- Typically, you don't render every possible dependency, because the class diagram would lose legibility.
- However, it is necessary to represent those dependencies that bring interesting meanings (e.g. classes that may appear unrelated to any other class unless by the dependency).
- Dependency can be also useful to detect high coupling between classes.



Exercise

How do you render this code in UML?

```
class MyClass {
    public int attribute1
    public String attribute2
    public AnotherClass attribute3
}
```

Summing up...

- When an attribute of a class appears in the code of another class, it generally represents the implementation of an association that should appear in the class diagram.
- This does not apply to primitive data types or to classes that behave as primitive data types (as `String`, `Date`, `BigDecimal`, etc.).
- **Important:** If the attribute is rendered as an association the name of the attribute is the name of the role that class plays in the association.
- **Never render a Java attribute as a UML attribute and an association at the same time**.

Exercise

Exercise: Card game

UML diagram of `Deck` and `Card` classes

Table of Contents

Interaction diagrams

- The main interaction diagrams of UML are **sequence diagrams** and **communication diagrams**.
- They are semantically equivalent to each other and, in theory, can be converted without losing information (some tools do that automatically).
- In practice, there are differences between the two diagrams with regard to how to show information: sequence diagrams focus on the temporal ordering of messages whereas communication diagrams focus on the structural organization of objects that send and receive messages.

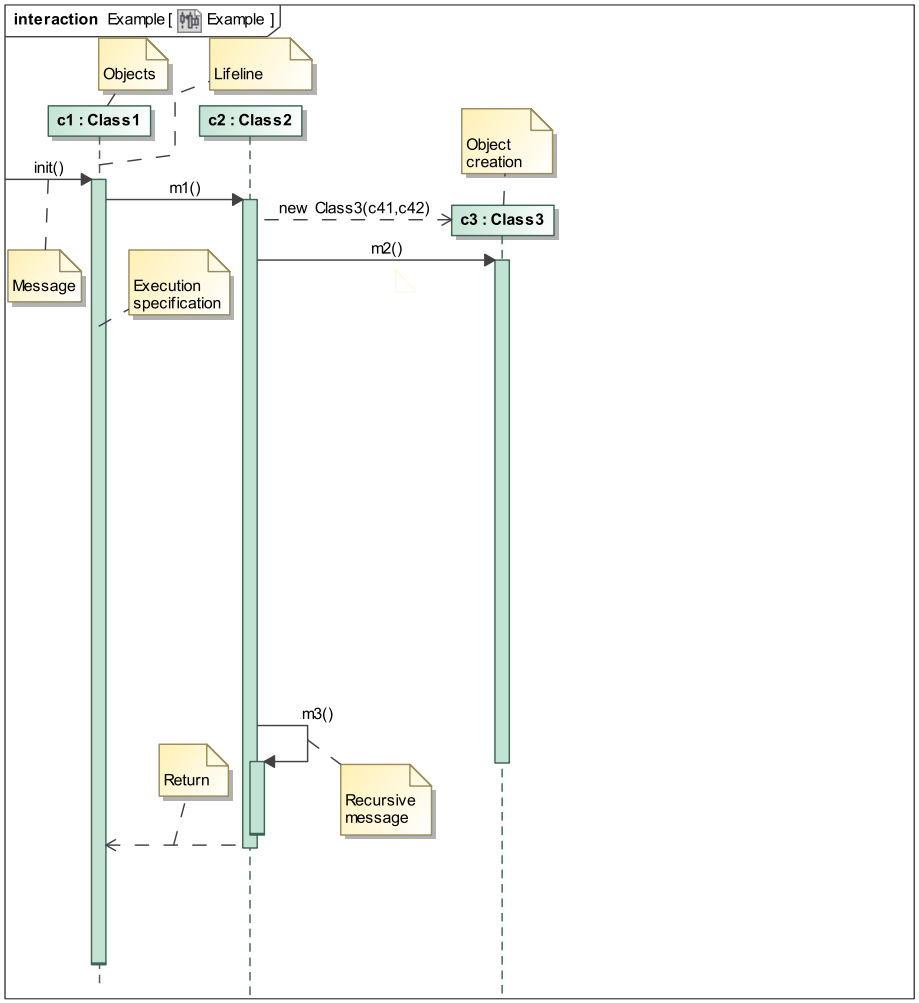
Sequence diagrams

Sequence diagrams

Describes an interaction by focusing on the sequence of messages that are exchanged, along with their corresponding occurrence specifications on the lifelines.

- **Advantages**: The temporal sequence of messages is easy to follow.
- **Drawbacks**: They have a rigid structure and it can be difficult to render interactions with multiple objects and messages.

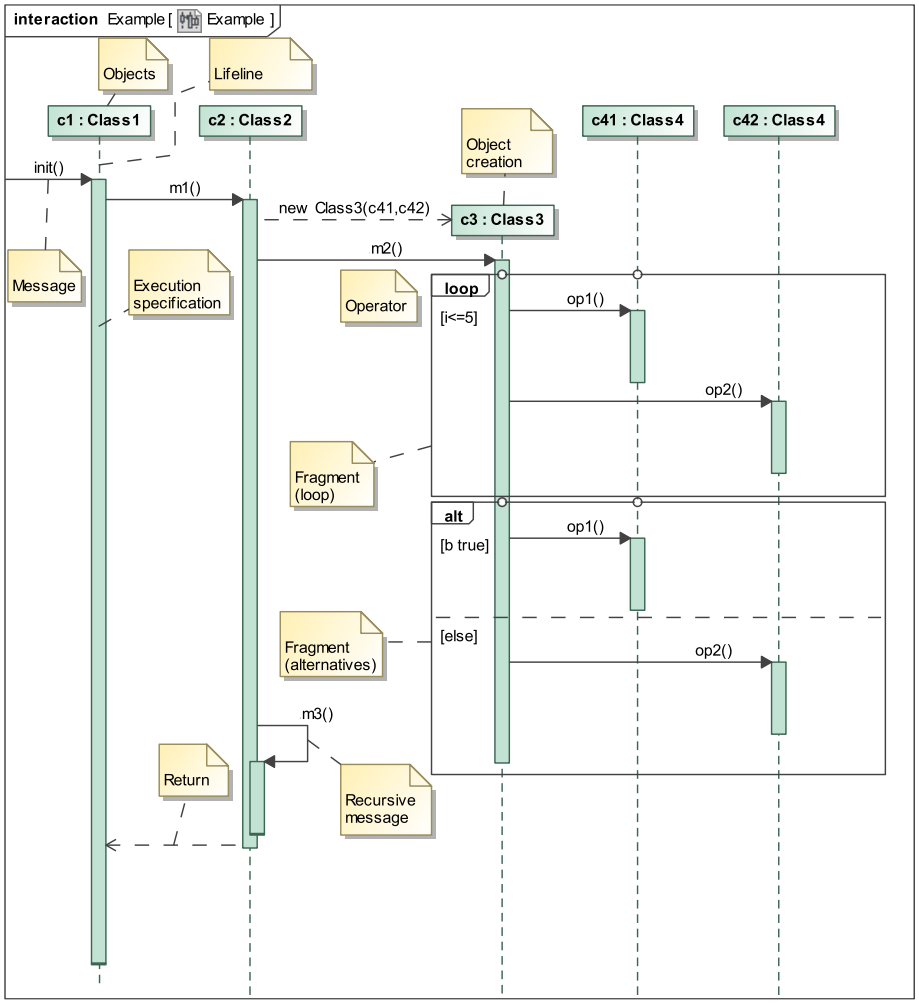
Sequence diagram



Example (I)

```
class Diagram {
  public static void main(String[] args) {
    Class4 c41 = new Class4();
    Class4 c42 = new Class4();
    Class2 c2 = new Class2(c41, c42);
    Class1 c1 = new Class1(c2);
    c1.init();
  }
}

class Class1 {
  Class2 c2;
  public Class1(Class2 c2) { this.c2 = c2; }
  public void init() { c2.m1(); }
}

class Class2 {
  Class4 c41, c42;
  public Class2(Class4 c41, Class4 c42) {
    this.c41 = c41;
    this.c42 = c42;
  }
  public void m1() {
    Class3 c3 = new Class3(c41, c42);
    c3.m2();
    this.m3();
  }
  public void m3() { }
}
```

Sequence diagram



Example (II)

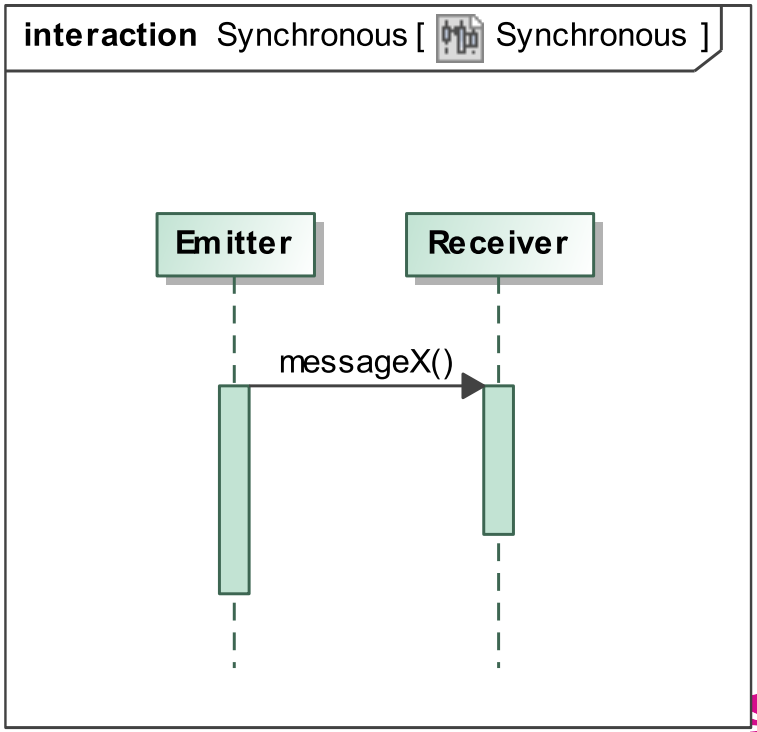
```
class Class3 {
  Class4 c41, c42;
  public Class3(Class4 c41, Class4 c42) {
    this.c41 = c41;
    this.c42 = c42;
  }

  public void m2() {
    boolean b = false;

    for (int i = 1; i <= 5; i++) {
      c41.op1();
      c42.op2();
    }

    if (b)
      c41.op1();
    else
      c42.op2();
  }
}

class Class4 {
    public void op1() { }
    public void op2() { }
}
```

Messages

Synchronous messages

The caller must wait until the message has finished.

