Моделиране на софтуерни системи. Унифициран език за моделиране (UML).

Първа Част

Models and modelling

- **Model**: (mathematical) presentation of structure and processes of a given system (used for analysis and planning)
- **Modeling**: process of describing of the system by means of its model (physical, conceptual, mathematical or based on imitation) and simulation of system activities by means of applying the model on a data set. Models represent real phenomenon that are difficult to observe directly System modeling is the process of developing abstract models of a system, with each model presenting a different view or perspective of that system.

System modelling

- System modeling is the process of developing abstract models of a system – existing or being under development, whereupon each model represents a different perspective (so called view) of the system.
- System models describe the system structure, behavior and functionality (external perspective of the system) and are used to communicate them with customers:
  - Static models – describe system structure, e.g. entity/relation data models.
  - Dynamic models – describe system behavior.

Models of existing and planned system

- Models of an existing system are used for collecting and describing the requirements. They model how the existing system functions and are applied for revealing its strengths and weaknesses. They provide a base for the requirements for a new version of the system.
- Models of a new system are used for design of the requirements in order to explain them to other stakeholders in the system. They help system engineers to select design proposals and document the system.
- Model engineering process – allows to generate a complete or partial implementation of the system from a system model.

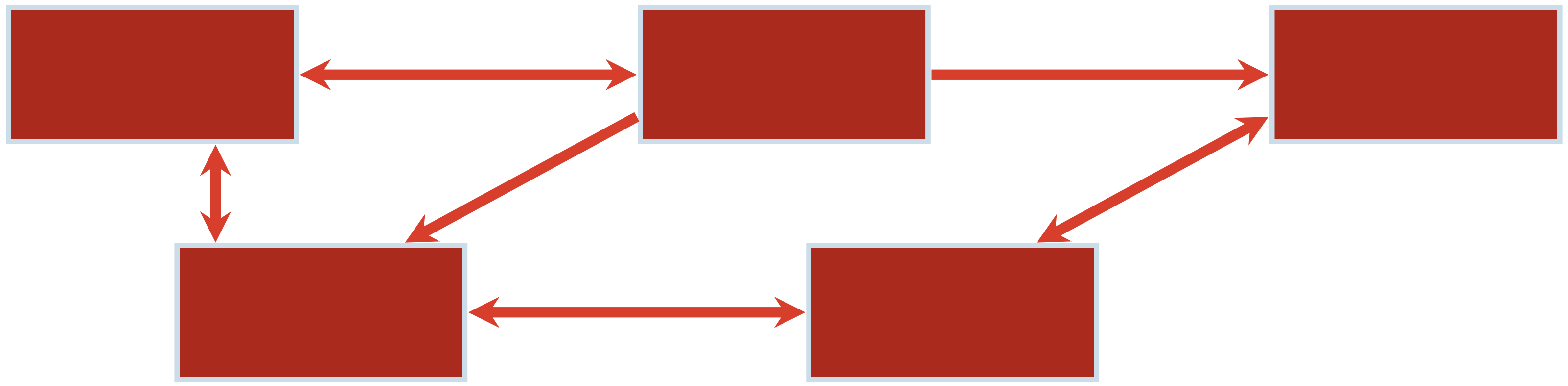
Appliance of system models

Practical system models should be:

- Describing the system in a correct way
- Formal – provide notations and techniques for unambiguous system specification
- Consistent – different views should not describe things being in conflict each other
- Easy to be explained to and understood by other people – as simple as possible but not oversimplified
- Easy for updates and maintenance
- In a form suitable for transfer to other people
- Balanced between visual and textual description

Object-oriented (OO) system modelling

- **OO system modelling** represents a system as a group of **objects** which co-operate, having a structure and constituting an organic, coherent set.
- A **system** is a collection of units connected and organized in order to accomplish a specific goal.



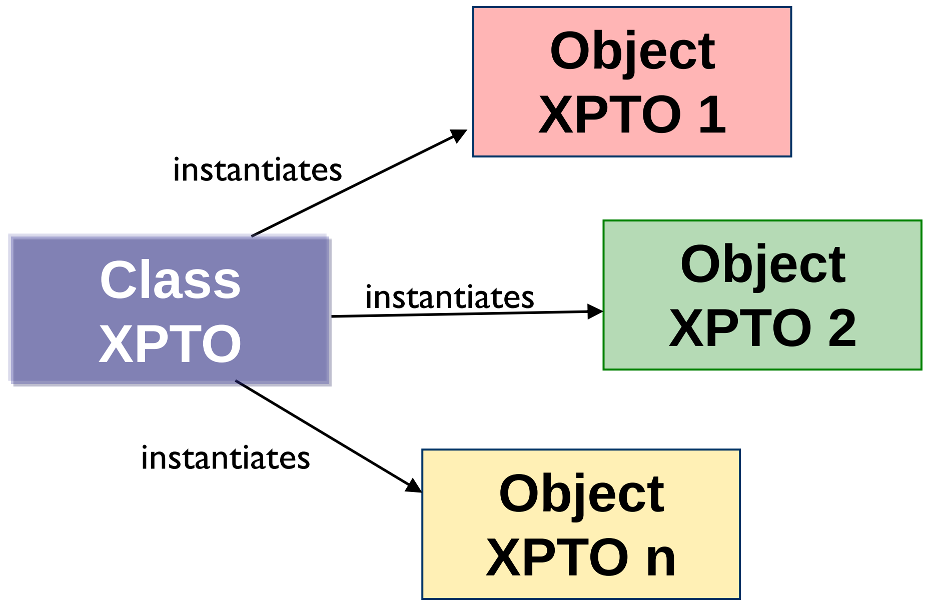
Objects

- An **object** is an atomic unit having:
  - an identity
  - a state (represented by the values of its properties) and
  - a behavior (operations with possible parameters and, in many cases, returning a result – realized as program methods to be executed)
- Class properties may be:
  - Data attributes of specific type (such as String, Integer, Float, …)
  - Structural relationships referencing other objects (like pointers to given objects)
- **Information hiding** – the internal representation, or state, of an object is hidden from the outside

