INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Conception d'Architectures

Nicolas VAN WAMBEKE

nicolas@vanwambeke.net

Based on material by Ernesto Exposito

Outline

Introduction

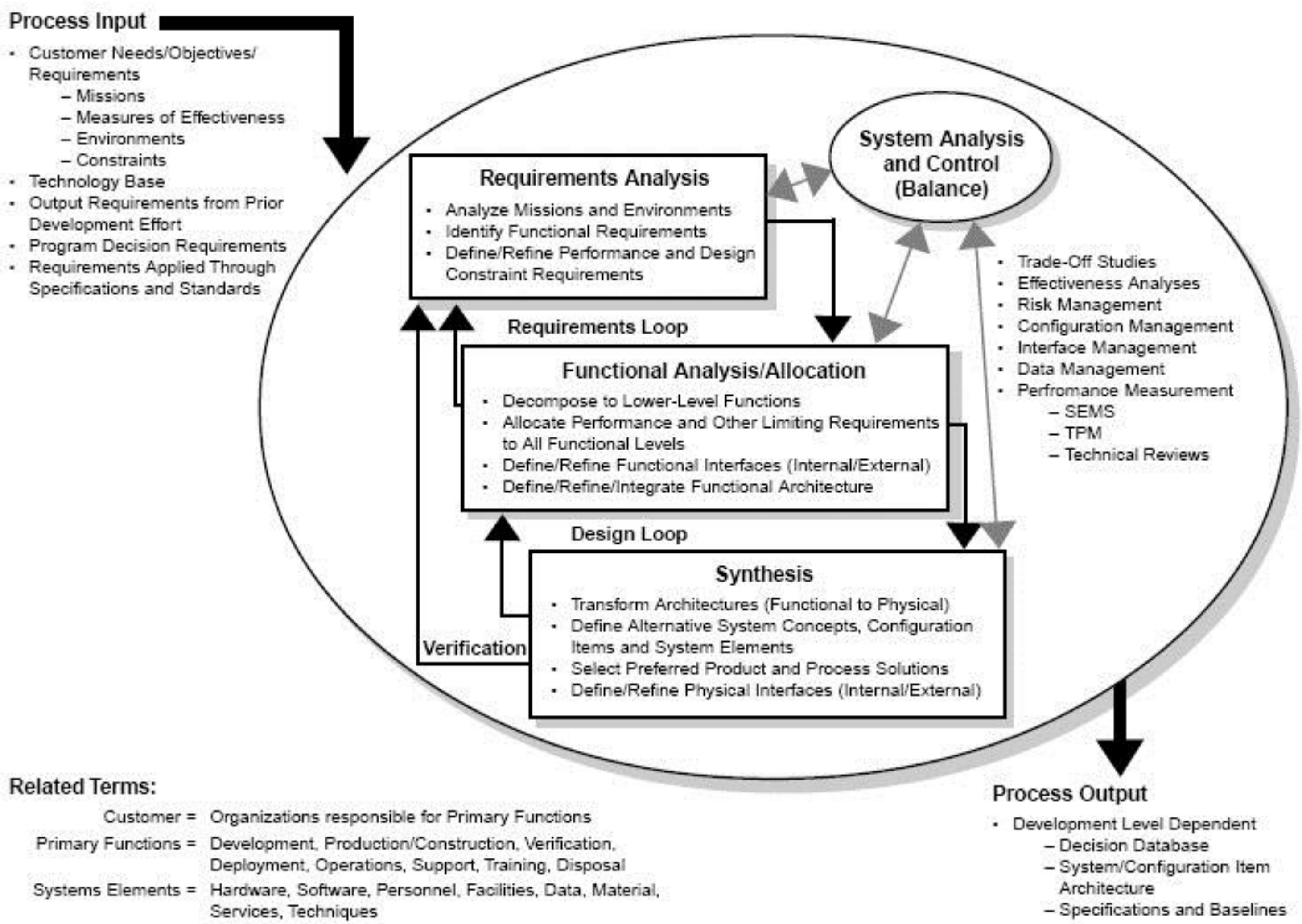
UML during Analysis and Design Phases

UML & Software Design Patterns

UML at the center of Software Engineering

Introduction

Introduction



From requirement to product

What the user needs

What the contractor understands

What the contractor specifies

What is delivered

What the subcontractor aims for

What the subcontractor understands

Introduction

- UML is not a methodology
- UML is not a way to achieve good design
- UML does not make your design better or worse

- UML can help you understand the software
- UML can help you explaining your ideas to others
- UML can help you avoid mistakes

UML is a language to ease exchanges between actors of the software engineering process

Software Development Phases

- 1/ Requirements
- 2/ Analysis
- 3/ Design
- Implementation
- Testing
- Deployment
- Maintenance

UML can help here

1/ Requirements

Requirements define what is required from a system to be built

Answering the questions:

- what is it? → Non-functional requirement
- what does it do? → Functional requirement

Ideally without any ambiguity

Requirements evolve during the lifetime of a system

2/ Analysis

- Understanding the system context (OO Approach)
  - relevant entities
  - their properties and their inter-relationships.
- Implies verification with:
  - Customers and end users
  - Other actors

3/ Design

- How to solve the problem
- **System design**
  - Structural: breaks the system down into :
    - logical subsystems and components
    - physical subsystems (computers and networks)
  - Behavioral:
    - collaboration of components to provide the services

Implementation / Testing / Deployment / Maintenance

- **Implementation**
  - Coding the system specification
- **Testing**
  - Against the system requirements to see if it fits the original goals
- **Deployment**
  - Delivery of hardware and software to the end users, along with manuals and training materials
- **Maintenance**
  - Correcting bugs and extensions…

UML during analysis and design phases

History of UML and methodologies

1960s-1970s:

– emergence of OO programming languages, such as Simula and Smalltalk

– OOA/D was informal during this period

1980s:

– OOD and OOA emerged as a topic.

