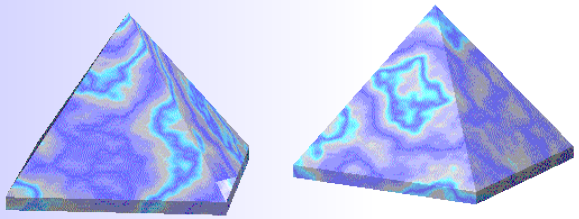
From Use-Cases to OO
(Agile development with UML)

Renaud Florquin
Isabelle Leclercq

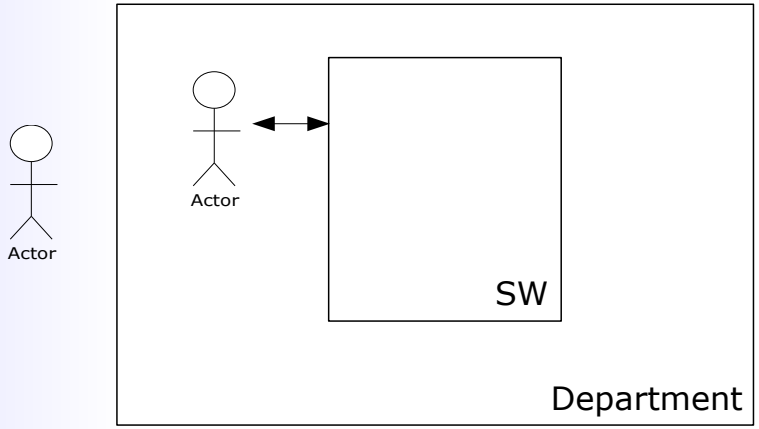


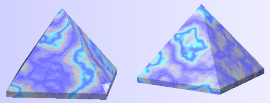
FloConsult SPRL

http://www.floconsult.be
mailto:info@floconsult.be

Identifying the System

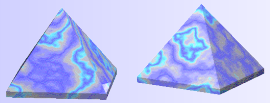
- First of all, define the boundaries
- Different points of view
  - Business (entire organization, department)
  - Software system
- Then the external interfaces => actors





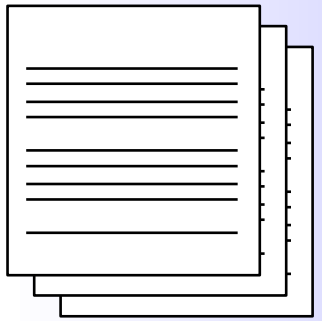
Actors

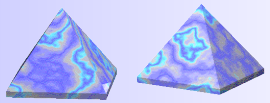
- Actor = external entity acting on the system:
  - a human being
  - a machine
  - other software systems
  - anything that exchanges information with the system
- Different types:
  - Primary (beginning exchanges with the system)
  - Secondary (participating)
  - Passive (only answering)



Use-Case = Text Description

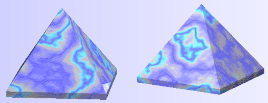
- Describe sequences of actors-system interactions
- Wording of Use Cases is very important
  - text description used to discover object
  - natural language can be very ambiguous
  - one sentence can mean different things to different people





Identifying Use Cases

- Identified through the actors: one UC for every complete course of events initiated by the actor
- Reading the use case from the actors perspective we ask:
  - What are the main tasks for each actor ?
  - Will the actor have to read / write / change any of the system information ?
  - Will the actor have to inform the system about outside changes ?
  - Does the actor wish to be informed about unexpected changes ?

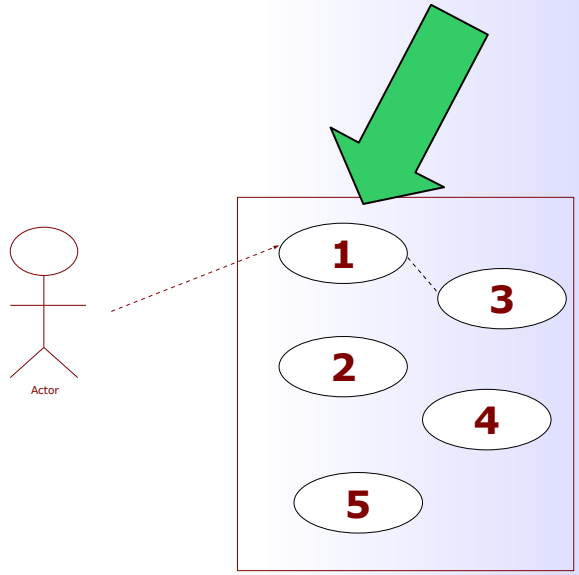


Different UC levels

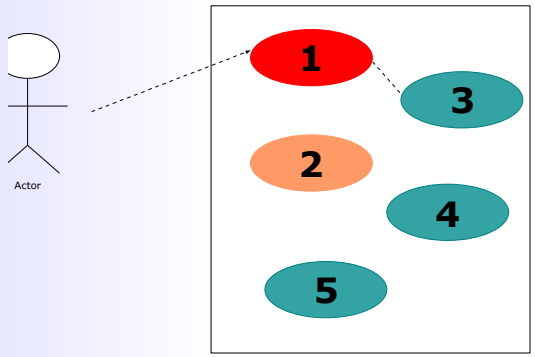
- High-level UC:
  - Brief, general description
  - Some sentences
  - Overview
- Detailed UC:
  - Much more detailed
  - Before detailed analysis

When used ?

- High-level UC typically in Inception:
  - All High-level UC identified
  - With their actors and relationships
- Then, selected UC for implementation are detailed
  - Typically 10-20% in Inception