Object and classes

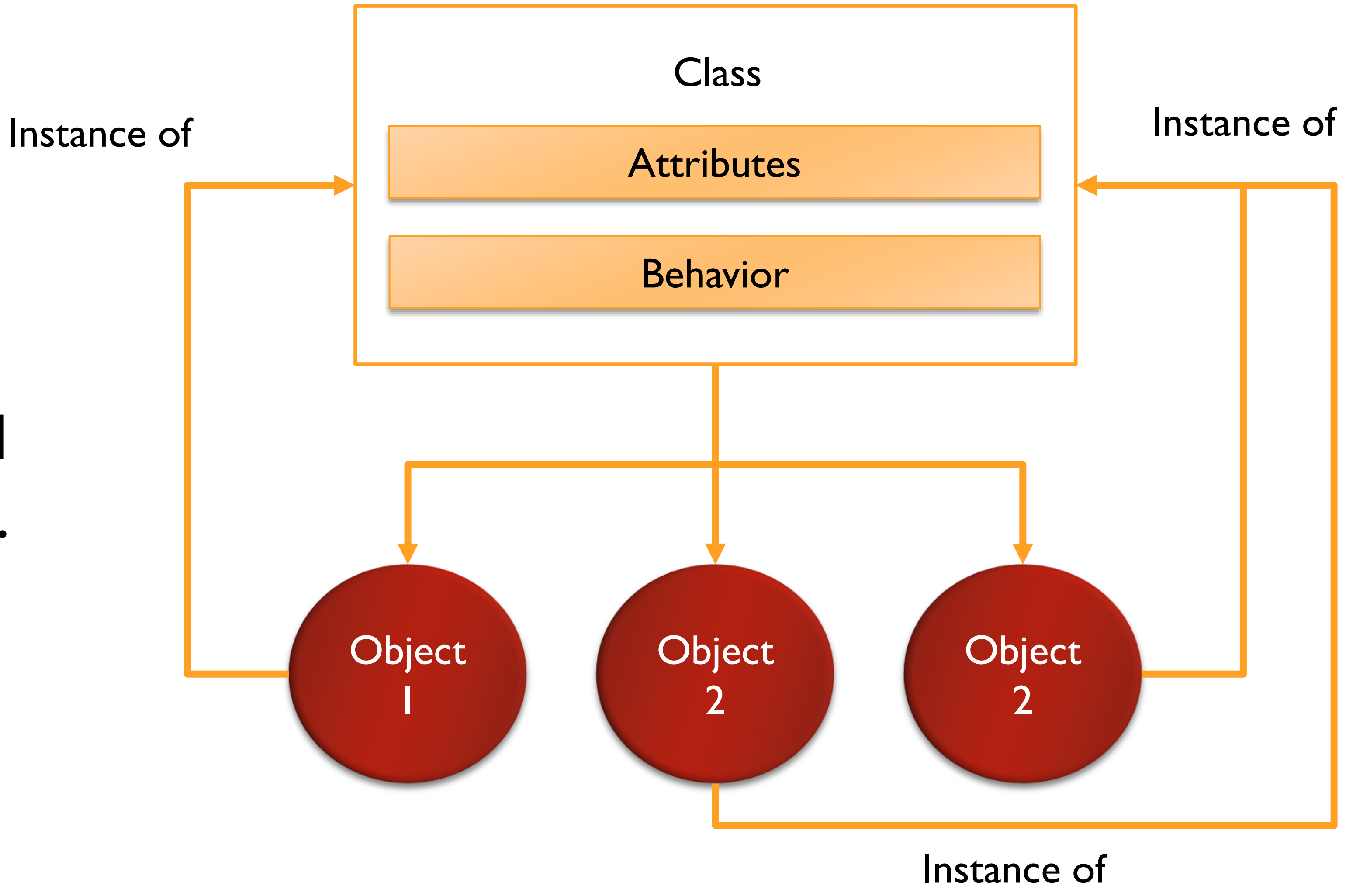
- A **class** is a template to create objects
- A **class** can be viewed as a container of objects with the same data and structural attributes, operations, and semantics
- Each object is an instance of a specific class
- The object cannot be instance (exemplar) of more that one class

- **Encapsulation** – bundling data and methods that work on that data within one unit



Object instance

An **object instance (i.e., exemplar)** is a specific **object** created from a particular **class**.



Object-oriented analysis and design

- OO analysis is a process of defining the problem in terms of objects:
  - real-world objects with which the system must interact, and
  - candidate software objects used to explore various solution alternatives.
- You can define all of your real-world objects in terms of their classes, attributes, and operations.
- OO design means defining the software solution of the problem by components, interfaces, objects, classes, attributes, and operations that will satisfy the requirements.
- You typically start with the candidate objects defined during analysis, and add or change objects as needed to refine a design solution.

Unified Modelling Language (UML)

UML is an OO modelling language for:

- specifying,
- visualizing,
- constructing, and
- documenting

the artifacts of software systems, as well as for business modeling and other non-software systems. As a modeling language UML includes:

- Model elements — fundamental modeling concepts and semantics
- Notation — visual rendering of model elements
- Guidelines — idioms of usage

Goals of UML 1/2

- Provide users with a ready-to-use, expressive visual modeling language to develop and exchange meaningful models.
- Furnish extensibility and specialization mechanisms to extend the core concepts:
  - build models using core concepts without using extension mechanisms for most normal applications,
  - add new concepts and notations for issues not covered by the core,
  - choose among variant interpretations of existing concepts, when there is no clear consensus,
  - specialize the concepts, notations, and constraints for particular application domains.

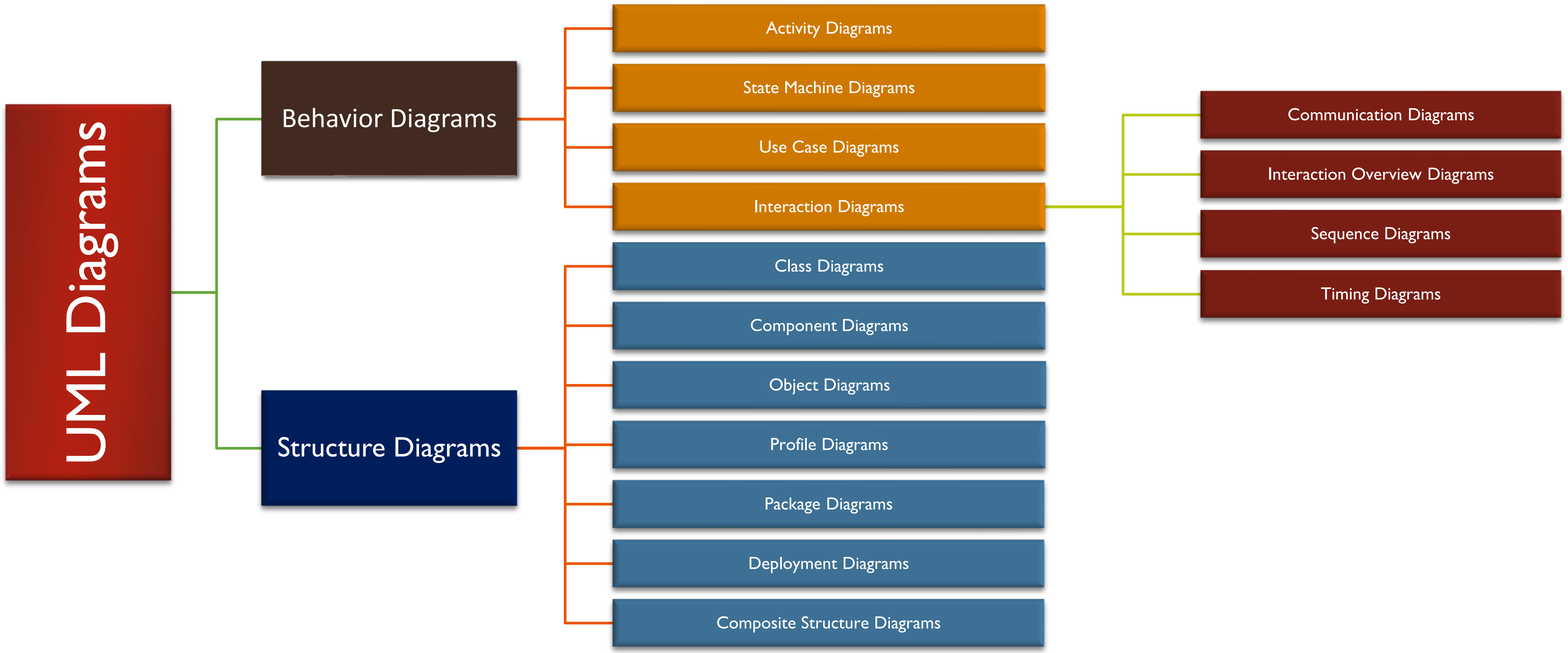
Goals of UML 2/2

- Support specifications that are independent of particular programming languages and development processes.
- Provide a formal basis for understanding the *process-independent* modeling language.
- Encourage the growth of the object tools market.
- Support higher-level development concepts such as components, collaborations, frameworks and patterns.
- Integrate best practices – UML fuses the concepts of Booch, OMT, and OOSE, in a single, common, and widely usable modeling language.