– First works of UML founders: G. Booch, I. Jacobson, J. Rumbaugh and others..

1990s:

– Unified Modeling Language started as an effort by Booch and Rumbaugh (1994) not only to create a common notation, but to combine their two methods (the Booch and OMT methods) .

– They were joined at Rational Corporation by Ivar Jacobson (Objectory method) -> the became the group of **three amigos**

– They decided to reduce the scope of their effort: focus on a common diagramming notation (UML) rather than a common method

History of UML and methodologies

1997:

– UML 1.0 results for a task force at the OMG (Object Management Group, an industry standards body for OO-related standards). (www.omg.org and www.uml.org)

2003:

– UML 2.0 standard officially adopted by the Object Management Group (OMG)

And the methodologies?

– The Unified Process or UP has emerged as a popular iterative software development process for building object-oriented systems.

– In particular, the Rational Unified Process or RUP, a detailed refinement of the Unified Process, has been widely adopted.

– Others: extreme programming (XP) and agile methodologies able to be responsive to change

UML introduction

- Non-proprietary specification language for object modeling
- Proposes a graphical notation intended to create abstract models of systems
- Industry standard mechanisms for visualizing, specifying, constructing and documenting software systems.
  - Documentation (ITU/ETSI/etc.)
- Intended for:
  - Modelers: to understand the problem
  - Designers: to explore possible solutions
  - Developers: to construct solutions

UML introduction

- Basis of model-driven technologies
  - Model Driven Development (MDD) and Model Driven Engineering (MDE) (implementation)
  - Model-Driven Architecture (MDA) (deployment)
  - modeler-> designer-> developer
  - Platform Independent Model (PIM) -> Platform Specific Model (PSM)
- UML tools
  - Specification (model checking) and visualization
  - Simulation and testing (model verifying)
- UML 2: 13 diagrams (behavior and structure)

Behavior

* Use case:
  – services that actors can request from a system
* State machine (or protocol state machine):
  – life cycle of an object

Activity:
  – modeling of concurrent control and data flow (Petri-nets like semantic)

Sub-group: Interaction Diagrams

– * Sequence diagram:
  • ordered exchange of messages between a group of objects.
– Interaction overview:
  • show many different interaction scenarios for the same collaboration (high-level view of interactions)
– Collaboration/Communication:
  • Similar to a sequence diagram (old collaboration diagrams)
– Timing:
  • States of an UML element (such as state machine) in function of time

* Included in this tutorial

Structure

* Class:
  - real word entities and their relationships
* Composite structure:
  - to show how a class is made (component-based design)
  - how parts of a (container) class are connected to each other to form an internal structure of the container
* Component:
  - Show the structure of the system as black boxes with their interfaces (replacement or reuse)

Deployment:
  - Run-time architecture of the system, hardware platforms, software artifacts, software environments (OS/VM)

Object:
  - illustrative examples of objects and their links

Package:
  - to organize model elements and dependencies

* Included in this tutorial

Uses cases

Captures the core features and interactions of a new system or software change in its execution context

Each **use case** provides one or more **scenarios** that convey *how the system should interact with the end user or another system to achieve a specific business goal*

Based on a “good” **System Requirements Specifications** (SRS).

From the SRS analysis following elements should be easily identified:

- **Subjects**
- **Verbs**
- **Details/Comments**

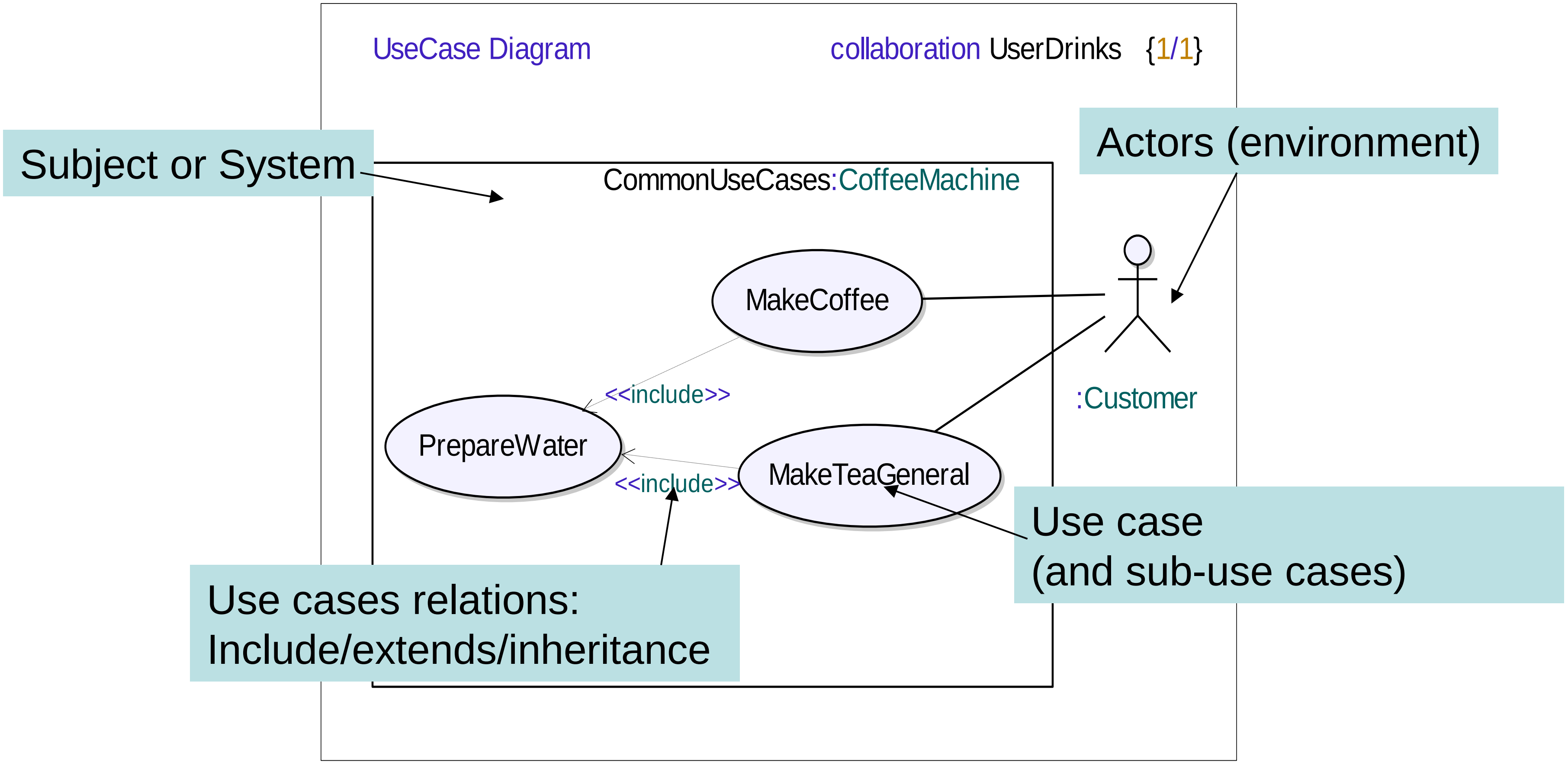
Uses cases

Identifying the **entities**
- Identified “**subjects**” from SRS
- Some of the identified subjects will be **actors** and some others will be **internal components** of the system

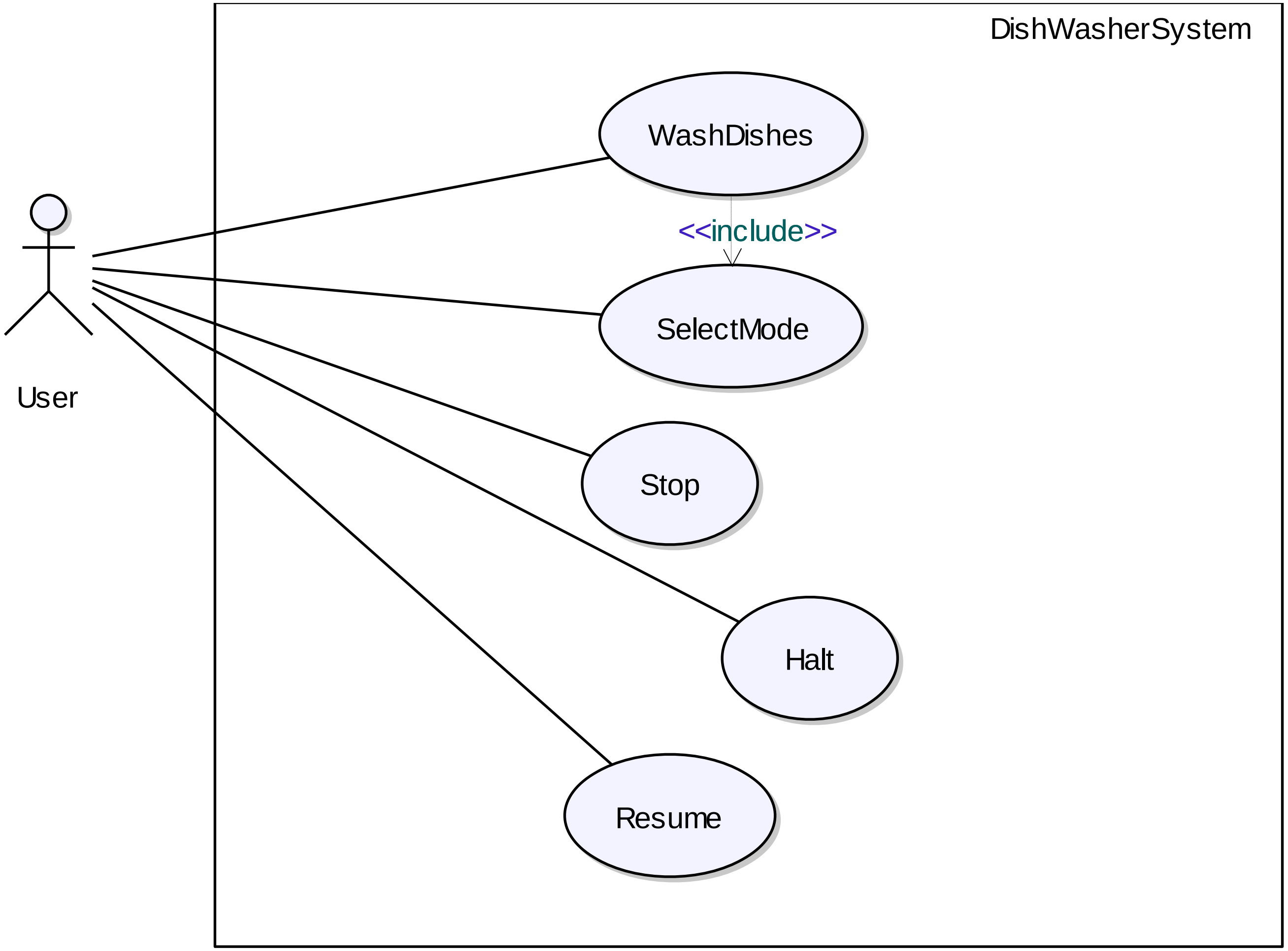
Identifying the **use cases**
- Identified “**verbs**” from the SRS
- Some of the identified verbs will be a **use case** while some others will be used as **restrictions** or **details** within the use cases, or some other elements **further** in the modeling process.