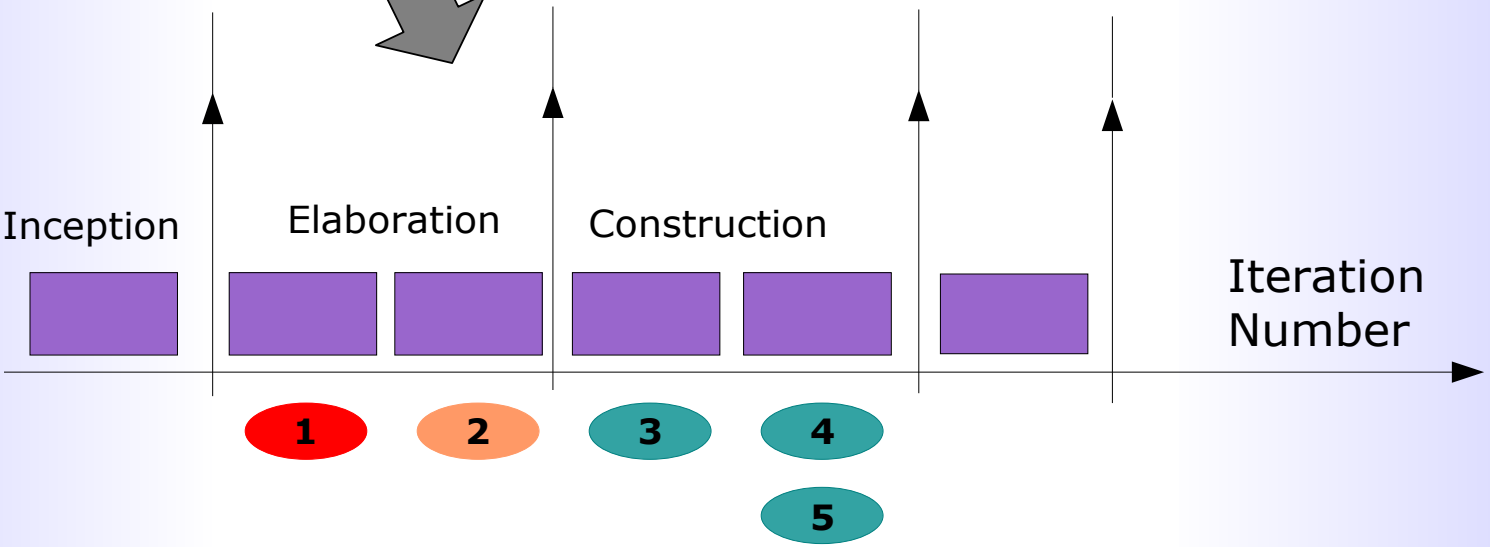
Use-case driven

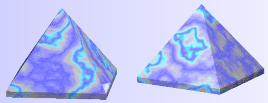


High risk, criticality, and /or wide-architectural coverage

Middle risk, criticality, and /or middle-architectural coverage

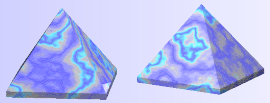
Low risk, criticality, and /or low-architectural coverage





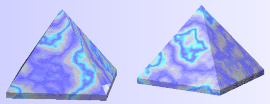
Other requirements (1)

- UC = functional view of the system
- There are other requirements:
  - Usability
  - Reliability
  - Performance
  - Implementation constraints
  - Interfaces
  - Legal issues ....



Other requirements (2)

- Captured in Supplementary Specifications
- In your experience ?
  - Proportion ?
  - Clear ?
  - Annoying ?
  - Underspecified ?
  - Bad ... ?

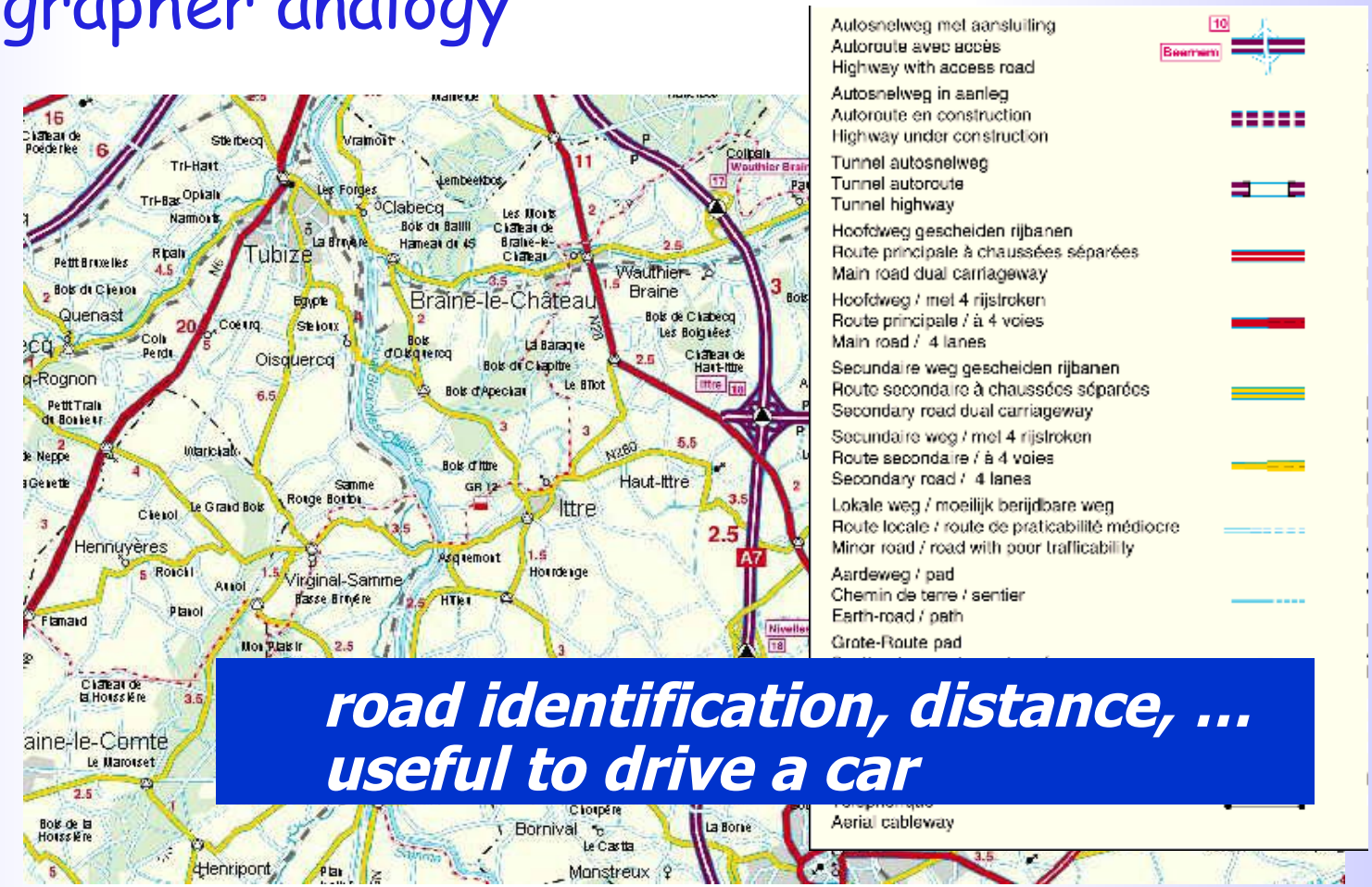


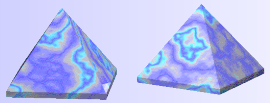
Transition from UC to OO

- Use Cases examined to get the 'first cut' of objects / classes
  - Use Cases scanned quickly to look for nouns
  - Use those nouns for you first go at an object / class diagram
- Many class diagrams produced if they can give alternative / additional views
  - Diagrams give information
  - Diagrams can be discussed and rejected if necessary
  - No Diagrams, No discussion, Bad Communication

Reality abstraction (I)

- Cartographer analogy

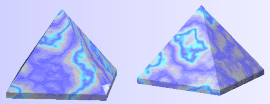




Reality abstraction (II)

- Extracting the relevant features
- Removing irrelevant details
- Do not redefine domain names (even if bad...)

