

Introduction to software engineering

- The software crisis
- What is software engineering
- Software process lifecycle models
- Unified Software Development process
- Criteria of software quality

Computing in 1968



Apollo Guidance Computer. Culmination of years of work to reduce the size of the Apollo spacecraft computer from the size of seven refrigerators side-by-side to a compact unit weighing only 70 lbs

CICS (Customer Information Control System), an IBM transaction processing system, is released. Before CICS was introduced, many industries used punched card batch processing for high-volume customer transactions.

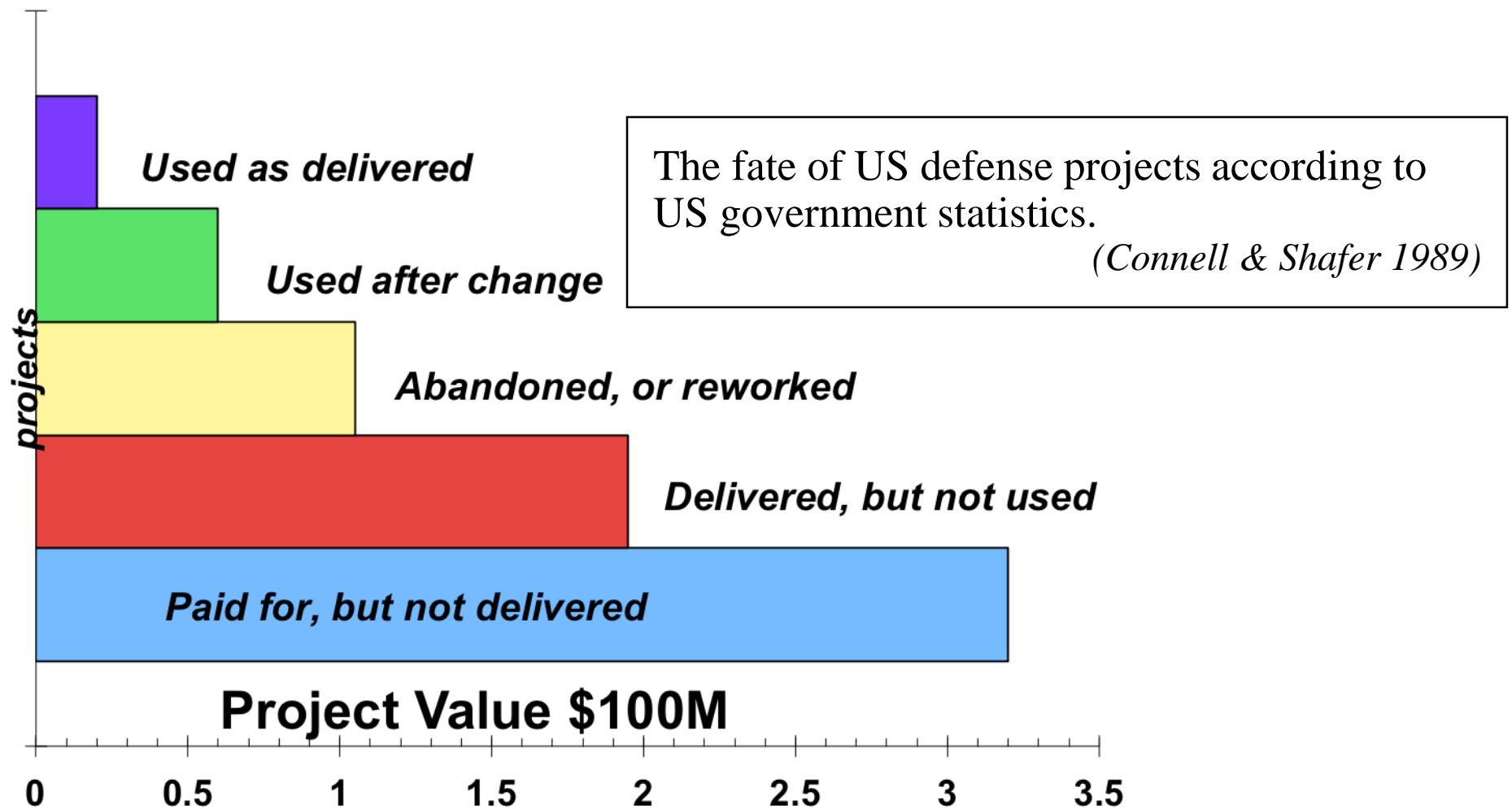
Data General designs the Nova minicomputer. It had 32 KB of memory and sold for \$8,000.

Douglas Engelbart and his team at SRI, with funding from ARPA, unveil their experimental 'OnLine System'. It included collaborative editing, videoconferencing, word processing, and a strange pointing device jokingly referred to as a "mouse."

Software crisis 1968

- ▶ ‘Software engineering’ : “*The application of a systematic, disciplined, quantifiable approach to the development ...and maintenance of software.*” [IEEE]
- ▶ NATO conference [Naur & Randell 69]
 - ▶ “The computer industry has a great deal of trouble in producing large and complex software systems.”
 - ▶ **Problems:** Late deployment, budget overflows, unreliable and unsatisfactory systems, systems never delivered

Software Crisis 1989



Connell, John L. and Shafer, Linda, Structured Rapid Prototyping, Prentice-Hall, Inc., 1989.

Software Crisis 2005

- ▶ “...software debacles are routine. And the more ambitious the project, the higher the odds of disappointment.” [Carr 05]
- ▶ Only 34% of projects: timely & within budget.
- ▶ Some case studies:
 - ▶ FBI
 - ▶ Since 2001: a database on suspected terrorists
 - ▶ Jan. 2005: \$M170, “not even close to having a working system”
 - ▶ Ford Motors
 - ▶ Since 2000, project “Everest”: buying supplies, replacing legacy
 - ▶ Aug. 2004: \$M200 over budget, abandoned (β -version slower than legacy)
 - ▶ McDonalds
 - ▶ Since 1999, project “Innovate”, budget \$M1,000
 - ▶ Killed in 2002, writing off \$M170

Software Crisis 2008

- ▶ Ness settles arbitration case [[Globes 29-Jan-2008](#)]
 - ▶ Ness Technologies (Nasdaq: NSTC) has signed a settlement with Harel Insurance Investments (TASE: HARL), one of its clients.
 - ▶ The dispute developed in late 2006 when Harel [claimed] that Ness had breached the contract by not delivering the required software on time.
 - ▶ Ness challenged the claim, noting that Harel had dramatically increased the scope of the project since its start.
 - ▶ The dispute went to mandatory arbitration in January 2007, under the terms of the contract. Harel sought reimbursement for what it had paid as well as damages, totalling approximately \$25 million.

<http://www.globes.co.il/serveen/globes/docview.asp?did=1000303419&fid=1725>

Software Crisis 2015

- ▶ The average success rate of software projects was **30.3%**, whereas **46%** of projects were challenged and **23.4%** of projects failed
- ▶ The average project time overrun rate was **76%**, project cost overrun was **52.5%** and the project feature delivery rate was **68.4%**
- ▶ A KPMG Survey, found that on average, about **70 %** of all IT-related projects fail to meet their objectives.
- ▶ Poorly defined applications (miscommunication between business and IT) contribute to a **66%** project failure rate, costing U.S. businesses at least **\$30 billion** every year.
 - ▶ 60% – 80% of project failures can be attributed directly to poor requirements gathering, analysis, and management.
 - ▶ 50% are rolled back out of production
 - ▶ 40% of problems are found by end users
 - ▶ 25% – 40% of all spending on projects is wasted as a result of re-work.
 - ▶ Up to 80% of budgets are consumed fixing self-inflicted problems

▶ 7 From: 'Identifying the Reasons for Software Project Failure and Some of their Proposed Remedial through BRIDGE Process Models'. January 2015. INTERNATIONAL JOURNAL OF COMPUTER SCIENCES AND ENGINEERING 3(1):118-126, Ardhendu Mandal

Problems in Focus

► **Problem 1:** Unreliable products

- ▶ Systems crash, halt, or demonstrate unexpected behaviour

Software and Cathedrals are much the same: First we build them then we pray

--Sam Redwine, Jr.

► **Problem 2:** Maintenance is expensive

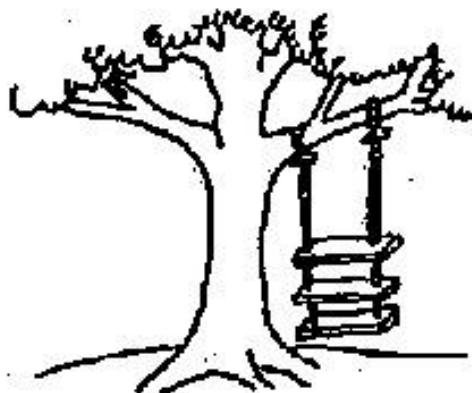
- ▶ In reality, more than 90% (!) of the resources dedicated to software
- ▶ Maintenance costs exceed initial estimate

Samuel T. Redwine, Jr. - quote made at 4th International Software Process Workshop and published in Proceedings of the 4th International Software Process Workshop, Moretonhampstead, U.K., 11-13 May 1988.

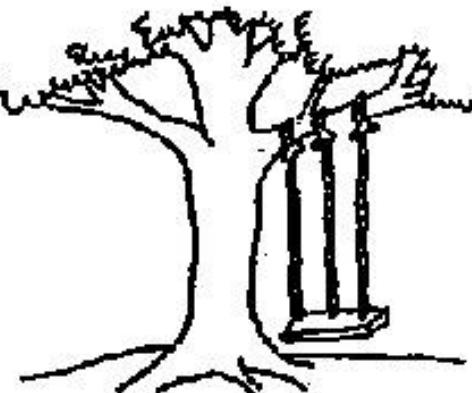


Problems in Focus (Cont.)

► Problem 3: Deployed systems are “Incorrect”



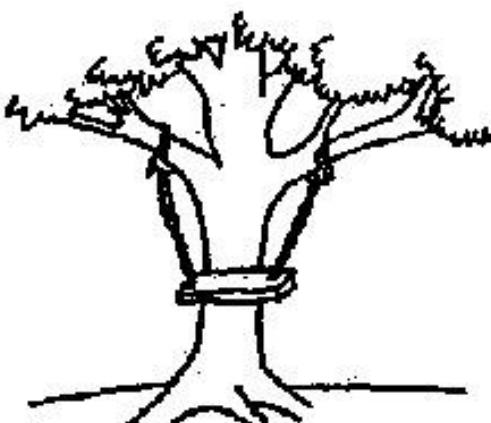
As proposed by the project sponsor.



As specified in the project request.



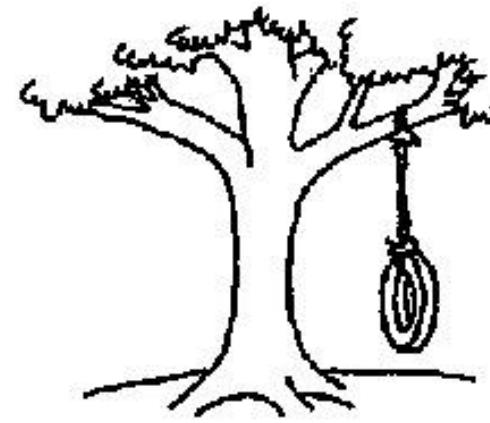
As designed by the senior analyst.



9 As produced by the programmers.

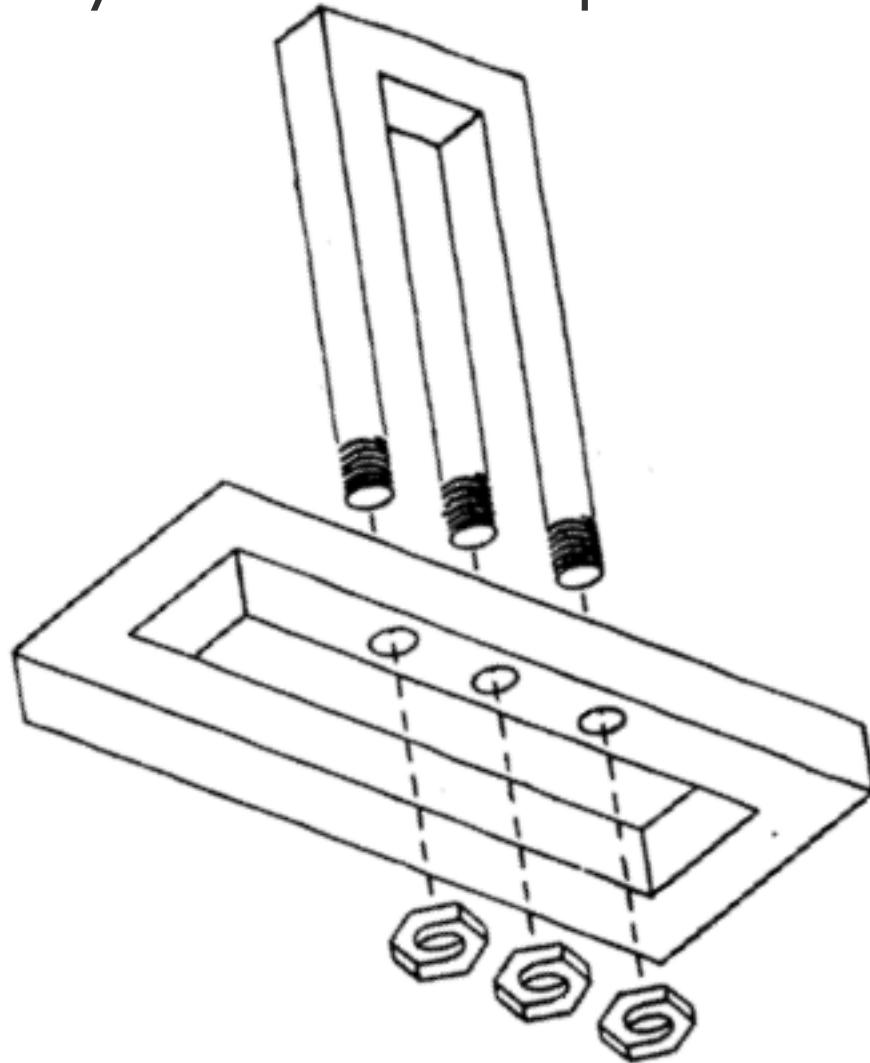


As installed at the user's site.



What the user wanted.

Can you develop this system?



Explanations to the (endless) crisis

- ▶ Software is very different from other manufactured (engineered) products
 - ▶ It is unlike cars, televisions, clothes, bridges, computers, ...
- ▶ Underlying difficulties [Brooks 1987]:
 - ▶ Complexity
 - ▶ Software entities are more complex for their size than perhaps any other human construct
 - ▶ no two parts are alike
 - ▶ Changeability
 - ▶ Moore's law
 - ▶ Operational environment in constant flux: hardware, operating system, communication protocols, technologies, components, ...
 - ▶ Invisibility
 - ▶ Bits are intangible & invisible

Brooks, Frederick P. (1987). *No Silver Bullet: Essence and Accidents of Software Engineering*.
(Reprinted in the 1995 edition of *The Mythical Man-Month*)

What is software engineering?

- ▶ The application of engineering to software
- ▶ Branch of computer science dealing with software systems that are—
 - ▶ large and complex
 - ▶ built by teams
 - ▶ exist in many versions
 - ▶ last many years
 - ▶ undergo changes

Software engineering is a race between programmers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.

-- Rich Cook, paraphrasing Einstein

Rich Cook – light fantasy author, USA, ‘Wizardry’ series of books

What is software engineering (cont.)

- ▶ Techniques (methods):
 - ▶ Formal procedures for producing results using some well-defined notation
 - ▶ Examples: functional/object decomposition, formal specification, interviews, creating class diagrams, prototyping, ...
- ▶ Methodologies:
 - ▶ Collection of techniques applied across software development and unified by a philosophical approach
- ▶ Tools:
 - ▶ Instrument or automated systems to accomplish a technique
 - ▶ Examples: compiler (Sun Java), integrated development environment (Eclipse), configuration management (CVS servers), ...
- ▶ Models
 - ▶ Class Diagrams, Interaction Diagrams, State machines, ...

Software Engineering: A Working Definition

Software Engineering is a collection of **techniques**, **methodologies** and **tools** that help with the production of:

*A **high quality** software system developed within a given **budget** before a given **deadline** while **change** occurs*

Challenge: Dealing with complexity and change



Software Engineering: A Problem Solving Activity

► **Analysis:**

- ▶ Understand the nature of the problem and break the problem into pieces

► **Synthesis/Design:**

- ▶ Put the pieces together into a large structure

For problem solving we use techniques, methodologies and tools.

We will look at the difference between analysis and design during the course



Types of Software

- ▶ Custom (“bespoke”) software
 - ▶ one particular customer
 - ▶ developed “in-house” or outsourced
 - ▶ example: software developed by the IT division of a bank
- ▶ ‘Commercial Off-The-Shelf’ (COTS) or ‘shrink-wrapped’
 - ▶ sold on the open market
 - ▶ can be cheaper and more reliable than custom software but might not fit the needs exactly
 - ▶ examples: email clients, image processors, word processors, ...
- ▶ Embedded software
 - ▶ runs on specific hardware devices
 - ▶ tied to specific hardware
 - ▶ examples: DVD player, microwave ovens, airplanes, weapons, ...



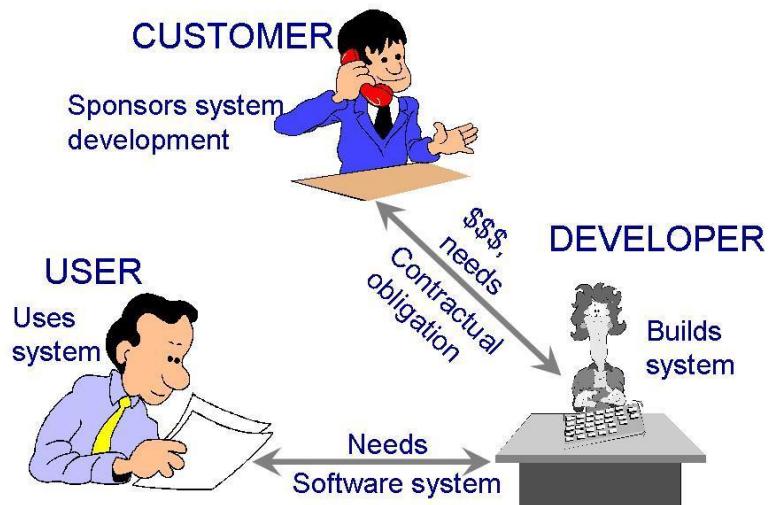
Characteristics of Systems

- ▶ Every system has:
 - ▶ Inputs and outputs
 - ▶ A purpose (related to transformation)
 - ▶ A boundary and an environment
 - ▶ Subsystems and interfaces
 - ▶ Control using feedback and feed-forward
 - ▶ Some emergent property
 - ▶ Example given by Ed Yourdon was the analyst who turned to a colleague and said “you are certainly more than the sum of your parts – you are an idiot!” The point being that if you look at the parts that make up a person, it would be difficult to find idiocy at any level of analysis other than a holistic view of the whole person.



Stakeholders in software project teams

- ▶ **Requirement analysts:** work with the customers to identify and document the requirements
- ▶ **Designers:** generate a system-level description of what the system is supposed to do
- ▶ **Programmers:** write lines of code to implement the design
- ▶ **Testers:** catch faults
- ▶ **Trainers:** show users how to use the system
- ▶ **Maintenance team:** fix faults that show up later
- ▶ **Librarians:** prepare and store documents such as software requirements
- ▶ **Configuration management team:** maintain correspondence among various artefacts

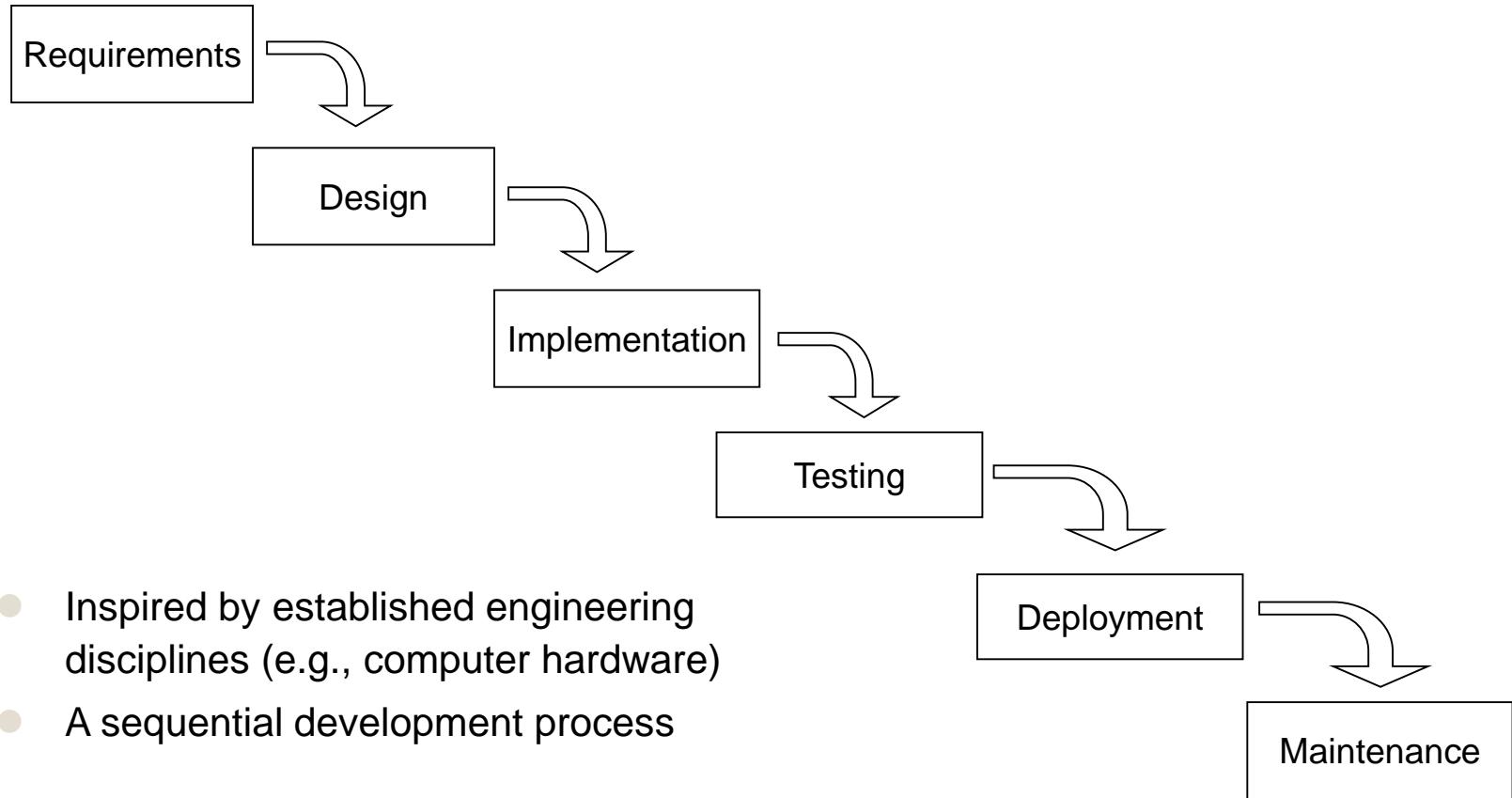


We will look at these different roles as we progress through the course

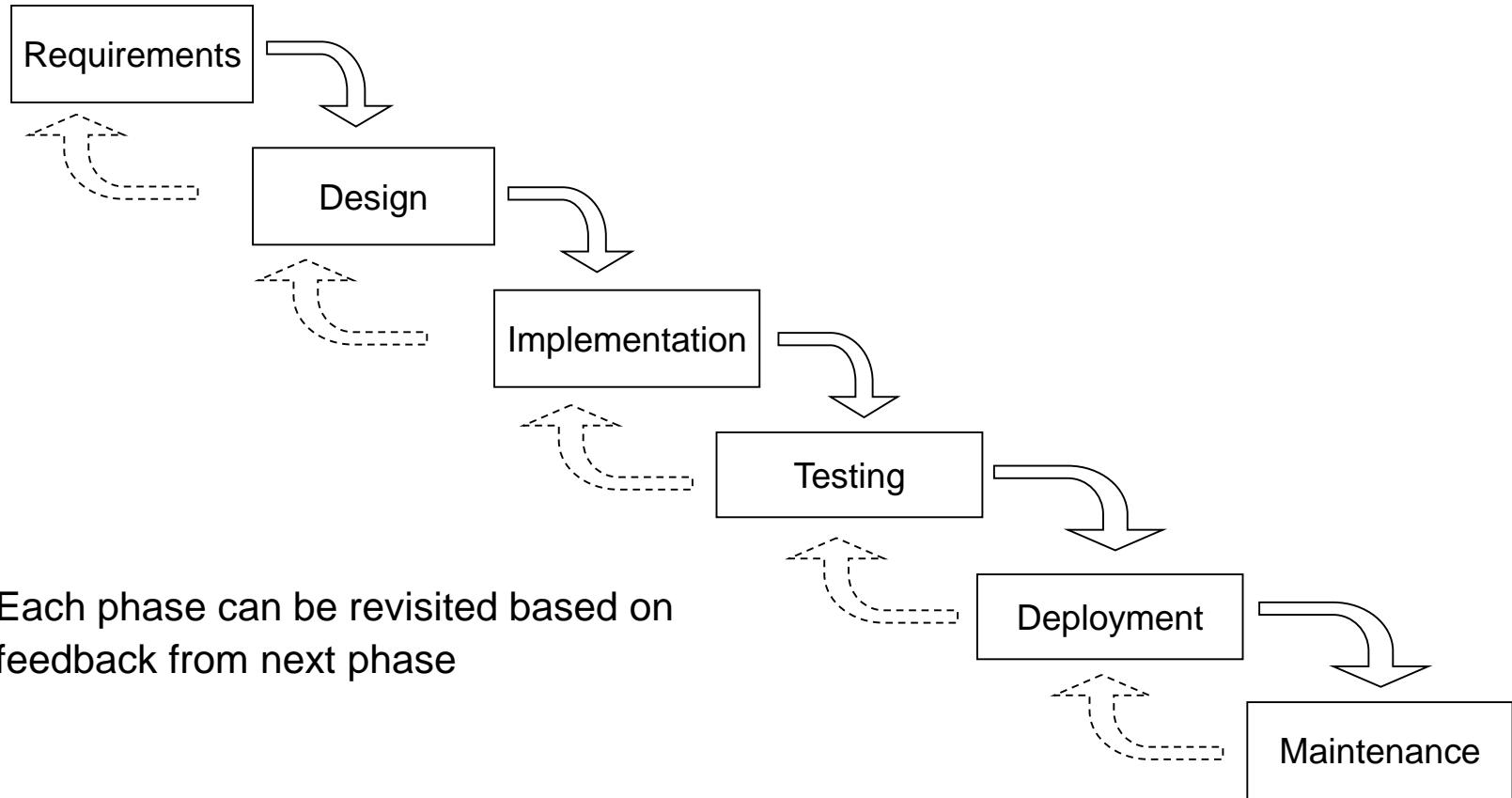
Software process lifecycle

- ▶ Structures the software engineering project
- ▶ Determines:
 - ▶ Tasks: activities
 - ▶ Resources: personnel, equipment
 - ▶ Timing
 - ▶ Deliverables: requirements specification, system design, ...
 - ▶ Tools
- ▶ Various models exist

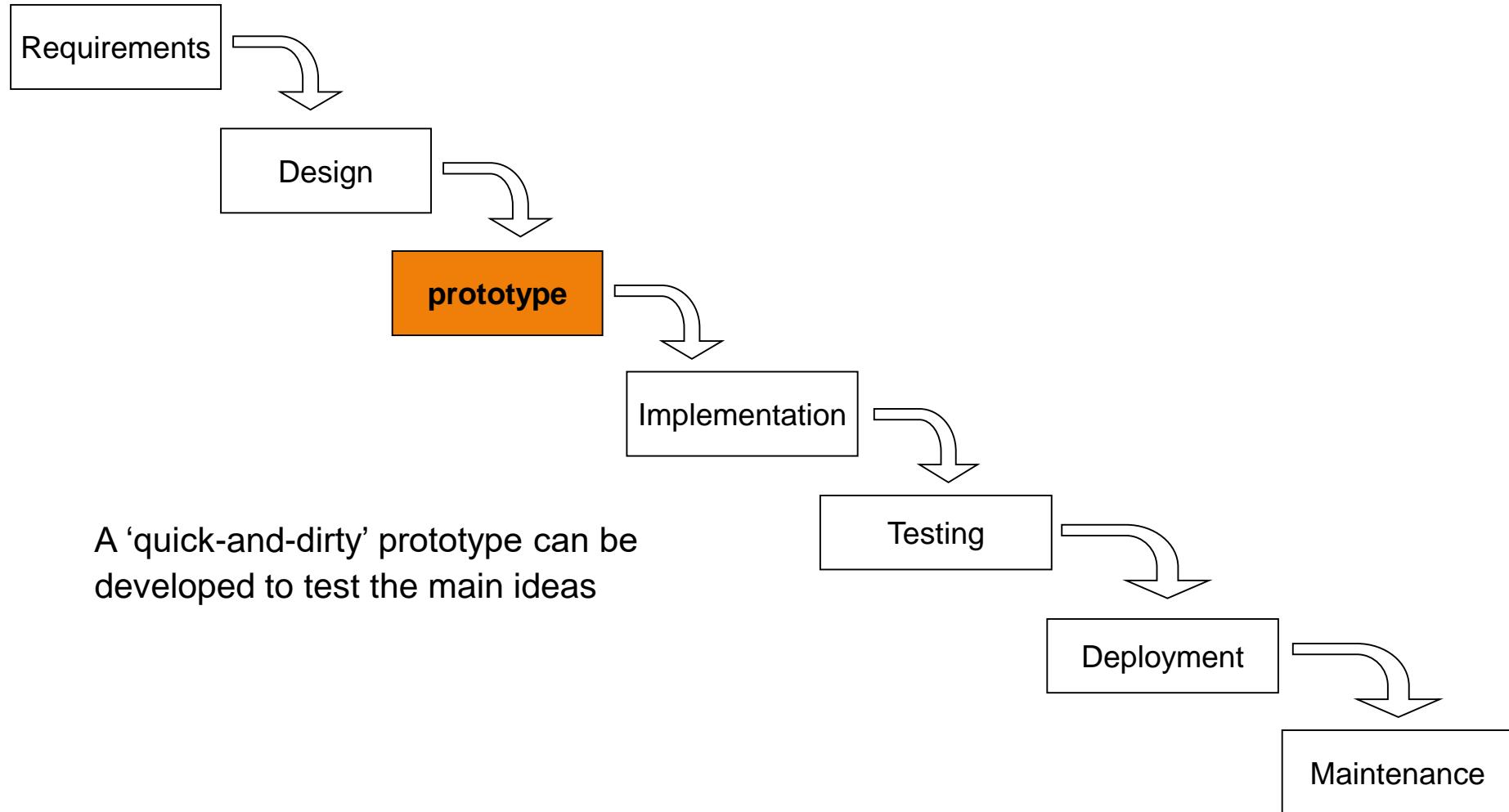
Waterfall model



Waterfall with feedback

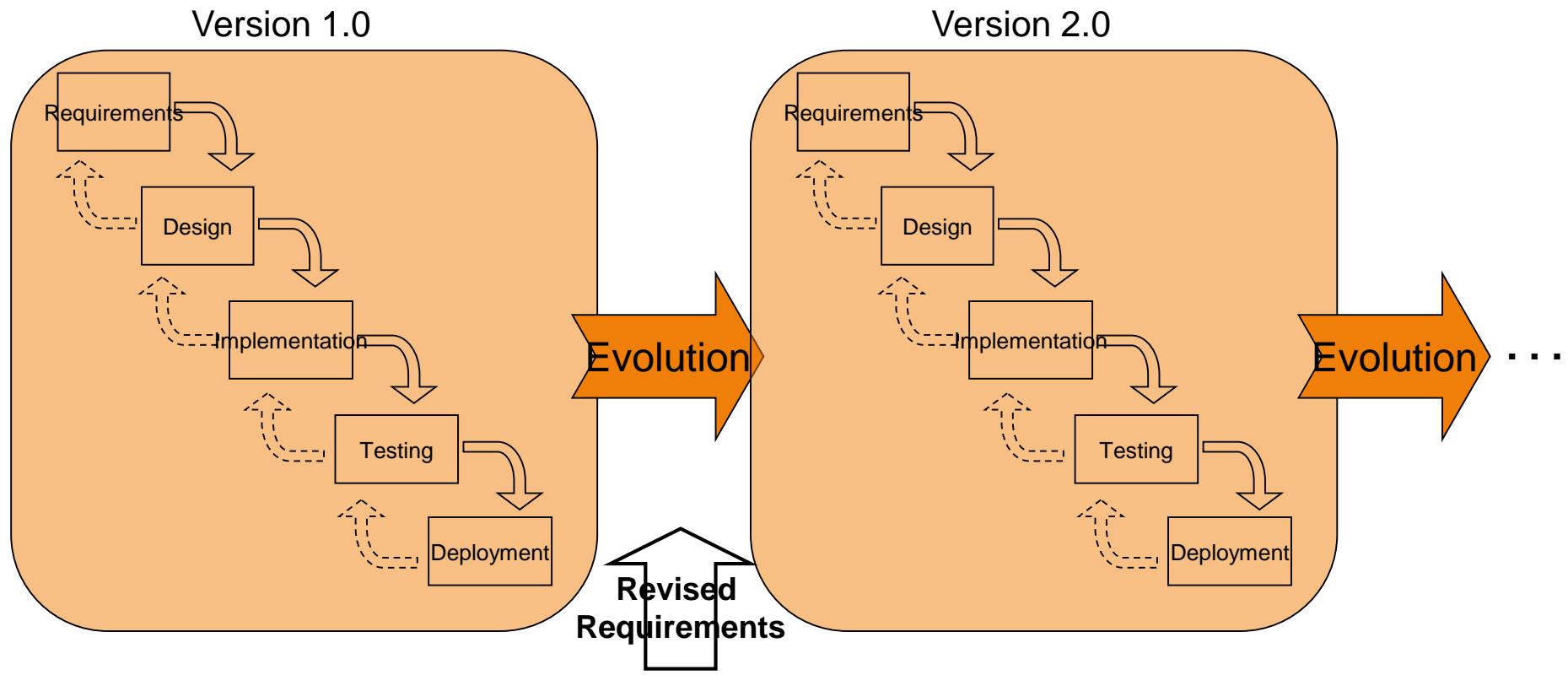


Waterfall with prototyping



Evolutionary model

Evolution, not maintenance



Unified Software Development Process

- ▶ Developed by the team that created UML
- ▶ Embodies best practice in system development
- ▶ Adopts an iterative approach with four main phases
- ▶ Different tasks are captured in a series of workflows



Four Phases

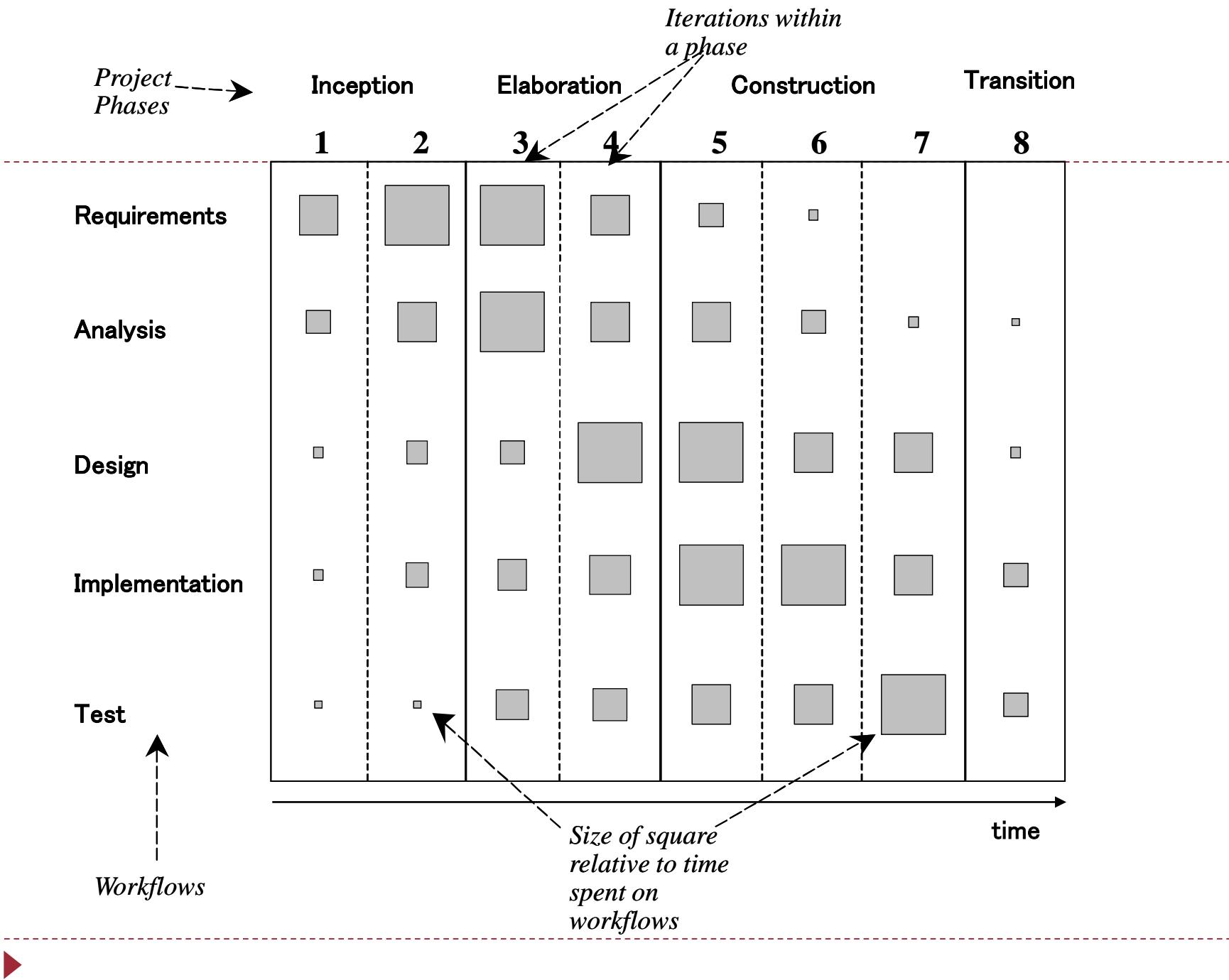
- ▶ Inception
- ▶ Elaboration
- ▶ Construction
- ▶ Transition



Phases, Workflows and Iterations

- ▶ Within each phase activities are grouped into workflows
- ▶ The balance of effort spent in each workflow varies from phase to phase
- ▶ Within phases there may be more than one iteration





Difference from Waterfall Life Cycle

- ▶ In a waterfall life cycle project the phases and the workflows are linked together
- ▶ In the Requirements phase, only Requirements workflow activities are carried out
- ▶ All Requirements activity should be completed before work starts on Analysis
- ▶ In an iterative life cycle project it is recognised that some Requirements work will be happening alongside Analysis work



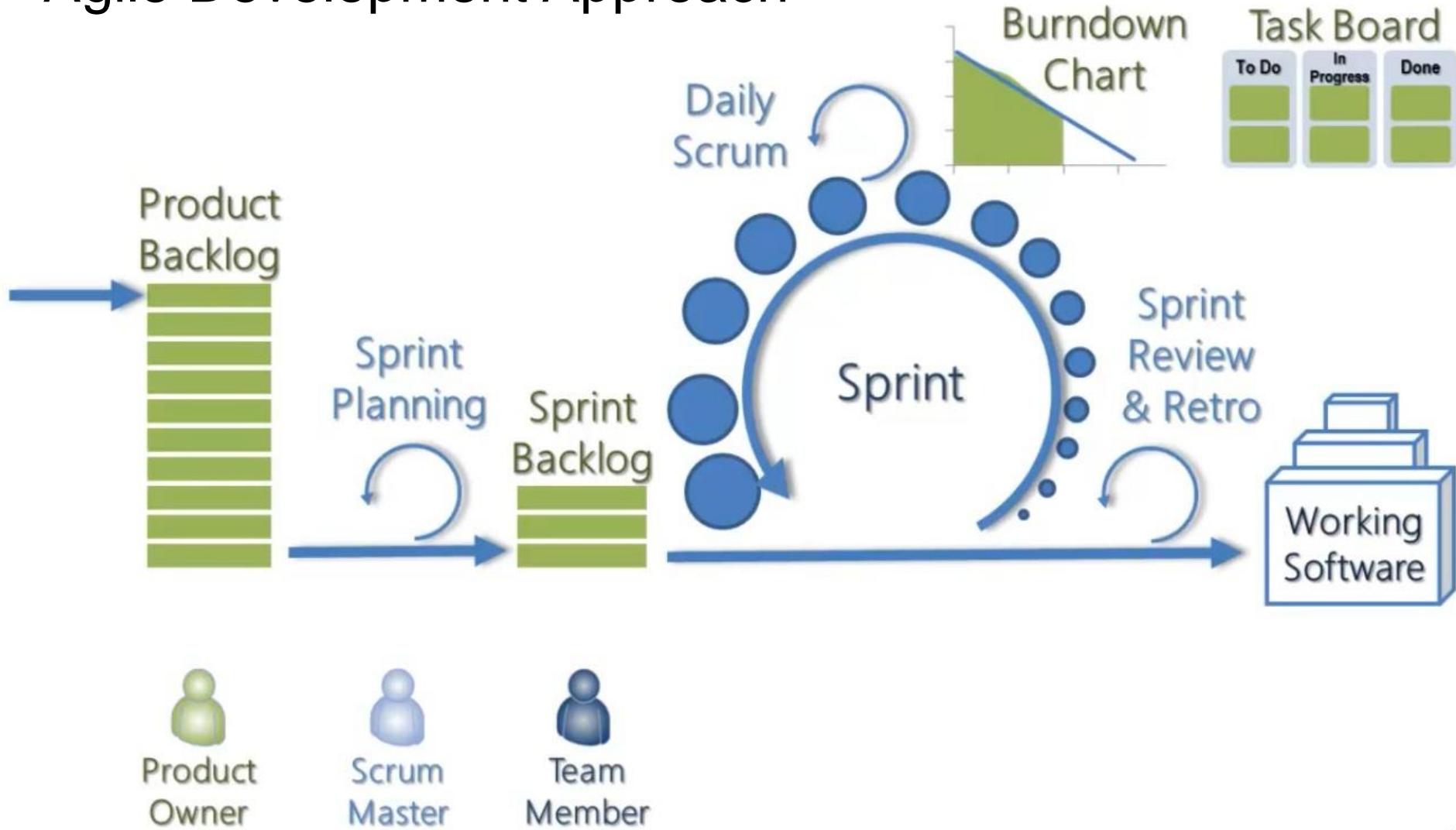
eXtreme or Agile programming

- ▶ The planning game (customer defines value)
- ▶ Small release
- ▶ Metaphor (common vision, common names)
- ▶ Simple design
- ▶ Writing tests first
- ▶ Refactoring
- ▶ Pair programming
- ▶ Collective ownership
- ▶ Continuous integration (small increments)
- ▶ Sustainable pace (40 hours/week)
- ▶ On-site customer
- ▶ Coding standard

Test Scenarios

The Agile approach believes that it is NOT possible to clearly define system requirements up front (as assumed by the waterfall approach).

Agile Development Approach



Metropolis Model

- ▶ Unstable resources
- ▶ Open teams
- ▶ Sufficient correctness
- ▶ Emergent behaviors
- ▶ Mashability
- ▶ Conflicting requirements
- ▶ Continuous evolution
- ▶ Focus on operations

The Metropolis Model: A New Logic for System Development – see <http://www.slideshare.net/rickkazman/metropolis-model>

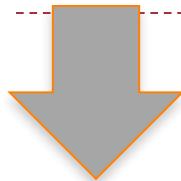
Rick Kazman, University of Hawaii and Software Engineering Institute/CMU

Development methodologies

- ▶ We have only taken a brief look at:
 - ▶ Waterfall
 - ▶ Unified
 - ▶ Extreme & Agile
 - ▶ Metropolis
- ▶ There are others, for example:
 - ▶ Rapid Application Development (RAD)
 - ▶ Dynamic Systems Development Model (DSDM)
 - ▶ Lean Development (LD)
 - ▶ Etc, etc
- ▶ During this course we will focus on the waterfall approach, but these techniques can be universally applied

Problem Definition
Viewpoints/Scope

Waterfall Lifecycle and Main Outputs



Functional &
Non-
Functional
requirements
Scenarios

Use-case
diagrams
Use-case
descriptions
Type Diagrams
Activity
Diagrams
Prototypes
Class Diagram
Interaction
Diagrams
State Machines

Prototypes
Class Diagram
Interaction Diagrams
Package diagrams
Architectural design

Coding
Unit testing
Version control

Integration testing
Unit testing

Deployment testing
User evaluation
Software evolution

All of these techniques
can also be used in agile
development

Finally: Criteria for software quality

- ▶ **External qualities** (directly visible to the client):
 - ▶ Correctness: Perform as intended by the client
 - ▶ Reliability: Absent from failures
 - ▶ Correct → Reliable
 - ▶ Robustness: Ability to survive failures and incorrect input
 - ▶ Efficiency (time/space): Affordable use of resources
 - ▶ Usability: Ease of learning & use
 - ▶ Determined largely by the human-computer interface (HCI)
 - ▶ Example: mobile phones
 - ▶ Safety: Does not pose a risk to humans & property
 - ▶ Secure: Vulnerability to malicious attacks

Criteria for software quality (cont.)

- ▶ **Internal qualities:** concern developers, not directly visible to clients
 - ▶ Maintainability: how well is the software designed, so as to make it
 - ▶ Flexible: can be evolved
 - ▶ Modular/ Loosely coupled: built from relatively independent modules
 - ▶ Cohesive: Maximize functional cohesion within modules
 - ▶ Comprehensible
 - ▶ Reusability: modules of it can be reused for related projects
 - ▶ Portability: can be adapted to run on different types of machines, operating systems
- ▶ We will particularly look at these issues in the **requirements** and **testing** phases of the course

History of UML

- ▶ Came out of the Unified Software Development Process (USDP) – Jacobson et al 1999
 - ▶ Built upon previous approaches
 - ▶ OO Software Engineering Method, Use cases (Jacobson 1992)
 - ▶ OO Design (Booch, 1994)
 - ▶ OO Analysis (OMT, Rumbaugh, 1991)
 - ▶ UML 1.1 1997
 - ▶ UML 2.0 2005
 - ▶ Latest release UML 2.5 (2015)
 - ▶ UML is not a development method by itself
 - ▶ Can be used by methods such as IBM Rational Unified Process (RUP)

Class Homework this week

- ▶ Look at the additional notes below
- ▶ Take a look at the SCRUM Methodology
 - ▶ See <https://www.agilevideos.com/videoscategory/intro-to-scrum-videos/>
 - ▶ Complete the 'Introduction to SCRUM' training package on that page
- ▶ Complete the quiz at the end of the Introduction

"The Scrum methodology of agile software development marks a dramatic departure from waterfall management. In fact, Scrum and other agile processes were inspired by its shortcomings. The Scrum methodology emphasizes communication and collaboration, functioning software, and the flexibility to adapt to emerging business realities — all attributes that suffer in the rigidly ordered waterfall paradigm."

In this lecture, we set the scene:

- ▶ Problems with creating software and why more rigour is needed (ie. software engineering)
- ▶ What is software engineering?
- ▶ Types of software and stakeholders involved
- ▶ Software development processes/methodologies
- ▶ Criteria for software quality
- ▶ History of UML

CE202 next steps

- ▶ Lecture next week
 - ▶ Overview of UML diagrams
 - ▶ UML Activity Diagrams
 - ▶ Requirements engineering & Use-case analysis
- ▶ Remember to do the SCRUM training homework!

Further reading ...

- ▶ Chapter 5 and 21 of Bennett, McRobb and Farmer includes more about the Unified Process as well as Agile alternatives
 - ▶ NATO Software Engineering conference 1968
 - ▶ <http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF>
 - ▶ Software crisis 1989:
 - ▶ Connell, John L. and Shafer, Linda, Structured Rapid Prototyping, Prentice-Hall, Inc., 1989.
 - ▶ Software crisis 2005:
 - ▶ New York Times: 'Does Not Compute',
<http://www.nytimes.com/2005/01/22/opinion/22carr.html?pagewanted=print&position=>
 - ▶ Software crisis 2008:
 - ▶ <http://www.globes.co.il/serveen/globes/docview.asp?did=1000303419&fid=1725>
 - ▶ Brooks, Frederick P. (1987). *No Silver Bullet: Essence and Accidents of Software Engineering*. (Reprinted in the 1995 edition of *The Mythical Man-Month*)
 - ▶ eXtreme programming: Twelve facets of XP: p86 Pfleeger
 - ▶ Metropolis model: ACM July 2009 article - <http://dl.acm.org/citation.cfm?id=1538808>
 - ▶ Jacobson, I., Booch, G., and Rumbaugh, J., *The Unified Software Development Process*, Addison-Wesley, ACM Press, 1999.
-

Specifications

Week 2 Class

CE202 Software Engineering

Cunjin Luo

Special thanks to Michael Ernst & John Chapin
MIT UPOP Program

Specifications tell you **what** to do (but not **how** to do it)

- **A perfect implementation is no good if it solves the wrong problem**
- **It is difficult to create a specification that is**
 - complete
 - consistent
 - precise
 - concise

**What do these cases have
in common?**



**Ariane 5 explosion
1996**

Ariane 5 launch vehicle, 1996

- **Went off course during launch**
 - Ariane 4 guidance software reused in Ariane 5
 - Ariane 5 accelerated much faster
 - velocity variable overflowed, computer crashed
- **"The failure of the Ariane 501 was caused by the complete loss of guidance and attitude information... due to specification and design errors in the software."**

ESA Inquiry Board

Berlin Bundestag 1992



Bundestag Sound System, 1992

- **No sound from speakers in new building**
 - system requirement: no feedback
 - new all-glass room
- **"This glass does not absorb the sound. The computers, detecting feedback, turn down the volume. A steady state is only achieved when the microphones are turned off."**

Dr. Debora Weber-Wulff

Mars Polar Lander Mission Overview

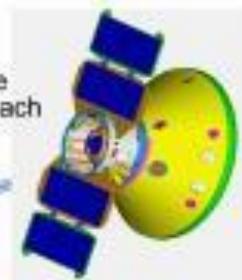
Cruise

Thruster attitude control

Four trajectory-correction maneuvers,
site-adjustment maneuver September 1, 1999,
contingency 5th TCM at entry -24 hours

Eleven month cruise

Near-simultaneous tracking with Mars Climate
Orbiter or Mars Global Surveyor during approach



Entry, Descent, & Landing

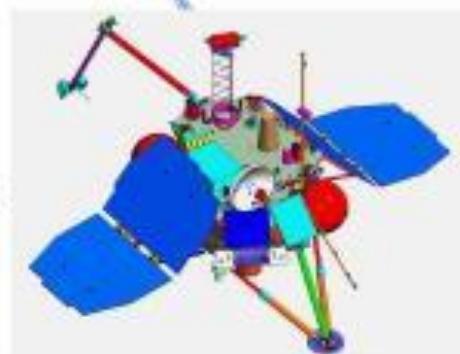
Arrival: December 3, 1999

Jettison cruise stage; microprobes separate
from cruise stage

Hypersonic entry

Parachute descent; propulsive landing

Descent imaging of landing site



Launch

Delta II 7425

Launched January 3, 1999

Launch mass: 574 kilograms

Landed Operations

Lands in Martian spring at 76 degrees
South latitude, 195 degrees West
longitude (76° S, 195° W)

90-day landed mission

Meteorology, imaging, soil analysis,
trenching

Data relay via Mars Climate Orbiter,
Mars Global Surveyor, or direct-to-Earth
high-gain antenna

Mars Polar Lander, 1999

- **Crashed while landing on Mars**
 - sensor transient when legs deployed
 - software thought vehicle had landed
 - engine shut down during descent
- **"There was no software requirement to clear spurious signals prior to using the sensor information to determine that landing had occurred."**

Mars program independent assessment team

Specifications matter

- **A specification:**
 - connects customer and engineer
 - ensures parts of implementation work together
 - defines correctness of implementation
- **Therefore everyone must understand specs**
 - Designers, implementers, testers, managers, marketing, technical support, ... users!
- **Good specifications are essential**

Specification Example

Desktop telephone

Handset (speaker and microphone)

Keypad

talk

redial

ansmachine

end

24-character display

Answering machine

Phone jack

Requirements

Display indicates current functionality

- caller ID
- number being called
- "Answering machine"
- "Ready"

Answering machine picks up after 2 rings

There are other aspects of system behavior

Definitions

Lineidle: phone is on-hook (“hung up”)

- sent from phone to phoneline

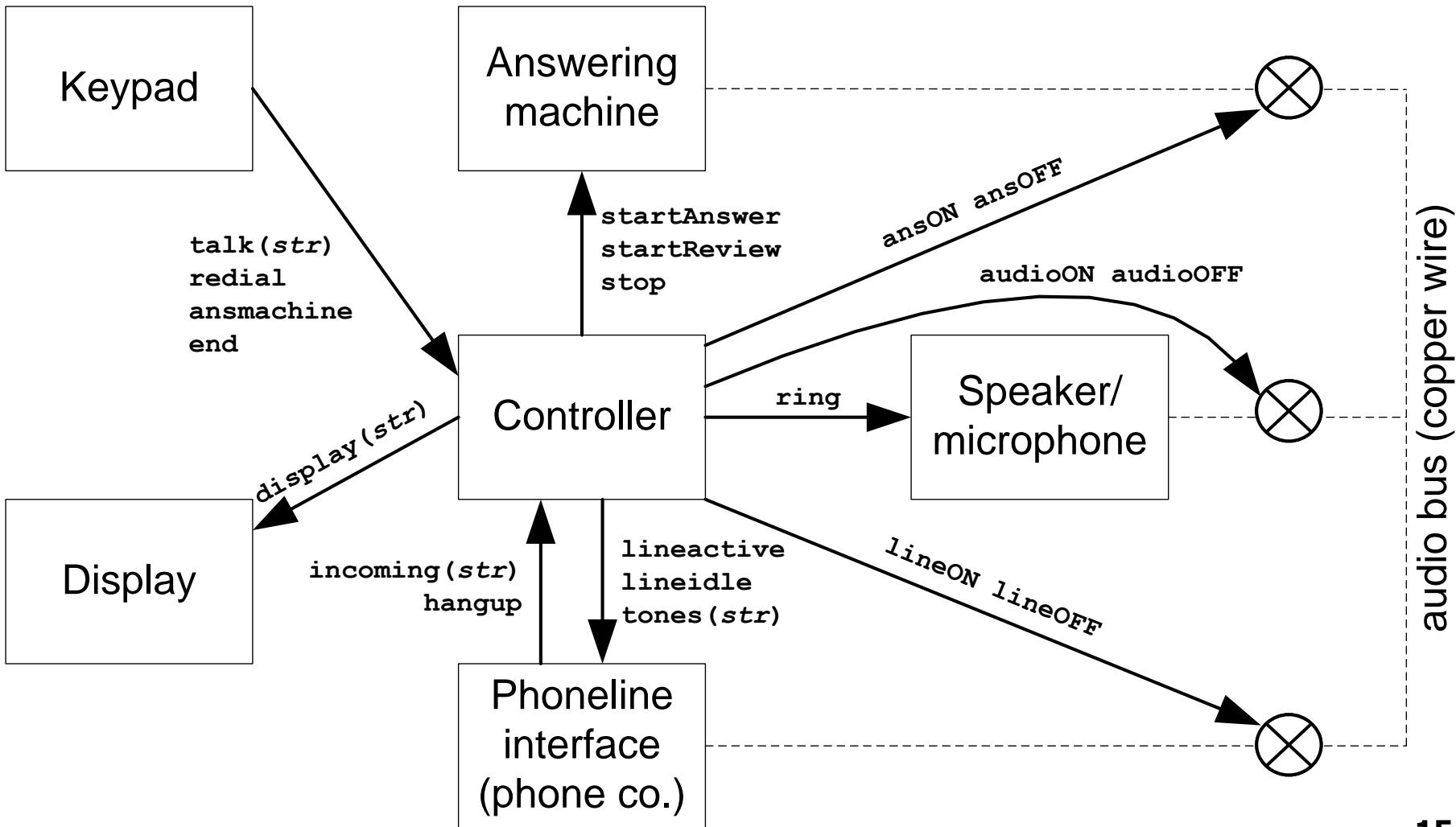
Lineactive: phone is off-hook (“picked up”)

- sent from phone to phoneline

Ring signal: causes phone to ring once

- sent from phoneline to phone

System architecture



Thank you

cunjin.luo@essex.ac.uk

Special thanks to Michael Ernst & John Chapin
MIT UPOP Program

Specification Example

Definitions

<i>lineidle</i>	The phone is hung up or “on hook.” In a traditional phone, this means the handset is lying in the cradle, but your phone uses the end key instead.
<i>lineactive</i>	The phone is picked up or “off hook.” In a traditional phone, this means the handset is not in the cradle (it is “off hook”), but your phone uses the talk key instead.
<i>ring signal</i>	A +/- 24 volt AC signal sent over the phone line, which causes a traditional phone to ring. The phone company only sends a ring signal if it detects the lineidle state.

