Object Oriented and Agile Software Development

Part 1: Agile and Iterative Software Development

Christine Solnon

INSA de Lyon - 4IF - 2022/2023

Context

Organisation of the IF Curriculum:

- Information systems
- Computer networks
- Computer architecture
- Operating systems
- Mathematical methods and tools
- General formation
- **Software development:**

Last year (3IF):

- OO Programming
- Algorithms
- Software engineering

This year (4IF):

- **PLD Agile**
- Formal approaches
- Grammars and languages

Skills (1/2)

Use UML diagrams to model systems

- Understand a given UML diagram
  $\rightsquigarrow$ IF3-GL, IF4-MARS, **IF4-Agile**
- Design a UML diagram to model a system
  $\rightsquigarrow$ IF3-GL, IF4-MARS, **IF4-Agile**
- Check the consistency of different UML diagrams modeling a same system
  $\rightsquigarrow$ IF3-GL, IF4-MARS, **IF4-Agile**

Design the architecture of an object oriented software

- Structure a software in loosely coupled and highly cohesive classes
  $\rightsquigarrow$ IF3-GL, IF3-C++, **IF4-Agile**
- Understand and use Design Patterns
  $\rightsquigarrow$ IF3-GL, IF3-C++, **IF4-Agile**

Skills (2/2)

Use a methodology to design, implement and maintain softwares

- Use an iterative software development process
  $\rightsquigarrow$ **IF4-Agile**
- Implement principles of the Agile manifesto
  $\rightsquigarrow$ **IF4-Agile**

Implement high quality softwares

- Use appropriate object oriented mechanisms (inheritance, genericity, ...)
  $\rightsquigarrow$ IF3-C++, IF3-GL, **IF4-Agile**

Organisation

Courses

- CM1 and CM2: Iterative Software Development
- CM3 and CM4: Object Oriented Design and Design Patterns
- CM5: Social and Environmental Challenges of Software Development
- CM6 and CM7: Software quality (P.-E. Portier)

Long Duration Project (PLD)

- 1 session to introduce the project
- 8 practical sessions of 4h

Design and implement a software for planning delivery tours

Evaluation

- Project deliverables

Some references that you may read ...

Books:

- Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development

  Craig Larman

- Modélisation Objet avec UML

  Pierre-Alain Muller, Nathalie Gaertner

- Head first: Design Patterns

  Eric Freeman & Elizabeth Freeman

Web sites:

- www.agilealliance.org
- www.scrumguides.org

Overview

1. **Introduction**
   - Motivations
   - Some (quick) recalls on the context
2. **Agile and Unified Software Development Process (UP)**
3. **Description of one iteration**

Two quotes for starting

Philippe Kruchten:

Programming is fun, but developing quality software is hard. In between the nice ideas, the requirements or the "vision", and a working software product, there is much more than programming. Analysis and design, defining how to solve the problem, what to program, capturing this design in ways that are easy to communicate, to review, to implement, and to evolve is what...
... you will learn in this course (?)

Craig Larman:

The proverb "owning a hammer doesn't make one an architect" is especially true with respect to object technology. Knowing an object-oriented language (such as Java) is a necessary but insufficient first step to create object systems. Knowing how to "think in objects" is also critical.

Software Crisis?

1968: NATO Software Engineering Conference

First mentions of "software crisis" and "software engineering"

1972: ACM Turing Award Lecture of Dijkstra

The major cause of the software crisis is that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and **now we have gigantic computers, programming has become an equally gigantic problem**.

1979: Study of the Government Accounting Office on 163 projects

- 29% have never been completed
- 45% have been completed, but not used
- 19% have been completed, but have been modified before utilisation
- 7% have been completed and used without modifications

Some examples

1994 $\rightsquigarrow$ 2005: Baggage handling system at Denver Airport

- Cost=193 M$; 16 months late $\rightsquigarrow$ Replaced by manual system in 2005

1999 $\rightsquigarrow$ 2011: New York City Automated Payroll (NYCAP) System

- Estimated cost = 66 M$ $\rightsquigarrow$ Actual cost > 360 M$

Logiciel unique à vocation interarmées de la solde (Louvois)

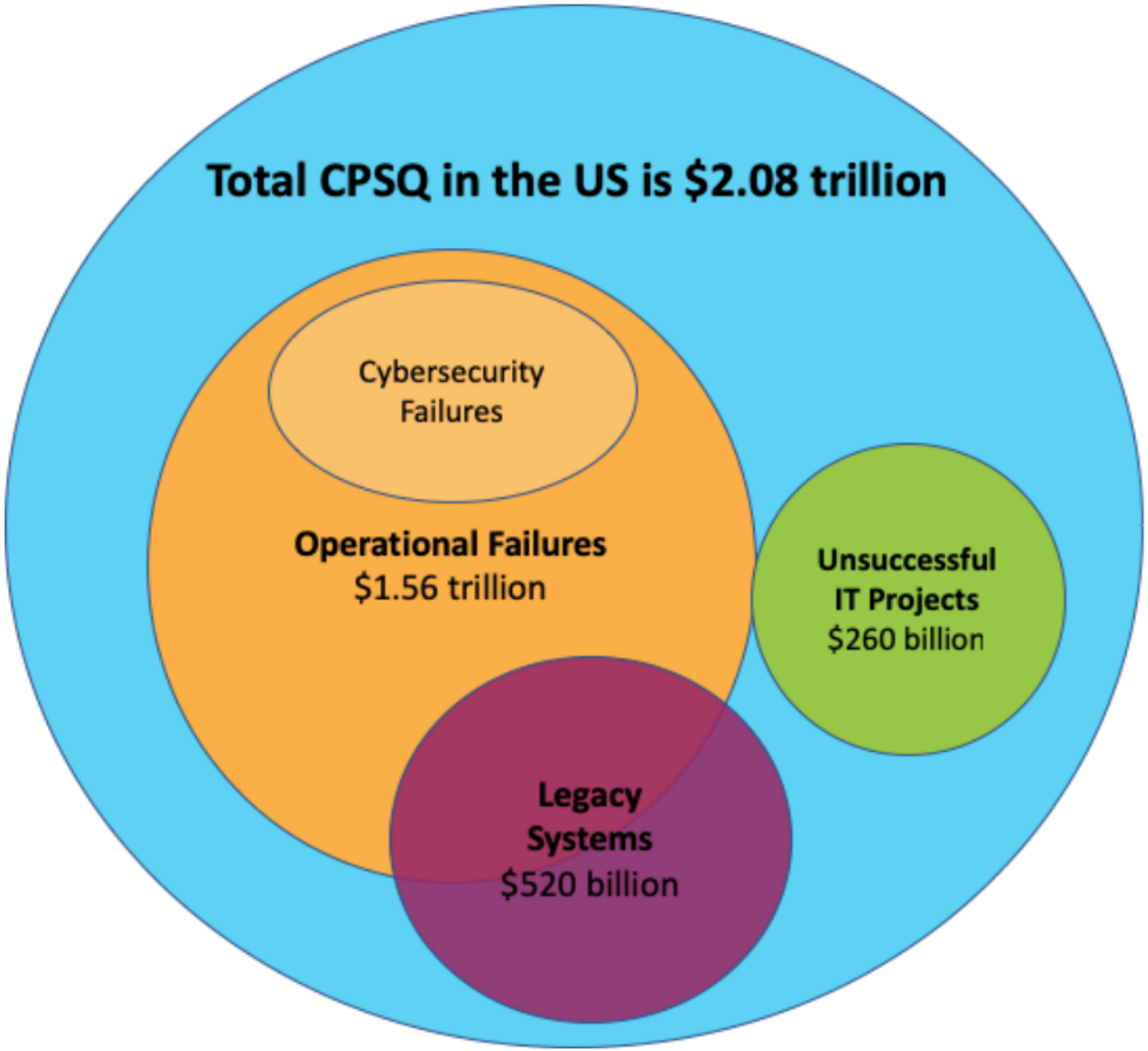
- From 1999 to 2003: Adaptation of an ERP; aborted (Cost=20M€)
- 2004: Dev. of an engine linked with an existing information system
  Progressive deployment in 2011 $\rightsquigarrow$ Abortion in 2013 (Cost=470M€)
- 2019: Launching of "Source Solde" (128M€)
- 2021: "Source Solde" is successfully used!

And two very expensive bugs...

- 1996: Ariane 5
- 1999: NASA Mars Climate Orbiter

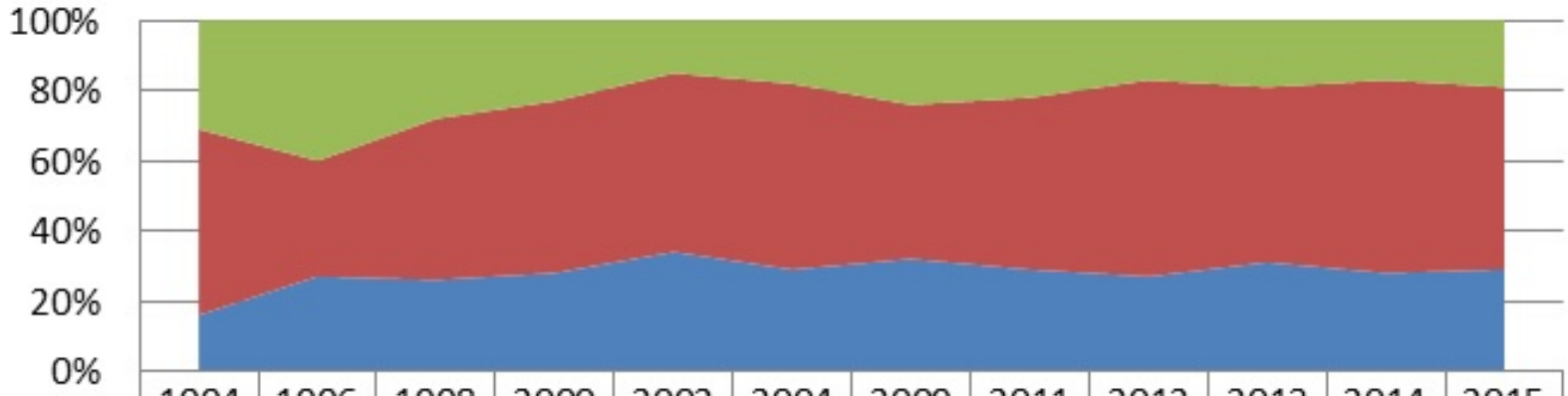
Cost of Poor Software Quality (CPSQ) in the US in 2020

`https://www.it-cisq.org/pdf/CPSQ-2020-report.pdf`



Study of the Standish Group (1/3)

Success/Failure of IT projects (over 50000 projects / year)



	1994	1996	1998	2000	2002	2004	2009	2011	2012	2013	2014	2015
Failed	31%	40%	28%	23%	15%	18%	24%	22%	17%	19%	17%	19%
Challenged	53%	33%	46%	49%	51%	53%	44%	49%	56%	50%	55%	52%
Successful	16%	27%	26%	28%	34%	29%	32%	29%	27%	31%	28%	29%

- Failed: cancelled at some point during the development cycle
- Challenged: completed and operational but over-budget, over the time estimate, and offers fewer features and functionalities
- Successful: completed on-time and on-budget, with all features and functionalities as initially specified

In 2020: 19% failed, 50% challenged, 31% success

Study of the Standish Group (2/3)

Influence of the size of the project on success (from 2011 to 2015)

CHAOS RESOLUTION BY PROJECT SIZE

	SUCCESSFUL	CHALLENGED	FAILED
Grand	2%	7%	17%
Large	6%	17%	24%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%
TOTAL	100%	100%	100%

The resolution of all software projects by size from FY2011–2015 within the new CHAOS database.

Conclusion of the Standish group:

It is critical to break down large projects into a sequence of smaller ones, prioritized on direct business value, and install stable, full-time, cross-functional teams that execute these projects following a disciplined agile and optimization approach.

Study of the Standish Group (3/3)

Main failure causes

1. Lack of implication of the user 12,8%
2. Incomplete requirements and specifications 12,3%
3. Modification of the requirements and specifications 11,8%

Conclusion:

"Research at the Standish Group indicates that **smaller time frames, with delivery of software components early and often, will increase the success rate**. Shorter time frames result in an **iterative process** of design, prototype, develop, test and deploy small elements. This process is known as **growing software** as opposed to the old concept of developping software. Growing software **engages the user earlier**."

1999: Unified Software Development Process (USDP or UP)

- Iterative process for developing software
- Flexible and open process $\rightsquigarrow$ Agile and XP compatible!

Overview

1. **Introduction**
   - Motivations
   - Some (quick) recalls on the context
2. **Agile and Unified Software Development Process (UP)**
3. **Description of one iteration**

What is a software?