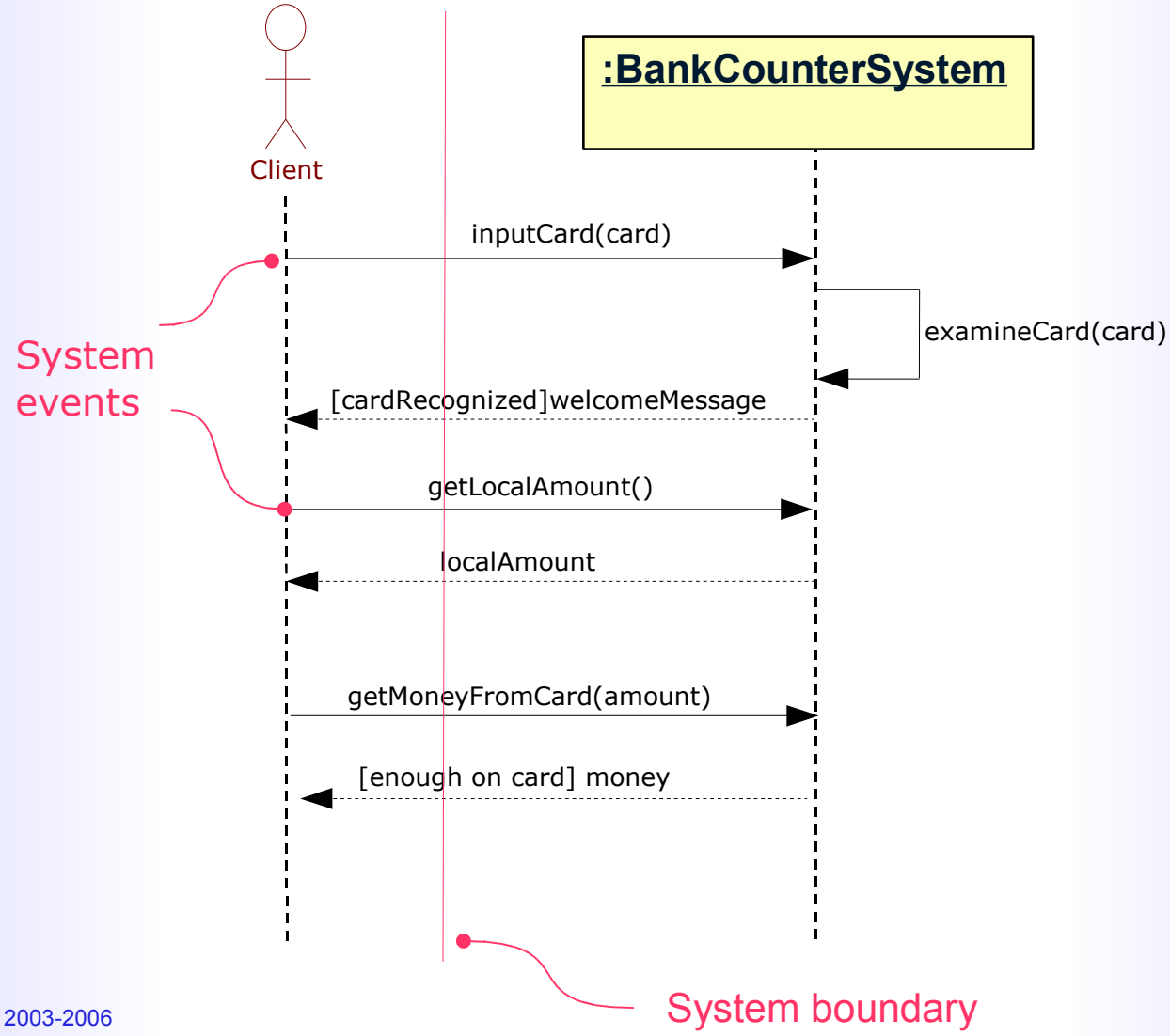
  - ***Depends on the observer point of view***
  - ***Powerful tool for dealing with the complexity***



System Sequence diagram

- Only one object: the System
  - As black box
- Interacting with Actors
- They exchange system events
- Used to visualize scenarios of System usage
  - The nominal one
  - Frequent or complex alternative ones
- "SSD"

SSD Example


:BankCounterSystem
Client
inputCard(card)
examineCard(card)
[cardRecognized]welcomeMessage
getLocalAmount()
localAmount
getMoneyFromCard(amount)
[enough on card] money
System events
System boundary

Relations UC/SSDs

- Direct link, that can be represented on the diagram:

UC1: consult local amount on bank card

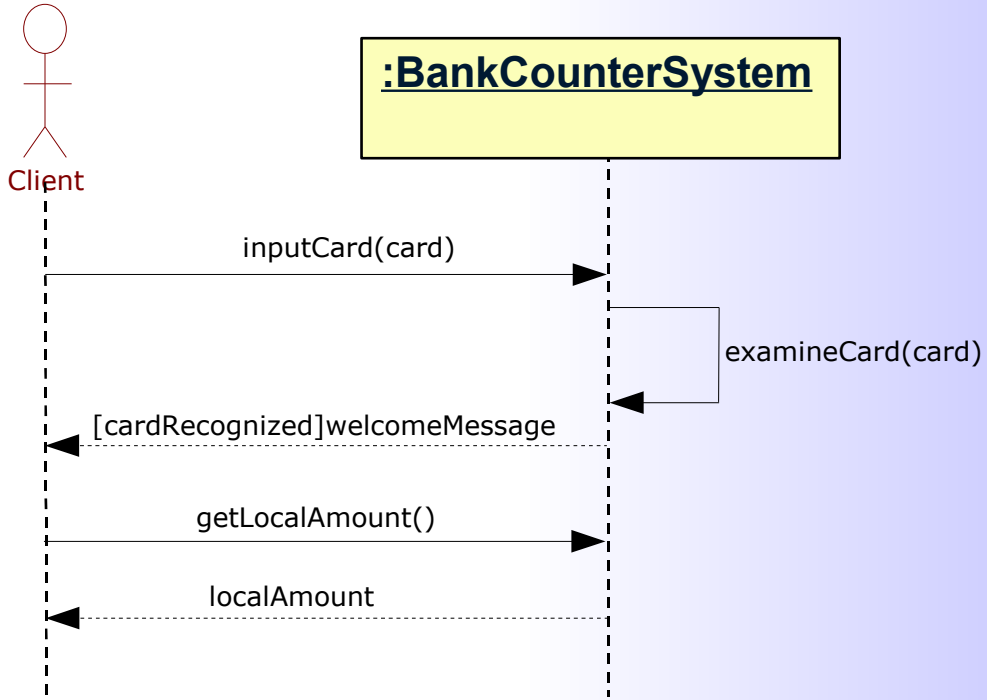
... the client inputs his card in the BankCounterSystem

... the System examines the card data

... and, if the card is authorized, it displays a welcome message

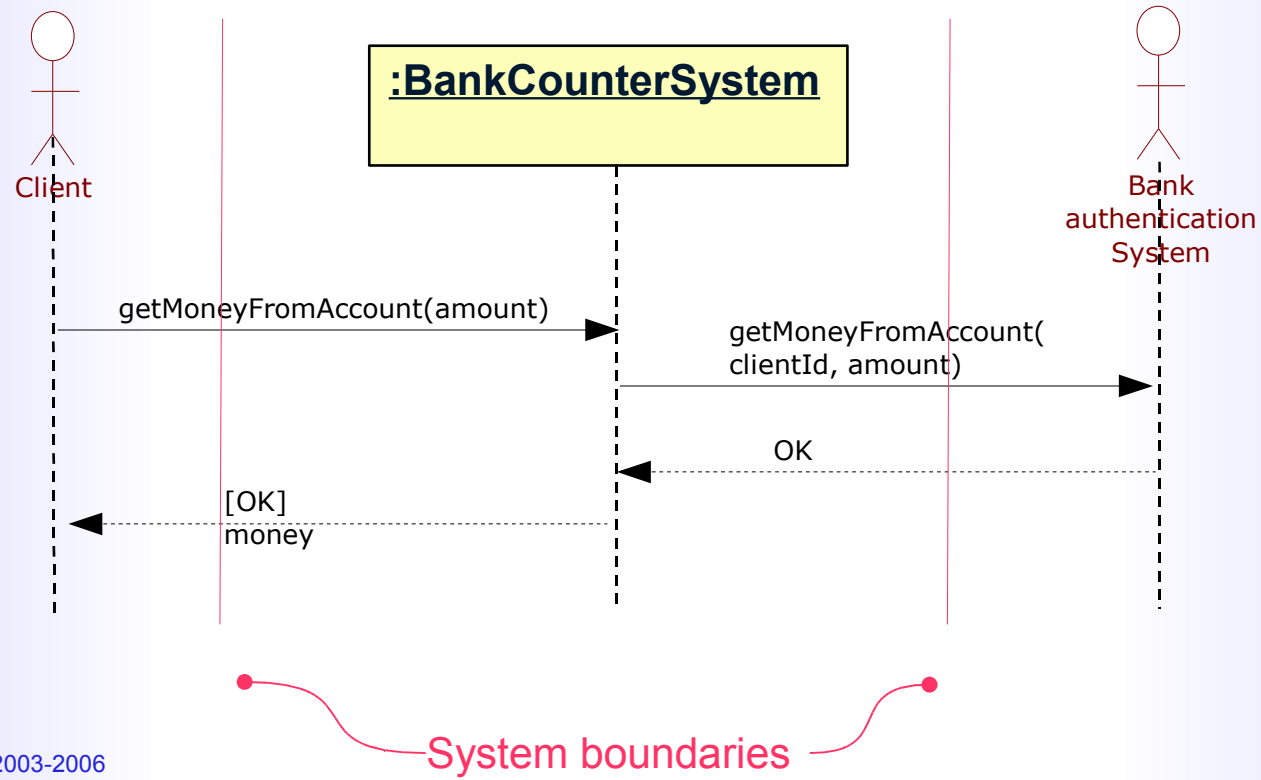
...the client asks for the local amount of money on the card (e.g. Proton balance)