System Specification

TELEPHONE COMPONENTS

- Handset (includes both speaker and microphone)
- 24-character display
- Answering machine
- Keypad with keys labeled **talk**, **redial**, **ansmachine**, and **end**.

*Simplification: The keypad also has 0 through 9, but in this exercise, you can ignore how those keypresses are handled. When the **talk** key is pressed, the digits previously entered by the user are delivered to the control software (much like a cellular phone). The **redial** key does not deliver any numbers. There is no hook or cradle as with a traditional phone, just the keys.*

FUNCTIONS

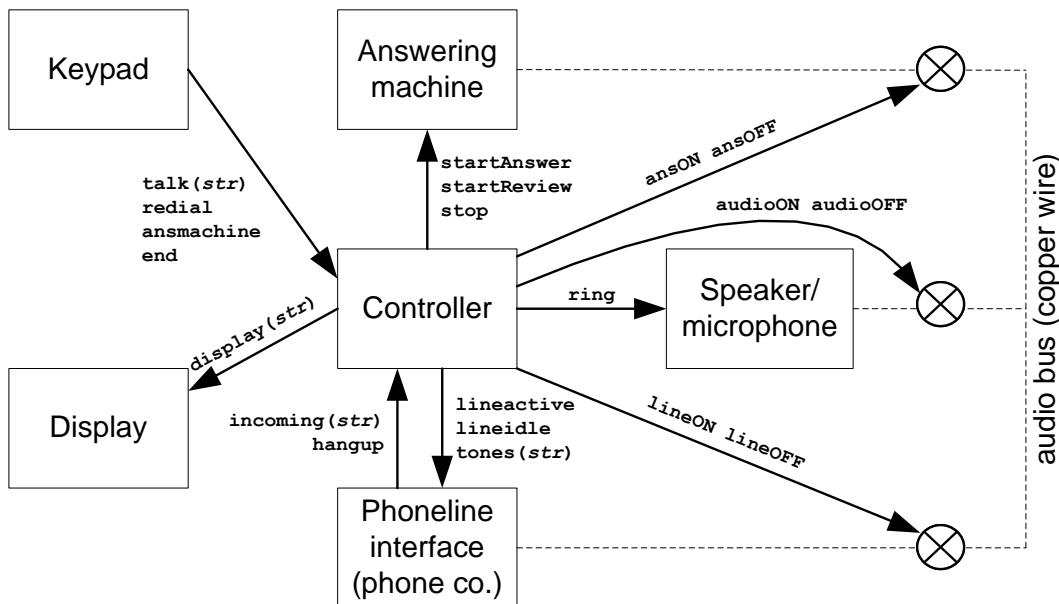
- The user places a call by pressing **talk** or **redial**. The user answers a call by pressing **talk**.
Simplification: Your phone is not required to handle call waiting.
- The user begins using the answering machine by pressing **ansmachine** on the handset.
Simplification: In this exercise, you will not be asked to specify the answering machine's behavior during message review.
- The user presses **end** to end a call or to stop using the answering machine.

REQUIREMENTS

- The display must show the appropriate information at all times.
 - If idle show “READY”
 - If a ring signal is being sent by the phone company show the caller ID information of the caller
 - If connected to an incoming call show the caller ID information of the caller
 - If connected to an outgoing call show the number being called
 - If using the answering machine show “ANSWERING MACHINE”
- If a ring signal is delivered, the telephone must ring and show the caller ID of the caller. If the user doesn't answer the call within 2 rings, the answering machine must pick it up.

System Architecture

The telephone has the following components. The messages that may be exchanged between the handset controller and the other components are labeled in the diagram. Analog audio links are shown with dashed lines. Switches (represented by \otimes) either make or break audio connections.



talk(string)	The user typed the digits in the argument string and then pressed talk
redial	The user pressed redial
ansmachine	The user pressed ansmachine
end	The user pressed end
display(string)	Makes the LCD display show the characters in string , a 24-character string
startAnswer	Play outgoing message and record the caller's message
startReview	Play back recorded messages and perform other user interactions
stop	Stop answering machine functions, return to idle state
incoming(string)	The phone company sent a ring signal with string as caller ID information. This message is repeatedly sent (every 6 seconds) until the call is answered or the caller hangs up.
hangup	The phone company indicates that the remote party has hung up
lineactive	Put the resistance across the phone line that indicates the phone is active
lineidle	Put the resistance across the phone line that indicates the phone is idle
tones(string)	Send the digits in string out over the phoneline as touch-tones
ring	Causes the speaker to play one ring tone
ansON ansOFF	Connect/disconnect the answering machine to the audio bus
audioON audioOFF	Connect/disconnect the speaker and microphone to the audio bus
lineON lineOFF	Connect/disconnect the phoneline to the audio bus

Simplification: Messages among telephone components are never lost or corrupted.

Software modelling

- Models and Diagrams
- Diagram examples
 - Use-case diagrams
 - Class diagrams
 - Package diagrams
 - Sequence diagrams
 - Collaboration diagrams
 - Type diagrams
- Activity Diagrams



CE202 Software Engineering, Autumn term

Dr Cunjin Luo, School of Computer Science and Electronic Engineering, University of Essex

Introduction to software modelling

- ▶ Purpose of modelling
 - ▶ Managing complexity
 - ▶ Abstraction
 - ▶ Communication: between stakeholders (programmers, designers, architects, clients, ...)
 - ▶ Documentation: for maintenance and evolution
- ▶ Kinds of models
 - ▶ Object models: classes, objects, messages, relations, ...
 - ▶ Functional models: what the system ‘does’ , e.g., flow charts, Statecharts
 - ▶ Physical models: which physical units the systems constitute
 - ▶ Task models: units of work, schedule, resources. E.g.: PERT, Gantt, ...
- ▶ Notations
 - ▶ (OMT: Predecessor to UML, Object Modelling Techniques (Rumbaugh et al))
 - ▶ UML: Unified Modeling Language
 - ▶ Statecharts
 - ▶ Data flow diagrams
 - ▶ Type diagrams



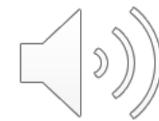
What is a Model

- ▶ Like a map, a model represents something else
- ▶ A useful model has the right level of detail and represents only what is important for the task in hand
- ▶ Many things can be modelled: bridges, traffic flow, buildings, economic policy



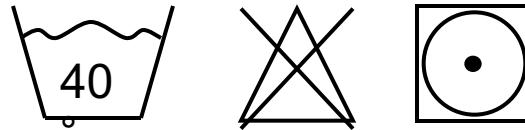
Why Use a Model?

- ▶ A model is quicker and easier to build than the real thing
- ▶ A model can be used in a simulation
- ▶ A model can evolve as we learn
- ▶ We can choose which details to include in a model
- ▶ A model can represent real or imaginary things from any domain



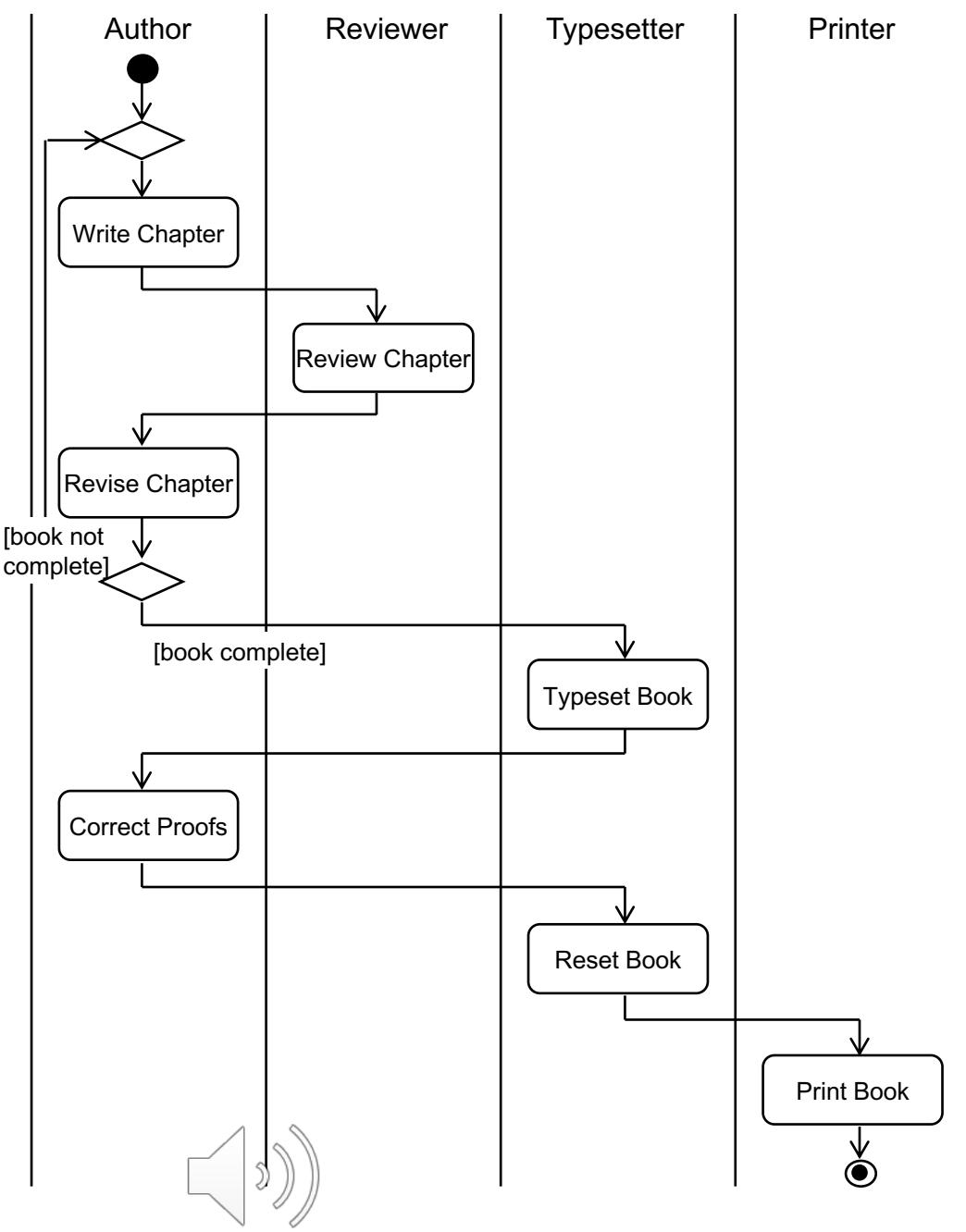
What is a Diagram?

- ▶ Abstract shapes are used to represent things or actions from the real world
- ▶ Diagrams follow rules or standards
- ▶ The standards make sure that different people will interpret the diagram in the same way

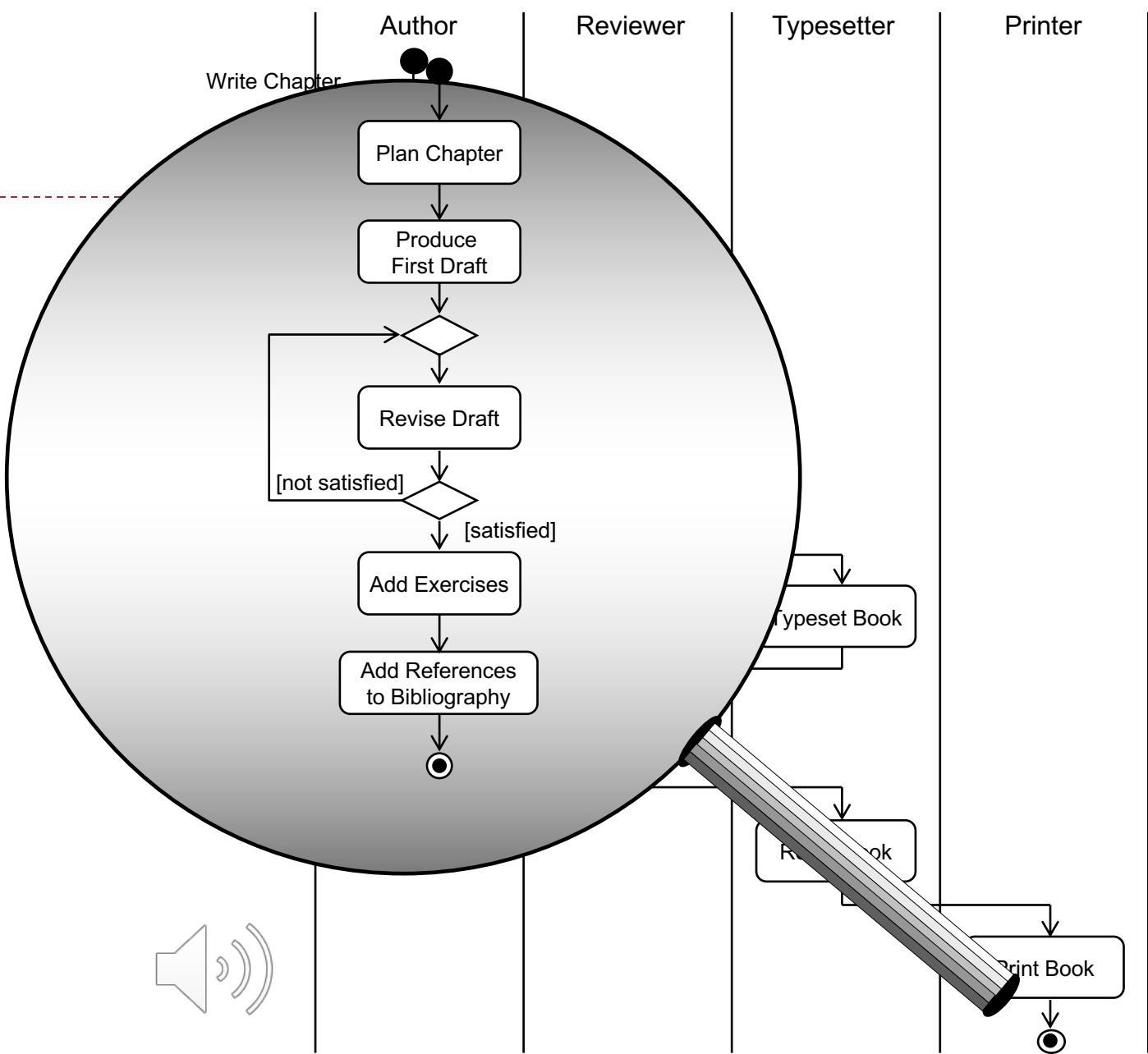


An Example of a Diagram

- ▶ An activity diagram of the tasks involved in producing a book.



Hiding Detail

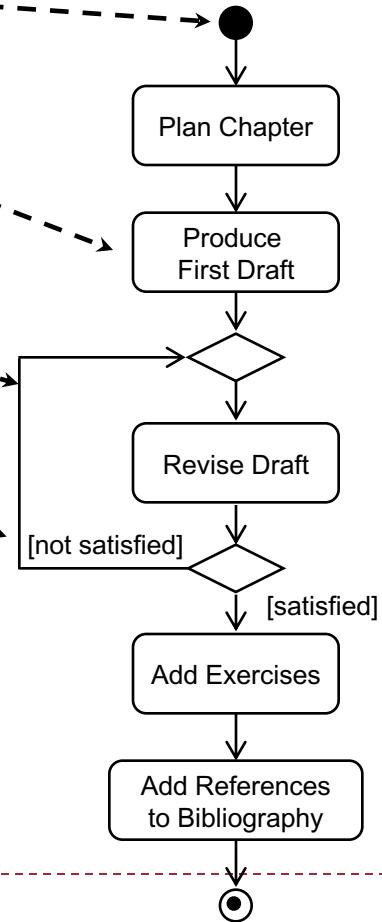
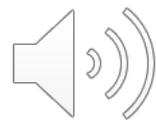


Diagrams in UML

- ▶ UML diagrams consist of:

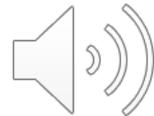
- ▶ icons
- ▶ two-dimensional symbols
- ▶ paths
- ▶ Strings

- ▶ UML diagrams are defined in the UML specification.



Developing Models

- ▶ During the life of a project using an iterative life cycle, models change along the dimensions of:
 - ▶ abstraction—they become more concrete
 - ▶ formality—they become more formally specified
 - ▶ level of detail—additional detail is added as understanding improves

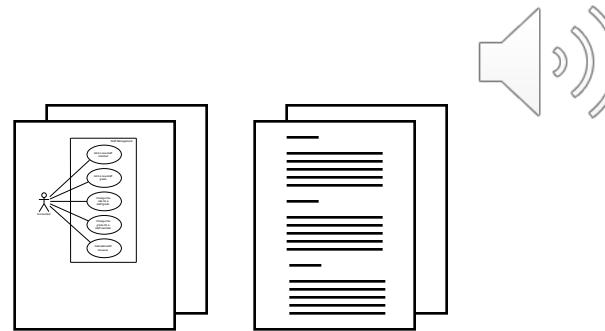


Development of the Use Case Model

Iteration 1

Obvious use cases.

Simple use case descriptions.

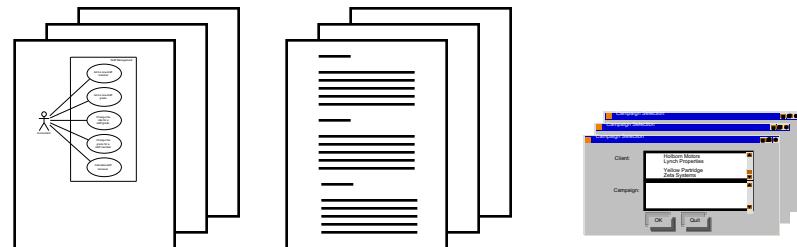


Iteration 2

Additional use cases.

Simple use case descriptions.

Prototypes.

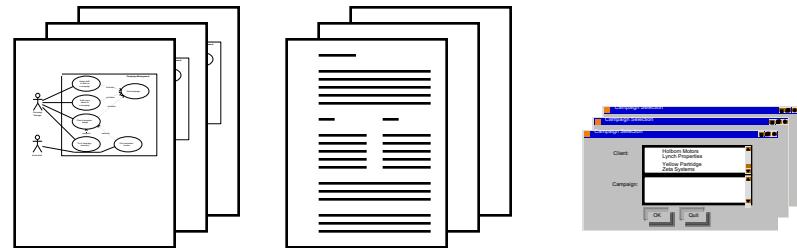


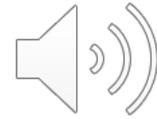
Iteration 3

Structured use cases.

Structured use case descriptions.

Prototypes.



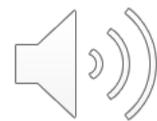


Some example diagrams

We will explore these in more detail in future lectures

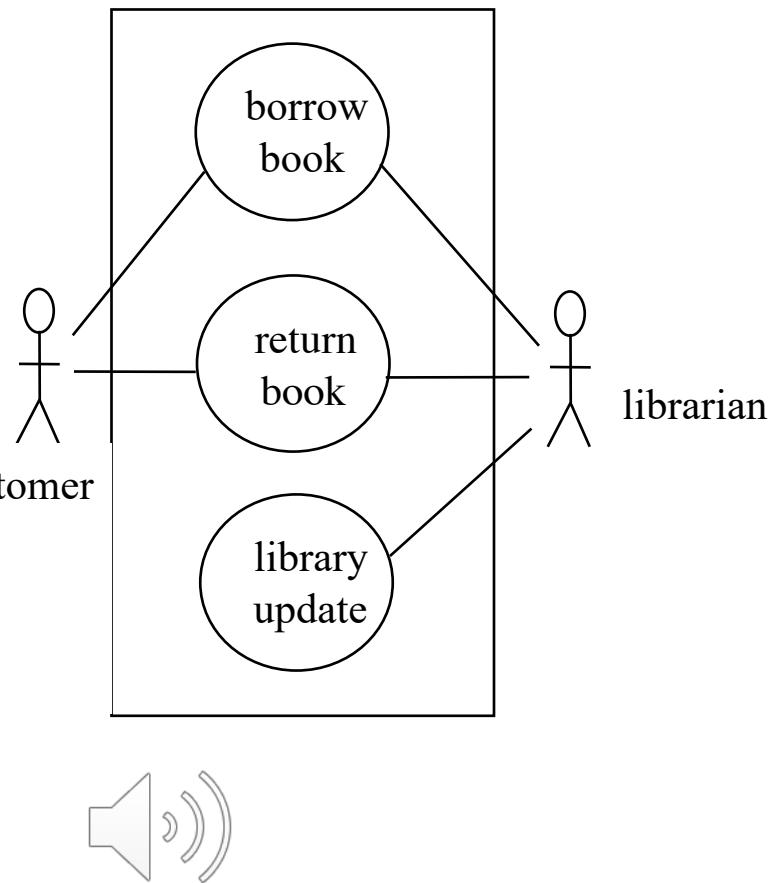
Types of diagram: Static vs. dynamic models

- ▶ **Static** models describe structural aspects:
 - ▶ Class Diagrams
 - ▶ Package Diagrams
 - ▶ Type Diagrams
- ▶ **Dynamic** models describe behavioral aspects:
 - ▶ Use Case Diagrams
 - ▶ Sequence Diagrams
 - ▶ Collaboration Diagrams
 - ▶ Activity Diagrams



UML use-case diagrams

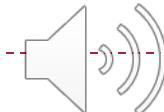
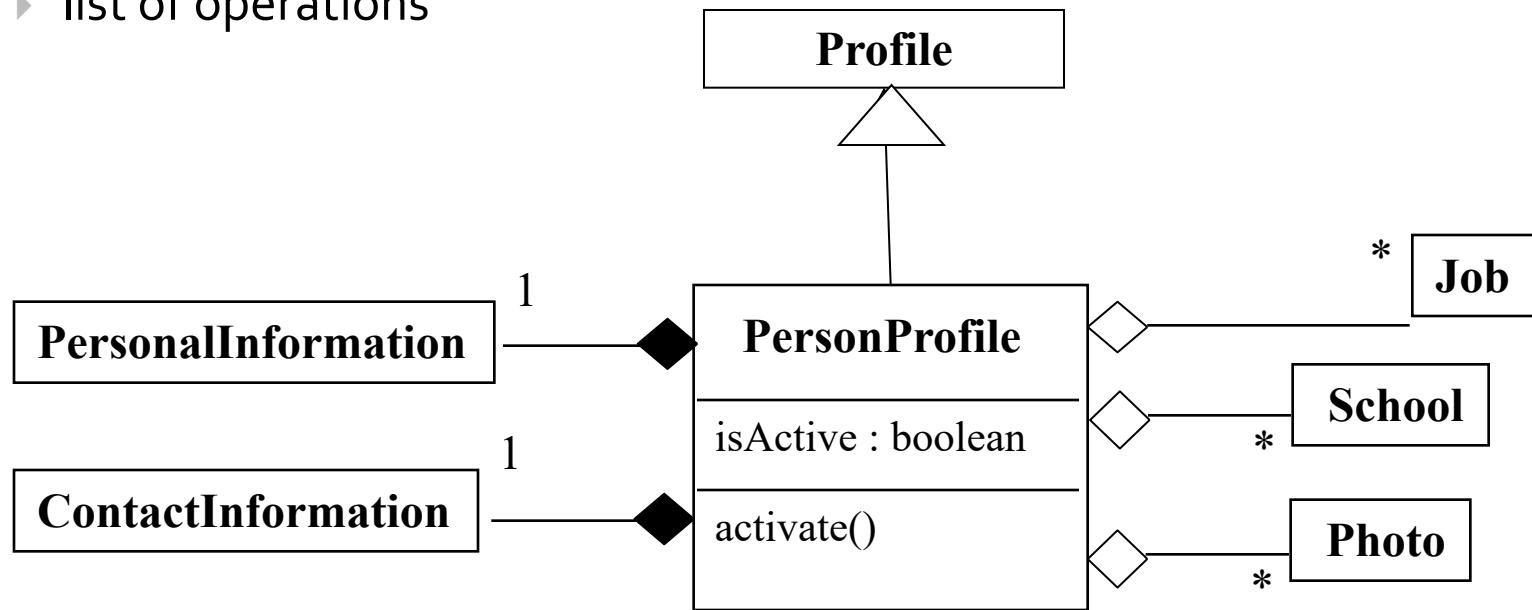
- ▶ Specify ONLY user views of essential system behaviour
- ▶ Do NOT specify the flow of processes in the implementation
- ▶ Components
 - ▶ A large box: system boundary
 - ▶ Stick figures outside the box: actors, both human and systems
 - ▶ Each oval inside the box: a use case that represents some major required functionality and its variant
 - ▶ A line between an actor and use case: the actor participates in the use case



UML class diagrams

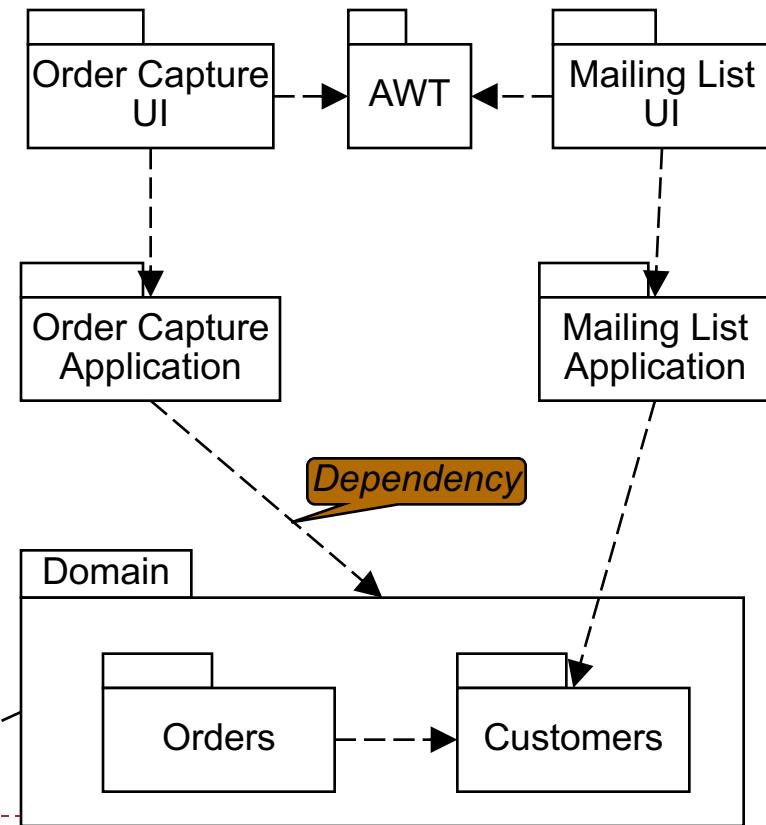
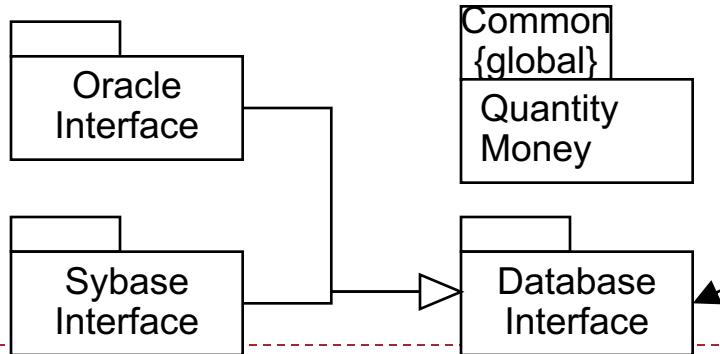
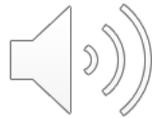
► Classes, associations, interfaces

- ▶ Class: solid rectangle with 3 compartments
 - ▶ class name and other general properties of the class
 - ▶ list of attributes
 - ▶ list of operations

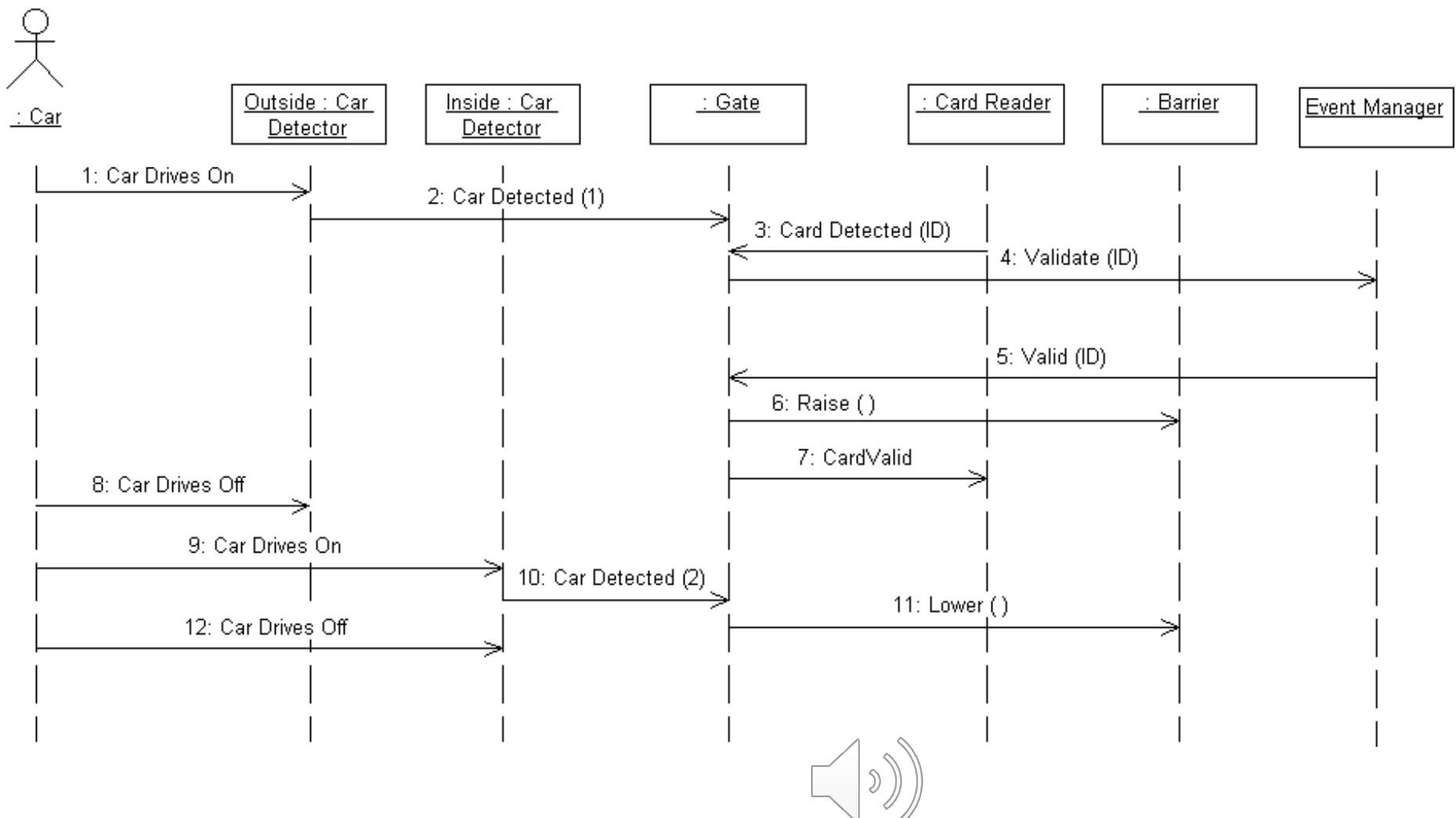


UML package diagram

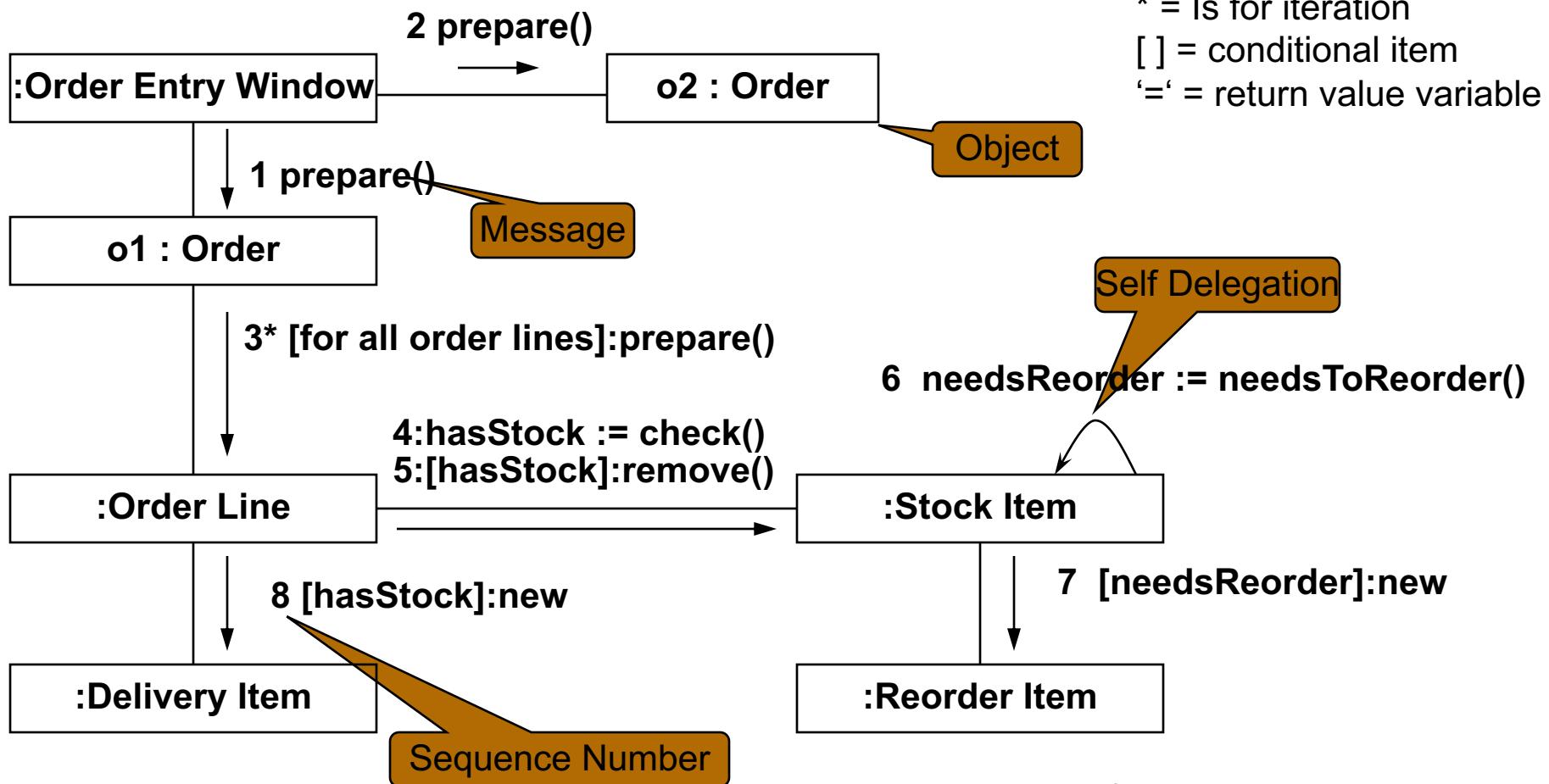
- ▶ UML package diagrams allow viewing a system as a small collection of packages each of which may be expanded to a larger set of classes
- ▶ See p120 & 244 Bennett

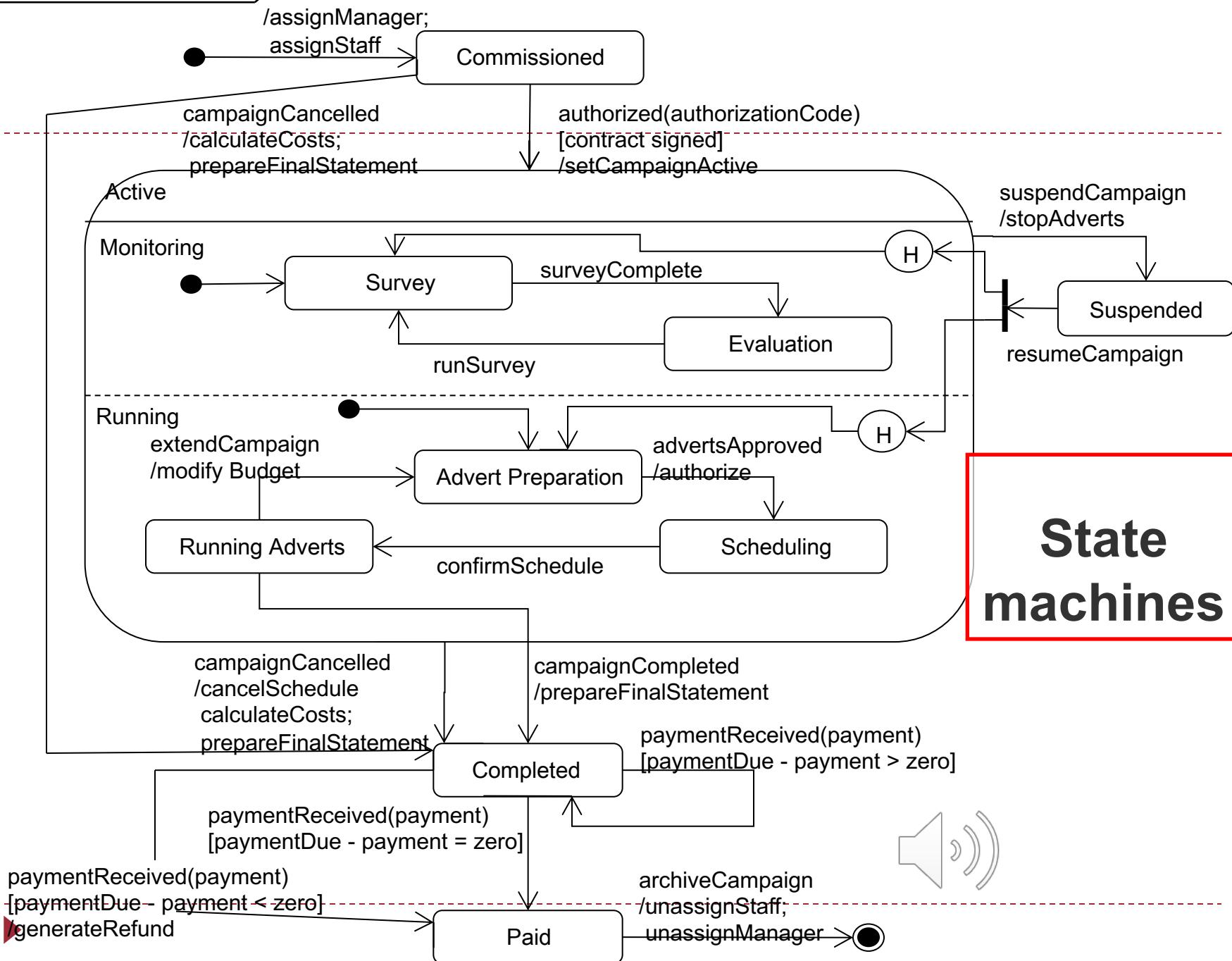


UML sequence diagrams



UML collaboration diagrams





State machines



UML pros and cons

▶ Pros

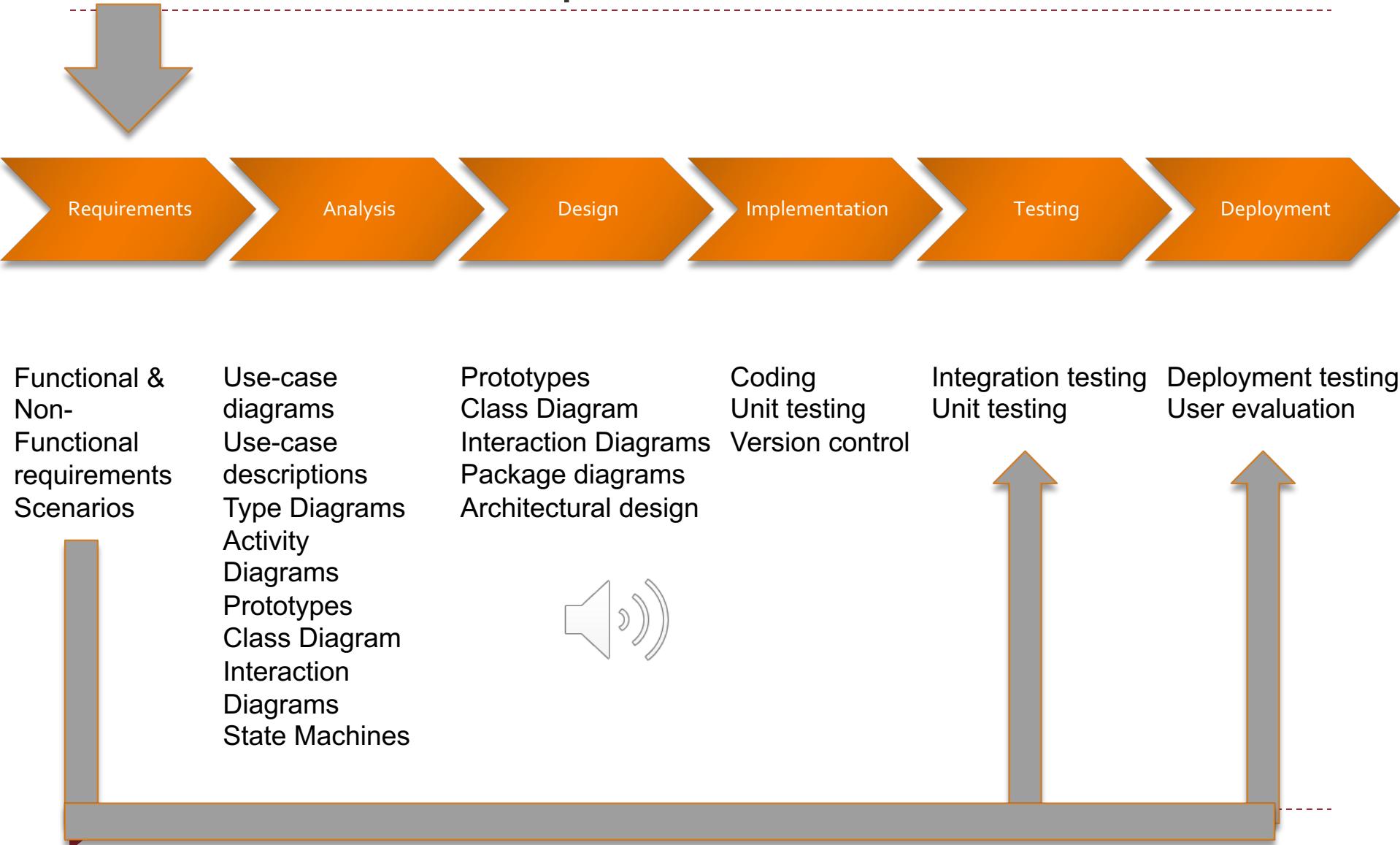
- ▶ De-facto industry standard for software representation
- ▶ A standard of communication
 - ▶ A common vocabulary
 - ▶ Accepted notation
 - ▶ A visual abstraction

▶ Cons

- ▶ Language? No well-defined semantics
 - No means for proving/refuting consistency with implementation
 - No automated verification
- ▶ Modelling? Low level of abstraction
 - Inadequate means for abstraction & navigation
- ▶ Unified? Little integration between sub-notations



Reminder: Waterfall Lifecycle and Main Outputs



Analysis Flow in UML (& linkages between diagrams)

1 Requirements

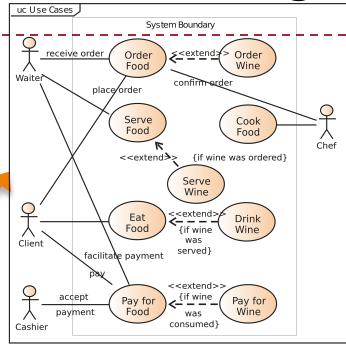
Problem Definition
Viewpoints/Scope
Functional & Non-Functional requirements
Scenarios

3 Use-case descriptions

Name	The Use Case name. Typically the name is of the format <action> + <object>.
ID	An identifier that is unique to the Use Case.
Description	A brief sentence that states what the user wants to be able to do and what benefit he will derive.
Actors	The type of user who interacts with the system to accomplish the task. Actors are identified by role name.
Organizational Benefits	The value the organization expects to receive from having the functionality described. Ideally this is a link directly to a Business Objective.
Frequency of Use	How often the Use Case is executed.
Triggers	Concrete actions made by the user within the system to start the Use Case.
Preconditions	Any states that the system must be in or conditions that must be met before the Use Case is started.
Postconditions	Any states that the system must be in or conditions that must be met after the Use Case is completed successfully. These will be met if the Main Course or any Alternate Courses are followed. Some Exceptions may result in failure to meet the Postconditions.
Main Course	The most common path of interactions between the user and the system. 1. Step 1 2. Step 2
Alternate Courses	Alternate paths through the system. AC1: <condition for the alternate to be called> 1. Step 1 2. Step 2 AC2: <condition for the alternate to be called> 1. Step 1
Exceptions	Exception handling by the system. EX1: <condition for the exception to be called> 1. Step 1 2. Step 2 EX2: <condition for the exception to be called> 1. Step 1

2

2 Use-case diagrams



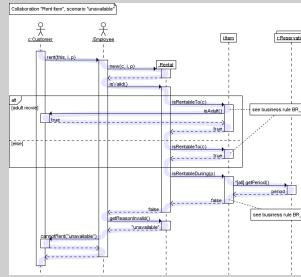
Actors

A sequence diagram for each use-case description

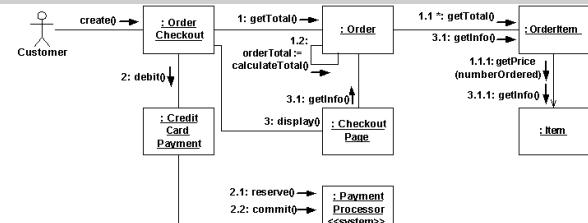
4 List of Candidate Classes:

Campaign
Order
Customer

Sequence diagrams



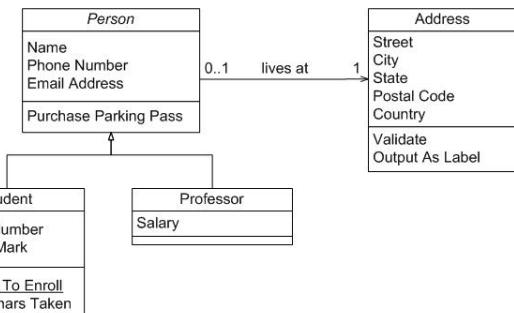
Collaboration diagrams



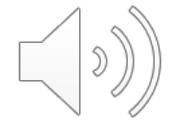
Names in bubbles match with use-case description names

5

6 Class diagram



Class names



UML Activity Diagrams

Our first example: Activity Diagrams

► Purpose

- ▶ to model a task (for example in business modelling)
- ▶ to describe a function of a system represented by a use case
- ▶ to describe the logic of an operation



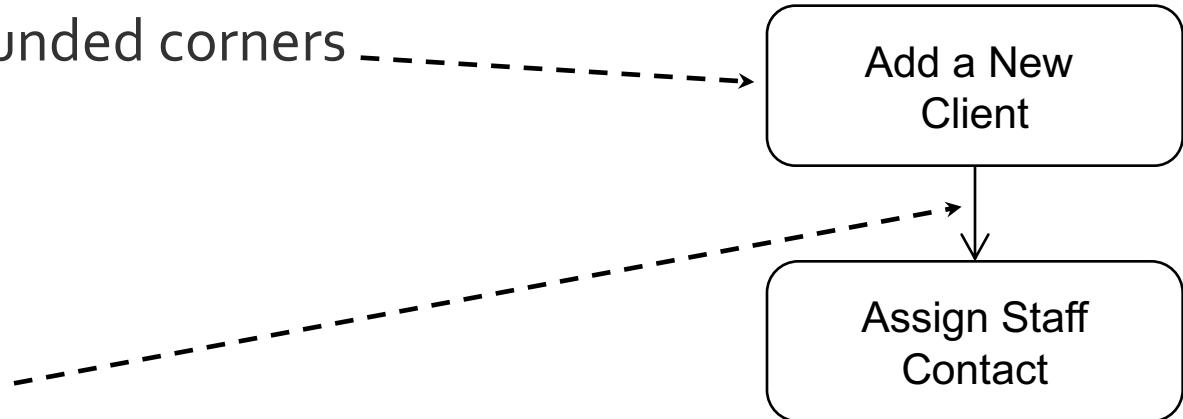
Notation of Activity Diagrams

► Actions

- ▶ rectangle with rounded corners
- ▶ meaningful name

► Control flows

- ▶ arrows with open arrowheads



Notation of Activity Diagrams

► Initial node

- ▶ black circle

► Decision nodes

(and merge nodes)

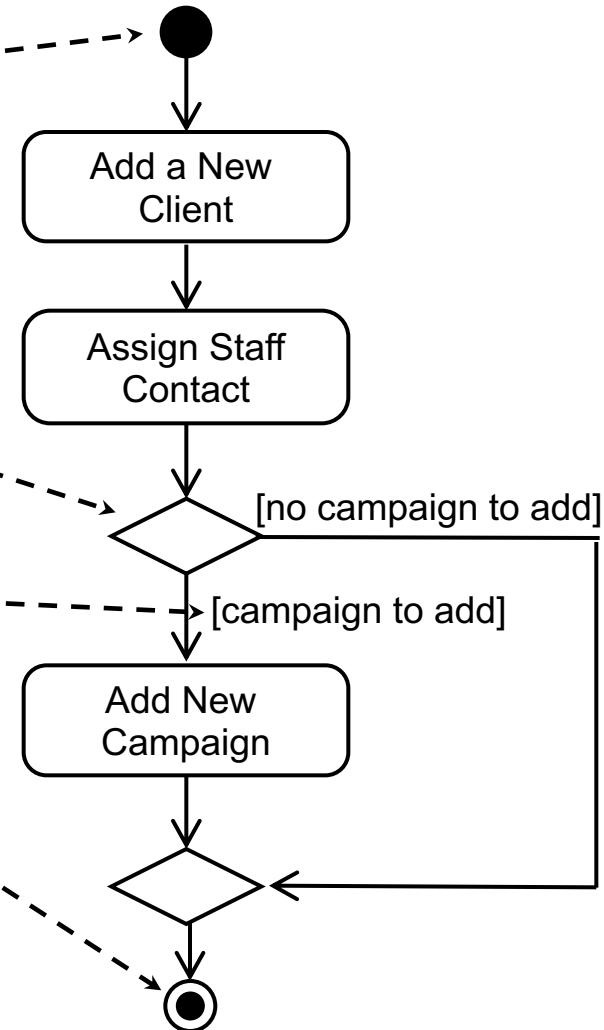
- ▶ diamond

► Guard conditions

- ▶ in square brackets

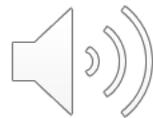
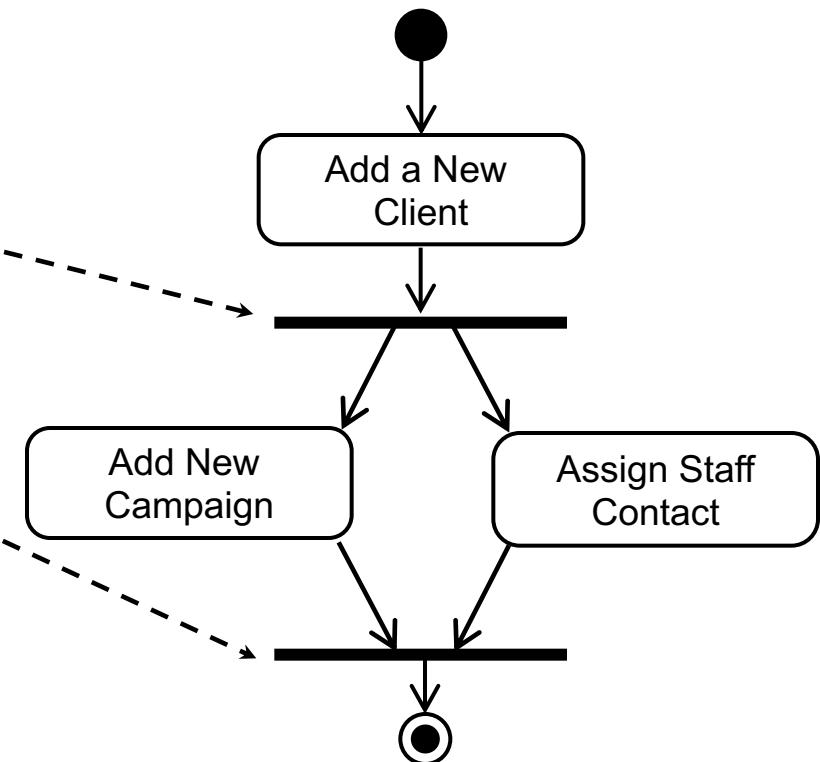
► Final node