Set of artifacts

- Code: Source, Binary, Tests, ...
- User documentation: Reference manuals, tutorials, ...
- Technical documentation: UML diagrams, ...
- ...

Designed by and for different actors

- Client
- Users
- Programmers
- Hotline
- ...

What is a good software?

Different points of view:

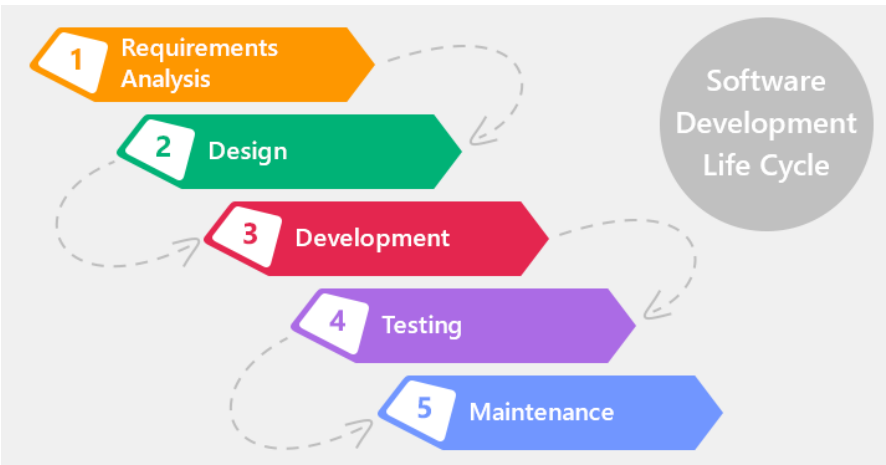
- The user — **What does it do?**
  - $\rightsquigarrow$ Functional and non functional needs
- The programmer — **How is it done?**
  - $\rightsquigarrow$ Source code, Architecture, Technical documents, ...
- The provider — **How much does it cost/payoff?**
  - $\rightsquigarrow$ Development and maintenance costs, ...
- The hotline — **Why does it fail?**
  - $\rightsquigarrow$ diagnostic, reproducibility, remote administration, ...
- ...

Activities of a Software Process (recalls from 3IF):

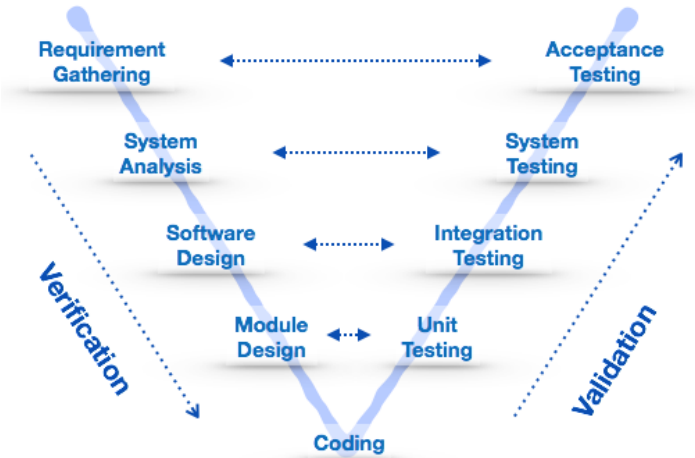
- Capture and analysis of the requirements
  $\leadsto$ Specification of functional and non functional needs
- Design
  $\leadsto$ Conceptual solution (models) that fulfills the requirements
- Development and Test
  $\leadsto$ Code
- Integration and Test
  $\leadsto$ Operational software
- Maintenance

Linear Software Development Life Cycle Models

Waterfall



V-shaped



Problem:

These models assume that

- It is possible to specify complete and correct needs
- Needs won't change

However, 90% of the costs are due to maintenance and evolution!

Maintenance and Evolution

Utilisation of specified functionalities in waterfall models [C. Larman]:

- Never 45%
- Rarely 19%
- Sometimes 16%
- Often 13%
- Always 7%

Repartition of maintenance costs [C. Larman]:

- User extensions 41,8%
- Error correction 21,4%
- Modification of data format 17,4%
- Hardware modification 6,2%
- Documentation 5,5%
- Efficiency 4%

If you think writing software is difficult, try re-writing software

Bertrand Meyer

Overview

1 Introduction

2 **Agile and Unified Software Development Process (UP)**

3 Description of one iteration

The Agile manifesto (2001) www.agilealliance.com

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions . . . over processes and tools
- Working software . . . over comprehensive documentation
- Customer collaboration . . . over contract negotiation
- Responding to change . . . over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

12 Principles of the Agile Manifesto (1/2)

1: Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

2: Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3: Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4: Business people and developers must work together daily throughout the project.

5: Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6: The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

12 Principles of the Agile Manifesto (2/2)

7: Working software is the primary measure of progress.

8: Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9: Continuous attention to technical excellence and good design enhances agility.

10: Simplicity (the art of maximizing the amount of work not done) is essential.

11: The best architectures, requirements, and designs emerge from self-organizing teams.

12: At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

Warning: “Be agile” doesn’t mean “Don’t design models”
$\rightsquigarrow$ Models are used to understand, communicate, and explore

Subway Map to Agile Practices

https://www.agilealliance.org/agile101/subway-map-to-agile-practices/

The UP Life Cycle

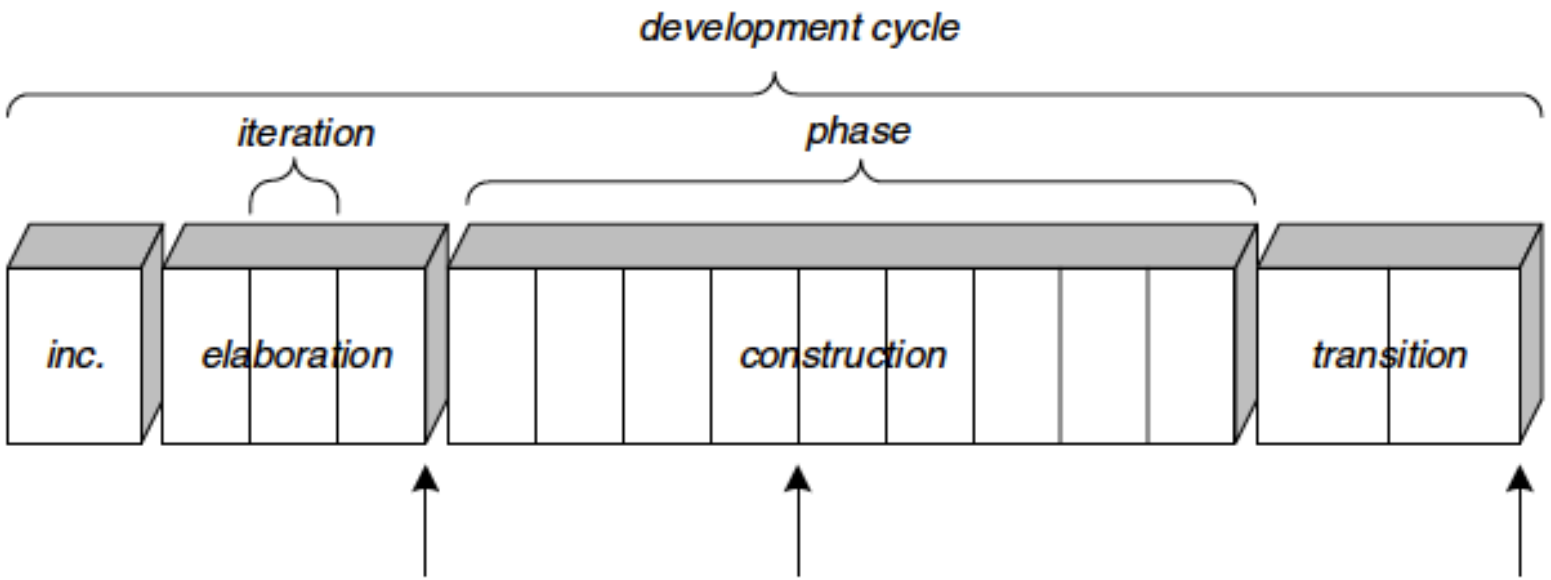
The life of a software is composed of cycles

- 1 cycle $\Rightarrow$ 1 new version of the software
- Each cycle is composed of 4 phases:
  $\rightsquigarrow$ Inception, Elaboration, Construction, Transition
- Each phase is composed of iterations $\Rightarrow$ Increments

Why an Iterative Process?

- User feedback at the end of each iteration
  $\rightsquigarrow$ Adapt the system to actual user needs
- Development team feedback at the end of each iteration
  $\rightsquigarrow$ Improve organisation
- Shorter and simpler steps
  $\rightsquigarrow$ Avoid the “analysis paralysis”
- Important risks are studied during the first iterations
  $\rightsquigarrow$ Design and stabilise the architecture quickly

Graphical Representation of the UP Life Cycle



[figure from C. Larman]

Phase 1: inception

- Very short phase (usually: only one iteration)
- Preliminary study to evaluate feasibility and risks
  - What should the system do?
  - What could the architecture look like?
  - What are the risks?
  - Very rough approximation of costs and durations

$\rightsquigarrow$ **Should we accept the project?**

Phase 2: Elaboration

A few iterations, guided by risks

- Identify and stabilize most needs
  $\rightsquigarrow$ Most use cases are specified
- Design the basic architecture
  $\rightsquigarrow$ Framework of the system
- Code and test the most critical use cases ($<$10% of the needs)
  $\rightsquigarrow$ Test as soon as possible, often, and within a realistic context
- Reliable estimation of costs and durations

$\rightsquigarrow$ **The main needs are identified**
$\rightsquigarrow$ **The architecture is stabilized**
$\rightsquigarrow$ **Risks are controlled**

Phase 3: Construction

Most expensive phase ($>$50% of the cycle)

- Incremental growth

  $\rightsquigarrow$ Stable architecture with minor changes

- Until all use cases are realized

$\rightsquigarrow$ **The system is correct and complete enough to be deployed**

Phase 4: Transition

- Beta version deployed
- Correction of the remaining errors
- Tests and improvements, user formation, installation of online help, ...