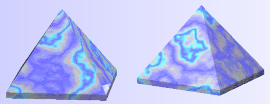
... and the System returns the amount seen on the card



Inter-system SSDs

- SSDs may be used to illustrate collaboration between systems



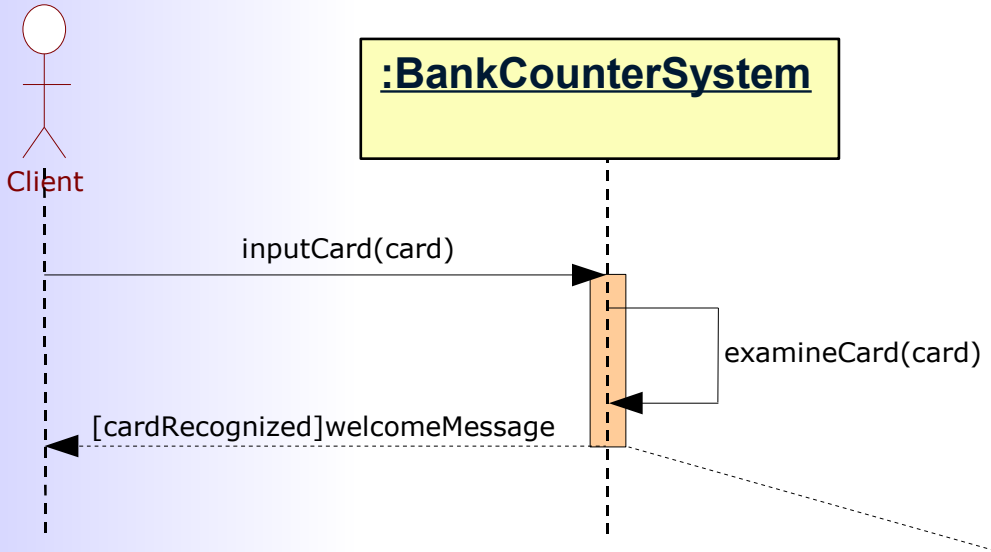


Operation Contracts

- Help to define system behavior
- Based on:
  - Domain model (class diagram)
  - System events
- Describe system behavior in terms of state changes to objects
- Will help to define object's operations
- Optional

Relations OC/SSDs

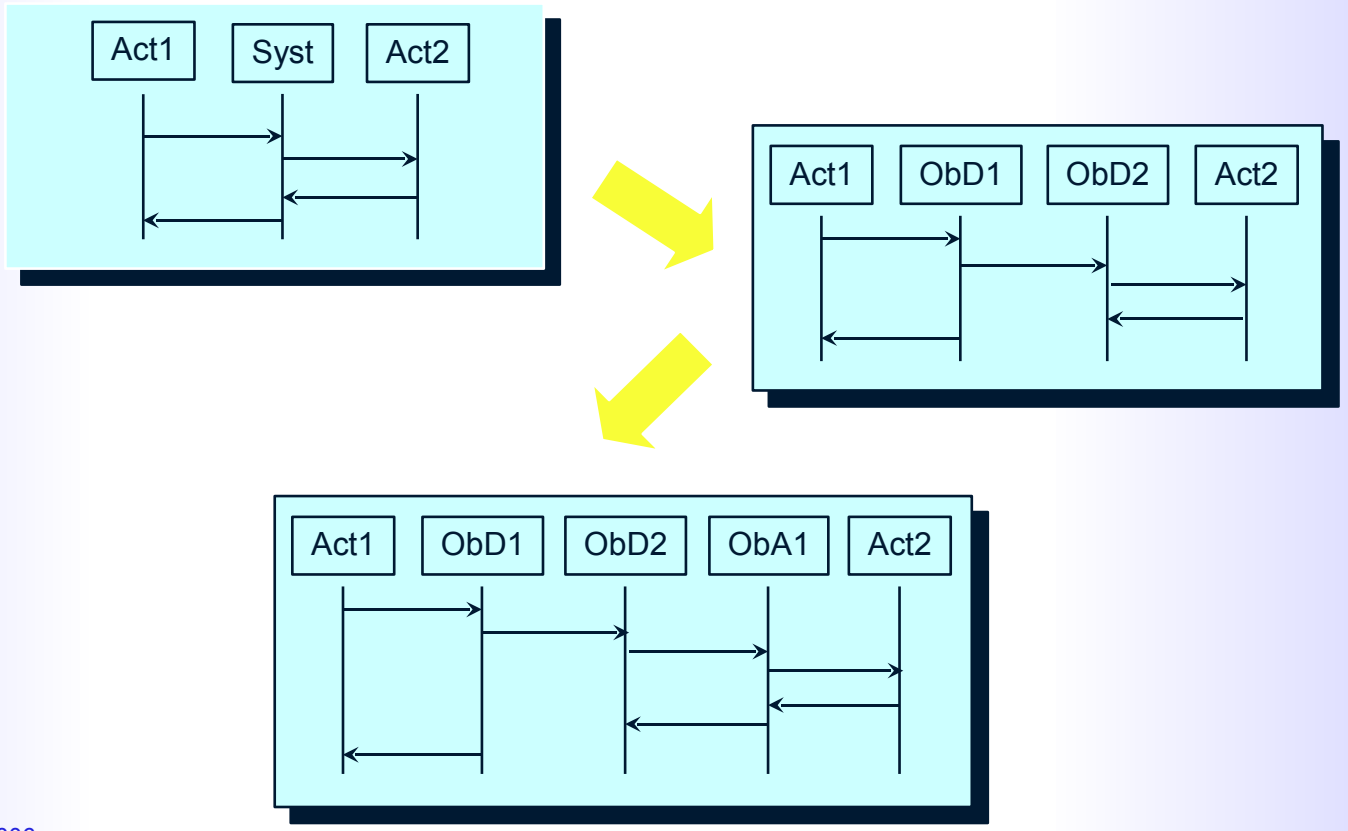
- Direct link, that can be represented on the diagram:

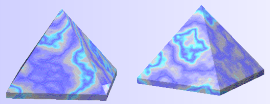


- a Client object has been created in the system (***instance creation***)
- the counter of the number of clients from today has been incremented by one (***attribute modification***)
- a welcome message has been displayed (***attribute modification***)

Formalize the scenarios

- Construct Interaction Diagrams for all the identified scenarios, to a useful level of abstraction:



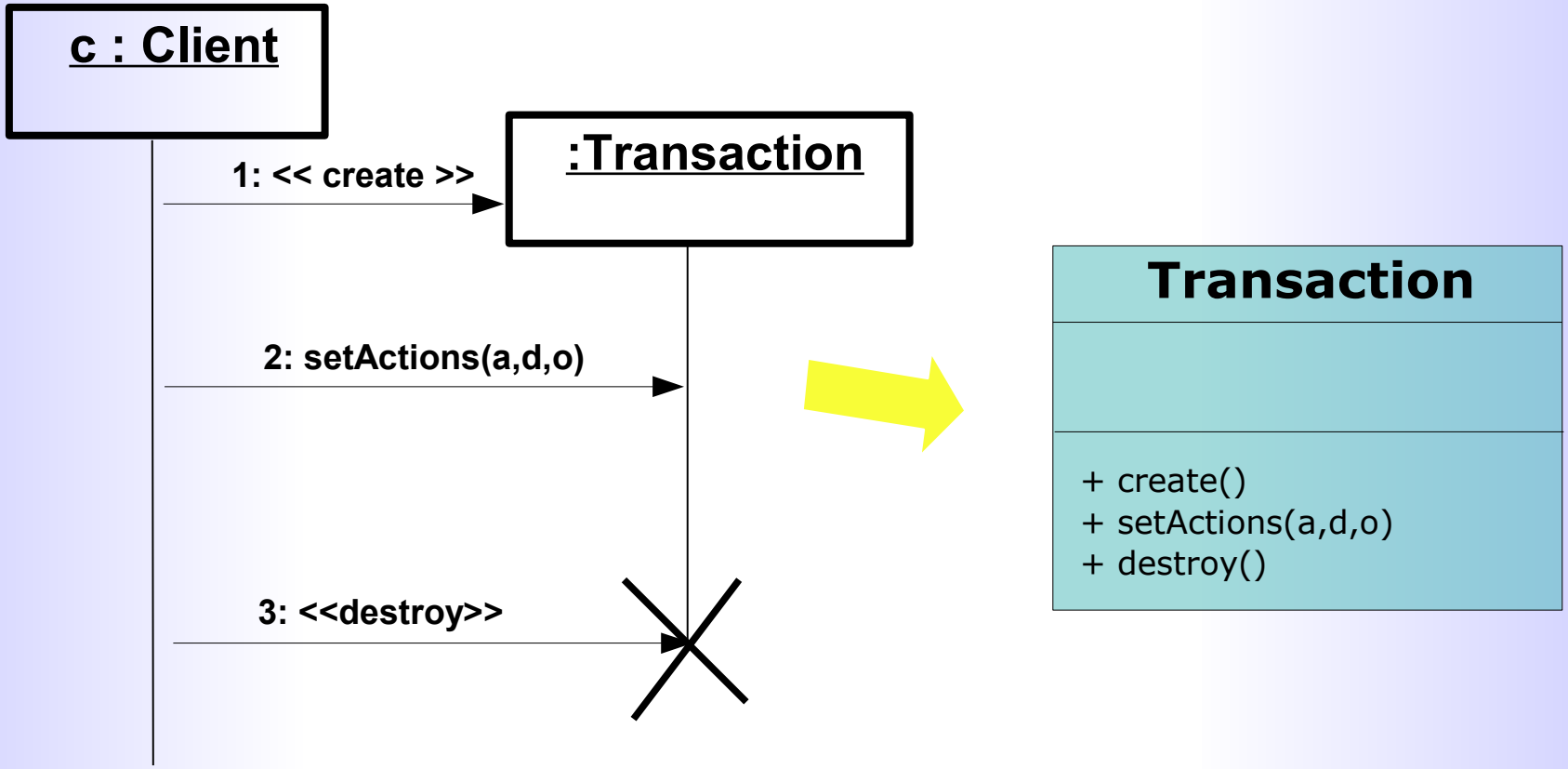


Obtaining a design class diagram

- Systematic way: through scenarios
- A scenario shows a particular sequence of interactions between objects, in one particular context of execution of a system.
- A scenario can be seen as one of the possible instances of one of the Use Cases (from start to finish)
  - Create scenarios for significant system events
  - => operations and FSMs