- ▶ black circle in white circle



Notation of Activity Diagrams

- ▶ Fork nodes
and join nodes
 - ▶ thick bar
- ▶ Actions carried
out in parallel



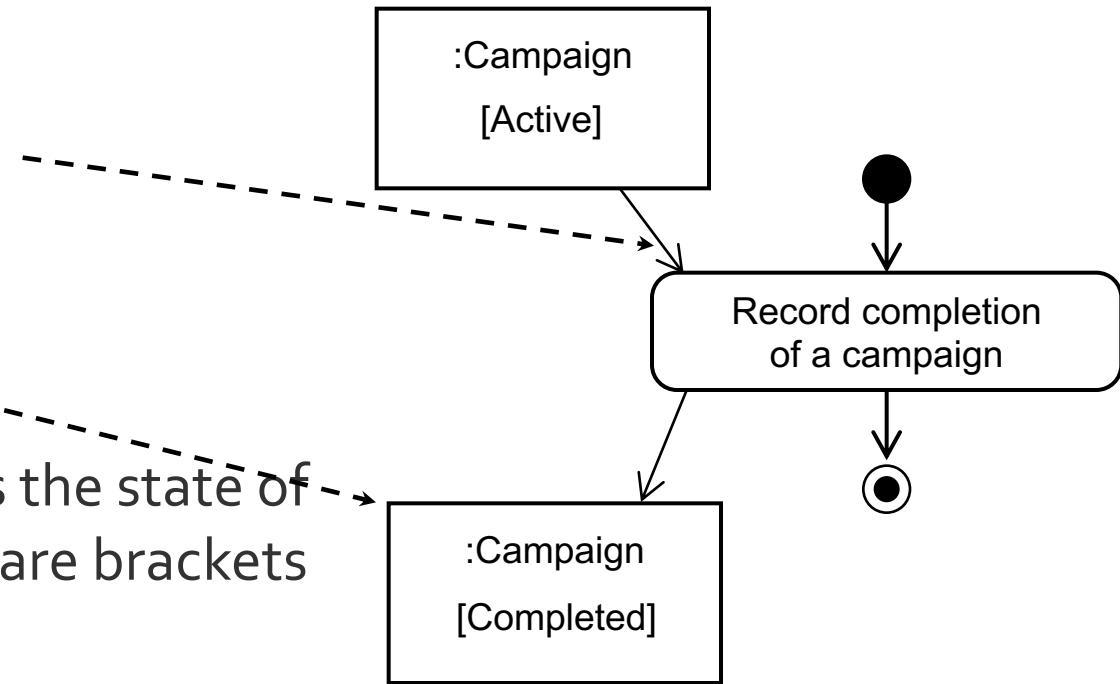
Notation of Activity Diagrams

- ▶ Object flows

- ▶ open arrow

- ▶ Objects

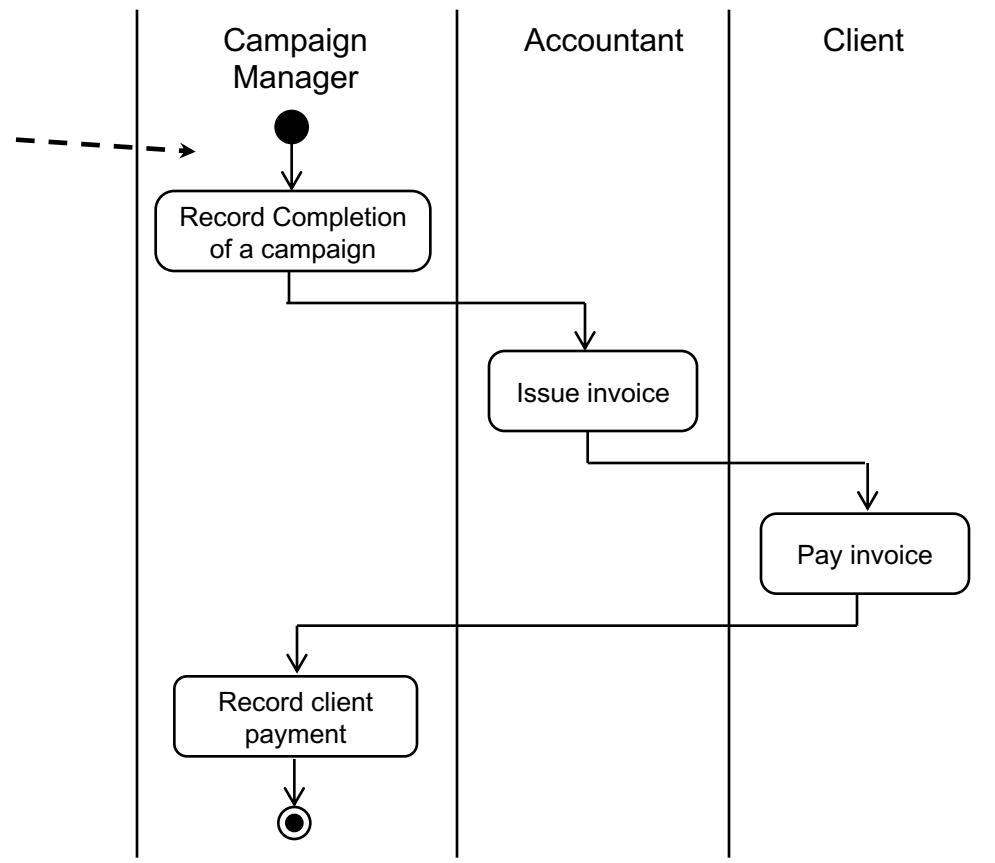
- ▶ rectangle
 - ▶ optionally shows the state of the object in square brackets



Notation of Activity Diagrams

▶ Activity Partitions (Swimlanes)

- ▶ vertical columns
- ▶ labelled with the person, organisation, department or system responsible for the activities in that column

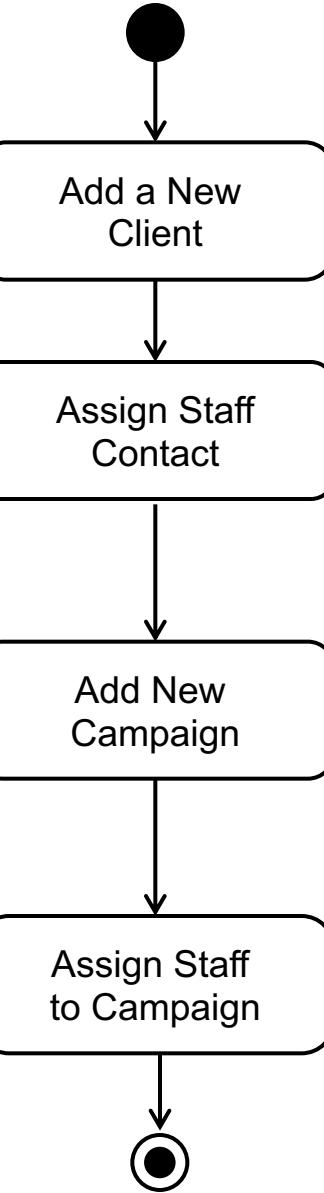


Drawing Activity Diagrams

- ▶ Identify actions
 - ▶ Eg. What happens when a new client is added in the system?
 - ▶ Add a New Client
 - ▶ Assign Staff Contact
 - ▶ Add New Campaign
 - ▶ Assign Staff to Campaign
- ▶ Organise the actions in order with flows

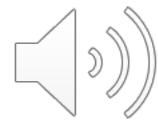


Drawing Activity Diagrams

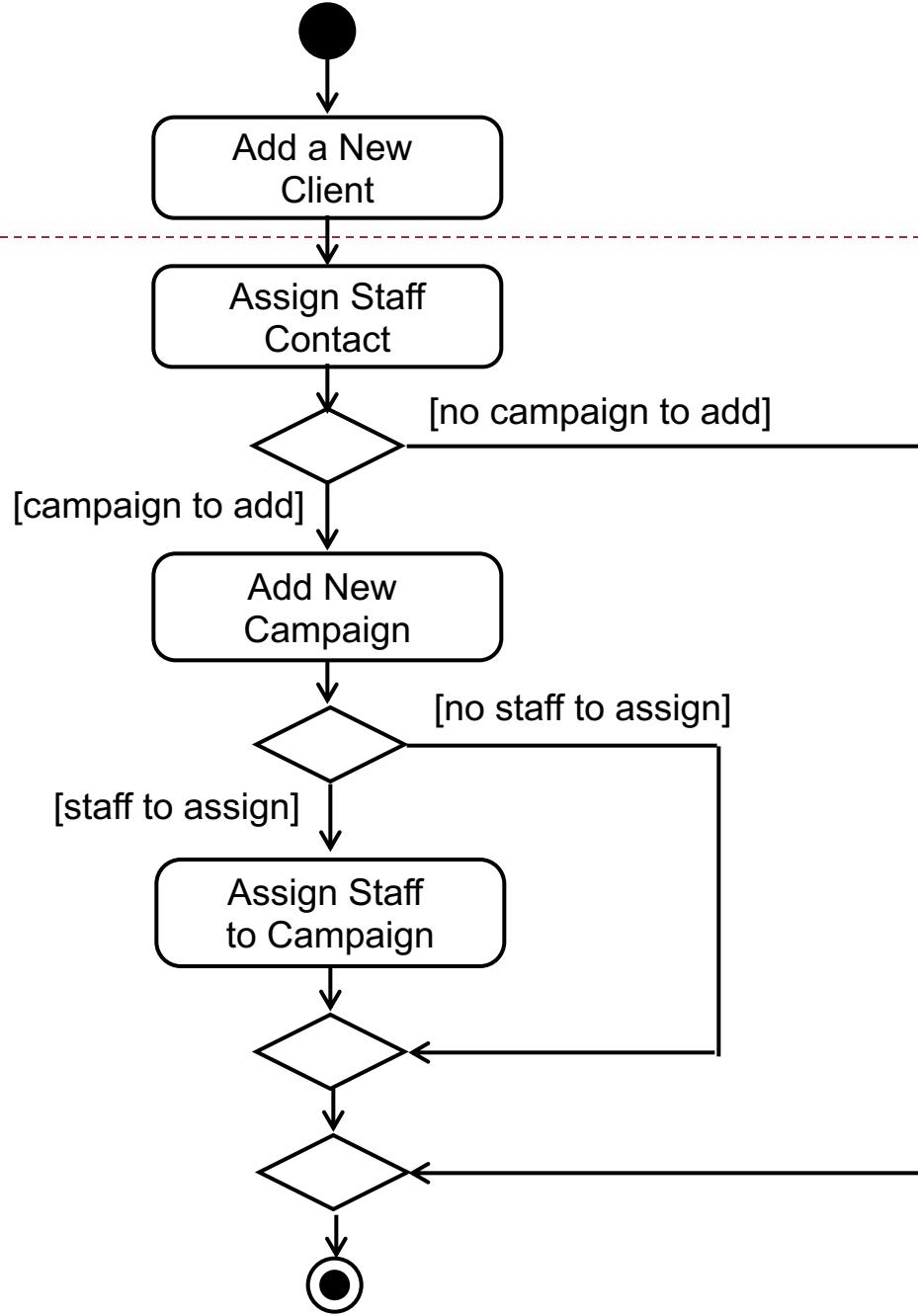
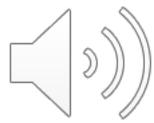


Drawing Activity Diagrams

- ▶ Identify any alternative flows and the conditions on them
 - ▶ sometimes there is a new campaign to add for a new client, sometimes not
 - ▶ sometimes they will want to assign staff to the campaign, sometimes not
- ▶ Add decision and merge nodes, flows and guard conditions to the diagram

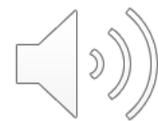


Drawing Activity Diagrams



Drawing Activity Diagrams

- ▶ Identify any actions that are carried out in parallel
 - ▶ there are none in this example
- ▶ Add fork and join nodes and flows to the diagram

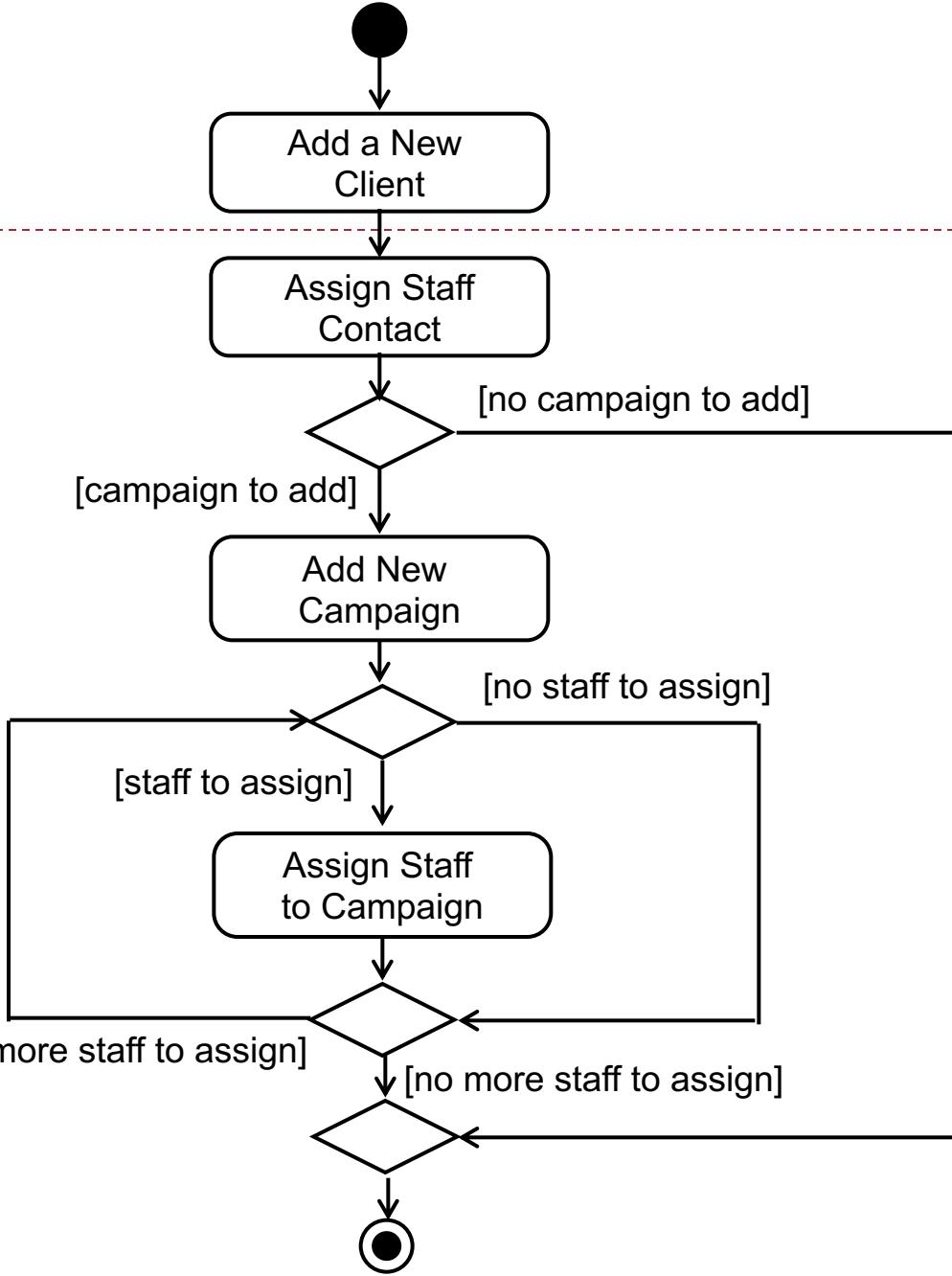


Drawing Activity Diagrams

- ▶ Identify any processes that are repeated
 - ▶ they will want to assign staff to the campaign until there are no more staff to add
- ▶ Add decision and merge nodes, flows and guard conditions to the diagram

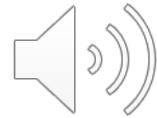


Drawing Activity Diagrams



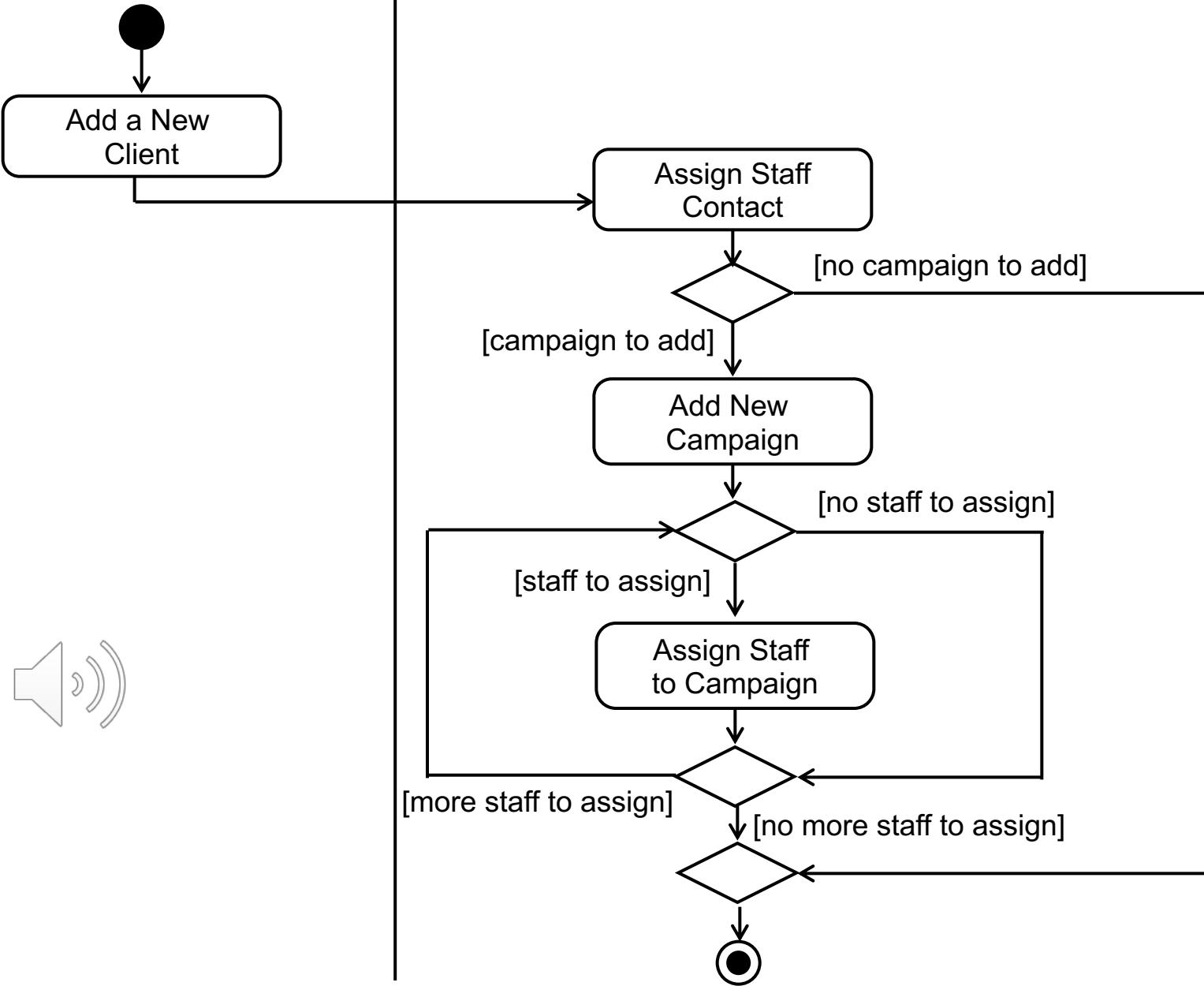
Drawing Activity Diagrams

- ▶ Are all the activities carried out by the same person, organisation or department?
- ▶ If not, then add swimlanes to show the responsibilities
- ▶ Name the swimlanes
- ▶ Show each activity in the appropriate swimlane



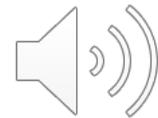
Administrator

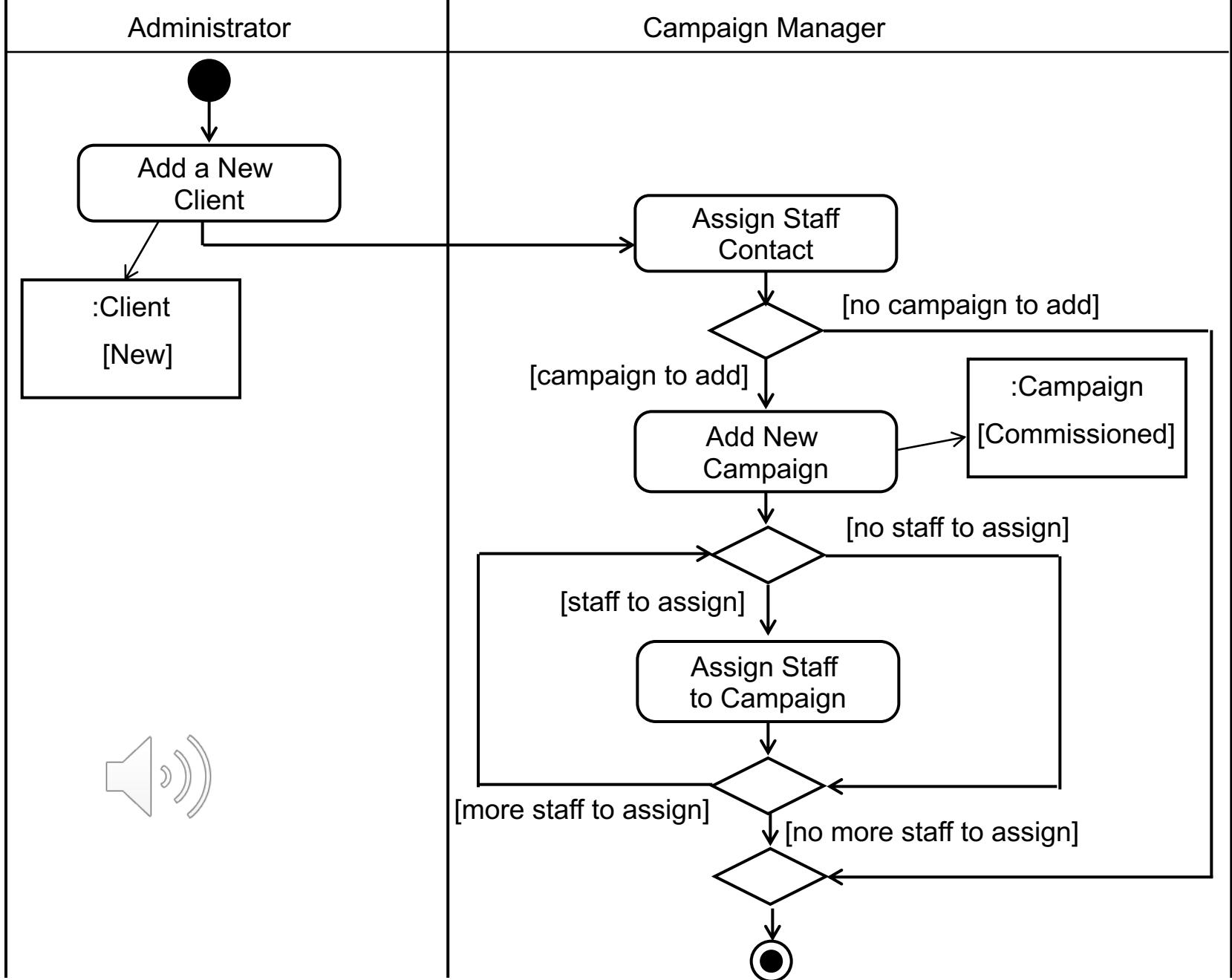
Campaign Manager



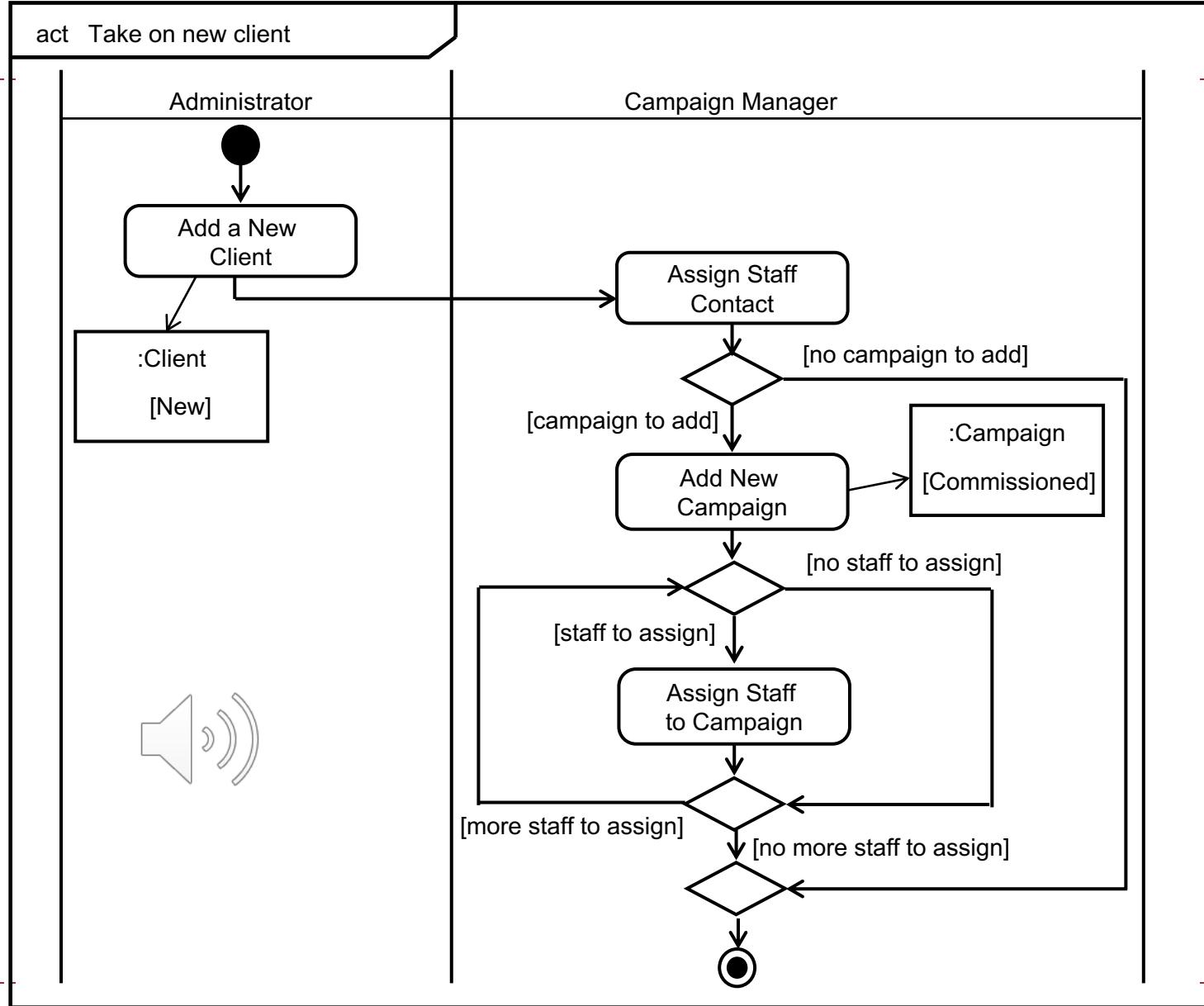
Drawing Activity Diagrams

- ▶ Are there any object flows and objects to show?
 - ▶ these can be documents that are created or updated in a business activity diagram
 - ▶ these can be object instances that change state in an operation or a use case
- ▶ Add the object flows and objects





► When printed or copied a frame is used

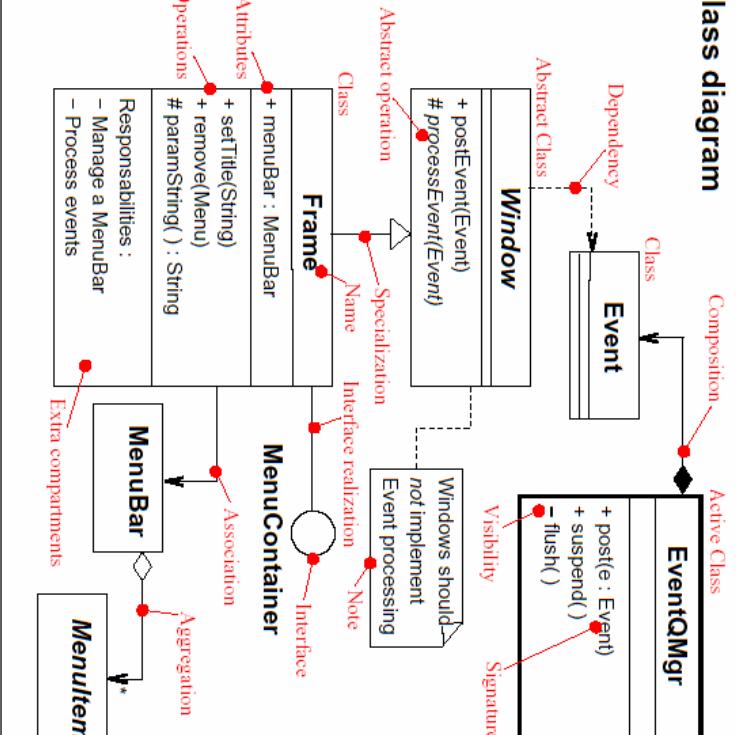


In this lecture we have looked at

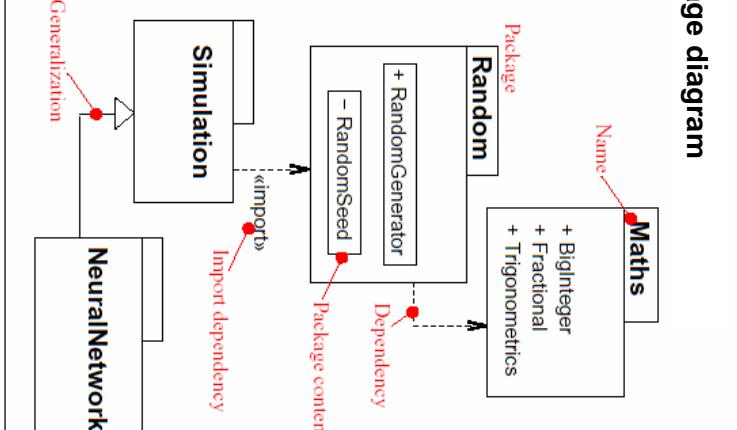
- Models and Diagrams in general
- Some diagram examples
 - Use-case diagrams
 - Class diagrams
 - Package diagrams
 - Sequence diagrams
 - Collaboration diagrams
 - Type diagrams
- ▶ Activity Diagrams in detail
- ▶ We will revisit these diagrams again during the rest of the module!



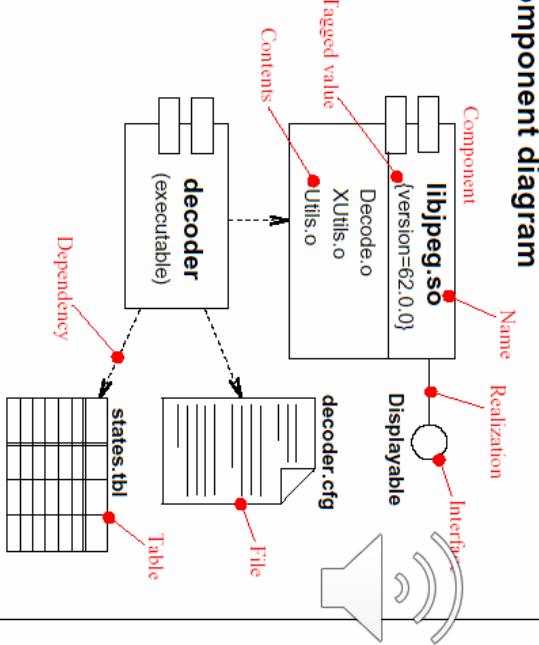
Class diagram



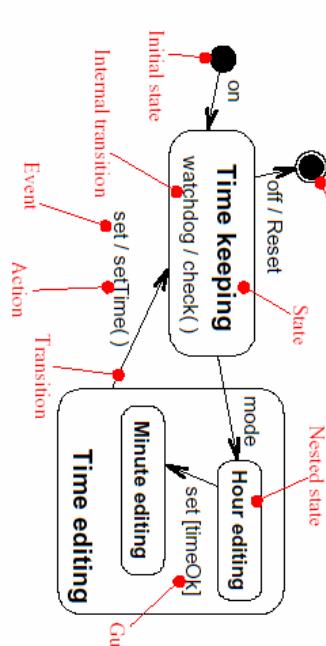
Package diagram



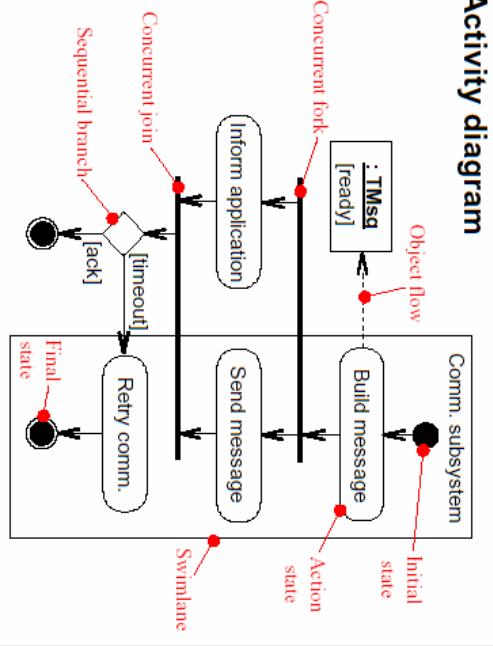
Component diagram



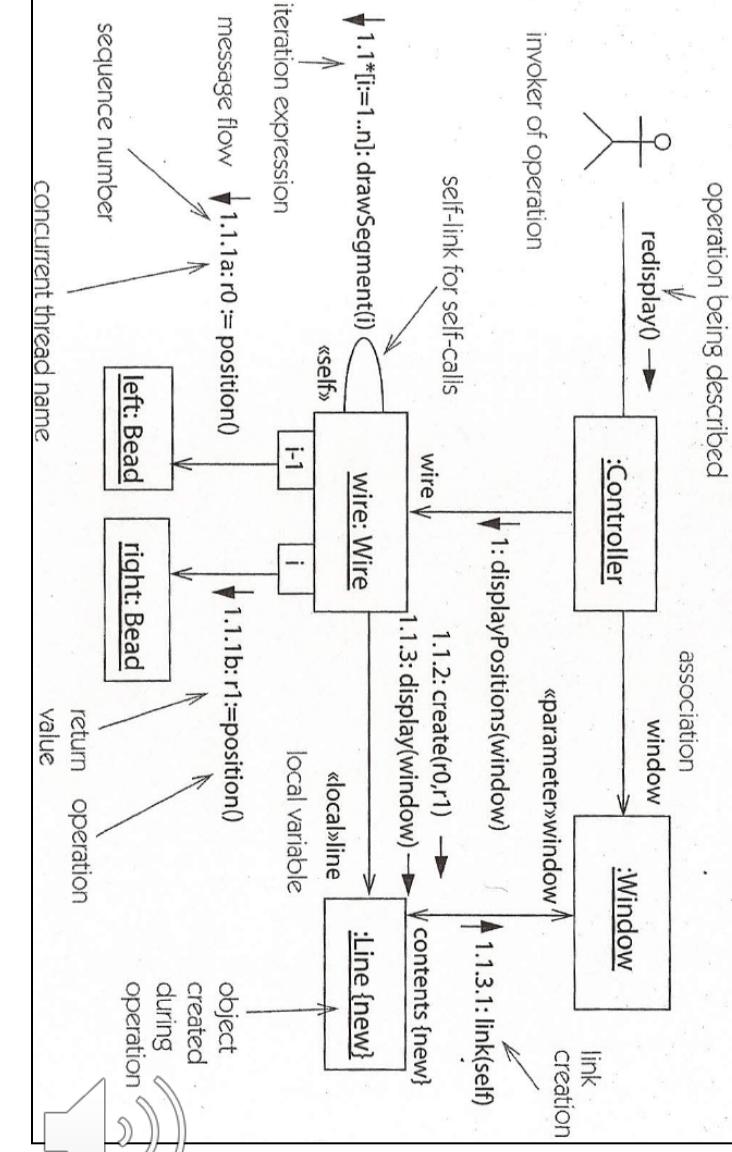
State diagram



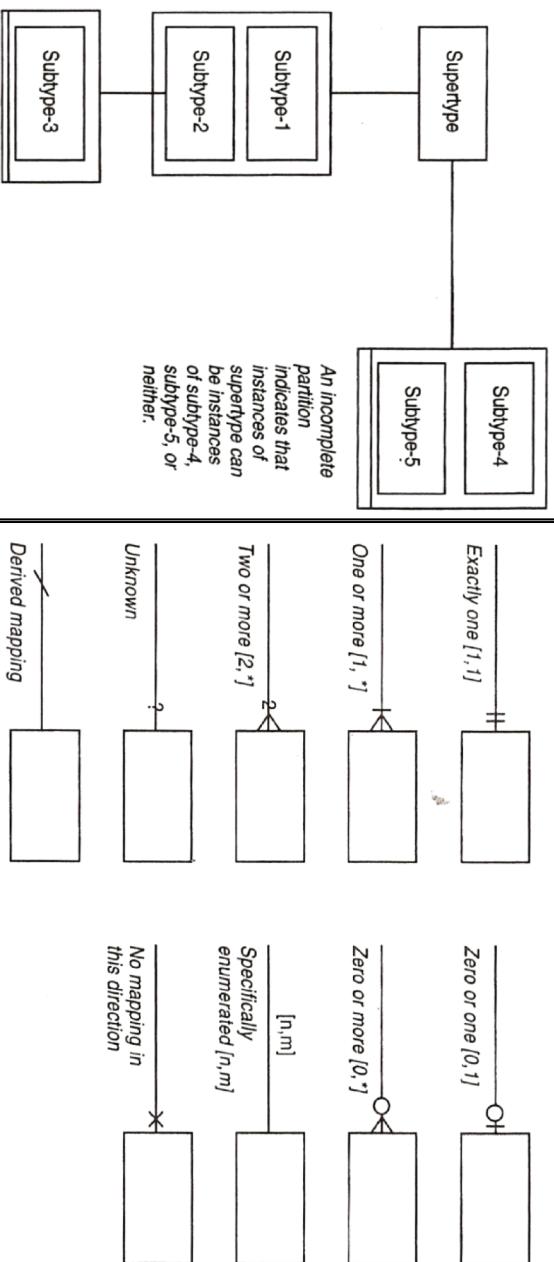
Activity diagram



Collaboration diagram

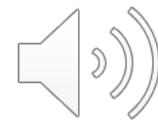


Type diagrams



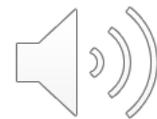
Class exercise (15 mins)

- ▶ Draw an activity diagram for the following use-case:
 - ▶ "Withdraw money from a bank account through an ATM"
- ▶ Add swimlanes to your diagram to show the main actors
- ▶ You could use Microsoft Visio to create the Diagram.



Some Hints for the class exercises

- ▶ You have three involved classes of the activity, they are Customer, ATM and Bank.
- ▶ Work out the activities for each class.



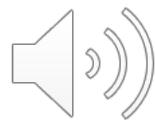
Hints for the class exercises

- ▶ You have three involved classes of the activity, they are Customer, ATM and Bank.
- ▶ The customer's activities are “insert card”, “enter pin”, “enter amount”, “take money from slot” and “take card”.
- ▶ The ATM Machine's activities are “show balance”, and “Eject card”.
- ▶ The Bank's activities are “authorize/verify PIN”, “check account balance” and “debit account”



Further reading

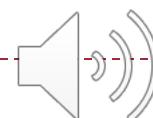
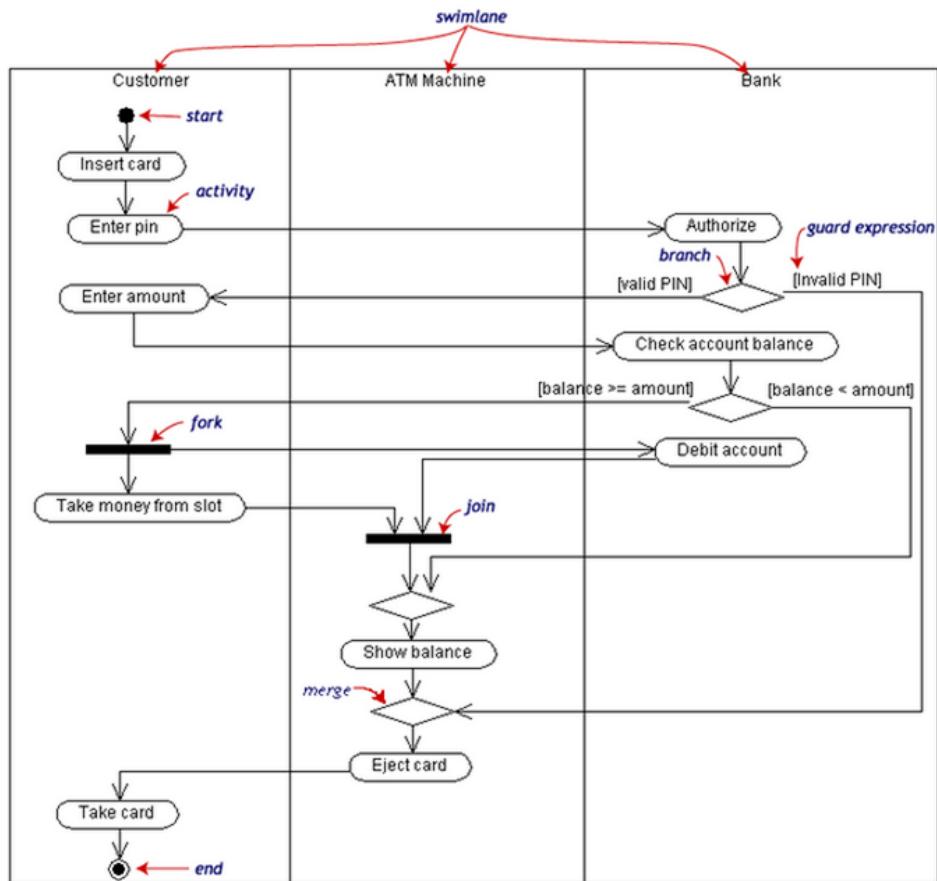
- ▶ Package diagrams – pages 120 and 244 Bennett
- ▶ Chapter 7 Bennett – Class diagrams
- ▶ Chapter 9 Bennett – Object interaction
- ▶ Booch, Rumbaugh and Jacobson (1999)
- ▶ Bennett, Skelton and Lunn (2005)



"Withdraw money from a bank account through an ATM."

The three involved classes (people, etc.) of the activity are **Customer**, **ATM Machine**, and **Bank**. The process begins at the black start circle at the top and ends at the concentric white/black stop circles at the bottom. The activities are rounded rectangles.

[Hide image](#)



Requirements engineering

- Requirements elicitation
- Requirements modelling
- Use case analysis
 - Modelling technique: use-case diagrams and descriptions
- Requirements validation (will do in the class this week)

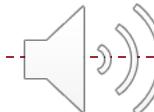


CE202 Software Engineering, Autumn term

Dr Cunjin Luo, School of Computer Science and Electronic Engineering, University of Essex

Requirements

- ▶ Requirements engineering: making decisions about “what are we going to build?”
 - ▶ Focus on WHAT (the user needs), not HOW (to achieve it)
- ▶ **Requirement:**
 - ▶ An expression of desired behaviour (‘feature’)
or
 - ▶ A constraint which must be satisfied
- ▶ A requirement exists either because —
 - ▶ The product demands certain functions
 - ▶ The client demands certain features
 - ▶ External (legal, organizational, financial etc.) demands
 - ▶ For example: the legal requirement from Website to be accessible to users with sight disabilities



Reminder: why Are Requirements Important?

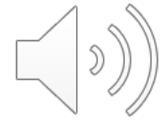
- ▶ Basili and Perricone report
 - ▶ 48% of the faults observed in a medium-scale software project were attribute to “incorrect or misinterpreted functional specification or requirements”
- ▶ Top factors that caused projects to fail
 - ▶ Incomplete requirements
 - ▶ Lack of user involvement
 - ▶ Unrealistic expectations
 - ▶ ...
- ▶ Requirements errors can be expensive if not detected early
 - ▶ It is much cheaper to ‘fix’ the requirements than the implementation





Types of requirements

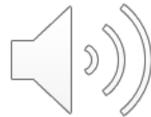
- ▶ **Functional requirement:** describes the functionality of the system: what it ‘does’
 - ▶ ‘The program should be able to send email’
 - ▶ ‘The program must guide the Mars Rover from point A to point B’
- ▶ **Non-functional requirement:** all other requirements
 - ▶ In particular: requirements that contribute to the internal qualities of software (safety, security, maintainability, ...)
 - ▶ Examples:
 - ▶ Hardware requirements (‘it must run on mobile phones with 1 MB memory’)
 - ▶ Operational environment requirements (‘it must run on the Android operating system’)
 - ▶ Process requirements (‘it must conform to level 3 in the CMM or above’)
 - ▶ Human-Computer Interaction requirements (‘it must use the Windows Vista native graphical user interface’)
 - ▶ Design and architectural requirements (‘it must be structured by the Client-Server architectural style’)
- ▶ **Constraints or pseudo-constraints:** legal, organizational, financial etc. Constraints
 - ▶ Example: “The Data Protection Act requires all personal data to be removed after client’s relationship with the company ends”



Requirement elicitation

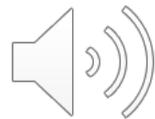
User Requirements

- ▶ Need to understand how the organization operates at present
- ▶ What are the problems with the current system?
- ▶ What are the requirements users have of a new system that are not in the current system?



The *Agile* approach

- ▶ Advocates of Agile methods focus on developing the new system and not on extensive analysis of the existing system
- ▶ In the Agile Manifesto they state that they value working software over comprehensive documentation



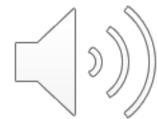
Requirements elicitation

- ▶ **Purpose:** To gather requirements from clients
- ▶ **Challenges:** bridging the gap between non-technical (users) and technical (software engineers) people
 - ▶ Customers do not always understand/aware of their needs
 - ▶ Customers have trouble articulating their needs
 - ▶ Customers do not like to be observed
 - ▶ Inconsistencies
- ▶ **Stakeholders**
 - ▶ **Clients:** pay for the software to be developed
 - ▶ **Customers:** buy the software after it is developed
 - ▶ **Users:** use the system
 - ▶ **Domain experts:** familiar with the problem
 - ▶ **Market researchers:** determine future trends & potential customers
 - ▶ **Lawyers or auditors**
 - ▶ **Software engineers** or other technology experts



Fact Finding Techniques

- ▶ Background Reading
- ▶ Interviewing
- ▶ Observation
- ▶ Document Sampling
- ▶ Questionnaires



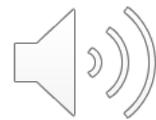
Background Reading

- ▶ Aim is to understand the organization and its business objectives
- ▶ Includes:
 - ▶ reports
 - ▶ organization charts
 - ▶ policy manuals
 - ▶ job descriptions
 - ▶ documentation of existing systems



Interviewing

- ▶ Aim is to get an in-depth understanding of the organization's objectives, users' requirements and people's roles
- ▶ Includes:
 - ▶ managers to understand objectives
 - ▶ staff to understand roles and information needs
 - ▶ customers and the public as potential users



Interviewing

► Advantages:

- ▶ personal contact allows the interviewer to respond adaptively to what is said
- ▶ it is possible to probe in greater depth
- ▶ if the interviewee has little or nothing to say, the interview can be terminated



Interviewing

► Disadvantages:

- ▶ can be time-consuming and costly
- ▶ notes must be written up or tapes transcribed after the interview
- ▶ can be subject to bias
- ▶ if interviewees provide conflicting information this can be difficult to resolve later



Observation

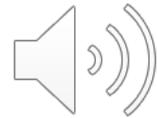
- ▶ Aim is to see what really happens, not what people say happens
- ▶ Includes:
 - ▶ seeing how people carry out processes
 - ▶ seeing what happens to documents
 - ▶ obtaining quantitative data as baseline for improvements provided by new system
 - ▶ following a process through end-to-end
- ▶ Can be open-ended or based on a schedule



Observation

► Appropriate situations:

- ▶ when quantitative data is required
- ▶ to verify information from other sources
- ▶ when conflicting information from other sources needs to be resolved
- ▶ when a process needs to be understood from start to finish



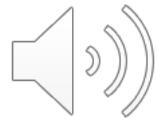
Document Sampling

- ▶ Aims to find out the information requirements that people have in the current system
- ▶ Also aims to provide statistical data about volumes of transactions and patterns of activity
- ▶ Includes:
 - ▶ obtaining copies of empty and completed documents
 - ▶ counting numbers of forms filled in and lines on the forms
 - ▶ screenshots of existing computer systems



Questionnaires

- ▶ Aims to obtain the views of a large number of people in a way that can be analysed statistically
- ▶ Includes:
 - ▶ postal, web-based and email questionnaires
 - ▶ open-ended and closed questions
 - ▶ gathering opinion as well as facts



YES/NO Questions

Do you print reports from the existing system?
(Please circle the appropriate answer.)

YES

NO

10

Multiple Choice Questions

How many new clients do you obtain in a year?
(Please tick one box only.)

a) 1-10

11

b) 11-20

c) 21-30

d) 31 +

Scaled Questions

How satisfied are you with the response time of the stock update?
(Please circle one option.)

1. Very satisfied

2. Satisfied

3. Dissatisfied

4. Very

dissatisfied

12

Open-ended Questions

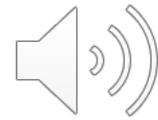
What additional reports would you require from the system?