Overview

1 Introduction

2 Agile and Unified Software Development Process (UP)

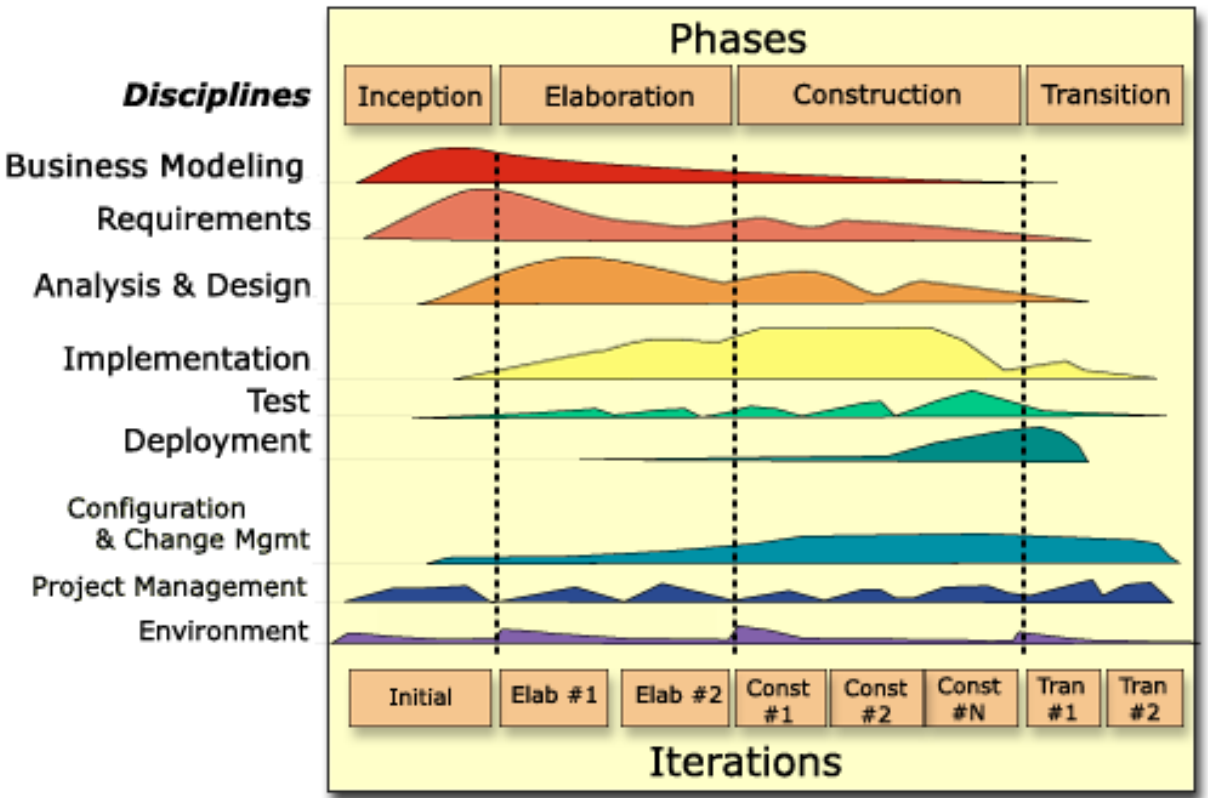
3 **Description of one iteration**

Overview of an iteration

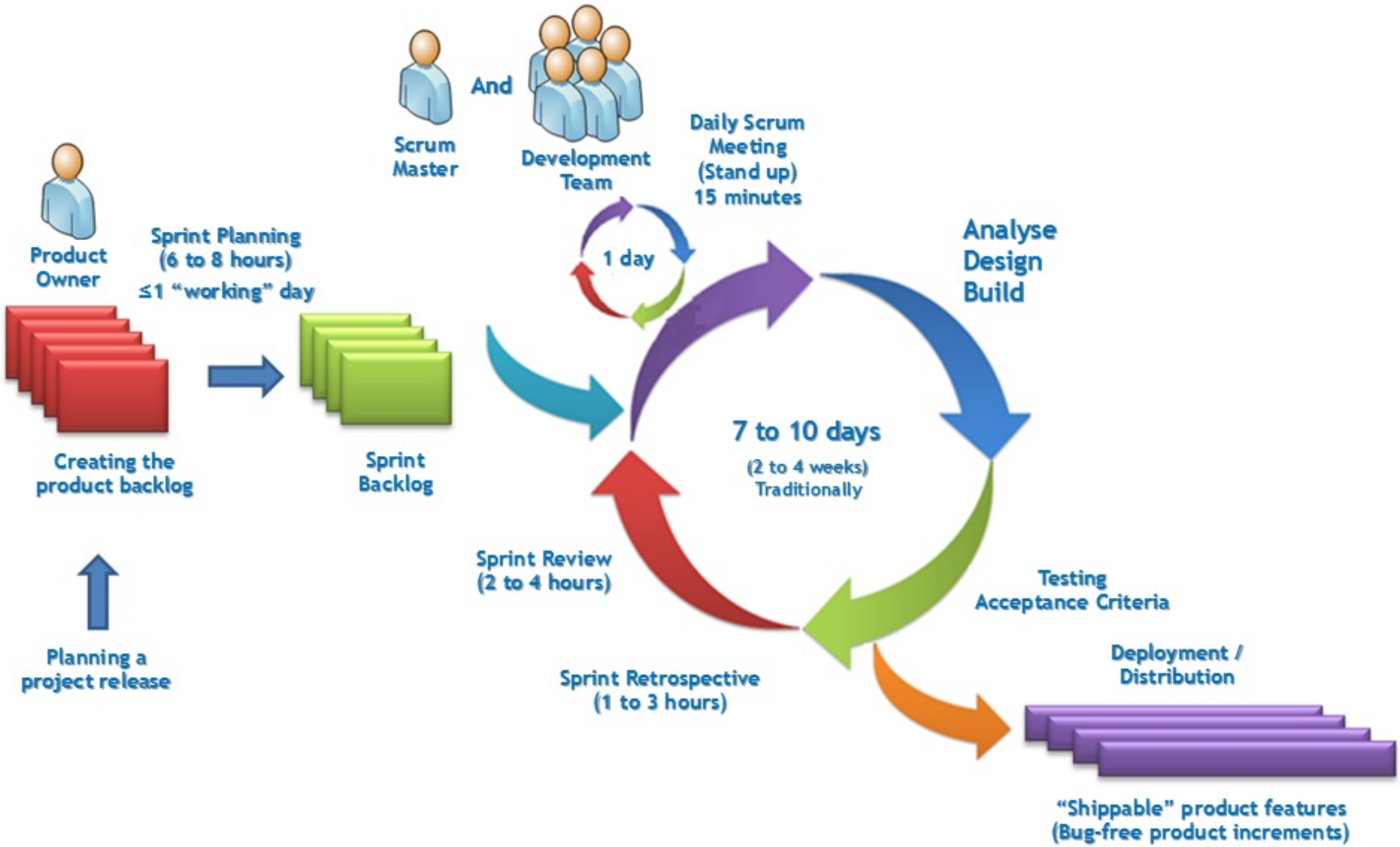
Each iteration is a small waterfall:

- Requirement, analysis $\rightsquigarrow$ Model the system seen from the outside
- Design $\rightsquigarrow$ Model the system seen from the inside
- Integration and tests $\rightsquigarrow$ Operational software

$\rightsquigarrow$ **Proportions of these activities vary from one iteration to another**



Agile iteration $\rightsquigarrow$ Sprint / Timebox



Mode details on www.scrumguides.org

Overview

1. **Introduction**
2. **Agile and Unified Software Development Process (UP)**
3. **Description of one iteration**
   - Capture and Analysis of the Requirements
   - Design
   - Development and Test
   - Project Management

Capture and Analysis of the Requirements

Goal: Agree on the features of the system to build

$\rightsquigarrow$ Specification of functional and non functional needs

Why is it difficult?

- Users don't really know their needs...
  ... and needs change, especially when introducing a new system!
- Developers may not know the application domain
- Users and developers have different languages
- A compromise between services and costs must be found
- ...

Goals and Artifacts

Goals

- Understand the system context
- Capture functional needs
- Capture non functional needs

Artifacts / Deliverables

- Domain Model and Business Object Model
- Glossary
- Use Case Model
- Supplementary Specifications

Domain Model

What is a domain model?

Class diagram with conceptual classes only (real world classes)
$\rightsquigarrow$ Few attributes, **no operations, no software classes**

How to build a domain model?

- Reuse (and modify) existing models!
- Use category lists:
  - Classes: *Business transactions, Products/services related to transactions, Actors, Places, ...*
  - Associations: *is-a-description-of*, *is-part-of*, ...
- Identify names (and noun phrases) in textual descriptions

Domain Models in Agile Projects

Goal: Understand key concepts and their relations $\rightsquigarrow$ Visual dictionary

- Don't try to be exhaustive and correct since the first iteration
- Domain models evolve through iterations

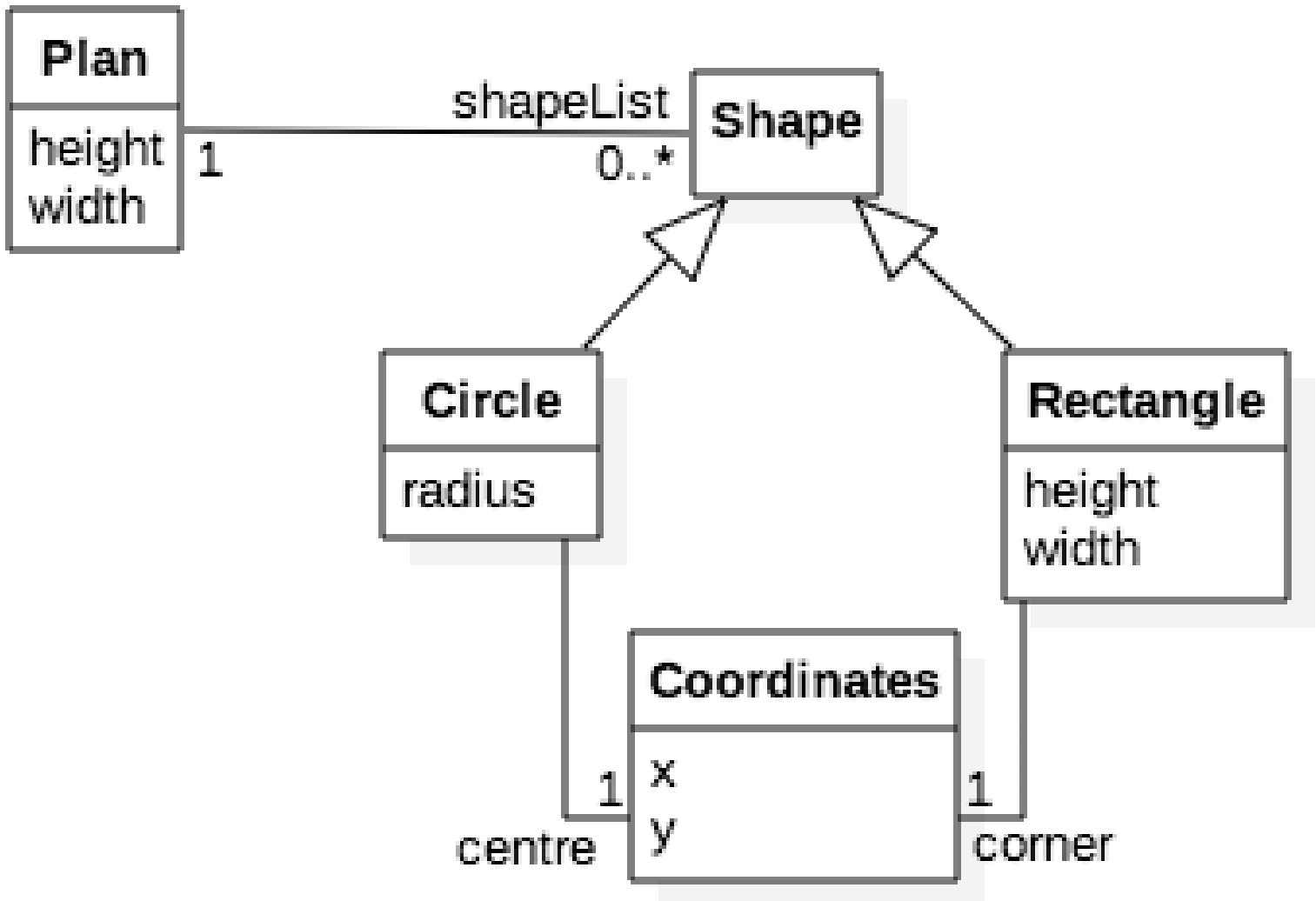
Exercise: Domain Model of PlaCo

A sawmill wants a system for drawing plans and transfer them to a wood cutting machine.

- A plan is a rectangle with an height and a width.
- The system must be able to add, delete and move shapes on a plan, to save and load plans, and to transfer a plan to the cutting machine.
- A shape is a rectangle or a circle:
  - A rectangle has an height and a width, and its position is defined by its upper left corner coordinates;
  - A circle has a radius, and its position is defined by its centre coordinates.

  Coordinates and length are integer values expressed with respect to some given unit. Shapes must have empty intersections.

First Domain Model of PlaCo



Other artifacts to “Understand the system context"

Business Object Model

- Model more general than the domain model

  Abstraction of the way workers and business entities are related, and collaborate in order to achieve activities

  $\rightsquigarrow$ Class, Activity, Collaboration, and Sequence Diagrams
  $\rightsquigarrow$ More informations in PLD MARS

Glossary

- Definition of the vocabulary related to the application
  $\rightsquigarrow$ Avoid ambiguities
- Each term occuring in Use Cases, Domain or Business Object Models must be defined in the glossary

Goals and Artifacts

Goals

- Understand the system context
- Capture functional needs
- Capture non functional needs

Artifacts / Deliverables

- Domain Model and Business Object Model
- Glossary
- Use Case Model
- Supplementary Specifications

Use Cases (Recalls from 3IF)

What is a Use Case?

- Utilisation of the system by an actor (human or exterior system)
  $\rightsquigarrow$ Sequence of interactions between the system and actors
- Usually composed of several scenarios
  $\rightsquigarrow$ Nominal scenario (basic flow) and extensions (alternative flows)

Warning: System = Black Box

$\rightsquigarrow$ Describe **what** the system must do, not how it will do it

Why Use Cases?

- Simple procedure for the client to describe her needs
  - Reach an agreement between clients and development teams
- Starting point for the next activities
  - Design and Implementation $\rightsquigarrow$ Realise Use Cases
  - Functional tests $\rightsquigarrow$ Use Case Scenarios

Use Cases (Recalls from 3IF)

How to discover Use Cases?