- **Notation**: A solid line with a filled arrowhead.
- **In code**: It is the traditional way to invoke a method.

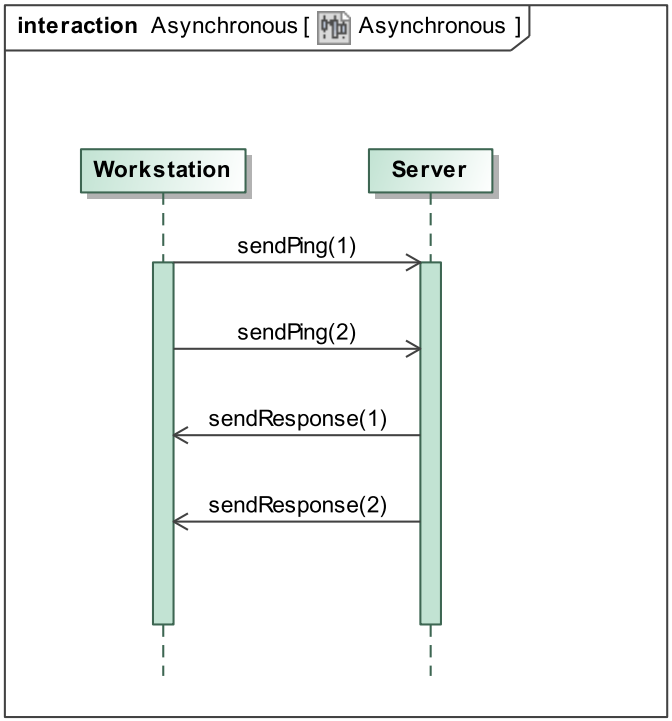


Messages

Asynchronous messages

The caller can continue and does not have to wait for a response.

- **Notation**: A solid line with an open arrowhead.
- **In code**: There are asynchronous calls in multithreaded applications and in message-oriented middleware.

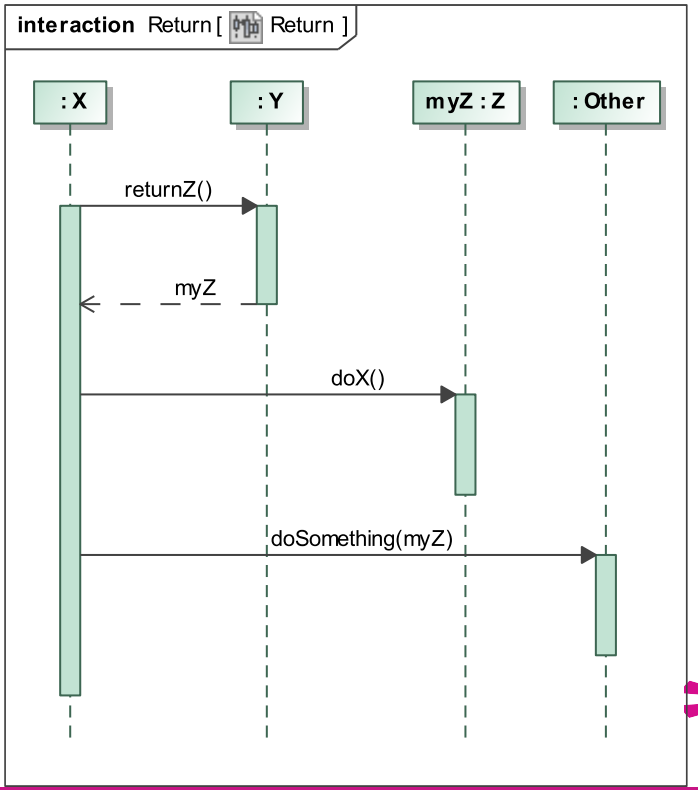


Messages

Return messages

Optional messages at the end of the execution bar that represent the return of control to the caller object.

- **Notation**: Shown as a dashed line with an open arrowhead.
- Only represented in a diagram if they add value, for example, emphasizing that the object obtained in the return is the object that is passed as a parameter in the next message.
- **In code**: `return` messages represent returning the control to the caller.

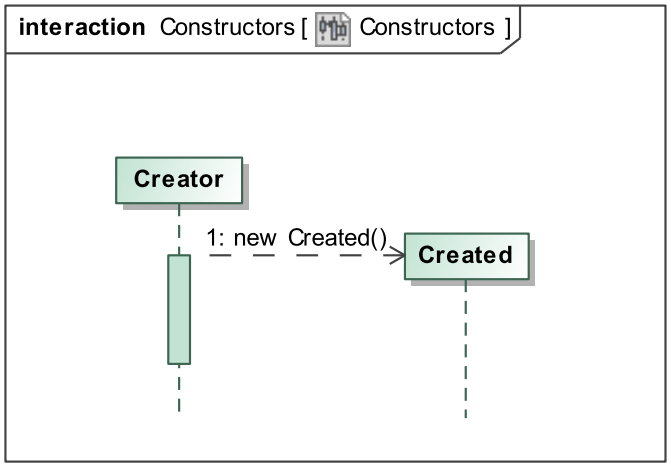


Messages

Creation messages

Messages that represent the creation of new objects during the interaction.

- **Notation**: It is shown as a dashed line with an open arrowhead (looks the same as reply message), and pointing directly to the created object instead of its lifeline.
- **In code**: Constructors called through the `new` operator are an example of creation messages.

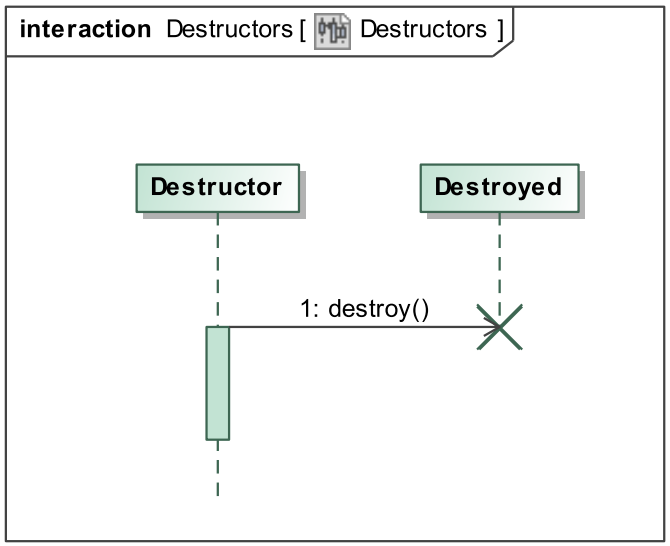


Messages

Destruction messages

Messages that represent the destruction of an object terminating its lifeline.

- **Notation**: A solid line ended in a diagonal cross at the bottom of the lifeline.
- **In code**: There are no destruction messages in Java, only a garbage collection mechanism. Other languages have explicit destruction messages.

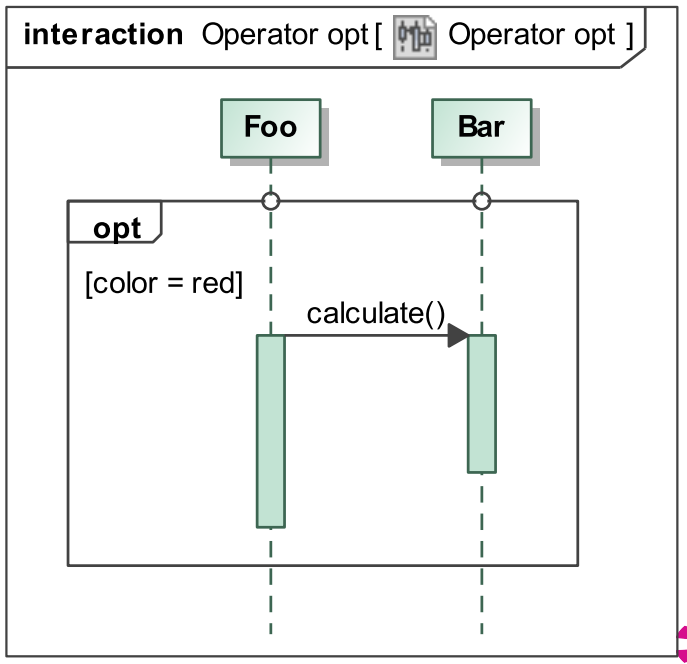


Fragments

Fragments

Combined fragments allow representing more complex flows of control than a simple sequential flow.