Identifying the **use case details**
- Every detail in the SRS is important and should be exploited.
- These details “details” will guide the discovery/design of
  - classes
  - Attributes and operations
  - Messages

Example: Coffee machine
Use case diagram



Example: Dishwasher
Use Case diagram

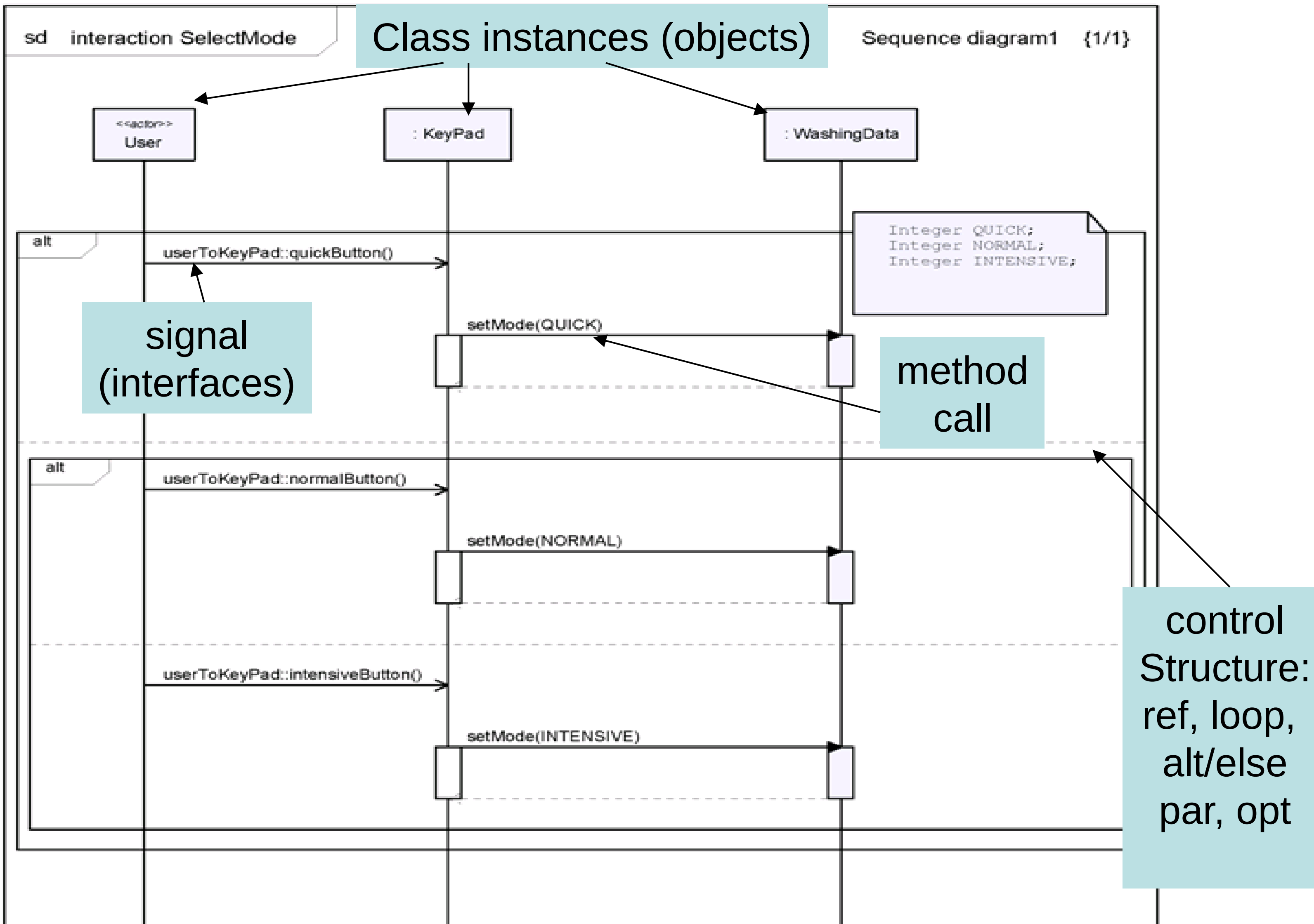


Sequence diagrams (SD)

Captures the exchanges between the components required in order to implement a specific use case

- A sequence diagram describes the order within a **time scale** of the **messages exchanged** between the **system components** in order to **implement** a specific **service** (i.e. **use case**).
- Instances of **classes** are modeled as a **lifeline** (vertical line) representing their role during the interaction. Classes can be active or passives
- An **active** class contains autonomous behavior, while a **passive** class contains attributes and methods to be used by active objects.
- The **messages** are represented by horizontal arrows between the lifelines.
- Messages can be **signal** instances (**asynchronous** messages) or **method calls** (**synchronous** messages).
- Messages can also be particular methods calls such as **constructors** (object creation) or **destructors** (object destruction)