- Identify the system boundary
- Identify primary actors
  - $\rightsquigarrow$ Those who achieve a goal when using the system
- For each primary actor, identify her goals
- Define Use Cases corresponding to these goals

$\rightsquigarrow$ Requirements workshop that gather clients, users, architects and programmers

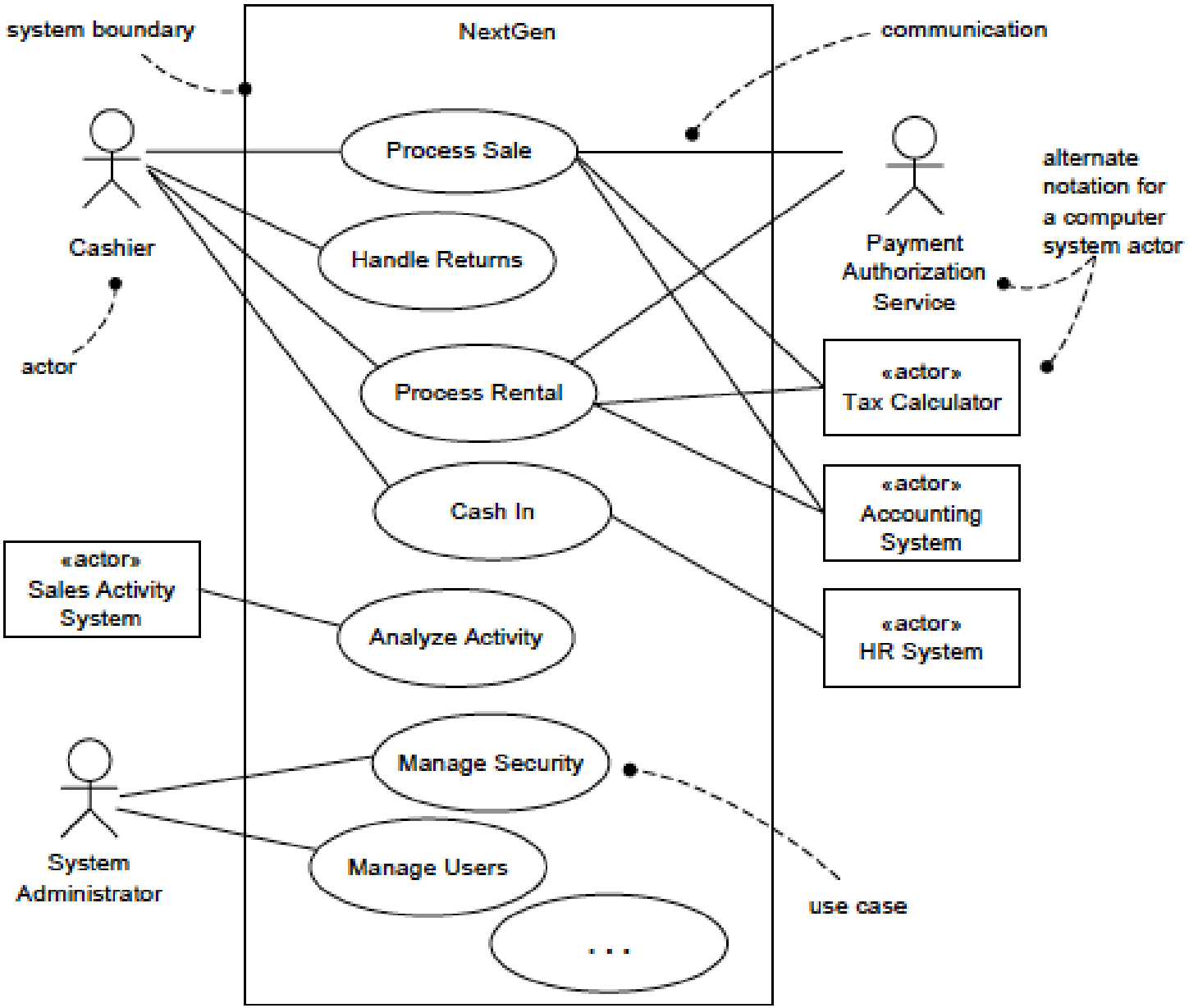
How to describe Use Cases?

Use Case Model:

- Use Case Diagram
- Textual Description of Use Cases
- System Sequence Diagrams of Use Case Scenarios

Use Case Diagram (Recalls from 3IF)

Relations between Use Cases and Actors (image from Craig Larman)



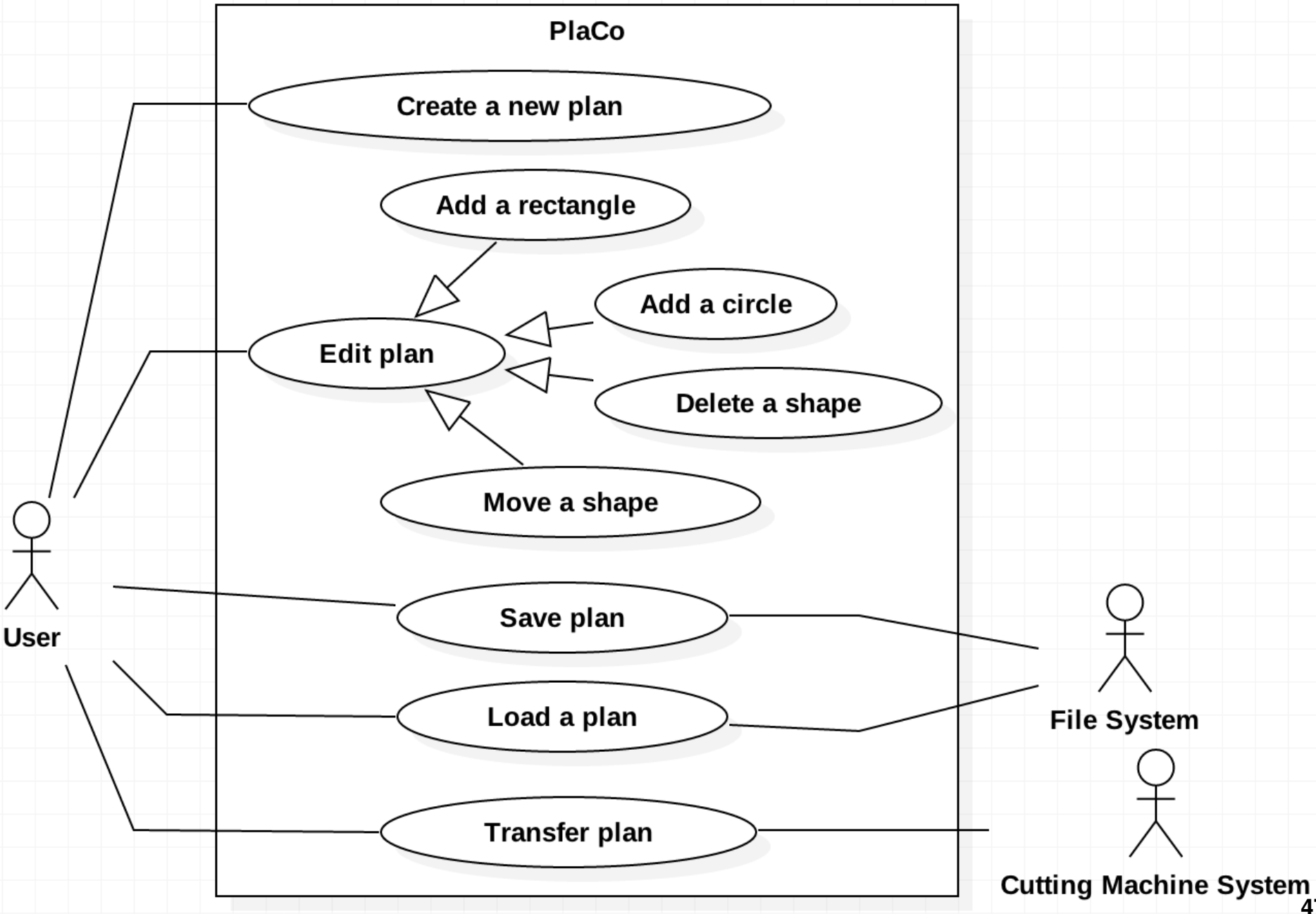
Exercise: Use Case Diagram for PlaCo

A sawmill wants a system for drawing plans and transfer them to a wood cutting machine.

- A plan is a rectangle with an height and a width.
- The system must be able to add, delete and move shapes on a plan, to save and load plans, and to transfer a plan to the cutting machine.
- A shape is a rectangle or a circle:
  - A rectangle has an height and a width, and its position is defined by its upper left corner coordinates;
  - A circle has a radius, and its position is defined by its centre coordinates.

  Coordinates and length are integer values expressed with respect to some given unit. Shapes must have empty intersections.

Use Case Diagram for PlaCo



Textual Description of a Use Case (Recalls from 3IF)

Each Use Case is composed of a set of scenarios

- Nominal scenario
- Alternate scenarios (one for each possible particular case)

What is a scenario?

Sequence of interactions between actors and the system

Description of a Use Case

- Brief format: Nominal scenario described in one paragraph
  $\rightsquigarrow$ Story of an actor who uses the system **to reach a goal**
- Structured description (according to Martin Fowler):
  - Title: Goal of the main actor of the use case (starts with a verb)
  - Preconditions: Conditions that must be true before starting
  - Nominal scenario: Sequence of interactions between actors and the system
  - Extensions: A sequence of interactions for each alternative case

Example: Description of "Add a rectangle" (1/2)

Brief Format:

The user tells the system she wants to add a rectangle. She enters the coordinates of two opposite corners of the rectangle. The system adds the rectangle to the plan.

Structured Description:

- Precondition: a plan is loaded
- Nominal Scenario:
  1. The user tells the system she wants to add a rectangle
  2. The system asks to enter the coordinates of a first corner
  3. The user enters the coordinates of a point p_1
  4. The system asks to enter the coordinates of the opposite corner
  5. The user enters the coordinates of a point p_2
  6. The system adds the rectangle defined by (p_1, p_2) in the plan and displays the plan

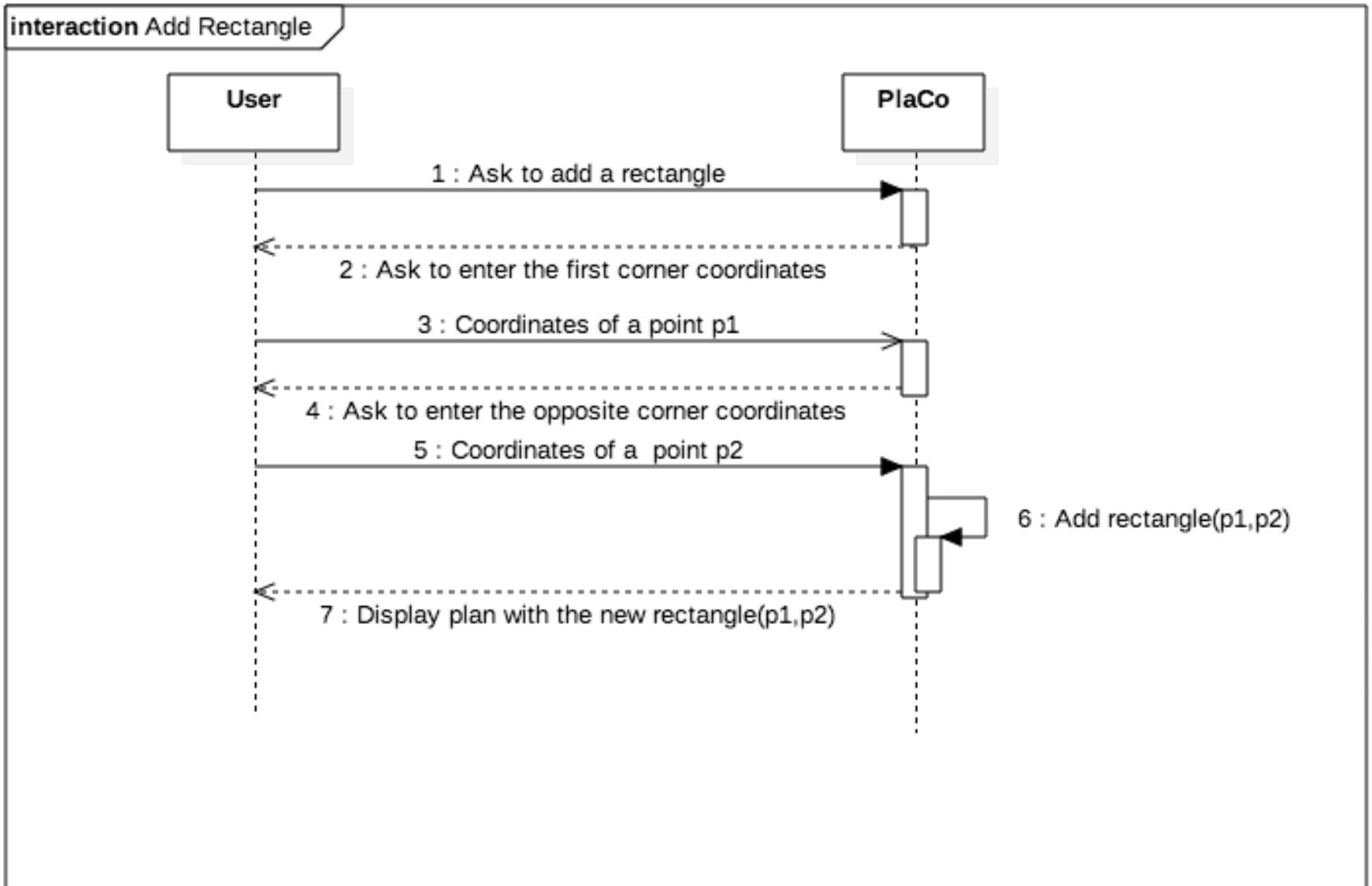
Example: Description of "Add a rectangle" (2/2)