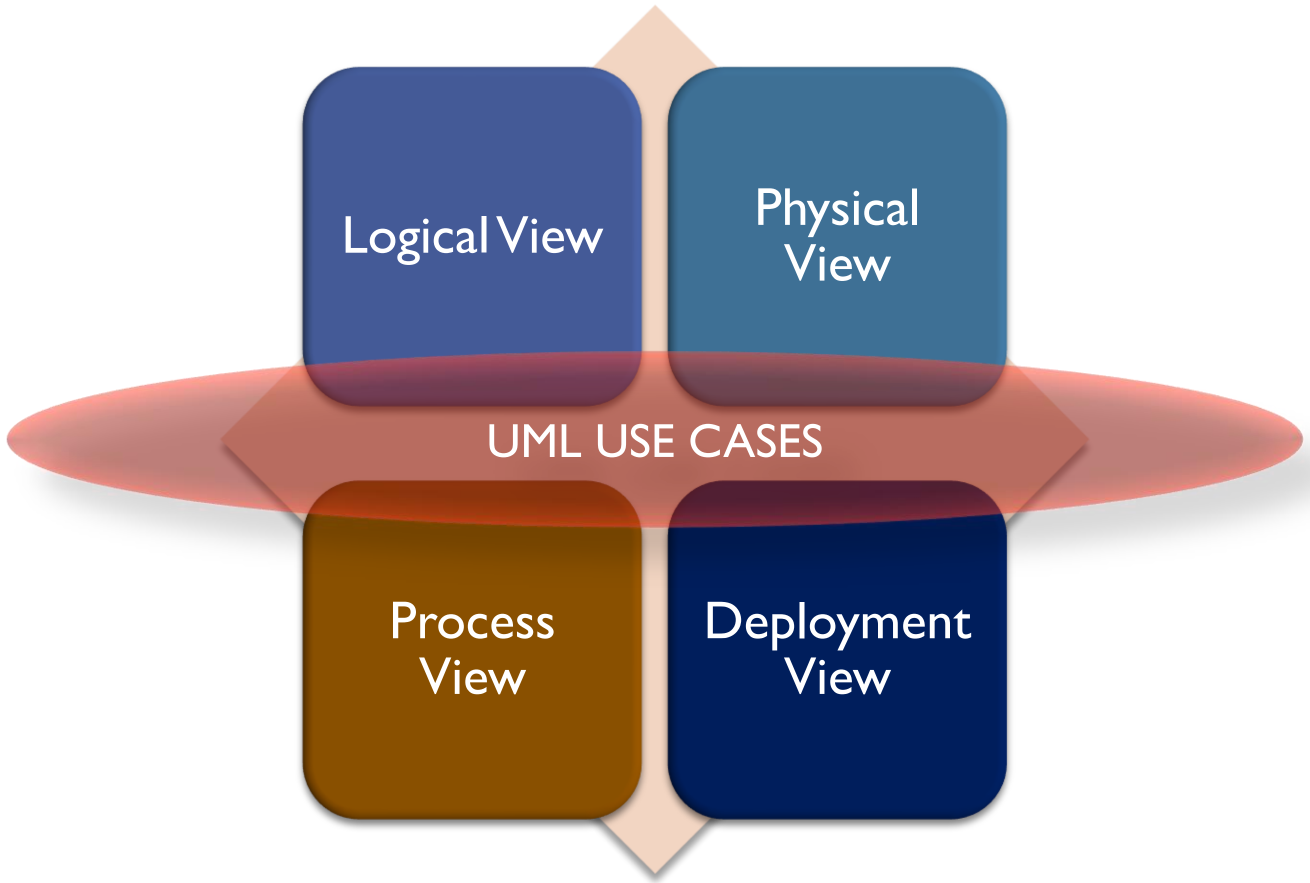
UML diagrams

- UML is the modern, general-purpose approach to modeling and documenting software systems and business processes
- UML facilitates it by diagrams of various types
- UML diagrams represent the OO analysis and design solutions
- You can draw UML diagrams by hand or by using CASE (Computer Aided Software Engineering) tools
- Using CASE tools requires some expertise, training, and commitment by the project management

Types of UML diagrams



The 4+1 views of the software architecture



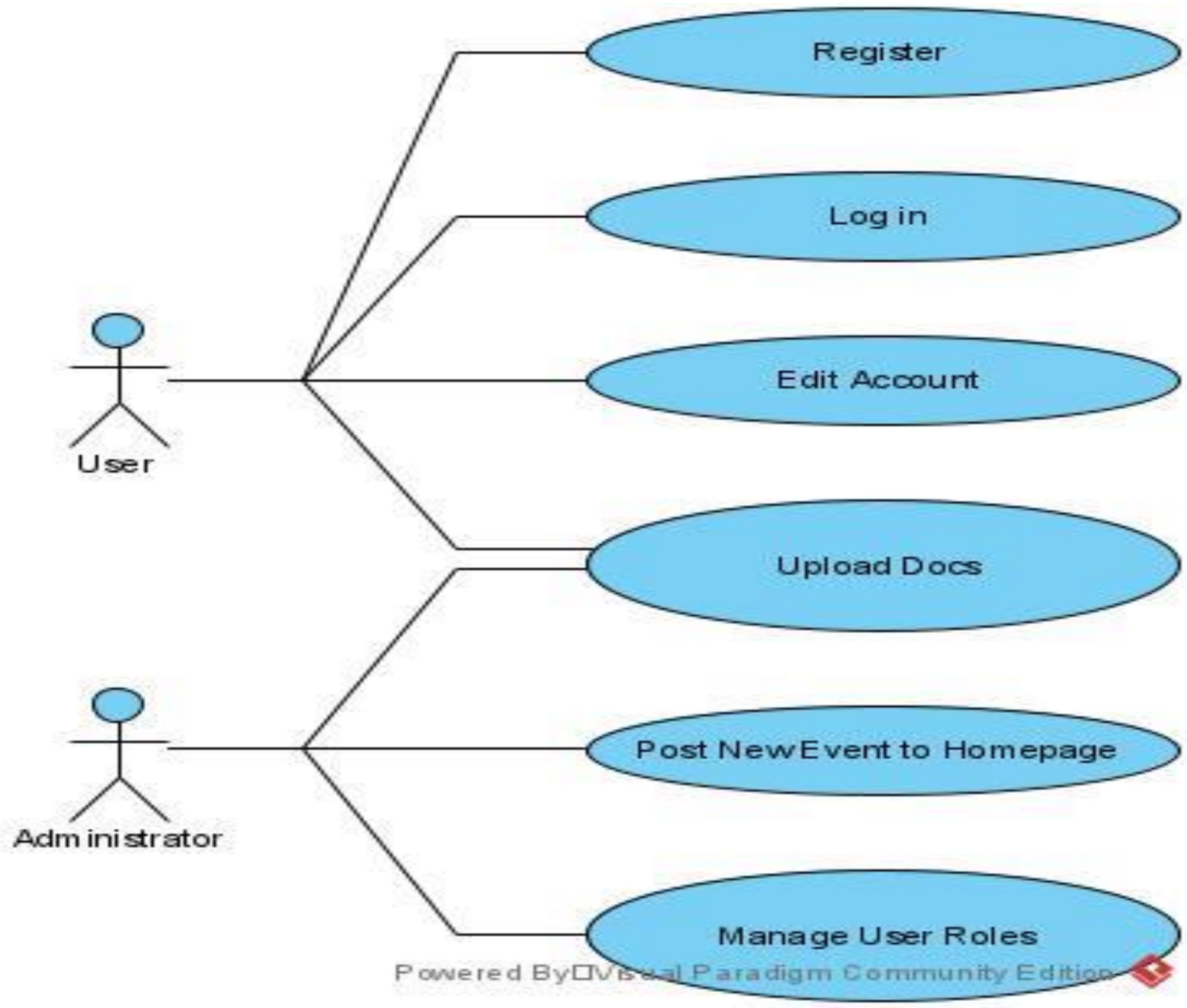
The 4+1 views of the software architecture