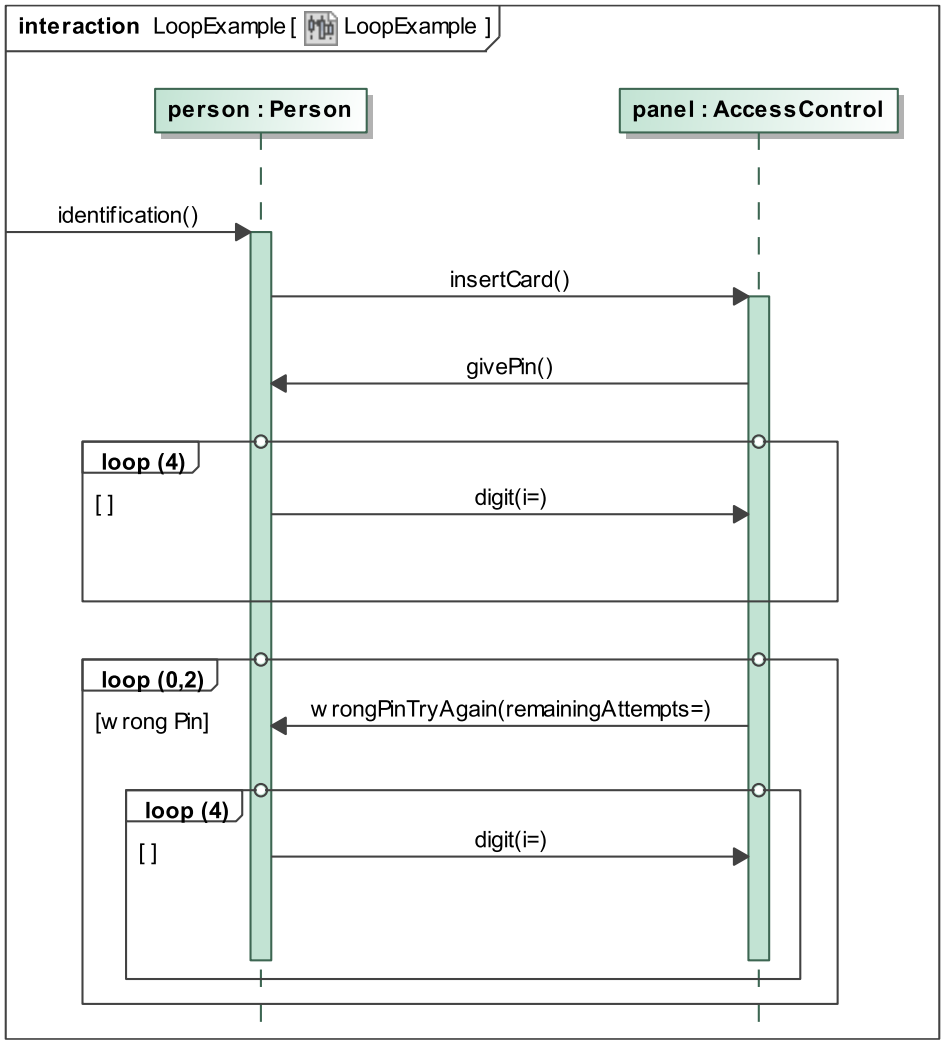
- **Notation**: A rectangle with a small pentagon in the upper left corner containing the interaction operator.
- For example, `opt` represents simple conditional sentences. The interaction will only occur if the condition is met.



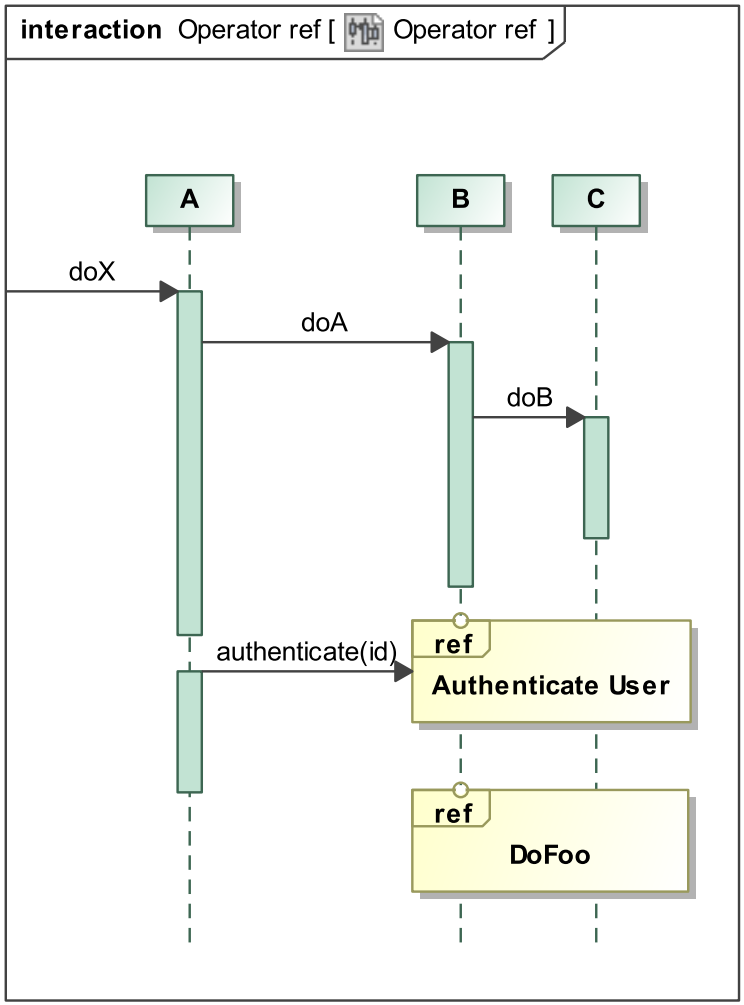
Fragments

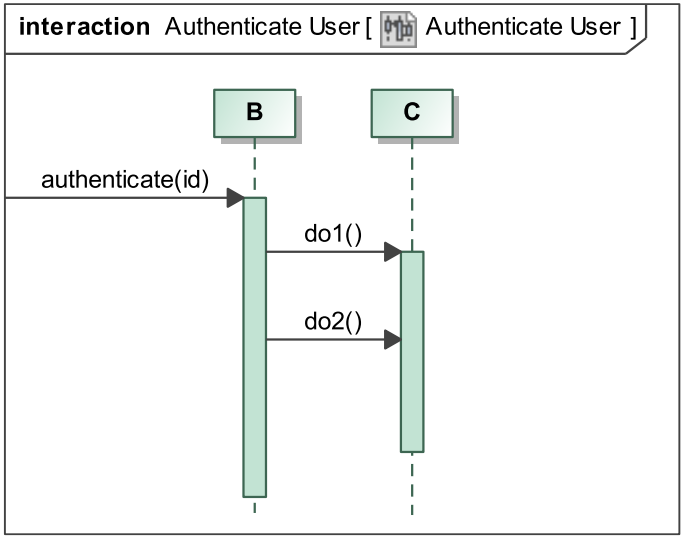
Operator	Parameter	Description
`alt`	`[guard1]...` `[guard2]...` `[else]...`	The combined fragment represents a choice or alternative behaviors. Corresponds to a `switch` or an `if-then-else`.
`opt`	`[guard]`	The interaction only occurs if `guard` is true. Corresponds to an `if-then`.
`loop`	`min, max,` `[guard]`	Loop will iterate at least the `min` number of times and at most the `max` number of times, as long as the `guard` is met. If they are the same we state only one of them. `min` and `max` follow the operator in parentheses.
`ref`		Refers to a separate interaction. Similar to the idea of calling a function.

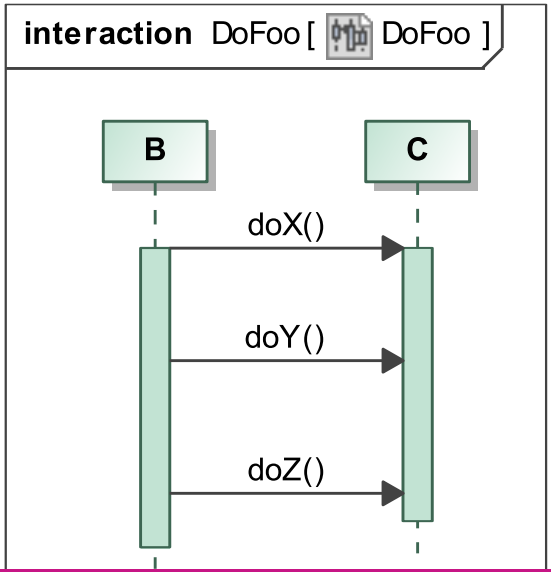
loop operator



ref operator







Exercise

Exercise: Card game

What is the sequence diagram of the constructor of `Deck`?

Polymorphic calls

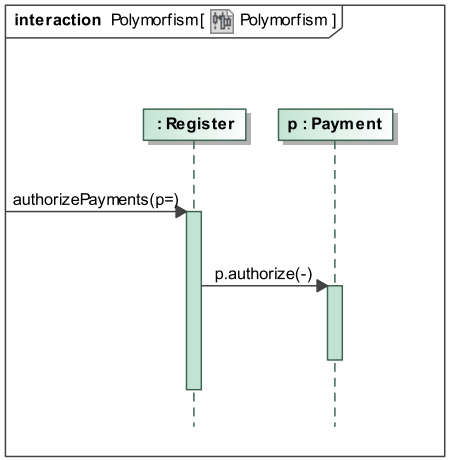
Given these classes

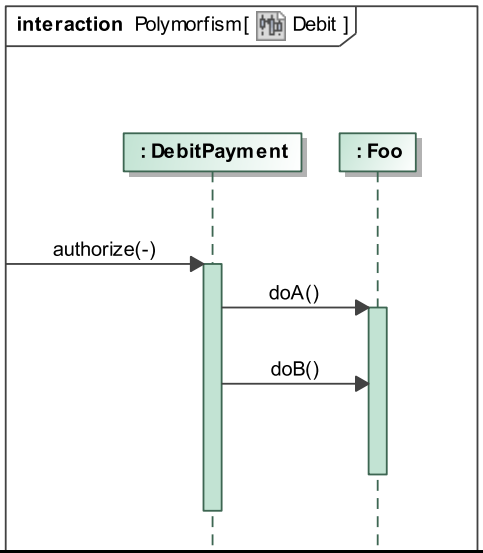
```
abstract class Payment {
    public abstract void authorize();
}

class DebitPayment extends Payment {
    Foo foo;
    @Override
    public void authorize() {
        foo.doA();
        foo.doB();
    }

}

class CreditPayment extends Payment {
    Bar bar;
    @Override
    public void authorize() {
        bar.doX();
    }

}
```

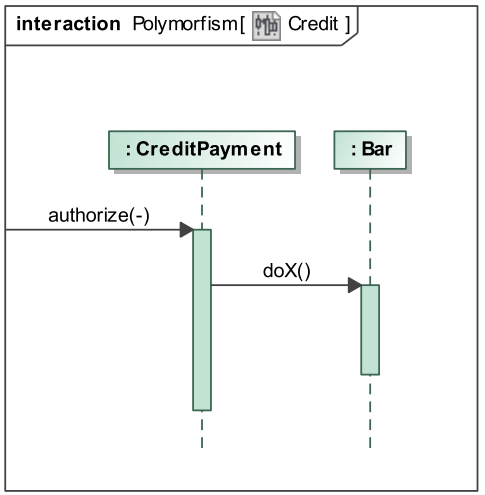
How do you represent the following call to `authorizePayments()`?

```
class Register {
    public void authorizePayments(Payment p)
        p.authorize();
    }

    public static void main(String[] args){
        Register r = new Register();
        r.authorizePayments(new DebitPayment(
    }
}
```