Structured Description (continued):

- Extensions:
  - **3a** Point p_1 does not belong to the plan
    - The system notifies the user that p_1 is not valid and goes back to Step 2
  - **3b** Point p_1 already belongs to a shape in the plan
    - The system notifies the user that p_1 is not valid and goes back to Step 2
  - **5a** Point p_2 does not belong to the plan
    - The system notifies the user that p_2 is not valid and goes back to Step 4
  - **5b** The rectangle defined by (p_1, p_2) has a non empty intersection with a shape of the plan
    - The system notifies the user that p_2 is not valid and goes back to Step 4
  - **1-5a** The user tells the system she wants to cancel the action
    - The system cancels the action

System Sequence Diagram of a Use Case (Recalls from 3IF)

↝ Graphical representation of a Use Case scenario



Iterative Modelling of Use Cases

Use Case Models are progressively refined:

- Iteration 1 / Inception Phase:
  - Most Use Cases are identified
  - Nearly 10% of them are analysed
    $\rightsquigarrow$ Most significant/risky/valuable cases
- Iteration 2 / Elaboration Phase:
  - Nearly 30% of the cases are analysed
  - Design and implementation of the most significant/risky/valuable ones
- Each of the next iterations of the Elaboration Phase:
  - Detailed analysis, design and implementation of some use cases
- Last iteration of the Elaboration Phase:
  - Most cases are identified
  - from 40 to 80% of them are analyzed
  - The most significant/risky/valuable ones are implemented

$\rightsquigarrow$ The architecture is stable

Goals and Artifacts

Goals

- Understand the system context
- Capture functional needs
- Capture non functional needs

Artifacts / Deliverables

- Domain Model and Business Object Model
- Glossary
- Use Case Model
- Supplementary Specifications

Goal “Capture non functional needs"

Supplementary Specifications:

- Some non functional needs are already expressed in use case models

  $\rightsquigarrow$ Gather them
- List other non functional needs: URPS+
  - Usability
  - Reliability
  - Performance
  - Supportability
  - +: Other needs
    - Languages and tools, hardware, etc
    - Interface with external systems
    - Legal issues, licence
    - ...

Overview

1. Introduction
2. Agile and Unified Software Development Process (UP)
3. Description of one iteration
   - Capture and Analysis of the Requirements
   - Design
   - Development and Test
   - Project Management

Design

Why designing models?

To understand and communicate:

- What are object responsibilities?
- How do objects collaborate?
- What design patterns can be used?

$\rightsquigarrow$ Documentation may be generated from the code (reverse engineering)

How to design models?

- Design several models concurrently
  - Dynamic diagrams to model the behavior
  - Static diagrams to model the structure

  $\rightsquigarrow$ Check the consistency of these models
- Design with programmers, not for them!

Goals and Artifacts

Goals

- Model the behaviour
- Model the structure

Artifacts / Deliverables

- Sequence diagrams
- Statechart diagrams
- Class, package, and deployment diagrams

Sequence Diagrams (Recalls from 3IF)

↝ **Temporal point of view of object interactions**

When capturing the requirements: System = black box

↝ Sequences of interactions between actors and the system

- Describe use case scenarios

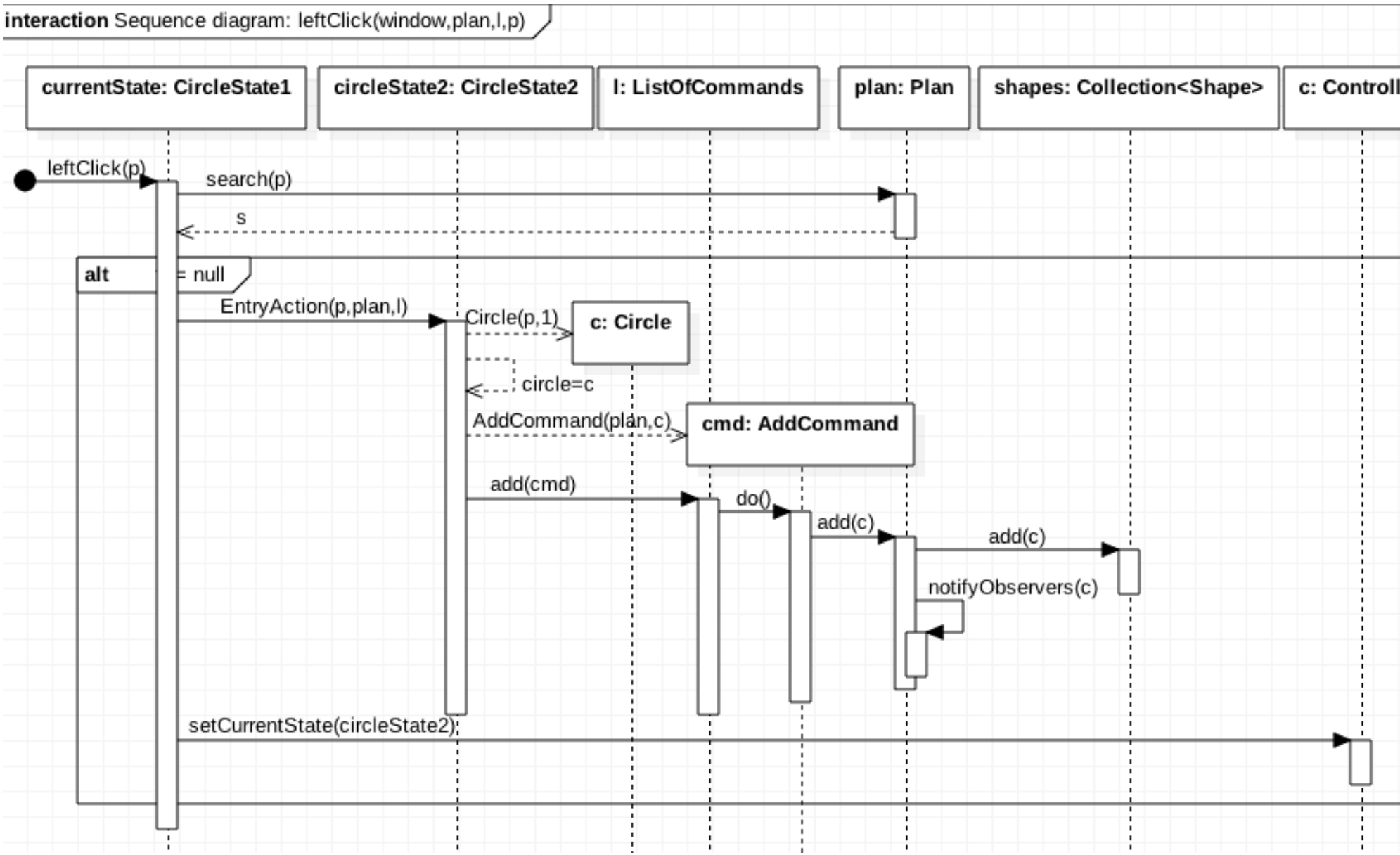
During the design step: Open the black box

↝ Interactions between software objects

- Assign responsibilities to objects
  - Who creates objects?
  - How to access to an object?
  - What object receives messages from the GUI?
  - ...

in order to have loosely coupled and highly cohesive objects

Example (see Part 2 of this course)



Goals and Artifacts

Goals

- Model the behaviour
- Model the structure

Artifacts / Deliverables

- Sequence diagrams
- Statechart diagrams
- Class, package, and deployment diagrams

Parenthesis on finite state automata (1/3)

A finite state automaton is defined by:

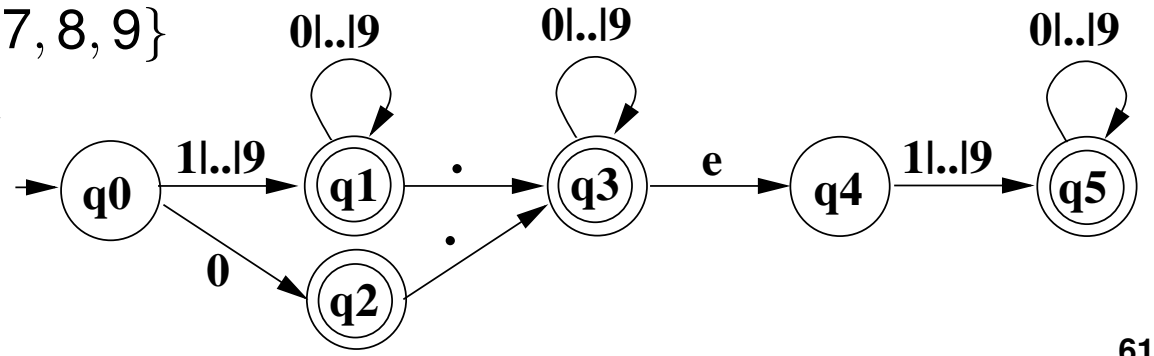
- A finite set of symbols (alphabet) Σ
- A finite set of states Q
- A set of initial states $I \subseteq Q$ and accept states $F \subseteq Q$
- A transition relation $R \subseteq Q \times \Sigma \times Q$
  - Interpretation of (q_i, s, q_j): Transition from q_i to q_j when reading s

Graph representation:

- Each state corresponds to a vertex
- Each transition (q_i, s, q_j) corresponds to an edge $q_i \xrightarrow{s} q_j$

Example:

- $\Sigma = \{e, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}$
- $I = \{q_0\}$
- $F = \{q_1, q_2, q_3, q_5\}$



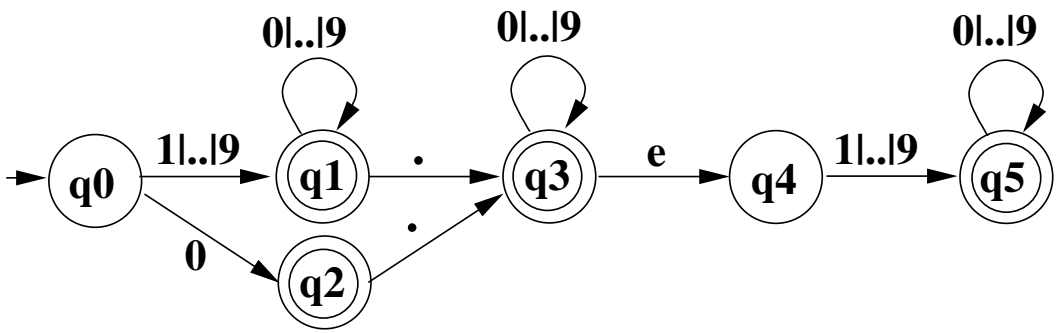
Parenthesis on finite state automata (2/3)

Using a finite state automaton to accept words:

- Input = a sequence of symbols from Σ
- Output = true (accept) or false (don't accept)
- A word $< s_1, ..., s_n >$ is accepted if there exist $< q_0, ..., q_n >$ such that: $q_0 \in I$, $q_n \in F$ and $\forall i \in [1..n], (q_{i-1}, s_i, q_i) \in R$
  $\rightsquigarrow$ Path from a state of I to a state of F

Example :

- Accepted words:
  `0`, `0.123e45`, `125`, ...
- Non accepted words:
  `012`, `4.5.6`, `1e2`, ...



How to modify the automaton to accept `1e2`?