Sequence diagrams (SD)



Class diagrams

Captures the different components of a software/system as well as the relationships that exist between them

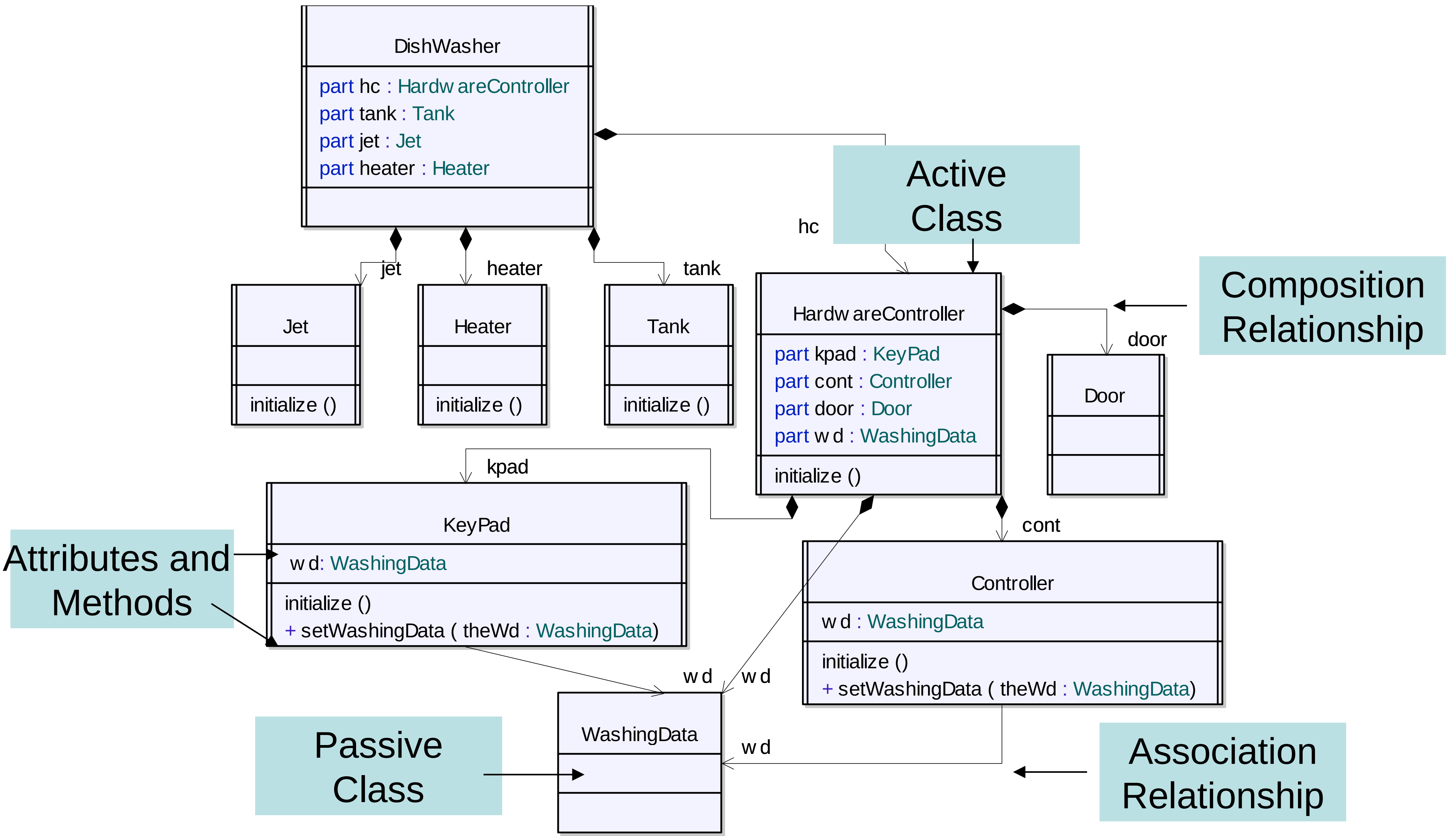
- Describes the **static view** of the **classes** and their **relationships**.
- For every class, the **attributes** (i.e. class's properties) and **methods** (i.e. class's operations) can be described.
- Each attribute is shown with at least its **name**, and optionally with its **type**.
- Each method is shown with at least its **name**, and optionally also with its **parameters** and **return type**.

Class diagrams

- Attributes and methods may have their visibility marked as follows:
  - "+" for **public** (accessible from any external object)
  - "#" for **protected** (accessible from internal components and inherited classes)
  - **"−"** for **private** (accessible from internal components)
  - Unspecified: **package** level (accessible within the package)

- **Expression of association**, **aggregation**, **composition**, **generalization/specialization** relationships

Class diagram



Class diagrams open a window for further details

Once classes have been identified, one can:

- describe their external interface:
  - how do they communicate?
  - **Component diagrams**
- their internal structure
  - how are they composed and connected?
  - **Composite diagrams**
- and the internal behavior
  - what do they do?
  - **Statechart diagrams**

Component diagrams

- Intended to describe how **active classes** are going to communicate with each other using
  - **required** (OUT) interfaces
  - **implemented**/provided (IN) **interfaces**
  - By the way of interaction points called **ports**.