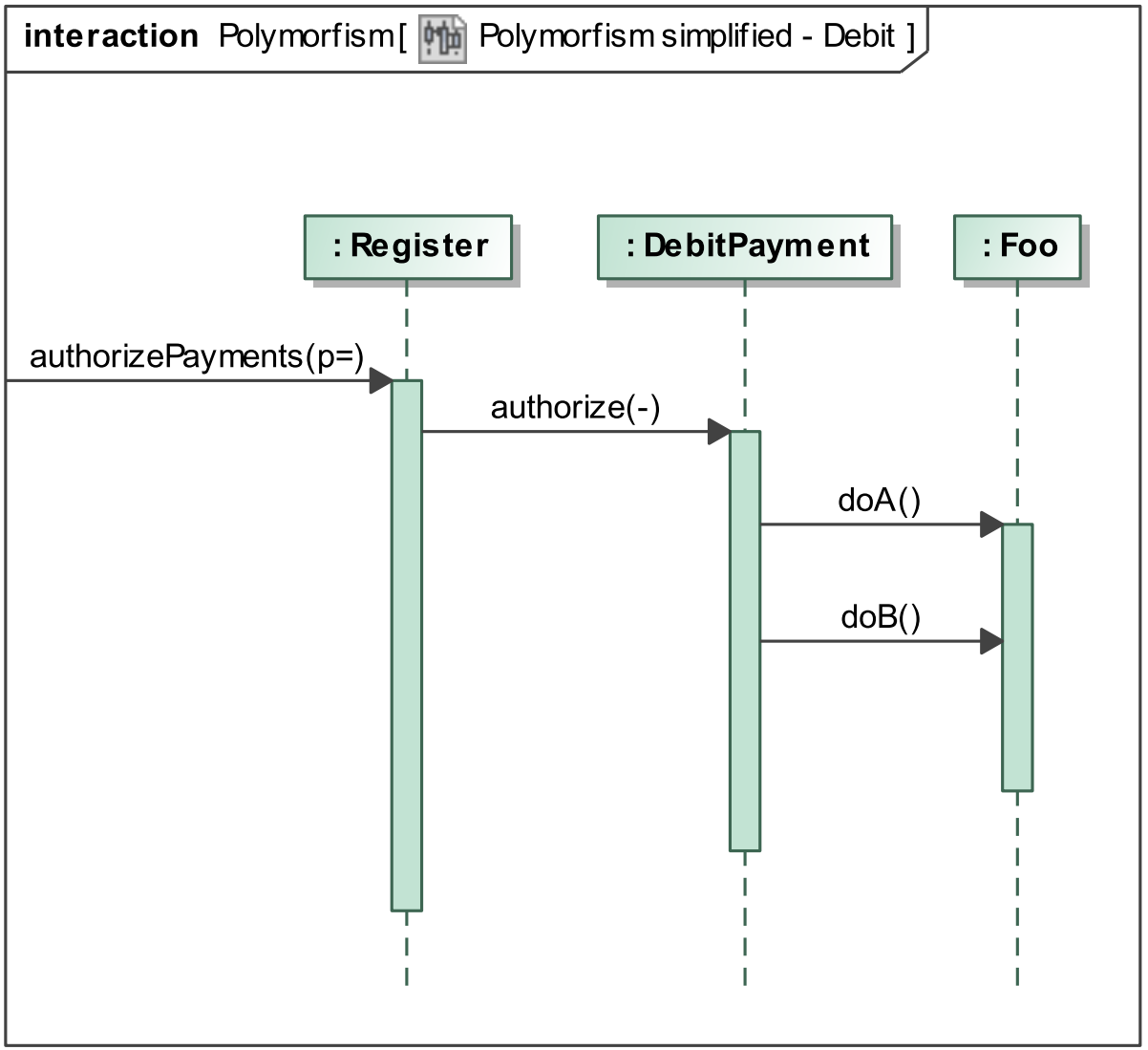
Polymorphic calls - accurate representation







Polymorphic calls - simplified representation



Communication diagram

Communication diagram

Shows interactions between objects using sequenced messages in a free-form arrangement.

- **Advantages**: Having a less rigid arrangement. Communication diagrams are more compact and readable than sequence diagrams.
- **Drawbacks**: The sequence of messages is more difficult to follow and elements such as loops or alternatives are more difficult to represent.

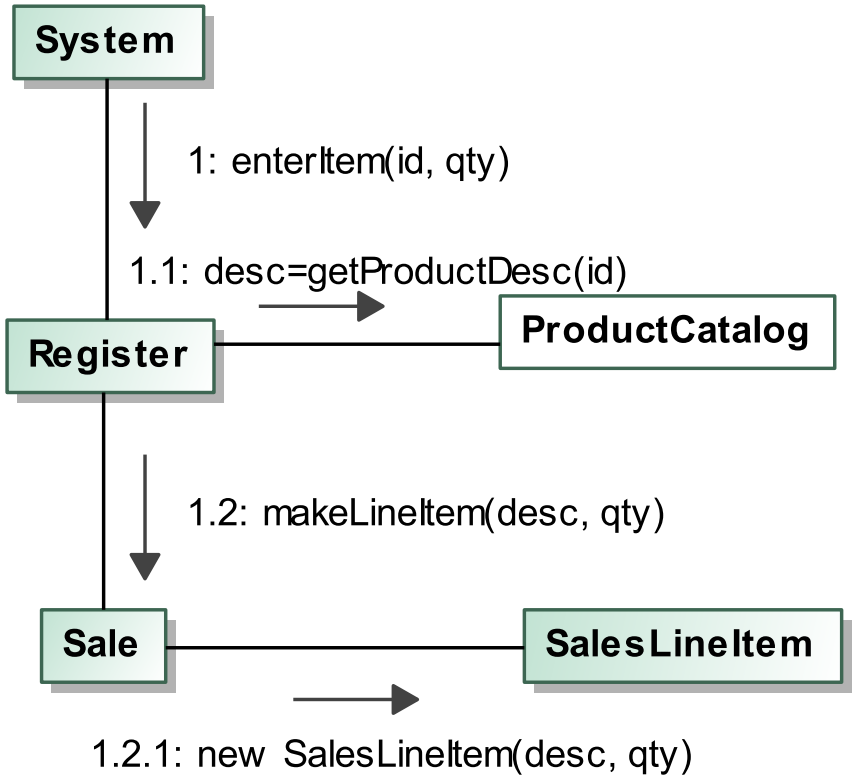
Communication diagram elements

- **Objects**
  - As in sequence diagrams, diagrams consist of objects that exchange messages.
- **Connectors**
  - When two objects exchange messages there is a solid line called "connector" between them.
- **Messages**
  - A message is shown as a line with a sequence expression and an arrow above the connector. The arrow indicates direction of the communication.
  - The sequence expression follows the Dewey scheme: 1, 1.1, 1.2, 2, etc.

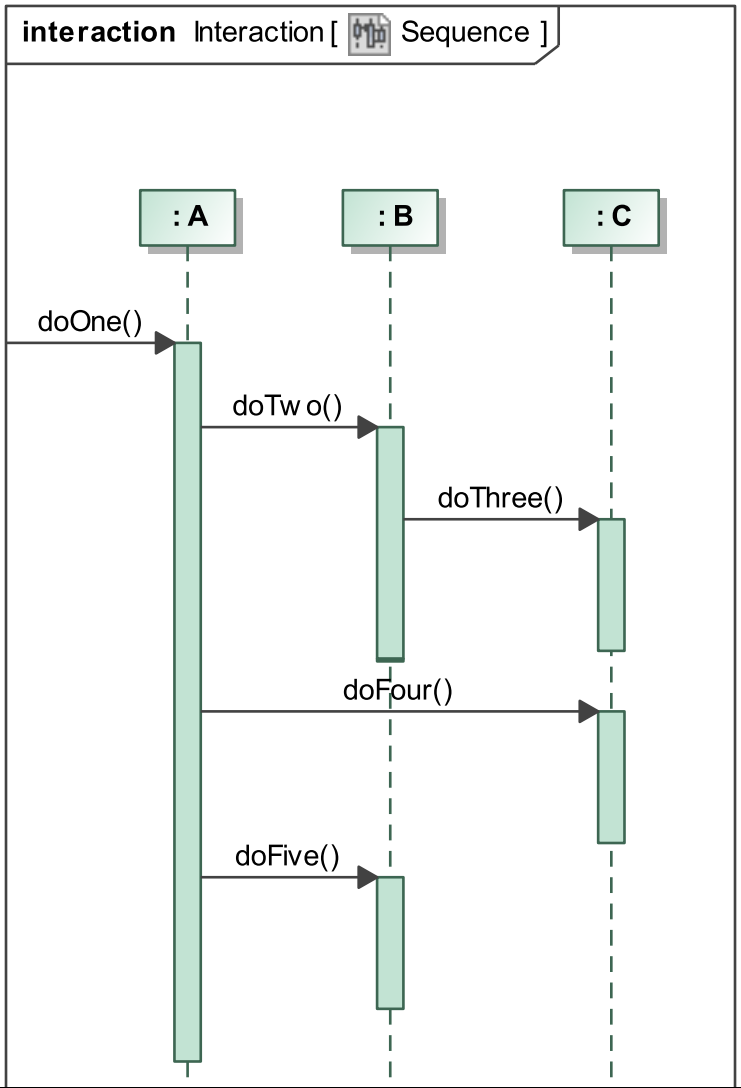
Communication diagram elements

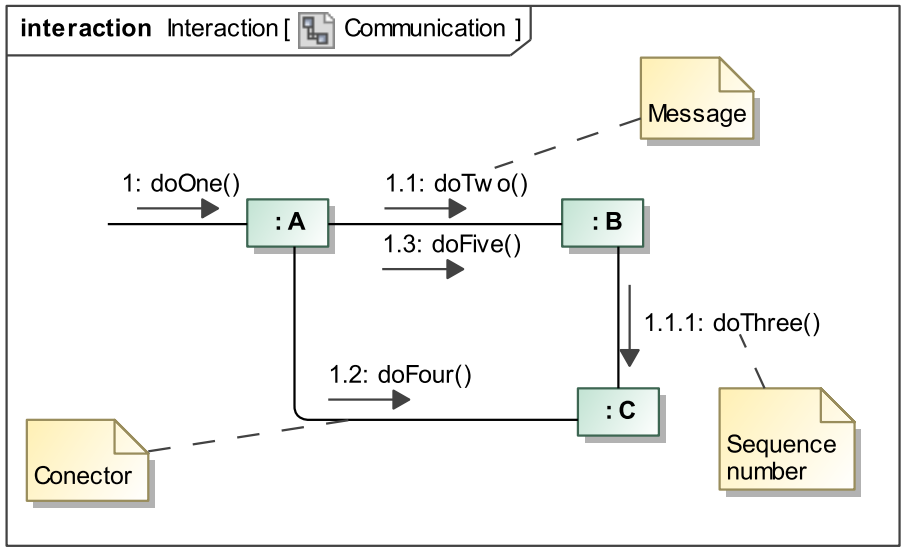
Code

```
class System {
// ...
register.enterItem(id, qty)
}

class Register {
    enterItem(int id, int qty) {
        String desc =
              catalog.getProductDesc(id);
        sale.makeLineItem(desc, qty);
    }
}

class Sale {
    makeLineItem(String desc, int qty) {
        sl = new SalesLineItem(desc, qty);
    }
}
```