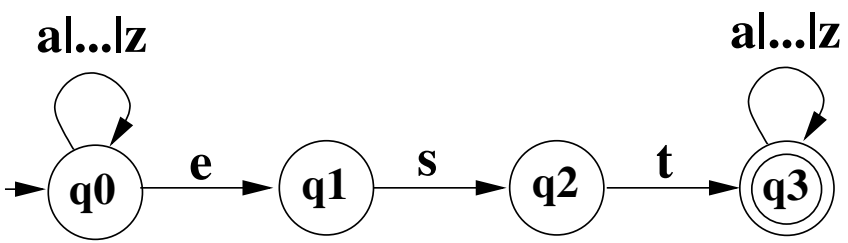
Parenthesis on finite state automata (3/3)

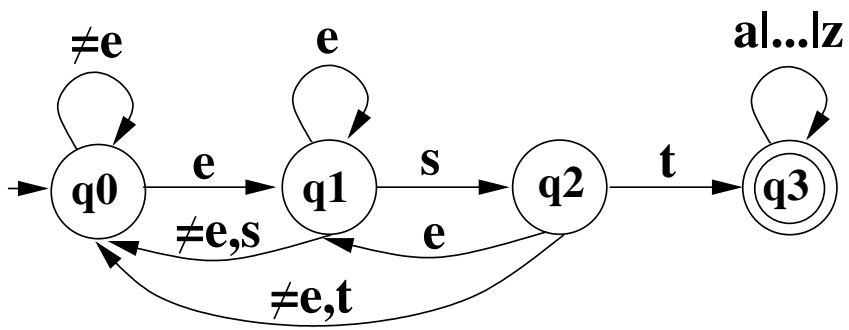
Deterministic and complete finite state automata

- An automaton is deterministic if R is a function from $Q \times \Sigma$ to Q
- An automaton is complete if R is a total function

It is always possible to transform a finite state automaton into an equivalent deterministic and complete automaton

Example:





Beyond finite state automata

Finite state automata are very efficient

- Time complexity linear wrt the size of the word
- But they cannot represent all languages
  - Ex: They cannot recognise well-formed parentheses!

Pushdown automata (finite state + stack) are more powerful

- They can recognise any context-free language (C++, Java, ...)
- But some languages are not context-free

A Turing machine (finite state + tape) is even more powerful

- It can recognise any decidable language
- But some languages are not decidable!

More on this topic in the course on Grammars and Languages!

Back to Statechart Diagrams

Why using Statechart diagrams?

To model the evolution of the state of a system with respect to events

What is the difference with automata?

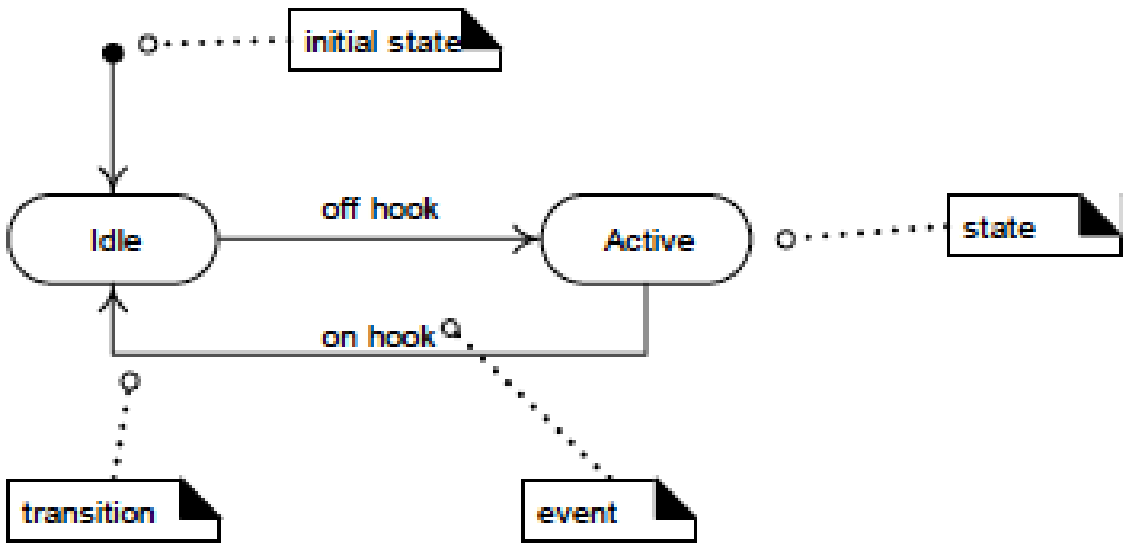
- Statechart diagram without actions nor guards:
  $\rightsquigarrow$ Finite state automata
- When actions are limited to pushing/poping symbols into/from a stack:
  $\rightsquigarrow$ Pushdown automata
- In the general case: Turing complete

Statechart Diagrams without Guards nor Actions

Particular kind of finite state automata:

- Replace symbols with events (reception of a signal, a message, ...)
- Events trigger transitions between states
  $\rightsquigarrow$ An event is lost if no transition is specified for it in the current state
- There is one initial state, but not necessarily an accept state

Example:



[Image from C. Larman]

Different Kinds of Events:

- Signals:

$$Q0 \xrightarrow{\texttt{signalName}} Q1$$

- Messages/Operations:

$$Q0 \xrightarrow{\texttt{opName(parametres)}} Q1$$

- Temporal events:

$$Q0 \xrightarrow{\texttt{after(x)}} Q1$$

$\rightsquigarrow$ Go to state $Q1$ `x` time units after arriving to state $Q0$

- Conditions:

$$Q0 \xrightarrow{\texttt{when(cond)}} Q1$$

$\rightsquigarrow$ Go to state $Q1$ when `cond` becomes true

Actions and Activities

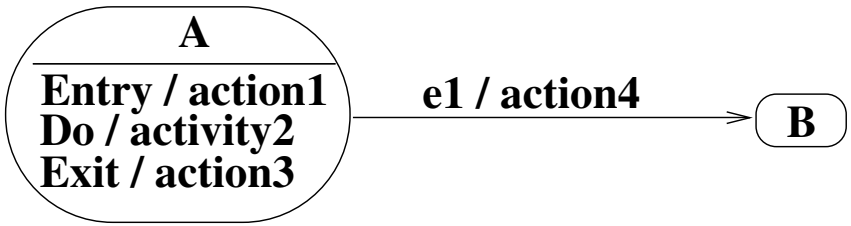
Actions (signal emission, method call, ...):

- Actions may be done:
  - During a transition (ex. : `action4`)
  - When entering a state (ex. : `action1`)
  - When leaving a state (ex. : `action3`)
- Actions are atomic (cannot be interrupted by an event)

Activities:

- May be executed in a state (ex. : `activity2`)
- May be continuous or not
- Are interrupted when leaving the state

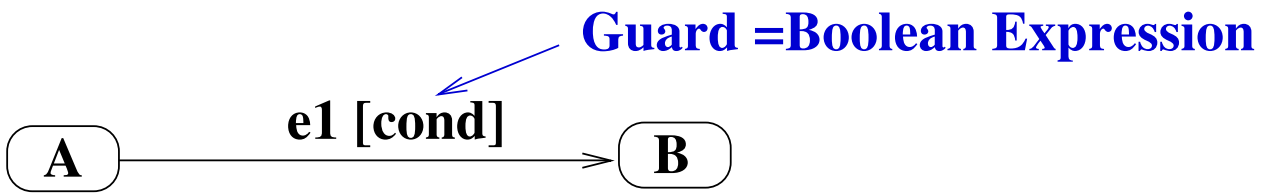
Example:



Order of execution : `action1 - activity2 - action3 - action4`

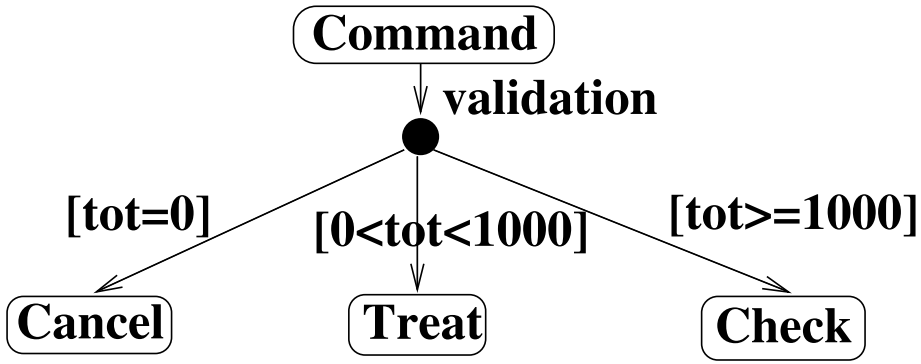
Guards and Composite Transitions

Guard Conditions:



- Transition from A to B if `cond` is true when `e1` occurs
  $\rightsquigarrow$ If `cond` is false then `e1` is lost

Composite Transitions:



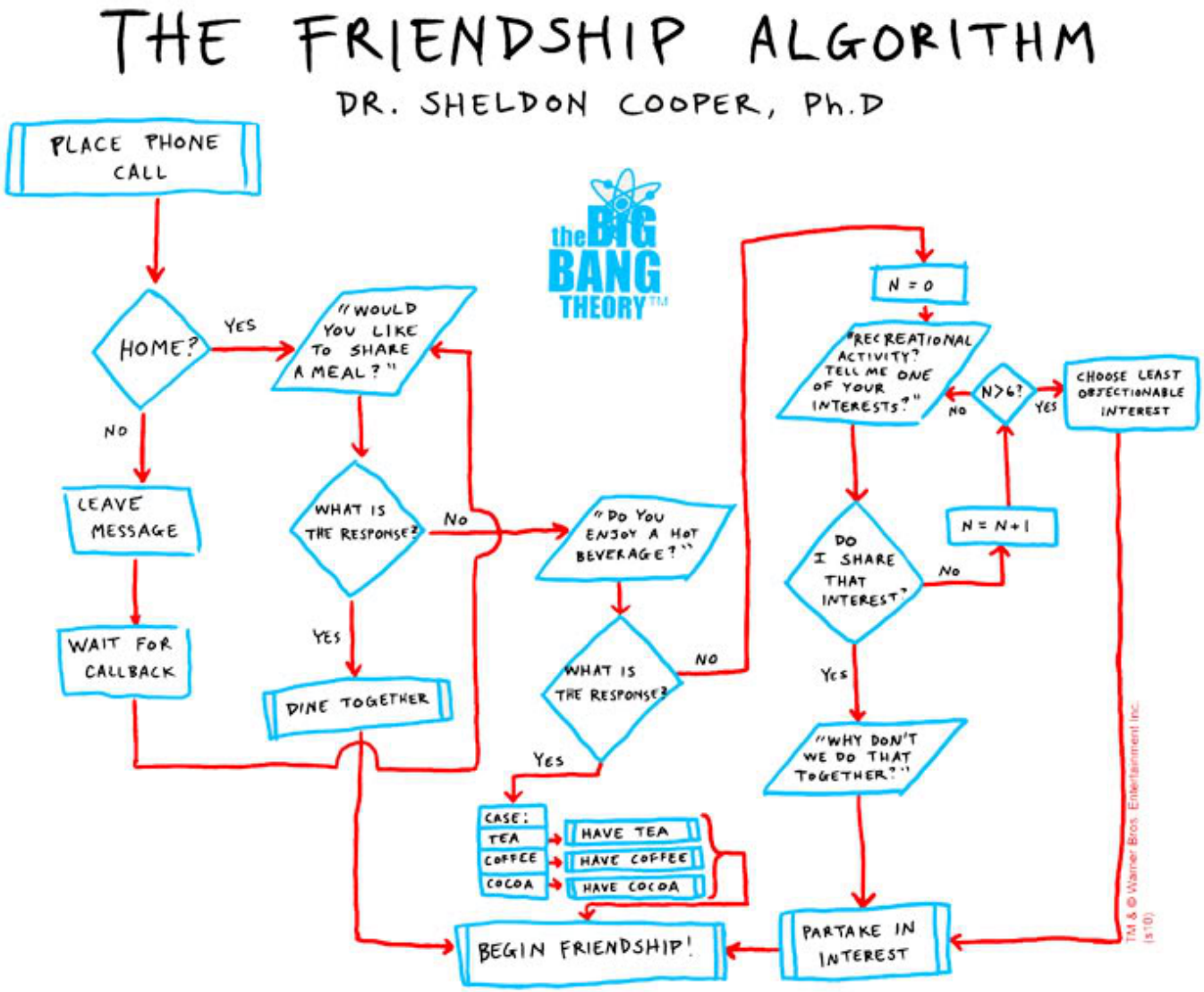
- Factorisation of the `validation` event
- Guards must be mutually exclusive to ensure determinism

Some Tips...

- Every transition must have an event
- In most cases, automata must be deterministic
  - If several transitions from a same state share a same event, then use guards to ensure determinism
- Every state must be reachable from the initial state
- If the modelled system has a finite life, then there must exist a path from every state to a final state

Other UML Diagrams for modelling behaviors

↝ Communication Diagrams, Timing Diagrams, Activity Diagrams, ...