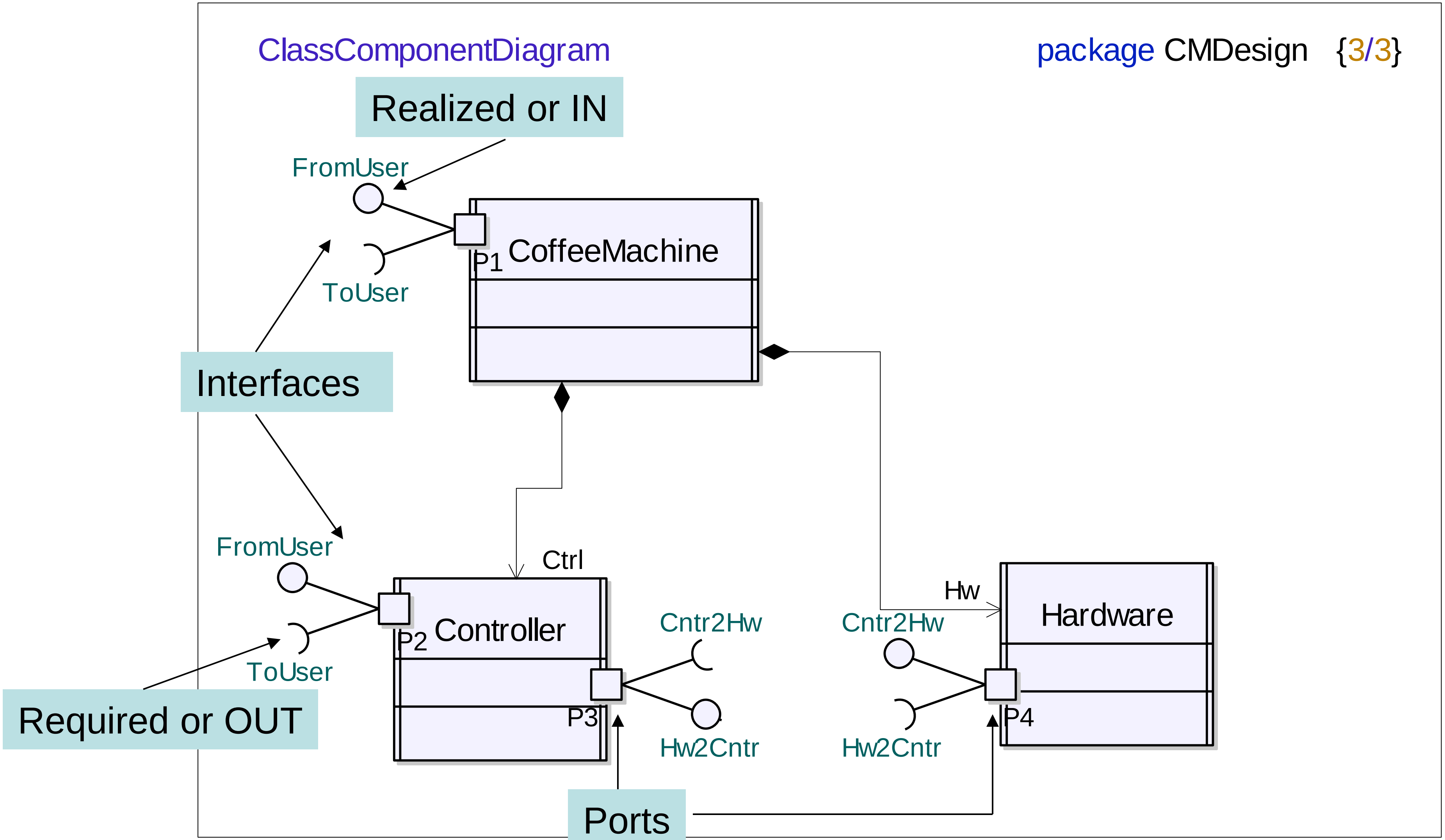
- This diagram describes the different communication points (**ports**) and which **signals** are sent and received on these ports. The signals are commonly grouped in **interfaces**.

Component diagrams

- When a component is able to process signals received from another component that means the component "**implements**" or **"provides**" the interface.

- If a component is a producer of signals, it means that the component "**requires**" the interface (it requires the interface to be implemented by another component in order to communicate with it).