Sequence diagrams vs. communication diagrams





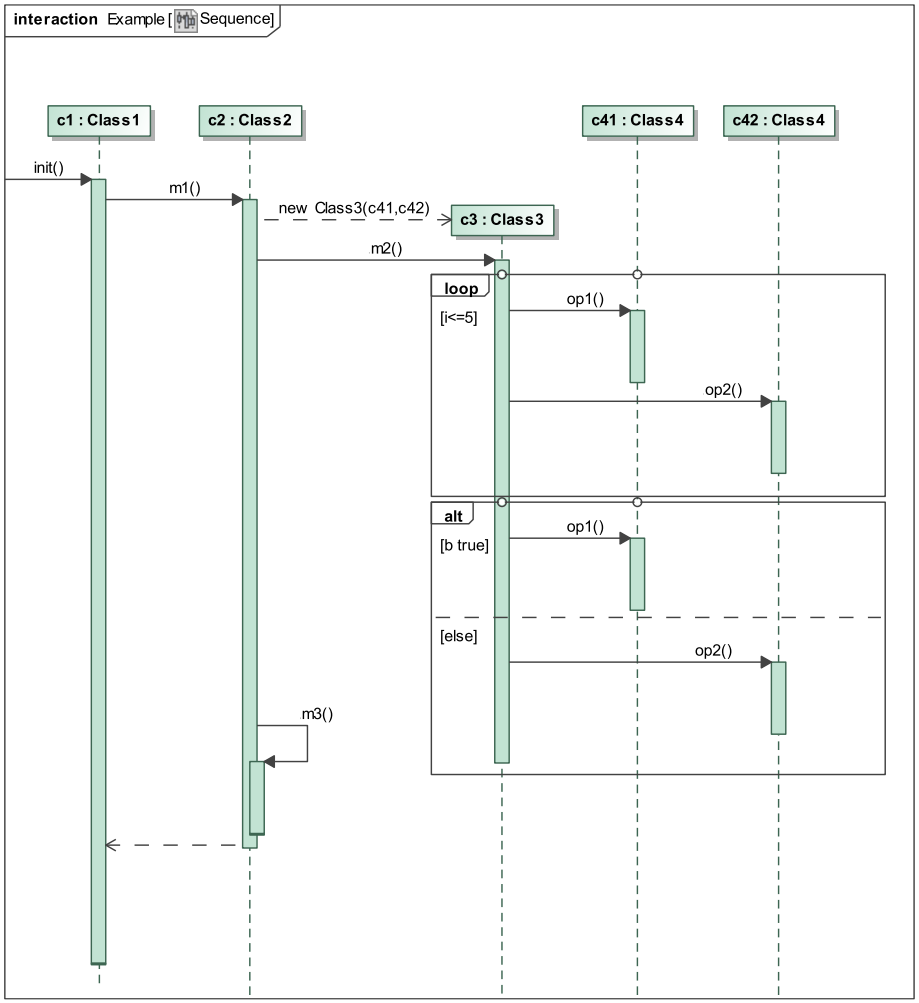
Iterative and conditional messages

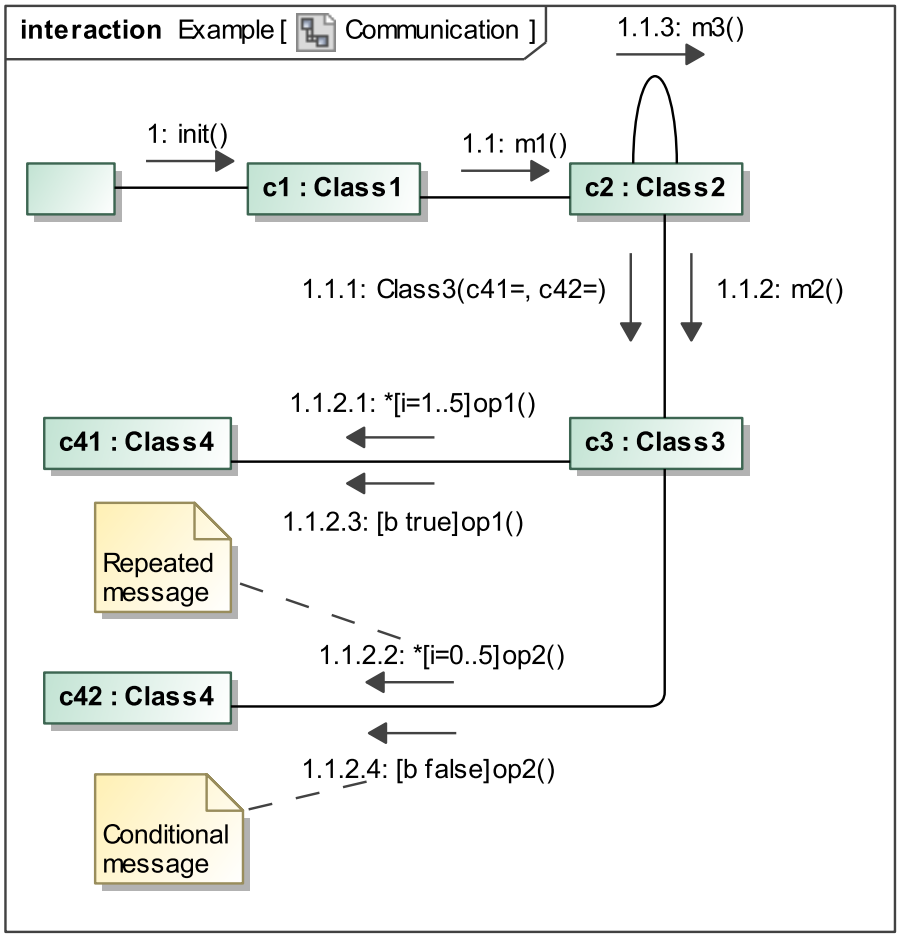
- Communication diagrams include modifiers that allow defining iterative and conditional messages.
- **Iterative messages**
  - They are messages that are sent repeatedly form one object to another.
  - Rendered as an asterisk (`*`) followed by a condition in square brackets indicating how many times the message is repeated.
  - UML does not specify how this condition should be stated, it could be done in natural language but more often is specified as a range of values (loop `for`) or a logical condition (loop `while`).
  - Example: `*[i=1..5]` (The message is repeated 5 times)

Iterative and conditional messages

- **Conditional messages**
  - They include a guard value in square brackets. The message will be only sent if the guard is true.
  - Example: `[b true]` (the message will be only sent if `b` is true).
- **Important to keep in mind**
  - The ability of communication diagrams to represent complex loops and conditional statements is limited.
  - If you want to represent such structures it would be better to choose a sequence diagram with combined fragments.