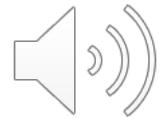


Questionnaires

► Appropriate situations:

- ▶ when views of large numbers of people need to be obtained
- ▶ when staff of organization are geographically dispersed
- ▶ for systems that will be used by the general public and a profile of the users is required



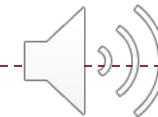


Requirements modelling & specification

Use cases
Object-oriented analysis

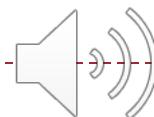
Requirements languages and notations

- ▶ Purpose: to represent the requirements in an explicit, structured form
- ▶ Input: a report (in natural language) on elicited requirements
- ▶ Output:
 - ▶ **Formal requirements specifications languages** use mathematical languages to represent or model functional specifications
 - ▶ (Logic, Z, OCL)
 - ▶ Statecharts
 - ▶ **Informal** requirement specification notations
 - ▶ Use-case diagrams
 - ▶ Class diagrams
 - ▶ Type diagrams
 - ▶ (sequence diagrams, interaction diagrams)



Requirements Specification

- ▶ Should include (as a minimum):
 - ▶ Problem Definition – what is the problem being addressed?
 - ▶ Viewpoint Structure – what is the scope and who is involved?
 - ▶ Functional Requirements – what should the system do?
 - ▶ Non-functional requirements – how should the system behave?
- ▶ Standards for Requirements Specification:
 - ▶ Software requirements specification IEEE 830
 - ▶ IEEE Recommended Practice for Software Requirements Specifications
 - ▶ http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=392555&sortType%3Dasc_p_Sequence%26filter%3DAND%28p_Publication_Number%3A3114%29



Functional and non-functional requirements

- ▶ The requirements specification should contain a **list** of functional and non-functional requirements
- ▶ A **functional requirement** defines **a function of a system** or its component. A function is described as a set of inputs, the behavior, and outputs.
 - ▶ Eg. The system shall print an invoice for the work completed
- ▶ A **non-functional requirement** is a requirement that specifies criteria that can be used to **judge the operation of a system**, rather than specific behaviors
 - ▶ Eg. the system should be implemented in Java
- ▶ All requirements should be **traceable** to their source, and if possible **prioritized** by their level of importance





Requirements elicitation: Use case diagrams

Drawing Use Case Diagrams

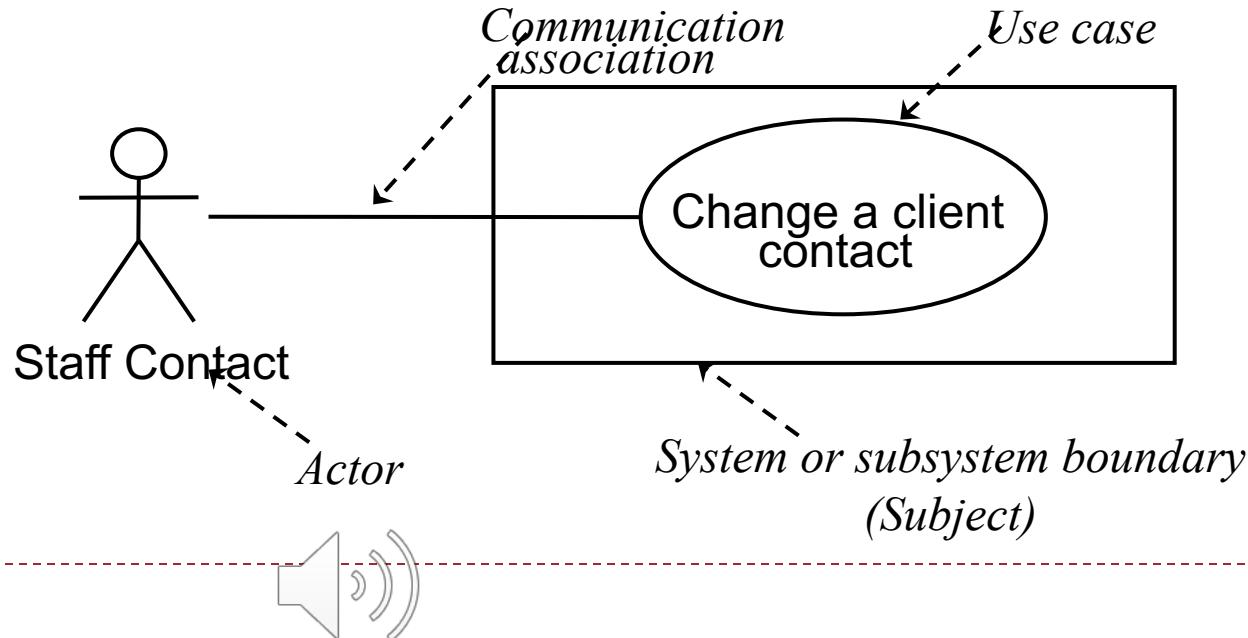
► Purpose

- ▶ document the functionality of the system from the users' perspective
- ▶ document the scope of the system
- ▶ document the interaction between the users and the system using supporting use case descriptions (behaviour specifications)
- ▶ Do NOT specify the flow of the implementation
- ▶ provide a starting point for capturing and analysing requirements
- ▶ **One of the first activities that you should do**



Modelling requirements technique: use-case diagrams

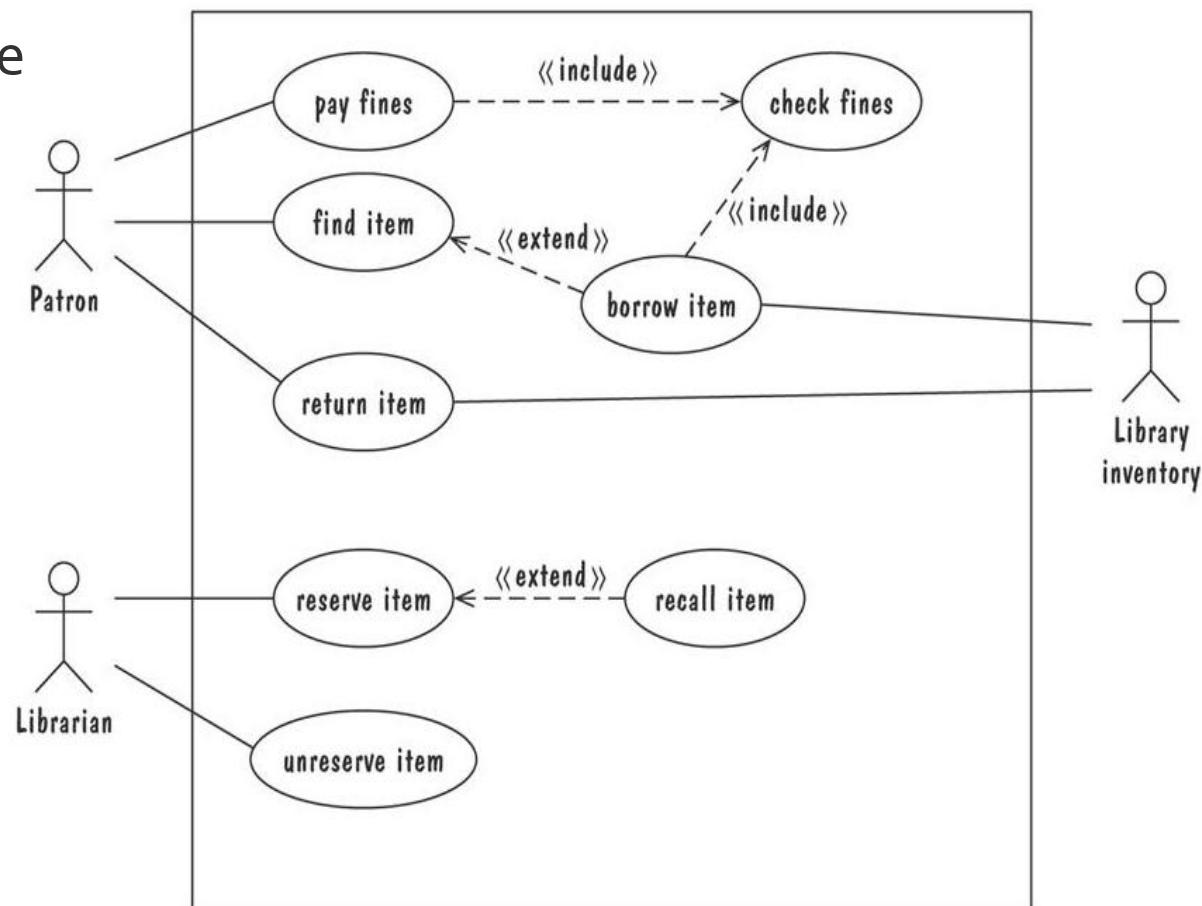
- ▶ Represent the results of use-case analysis
- ▶ Vocabulary:
 - ▶ Box: system boundary
 - ▶ Stick figures (outside the box): actors, human and systems
 - ▶ Oval (inside the box): a use case
 - ▶ Represents some major required functionality and its variant
 - ▶ Heuristic: individual menu item often corresponds to a use case
 - ▶ Line between actor—use case : the actor participates in the use case



Use -case diagrams: example

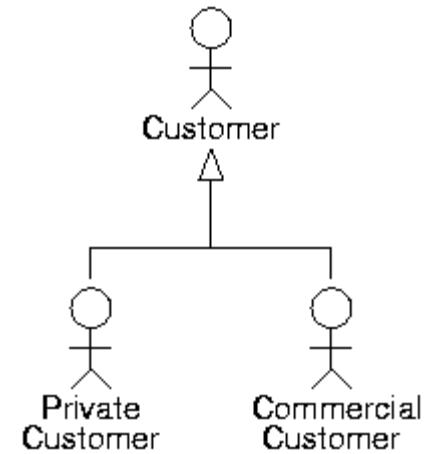
► Library use cases:

- ▶ Borrowing a book
- ▶ Returning a borrowed book
- ▶ Paying a library fine
- ▶ Reserving items
- ▶ Unreserving items



Actors

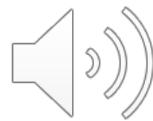
- ▶ A coherent set of roles that users of use cases play when they interact with use cases
 - ▶ Typically represents the role that a human, hardware device or another system plays with the system
 - ▶ Actors are not part of the system
 - ▶ As an actor is a class, it can be generalized
- ▶ Examples:
 - ▶ *Registrar*: maintain the curriculum
 - ▶ *Billing System*: receive billing information from registration



Notation of Use Case Diagrams

▶ Use cases

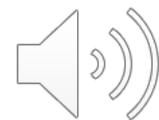
- ▶ drawn as ellipses with a name in or below each ellipse
- ▶ describe a sequence of actions that the system performs to achieve an observable result of value to an actor
- ▶ the name is usually an active verb and a noun phrase



Notation of Use Case Diagrams

► Communication associations

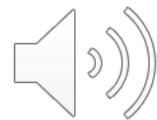
- ▶ line drawn between an actor and a use case
- ▶ represent communication link between an instance of the use case and an instance of the actor. These are **NOT** labelled.



Notation of Use Case Diagrams

► Subjects (subsystems)

- ▶ drawn as a rectangle around a group of use cases that belong to the same subject
- ▶ in a CASE tool, use cases for different subjects are usually placed in separate use case diagrams



Notation of Use Case Diagrams

- ▶ Dependencies
 - ▶ **Extend** and **Include** relationships between use cases
 - ▶ shown as stereotyped dependencies
 - ▶ stereotypes are written as text strings in guillemets: «extend» and «include»
- ▶ Generalisation
 - ▶ To show that one use case is a **type** of another use case
 - ▶ Show which **specific** use case is the **generalised** type and which are the types
 - ▶ Use <>generalise<> or inheritance notation



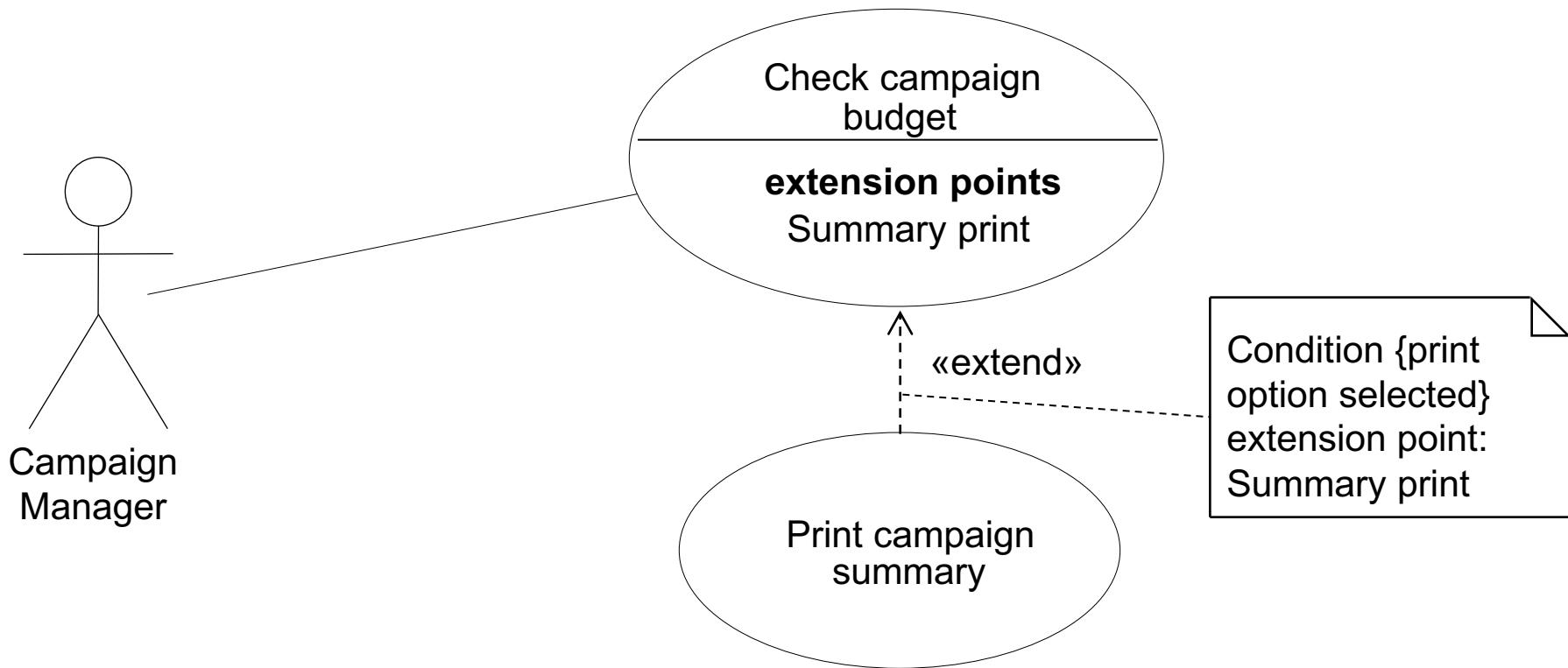
Notation of Use Case Diagrams

► Extend relationship

- ▶ used when one use case provides additional functionality that **may** be required in another use case
- ▶ there may be multiple ways of extending a use case, which represent variations in the way that actors interact with the use case
- ▶ extension points show when the extension occurs
- ▶ a condition can be placed in a note joined to the dependency arrow (Note that it is not put in square brackets, unlike conditions in other diagrams.)



Extend relationship



Arrow direction goes from sub-use-case towards the larger use-case



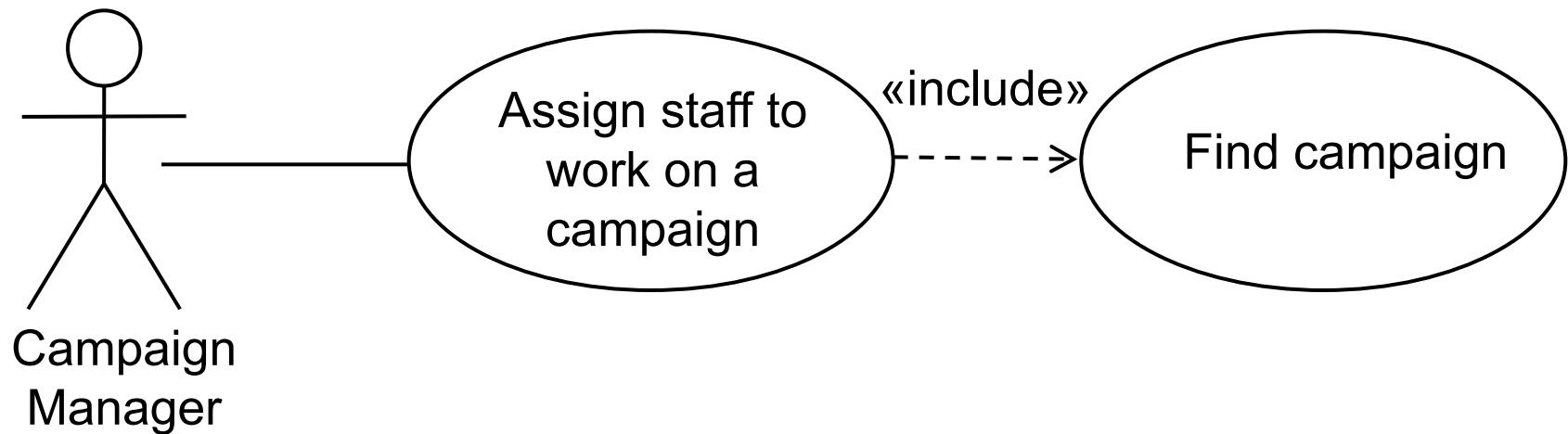
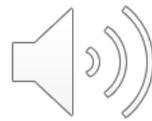
Notation of Use Case Diagrams

► Include relationship

- ▶ used when one use case **always** includes the functionality of another use case
- ▶ a use case may include more than one other
- ▶ can be used to separate out a sequence of behaviour that is used in many use cases
- ▶ should not be used to create a hierarchical functional decomposition of the system



Include Relationship



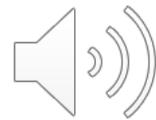
Arrow direction goes from the larger use-case towards sub-use-case
(different to <<extend>>)

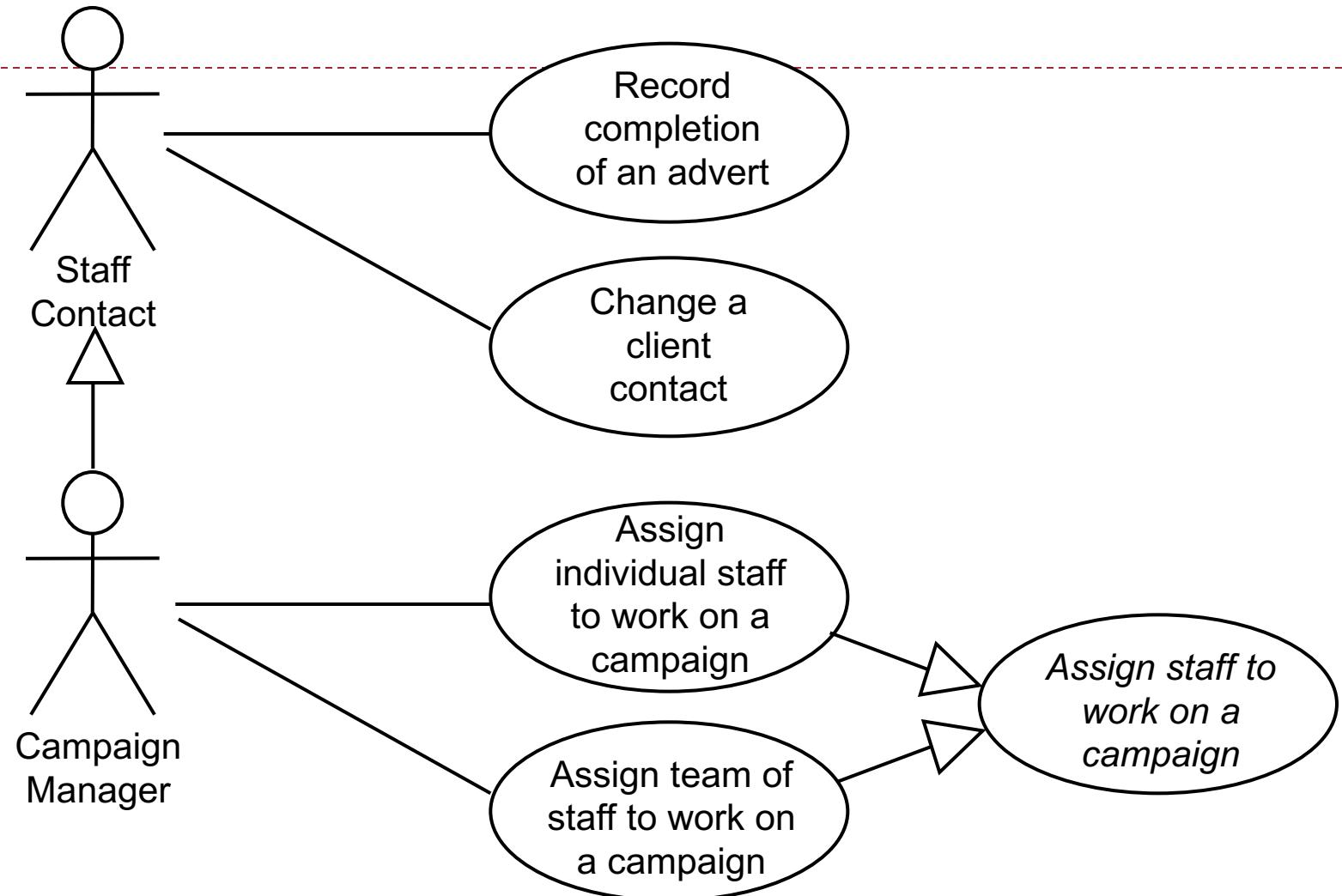


Notation of Use Case Diagrams

► Generalization

- ▶ shows that one use case provides all the functionality of the more general use case and some additional functionality
- ▶ shows that one actor can participate in all the associations with use cases that the more general actor can plus some additional use cases





Arrow direction goes from the child use-case towards parent use-case



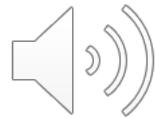
Summary of relations between use cases

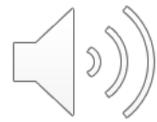
- ▶ <<include>>: one use case incorporates the behaviour of another use case (eg. where behaviour is used frequently in a number of use cases, to avoid repetition)
 - ▶ ‘borrow item’ includes ‘check fine’
 - ▶ ‘pay fine’ include ‘check fine’
- ▶ <<extend>>: One use case adds to another (ie. provides additional/optional functionality)
 - ▶ ‘Recall Item’ extends ‘Reserve Item’ : It involves reserving an item and adds additional steps to it
- ▶ <<generalize>>: One use case is special case of (is-kind-of) another
 - ▶ ‘Enter book details’ is a generalization of ‘scan book details’
 - ▶ ‘Enter book details’ is a generalization of ‘type book number’



Drawing Use Case Diagrams

- ▶ Identify the actors and the use cases
- ▶ Prioritize the use cases
- ▶ Develop each use case, starting with the priority ones, writing a description for each
- ▶ Add structure to the use case model: generalization, include and extend relationships and subsystems





Requirements elicitation: Use case descriptions



Use Case Descriptions

- ▶ Very similar to a scenario
- ▶ Based on predicted uses of the new system
- ▶ Should aim to capture use-case descriptions for all scenarios
- ▶ Ideally there should be a **use-case description** for every use case bubble in the **use-case diagrams**
- ▶ Each use-case description may only be a simple paragraph ...

Assign staff to work on a campaign

- ▶ *The campaign manager wishes to record which staff are working on a particular campaign. This information is used to validate timesheets and to calculate staff year-end bonuses.*



Requirement analysis technique: scenario analysis

- ▶ Jennifer is standing in front of the lift on floor 3 in the Computer Science building.
- ▶ She presses a lift request button for upward direction.
- ▶ The lift arrives and the doors open.
- ▶ Jennifer enters the lift and presses the floor request button for floor 5A. The doors close.
- ▶ The lift travels to floor 5A and the doors open.
- ▶ Jennifer leaves the lift.



Use Case Descriptions

- ▶ It is more normal that it is a step-by-step breakdown of interaction between actor and system

Assign staff to work on a campaign

Actor Action

1. The actor enters the client name.
3. Selects the relevant campaign.
5. Highlights the staff members to be assigned to this campaign.

System Response

2. Lists all campaigns for that client.
4. Displays a list of all staff members not already allocated to this campaign.
6. Presents a message confirming that staff have been allocated.

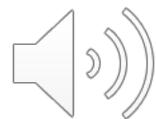
Alternative Courses

Steps 1–3. The actor knows the campaign name and enters it directly.



Use Case Descriptions

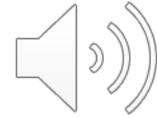
- ▶ Many projects use templates. Typical headings might be:
 - ▶ name of use case
 - ▶ pre-conditions
 - ▶ post-conditions
 - ▶ actors involved
 - ▶ purpose
 - ▶ description
 - ▶ alternative courses
 - ▶ errors



Use-case description (for illustration only)

- ▶ **Name of use case:** Lift operation
- ▶ **Pre-conditions:**
 - ▶ User standing in front of lift
 - ▶ Lift request button is enabled
- ▶ **Post-conditions:**
 - ▶ Lift request button is enabled on that floor
- ▶ **Actor:** Lift user
- ▶ **Purpose:** description of lift operation
- ▶ **Description:**
 - ▶ User presses lift request button
 - ▶ The direction indicators highlight the requested direction of travel.
Disable lift request button on that floor.
 - ▶ Lift travels to the users floor
 - ▶ Etc

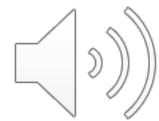




Requirements elicitation: Prototyping

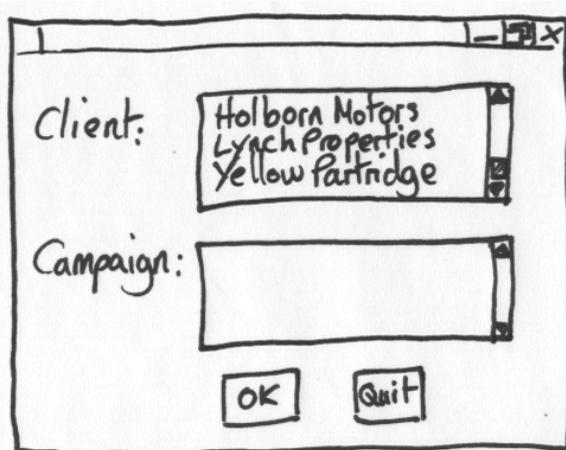
Prototyping

- ▶ Use case modelling can be supported with prototyping
- ▶ Prototypes can be used to help elicit requirements
- ▶ Prototypes can be used to test out system architectures based on the use cases in order to meet the non-functional requirements

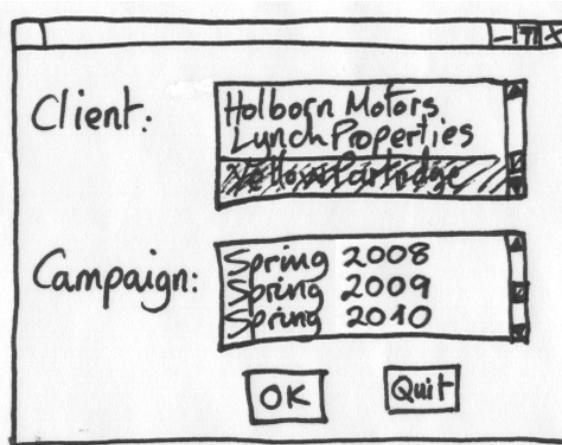


Prototyping

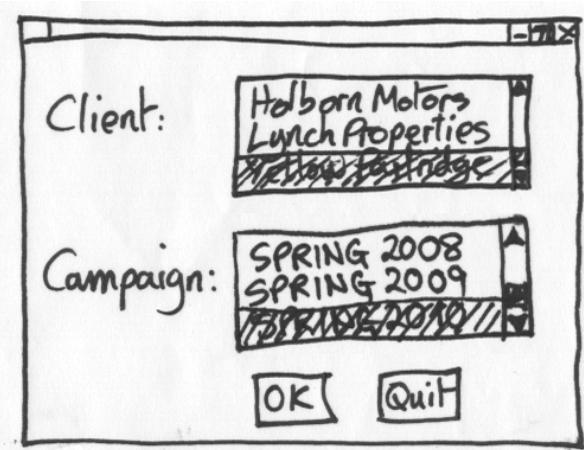
- For user interface prototypes, storyboarding can be used with hand-drawn designs



Dialogue initialized.



User selects Client. Campaigns listed.

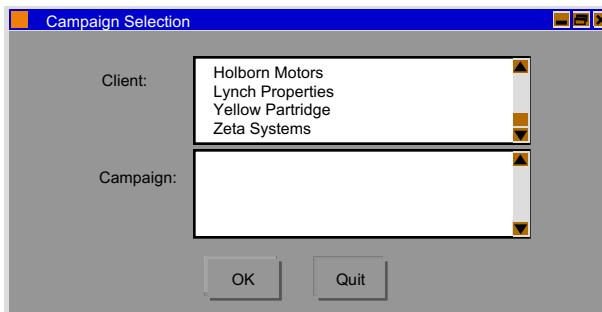


User selects Campaign.



Prototyping

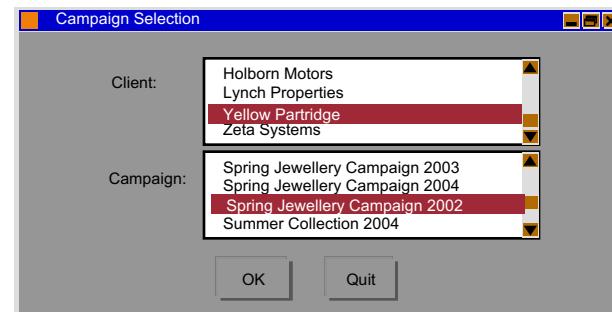
- ▶ User interface prototypes can be implemented using languages other than the one that the system will be developed in



Dialogue initialized.



User selects Client. Campaigns listed.



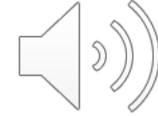
User selects Campaign.



In this lecture we have looked at:

- Requirements elicitation
- Requirements modelling & specification
 - Formal and informal specifications
 - Types of requirements: functional & non-functional
 - The requirements specification
- Use case analysis
 - ▶ Scenarios and use case descriptions
 - ▶ Modelling technique: use-case diagrams
 - ▶ The link between use-case diagrams and use-case descriptions
 - ▶ Use of prototyping
- Requirements validation – class exercise (next)





Class exercise: Requirements validation

Requirements verification & validation

- ▶ Verification: compare one document to another
- ▶ Validation: check requirements with customer



Requirements Validation

- ▶ A critical step in the development process
- ▶ Requirements validation criteria:
 - ▶ Correctness:
 - ▶ The requirements represent the client's view.
 - ▶ Completeness:
 - ▶ All possible scenarios, in which the system can be used, are described, including exceptional behavior by the user or the system
 - ▶ Consistency:
 - ▶ There are functional or nonfunctional requirements that contradict each other
 - ▶ Realism:
 - ▶ Requirements can be implemented and delivered
 - ▶ Traceability:
 - ▶ Each system function can be traced to a corresponding set of functional requirements



Example: requirement from a word-processor

- ▶ Requirements should be clear, unambiguous, understandable

Selecting is the process of designating areas of the document that you want to work on. Most editing and formatting actions require two steps: first you select what you want to work on, such as text or graphics; then you initiate the appropriate action.

- ▶ Question: does a selected area need to be continuous?



Example: requirement from a real safety-critical system

- ▶ Precise, unambiguous, clear

The message must be triplicated. The three copies must be forwarded through three different physical channels. The receiver accepts the message on the basis of a two-out-of-three voting policy.

- ▶ **Question:** can a message be accepted as soon as we receive 2 out of 3 identical copies of the message or do we need to wait for receipt of the 3rd?



Example: requirement from a word-processor

► Consistent

The whole text should be kept in lines of equal length. The length is specified by the user. Unless the user gives an explicit hyphenation command, a carriage return should occur only at the end of a word.

► **Question:** What if the length of a word exceeds the length of the line?



Techniques of validating requirements

Validation

Walkthroughs
Readings
Interviews
Reviews
Checklists
Models to check functions and relationships
Scenarios
Prototypes
Simulation
Formal inspections

Verification

Cross-referencing
Simulation
Consistency checks
Completeness checks
Check for unreachable states of transitions
Model checking
Mathematical proofs

Checking



Exercise (1)

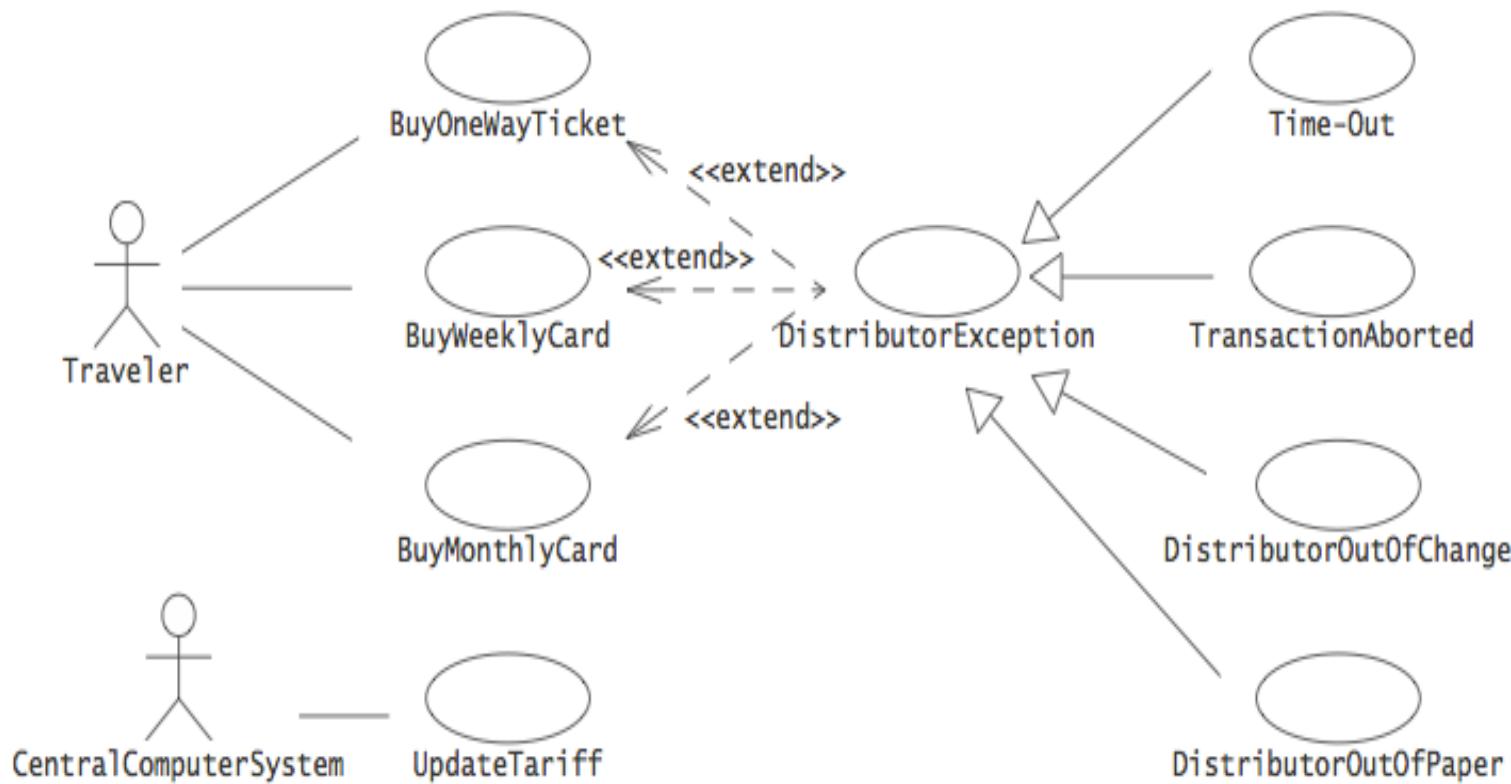
- ▶ Below are examples of non-functional requirements. Specify which of these requirements can be validated and which cannot.
 - ▶ “The system must be usable.”
 - ▶ “The system must provide visual feedback to the user within 1 second of issuing a command.”
 - ▶ “The availability of the system must be above 95%.”
 - ▶ “The user interface of the new system should be similar enough to the old system such that users familiar with the old system can be easily trained to use the new system.”



Exercise (2)

- ▶ Draw a use case diagram for a ticket distributor for a train system. The system includes two actors: a traveller, who purchases different types of tickets, and a central computer system, which maintains a reference database for the tariff.
- ▶ Use cases should include: BuyOneWayTicket, BuyWeeklyCard, BuyMonthlyCard, UpdateTariff.
- ▶ Also include the following exceptional cases: Time-Out (i.e., traveler took too long to insert the right amount), TransactionAborted (i.e., traveler selected the cancel button without completing the transaction), DistributorOutOfChange, and DistributorOutOfPaper.





Exercise 3: Divide by functional/nonfunctional

The allocation of staff to production lines should be mostly automated. A process will be run once a week to carry out the allocation based on the skills and experience of operatives. Details of holidays and sick leave will also be taken into account. A first draft Allocation List will be printed off by 12.00 noon on Friday for the following week. Only staff in Production Planning will be able to amend the automatic allocation to fine-tune the list. Once the amendments have been made, the final Allocation List must be printed out by 5.00pm. The system must be able to handle allocation of 100 operatives at present, and should be capable of expansion to handle double that number.



Examples of functional requirements are:

- the need for a process to be run that allocates staff to lines based on their skills and experience, holidays and sick leave;
- printing out an allocation list;
- amending the allocation list.

Examples of non-functional requirements include:

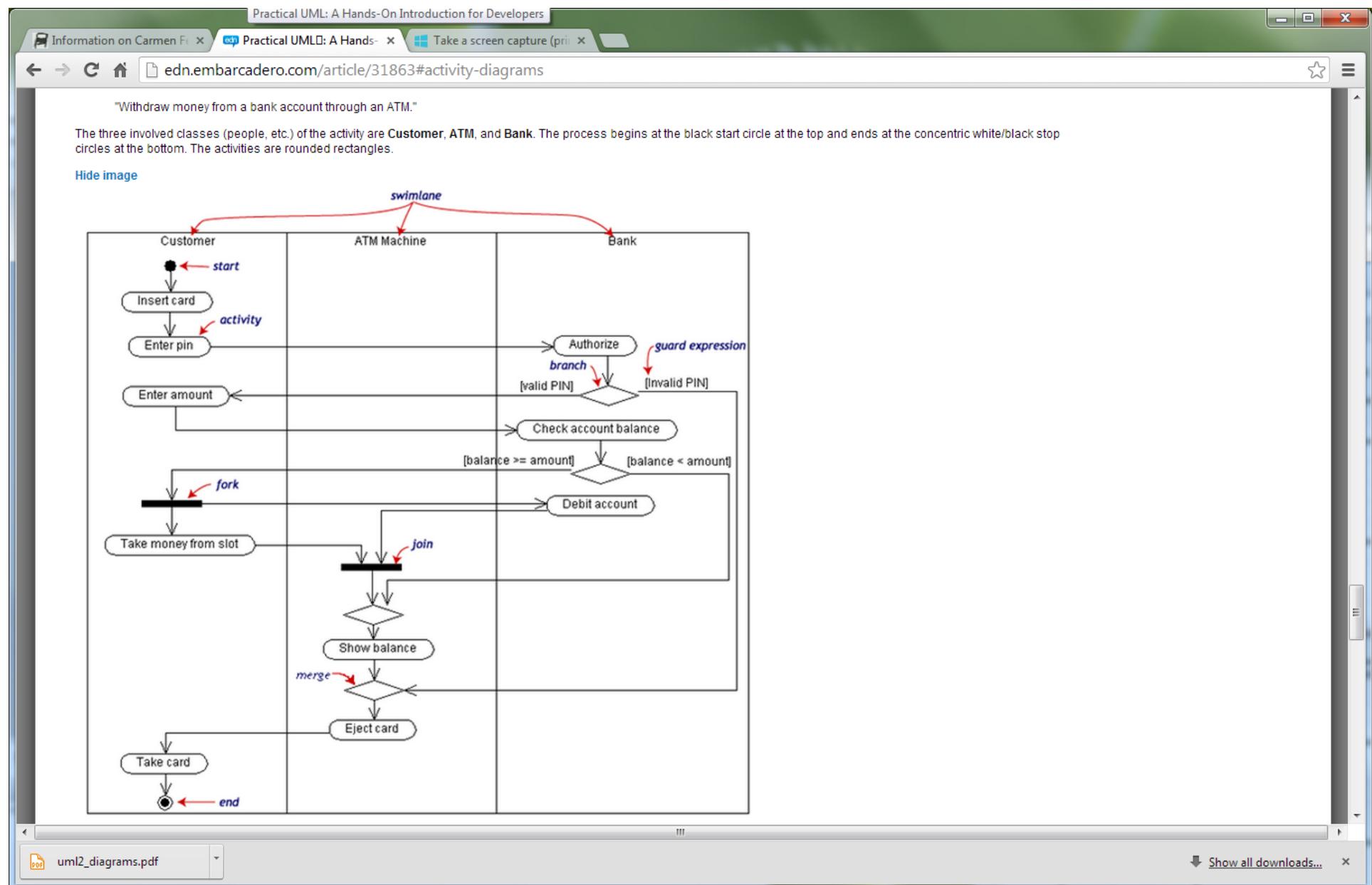
- printing the allocation list by 12.00 noon;
- the need to handle 200 operatives' details.



Further reading

- ▶ Basili and Perricone (1984). Software Errors and Complexity: An Empirical Investigation. Communications of the ACM, pp42-52, Jan 1984.
- ▶ Brooks, Frederick P. (1987). *No Silver Bullet: Essence and Accidents of Software Engineering*. (Reprinted in the 1995 edition of *The Mythical Man-Month*)
- ▶ SUNA -
http://www.jisc.ac.uk/media/documents/programmes/eframework/scenario_based_design_chris_fowler.pdf





Principles of Object-Orientation

- Encapsulation
- Inheritance
- Polymorphism

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

Object-orientation

- ▶ So far we have briefly touched on how analysis and design (and implementation) can make use of objects
 - ▶ Eg. objects in activity diagrams, class diagrams, etc
- ▶ In this lecture we will explore common object-oriented principles
- ▶ As we get more into UML we will make more use of objects



Advantages of O-O

- ▶ Can save effort
 - ▶ Reuse of generalized components cuts work, cost and time
- ▶ Can improve software quality
 - ▶ Encapsulation increases modularity
 - ▶ Sub-systems less coupled to each other
 - ▶ Better translations between analysis and design models and working code
 - ▶ Objects are good for modelling what happens in the real world
 - ▶ Can be used throughout the software lifecycle ie.
requirements -> design -> implementation -> testing



OO Analysis & Design: mechanisms of abstraction

- ▶ Fundamentally: the same abstraction mechanisms as object-oriented programming:
 - ▶ Encapsulation: classes and objects
 - ▶ Other possible modularization techniques: interfaces, packages/namespaces
 - ▶ Inheritance
 - ▶ Generalization/specialization
 - ▶ Subtyping
 - ▶ Subclassing
 - ▶ Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding

Encapsulation

Objects

An object is:

“an abstraction of something in a problem domain,
reflecting the capabilities of the system to

- ▶ keep information about it,
- ▶ interact with it,
- ▶ or both.”

Coad and Yourdon (1990)



Objects

“Objects have state, behaviour and identity.”

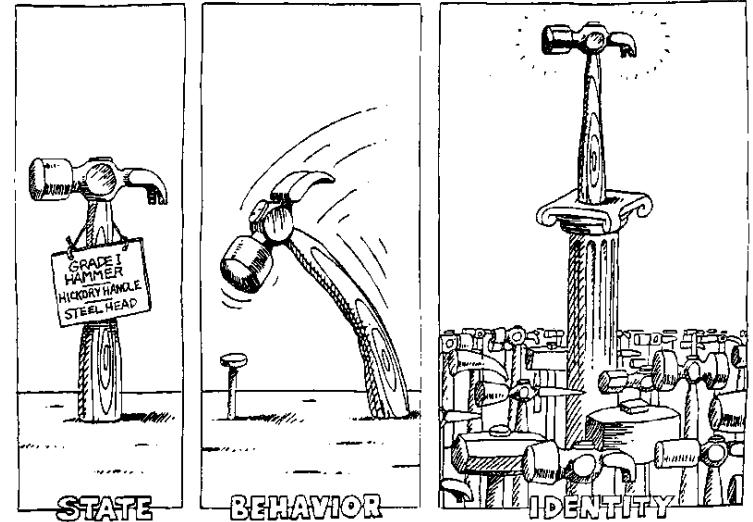
Booch (1994)

- ▶ *State*: the condition of an object at any moment, affecting how it can behave
- ▶ *Behaviour*: what an object can do, how it can respond to events and stimuli
- ▶ *Identity*: each object is unique



Object: Definition

- ▶ In OOP: Primary mean of abstraction
- ▶ An object is defined by:
 - ▶ State: Attributes (data)
 - ▶ Behaviour: operations
 - ▶ Identity
 - ▶ does not depend on the current value of the attributes
 - ▶ never changes
- ▶ → Each object has at each point in time—
 - ▶ state: the current value of the attributes
 - ▶ behaviour: the set of operations they recognize, and the way they are interpreted
 - ▶ identity



Examples of Objects

Object	Identity	Behaviour	State
A person	'Hussain Pervez.'	Speak, walk, read.	Studying, resting, qualified.
A shirt	My favourite button white denim shirt.	Shrink, stain, rip.	Pressed, dirty, worn.
A sale	Sale no #0015, 18/05/05.	Earn loyalty points.	Invoiced, cancelled.
A bottle of ketchup	<i>This</i> bottle of ketchup.	Spill in transit.	Unsold, opened, empty.

Can you suggest other behaviours and states for these objects?

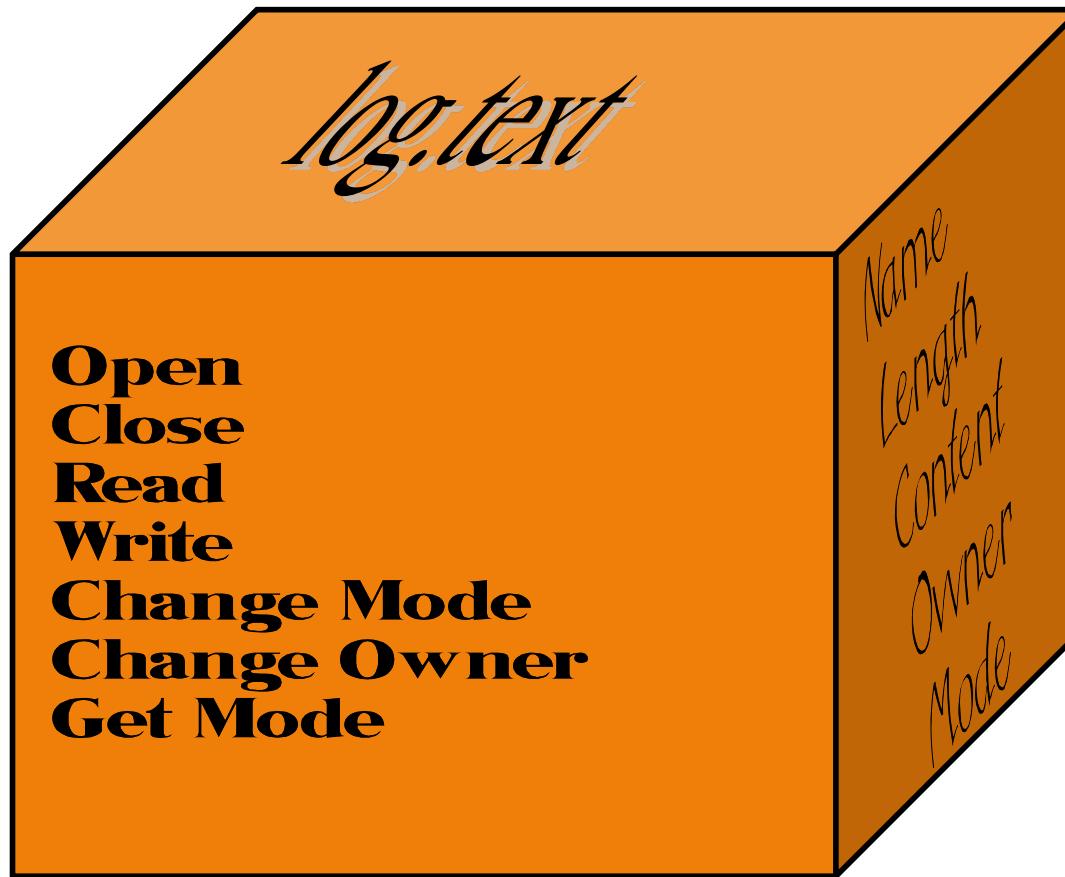
How can external events and object behaviour both result in a change of state?

How can state restrict the possible behaviours of an object?

Examples of objects (cont.)

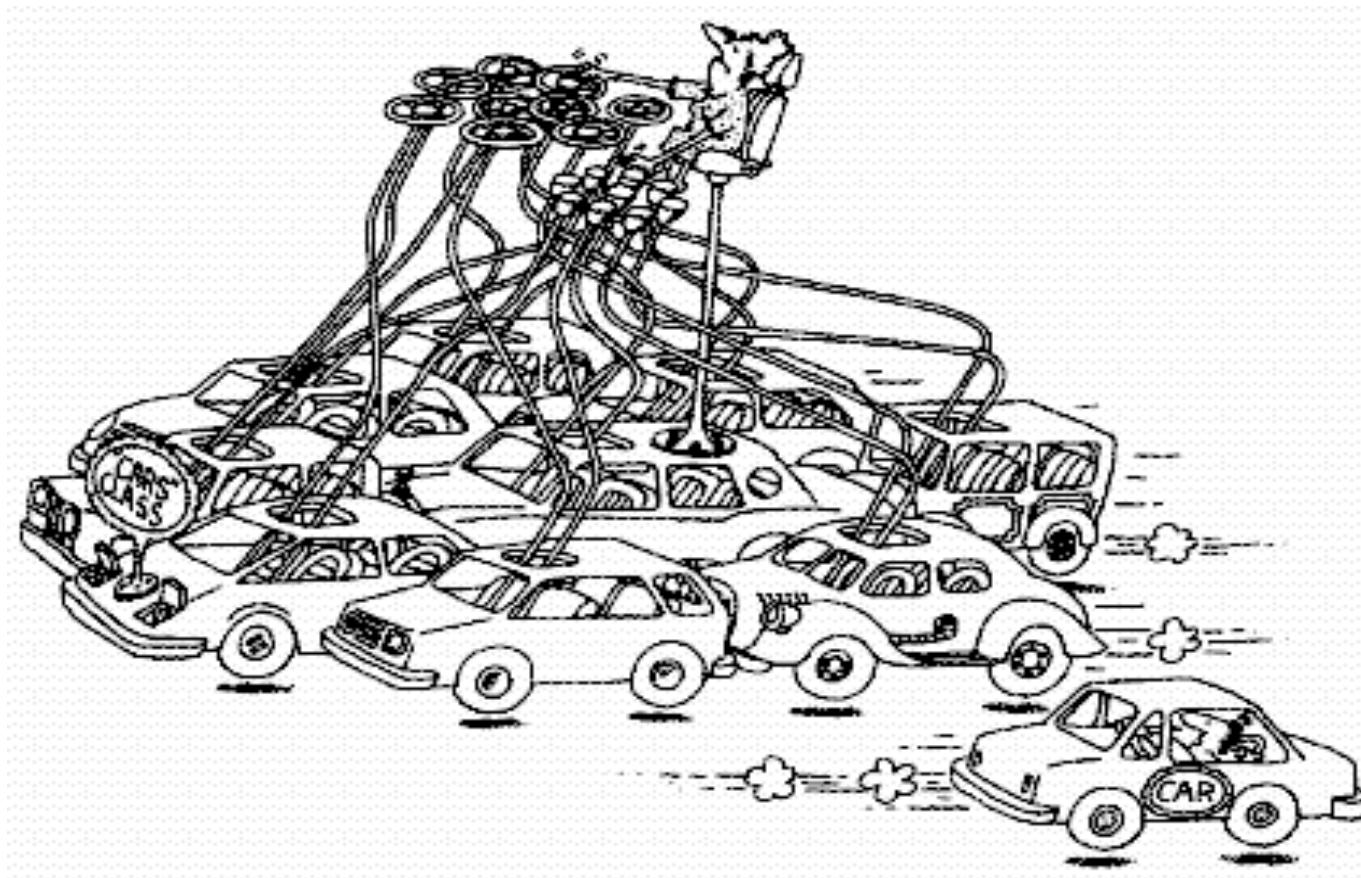
- ▶ Examples:
 - ▶ Time point
 - ▶ Data: 16:45:00, Feb. 21, 1997
 - ▶ Operations: add time interval, calculate difference from another time point,
 - ▶ Acts. For example: Measurement of a patients' fever
 - ▶ Data: 37.1C, by Deborah, at 10:10 am
 - ▶ Operations: Print, update, archive
 - ▶ File
 - ▶ Data: log.txt, -rwx-----, Last read 21:07 June 1, 1999, ...
 - ▶ Operations: read, write, execute, remove, change directory, ...
 - ▶ A communication event (time, length, phone-number, ...)
 - ▶ Transaction in a bank account (withdraw \$15, time, ...)
 - ▶ Elements of ticket machine dispenser: ticket, balance, zone, price, ...

Object: Example



Class: Abstraction Over Objects

- ▶ A class represents a set of objects that share a common structure and a common behavior.



Class and Instance

- ▶ All objects are *instances* of some *class*
- ▶ A Class is a description of a set of objects with similar:
 - ▶ features (attributes, operations, links);
 - ▶ semantics;
 - ▶ constraints (e.g. when and whether an object can be instantiated).

OMG (2009)



Class and Instance

- ▶ An object is an instance of some class
- ▶ So, instance = object
 - ▶ but also carries connotations of the class to which the object belongs
- ▶ Instances of a class are similar in their:
 - ▶ *Structure*: what they *know*, what information they hold, what links they have to other objects
 - ▶ *Behaviour*: what they *can do*



What is a Class?

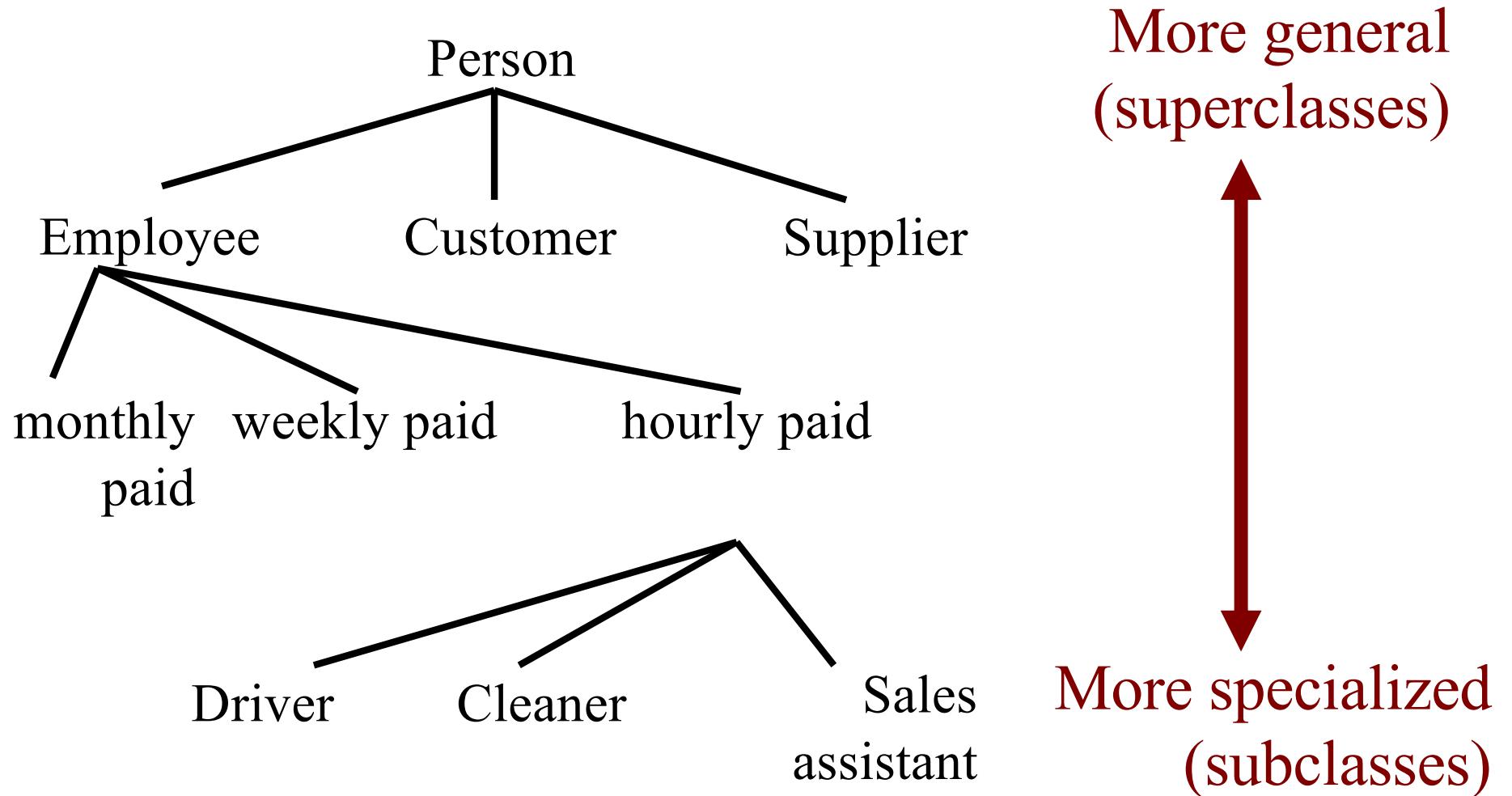
- ▶ Stands for a single human concept
 - ▶ An abstraction technique
- ▶ Mould for objects
 - ▶ used to instantiate objects (instances) with distinct identities that share protocol, behavior and structure but may assume different states.
 - ▶ In contrast to concrete object, a class does not necessarily exist in (run) time and (memory) space.
- ▶ What's not a Class?
 - ▶ An object is not a class.
 - ▶ ... but a class may be an object:
 - ▶ In “exemplar based” languages eg. Smalltalk, there are no classes. New objects are “instantiated” from existing objects.