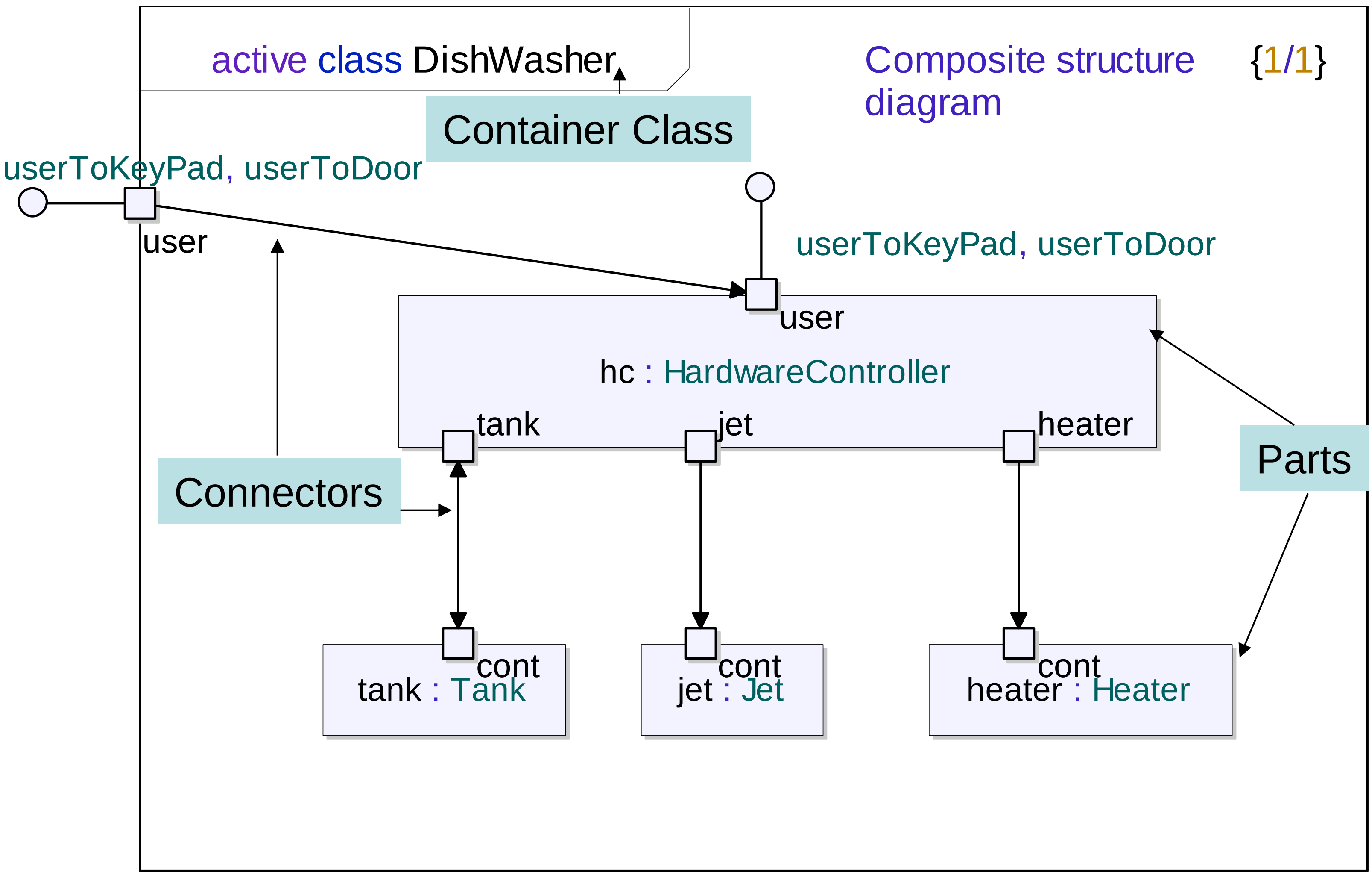
Component diagrams



Composite structure diagram

- Intended to describe the internal structure of a **container class**, including the components or **parts** and the connections between them.
- The parts are translated as "**composition relationships**" between the internal components and their containers.
- The container provides ports to receive and send signals in order to communicate with its environment.
- These signals are consumed or produced by the internal parts using internal ports and implementing or declaring as requiring specific interfaces.
- Lines connecting external and internal ports are called "**connectors**" and can be uni or bi-directional.

Composite structure diagram



Statechart Diagrams

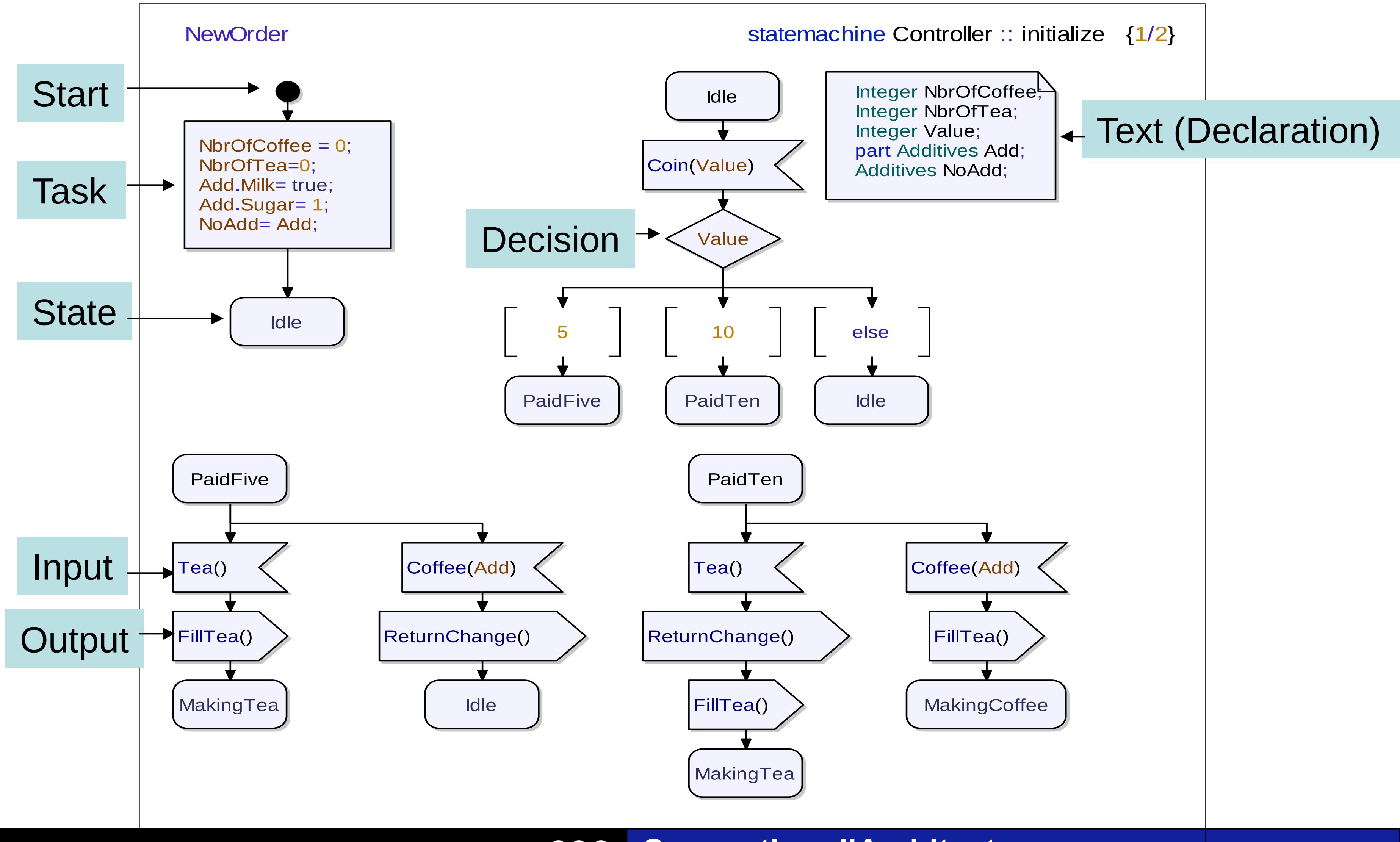
- Intended to describe the behavior of an **active class** using a **state machine**.
- A state machine has one or more possible **states** and a change of state is triggered by a **signal reception**.
- In UML 2, two notations can be used:
  - **transition-oriented:** suitable for detailed design (SDL)
  - **state-oriented:** overview of large design