Scenarios
(UML Use Cases)

Logical View (Object-oriented Decomposition)	Development View (Subsystem Decomposition)	Physical View (Mapping the Software to Hardware)	Process View (Process Decomposition)
So-called conceptual view - describes the object model of the design	Describes the static organization or structure of the code in the development environment	Describes the deployment of the software on the hardware	Describes the aspects of competitiveness and synchronization

UML use case diagrams

- UML use case diagrams provide overview of usage requirements for a system.
- For actual system or software engineering use case diagrams describe actual system/software requirements
- Useful also for simple presentations to management and/or project stakeholders



- A use case diagram shows user's interaction with the system
- The use case diagrams are used mainly in specification and analysis of requirements

Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com/ last accessed 10.03.2023

Elements of use case

Actors - a person, organization, or external system that plays a role in one or more interactions with your system

Use cases - describe a sequence of actions that provide something of measurable value to an actor

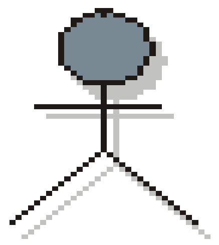
Associations - exist whenever an actor is involved with an interaction described by a use case

Other relations – *include*, *extend*, *generalize* and *depend*

System boundary boxes (optional) - rectangles around the use cases to indicates the scope of your system

Packages (optional) - UML constructs that enable you to organize model elements (such as use cases) into groups

Defining Actors

 **Actor**	An **actor** ***instance*** is *someone* or *something* outside the system that interacts with the system. An **actor** ***class*** defines a set of actor instances, in which each actor instance plays the same role in relation to the system.

To fully understand the system's purpose you must know **who** the system is for, or who will use the system. Different user types are represented as actors.

An actor is **anything** that exchanges data with the system. An actor can be a user, external hardware, or another system

How to find actors

- Who will supply/use/remove information?
- Who will use this functionality?
- Who is interested in any requirement?
- Where in the organization is the system used?
- Who will support/maintain the system?
- What are the system’s external resources?
- What other systems will need to interact with this one?

Defining use cases

A **use case** ***instance*** **(scenario)** is a sequence of actions a system performs that yields an observable result of value for one or more particular actors or other stakeholders of the system.

A **use case (*class*)** defines a set of use-case instances.

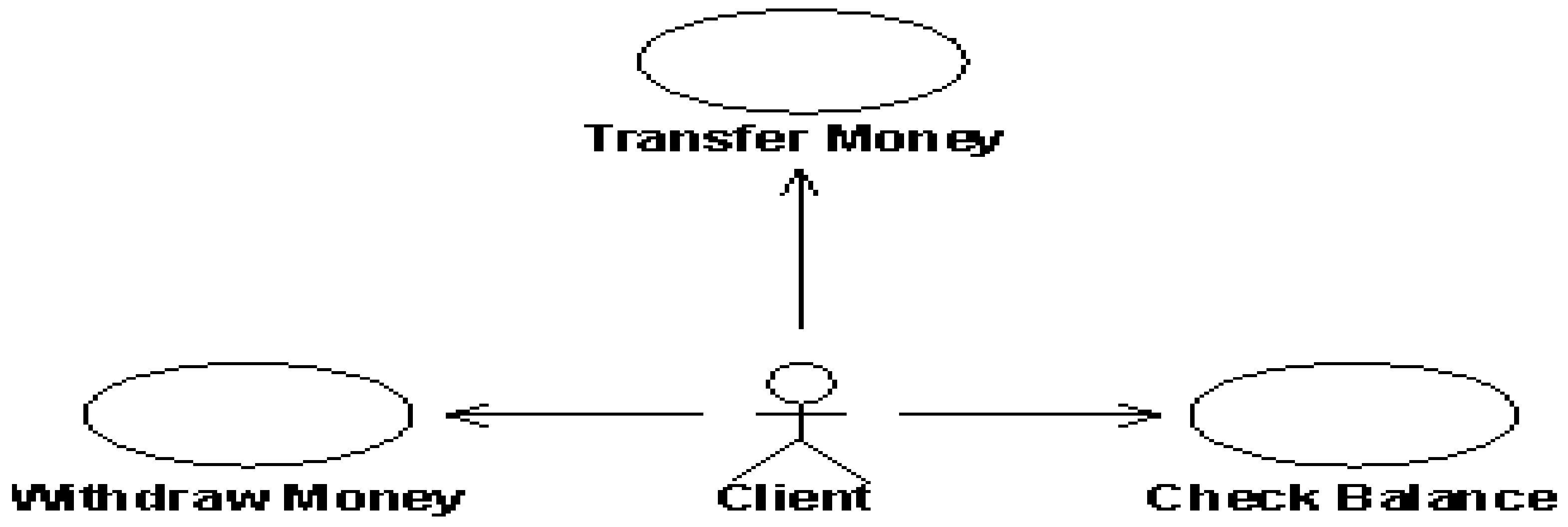
How to find use cases

- What are the system tasks for each actor you have identified?
- Does the actor need to be informed about certain occurrences in the system?
- What information must be modified or created in the system?
- Does the system supply the business with the correct behavior?
- What use cases will support and maintain the system?

Associations (relationships) in use case diagrams

- Associations between actors and/or use cases are indicated in use case diagrams by solid lines.
- An association exists whenever an actor is involved with an interaction described by a use case.
- Associations are modeled as lines connecting use cases and actors to one another
- The arrowhead is often used *to indicate the direction of the initial invocation of the relationship* (but not the direction of information exchange)

A sample use case diagram



An ATM example - the system functionality is defined by different use cases, each of which represents a specific flow of events, defines what happens in the system when the use case is performed, and has a task of its own to perform.