Generalization and Specialization

- ▶ Classification is hierachic in nature
- ▶ For example, a person may be an employee, a customer, a supplier of a service
- ▶ An employee may be paid monthly, weekly or hourly
- ▶ An hourly paid employee may be a driver, a cleaner, a sales assistant



Specialization Hierarchy



Generalization and Specialization

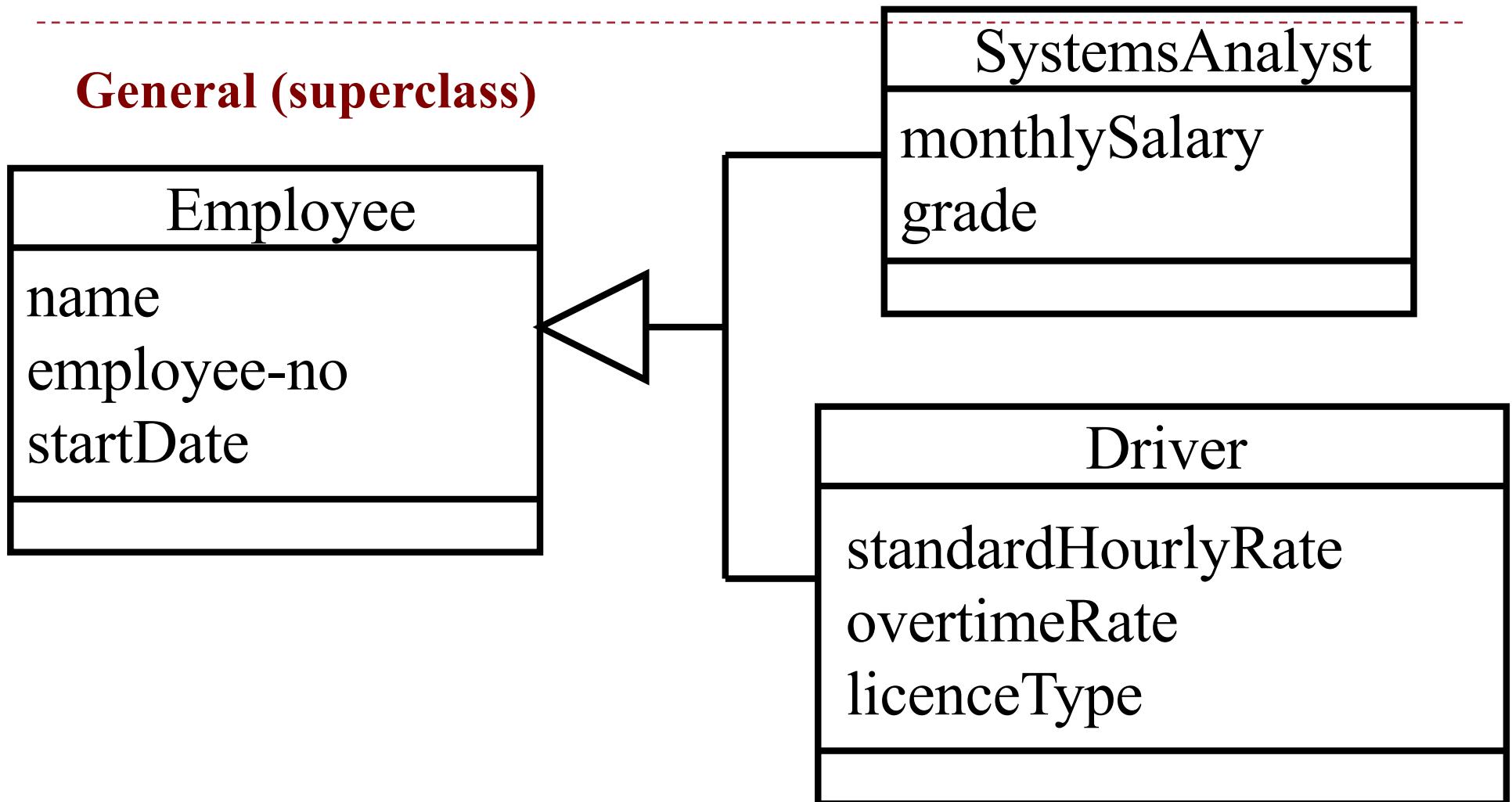
- More general bits of description are *abstracted out* from specialized classes:

SystemsAnalyst
name
employee-no
startDate
monthlySalary
grade

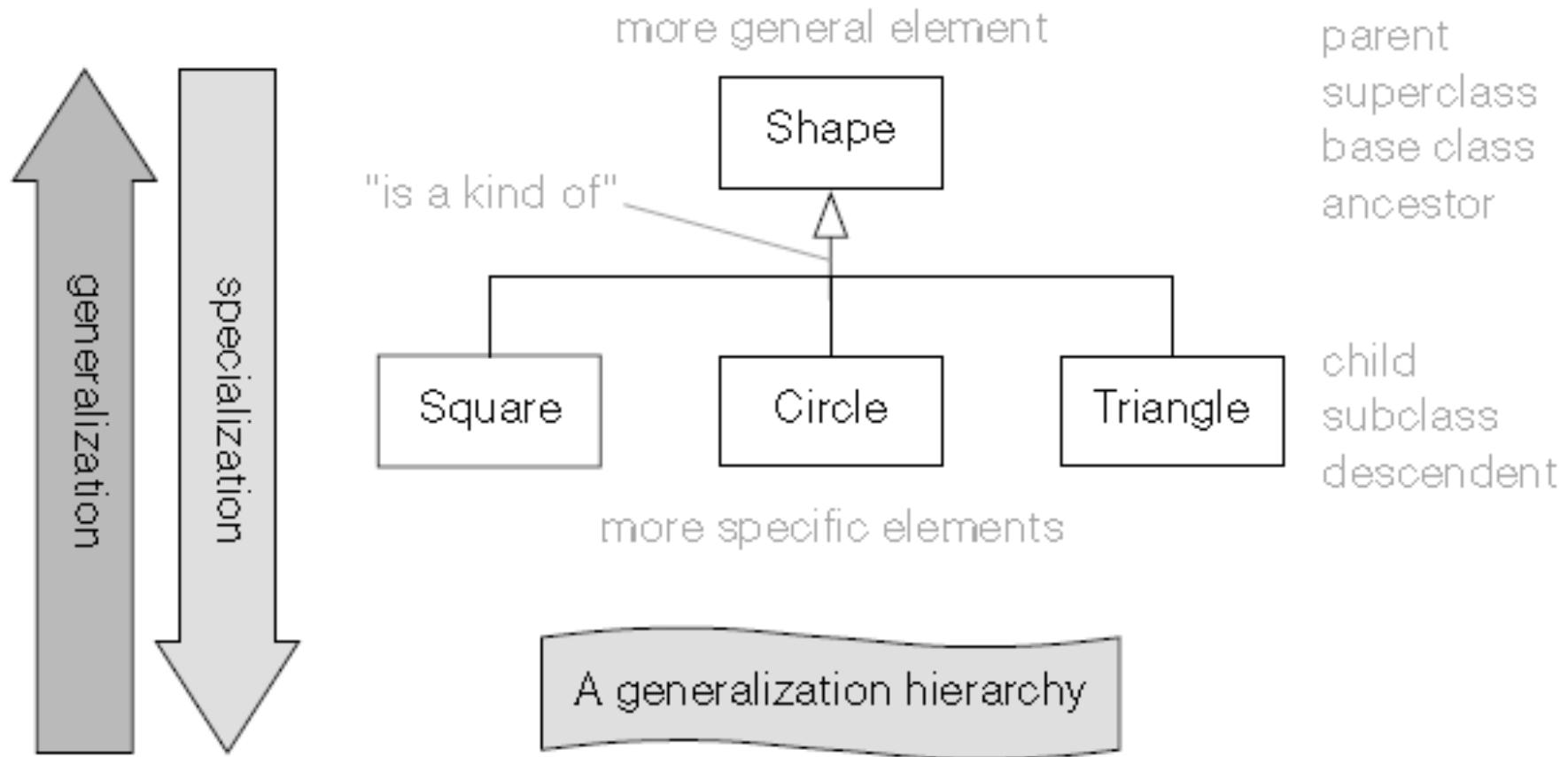
Driver
name
employee-no
startDate
standardHourlyRate
overtimeRate
licenceType



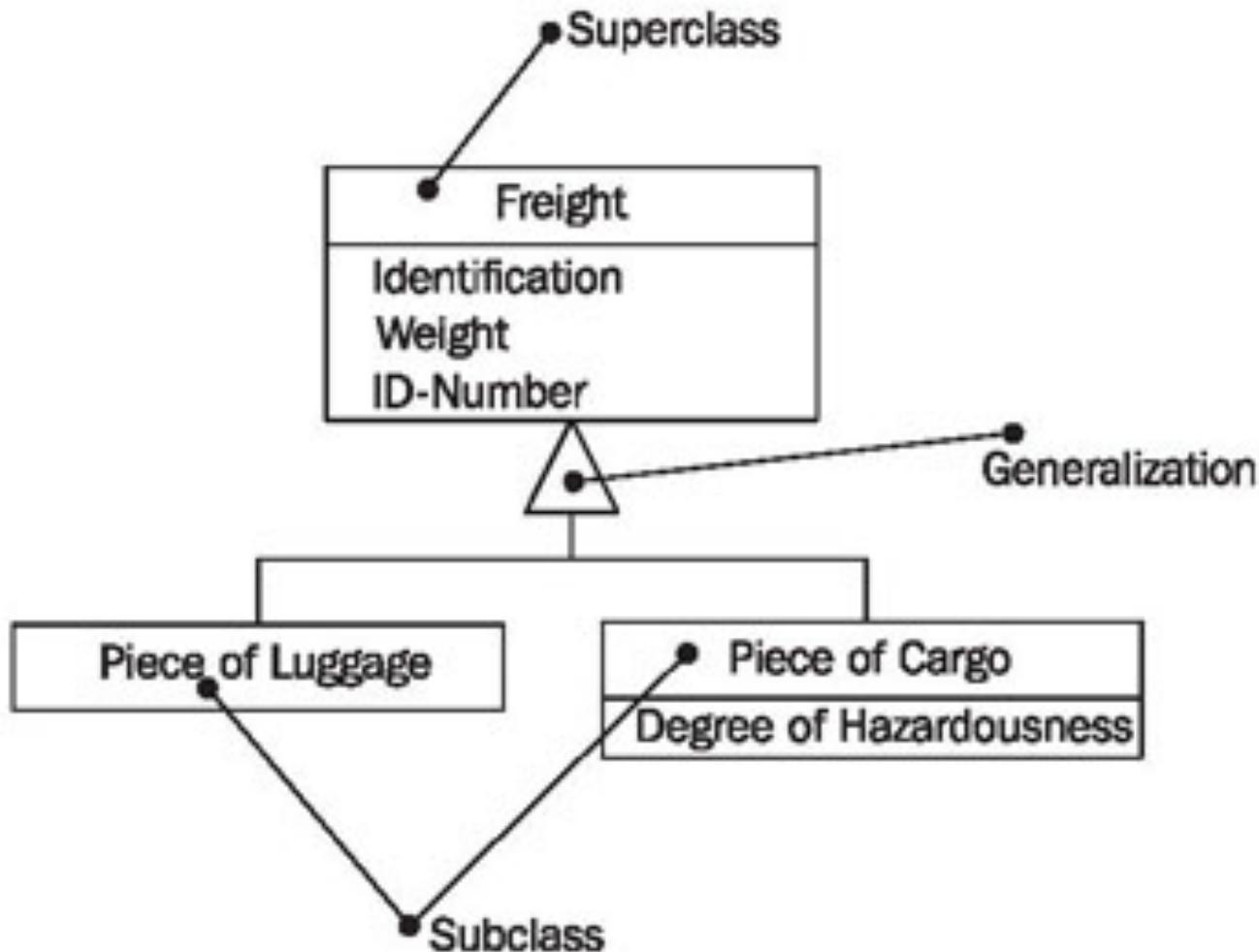
Specialized (subclasses)



Generalization

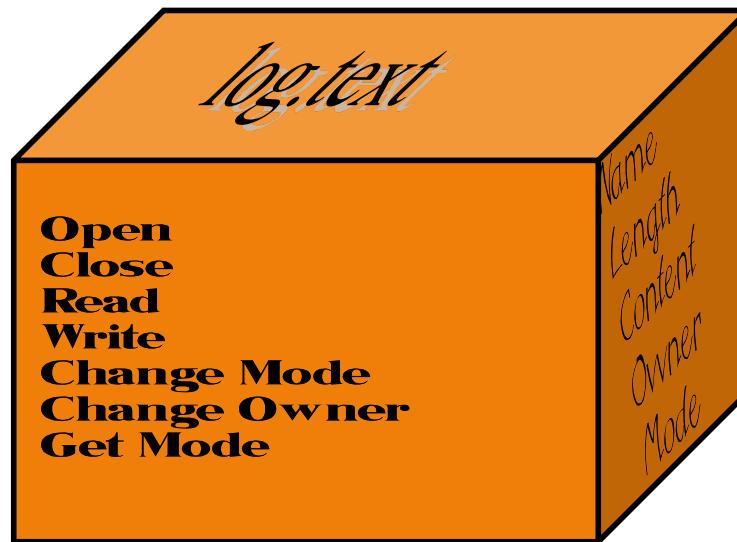


Object-oriented modelling notation: genericity in class diagrams



Encapsulation definition

- ▶ Encapsulation: an object's data is located with the operations that use it



Message-passing

- ▶ Several objects may collaborate to fulfil each system action/use-case
- ▶ “Record CD sale” could involve:
 - ▶ A CD stock item object
 - ▶ A sales transaction object
 - ▶ A sales assistant object
- ▶ These objects communicate by sending each other messages



Message-passing and Encapsulation

‘Layers of an onion’
model of an object:

An outer layer of
operation signatures...

...gives access to middle
layer of operations...

...which access an
inner core of data

Message from another object
requests a service.

Operation signature is an interface
through which an operation can be
called.

Operations are located
within an object.

Data used by an
operation is located in
the object too



Classes

- ▶ Kinds of classes:
 - ▶ Abstract (also: Java interface)
 - ▶ No instances!
 - ▶ Serve as an interface, or a common base with similar behavior
 - ▶ Example: Collection (Smalltalk, Java)
 - ▶ Singleton
 - ▶ One instance!
 - ▶ Example: True (Smalltalk)
 - ▶ Language support for singletons:
 - Self: No classes! All objects are singletons
 - BETA: Objects may be defined as singletons
 - ▶ Concrete/effective
 - ▶ Any number of instances

HOW TO SLAUGHTER A PIG



Classes in O-O programming languages

► C++

```
class Employee: public Person {  
private:                      // visible to none  
    Date birthday;           // data member  
public:                       // visible to all  
    void hire(Reason why)    // function member  
    { /* ... */ }  
};
```

► Java:

```
class Employee extends Person {  
    private Date birthday;      // field, visible to none  
    public void Hire(Reason why) { // method, visible to all  
        { /* ... */ }  
    }  
}
```

Classes in OOPLs (Cont.)

► Smalltalk:

```
Person subclass: Employee
instanceVariables: 'birthday' "instance variable, visible to none"
classVariables: ''
poolDictionaries: ''

hire: why           "method, visible to all"
"..."
```

► Eiffel:

```
class EMPLOYEE
inherit
  PERSON
feature {NONE}          -- visible to none
  DATE birthday;         -- attribute
feature {ALL}            -- visible to all
  hire(why: REASON) is -- routine
    do
      -- ...
    end
end -- class EMPLOYEE
```

Information Hiding

Information hiding

- ▶ Also known As: Data Abstraction, Data Hiding

... like us, objects have a private side. The private side of an object is how it does these things, and it can do them in any way required. How it performs the operations or computes the information is not a concern of other parts of the system. This principle is known as information hiding [Wirfs-Brock 1990, p.6].

- ▶ Why?

“It is an attempt to minimize the expected cost of software and requires that the designer estimate the likelihood of changes.” [Parnas 1985]

HOW TO SLAUGHTER A PIG

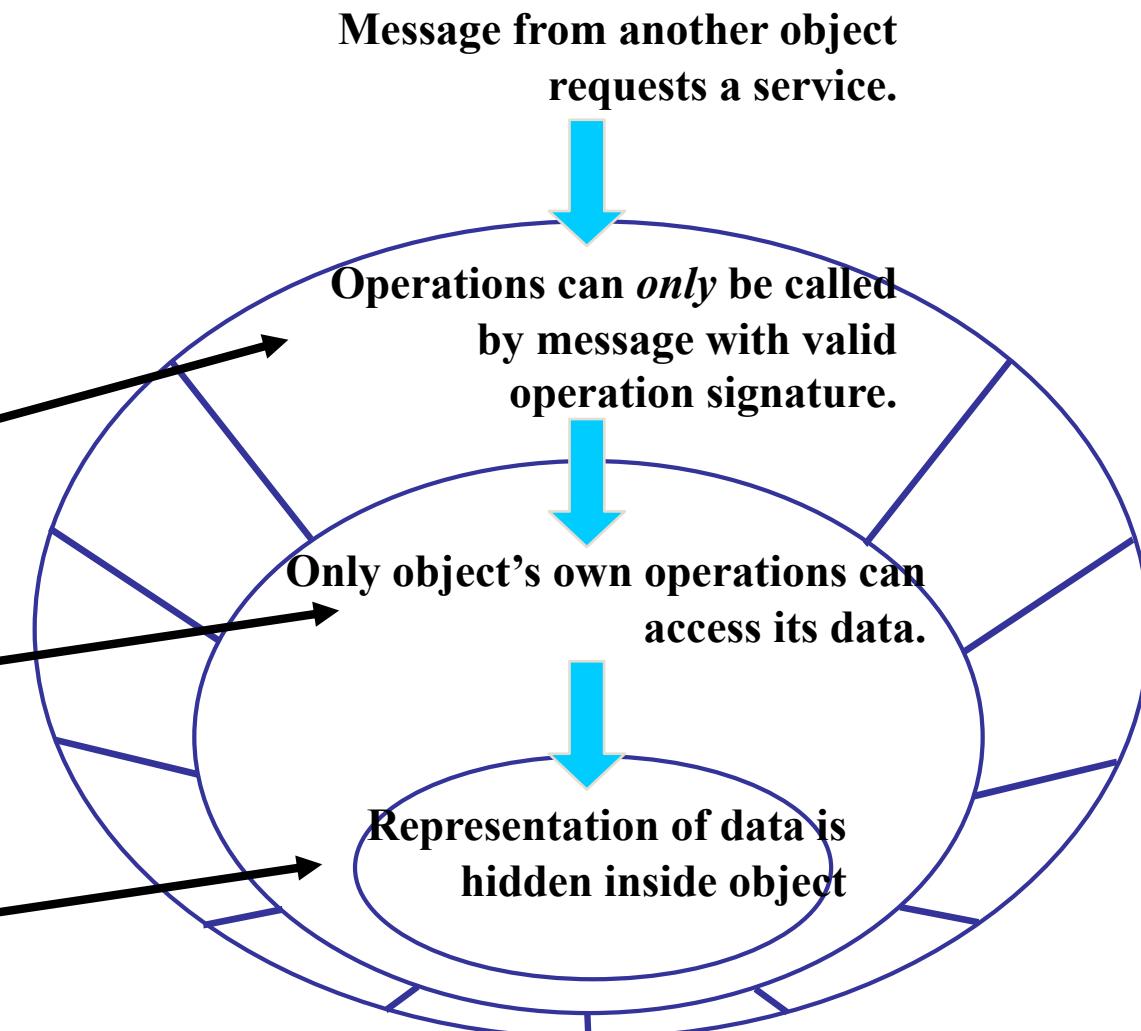


PHASE 1: TRY IT WITHOUT - 9

Information Hiding: the onion model

‘Layers of an onion’
model of an object:

Only the outer layer is
visible to other objects...
...and it is the only way to
access operations...
...which are the only way
to access the hidden data



Note:

Information Hiding Vs. Encapsulation

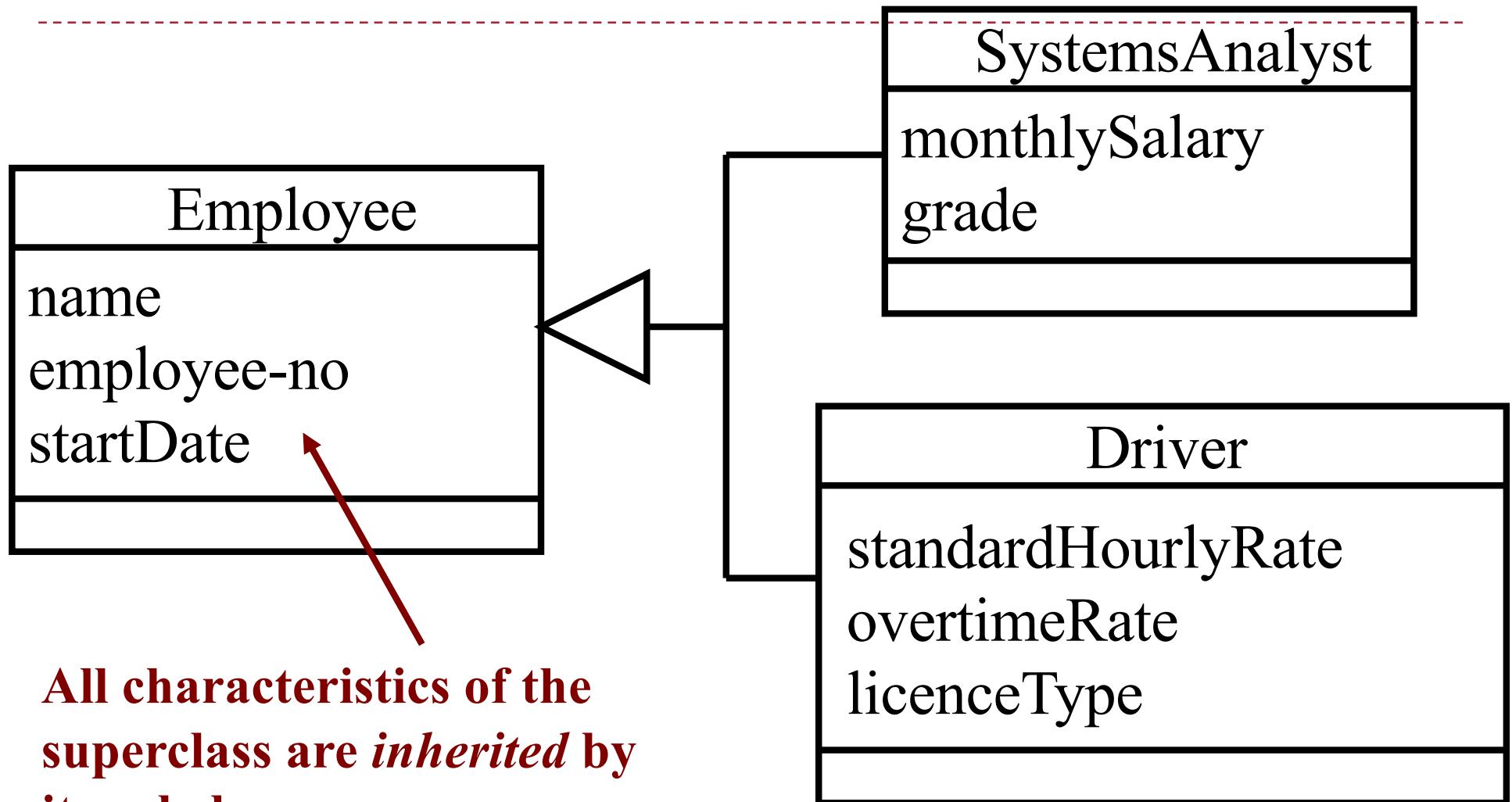
- ▶ The terms are sometimes confused and used interchangeably
 - ▶ Some people say “encapsulation” with reference to what we described as information hiding
- ▶ We shall adhere to the interpretation in these slides
 - ▶ Encapsulation: an object’s data is located with the operations that use it
 - ▶ **Information hiding: only an object’s Interface is visible to other objects**

Inheritance

Inheritance

- ▶ The *whole* description of a superclass applies to *all* its subclasses, including:
 - ▶ Information structure (including associations)
 - ▶ Behaviour
- ▶ Often known loosely as *inheritance*
- ▶ (But actually inheritance is how an O-O programming language *implements* generalization / specialization)

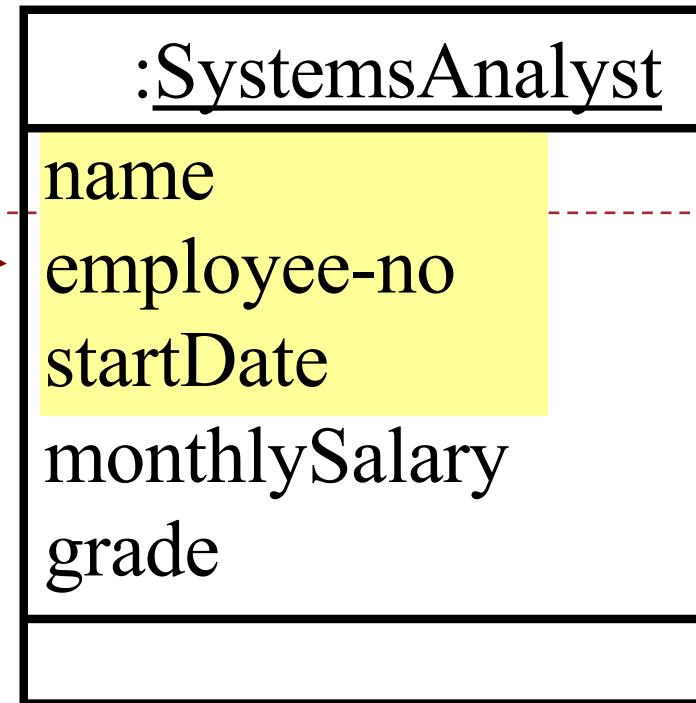
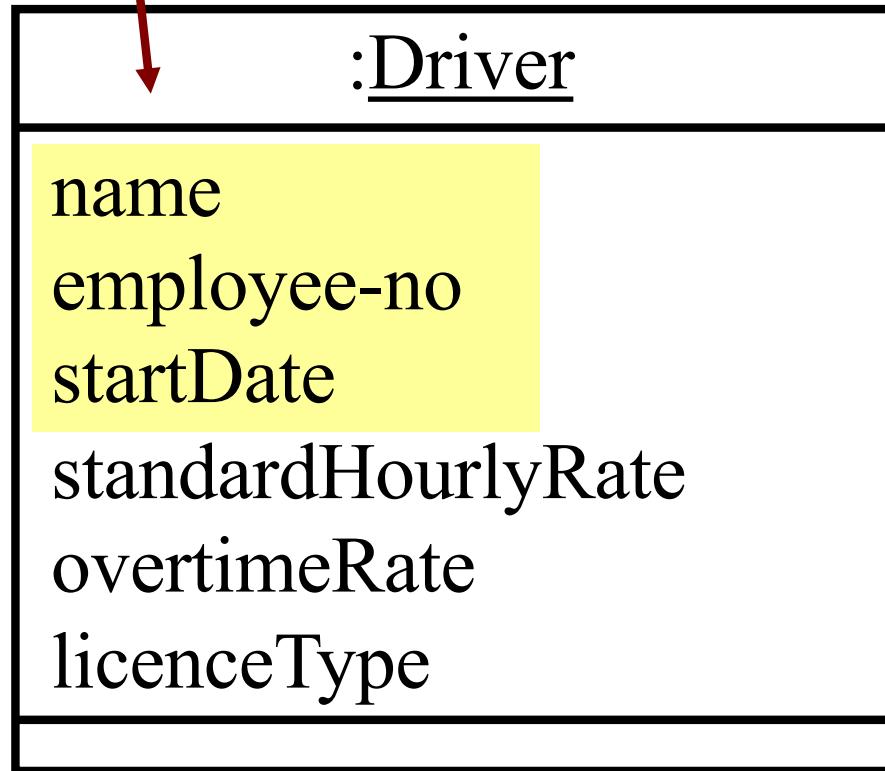




All characteristics of the superclass are *inherited* by its subclasses



Instances of each subclass include the characteristics of the superclass (but not usually shown like this on diagrams)

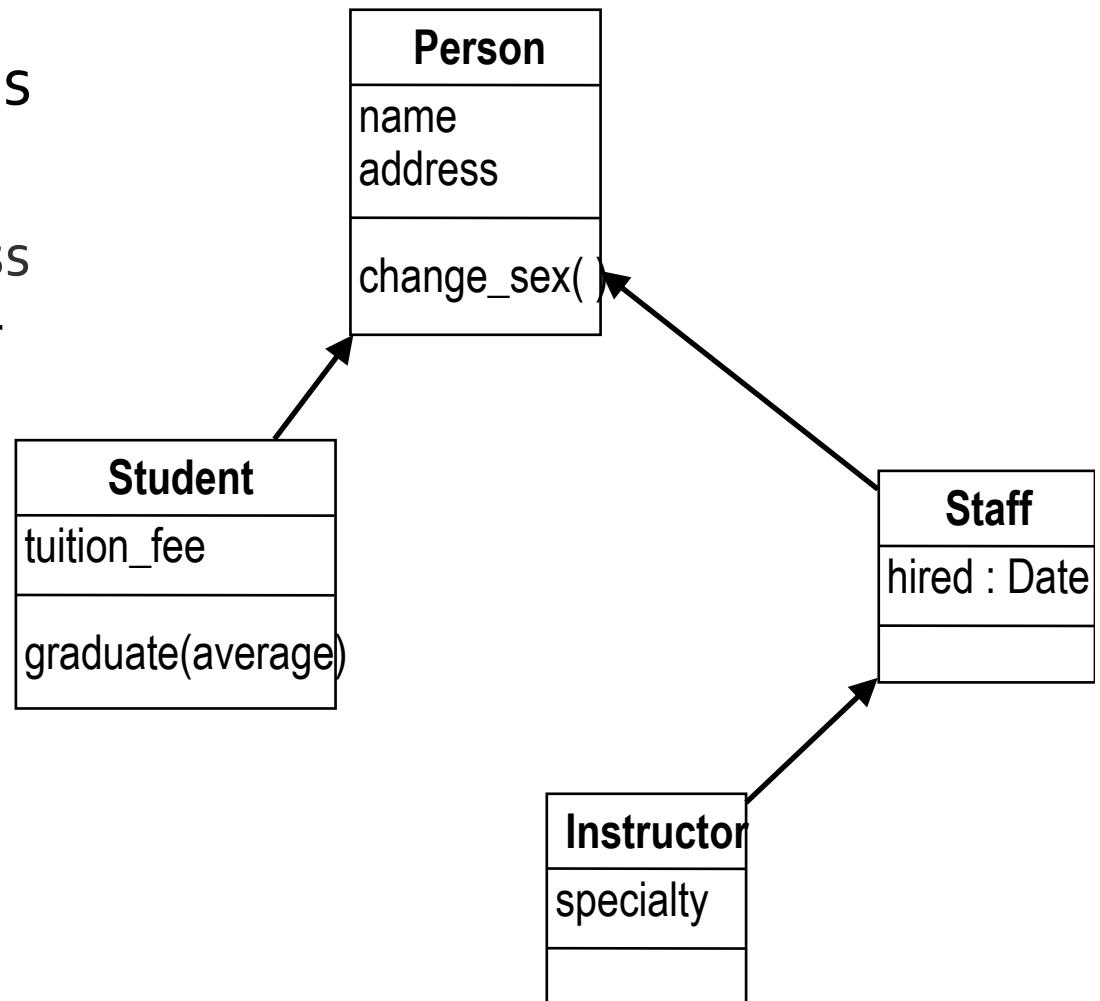


Inheritance

- ▶ Represents separate notions:
 - ▶ Generalization: ‘is-kind-of’ relation
 - ▶ Instances of the specialized class are a subcategory of the generalized class
 - ▶ Subtyping (in Java: ‘implements’)
 - ▶ The subtypes supports all the operations on supertype
 - ▶ Subclassing (in Java: extends)
 - ▶ A mechanism of code reuse

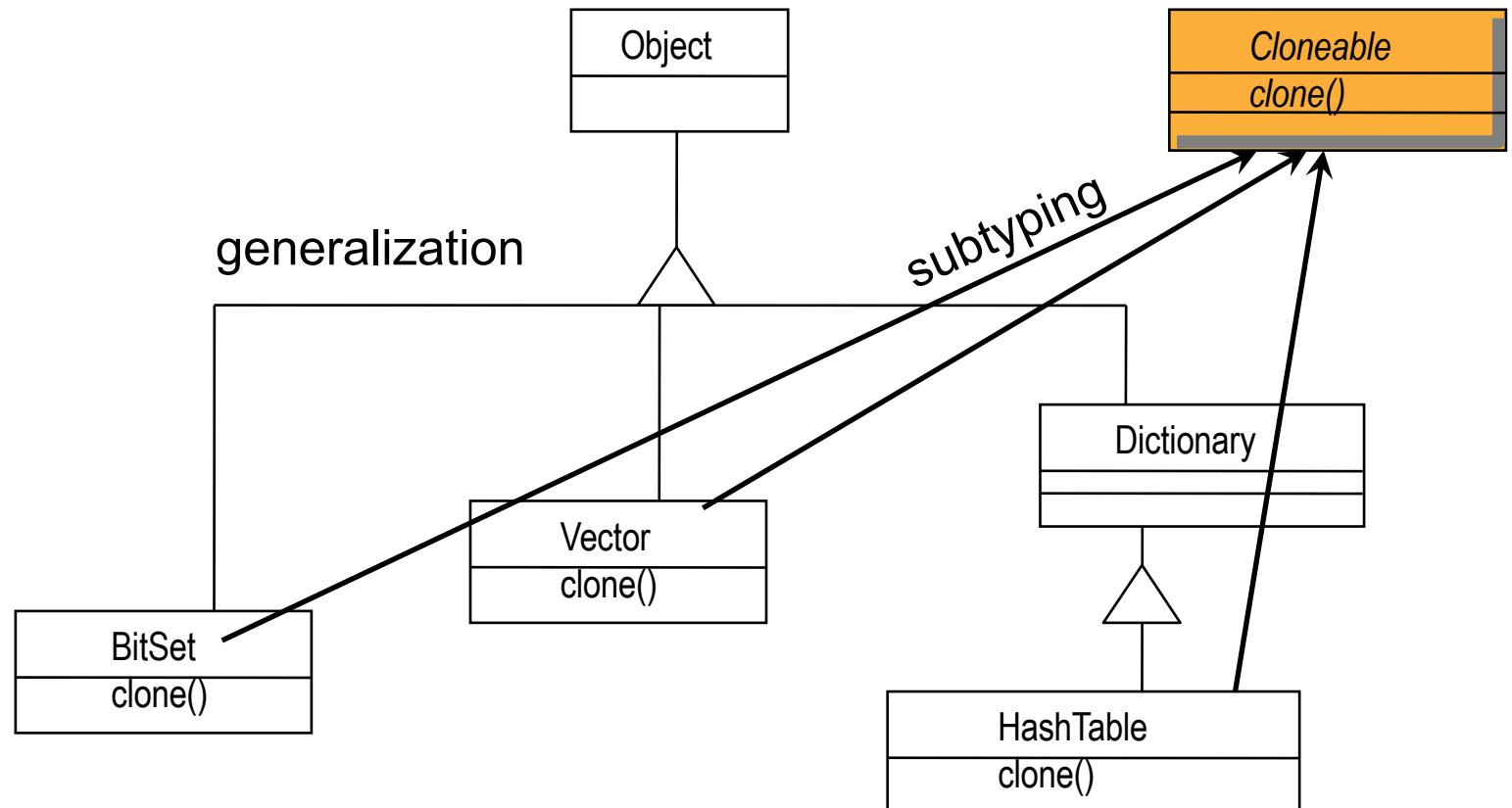
Inheritance and generalization

- ▶ Superclass is a generalization of Subclass
 - ▶ Also: subclass is a specialization of superclass
- ▶ Also known as: is-kind-of relation:
 - ▶ Staff is-kind-of Person
 - ▶ Instructor is-kind-of Person



Inheritance and subtyping

- ▶ Subtype supports the same operations as supertype



Subtyping enables a given type to be substituted for another type or abstraction. Subtyping is said to establish an **is-a** relationship between the subtype and some existing abstraction, either implicitly or explicitly, depending on language support. The relationship can be expressed explicitly via inheritance in languages that support inheritance as a subtyping mechanism.

In some OOP languages, the notions of code reuse and subtyping coincide because the only way to declare a subtype is to define a new class that inherits the implementation of another.

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

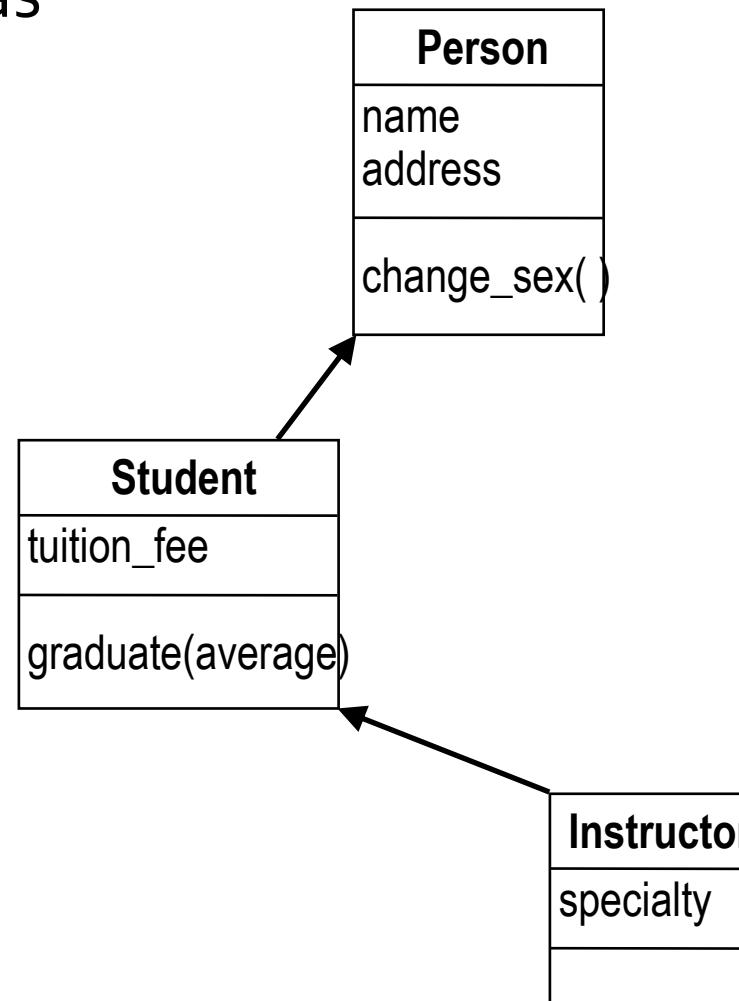
Invoking `Object`'s `clone` method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method.

See `Object.clone()` for details on overriding this method.

Inheritance and subclassing

- ▶ Inheritance can be used as a crude mechanism of reuse
- ▶ A very problematic and dangerous tactic



In this lecture we have looked at

- ▶ What is an object
- ▶ What is a class
- ▶ Advantages of OO
- ▶ OO mechanisms of
 - ▶ Encapsulation/Information Hiding
 - ▶ Inheritance
- ▶ In the class we will look at Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity

Class exercise: Polymorphism

Exercises: OO inheritance

- ▶ What rules describe the relationship between a subclass and its superclass?
- ▶ For each one of the following class pairs, determine the appropriate kind of inheritance relation between them:
 - ▶ Person, Parent
 - ▶ Person, Mammal
 - ▶ Bird, FlyingObject
- ▶ What is the difference between generalization and subtyping?

Polymorphism

- ▶ **Definition:** refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.
- ▶ Polymorphism allows one message to be sent to objects of different classes
- ▶ Sending object need not know what kind of object will receive the message
- ▶ Each receiving object knows how to respond appropriately
- ▶ For example, a ‘resize’ operation in a graphics package
 - ▶ the way that circles, rectangles, bitmaps, groups of objects, etc will all need their own distinct methods to implement the operation. However, from the perspective of a user (or indeed a boundary object) all are called in the same way.



Polymorphism

- ▶ Polymorphic vs. monomorphic programming languages:
 - ▶ In a monomorphic language “every value, variable, and operation can be interpreted to be of one and only one type.”
 - ▶ In polymorphic languages “some values, variables, and operations may have more than one type.”
- ▶ Kinds of polymorphism:
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity

Polymorphism I: Overloading

- ▶ Operations and methods with same name are defined for arguments of different types

```
class example {  
    void demo(int a, int b, double x, double y) {  
        a / b; // integer division  
        x / y; // floating-point division  
    }  
  
    void demo() { // method overloading  
        ...  
    }  
}
```

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.

Thus, overloaded methods must differ in the type and/or number of their parameters.

While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

Polymorphism II: Dynamic binding

- ▶ When a message is dispatched at run time, the operation that is invoked is chosen according to the class of the object. This is known as **dynamic (late) binding**.
- ▶ All OOP languages support dynamic binding.

```
public class Parent {  
    void whoami () {  
        System.out.println("I am a parent");  
    }  
}
```

```
public class Child extends Parent {  
    void whoami () {  
        System.out.println("I am a child");  
    }  
}
```

```
public class ParentTest {  
    public static void main(String args[]) {  
        Parent p = new Parent();    p.whoami();  
        Child c = new Child();      c.whoami();  
        Parent q = new Child();    q.whoami();  
    }  
}
```

- In this example of Dynamic Binding we have used concept of method overriding. Child extends Parent and overrides its whoami() method and when we call whoami() method from a reference variable of type Parent, it doesn't call whoami() method from Parent class instead it calls whoami() method from Child subclass because object referenced by Parent type is a Child object. This resolution happens only at runtime because object only created during runtime and called dynamic binding in Java. Dynamic binding is slower than static binding because it occurs in runtime and spends some time to find out actual method to be called.
- If we wanted to force a type then we would use static binding and not an object type for the variable.

Read more: <http://javarevisited.blogspot.com/2012/03/what-is-static-and-dynamic-binding-in.html#ixzz2A7qZfS8F>

Dynamic binding: counterexample

```
public class OperatingSystem {  
    Printer myPrinter;  
    void PrintJob(Job j) {  
        switch(myPrinter.type()) {  
            case ASCII: ASCIIPrinter.print(j);  
            case PS: PSPrinter.print(j);  
            ...  
        }  
    }  
}
```

```
public class ASCIIPrinter extends Printer { ...  
    static public boolean print(Job j) {...}  
}
```

```
public class PSPrinter extends Printer { ...  
    static public boolean print(Job j) {...}  
}
```

Question:
What is wrong
with this
solution?

Dynamic binding: example

```
public class OperatingSystem {  
    Printer myPrinter;  
    void PrintJob(Job j) {  
        myPrinter.print(j)  
    }  
}
```

```
public abstract class Printer { ...  
    abstract public void boolean print(Job j);  
}
```

```
public class ASCIIPrinter extends Printer { ...  
    public boolean print(Job j) {... }  
}
```

```
public class PSSPrinter extends Printer { ...  
    public boolean print(Job j) {... }  
}
```

Question:

How is this better
than the
counterexample?

Polymorphism 3: Genericity

- ▶ A mechanism allowing type-safe programming using generic classes ('templates')

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0);      // Run time error
```

```
List<String> v = new ArrayList<String>();
v.add("test");
Integer i = v.get(0); // (type error) Compile time error
```

Example 1. The block of Java code illustrates a problem that exists when not using generics.

First, it declares an ArrayList of type Object.

Then, it adds a String to the ArrayList.

Finally, it attempts to retrieve the added String and cast it to an Integer.

Although the code compiles without error, it throws a runtime exception (java.lang.ClassCastException) when executing the third line of code.

This type of problem can be avoided by using generics and is the primary motivation for using generics.

Example 2. Using generics, the first code fragment is rewritten.

The type parameter String within the angle brackets declares the ArrayList to be constituted of String (a descendant of the ArrayList's generic Object constituents).

With generics, it is no longer necessary to cast the third line to any particular type, because the result of v.get(0) is defined as String by the code generated by the compiler.

Compiling the third line of this fragment with J2SE 5.0 (or later) will yield a compile-time error because the compiler will detect that v.get(0) returns String instead of Integer.

Exercises

- ▶ What will be the output of ParentTest. main () ?
- ▶ In Java, is int a subtype of float? Vice versa? Is double a subtype of float? Vice versa?

Summary

- ▶ OO Analysis & Design has same mechanisms as OO programming
- ▶ OO provides many benefits
- ▶ Difference between Objects and Classes
- ▶ Looked at core OO concepts:
 - ▶ Encapsulation
 - ▶ Information Hiding
 - ▶ Inheritance
 - ▶ Generalization/specialization
 - ▶ Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity

Further reading

- ▶ Object-orientation
 - ▶ Chapter 4, Bennett
 - ▶ Coad, P & Yourdan, E (1990) Object-oriented analysis. Prentice-Hall.
 - ▶ Booch, G (1994) Object-oriented analysis and design with applications. Menlo Park.
- ▶ Information hiding & abstraction:
 - ▶ Wirfs-Brock, Rebecca, Wilkerson, Brian, and Wiener, Lauren. 'Designing Object-Oriented Software'. Prentice-Hall, 1990.
 - ▶ Parnas, 1985, Communications of the ACM.
- ▶ Look at Java Interfaces
 - ▶ Eg. <http://docs.oracle.com/javase/tutorial/java/concepts/interface.html>
 - ▶ Look at Java abstraction
 - ▶ Eg. <http://javarevisited.blogspot.co.uk/2010/10/abstraction-in-java.html>
- ▶ Exemplar based object-orientation:
 - ▶ 55
 - ▶ 'An exemplar based Smalltalk'. OOPSLA 1986 Proceedings.



Requirements engineering: OOA with Type Diagrams

- Object-oriented analysis example
- Modelling techniques: type diagrams
- Analysis patterns

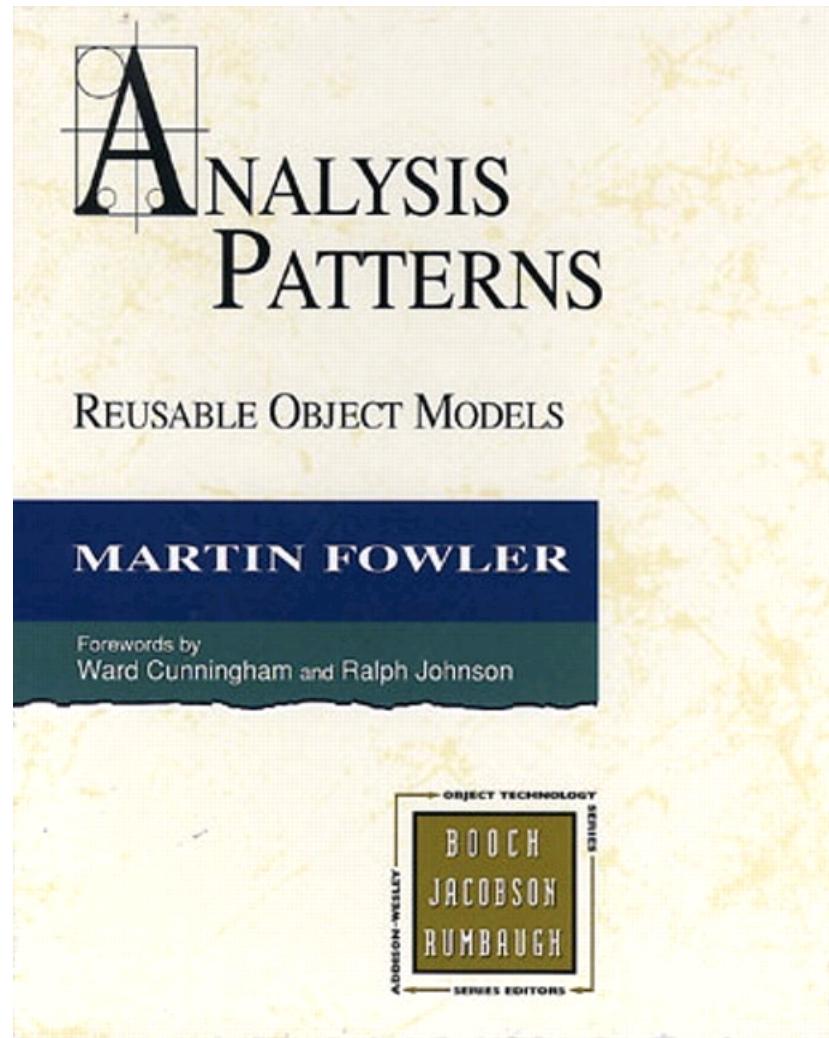
CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



OOA with Type Diagrams

- ▶ M. Fowler (1997). *Analysis Patterns*. Addison-Wesley.
- ▶ Mainly *Chapter 3: Observations and Measurements*
- ▶ Also Page 313 – Appendix A



Analysis of the problem domain

- ▶ Models produced from analysis and design are deliberately similar
 - ▶ The emphasis in analysis is on understanding the **problem**
 - ▶ The emphasis in design is on understanding the **solution**
- ▶ Last week we looked at requirements in general and how use case diagrams and descriptions can be used to analyse the requirements
- ▶ In analysis you may need to look behind the surface requirements (rather than just listing requirements in use-cases)

Example: understanding Snooker



Write a simulation of the game of snooker

- Could write a set of use-cases which describe the **surface features**
 - eg. “The player hits the white ball so that it travels at a certain speed; it hits the red ball at a certain angle, and the red ball travels a certain distance and direction”
- In order to understand the problem, you need to look behind the surface to understand the laws of motion that relate mass, velocity, momentum, etc.
- Unlike snooker, in most enterprises these foundations/laws are not well understood.
 - So we need to create **conceptual models** that allow us to understand and simplify the problem.

How can we develop conceptual models as reusable patterns?

► Analysis patterns:

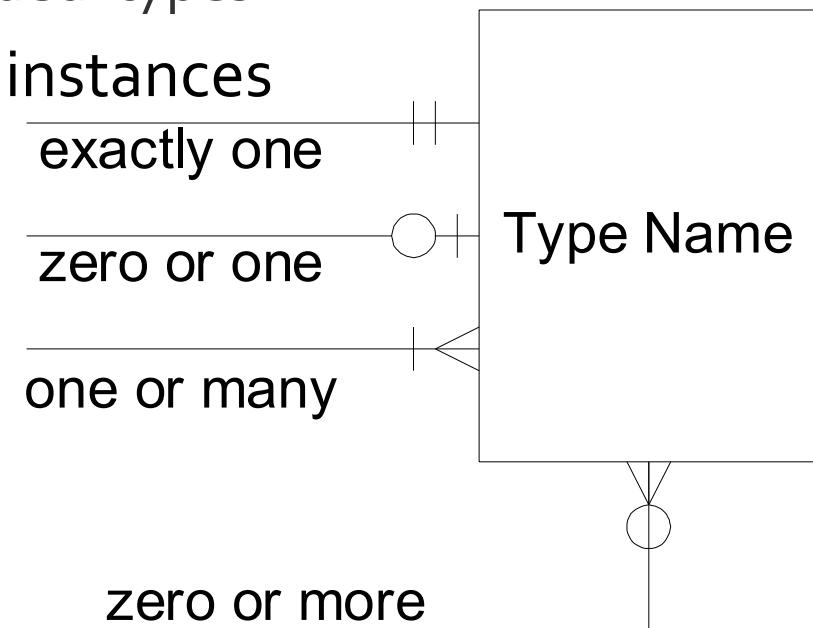
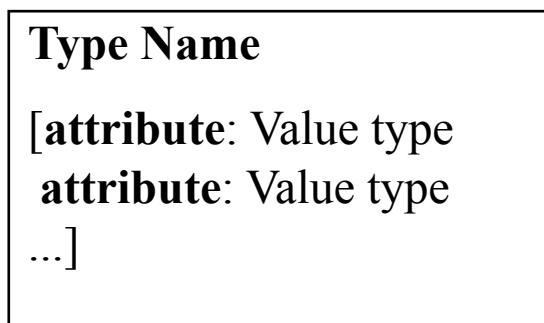
- ▶ Groups of concepts that represent a common construction in business modeling.
- ▶ Relevant to one or more domains
- ▶ Can be re-used for different enterprises that share the same problem
- ▶ Overall principle:
 - ▶ Patterns are a starting point, not a destination
 - ▶ Models are not right or wrong, they are more or less useful

Modeling technique: the TYPE diagram

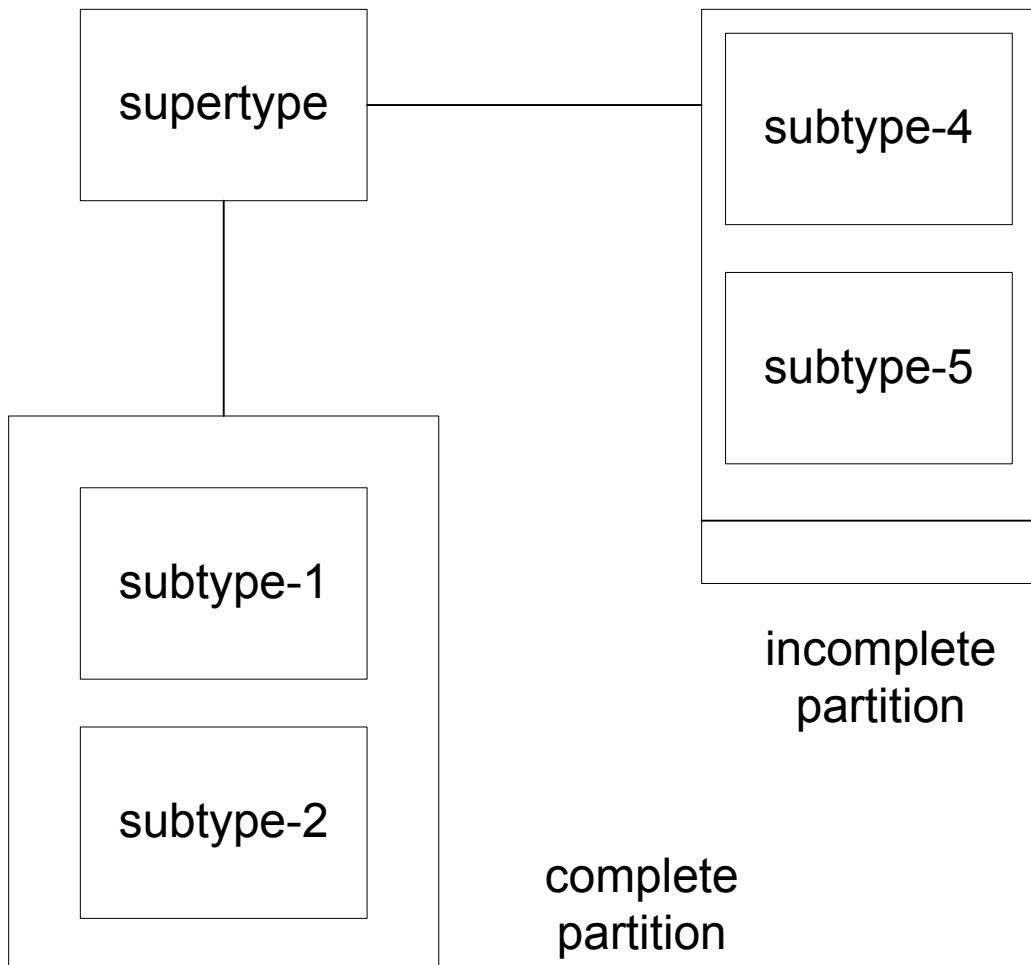
- ▶ Shows a structural view of a system
- ▶ Describes the **types of objects** and **relationships** that exist amongst them
- ▶ Based on Odell's notation (see earlier reference)
- ▶ Not based on UML

Requirements modelling technique: Type Diagrams

- ▶ Basic notion: Type
 - ▶ Stands for a set of instances
 - ▶ Describes the visible *interface* of a class
 - ▶ A *Type* can be implemented by many classes
 - ▶ Similar to ER diagrams with added ‘types’
- ▶ Cardinality: relation between instances



Type Diagrams II: subtypes



Complete partition: all instances of supertype must be instances of either subtype

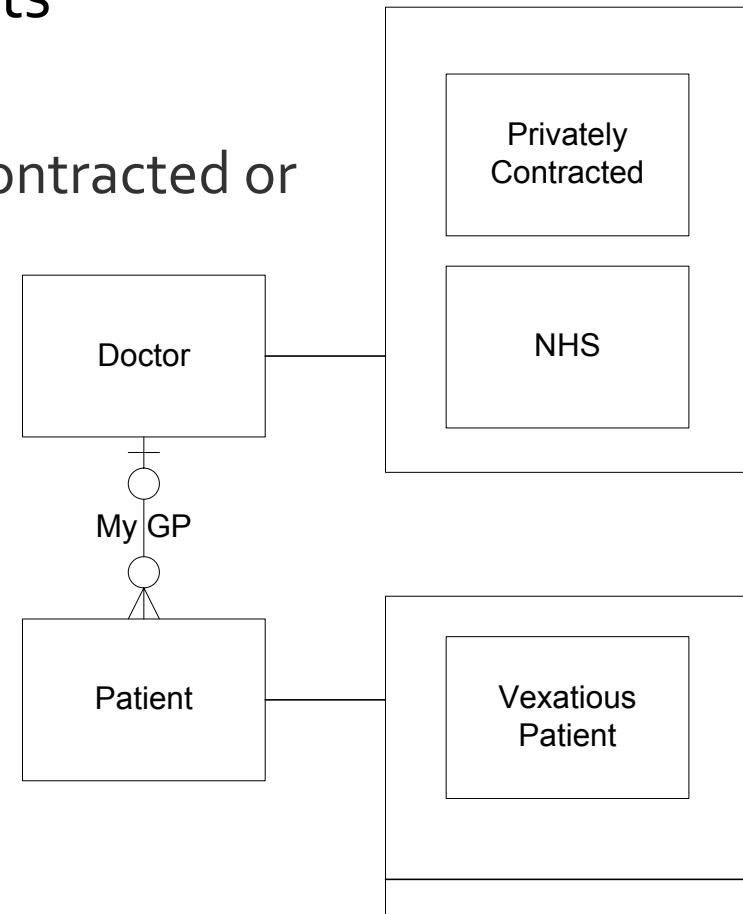
Incomplete partition: instances of supertype can be instances of either subtype OR neither

Type diagrams

- ▶ Example: Suppose a patient has one GP at the most, but a doctor can have many patients
 - ▶ Some patients are 'vexatious'
 - ▶ All doctors are either privately contracted or NHS-employed

Complete partition = all instances of supertype must be instances of at least one of the subtypes

Incomplete partition = instances of supertype can be instances of either subtype or neither



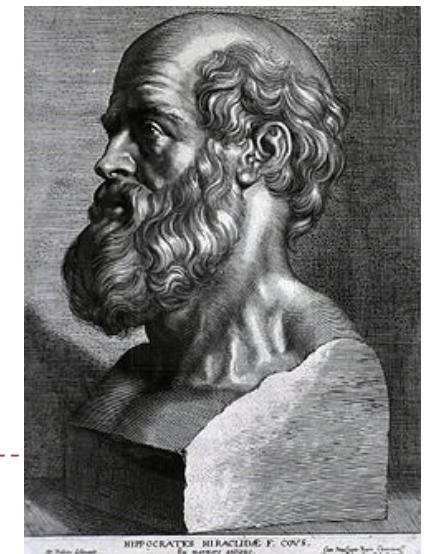
Types and objects

- ▶ A **type** (of objects) specifies a domain of objects together with the operations applicable to them.
- ▶ An object can be a member of several types.

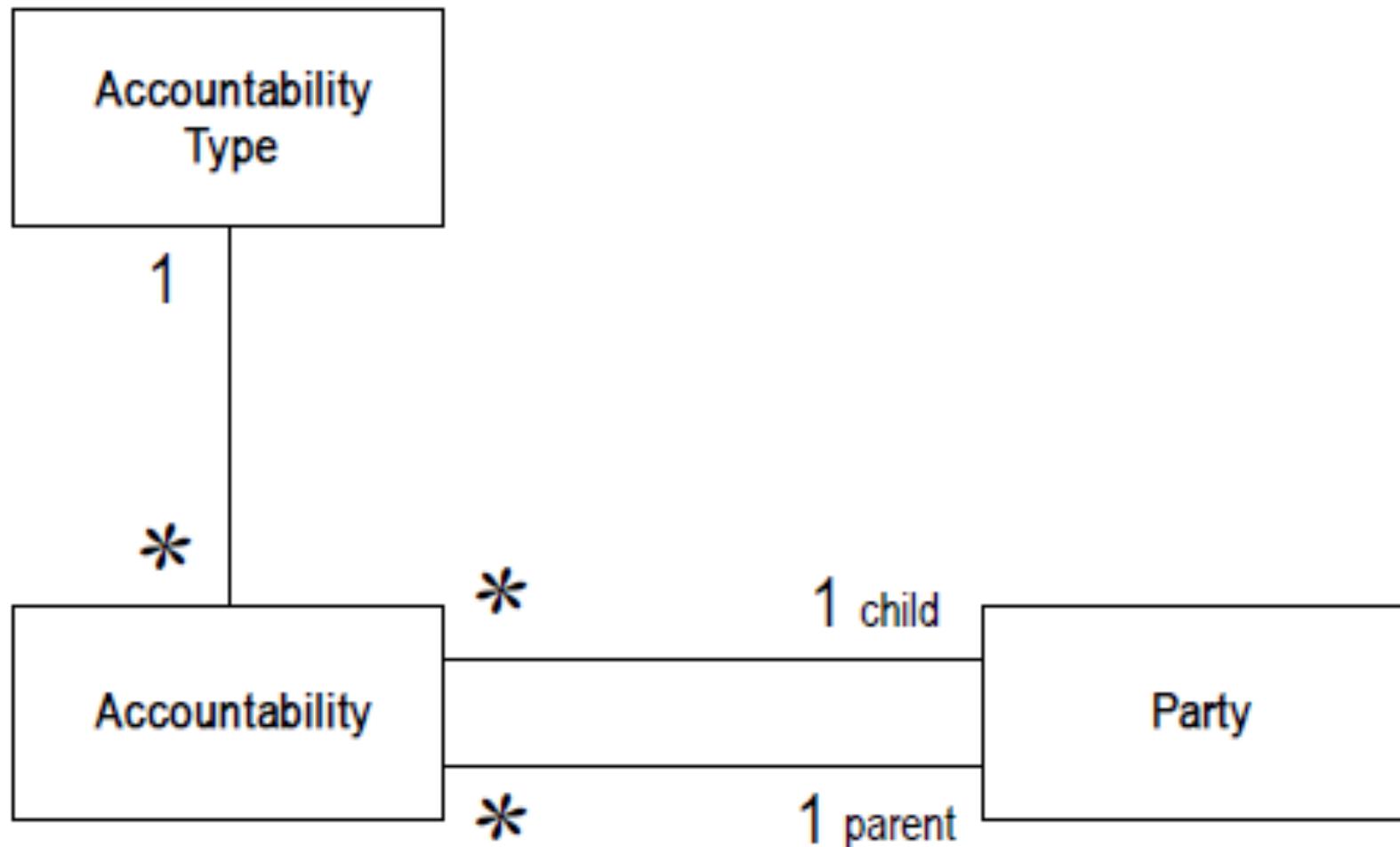
Patient

height
weight
blood glucose level

Doctor



We can use Type diagrams to create a conceptual model for a given domain



Example (pattern): observations and measurements

- ▶ (see ch. 3 of Fowler)
- ▶ Most systems record information about objects in the real world
- ▶ Typical solution is to record this using an attribute to an object
 - ▶ Eg. Person object with an attribute weight with a value of 185
- ▶ Problems with this solution
- ▶ We will see how a pattern/model can be developed to handle both quantitative and qualitative information

Example: *Quantity*

- ▶ Problem: How to record measurements?
- ▶ Context: Registering measurements performed by hospital staff
- ▶ Issues:
 - ▶ Different units may be used
 - ▶ Accuracy is essential
 - ▶ Record exactly what was measured (use original units delivered)
 - ▶ Motivation: Converting to a different unit a wrong value in one order of magnitude ("Her temperature is 3.71 degrees") reduces the chance to trace the error.



Quantity: wrong solution

- ▶ Model the measurement using an attribute of the object involved; either

- ▶ Use only numbers:

- ▶ height: int

- ▶ Introduce the unit in the attribute name:

- ▶ height_in_inches: int

- ▶ What's wrong with this solution?

- ▶ Low level of abstraction

- ▶ Mistakes not easily discovered

- ▶ What does it mean to say my height is 6 or that my weight is 185? We need units.

- ▶ One solution is to introduce the unit into the attribute name eg. 'weight in pounds'. But what happens if someone gives me

▶¹⁴ their weight in kilos?

Person

height: int

weight: float

blood glucose level: float

Quantity: solution

- ▶ Combine unit and amount in a single concept *Quantity*:

Person

height: Quantity

weight: Quantity

blood glucose level: Quantity

Quantity

amount: long

units: Unit

+ * - / = < > (allowed ops.)

- ▶ **Modeling Principle:** “When multiple attributes interact with behavior that might be used in several types, combine the attributes into a new fundamental type.”

(This also maps well to an implementation in an OO language, such as Java)

Quantity: sample object

- ▶ Representing John's attributes:
 - ▶ A weight of 185 pounds:
 - ▶ amount: 185
 - ▶ unit: pound
 - ▶ 15 American dollars:
 - ▶ amount: 15
 - ▶ unit: USD

Person