Statechart Diagrams

- Once the state charts have been specified, UML tools can use them in order to **generate code** for **simulations** purposes or to **implement** the system.

- Manual (or automatic) **tests** can be done, in order to compare if the **implemented behavior** (specified using State Charts) corresponds to the **specified behavior** previously described in the sequence diagrams.

Statechart Diagrams



Defining passive behavior: methods

public void setMode(Integer mode)

Text diagram

```
if (mode==QUICK)
    {
      rinseTime=10;
      washTime=8;
      dryTime=3;
    }
else if (mode==NORMAL)
    {
        rinseTime=10;
        washTime=20;
        dryTime=5;
    }
else if (mode==INTENSIVE) {
        rinseTime=20;
        washTime=30;
        dryTime=5;
    }
```

public Duration getWashTime()

Text diagram

```
return Duration(washTime);
```

WashingData

- washTime : Integer = 20
- rinseTime : Integer = 10
- dryTime : Integer = 5

+ setMode (mode : Integer)
+ getRinseTime () : Duration
+ getWashTime() : Duration
+ getDryTime() : Duration

When UML is the center of the Software Engineering Process

Generating implementation: MDE

- Model-Driven Engineering (or MDE) refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle.
- MDE can be applied to software, system, and data engineering.
- MDE initiatives:
  - MDD: Model-Driven Development
  - MDA: Model-Driven Architecture

MDD

- Model-Driven Development refers to a range of development approaches that are based on the use of software modeling as a primary form of expression.
- Sometimes models are constructed to a certain level of detail, and then code is written by hand in a separate step.
- Sometimes complete models are built including executable actions.
- Code can be generated from the models, ranging from system skeletons to complete, deployable products.

MDD and MDA

- MDD technologies with a greater focus on architecture and corresponding automation yield higher levels of abstraction in software development.
- This abstraction promotes simpler models with a greater focus on problem space.
- Combined with executable semantics this elevates the total level of automation possible.
- The OMG has developed a set of standards called Model Driven Architecture (MDA), building a foundation for this advanced architecture-focused approach.

MDA

- MDA provides a set of guidelines for structuring specifications expressed as models.
- The MDA approach defines system functionality using a platform-independent model (PIM) using an appropriate domain-specific language.
- Then, given a platform definition model (PDM) corresponding to CORBA, .NET, the Web, etc., the PIM is translated to one or more platform-specific models (PSMs) that computers can run.
- The PSM may use different Domain Specific Languages, or a General Purpose Language like Java, C#, Python, etc.

MDA models

- **Computation-independent model (CIM)**
  - Describes the requirements for a system and the business context in which the system will be used.
  - The model typically describes what a system will be used for, not how it is implemented.
  - CIMs are often expressed in business or domain-specific language and make only limited reference to the use of IT systems when they are part of the business context.
- **Platform-independent model (PIM)**
  - Describes how the system will be constructed, without reference to the technologies used to implement the model.
  - This model does not describe the mechanisms used to build the solution for a specific platform.
  - A PIM may be appropriate when implemented by a particular platform or it may be suitable for implementation on many platforms.
- **Platform-specific model (PSM)**
  - Describes a solution from a particular platform perspective.
  - It includes the details that describe how the CIM can be implemented and how the implementation is realized on a specific platform.
  - It is obtained by transformation using the adequate Platform Definition Model (PDM)

References

Ian Sommerville and Pete Sawyer. 1997. *Requirements Engineering: A Good Practice Guide* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

Derek Hatley, Peter Hruschka, and Imtiaz A. Pirbhai. 2000. *Process for System Architecture and Requirements Engineering*. Dorset House Publ. Co., Inc., New York, NY, USA.

« Object-Oriented Software Engineering: A Use Case Driven Approach » by Jacobson I. et al.

« Object-Oriented Analysis and Design: Understanding System Development with UML 2.0 » by Mike O'Docherty

« Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development » by Craig Larman

« Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives» , T. Gardner and L. Yusuf (IBM)