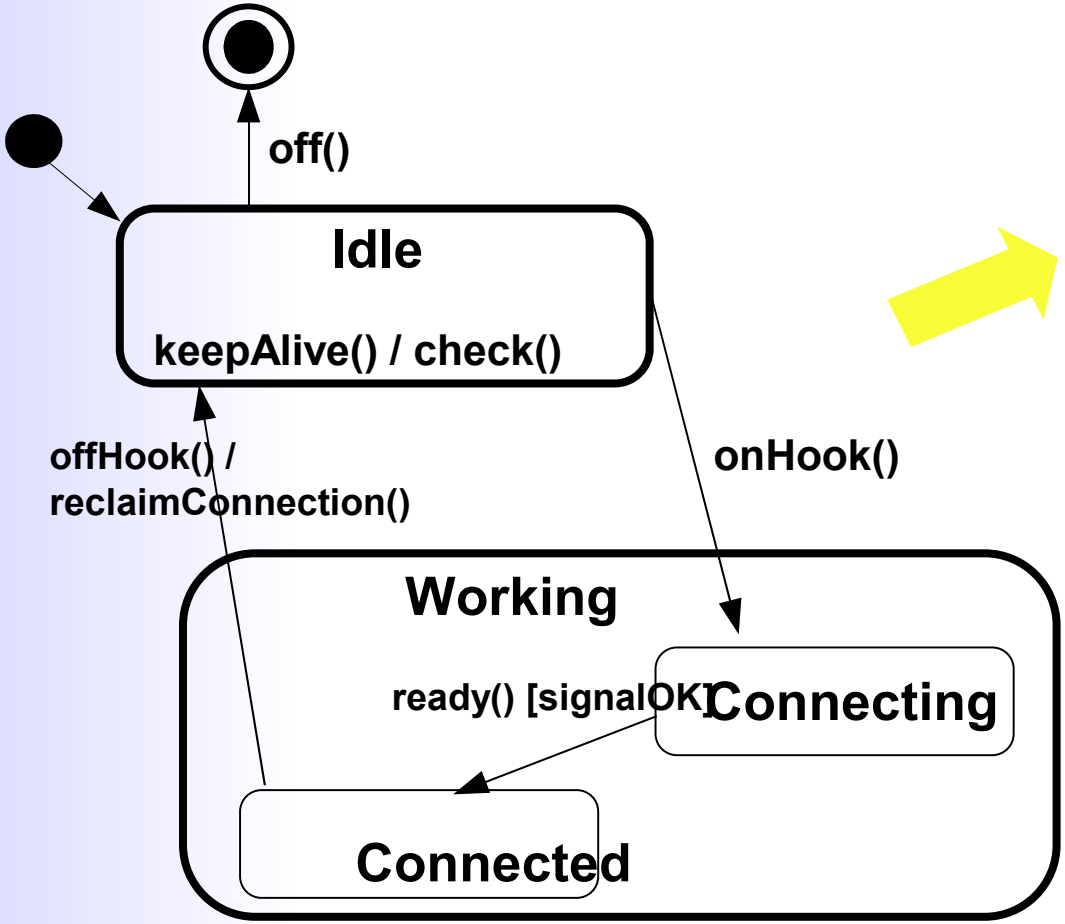
From Inter. diagrams to classes

- Identified events => class operations

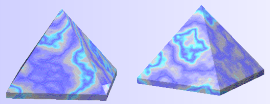


From FSMs to classes

- Identified events => class operations

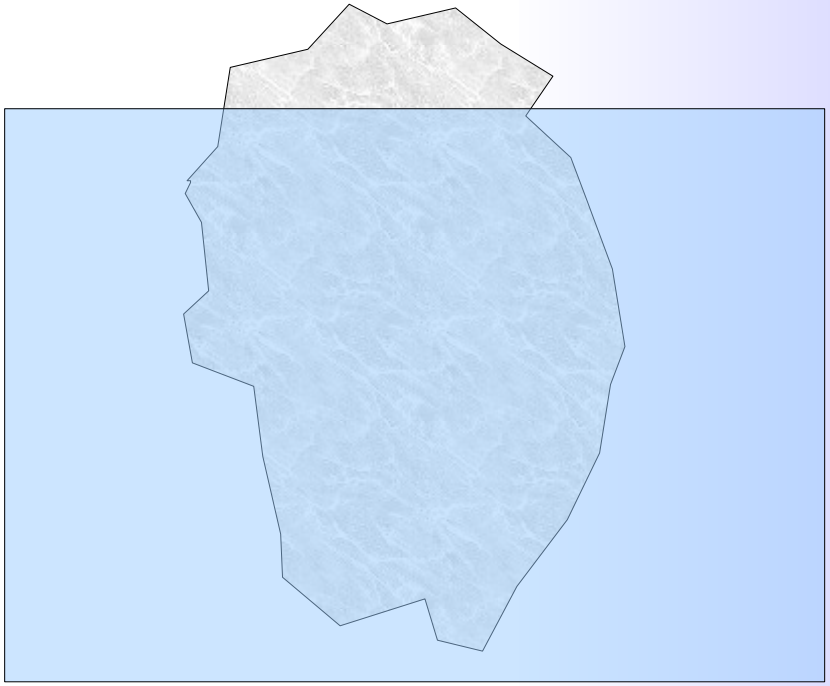


Phone
- state
+ create() + keepAlive() + onHook() + offHook() + ready() - reclaimConnection() - check()



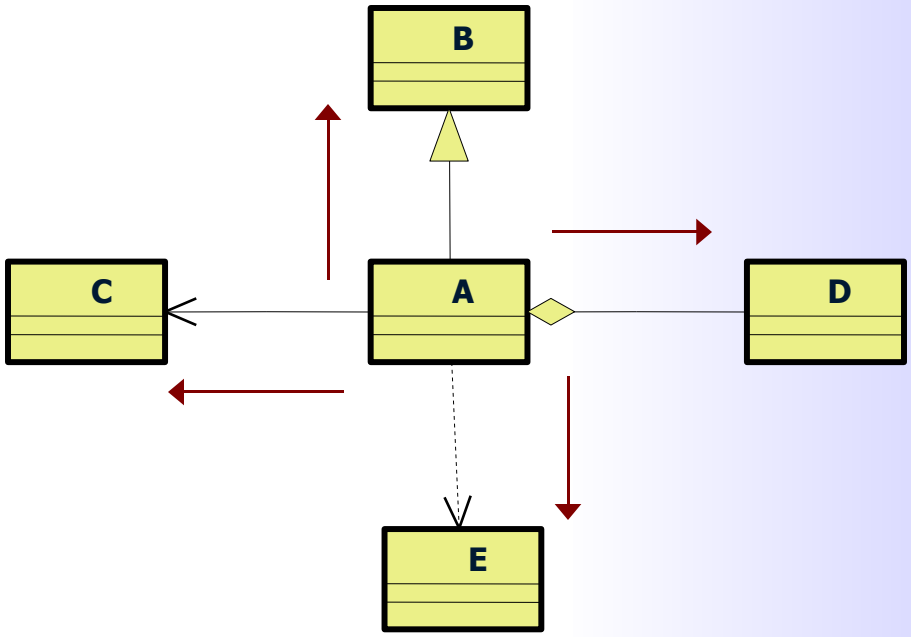
Coupling

- Information hiding
  - Hide implementation decisions
- Module = Iceberg
  - 1/8 visible
  - 7/8 hidden

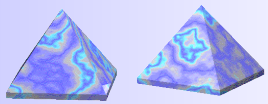


OO Design: Class Coupling

- Class coupling
  - Inheritance
  - Association
  - Aggregation
  - Dependency

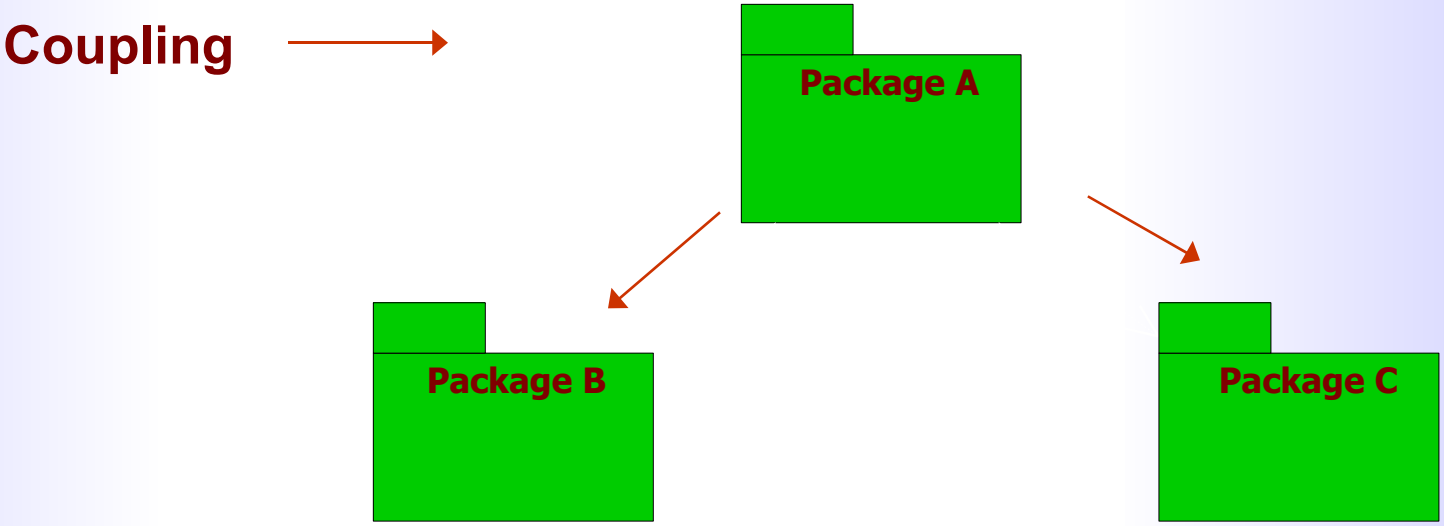


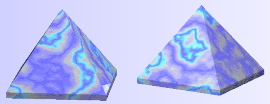
Coupling →



OO Design: Package Coupling

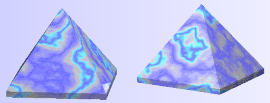
- Package coupling
  - Dependency
  - Inheritance