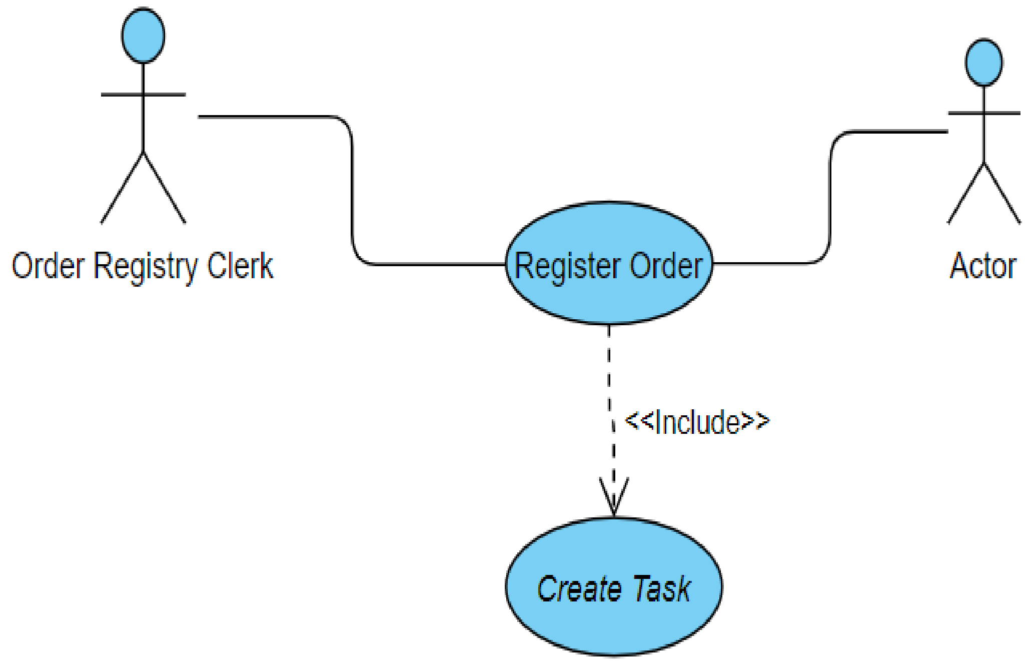
Use case documenting – flow of events

The **Flow of Events** of a use case contains the most important information derived from use-case modeling work. Its contents

- Describe how the use case starts and ends
- Describe what data is exchanged between the actor and the use case
- Do not describe the details of the user interface, unless it is necessary to understand the behavior of the system
- Describe the flow of events, not only the functionality. To enforce this, start every action with "When the actor ... "
- Describe only the events that belong to the use case, and not what happens in other use cases or outside of the system
- Avoid vague terminology such as "for example", "etc. " and "information"
- Detail the flow of events - all "*whats*" should be answered.

Concrete and abstract use cases

- A **concrete** use case is initiated by an actor and constitutes a complete flow of events (instance of the use case performs the entire operation called for by the actor).
- An *abstract* use case (written in *italics*) is never instantiated in itself. Abstract use cases are included in, extended into, or generalizing other use cases. When a concrete use case is initiated, an instance of the use case is created. This instance also exhibits the behavior specified by its associated abstract use cases. Thus, no separate instances are created from abstract use cases.

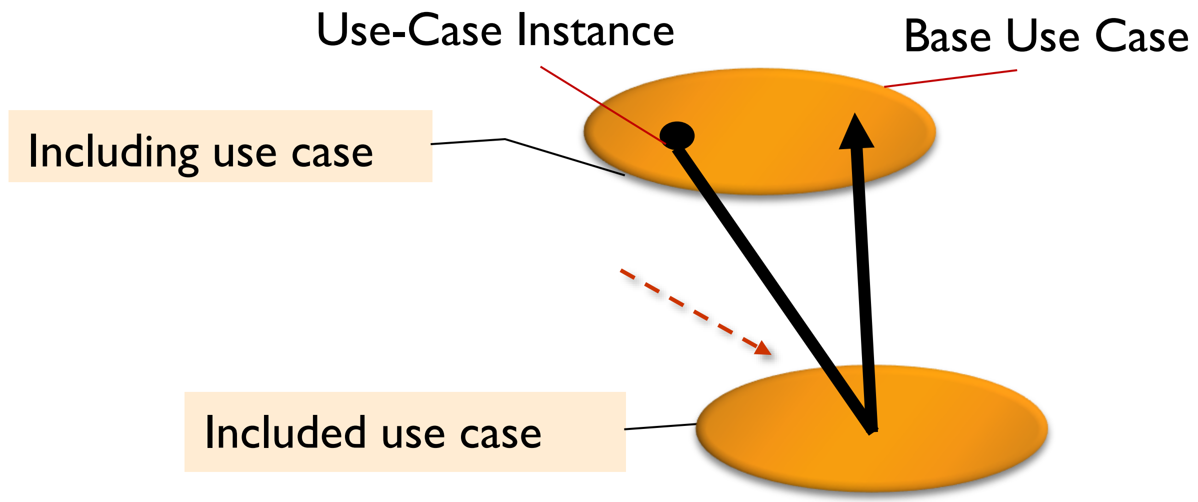


Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com/ last accessed 10.03.2023

<<Include>> relationship

An **include-relationship** is a directed relationship from a base use case to an inclusion use case, specifying how the behavior defined for the inclusion use case is non-optionally, explicitly inserted into the behavior defined for the base use case.