Goals and Artifacts

Goals

- Model the behaviour
- Model the structure

Artifacts / Deliverables

- Sequence diagrams
- State-transition diagrams
- Class, package, and deployment diagrams

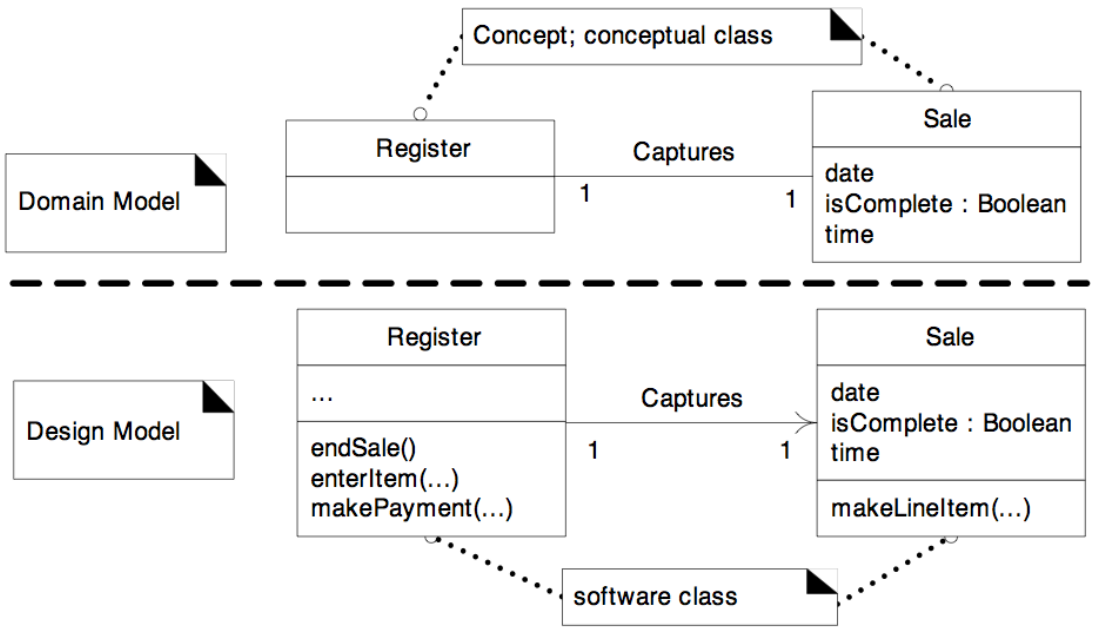
Class Diagrams (Recalls from 3IF)

When capturing requirements:

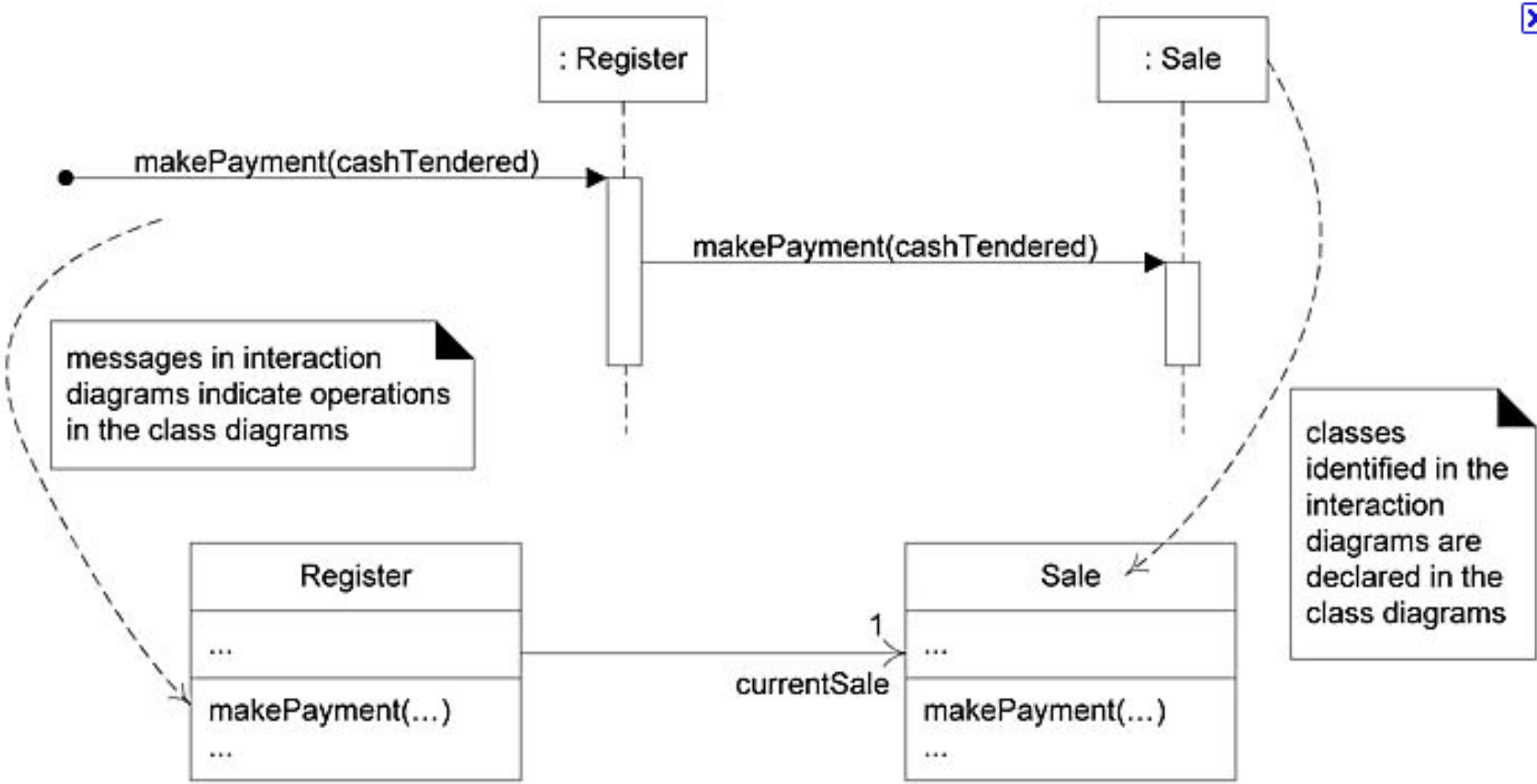
- Classes = real-world (conceptual) classes
- Few attributes, no operations, no visibility

When designing the application:

- Classes = Software Classes
- Add visibility, interfaces, methods, ...



Relation between Sequence and Class Diagrams



[Figure from C. Larman]

Package Diagrams (Recalls from 3IF)

Why designing Package Diagrams?

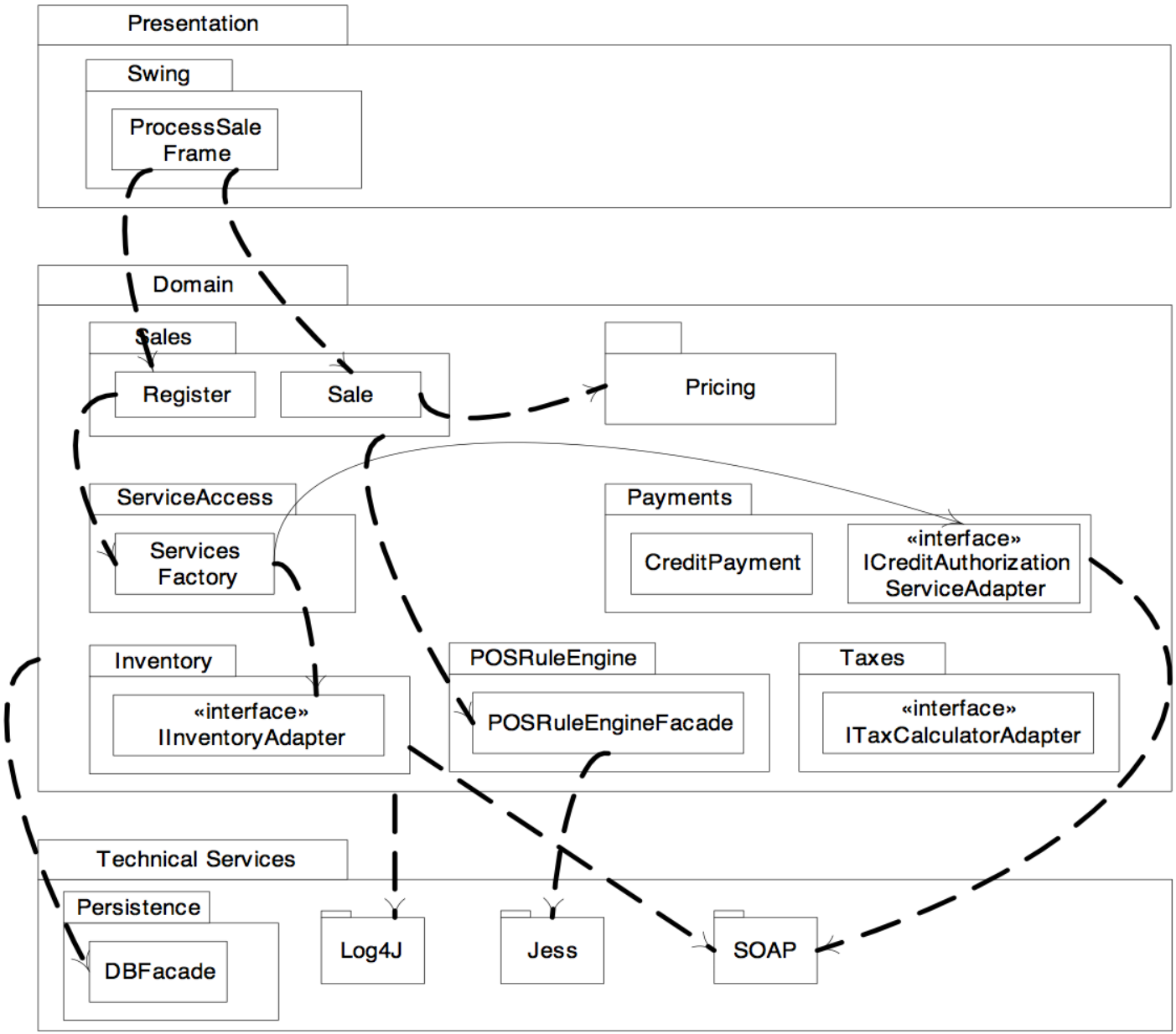
- To group Classes into Packages corresponding to sub-systems
- To model inclusion relations between these groups
- To model dependency relations between these groups

Why structuring a system in sub-systems?

- To encapsulate and decompose complex systems
- To ease collaborative development
- To favour reuse

High Cohesion, Low Coupling and Protected Variations

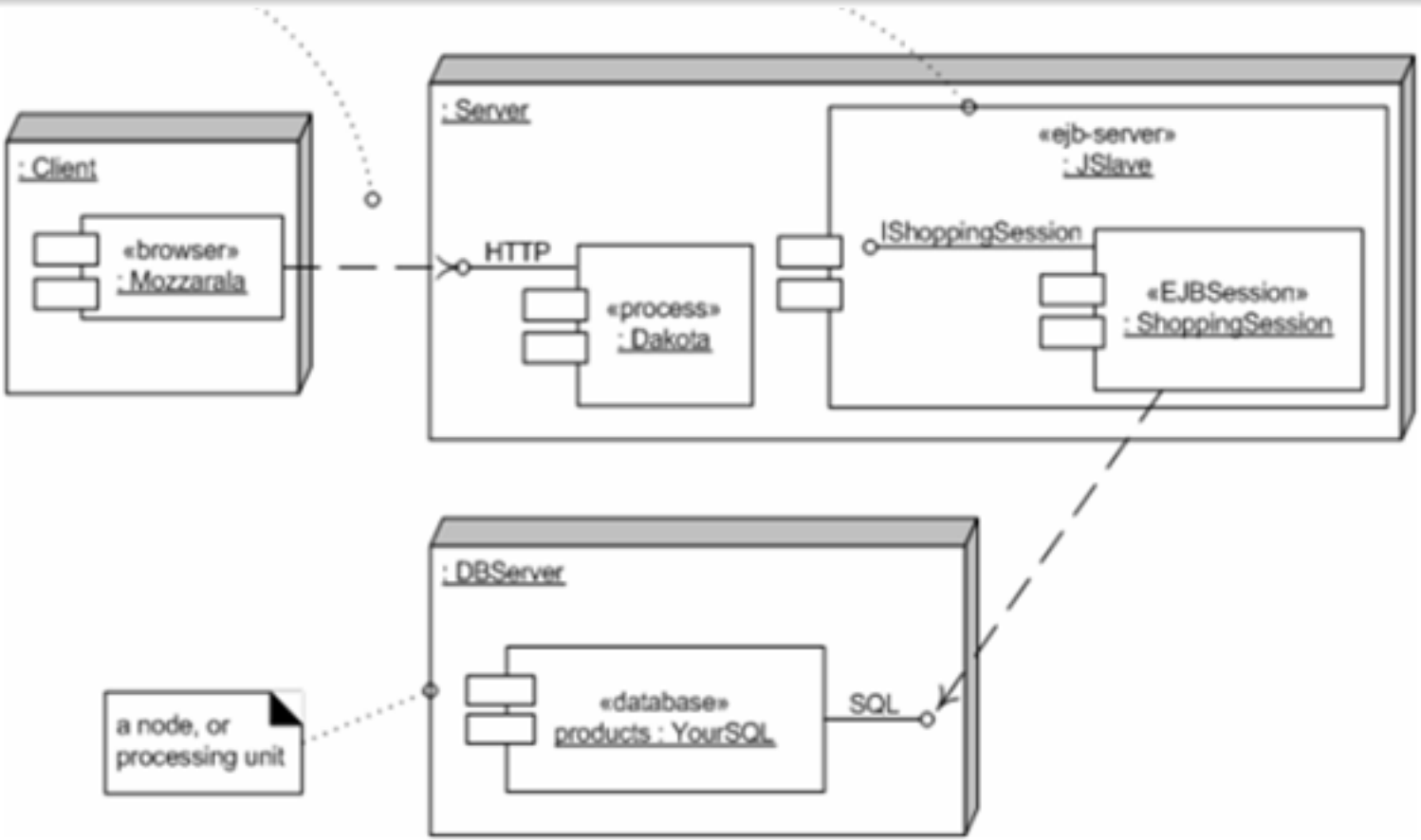
Example of Package Diagram



Deployment Diagram (Recalls from 3IF)

Goal: Describe

- Distribution of software components on hardware components
- Communication between hardware components



From UML to Object Oriented Design

Craig Larman:

Drawing UML diagrams is a reflection of making decisions about the object design. The object design skills are what really matter, rather than knowing how to draw UML diagrams.