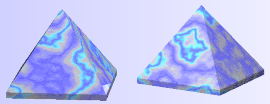
OO Design : Cohesion

- 3 levels of cohesion
  - method : important
  - class : important
    - maximum 2 or 3 responsibilities
    - class types :
      - Creator: creating other objects
      - Controller: facade
      - Expert: "Do it myself"
      - ...
  - package : very important
    - system responsibility



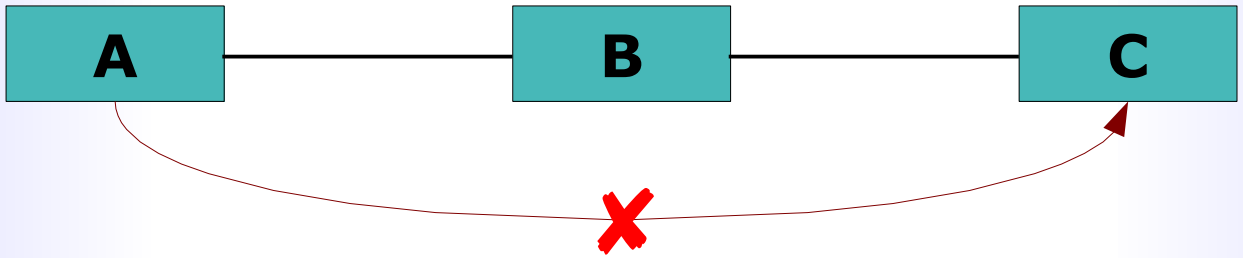
Design By Contract

- Advantages
  - Help consistency capture
  - Define responsibility
  - Communication without misunderstanding
  - Improved precision
  - Better documentation

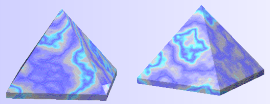


Demeter's law (1)

- "Don't speak to strangers"
- Ex:
  - A knows B
  - B knows C
  - A cannot access C directly, always has to pass by B

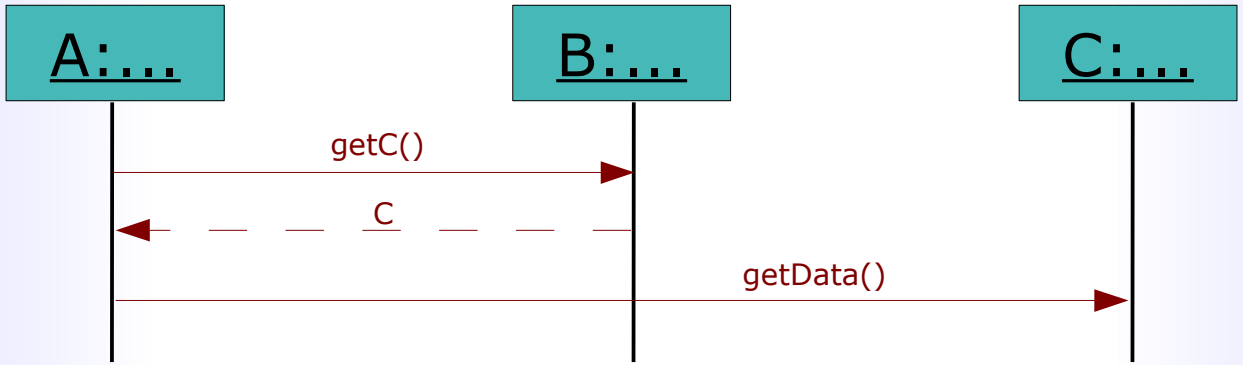


- → to preserve information hiding

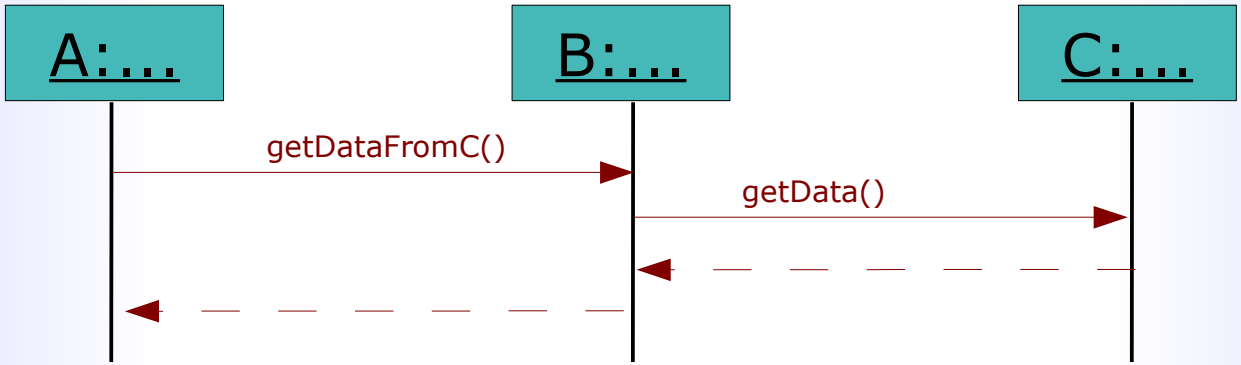


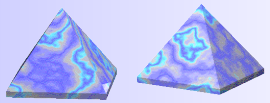
Demeter's law (2)

- Not allowed:



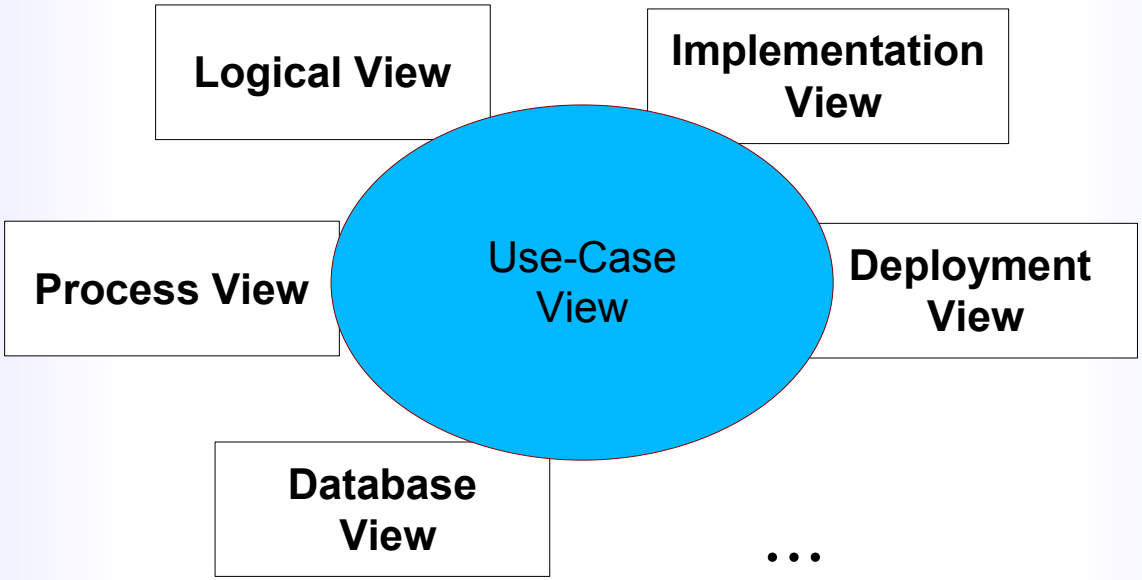
- Allowed:





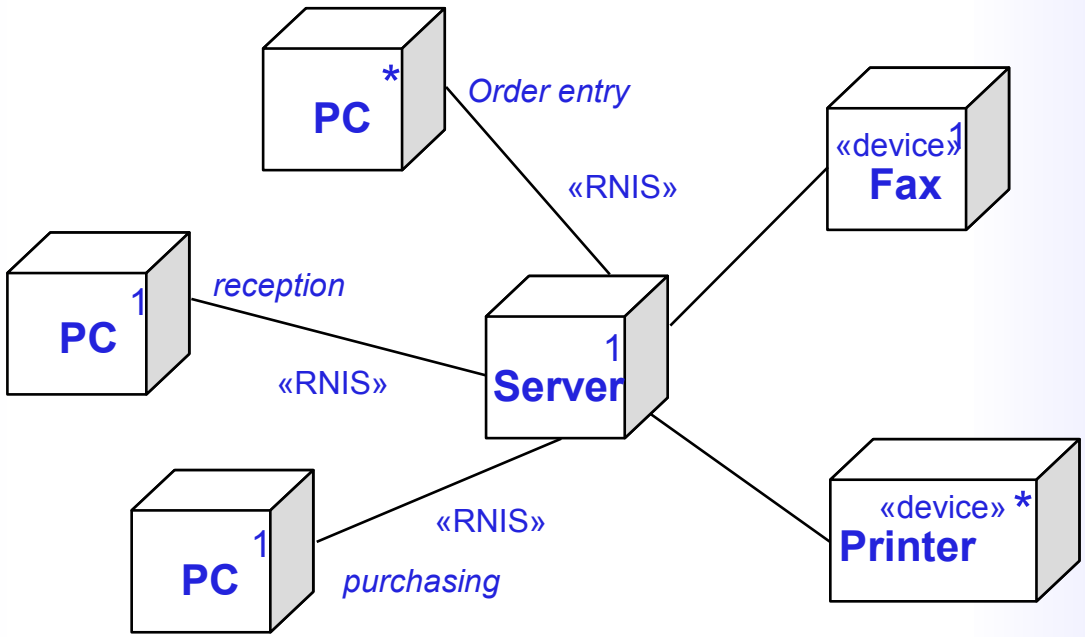
A Model for SW Architecture

- N+1 Views (was 4+1 Views):



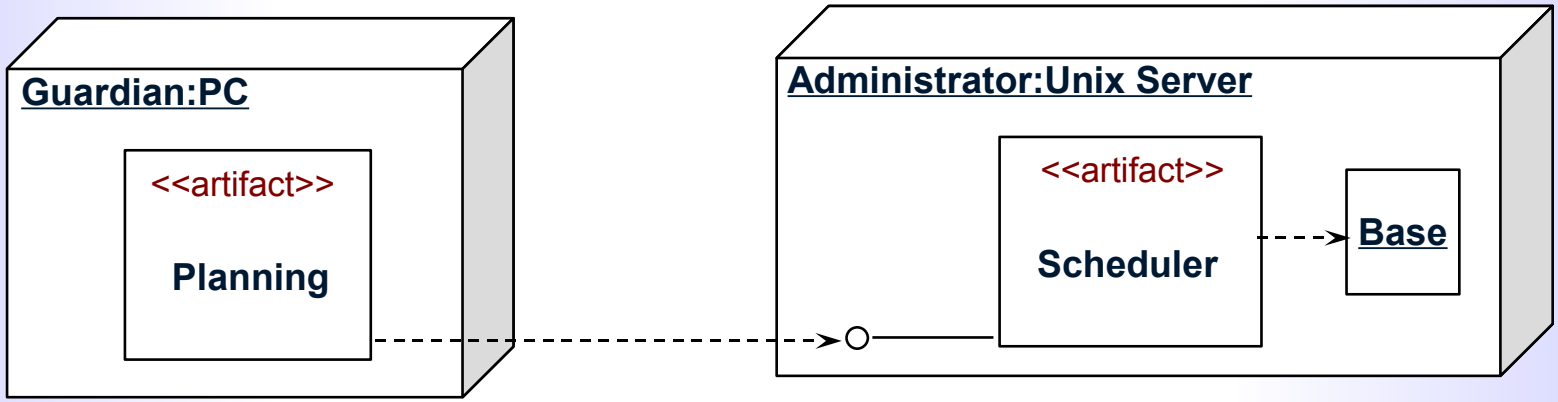
Deployment diagram

- Describes the material configuration of a system:
  - Contains **nodes**: real world resources on which the logic model elements can be distributed and executed

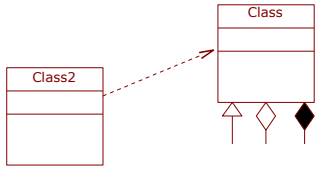
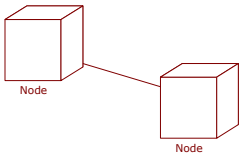
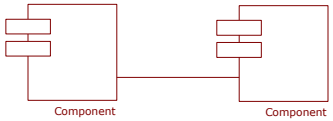


Deployment + Component diagrams

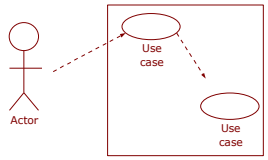
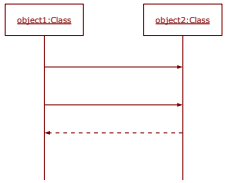
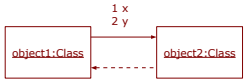
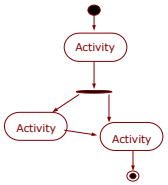
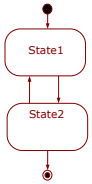
- Allows description of the allocation of components onto nodes :

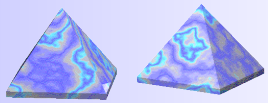


UML Basic diagrams - static

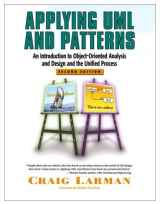
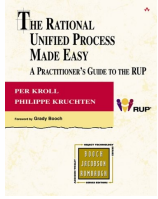
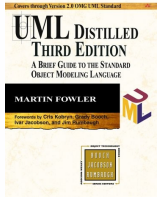
Static Diagram	Notation	Used in ...	OO
Class	 	Inception analysis/design (Domain model) Elaboration analysis/design (Design model)	✔
Deployment		Elaboration analysis/design (N Views)	
Component		Elaboration analysis/design (N Views)	✔

UML Basic diagrams - dynamic

Dynamic Model	Notation	Used in ...	OO
Use Case		Inception requirements (High-level, 20 % detailed) Elaboration requirements	
Sequence		Elaboration requirements (System Sequence Diagrams) Elaboration analysis/design (between objects)	(✔)
Collaboration (Communication)		Elaboration analysis/design (between objects)	✔
Activity		Elaboration requirements (to illustrate Use Cases)	(✔)
State		Elaboration analysis/design (between objects)	(✔)



Bibliography

	**Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development , 3rd Edition** by Craig Larman; 736 p, October 2004, Prentice Hall, ISBN 0-13-148906-2
	**The Rational Unified Process made Easy: A Practitioner's guide to RUP** by Per Kroll, Philippe Kruchten; 464 p, April 2003, Addison-Wesley, ISBN 0-321-16609-4
	**UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3d Edition** by Martin Fowler; 192 p, September 2003, Addison-Wesley, ISBN 0-321-19368-7