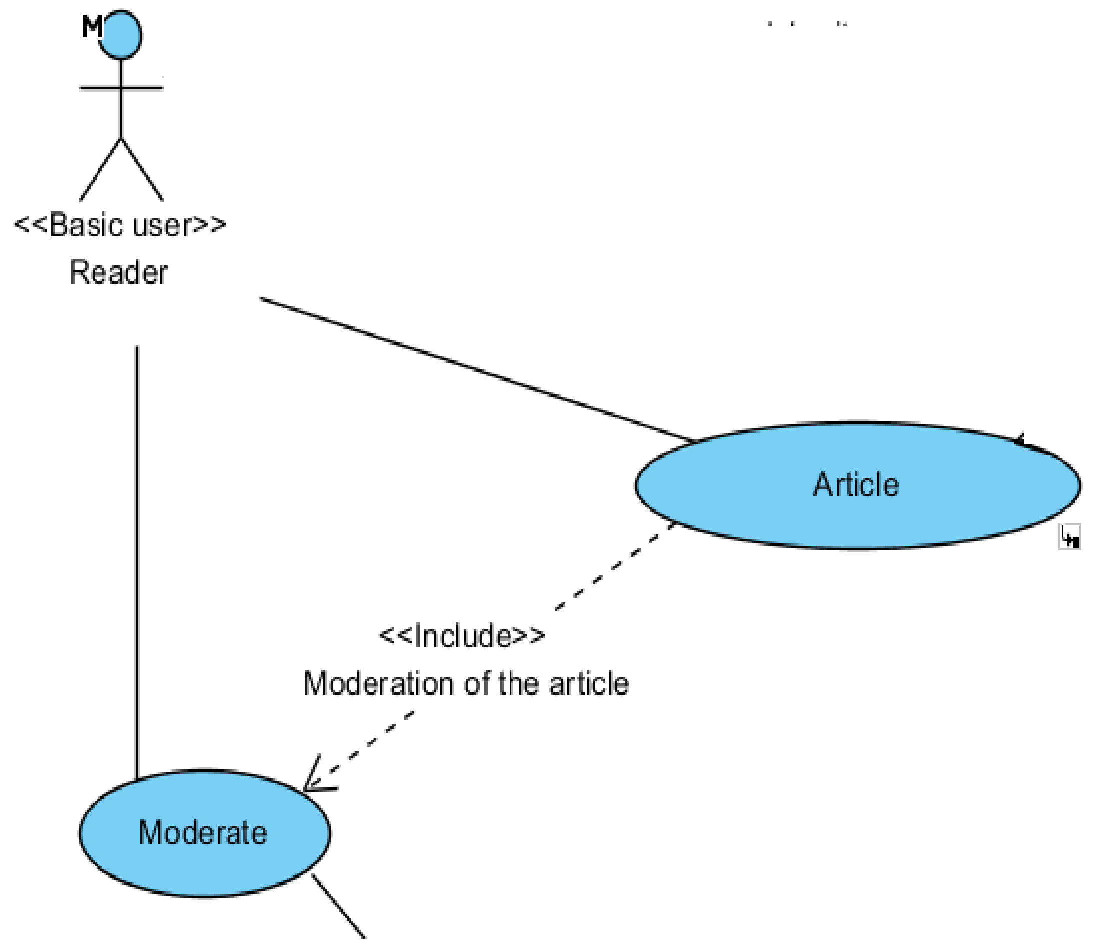
«include»



Executing a use-case instance following the description of a base use case including its inclusion.

More about <<Include>> relationship

- Including use case includes the “addition” and owns the include relationship.
- Addition is use case that is to be included.
- The including use case may only depend on the result (value) of the included use case.
- This value is obtained as a result of the execution of the included use case.

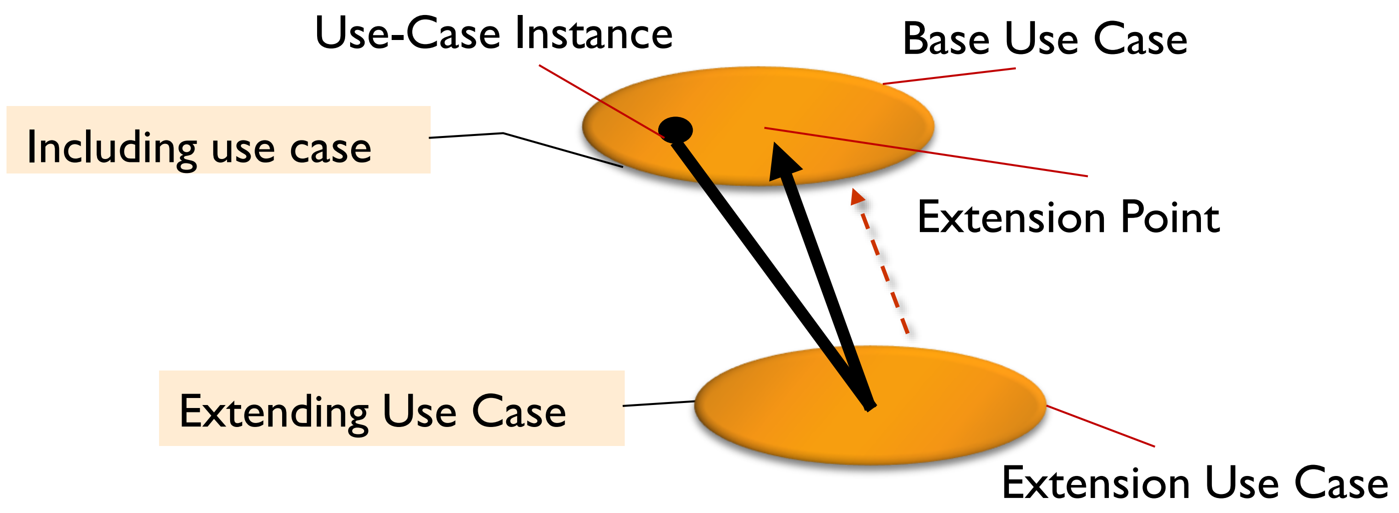


Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com/ last accessed 10.03.2023

<<Extend>> relationship

An **extend-relationship** goes from an extension use case to a base use case, specifying how the behavior defined for the extension use case can be inserted into the behavior of the base use case. It is implicitly inserted in the sense that the extension is not shown in the base use case.

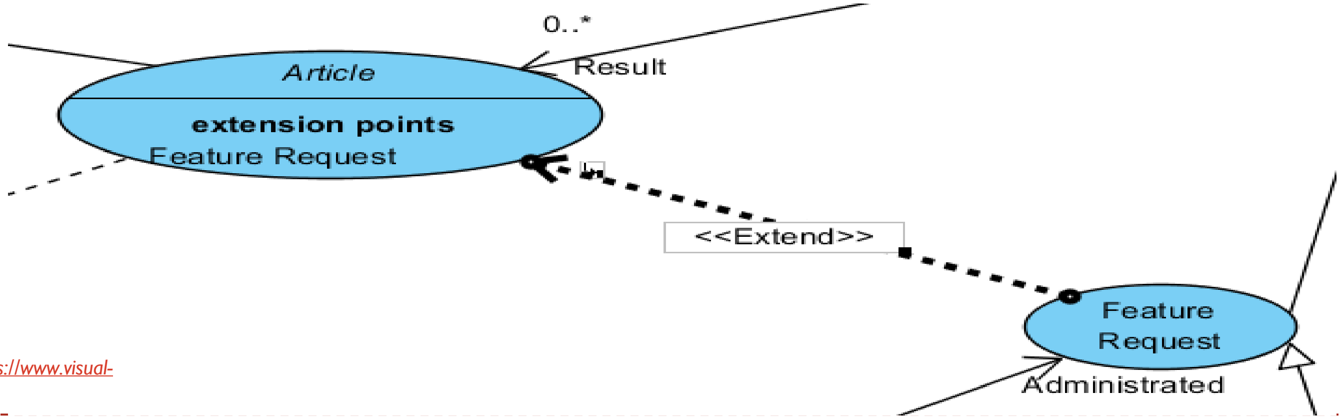
«extend»



Execution of a use-case instance follows a base use case and its extension.

More about <<Extend>> relationship

- This relationship specifies that the behavior of a use case may be extended by the behavior of another (supplementary) use case.
- The extended use case is defined independently of the extending use case and is meaningful independently of the extending use case.
- On the other hand, the extending use case typically defines behavior that may not necessarily be meaningful by itself. Instead, the extending use case defines a set of modular behavior increments that augment an execution of the extended use case **under specific conditions**.