Sequence diagrams vs. communication diagrams

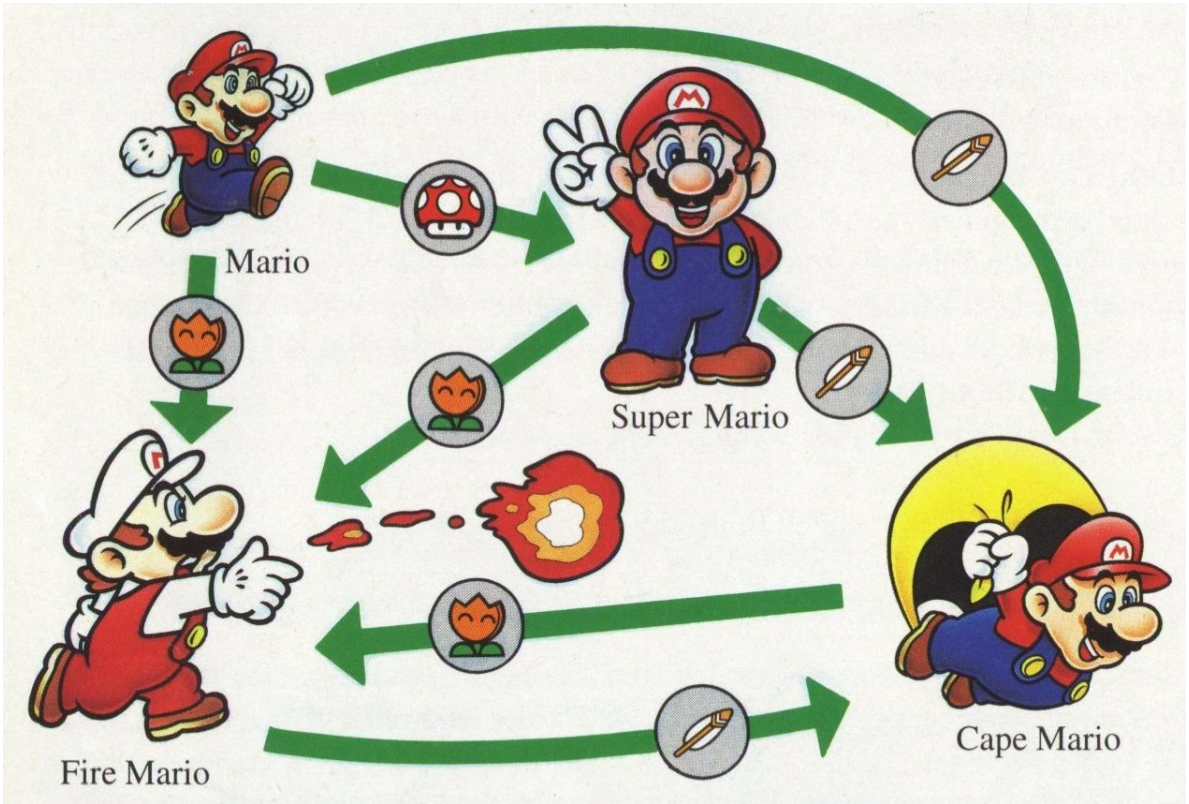




State machine diagram

State machine diagram

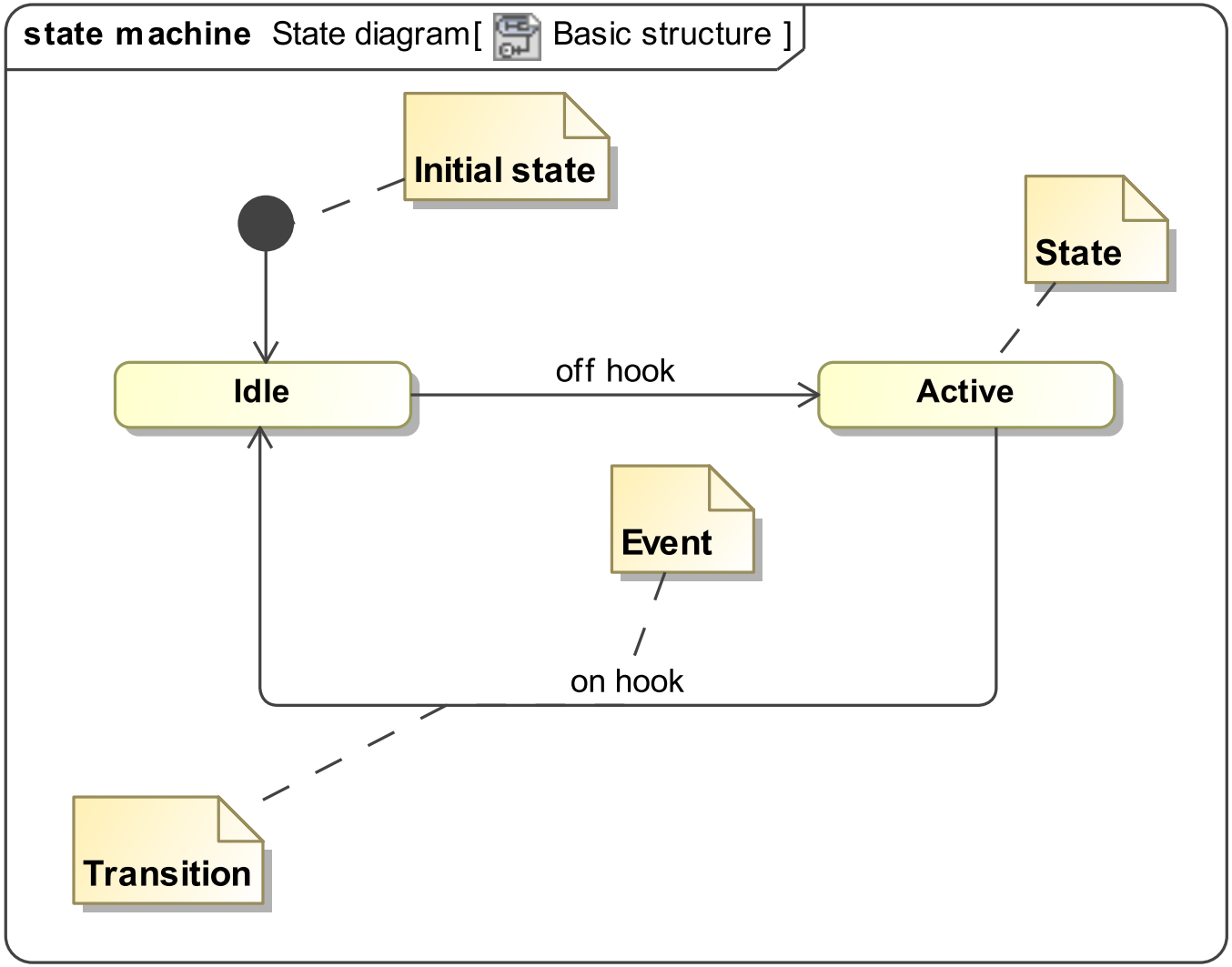
Illustrates the states of an object, the events that it may receive and the behavior of that object in reaction to an event.



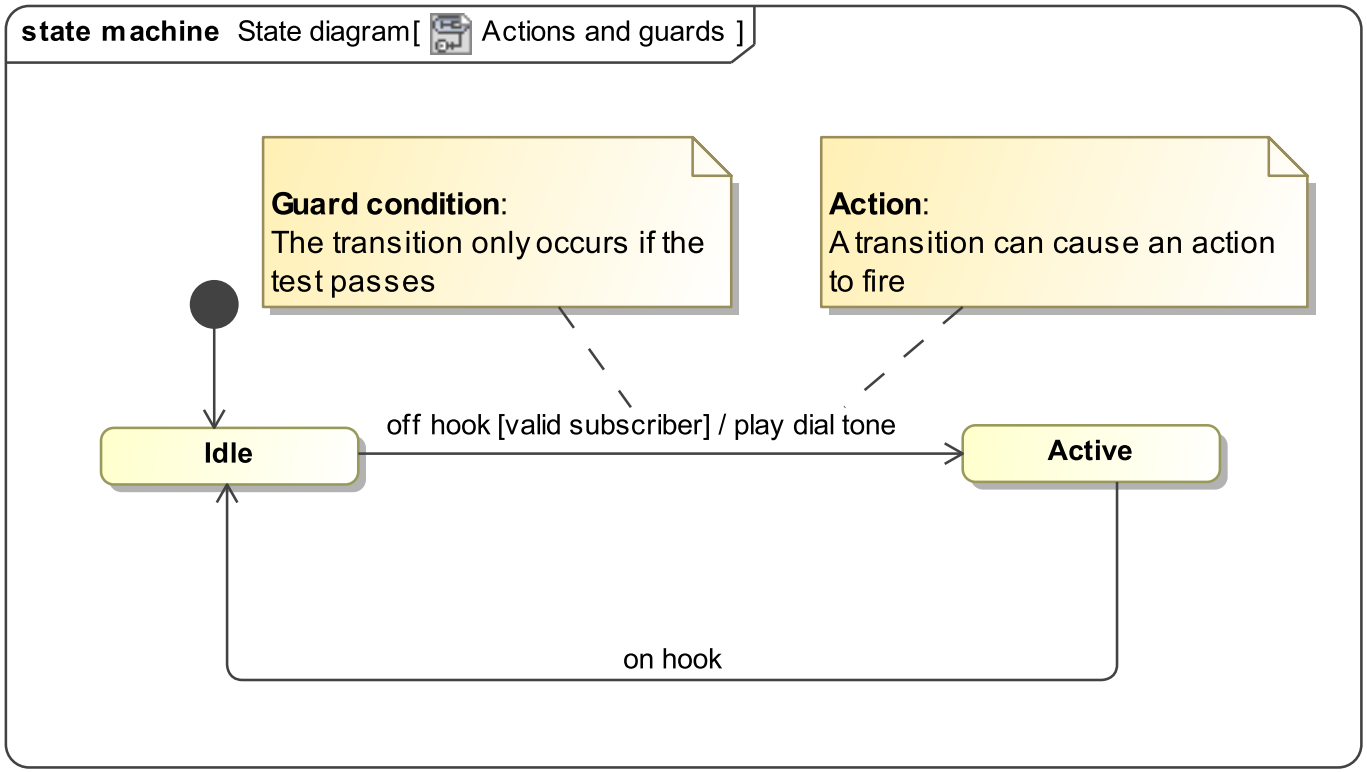
Elements of the state machine diagram

- **Events**:
  - Significant or noteworthy occurrences.
- **States**:
  - Condition of an object at a moment in time (i.e., between events).
- **Transitions**:
  - Relationships between two states indicating that the object moves from one state to another when an event occurs.
- **Another additional elements**:
  - Actions, guards, nested states, internal activities, initial and final states, etc.