```
height: Quantity
weight: Quantity
blood glucose level: Quantity
```

Example continued: *Quantity Conversion*

- ▶ Context:
 - ▶ Measurements are recorded in different units
 - ▶ Different measurements are combined in a single operation
- ▶ Issues
 - ▶ Do not affect the original measurement
 - ▶ For example: 17 mm + 1.3 inch

Conversion: wrong solution

- ▶ Convert everything to a single unit
- ▶ For example:
 - ▶ Measurement of temperature: Celsius or Fahrenheit
 - ▶ At 7am Nurse Betty measured Alice's temperature: 37.4° Celsius
 - ▶ at 8am Nurse John measured Alice's temperature: 102.2° Fahrenheit
 - ▶ Ask the application to calculate the 'spike' (increase) in Alice's temperature between the measurements
- ▶ Procedure:
 - ▶ Convert 102.2° Fahrenheit $\rightarrow 39^{\circ}$ Celsius
 - ▶ Represent in records only the converted temperature 39° Celsius
- ▶ What's wrong with this solution?
 - ▶ forces uses to only take (or convert them to) measurements in a single unit.

Conversion : Solution

- ▶ Solution: Introduce 'Conversion' type
 - ▶ Converts objects from one unit to another



Example 1. We can convert between inches and feet by defining a conversion ratio from feet to inches with the number 12.

Example 2. We can convert between inches and millimeters by defining a conversion ratio from inches to millimeters with the number 25.4. We can then combine this ratio with the conversion ratio from feet to inches to convert from feet to millimeters

Conversion : sample objects

- ▶ Quantity 1: Nurse Betty's measurement of Alice's temperature:
 - ▶ Unit: Celsius
 - ▶ Number: 37.4
- ▶ Quantity 2: Nurse John's measurement of Alice's temperature
 - ▶ Unit: Fahrenheit
 - ▶ Number: 102.2
- ▶ Conversion:
 - ▶ $C = (5/9) \times (F - 32)$
 - ▶ $F = (9/5) \times C + 32$



Example 3: *Measurement*

- ▶ Context: There are thousands of quantities of different phenomenon types
 - ▶ Height, weight, blood glucose level, heart rate, blood pressure, liver performance, lung capacity, white blood cell count, red blood cell count, ...
- ▶ Question: How to represent all the different measurements of a single patient?

Measurement: wrong solution

- ▶ Possible solution: A very large type
- ▶ What's wrong with this solution?
 1. Missing data: who measured? when?
 2. Different measurements at different times
 3. Inflexible to adding new measuring equipment

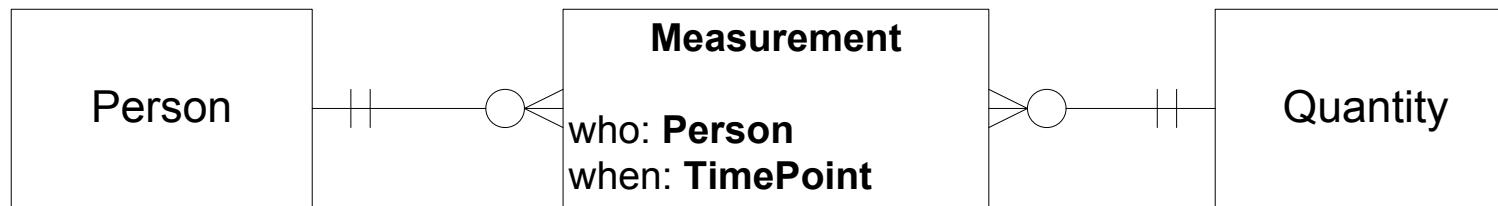
This is part of the analysis problem of when to model something as an attribute or an association

Person

height: quantity
weight: quantity
blood glucose level: quantity
heart rate: quantity
blood pressure: quantity
liver performance: quantity
lung capacity: quantity
white blood cell count: quantity
red blood cell count: quantity
...

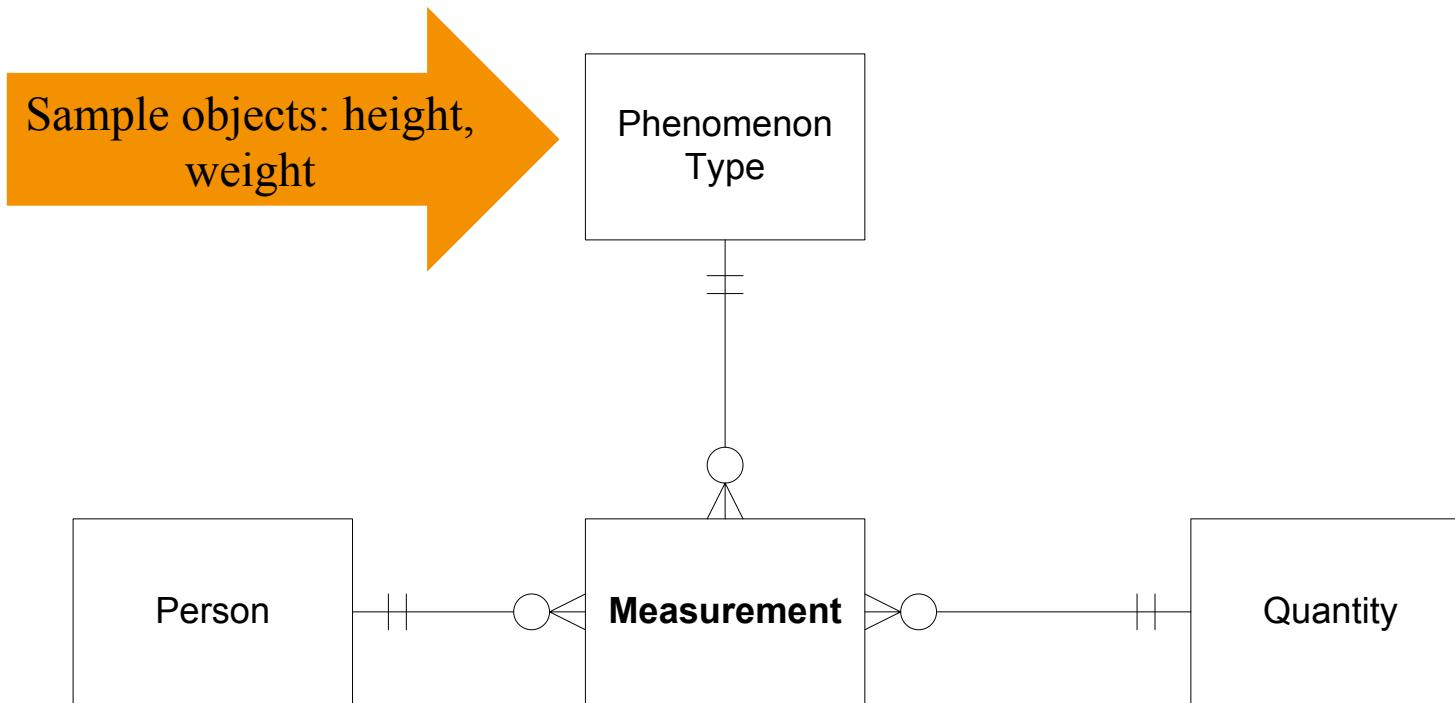
Measurement: solution, part 1

- ▶ Introduce a Measurement type
- ▶ Contains: details of the act of measurement
 - ▶ who performed it
 - ▶ when it took place
 - ▶ which patient
 - ▶ ...



Measurement: Solution, part 2

- ▶ Identify measurement types with a **PhenomenonType**



This model/pattern is useful if a large number of possible measurements would make person too complex. The phenomenon types are things we know we can measure. Such knowledge is at the **analysis/knowledge** level of the problem being investigated.

Measurement: Sample objects

- ▶ “John Smith is 6 feet tall”: measurement of
 - ▶ person: John Smith
 - ▶ phenomenon type: height
 - ▶ quantity:
 - ▶ amount: 6
 - ▶ unit: feet
- ▶ “John Smith has PEF rate of 180 liters per minute”: measurement of
 - ▶ person: John Smith
 - ▶ phenomenon type: “peak expiatory flow”
 - ▶ quantity:
 - ▶ amount: 180
 - ▶ unit: compound unit
 - direct: liter
 - inverse: minute

Modeling principles for this pattern

- ▶ Measurements are created as part of the day to day operation and will change frequently (operational level)
 - ▶ Phenomenon types (eg. blood pressure) are created infrequently (knowledge level)
-
- ▶ **Principle 1:** The operational level has those concepts that change on a day to day basis. Their configuration is constrained by a knowledge level that changes much less frequently
 - ▶ **Principle 2:** If a *type* has many similar associations, make all these associations objects of a new type. Create a knowledge level type to differentiate between them.
-

Handling Observations

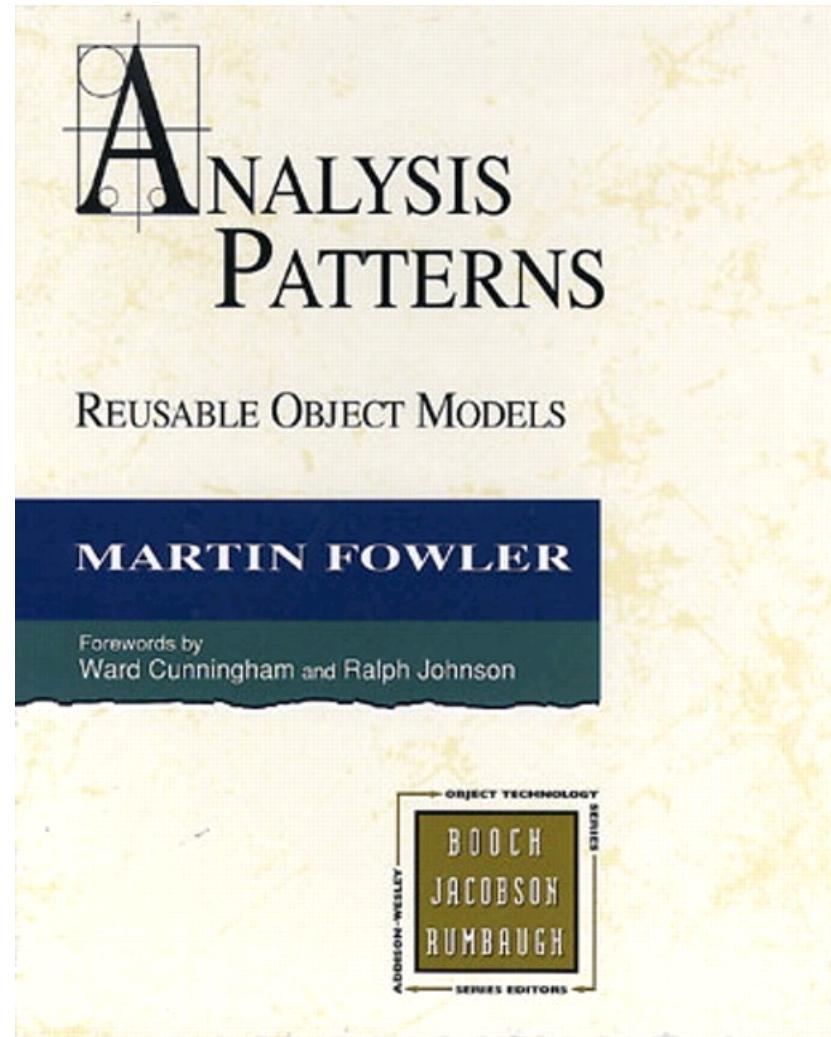
- ▶ Based on principle 2:
 - ▶ **Principle 2:** If a *type* has many similar **associations**, make all these associations objects of a **new type**. Create a knowledge level type to differentiate between them.
 - ▶ A **person** has many similar **measurements**, therefore make a new type '**Measurement**' to differentiate between them.
- ▶ In the class we will look at how this pattern can be extended to also handle qualitative information such as observations

Summary

- ▶ In analysis you may need to look behind the surface requirements (rather than just listing requirements in use-cases)
- ▶ So we need to create conceptual models that allow us to understand and simplify the problem
- ▶ Type diagrams are one way of creating these conceptual models
- ▶ We looked at examples of using Type diagrams to model observations and measurements
- ▶ These models can be reused as analysis patterns for other similar problems

OOA with Type Diagrams

- ▶ M. Fowler (1997). *Analysis Patterns*. Addison-Wesley.
- ▶ Mainly *Chapter 3: Observations and Measurements*
- ▶ Martin, J & Odell J (1995). *Object-oriented methods: A foundation*. Prentice-Hall.



Principles of Object-Orientation

- Encapsulation
- Inheritance
- Polymorphism

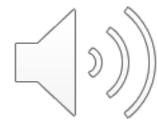


CE202 Software Engineering, Autumn term

Dr Cunjin Luo, School of Computer Science and Electronic Engineering, University of Essex

Object-orientation

- ▶ So far we have briefly touched on how analysis and design (and implementation) can make use of objects
 - ▶ Eg. objects in activity diagrams, class diagrams, etc
- ▶ In this lecture we will explore common object-oriented principles
- ▶ As we get more into UML we will make more use of objects



Advantages of O-O

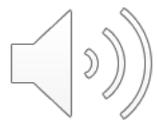
- ▶ Can save effort
 - ▶ Reuse of generalized components cuts work, cost and time
- ▶ Can improve software quality
 - ▶ Encapsulation increases modularity
 - ▶ Sub-systems less coupled to each other
 - ▶ Better translations between analysis and design models and working code
 - ▶ Objects are good for modelling what happens in the real world
 - ▶ Can be used throughout the software lifecycle ie.
requirements -> design -> implementation -> testing



OO Analysis & Design: mechanisms of abstraction

- ▶ Fundamentally: the same abstraction mechanisms as object-oriented programming:
 - ▶ Encapsulation: classes and objects
 - ▶ Other possible modularization techniques: interfaces, packages/namespaces
 - ▶ Inheritance
 - ▶ Generalization/specialization
 - ▶ Subtyping
 - ▶ Subclassing
 - ▶ Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding





Encapsulation

Objects

An object is:

“an abstraction of something in a problem domain,
reflecting the capabilities of the system to

- ▶ keep information about it,
- ▶ interact with it,
- ▶ or both.”

Coad and Yourdon (1990)



Objects

“Objects have state, behaviour and identity.”

Booch (1994)

- ▶ *State*: the condition of an object at any moment, affecting how it can behave
- ▶ *Behaviour*: what an object can do, how it can respond to events and stimuli
- ▶ *Identity*: each object is unique



Examples of Objects

Object	Identity	Behaviour	State
A person	'Hussain Pervez.'	Speak, walk, read.	Studying, resting, qualified.
A shirt	My favourite button white denim shirt.	Shrink, stain, rip.	Pressed, dirty, worn.
A sale	Sale no #0015, 18/05/05.	Earn loyalty points.	Invoiced, cancelled.
A bottle of ketchup	<i>This</i> bottle of ketchup.	Spill in transit.	Unsold, opened, empty.

Can you suggest other behaviours and states for these objects?

How can external events and object behaviour both result in a change of state?

How can state restrict the possible behaviours of an object?

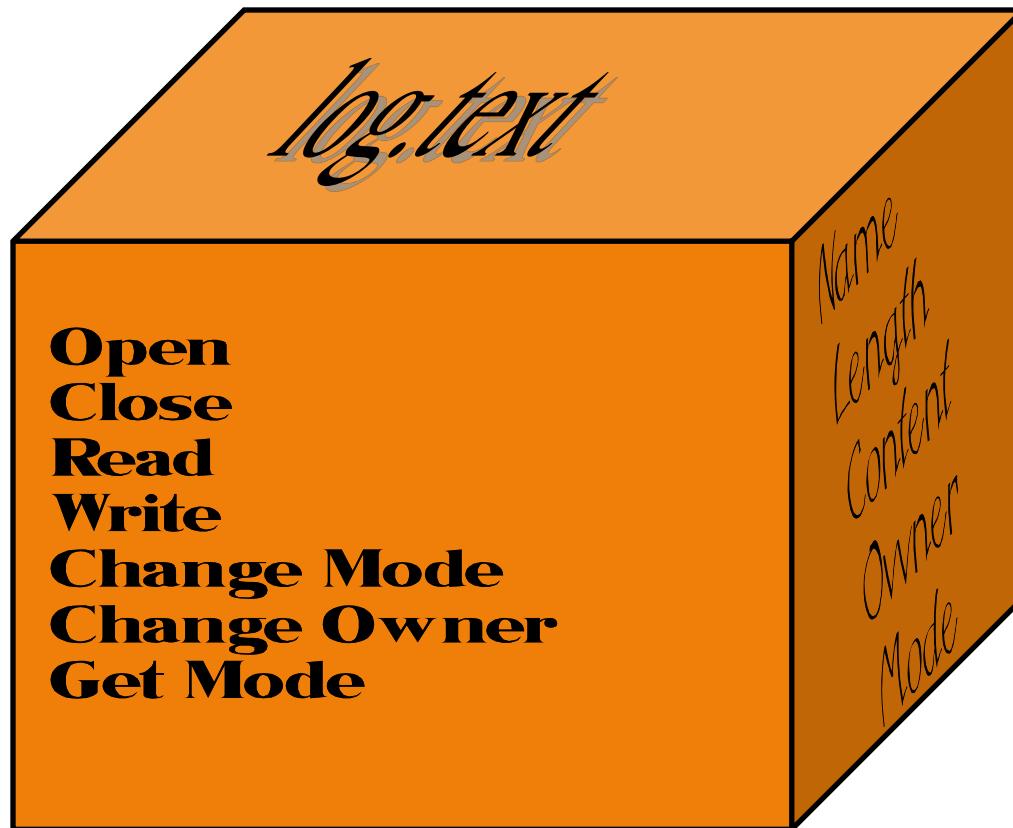
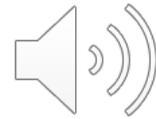


Examples of objects (cont.)

- ▶ Examples:
 - ▶ Time point
 - ▶ Data: 16:45:00, Feb. 21, 1997
 - ▶ Operations: add time interval, calculate difference from another time point,
 - ▶ Acts. For example: Measurement of a patients' fever
 - ▶ Data: 37.1°C, by Deborah, at 10:10 am
 - ▶ Operations: Print, update, archive
 - ▶ File
 - ▶ Data: log.txt, -rwx-----, Last read 21:07 June 1, 1999, ...
 - ▶ Operations: read, write, execute, remove, change directory, ...
 - ▶ A communication event (time, length, phone-number, ...)
 - ▶ Transaction in a bank account (withdraw \$15, time, ...)
 - ▶ Elements of ticket machine dispenser: ticket, balance, zone, price, ...

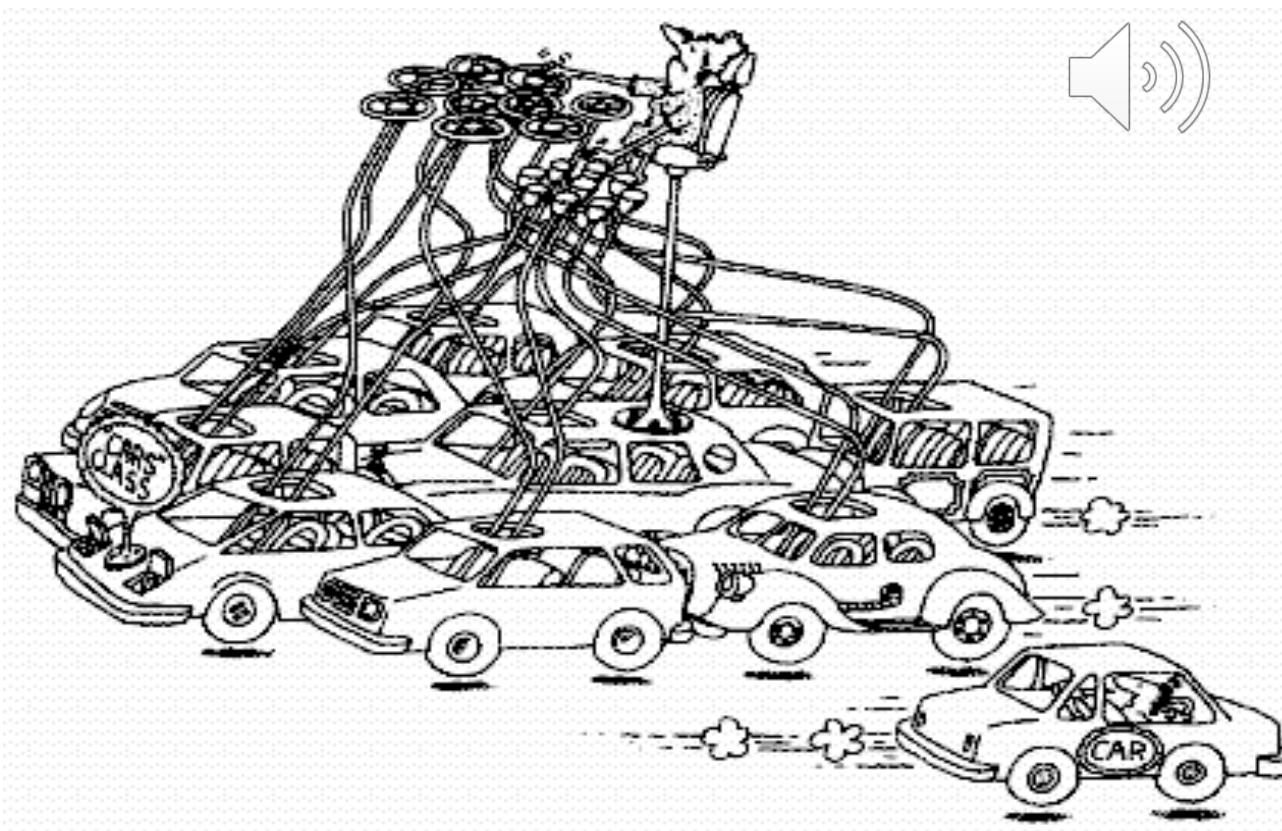


Object: Example



Class: Abstraction Over Objects

- ▶ A class represents a set of objects that share a common structure and a common behavior.



Class and Instance

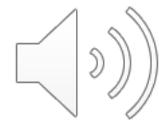
- ▶ All objects are *instances* of some *class*
- ▶ A Class is a description of a set of objects with similar:
 - ▶ features (attributes, operations, links);
 - ▶ semantics;
 - ▶ constraints (e.g. when and whether an object can be instantiated).

OMG (2009)



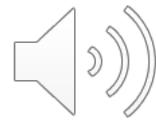
Class and Instance

- ▶ An object is an instance of some class
- ▶ So, instance = object
 - ▶ but also carries connotations of the class to which the object belongs
- ▶ Instances of a class are similar in their:
 - ▶ *Structure*: what they *know*, what information they hold, what links they have to other objects
 - ▶ *Behaviour*: what they *can do*

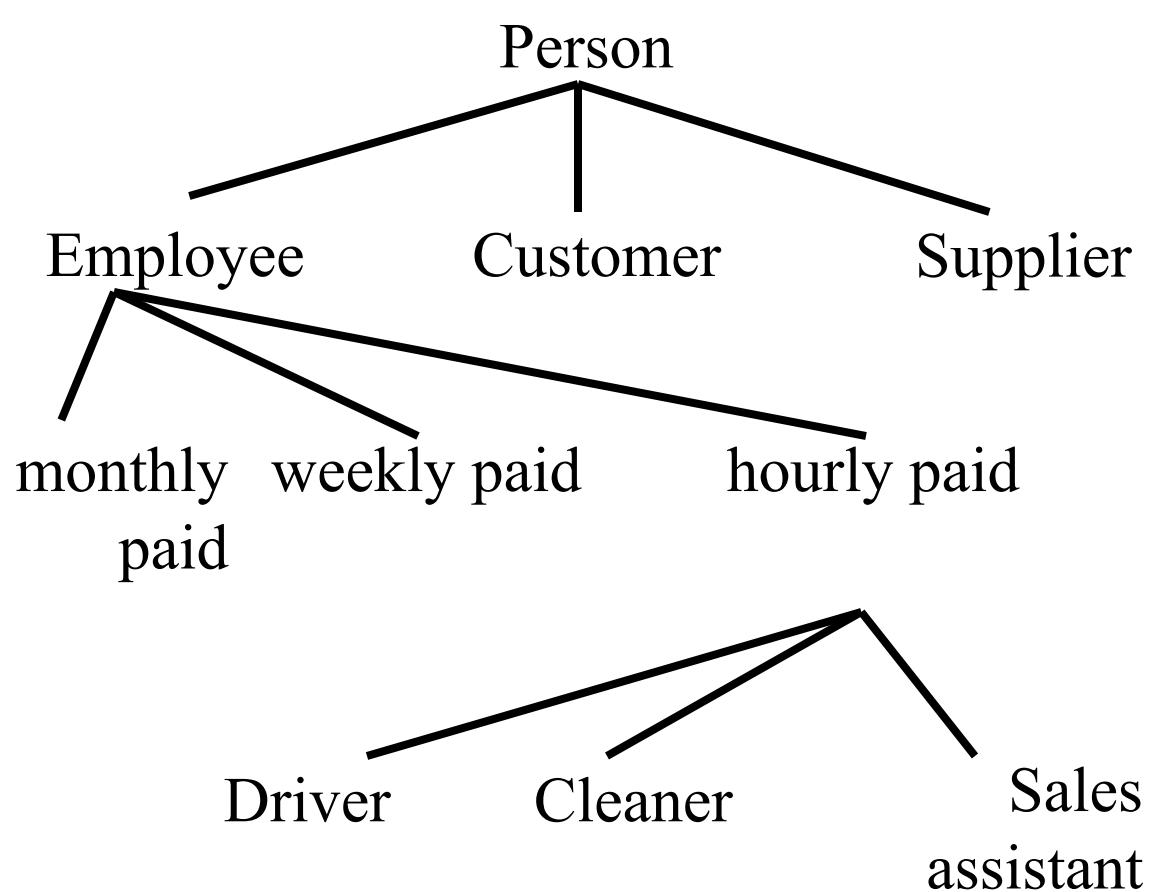


Generalization and Specialization

- ▶ Classification is hierachic in nature
- ▶ For example, a person may be an employee, a customer, a supplier of a service
- ▶ An employee may be paid monthly, weekly or hourly
- ▶ An hourly paid employee may be a driver, a cleaner, a sales assistant



Specialization Hierarchy



More general
(superclasses)



More specialized
(subclasses)



Generalization and Specialization

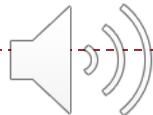
- More general bits of description are *abstracted out* from specialized classes:

SystemsAnalyst

name
employee-no
startDate
monthlySalary
grade

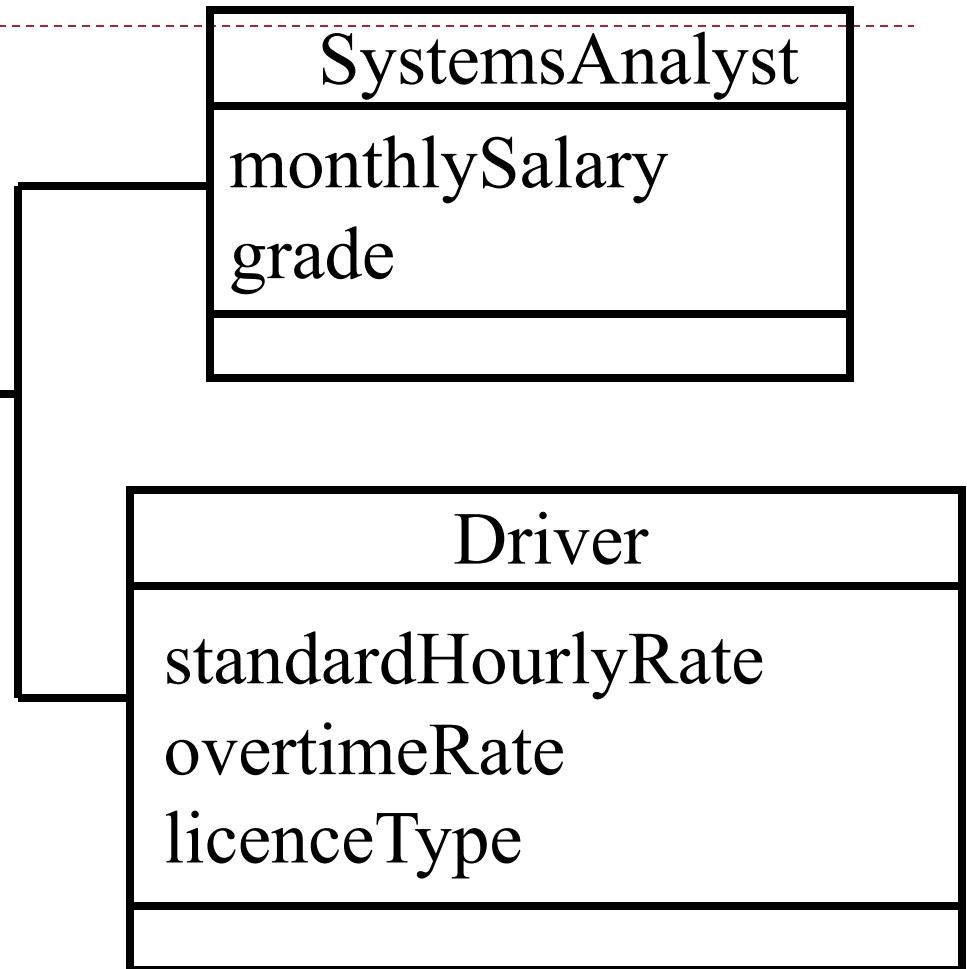
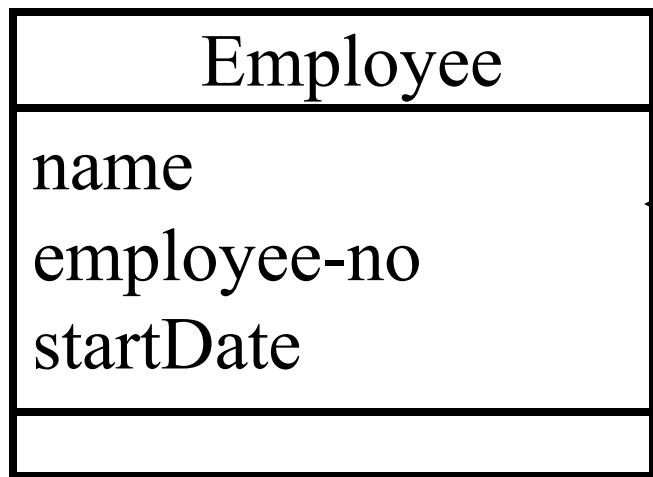
Driver

name
employee-no
startDate
standardHourlyRate
overtimeRate
licenceType

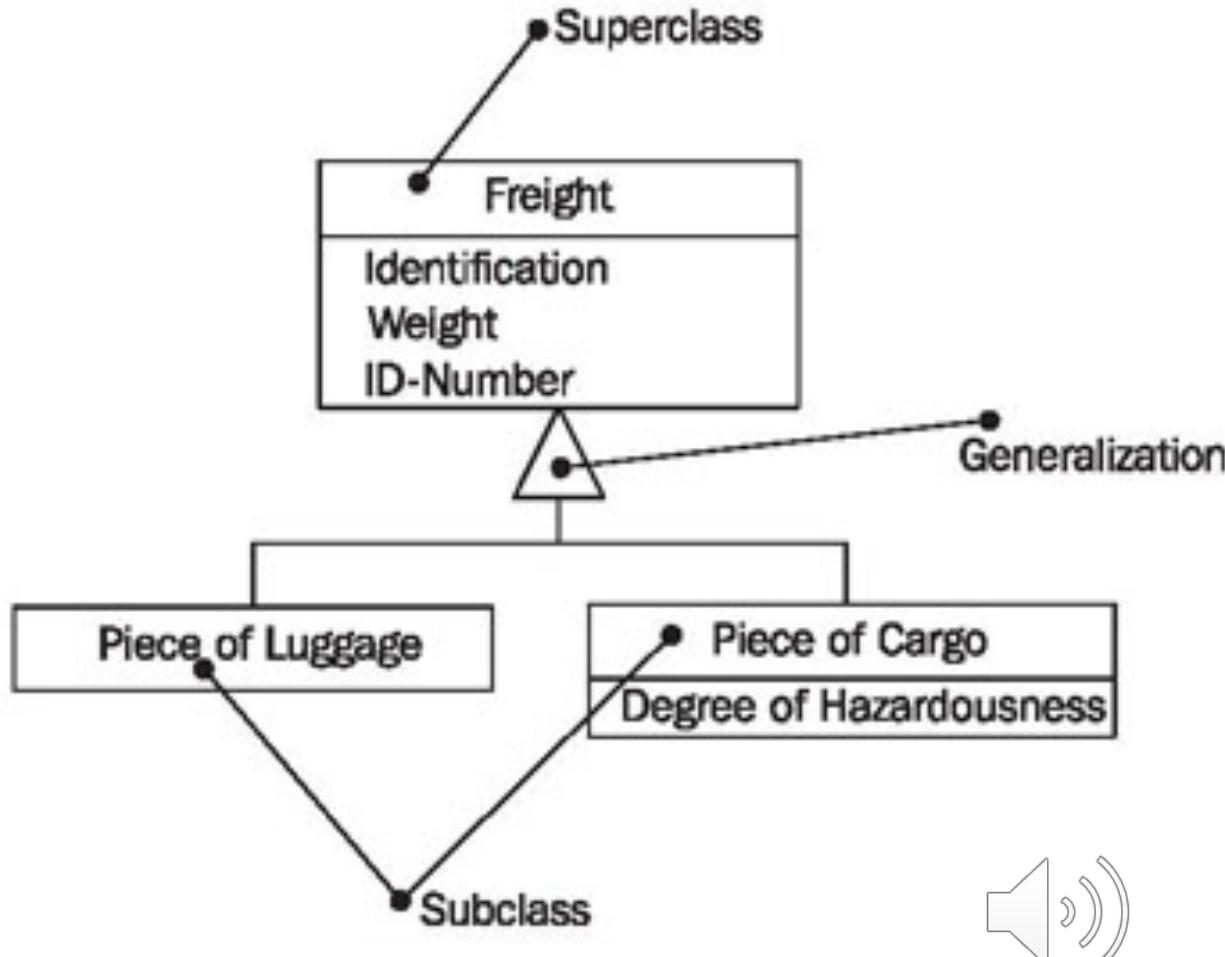


Specialized (subclasses)

General (superclass)

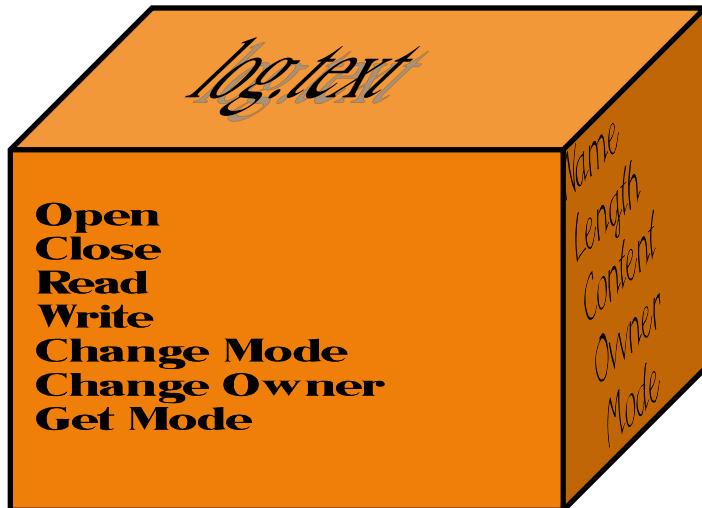


Object-oriented modelling notation: genericity in class diagrams



Encapsulation definition

- ▶ Encapsulation: an object's data is located with the operations that use it



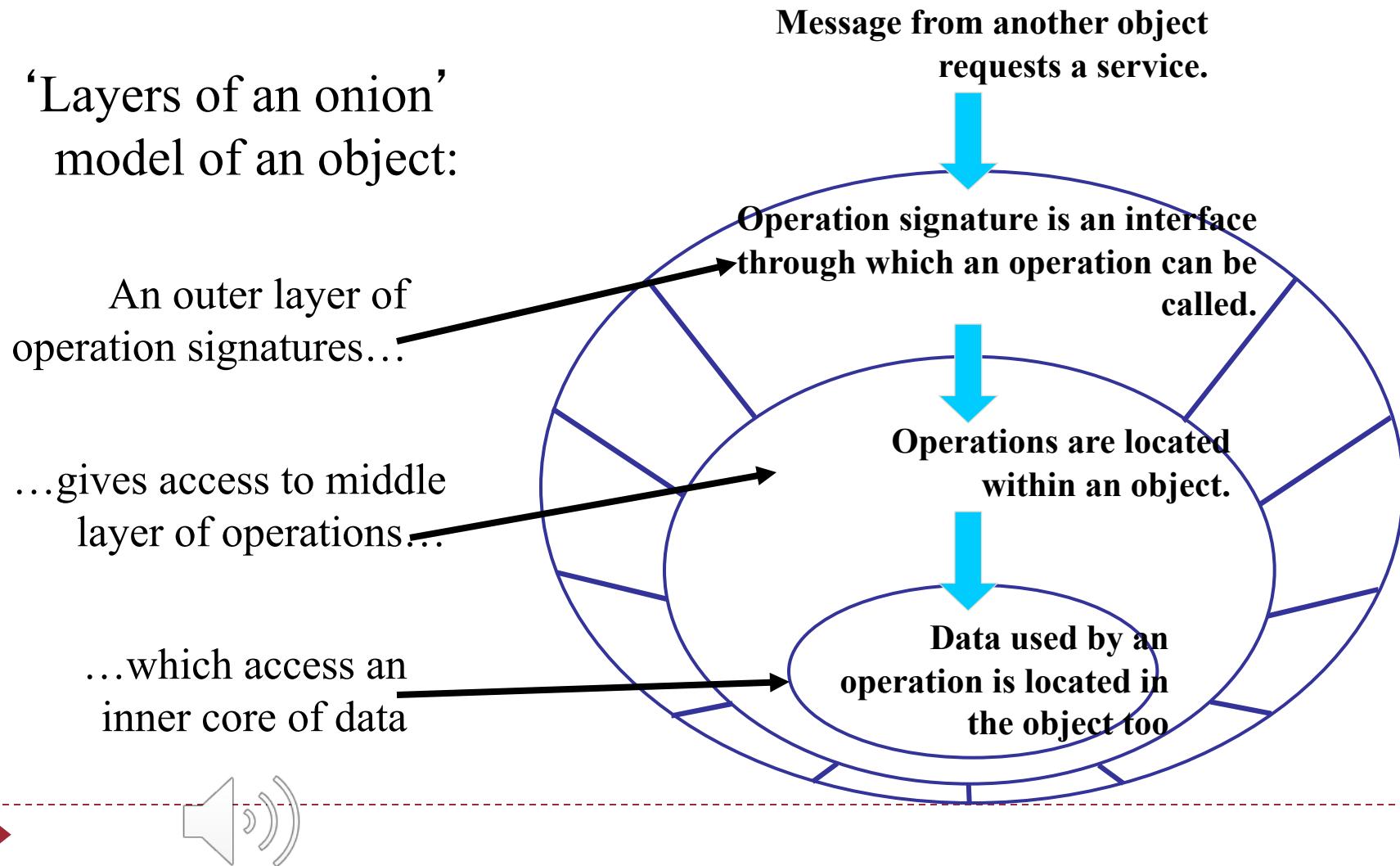
Message-passing

- ▶ Several objects may collaborate to fulfil each system action/use-case
- ▶ “Record CD sale” could involve:
 - ▶ A CD stock item object
 - ▶ A sales transaction object
 - ▶ A sales assistant object
- ▶ These objects communicate by sending each other messages
- ▶ We will model these messages when we look at sequence and collaboration diagrams



Message-passing and Encapsulation

‘Layers of an onion’
model of an object:



Classes

- ▶ Kinds of classes:
 - ▶ Abstract (also: Java interface)
 - ▶ No instances!
 - ▶ Serve as an interface, or a common base with similar behavior
 - ▶ Example: Collection (Smalltalk, Java)
 - ▶ Singleton
 - ▶ One instance!
 - ▶ Example: True (Smalltalk)
 - ▶ Language support for singletons:
 - Self: No classes! All objects are singletons
 - BETA: Objects may be defined as singletons
 - ▶ Concrete/effective
 - ▶ Any number of instances

HOW TO SLAUGHTER A PIG



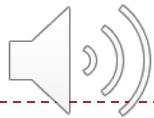
Classes in O-O programming languages

► C++

```
class Employee: public Person {  
private:                                // visible to none  
    Date birthday;                      // data member  
public:                                 // visible to all  
    void hire(Reason why)    // function member  
    { /* ... */ }  
};
```

► Java:

```
class Employee extends Person {  
    private Date birthday;                // field, visible to none  
    public void Hire(Reason why) {        // method, visible to all  
        { /* ... */ }  
    }  
}
```



Classes in OOPLs (Cont.)

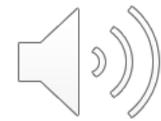
► Smalltalk:

```
Person subclass: Employee
instanceVariables: 'birthday' "instance variable, visible to none"
classVariables: ''
poolDictionaries: ''

hire: why           "method, visible to all"
"..."
```

► Eiffel:

```
class EMPLOYEE
inherit
    PERSON
feature {NONE}          -- visible to none
    DATE birthday;        -- attribute
feature {ALL}            -- visible to all
    hire(why: REASON) is -- routine
        do
            -- ...
        end
end -- class EMPLOYEE
```



Information Hiding

Information Hiding: the onion model

‘Layers of an onion’
model of an object:

Only the outer layer is
visible to other objects...

...and it is the only way to
access operations...

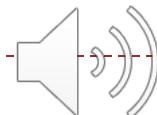
...which are the only way
to access the hidden data

Message from another object
requests a service.

Operations can *only* be called
by message with valid
operation signature.

Only object’s own operations can
access its data.

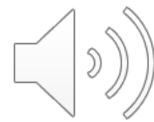
Representation of data is
hidden inside object



Note:

Information Hiding Vs. Encapsulation

- ▶ The terms are sometimes confused and used interchangeably
 - ▶ Some people say “encapsulation” with reference to what we described as information hiding
- ▶ We shall adhere to the interpretation in these slides
 - ▶ Encapsulation: an object's data is located with the operations that use it
 - ▶ **Information hiding: only an object's Interface is visible to other objects**



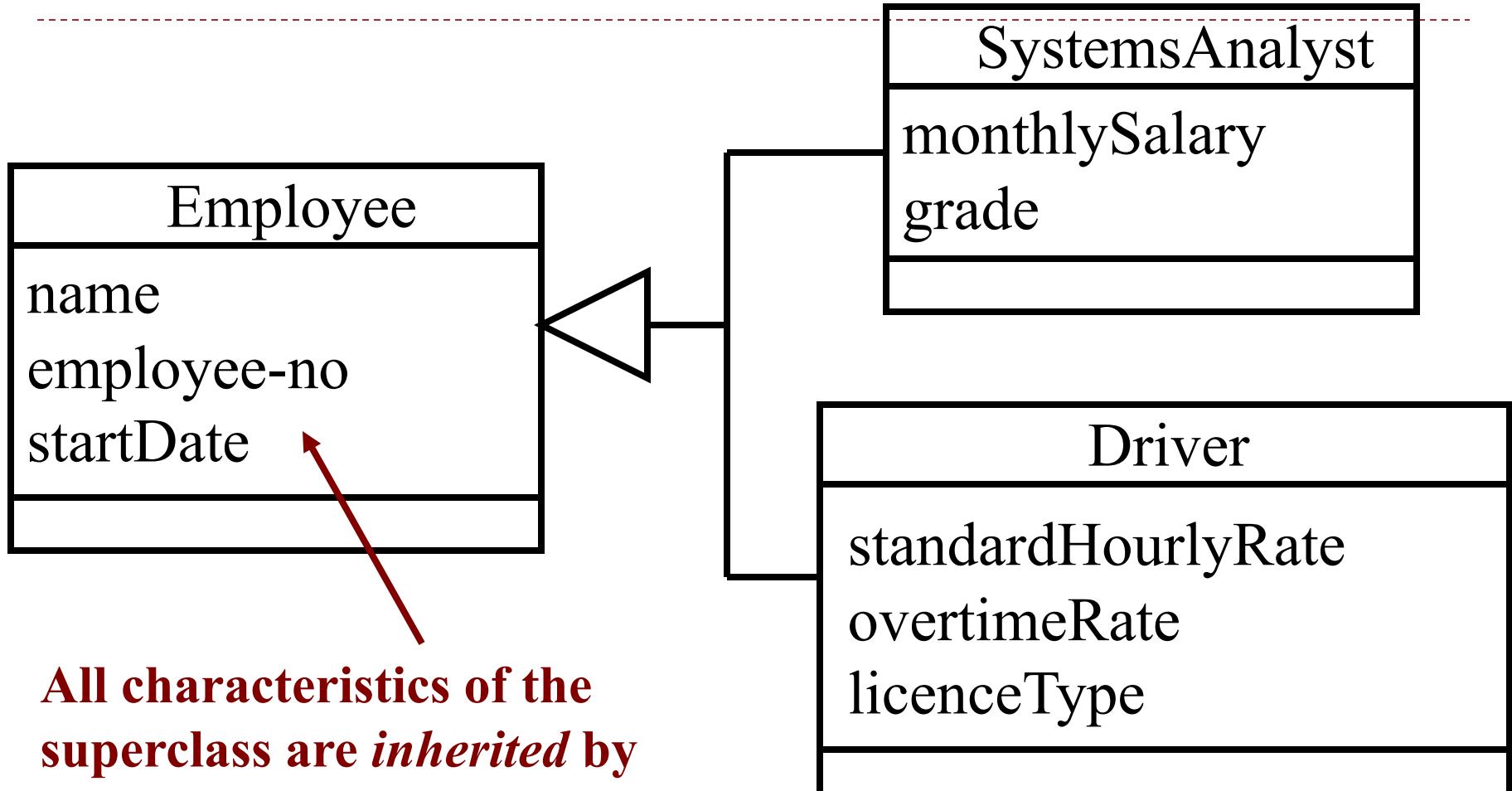


Inheritance

Inheritance

- ▶ The *whole* description of a superclass applies to *all* its subclasses, including:
 - ▶ Information structure (including associations)
 - ▶ Behaviour
- ▶ Often known loosely as *inheritance*
- ▶ (But actually inheritance is how an O-O programming language *implements* generalization / specialization)

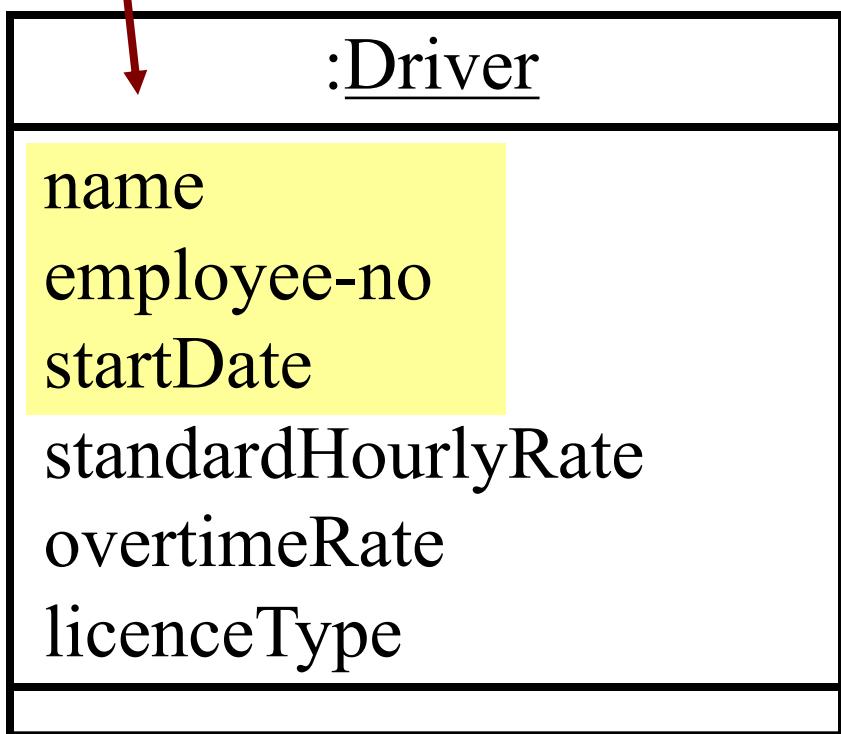




All characteristics of the superclass are *inherited* by its subclasses



Instances of each subclass include the characteristics of the superclass (but not usually shown like this on diagrams)



Inheritance

- ▶ Represents separate notions:
 - ▶ Generalization: ‘is-kind-of’ relation
 - ▶ Instances of the specialized class are a subcategory of the generalized class
 - ▶ Subtyping (in Java: ‘implements’)
 - ▶ The subtypes supports all the operations on supertype
 - ▶ Subclassing (in Java: extends)
 - ▶ A mechanism of code reuse



Inheritance and generalization

- ▶ Superclass is a generalization of Subclass
 - ▶ Also: subclass is a specialization of superclass
- ▶ Also known as: is-kind-of relation:
 - ▶ Staff is-kind-of Person
 - ▶ Instructor is-kind-of Person



Person
name
address
change_sex()

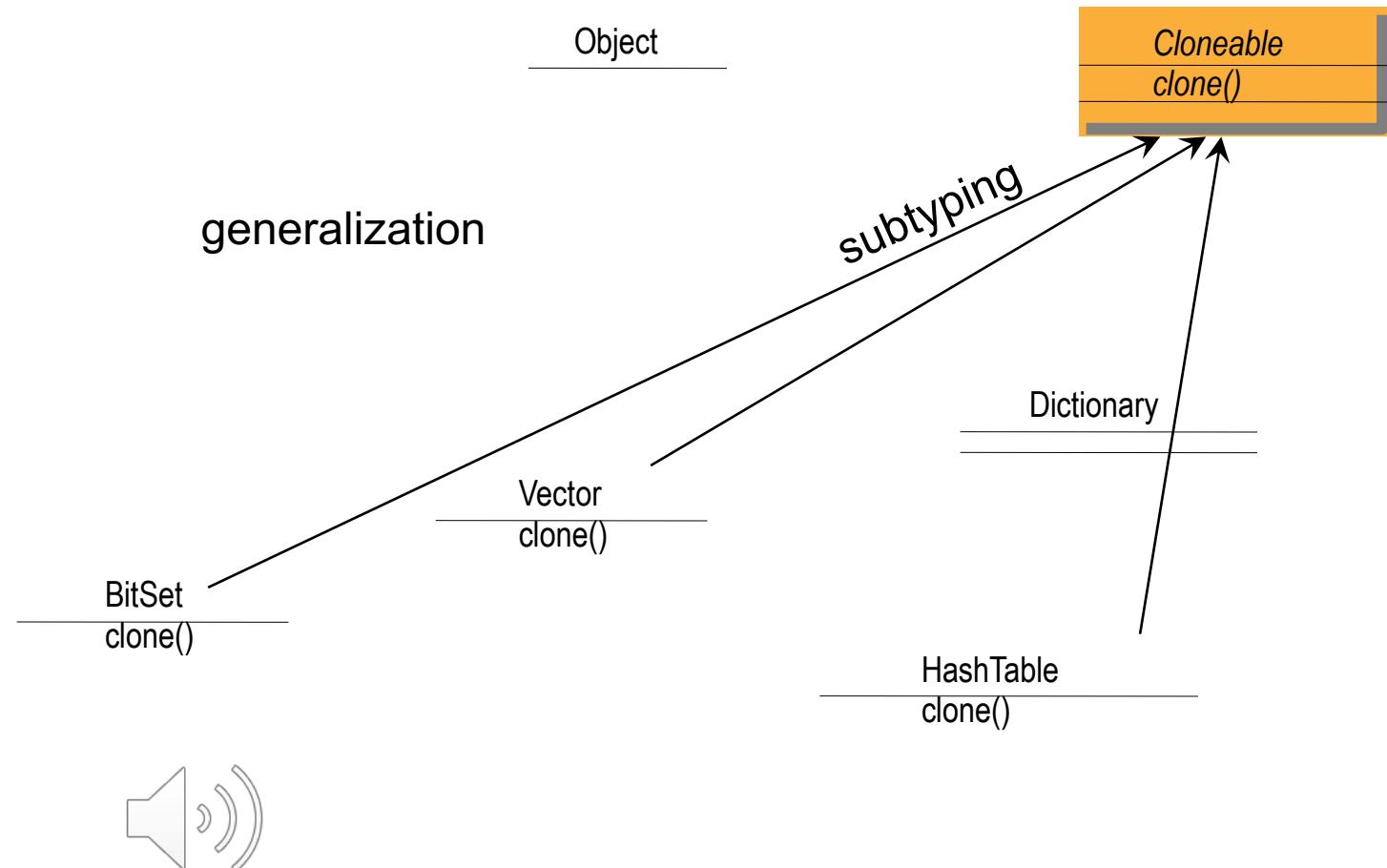
Student
tuition_fee
graduate(average)

Staff
hired : Date

Instructor
specialty

Inheritance and subtyping

- ▶ Subtype supports the same operations as supertype



Inheritance and subclassing

- ▶ Inheritance can be used as a crude mechanism of reuse
- ▶ A very problematic and dangerous tactic

Person

name

address

change_sex()

Student

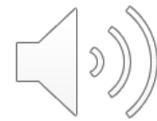
tuition_fee

graduate(average)



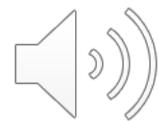
In this session we have looked at

- ▶ What is an object
- ▶ What is a class
- ▶ Advantages of OO
- ▶ OO mechanisms of
 - ▶ Encapsulation/Information Hiding
 - ▶ Inheritance
- ▶ In the class we will look at Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity



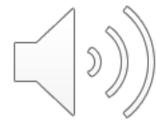
Exercises: OO inheritance

- ▶ What rules describe the relationship between a subclass and its superclass?
- ▶ For each one of the following class pairs, determine the appropriate kind of inheritance relation between them:
 - ▶ Person, Parent
 - ▶ Person, Mammal
 - ▶ Bird, FlyingObject
- ▶ What is the difference between generalization and sub-typing?



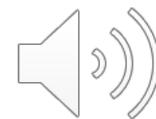
Exercises: OO inheritance

- ▶ What rules describe the relationship between a subclass and its superclass?
- ▶ For each one of the following class pairs, determine the appropriate kind of inheritance relation between them:
 - ▶ Person, Parent
 - ▶ Person, Mammal
 - ▶ Bird, FlyingObject
- ▶ What is the difference between generalization and subtyping?



Summary

- ▶ OO Analysis & Design has same mechanisms as OO programming
- ▶ OO provides many benefits
- ▶ Difference between Objects and Classes
- ▶ Looked at core OO concepts:
 - ▶ Encapsulation
 - ▶ Information Hiding
 - ▶ Inheritance
 - ▶ Generalization/specialization
 - ▶ Polymorphism
 - ▶ Overloading
 - ▶ Dynamic binding
 - ▶ Genericity





Further reading

- ▶ Object-orientation
 - ▶ Chapter 4, Bennett
 - ▶ Coad, P & Yourdan, E (1990) Object-oriented analysis. Prentice-Hall.
 - ▶ Booch, G (1994) Object-oriented analysis and design with applications. Menlo Park.
- ▶ Information hiding & abstraction:
 - ▶ Wirfs-Brock, Rebecca, Wilkerson, Brian, and Wiener, Lauren. 'Designing Object-Oriented Software'. Prentice-Hall, 1990.
 - ▶ Parnas, 1985, Communications of the ACM.
- ▶ Look at Java Interfaces
 - ▶ Eg. <http://docs.oracle.com/javase/tutorial/java/concepts/interface.html>
 - ▶ Look at Java abstraction
 - ▶ Eg. <http://javarevisited.blogspot.co.uk/2010/10/abstraction-in-java.html>
- ▶ Exemplar based object-orientation:
 - ▶ 'An exemplar based Smalltalk'. OOPSLA 1986 Proceedings.



Class: Type Diagrams

- Dealing with observations

CE202 Software Engineering, Autumn term

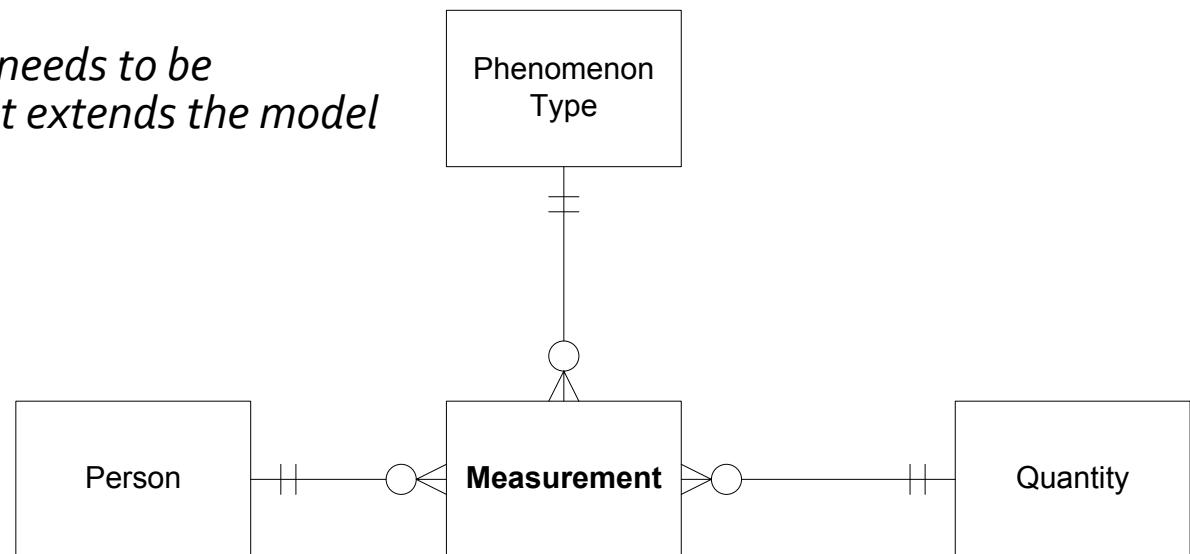
Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Exercise: OOA with Type Diagrams

Observation

- ▶ Reminder: Measurement example
- ▶ Problem: Qualitative observations
 - ▶ Observations are of a discreet, fixed (small) range
 - ▶ Examples: gender{male/female}, blood-type {A/B/AB/O}, has-diabetes{True/False}, ...
 - ▶ They do not fit well into the Quantity idea because value entered can range for no reason
 - ▶ Example: if string is used then Gender = "Male" or "male" or "M" or "Man" or...
- ▶ Suggest a solution for modelling observations faithfully
- ▶ *Hint: the Measurement type needs to be replaced with a new type that extends the model*

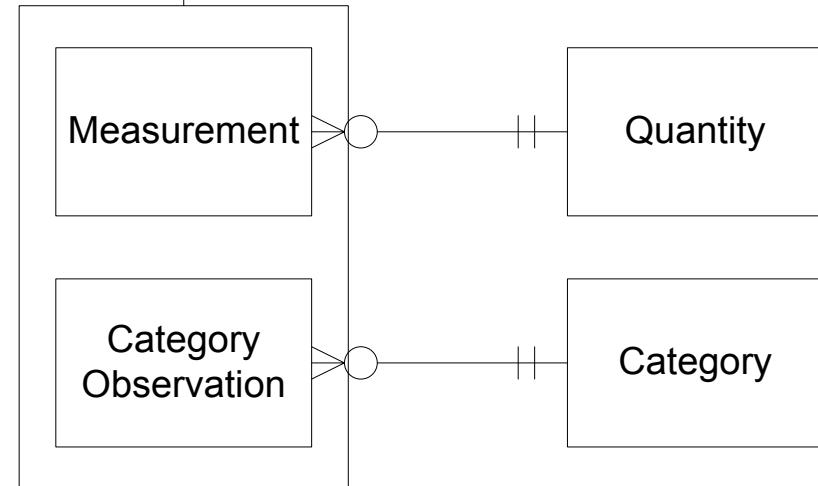


Observation: possible solution



► Sample objects:

- phenomenon-type: Blood-type
 - category: "A/B"
- phenomenon-type: Gender
 - category: Female



1. Blood type is the instance of phenomenon type, and 'A/B' is the instance of the category. To record that a person has a blood type of 'A/B', we create an observation with a category of 'A/B' and a phenomenon type of Blood type.
2. Gender is the instance of phenomenon type, and male and female are instances of category. To record that a person is female , we create an observation with a category of female and a phenomenon type of gender.

Summary

- ▶ Difference and similarities between analysis and design
- ▶ Requirements engineering may need you to create **conceptual models** that allow us to understand and simplify the problem domain
- ▶ Conceptual models can become **reusable analysis patterns**
- ▶ Can use **TYPE diagrams** to create these models
- ▶ Example patterns in the area of observation & measurement
 - ▶ Dealing with quantities
 - ▶ Quantity conversion
 - ▶ Multiple measurements
 - ▶ Handling more general observations

Software analysis and design: realisation

- Why analyse requirements
- Class diagrams

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

Why Analyse Requirements?

- ▶ Requirements (Use Case) model alone is not enough
 - ▶ There may be repetition
 - ▶ Some parts may already exist as standard components
 - ▶ Use cases give little information about structure of software system



The Purpose of Analysis

- ▶ Analysis aims to identify:
 - ▶ A software structure that can meet the requirements
 - ▶ Common elements among the requirements that need only be defined once
 - ▶ Pre-existing elements that can be reused
 - ▶ The interaction between different requirements



What an Analysis Model Does

An analysis model must confirm what users want a new system to do:

- ▶ Understandable for users
- ▶ Correct scope
- ▶ Correct detail
- ▶ Complete
- ▶ Consistent between different diagrams and models



How to Model the Analysis

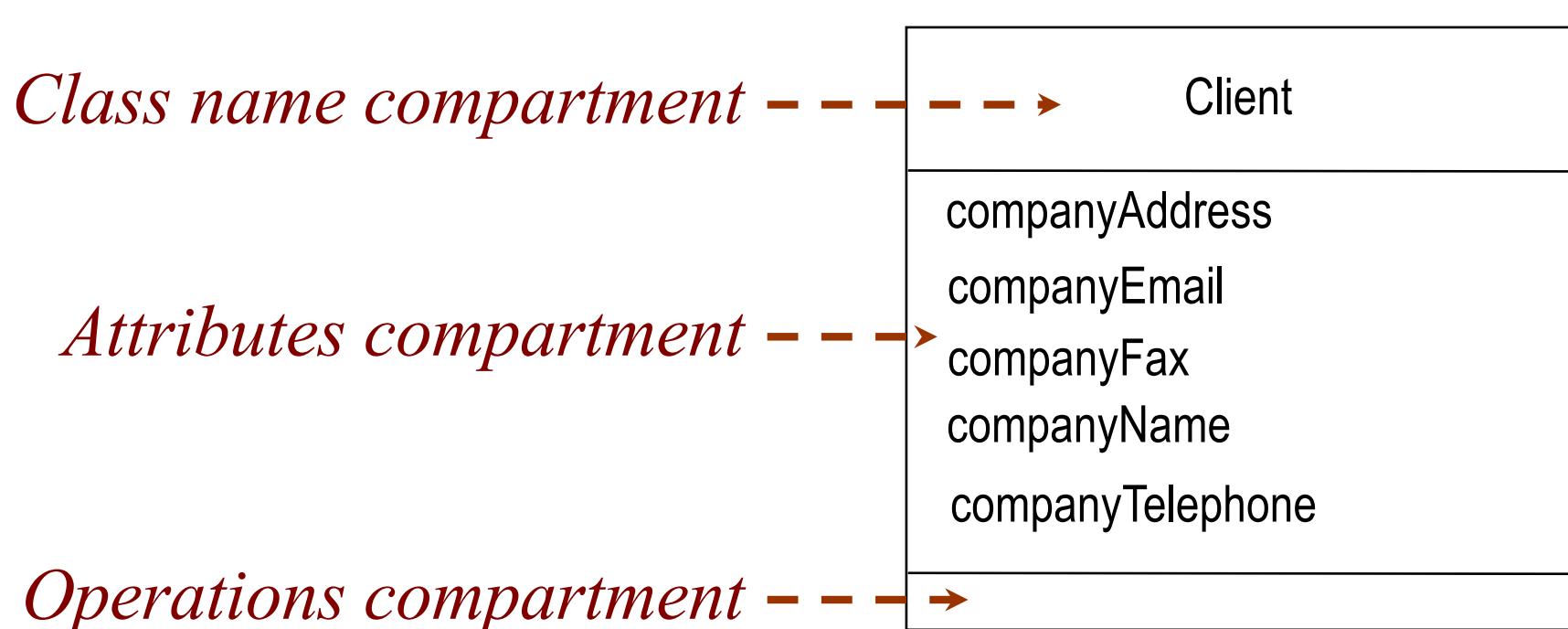
- ▶ The main technique for analysing requirements is the class diagram
- ▶ Two main ways to produce this:
 - ▶ Directly based on knowledge of the application domain (from a Domain Model)
 - ▶ By producing a separate class diagram for each use case, then assembling them into a single model (an Analysis Class Model)
- ▶ We will look at both approaches



Introduction to Class Diagrams

Class Diagram: Class Symbol

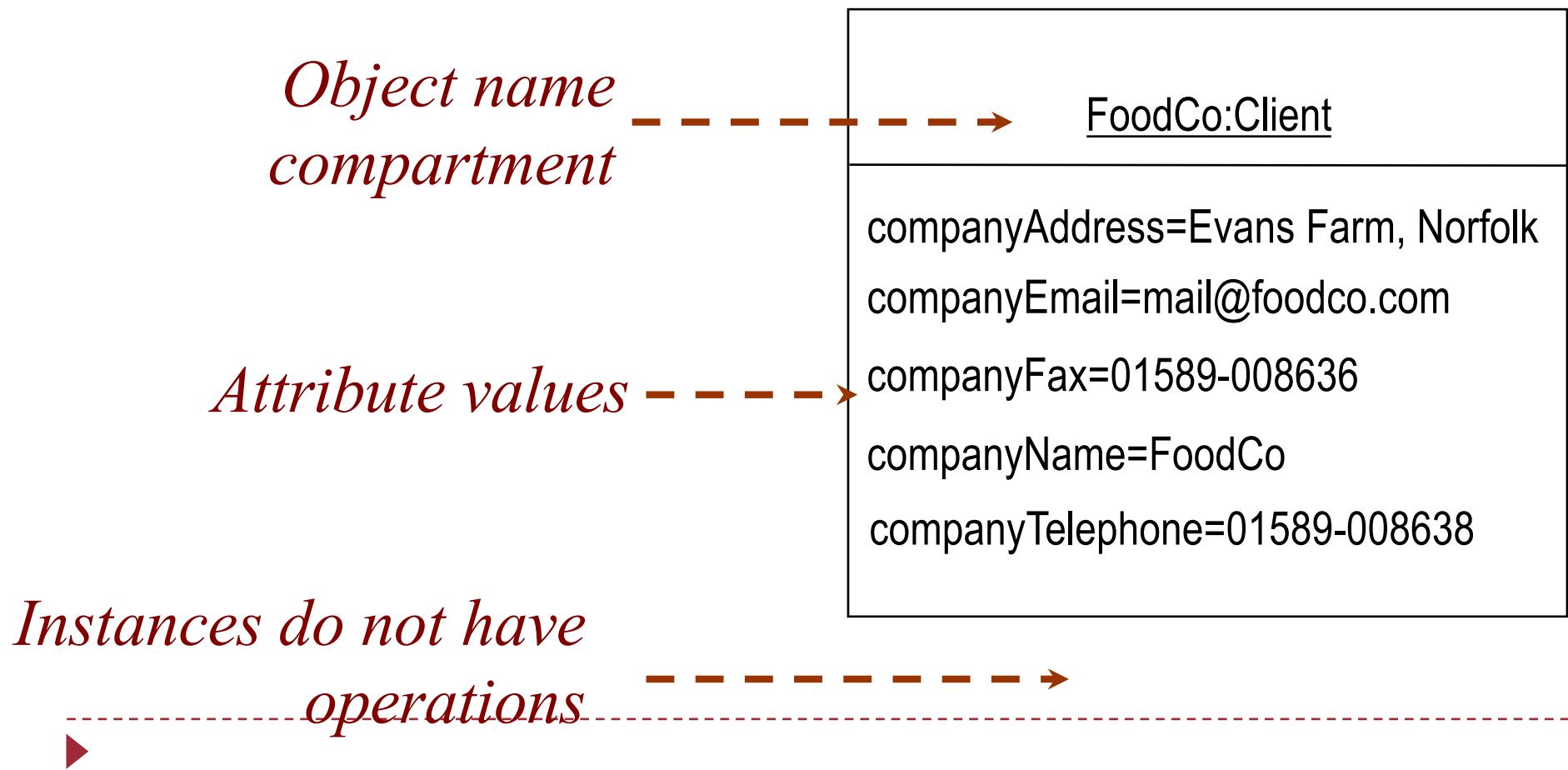
- A Class is “a description of a set of objects with similar features, semantics and constraints” (OMG, 2009)



Some class diagrams may only contain the class name information

Class Diagram: Instances

An object (instance) is: “an abstraction of something in a problem domain...”



Class Diagram: Attributes

Attributes are:

- ▶ Part of the essential description of a class
- ▶ The common structure of what the class can ‘know’
- ▶ Each object has its own *value* for each attribute in its class:
 - ▶ *Attribute= “value”*
 - ▶ *companyName=FoodCo*



Class Diagram: Operations

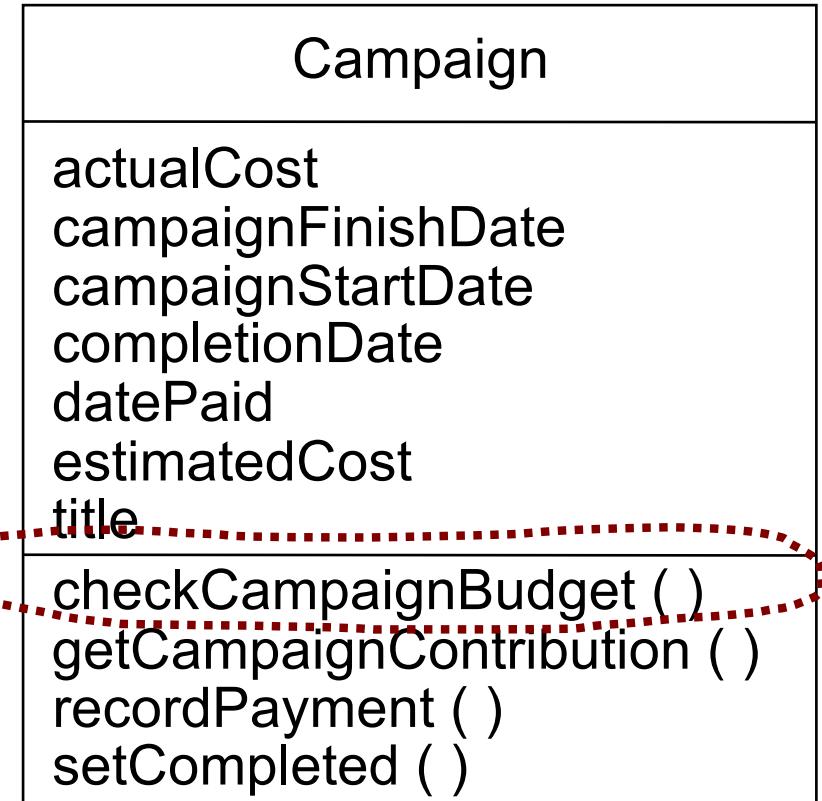
Operations are:

- ▶ An essential part of the description of a class
- ▶ The common behaviour shared by all objects of the class
- ▶ Services that objects of a class can provide to other objects



Class Diagram: Operations

- ▶ Operations describe what instances of a class can do:
 - ▶ Set or reveal attribute values
 - ▶ Perform calculations
 - ▶ Send messages to other objects
 - ▶ Create or destroy links



Class Diagram: Associations

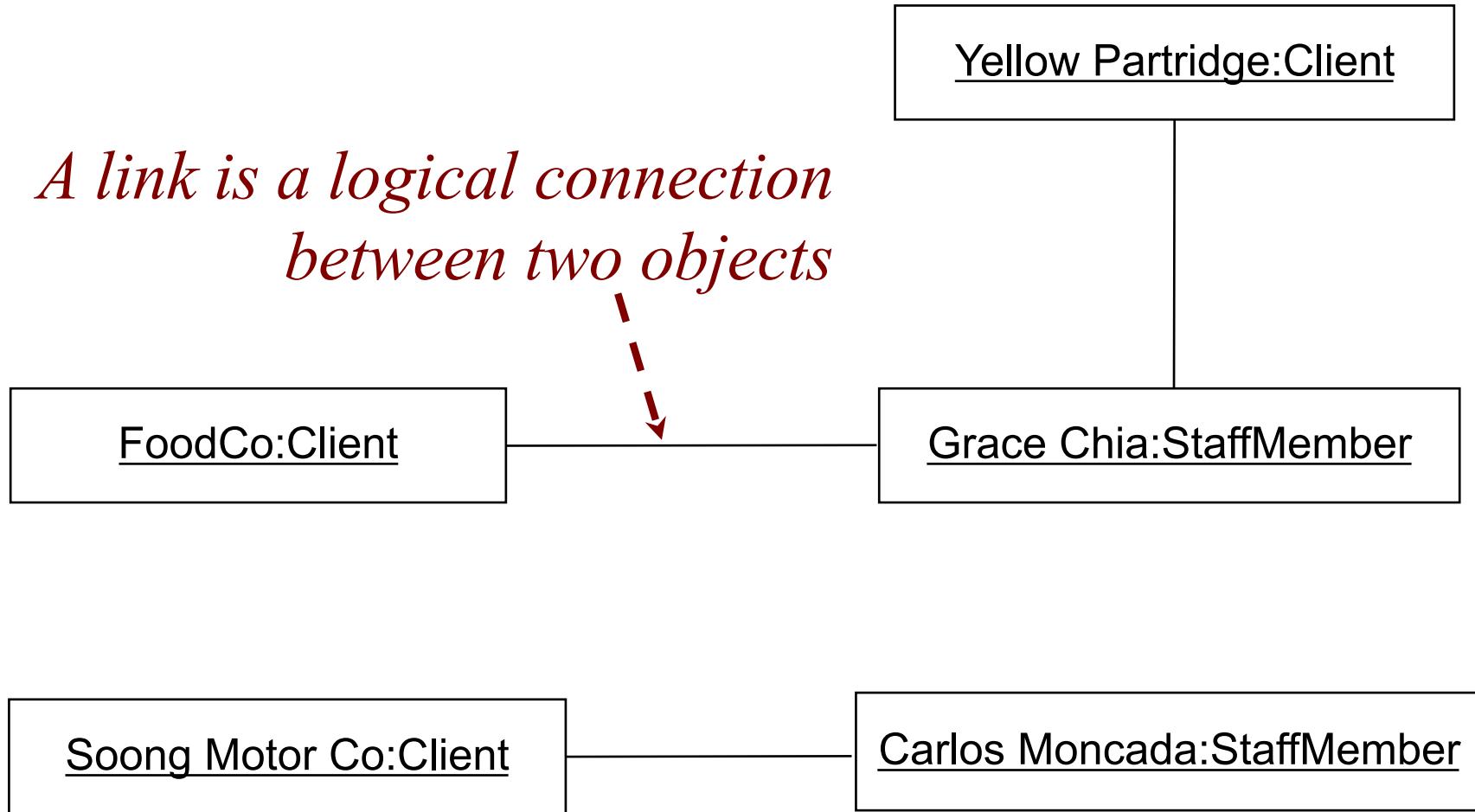
Associations represent:

- ▶ The possibility of a logical relationship or connection between objects of one class and objects of another
 - ▶ “Grace Chia is the staff contact for FoodCo”
 - ▶ *An **employee** object is linked to a **client** object*
- ▶ If two objects are linked, their classes are said to have an association

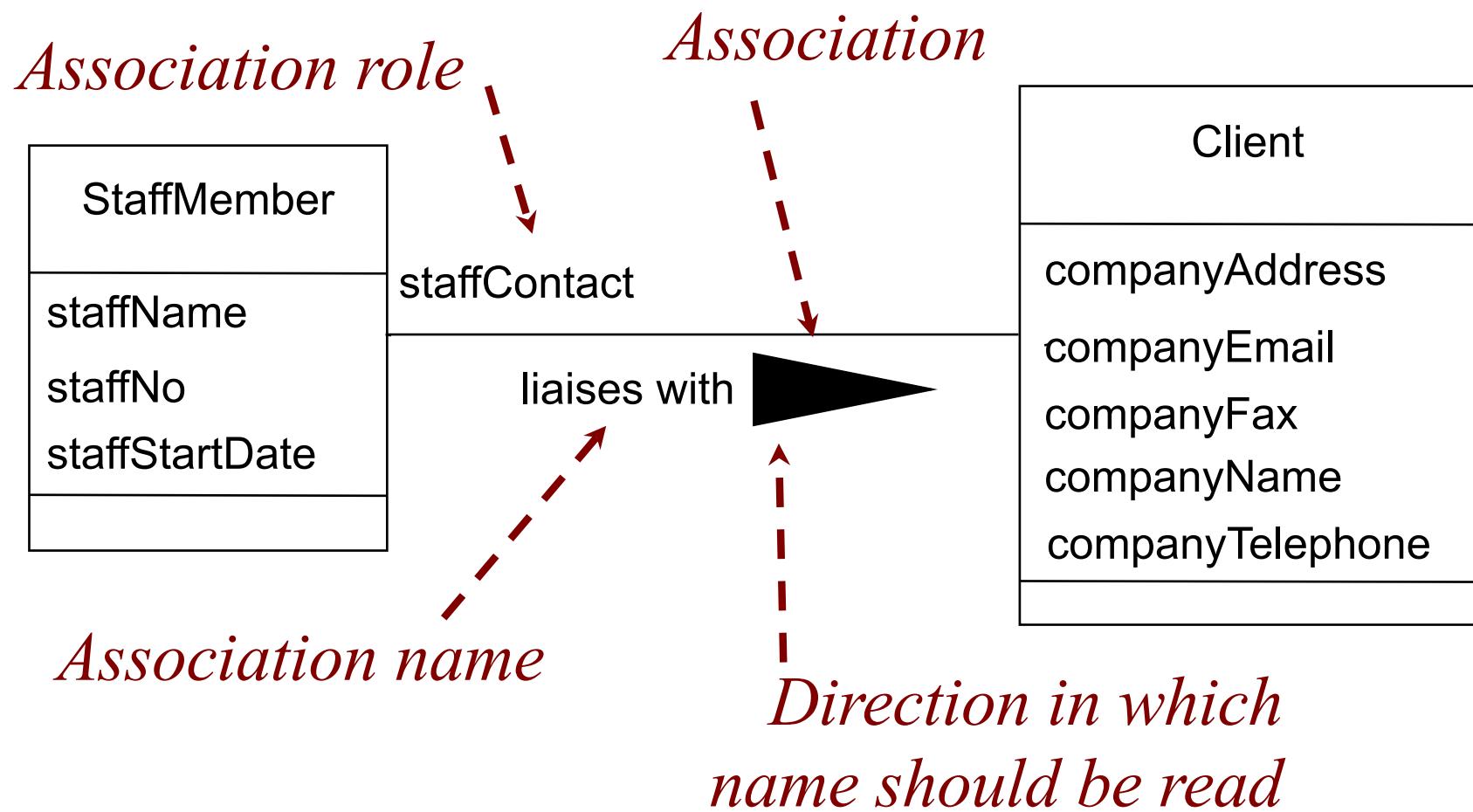


Class Diagram: Links

*A link is a logical connection
between two objects*

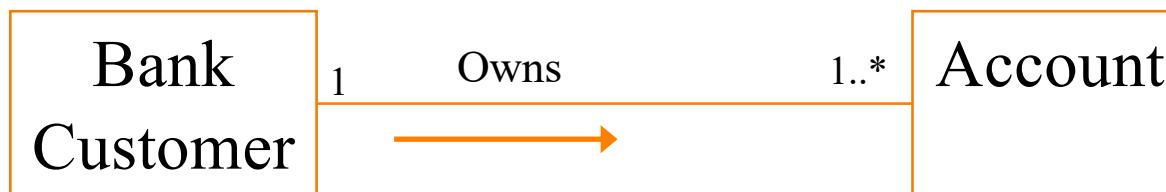


Class Diagram: Associations

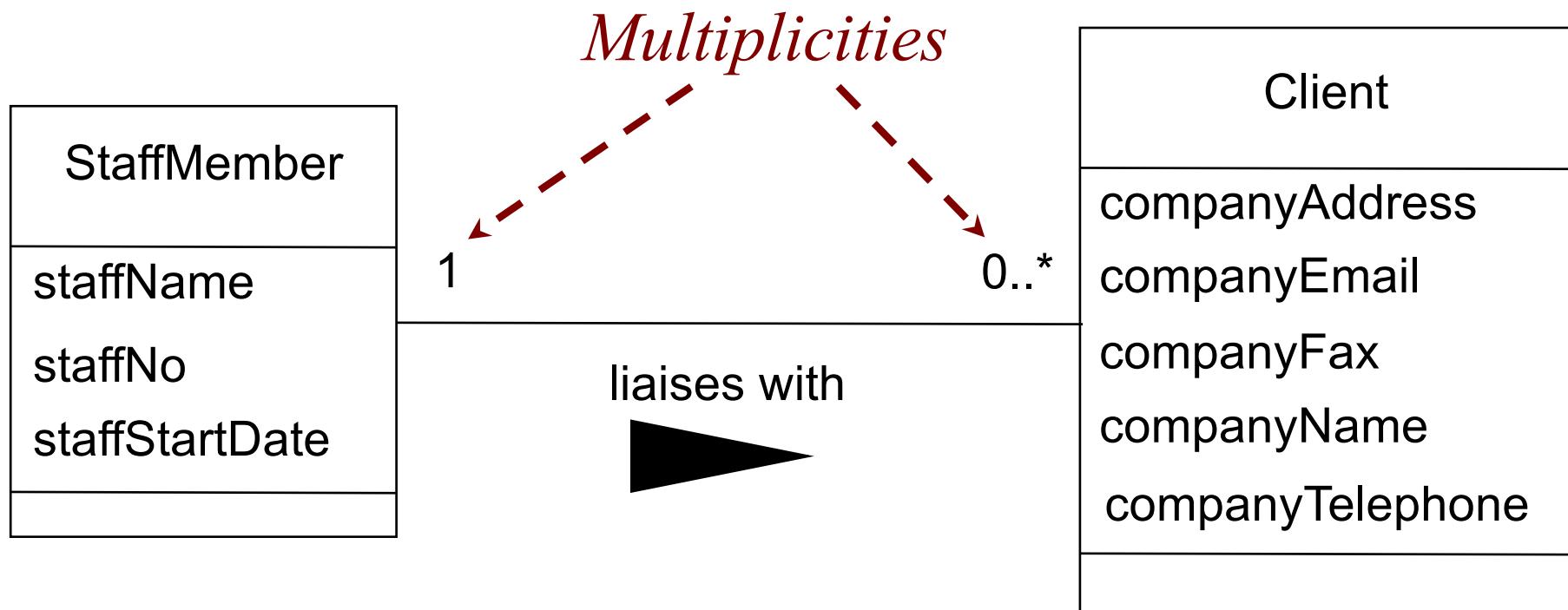


Class Diagram: Multiplicity

- ▶ Associations have multiplicity: the range of permitted cardinalities of an association
- ▶ Represent *enterprise* (or *business*) rules
- ▶ These always come in pairs:
 - ▶ Associations must be read separately from both ends
 - ▶ Each **bank customer** may have 1 or more **accounts**
 - ▶ Every **account** is for 1, and only 1, **customer**



Class Diagram: Multiplicity



- Exactly one staff member liaises with each client
- A staff member may liaise with zero, one or more clients



Relationships in class diagrams

Between CLASSES

- ▶ Named association with cardinality and direction (can be recursive, can have association classes). See previous slides.
 - ▶ Inheritance or 
 - ▶ Whole/Part relationships (discussed next week):
 - ▶ Aggregation or 
 - ▶ Composition or 
- (can be included in a named association)



How to create a Class Diagram

Robustness Analysis

- ▶ Aims to produce a set of classes robust enough to meet the requirements of a use case
- ▶ Makes some assumptions about the interaction:
 - ▶ Assumes some class or classes are needed to handle the user interface
 - ▶ Abstracts logic of the use case away from *entity* classes (that store persistent data)



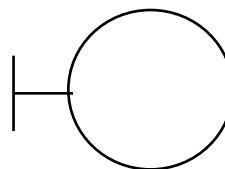
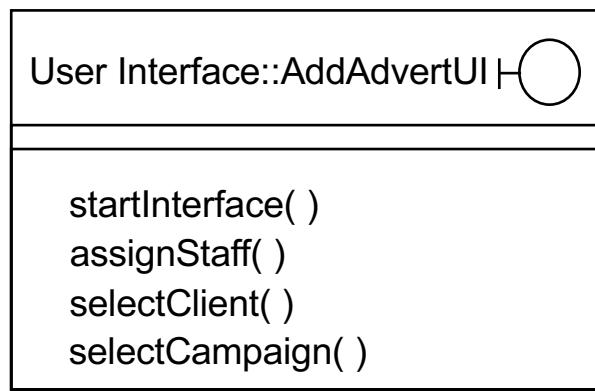
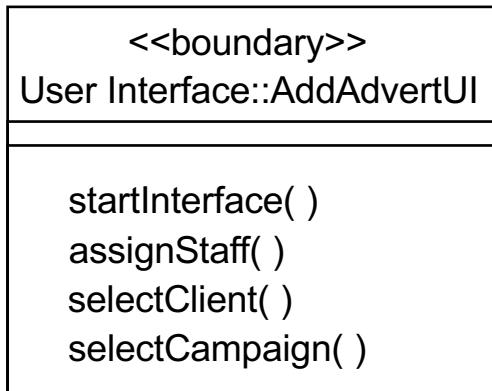
Robustness Analysis: Class Stereotypes

- ▶ Class stereotypes differentiate the roles objects can play:
 - ▶ **Boundary** objects model interaction between the system and actors (and other systems)
 - ▶ **Control** objects co-ordinate and control other objects
 - ▶ **Entity** objects represent information and behaviour in the application domain
 - ▶ Entity classes may be imported from domain model
 - ▶ Boundary and control classes are more likely to be unique to one application



Boundary Class Stereotype

- ▶ Boundary classes represent interaction with the user
 - likely to be unique to the use case but inherited from a library
- ▶ Alternative notations:

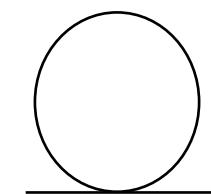
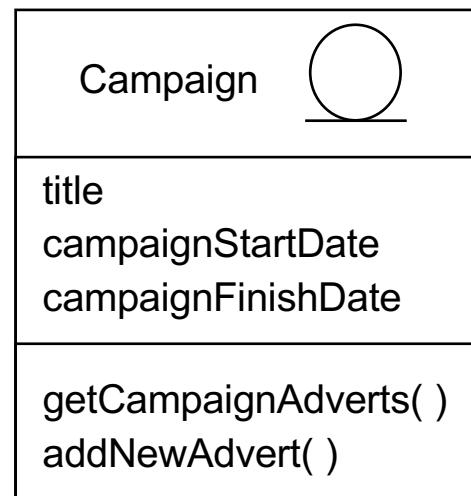
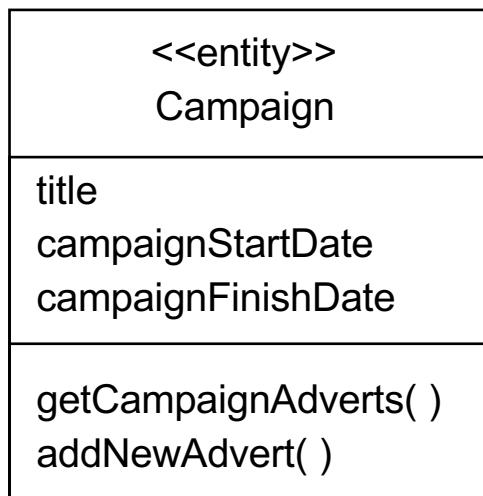


User Interface::AddAdvertUI



Entity Class Stereotype

- ▶ Entity classes represent persistent data and common behaviour likely to be used in more than one application system
- ▶ Alternative notations :

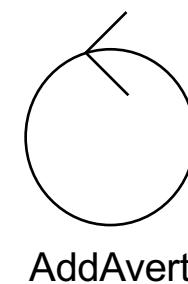
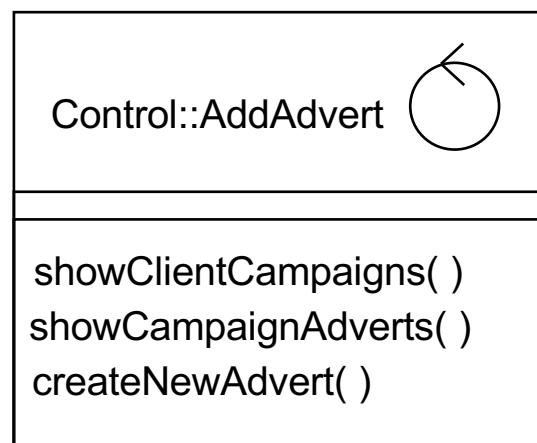
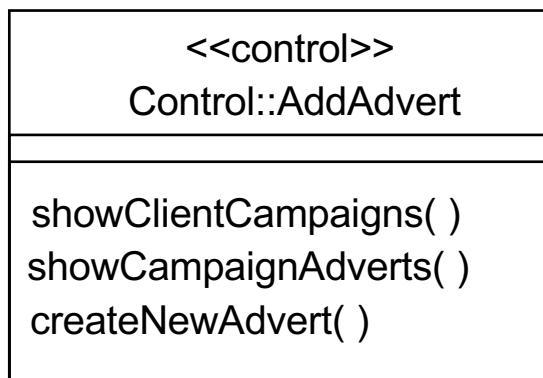


Campaign



Control Class Stereotype

- ▶ Control classes encapsulate unique behaviour of a use case
- ▶ Specific logic kept separate from the common behaviour of entity classes
- ▶ Alternative notations:



From Requirements to Classes

- ▶ Requirements (use cases) are usually expressed in user language
- ▶ Use cases are units of development, but they are not structured like software
- ▶ The software we will implement consists of classes
- ▶ We need a way to translate requirements into classes



Goal of Realization

- ▶ An analysis class diagram is only an interim product
- ▶ This in turn will be realized as a design class diagram
- ▶ The ultimate product of realization is the software implementation of that use case



Assembling the Class Diagram from use cases

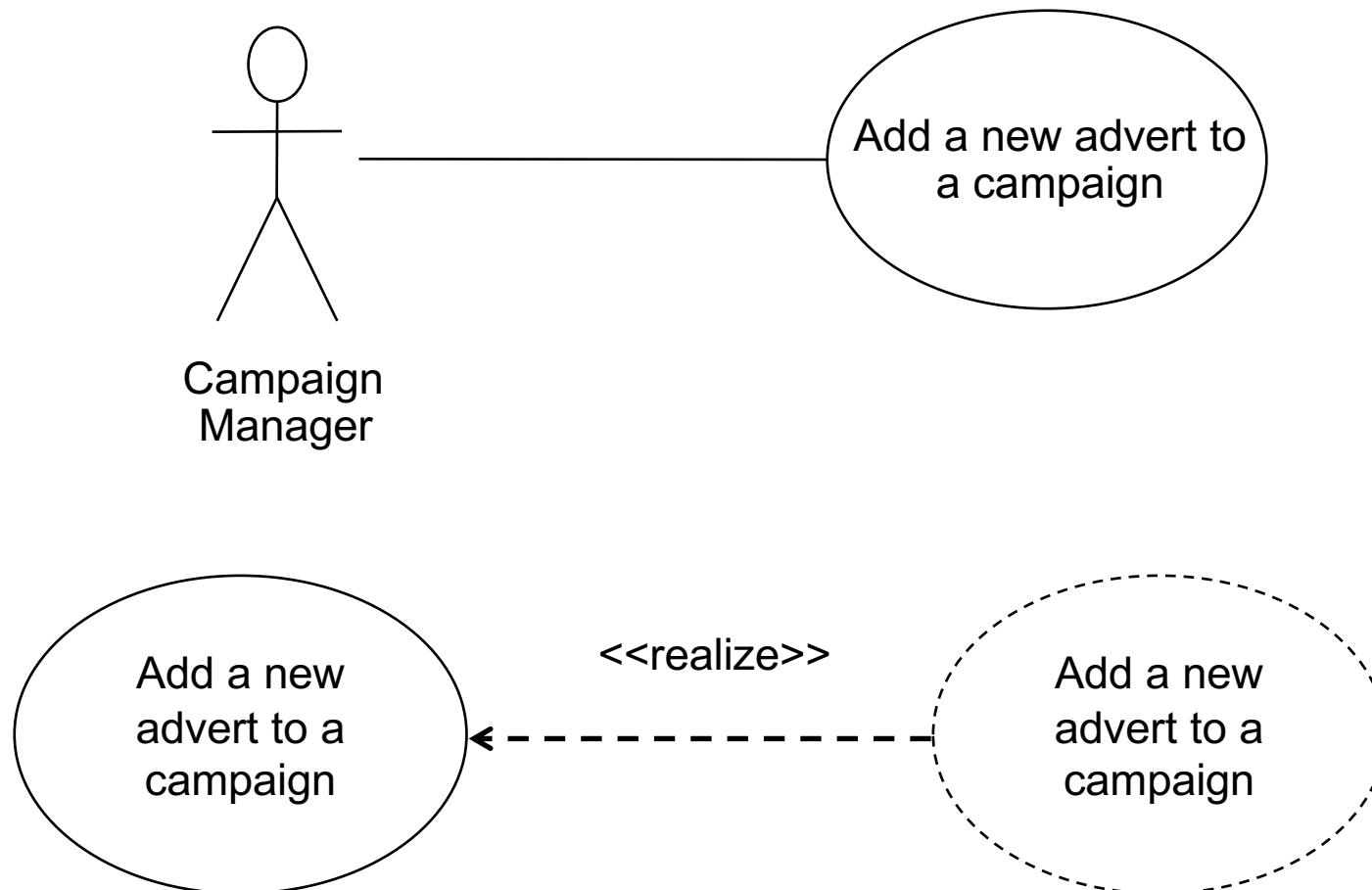
- ▶ However individual use cases are analysed, the aim is to produce a single analysis class diagram
- ▶ This models the application as a whole
- ▶ The concept is simple:
 - ▶ A class in the analysis model needs *all* the details required for that class in each separate use case
 - ▶ You then assemble a final single class diagram from the separate class diagrams (from each use case)



Realization of class diagram (based on knowledge of the application domain) - process

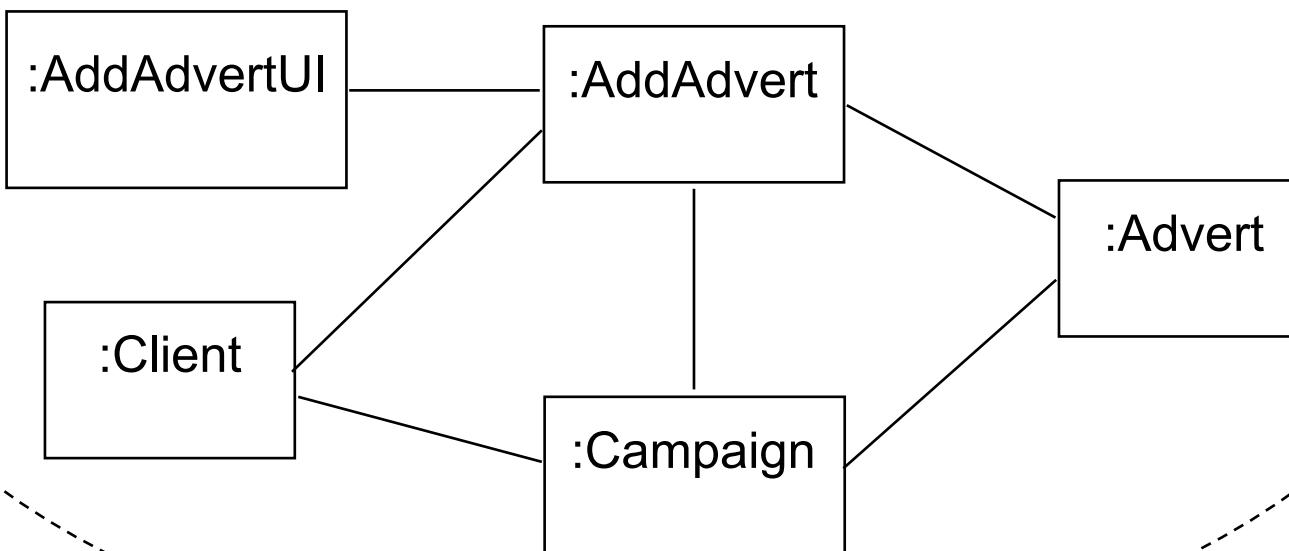
- ▶ 1. Define Entity objects
- ▶ 2. Then add in Control objects
- ▶ 3. Then add in Boundary objects

Example: assembling a class diagram from a Use Case

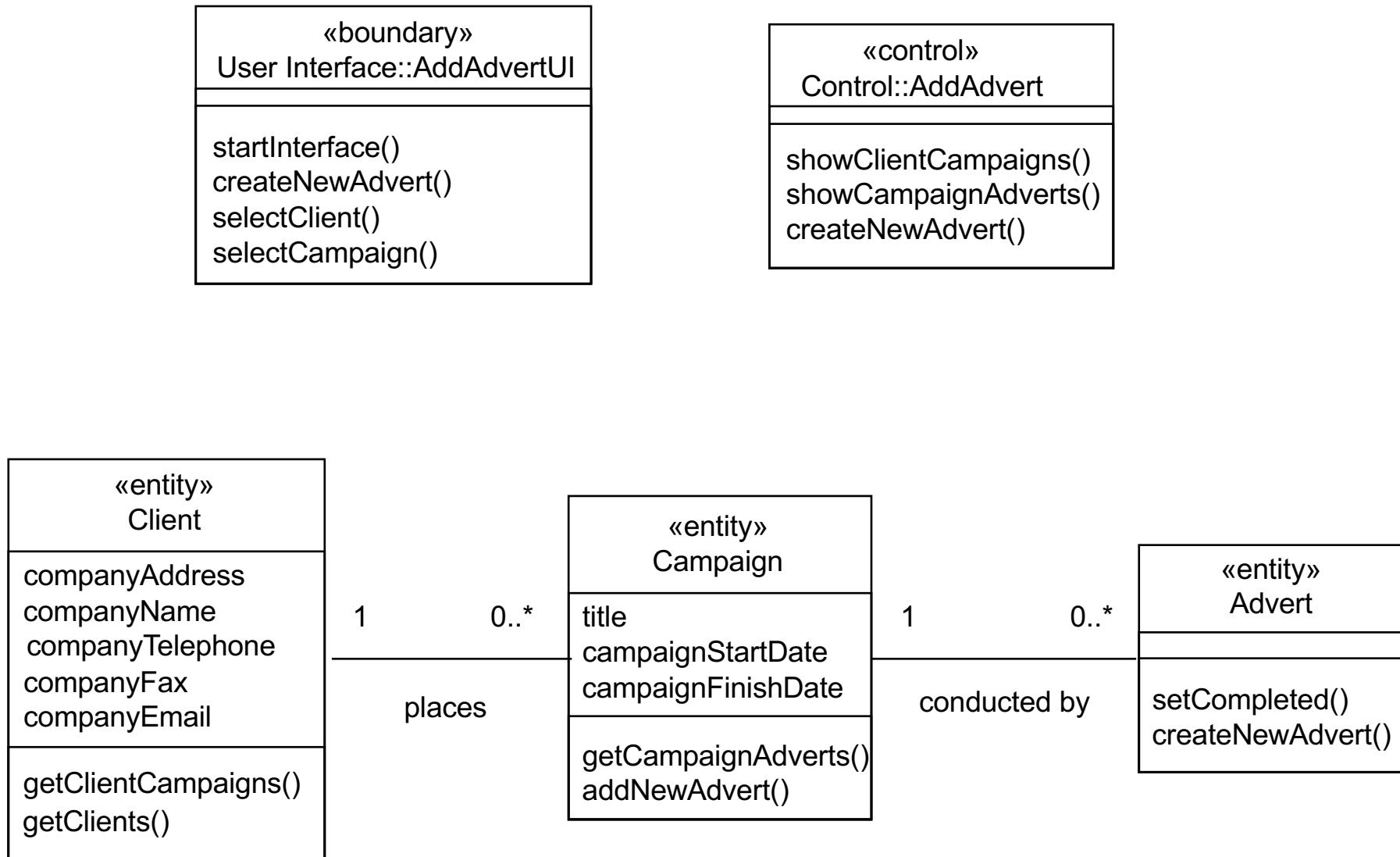


A Possible Collaboration (diagram)

Add a new advert to a campaign



Resulting (incomplete) Class Diagram



Reasonability Checks for Candidate Classes

- ▶ A number of tests help to check whether a candidate class is reasonable
 - ▶ Is it beyond the scope of the system?
 - ▶ Does it refer to the system as a whole?
 - ▶ Does it duplicate another class?
 - ▶ Is it too vague?

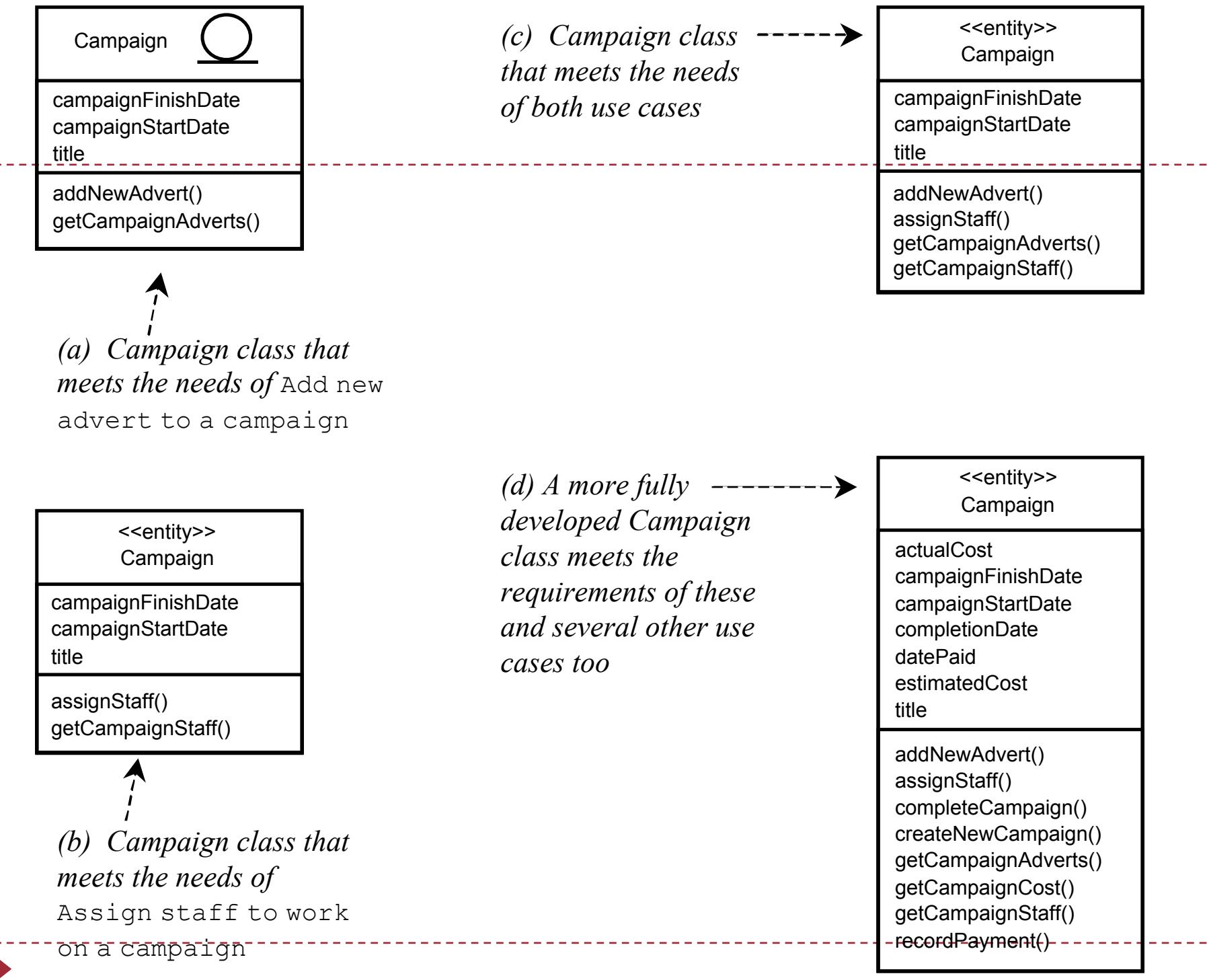
(More on next slide)



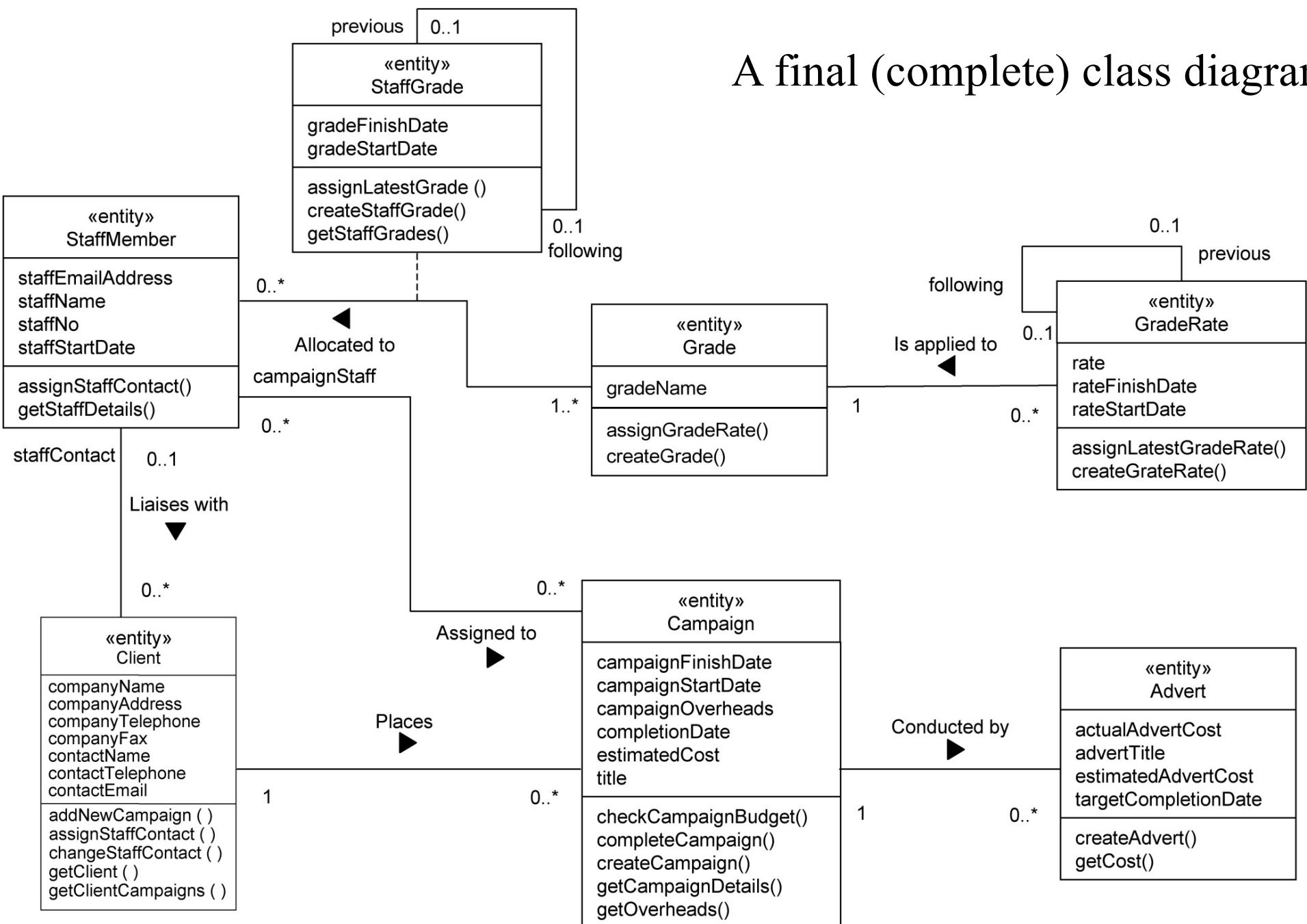
Reasonability Checks for Candidate Classes (cont' d)

- ▶ Is it too tied up with physical inputs and outputs?
- ▶ Is it really an attribute?
- ▶ Is it really an operation?
- ▶ Is it really an association?
- ▶ If any answer is ‘Yes’ , consider modelling the potential class in some other way (or do not model it at all)
- ▶ Only show permanent/static relationships on the class diagram. Leave out temporary relationships associated with particular instances





A final (complete) class diagram



Communication Diagram Approach

- ▶ **Analyse one use case at a time**
- ▶ **Identify likely classes involved (the use case collaboration)**
 - ▶ These may come from a domain model
- ▶ **Draw a communication diagram that fulfils the needs of the use case (see next lecture)**
- ▶ Translate this into a use case class diagram
- ▶ Repeat for other use cases
- ▶ Assemble the use case class diagrams into a single analysis class diagram
- ▶ (see next lecture)



Summary

In this lecture you have learned:

- ▶ What is meant by *use case realization*
- ▶ How to realize use cases with robustness analysis and communication diagrams
- ▶ How to assemble the analysis class diagram
- ▶ Started looking at the syntax of class diagrams



Exercises

1. Draw a class diagram representing a book defined by the following statement: "A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections." Focus only on classes and relationships.
 2. Add multiplicity to the class diagram you produced.
 3. Extend the class diagram to include the following attributes:
 - ▶ a book includes a publisher, publication date, and an ISBN
 - ▶ a part includes a title and a number
 - ▶ a chapter includes a title, a number, and an abstract
 - ▶ a section includes a title and a number
 4. Note that the Part, Chapter, and Section classes all include a title and a number attribute. Add an abstract class and a generalization relationship to factor out these two attributes into the abstract class.
-

Further reading

- ▶ Parnas (1985) - SOFTWARE ASPECTS OF STRATEGIC DEFENSE SYSTEMS, 1985 ACM OflOI-0782/85/1200-1326 750
- ▶ Designing Object-Oriented Software, Rebecca J Wirfs-Brock, Brian Wilkerson, and Lauren Wiener, Prentice Hall, 1990
- ▶ See Chapter 7 Bennett



Interaction diagrams

- Sequence diagrams
- Collaboration diagrams

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

Reminder (last lecture): from requirements to classes

- ▶ Requirements (use cases) are usually expressed in user language
- ▶ Use cases are units of development, but they are not structured like software
- ▶ The software we will implement consists of classes
- ▶ We need a way to translate requirements into classes



Reminder (last lecture): communication Diagram Approach

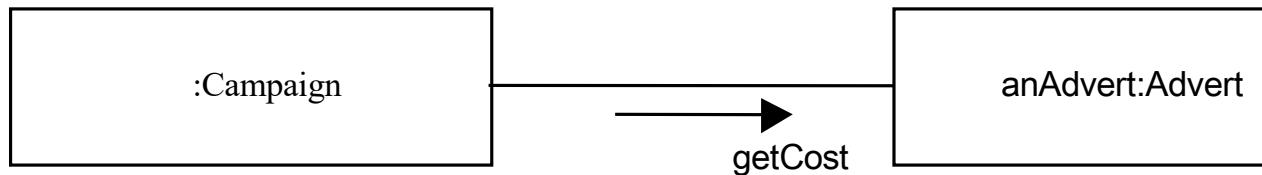
- ▶ Analyse one use case at a time
- ▶ Identify likely classes involved (the use case collaboration)
 - ▶ These may come from a domain model
- ▶ **Draw a communication diagram** that fulfils the needs of the use case (see next lecture)
- ▶ Translate this into a use case class diagram
- ▶ Repeat for other use cases
- ▶ Assemble the use case class diagrams into a single analysis class diagram



Object Messaging

Objects communicate by sending messages.
Sending the message `getCost()` to an `Advert` object, might use the following syntax.