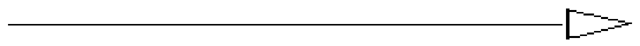
Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com/ last accessed 10.03.2023

Extension points

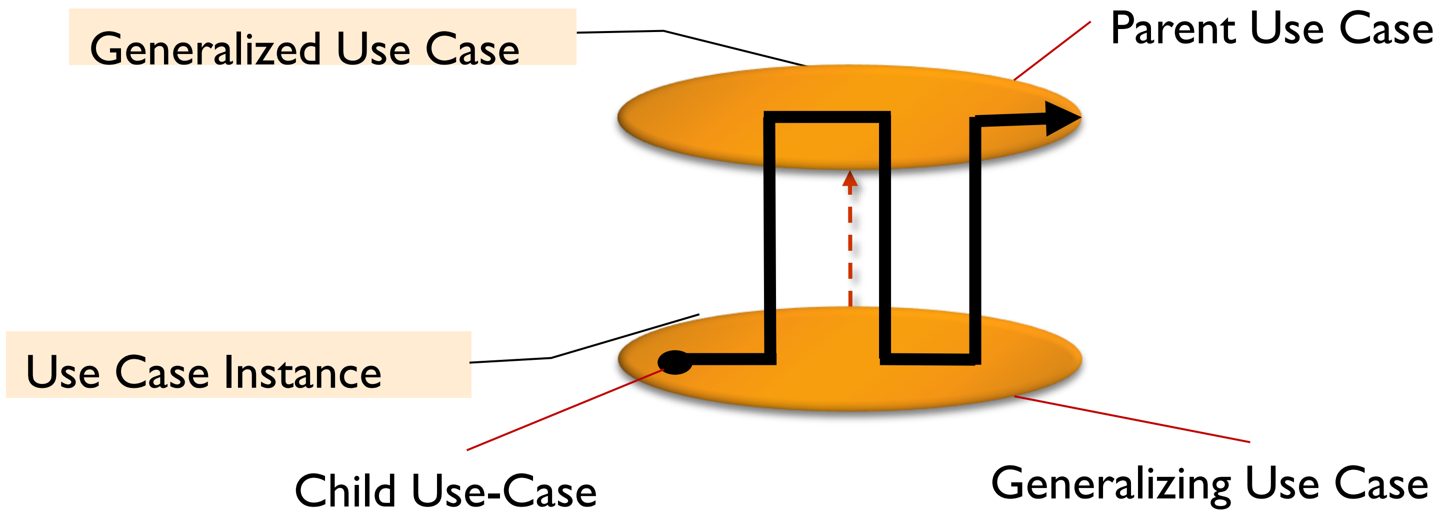
- Extension points (since UML 2.0) show the actual logic necessary for one use case to extend another.
- An extension point identifies the point in the base use case where the behavior of an extension use case can be inserted.
- The extension point is specified for a base use case and is referenced by an extend relationship between the base use case and the extension use case.

Use case generalization

A **use-case-generalization** is a taxonomic relationship from a child use case to a more general, parent use case, specifying how a child can specialize all behavior and characteristics described for the parent.



Use case generalization

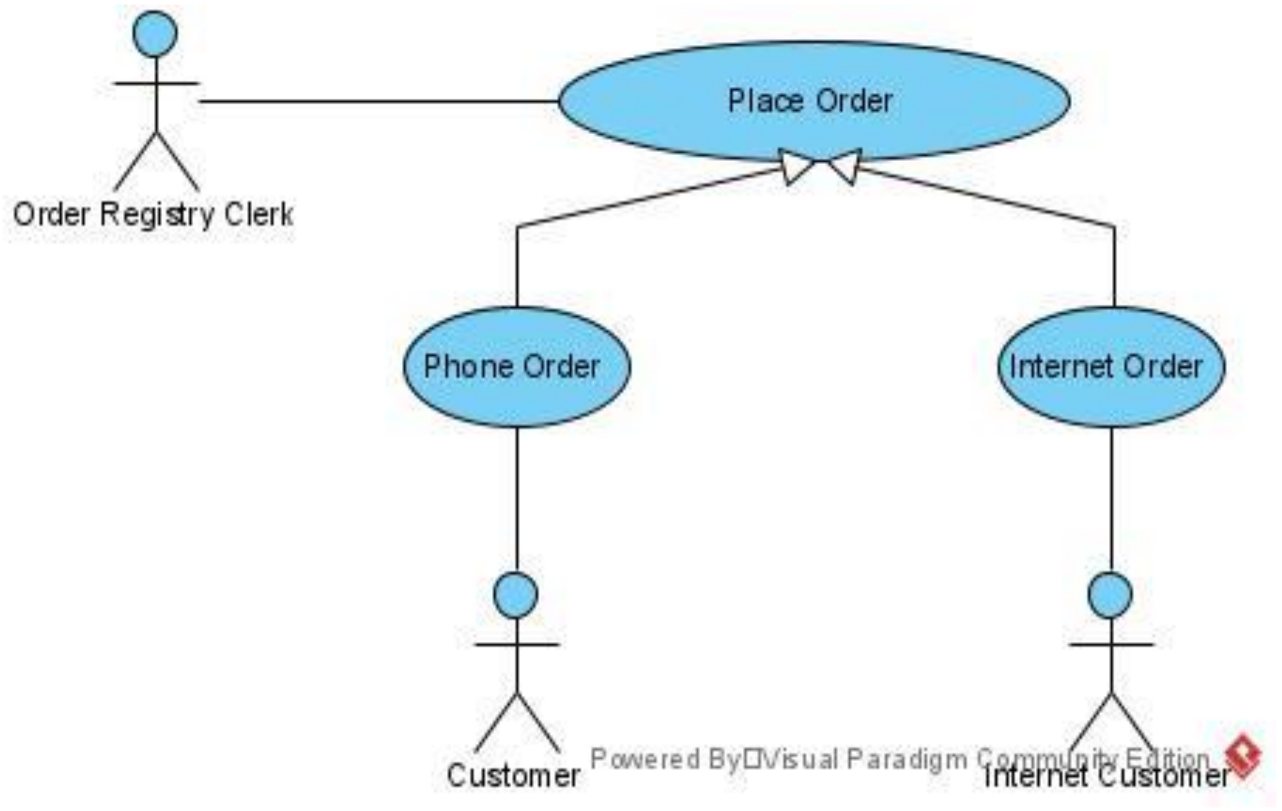


Execution: the use-case instance follows the parent use case, with behavior inserted or modified as described in the child use case

More about generalization

General Use Case	References the general classifier in the Generalization relationship.
Specific Use Case	References the specializing classifier in the Generalization relationship.
Substitutable	Indicates whether the specific classifier can be used wherever the general classifier can be used. If true, the execution traces of the specific classifier will be a superset of the execution traces of the general classifier.