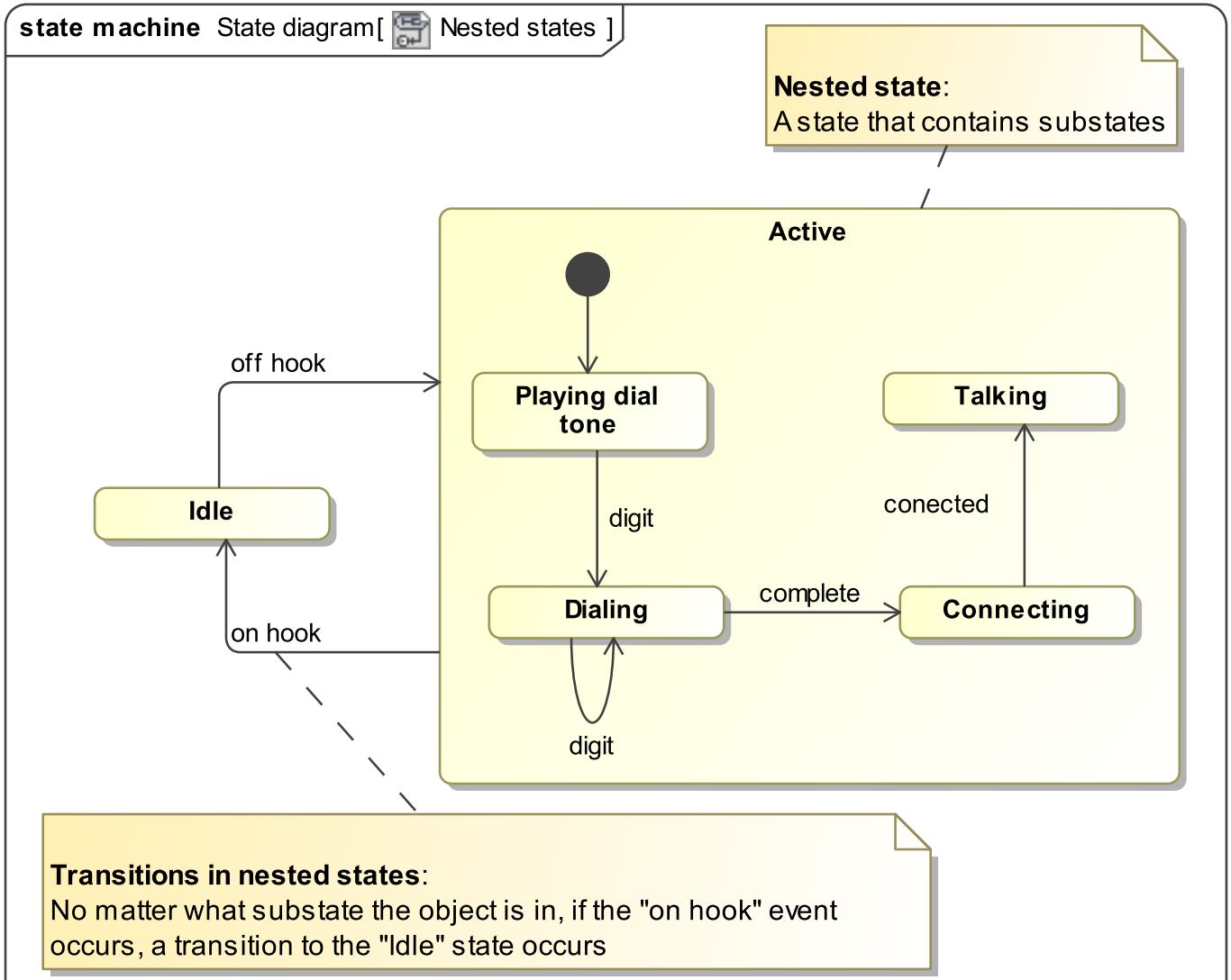
State machine diagram structure



Actions and guards



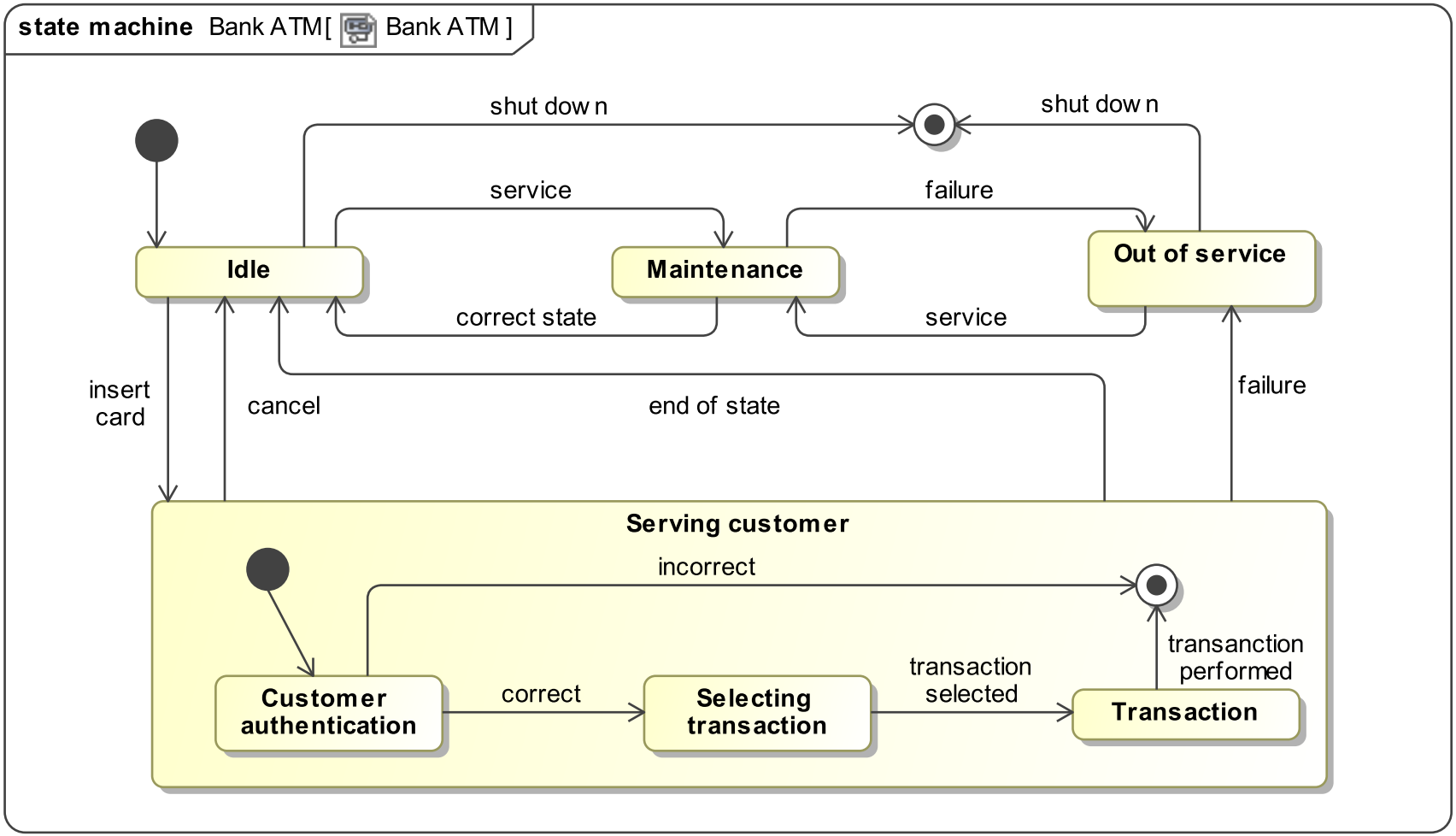
Nested states



Initial and final states

- **Initial state**
  - A pseudostate that designates the default starting state.
  - The transition to the starting state is automatic.
  - Displayed as a small black disc.
- **Final state**
  - A special state whose activation indicates that the enclosing state has completed activity.
  - If an object reaches its top-level final state, the state machine terminates.
  - Displayed as a small black disc surrounded by a circle.

Initial and final states (example)



Internal activities

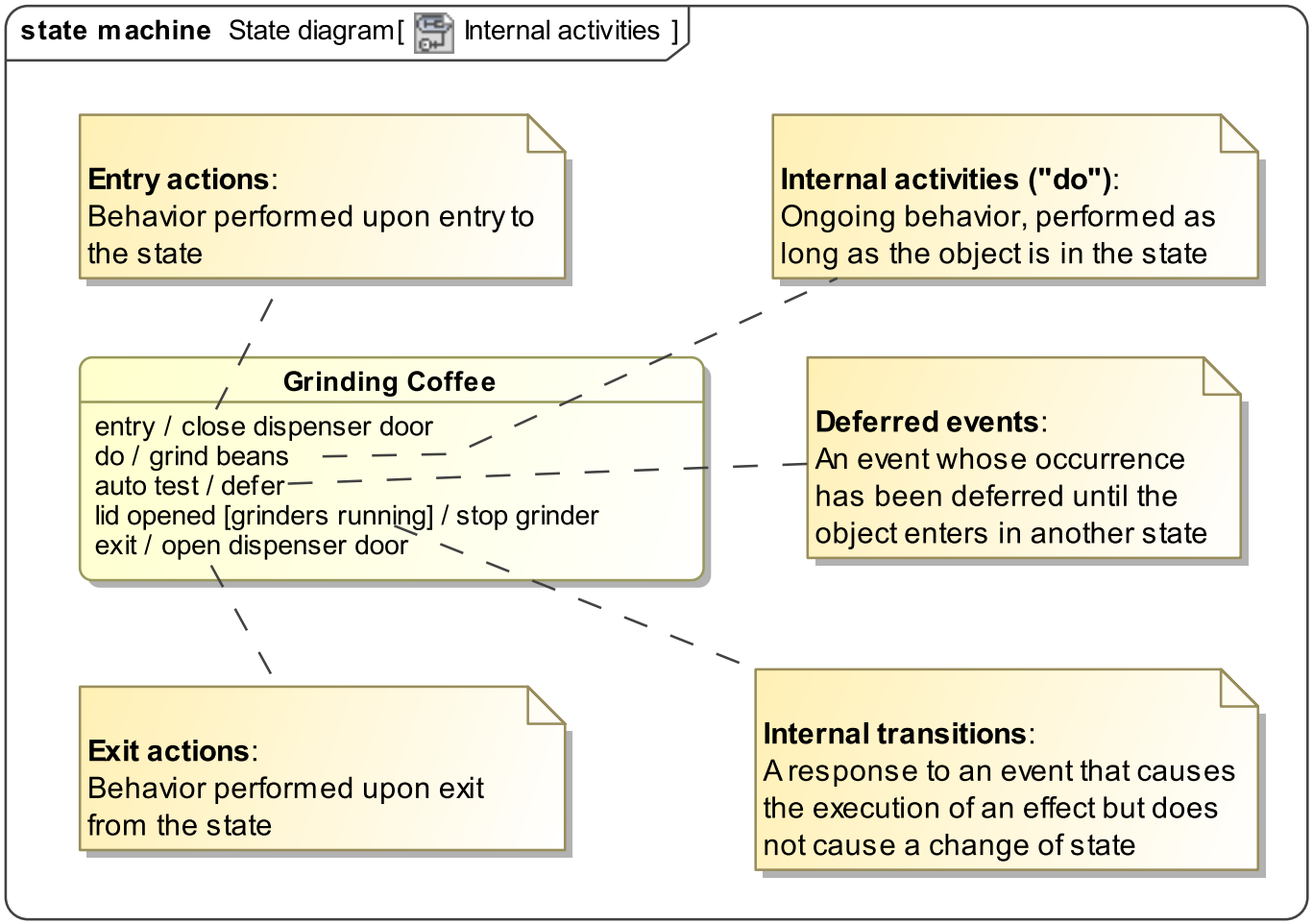


Table of Contents

Forward engineering

Forward engineering

Generating code from diagrams.