Fundamental object design requires knowledge of:

- *Principles of responsibility assignments*
- *Design patterns*

We'll come back to this in Part 2 ...

$\rightsquigarrow$ Illustration with PlaCo

... and in the PLD too!

Overview

1. Introduction
2. Agile and Unified Software Development Process (UP)
3. Description of one iteration
   - Capture and Analysis of the Requirements
   - Design
   - Development and Test
   - Project Management

From Design Models to Code

Goal:

- Write code that implements the targeted use cases
- Test this code to ensure that it has no error and that it actually corresponds to needs

Code skeletons can be automatically generated from design models:

- From Class Diagrams:
  - Declaration of Classes, Attributes, Method Signatures, ...
  - Encode 1-n associations with Collections
  - ...
- From Sequence Diagrams:
  - Sequences of method calls
  - Constructor signatures
  - ...
- ...

Iterative Development and Reverse Engineering

Code skeletons must be completed

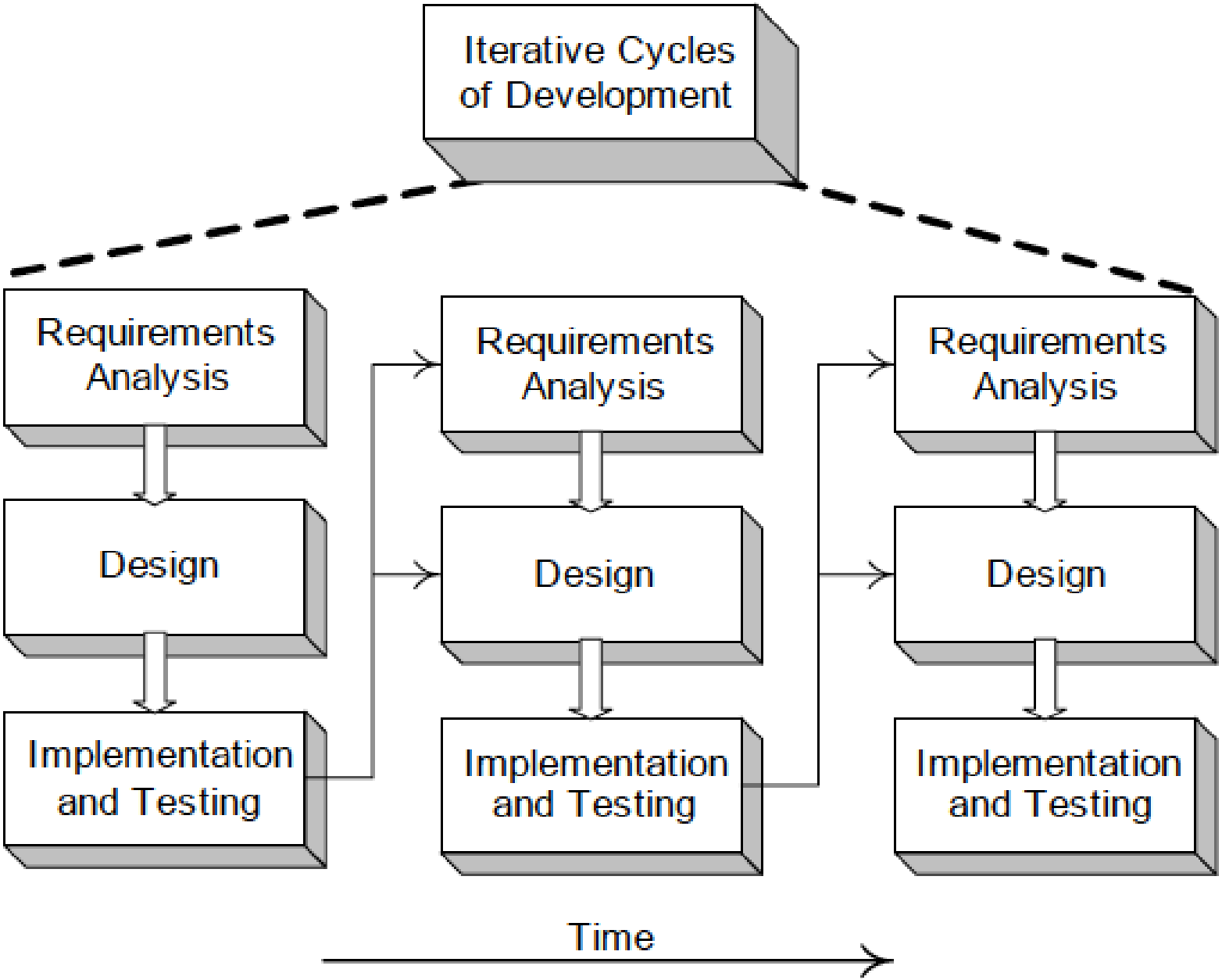
- Implement visibility whenever o_1 must send messages to o_2:
  - Persistent visibility:
    - Attribute visibility: o_2 is attribute of o_1
    - Global visibility: o_2 is a public static attribute or a Singleton instance
  - Temporary visibility, within a method p of o_1:
    - Parameter visibility: o_2 is parameter of p
    - Local Visibility: o_2 is a local variable of p
- Handle exceptions and errors
- ...

In general, automatically generated code must be modified

$\rightsquigarrow$ Add new attributes, methods, classes, ...

Use reverse engineering tools at the end of each iteration

$\rightsquigarrow$ Update design models for the next iteration



[Image from C. Larman]

Test-Driven Development (TDD)

TDD Cycle:

- Write unit tests (before starting implementation)
- While some tests fail do:
  - Complete code

  $\rightsquigarrow$ Most simple implementation with respect to tests
- Refactor, and test again

Advantages:

- Unit tests are actually written
- Nice and challenging way of programming, with a clear goal
- Tests provide an operational specification of method behaviors
- Tools (JUnit, CTest, ...) may be used to automate the test process
- Non regression is automatically checked when refactoring

Refactoring

Goal:

- Transform/restructure code without changing behavior
  $\rightsquigarrow$ Remove “code smells"
- Warning: Run tests after each modification

Examples:

- Suppress code duplication by creating new methods
- Rename variables, methods, ... to improve readability
- Shorten long methods by creating new methods
  $\rightsquigarrow$ Ensure the single responsibility principle
- Replace magic literals (3.14, 9.81, ...) with symbolic constants
- Remove dead code
- ...

cf `http://refactoring.com/catalog`

KISS

Keep It Simple, Stupid

> *Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*
>
> — Martin Fowler

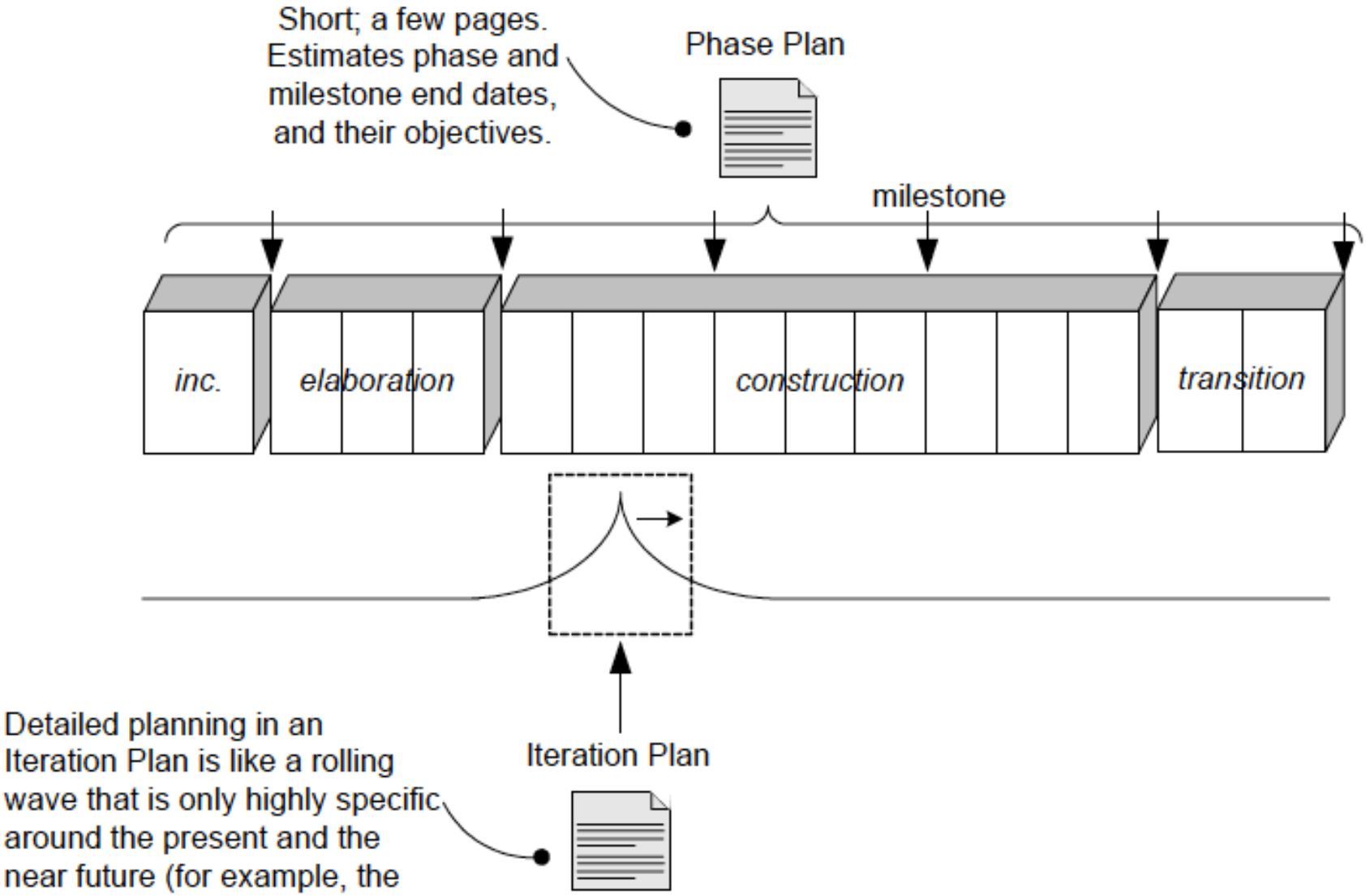
> *Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. Code for readability.*
>
> — John F. Woods

[Slide from Laurent Cottereau]

Overview

1. Introduction
2. Agile and Unified Software Development Process (UP)
3. Description of one iteration
   - Capture and Analysis of the Requirements
   - Design
   - Development and Test
   - Project Management

Phase Planning vs Iteration Planning



[Image from C. Larman]

Iteration Planning

When should we plan an iteration?

On the first iteration day

Who is involved in iteration planning?

Product Owner (PO), Team members, Scrum master

How to plan an iteration?

- The PO identifies and ranks candidate items in the product backlog
- For each item, by order of priority:
  - Quick estimation of the tasks that must be done
  - Quantification of the time wrt available human resources
    $\rightsquigarrow$ Planning poker (http://www.planningpoker.com/)

  Until total time = Iteration duration