Example of use case generalization

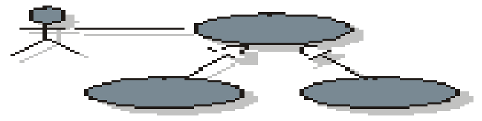


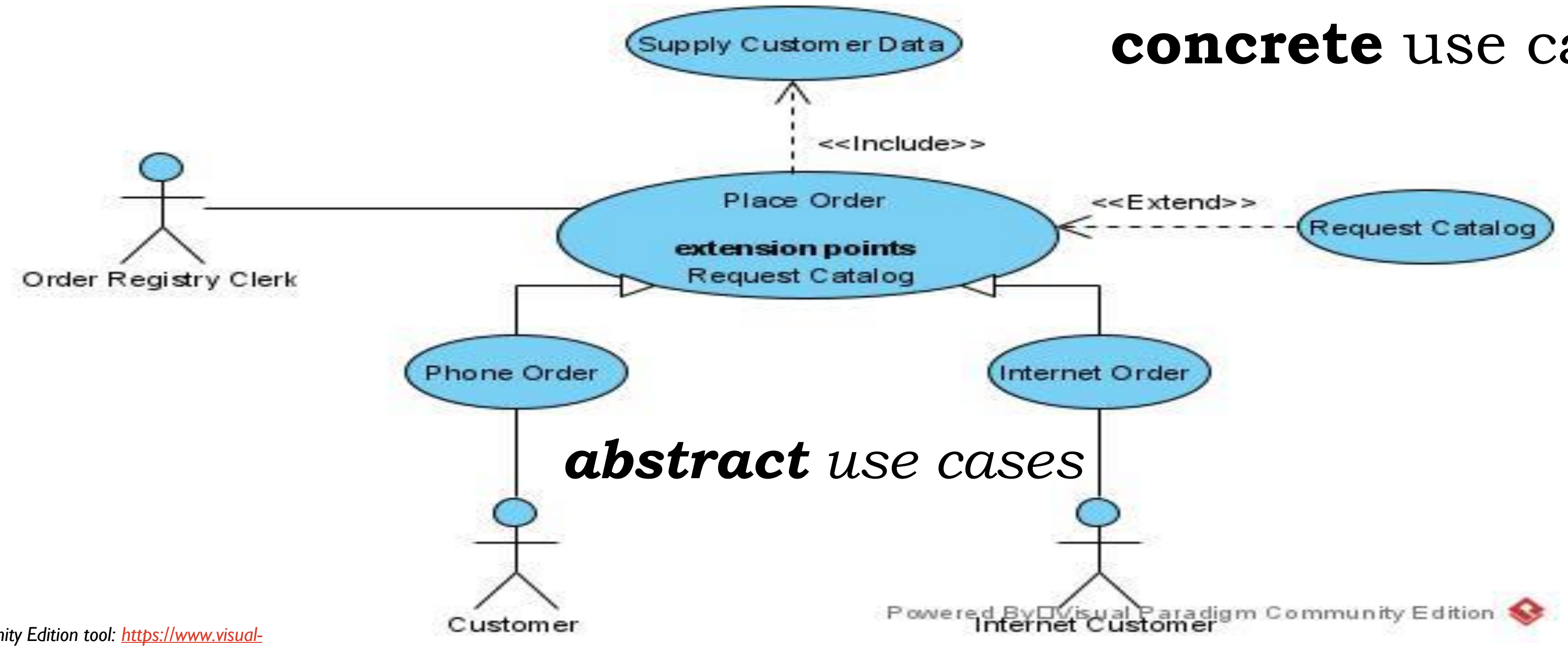
Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com last accessed 10.03.2023

The actor Order Registry Clerk can instantiate the general use case Place Order. Place Order can also be specialized by the use cases Phone Order or Internet Order.

The child may modify behavior segments inherited from the parent. The structure of the parent use case is preserved by the child. Both use-case-generalization and include can be used to reuse behavior among use cases.

Use case model of an Order Management System

 **Use case model**	The **use-case model** is a model that describes a system's requirements in terms of use cases.



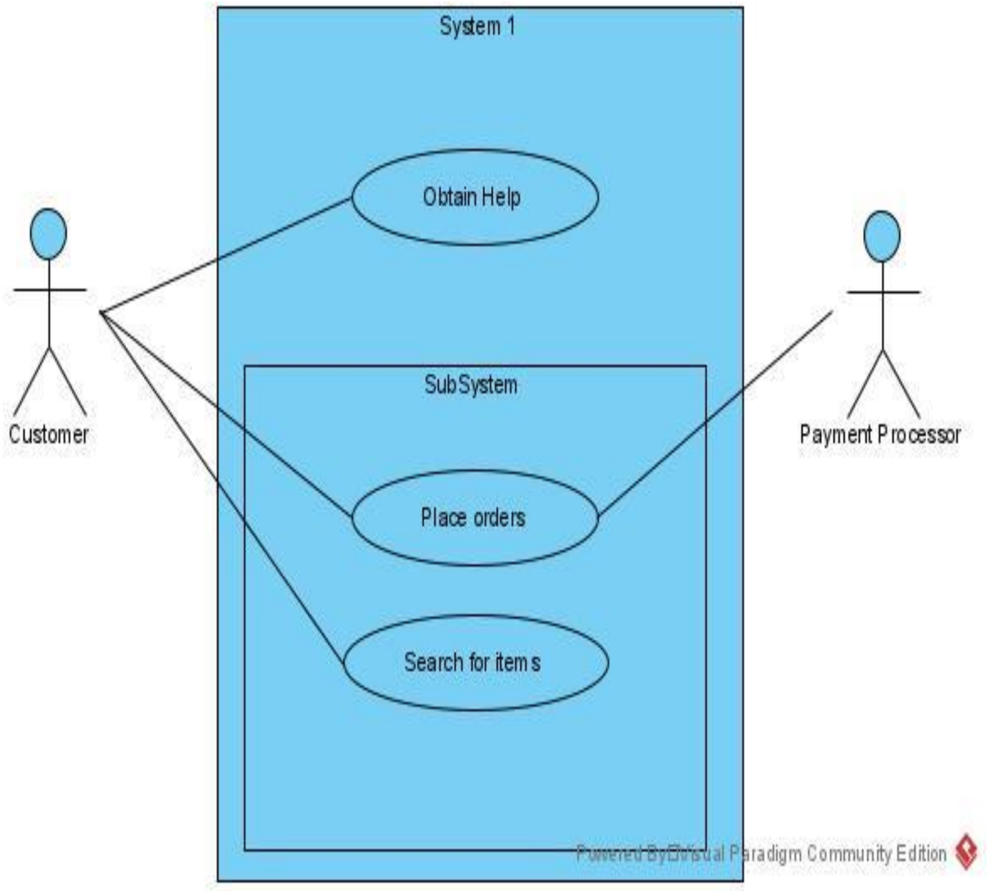
concrete use cases

abstract *use cases*

Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com/ last accessed 10.03.2023

System boundary boxes

- System boundary box (optional) - a rectangle around the use cases to indicates the scope of your sub-system
- Anything within the box represents functionality that is in scope and anything outside the box is not
- Rarely used – i.e., to identify which use cases will be delivered in each major releas of a system



Used Visual Paradigm Community Edition tool: https://www.visual-paradigm.com/ last accessed 10.03.2023

References

- *These slides are for educational purposes and used in the FMI Course "Software Technologies" and are part of the ICT-TEX Project 2022.*
- *Sommerville, I. Software Engineering. 10th edition, Published by Pearson Education, ISBN: 978-1-292-09613-1 (2016)*
- *Pressman, R., Maxim, B. Software Engineering: A Practitioner's Approach. 9th edition, Published by McGraw-Hill Education, ISBN: 9781260548006, (2019)*
- *Page-Jones, M., Constantine, L. Fundamentals of object-oriented design in UML, Addison-Wesley, ISBN: 0-201-69946-X (2000)*