- **Advantages**:
  - You do not need to separately develop diagrams and code.
  - The fundamental structure and details of code are obtained from the model.
- **Drawbacks**:
  - UML models are semantically richer than object-oriented code, so you can probably lose information during the conversion.
  - The resulting code may not be the one we would have written.
  - A programming IDE is typically more usable than a UML tool.

Reverse engineering

Reverse engineering

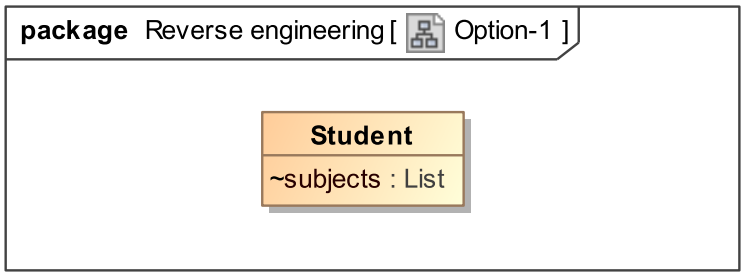
Generating diagrams from code.

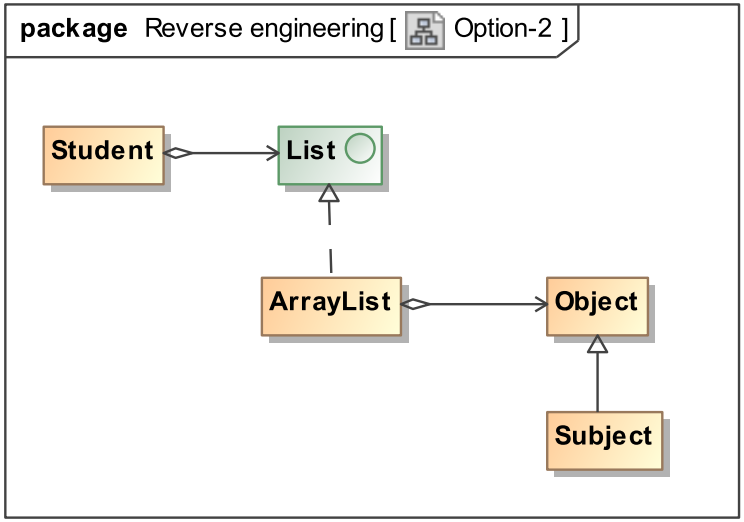
- **Advantages**:
  - It is a good way of documenting existing code and helps to visualize its general structure.
- **Drawbacks**:
  - The code has a great amount of low-level details that are not needed to develop useful models.
  - Normally you cannot recreate a complete model from a model since some modeling details are not included in the code.

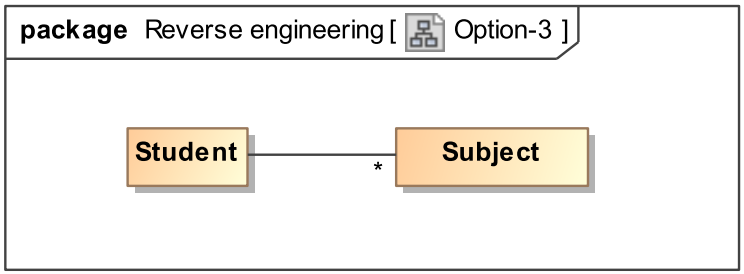
Reverse engineering problems

How to represent this code?

```
class Student {
    List subjects;
}
```





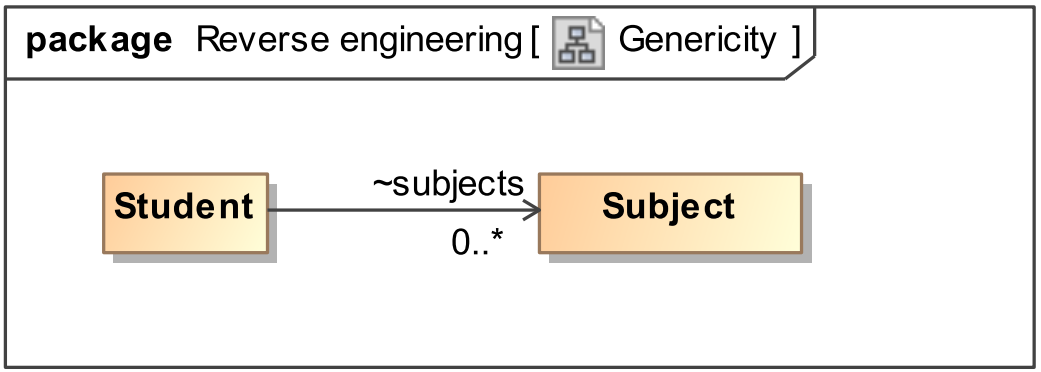


Reverse engineering problems

- Do generics help in the reverse engineering process?

How to represent this code?

```
class Student {
    List<Subject> subjects;
}
```



Round-trip engineering

Round-trip engineering

Generating UML diagrams and code in either direction. Ideally, changes are applied automatically and immediately.

- **Advantages**:
  - Models and code are kept in sync.
- **Drawbacks**:
  - Needs sophisticated tools.
  - Its functioning is not always appropriate because of the problems discussed in forward and reverse engineering.

UML modes

UML as Sketch

Informal and incomplete diagrams (often hand sketched on whiteboards) created to explore difficult parts of the problem or solution space, exploiting the power of visual languages.

- Agile modeling emphasizes *UML as sketch*.
- The aim is to use the sketches to help communicate ideas and alternatives.
- Their emphasis is on selective communication rather than complete specification.

UML modes

UML as Blueprint

Relatively detailed design diagrams used either for reverse engineering to visualize and better understand existing code in UML diagrams, or for code generation (forward engineering)

- Focuses on completeness. Blueprints aim to convey detailed information about the code.
- Blueprints require much more sophisticated tools than sketches in order to handle the details required for the task.
- Some tools use the source code itself as the repository and use diagrams as a graphic viewport on the code.

UML modes

UML as Programming Language

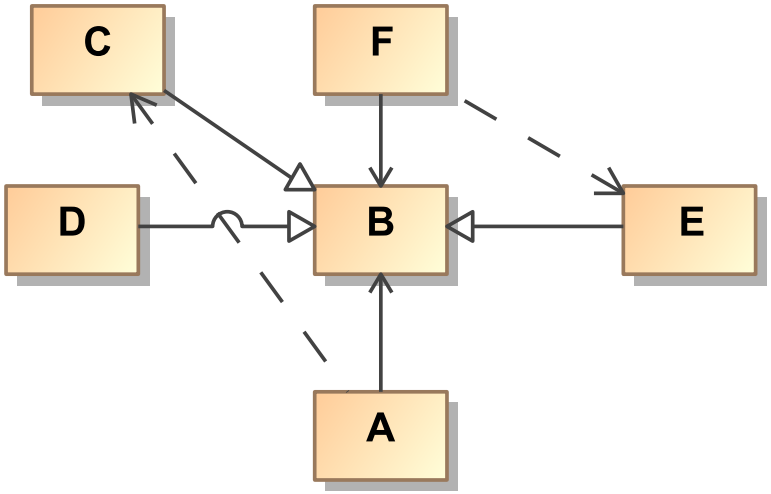
Complete executable specification of a software system in UML.

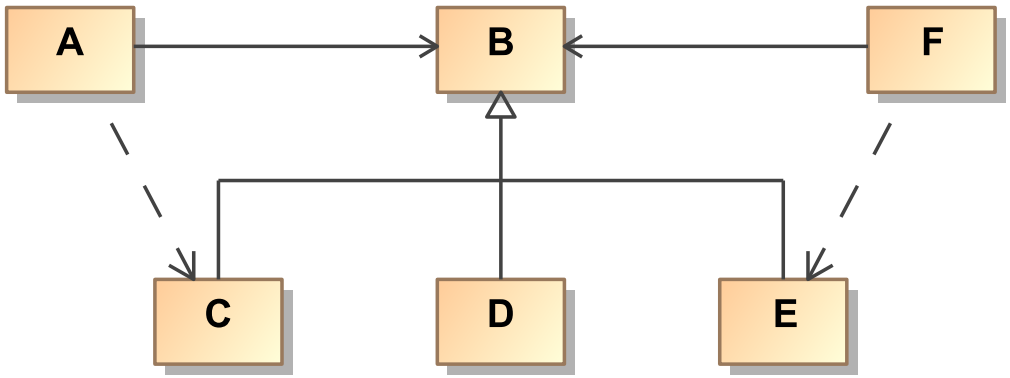
- Executable code will be automatically generated, but is not normally seen or modified by developers, who work only in the UML "programming language".
- Requires a practical way to diagram all behavior or logic (probably using interaction or state diagrams).
- It is still under development in terms of theory, tool robustness and usability.

UML modes

- **Martin Fowler's advice for using diagrams well:**
  - The primary value is communication. Effective communication means selecting important things and neglecting the less important.
  - The code is the best source of comprehensive information. For diagrams, comprehensiveness is the enemy of comprehensibility.
  - Use diagrams to explore a design before you start coding it, and treat the resulting design as a sketch, not as a final design. It's perfectly reasonable to throw these diagrams away.
  - If diagrams are used as an on-going documentation focus on diagrams that you can keep up to date without noticeable pain (simple diagrams, stabilized code, etc.) or use them as handover documentation (to new designers, customers, etc.).

Clarity matters...





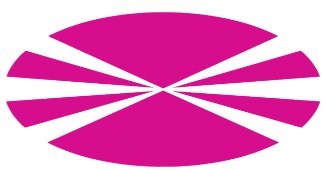
- Inheritance in vertical joining lines.
- Associations in horizontal.
- Do not cross lines.
- In short, the organization of the diagram should facilitate its understanding.

Unit 4: UML (Unified Modeling Language)

Software Design (614G01015)

David Alonso Ríos
Eduardo Mosqueira Rey (Coordinator)

Department of Computer Science, Faculty of Computer Science



UNIVERSIDADE DA CORUÑA