```
currentadvertCost = anAdvert.getCost()
```



Interaction & Collaboration

- ▶ A **collaboration** is a group of objects or classes that work together to provide an element of functionality or behaviour.
- ▶ An **interaction** defines the message passing between lifelines (e.g. objects) within the context of a collaboration to achieve a particular behaviour.



Modelling Interactions

- ▶ Interactions can be modelled using various notations
 - ▶ **Sequence** diagrams
 - ▶ **Collaboration** diagrams
 - ▶ Interaction overview diagrams – not covered
 - ▶ Timing diagrams – not covered



Sequence Diagrams

Sequence Diagrams

- ▶ Shows an interaction between lifelines (e.g. objects) arranged in a time sequence.
- ▶ Can be drawn at different levels of detail and to meet different purposes at several stages in the development life cycle.
- ▶ Typically used to represent the detailed object interaction that occurs **for one use case** or for one operation.

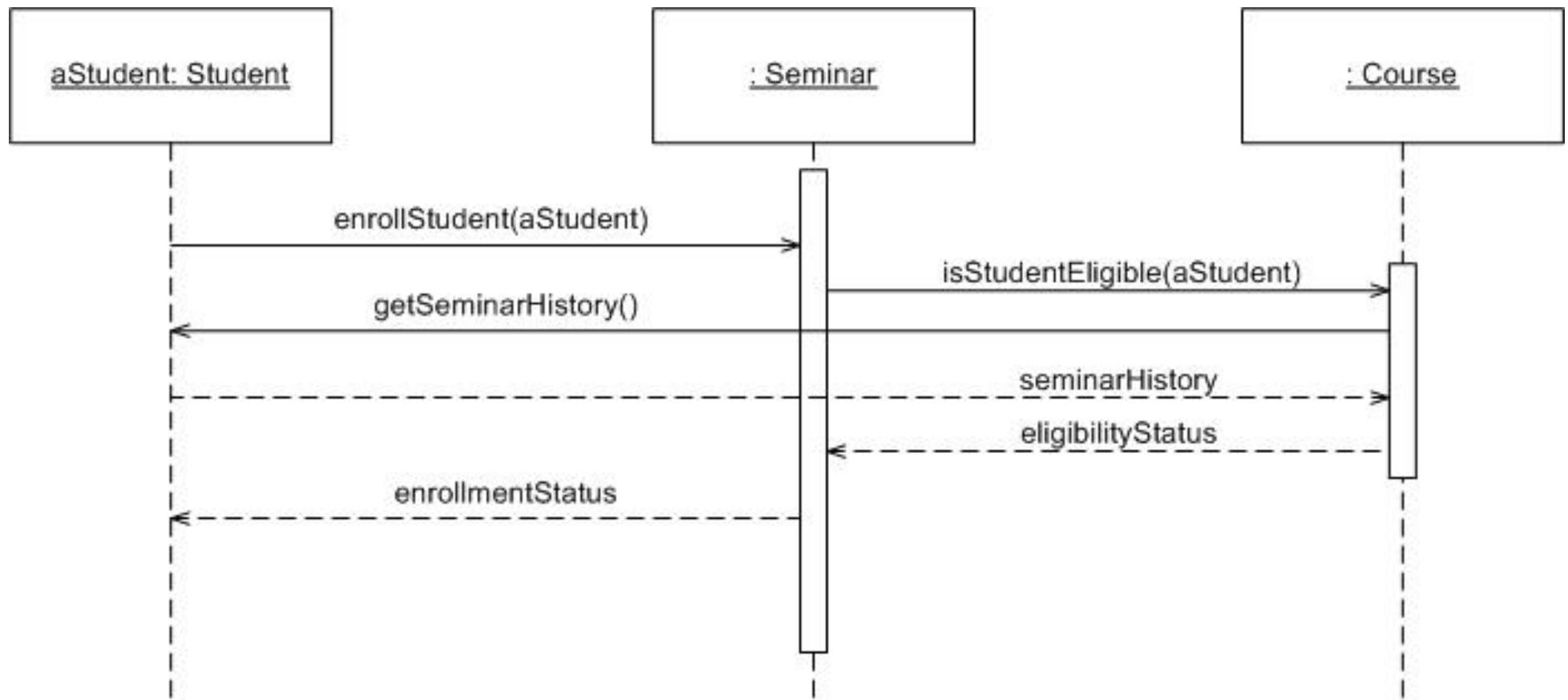


Sequence Diagrams

- ▶ Vertical dimension shows time.
- ▶ Objects (or subsystems or other connectable objects) involved in interaction appear horizontally across the page and are represented by lifelines.
- ▶ Messages are shown by a solid horizontal arrow.
- ▶ The execution or activation of an operation is shown by a rectangle on the relevant lifeline.



Example sequence diagram



Guidelines for Sequence Diagrams

1. Decide at what level you are modelling the interaction.
Is it describing an operation, a use case, the messaging between components or the interaction of subsystems or systems?
2. Identify the main elements involved in the interaction.
If the interaction is at use case level the collaborating objects may already have been identified through the use of CRC cards (discussed later)
3. Consider the alternative scenarios that may be needed.
4. Identify the main elements involved in the interaction.



Guidelines for Sequence Diagrams

5. Draw the outline structure of the diagram.
6. Add the detailed interaction.
7. Check for consistency with linked sequence diagrams and modify as necessary.
8. Check for consistency with other UML diagrams or models.



Model Consistency (with the class diagram)

- ▶ The allocation of operations to objects must be consistent with the class diagram and the message signature must match that of the operation.
 - ▶ Can be enforced through CASE tools.
- ▶ Every sending object must have the object reference (**attribute**) for the destination object.
 - ▶ Either an association exists between the classes or another object passes the reference to the sender.
 - ▶ This issue is key in determining association message pathways and should be carefully analysed.
- ▶ Every receiving object must have the message listed as an **operation** in the class diagram

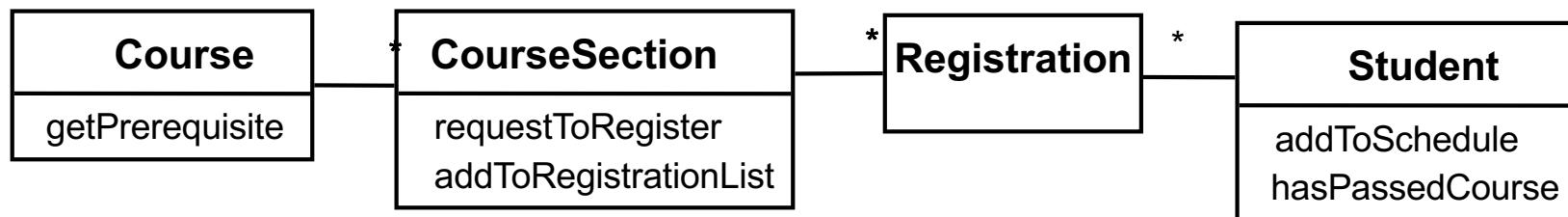
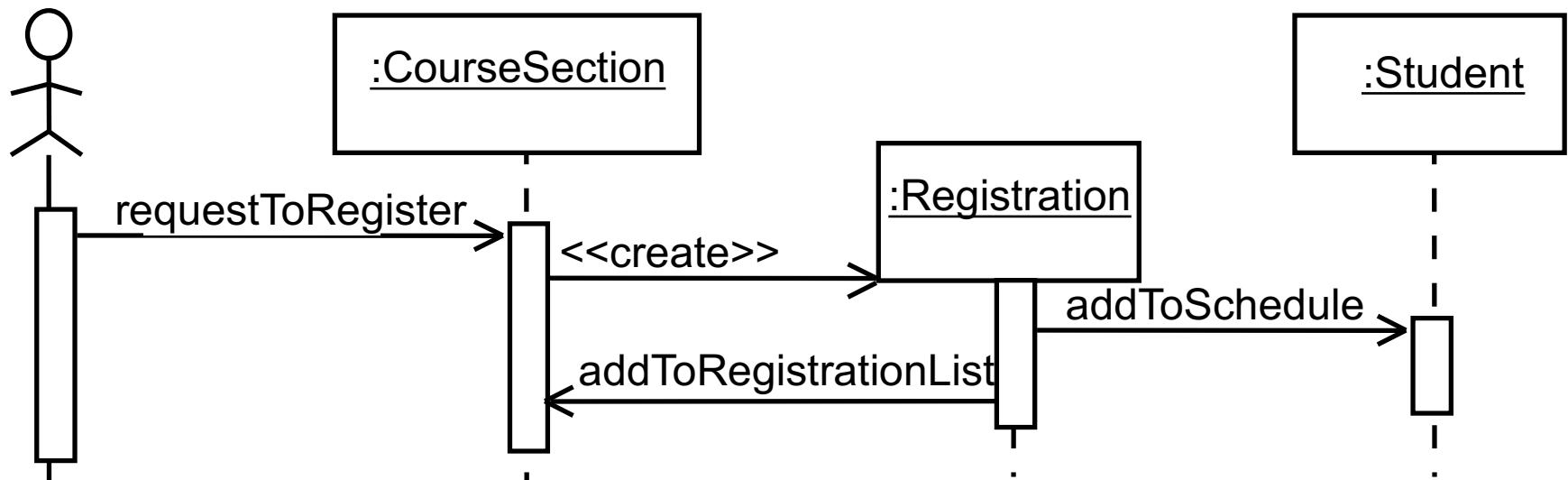


Model Consistency

- ▶ All forms of interaction diagrams used should be consistent.
- ▶ Messages on interaction diagrams must be consistent with the state machine for the participating objects. (see lecture next week)
- ▶ Implicit state changes in interactions diagrams must be consistent with those explicitly modelled in the state machine. (see lecture next week)



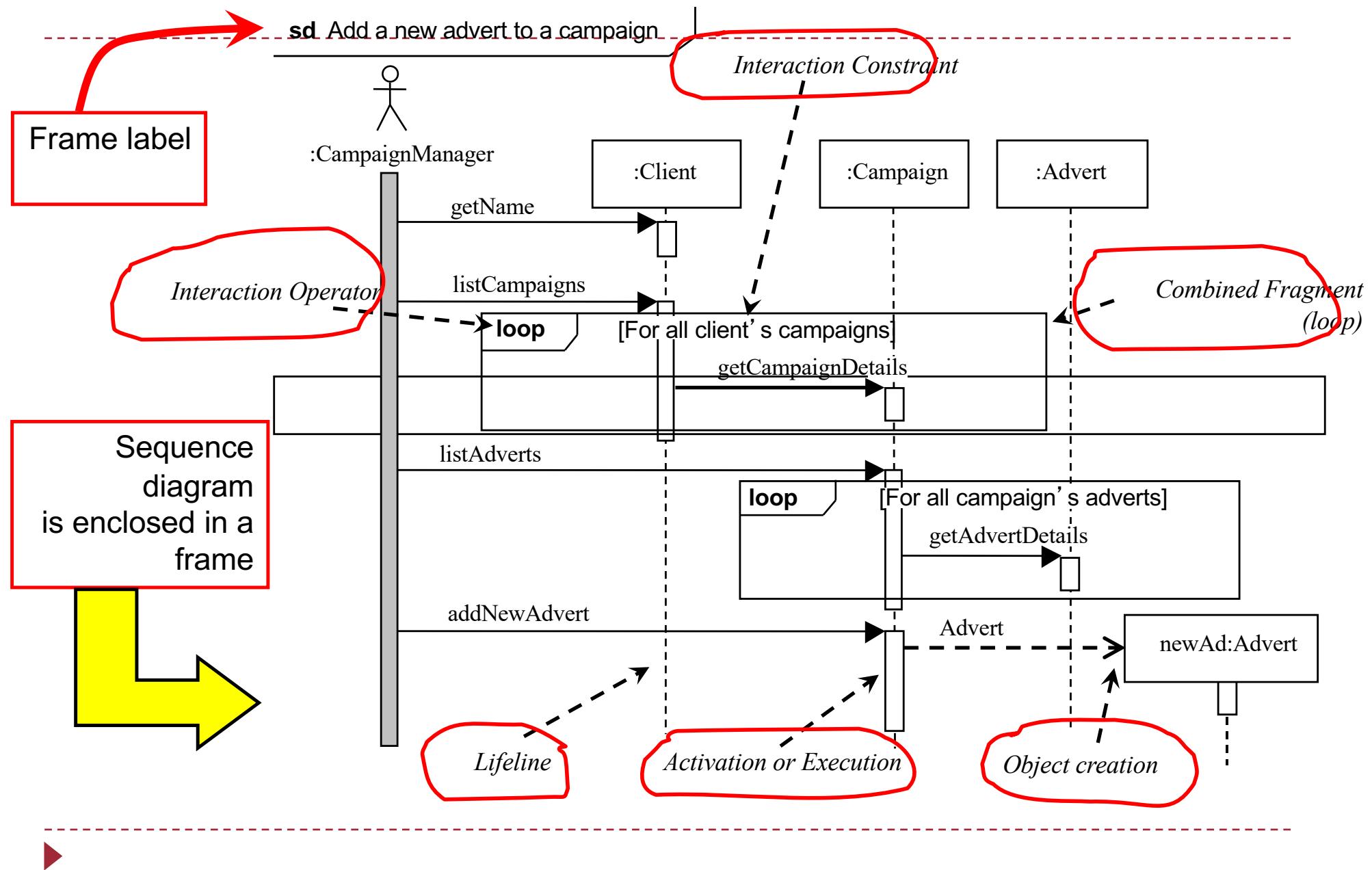
Sequence vs. class diagrams (consistency)



Sequence diagrams

- ▶ In the main we will be involved with fairly **simple** sequence diagrams which model the message passing between objects in a use case (see previous example)
- ▶ The following slides provide additional detail on the **full** use of sequence diagrams

Sequence diagram (full syntax)



Sequence Diagram

- ▶ Iteration is represented by *combined fragment* rectangle with the *interaction operator* ‘loop’.
- ▶ The loop combined fragment only executes if the guard condition in the interaction constraint evaluates as true.
- ▶ Object creation is shown with the construction arrow (dashed) going to the object symbol for the Advert lifeline.

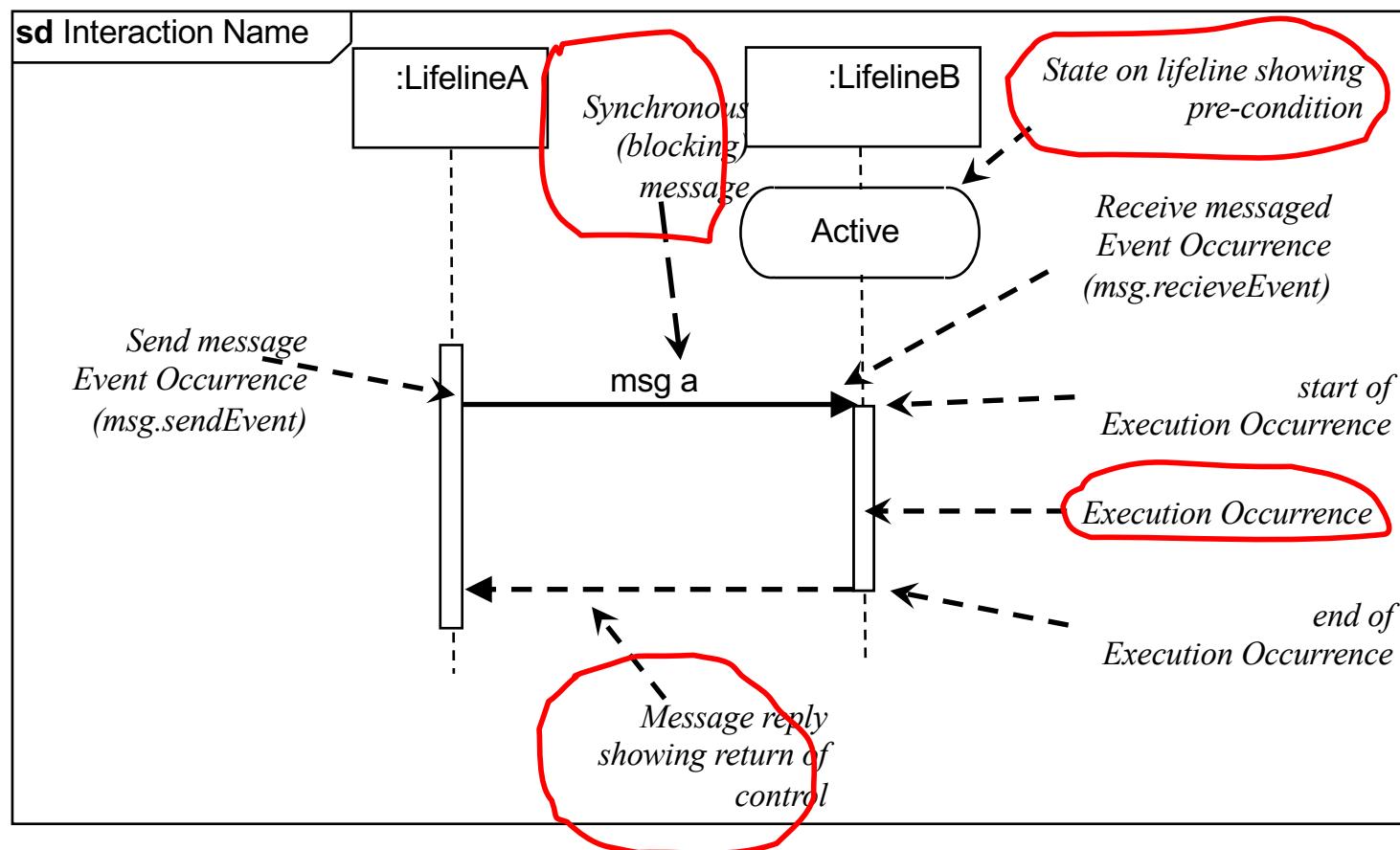


Synchronous Message

- ▶ A *synchronous message* or *procedural call* is shown with a full arrowhead, causes the invoking operation to suspend execution until the focus of control has been returned to it.



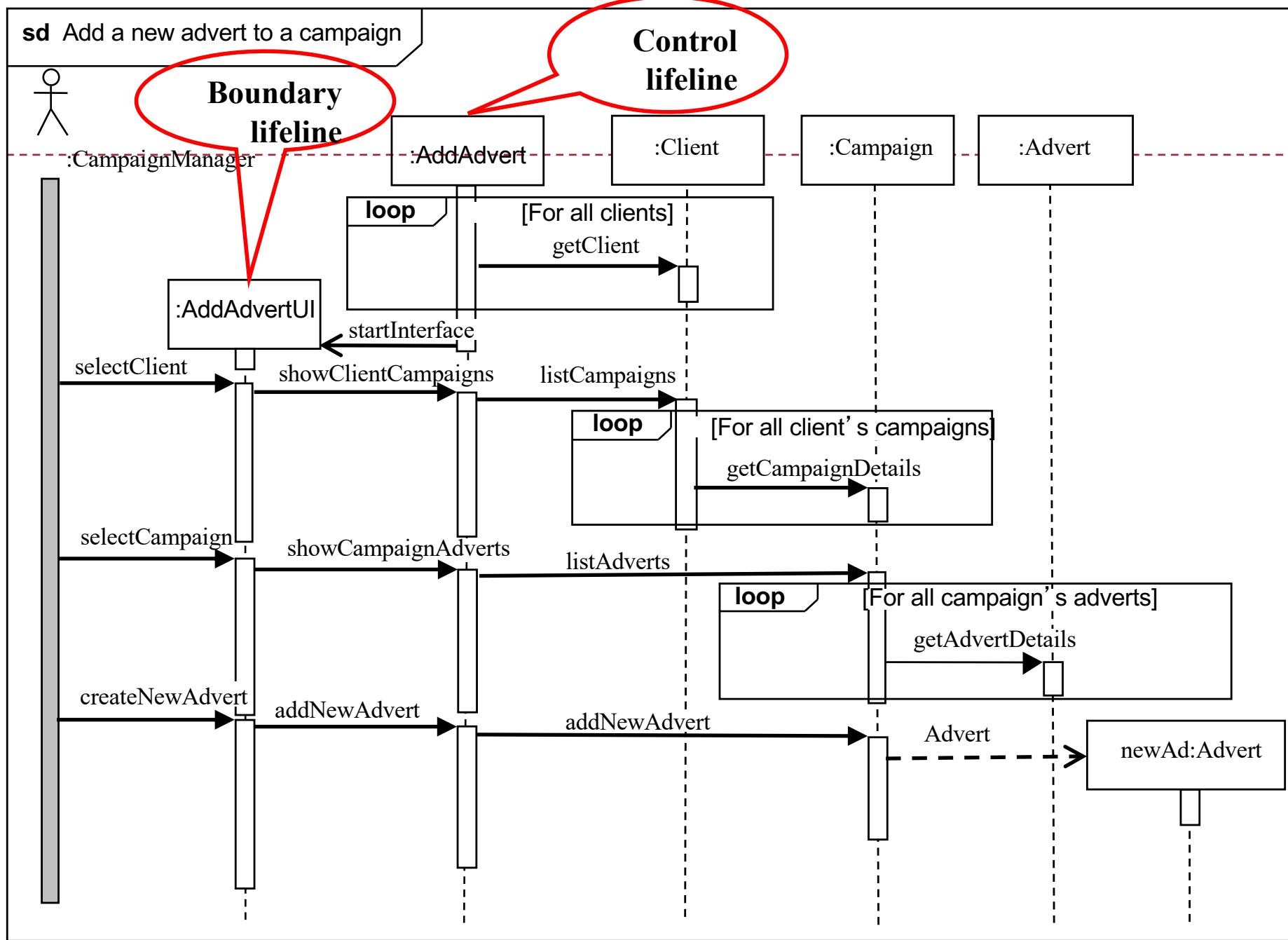
Further Notation



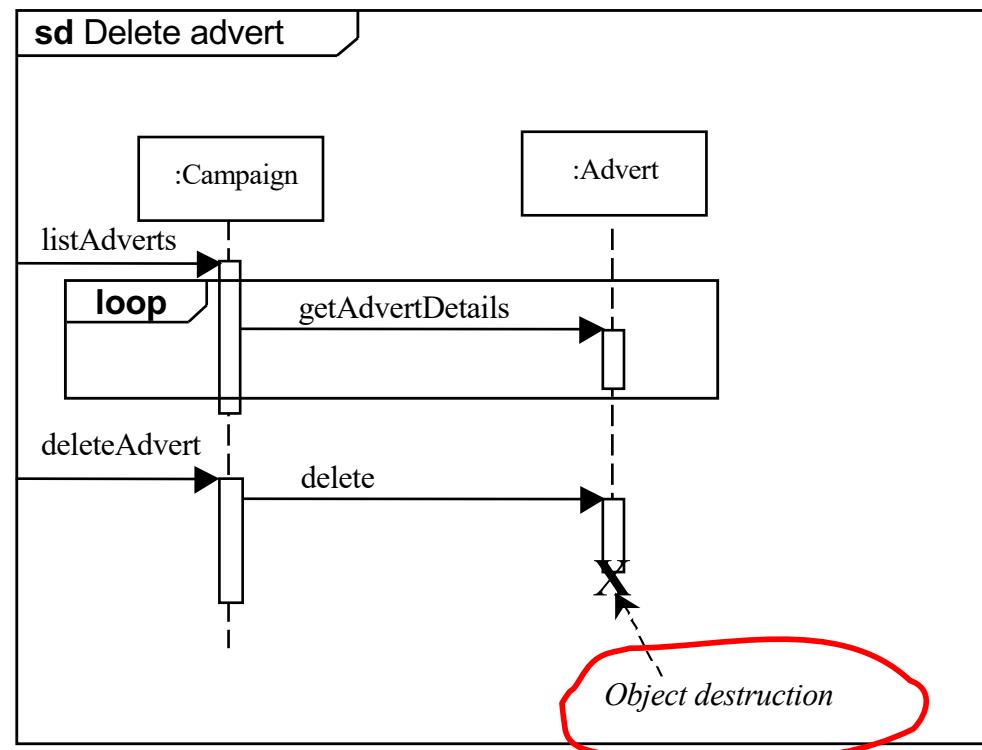
Boundary & Control Classes

- ▶ Most use cases imply at least one boundary object that manages the dialogue between the actor and the system – in the next sequence diagram it is represented by the lifeline :AddAdvertUI
- ▶ The control object is represented by the lifeline :AddAdvert and this manages the overall object communication.

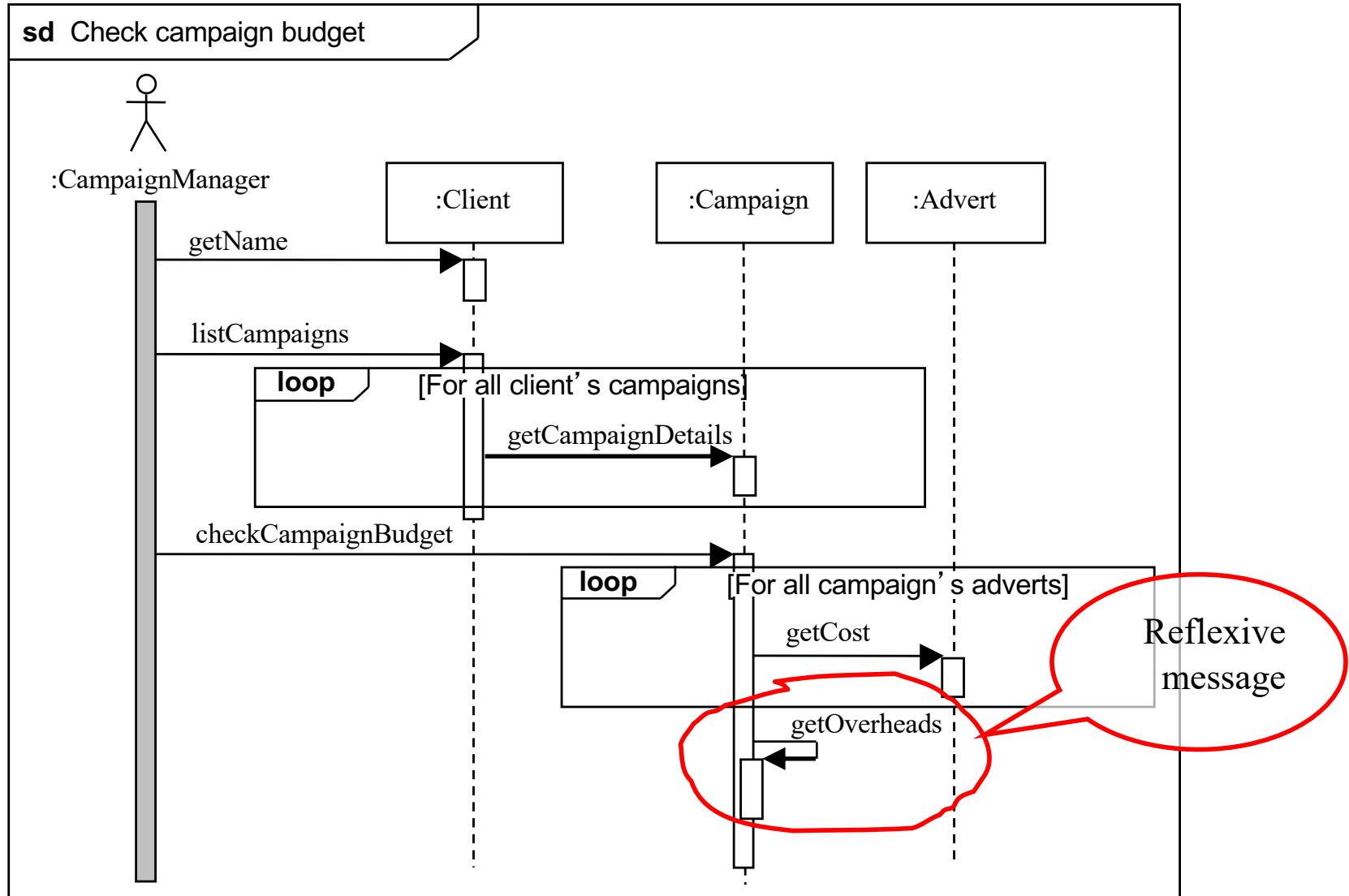




Object Destruction



Reflexive Messages

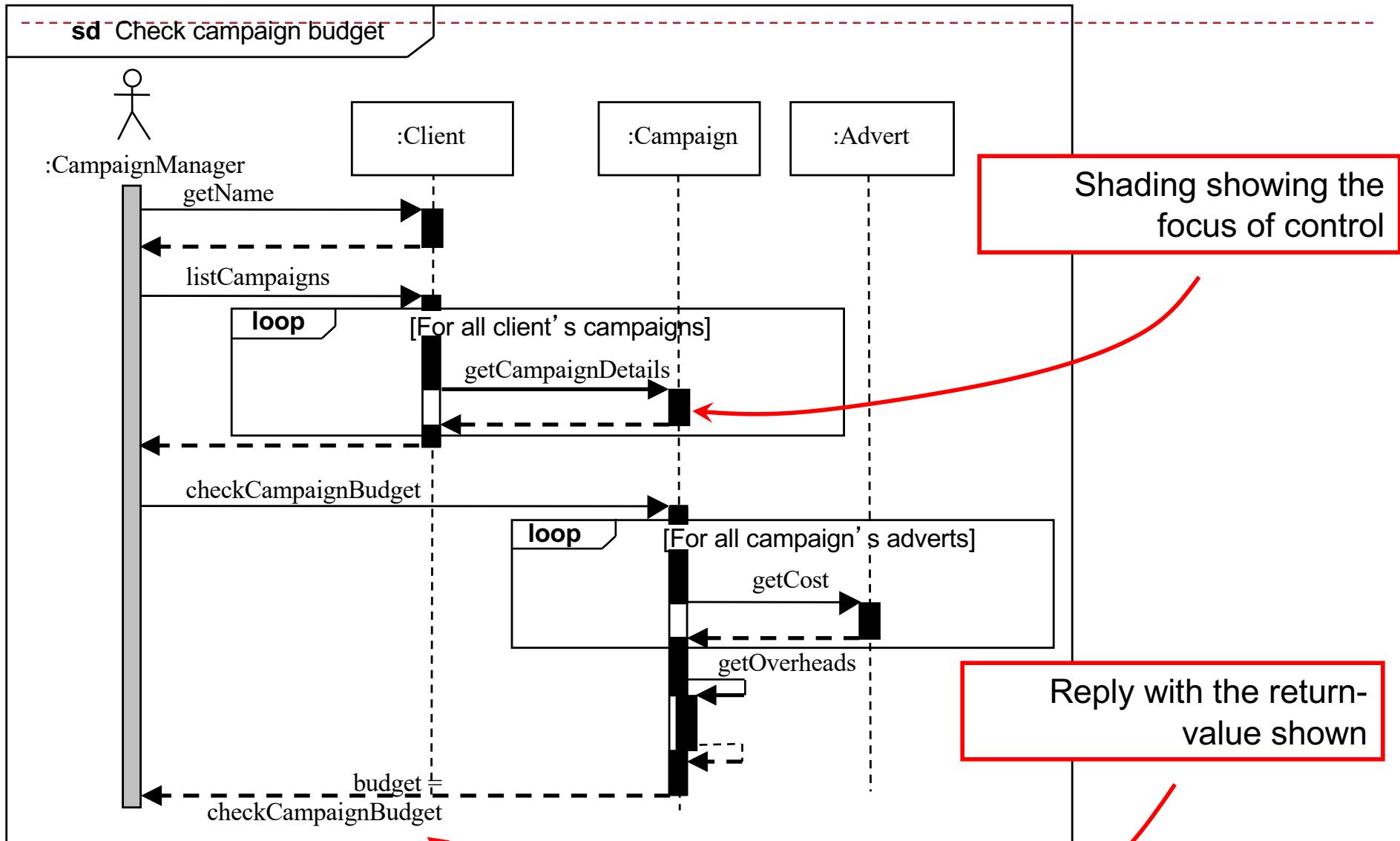


Focus of Control

- ▶ Indicates times during an activation when processing is taking place within that object.
- ▶ Parts of an activation that are not within the focus of control represent periods when, for example, an operation is waiting for a return from another object.
- ▶ May be shown by shading those parts of the activation rectangle that correspond to active processing by an operation.



Focus of Control

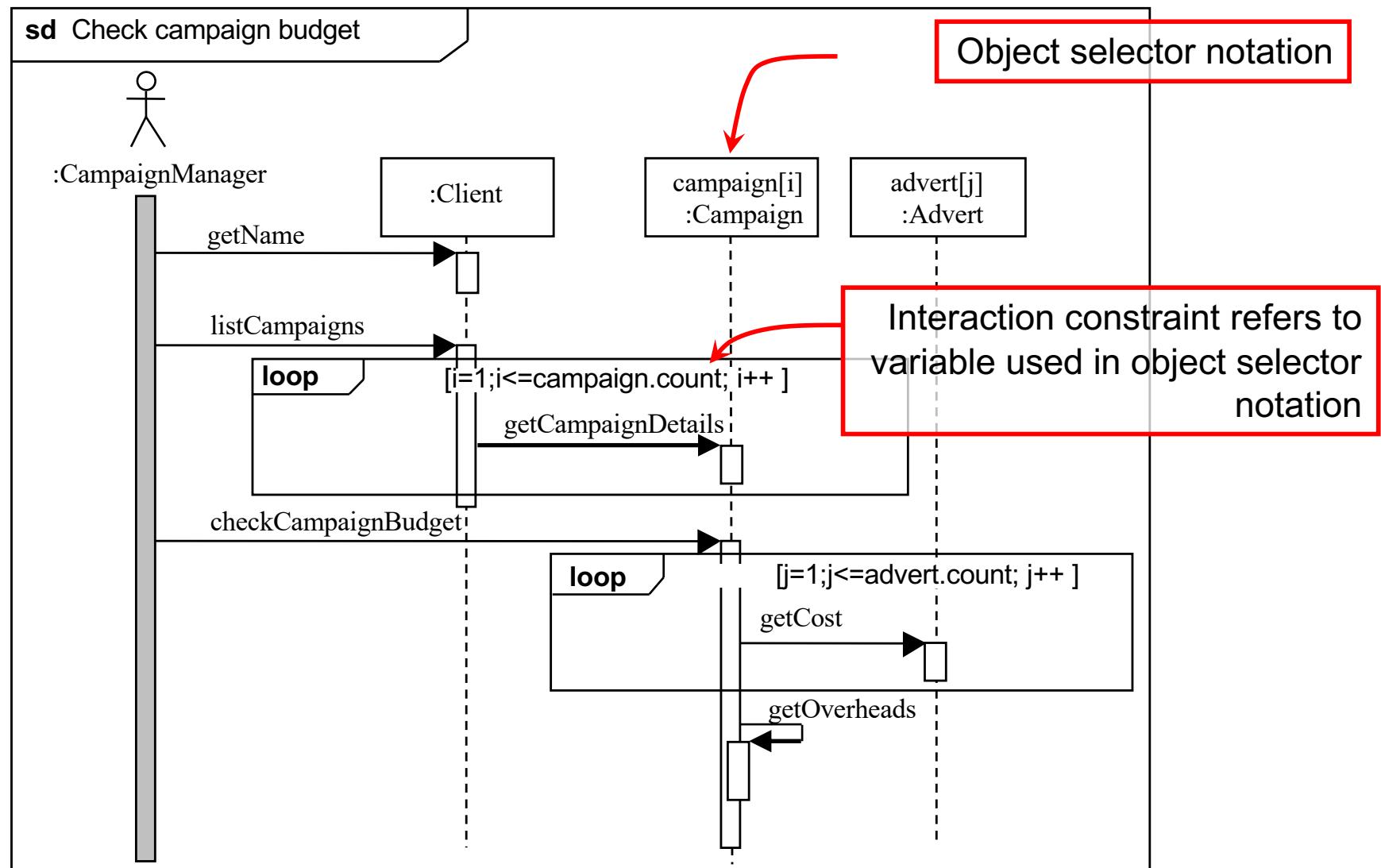


Reply Message

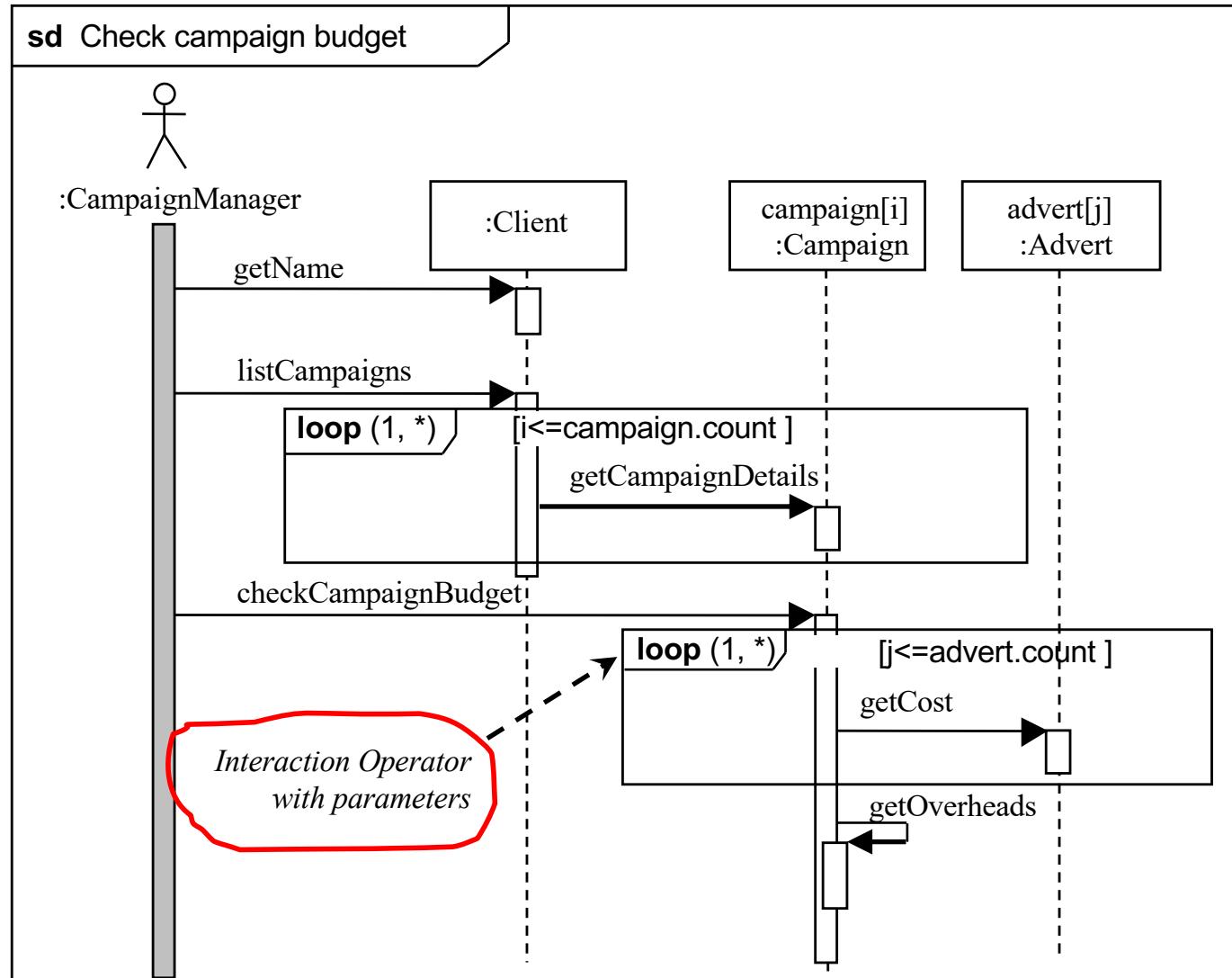
- ▶ A *reply message* returns the control to the object that originated the message that began the activation.
- ▶ Reply messages are shown with a dashed arrow, but it is optional to show them at all since it can be assumed that control is returned to the originating object at the end of the sequence



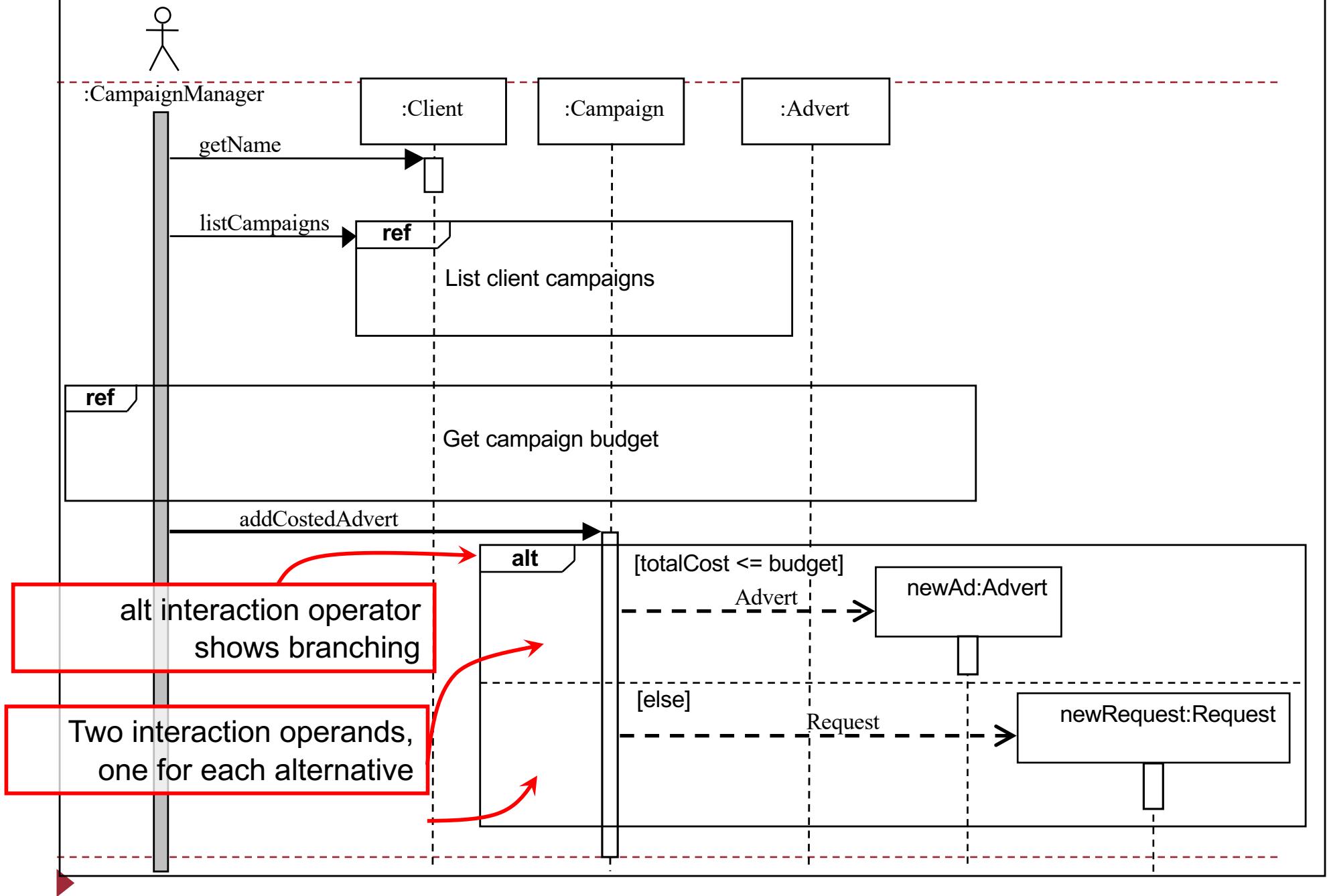
Object Selector Notation



Interaction Operators



sd Add a new advert to a campaign if within budget

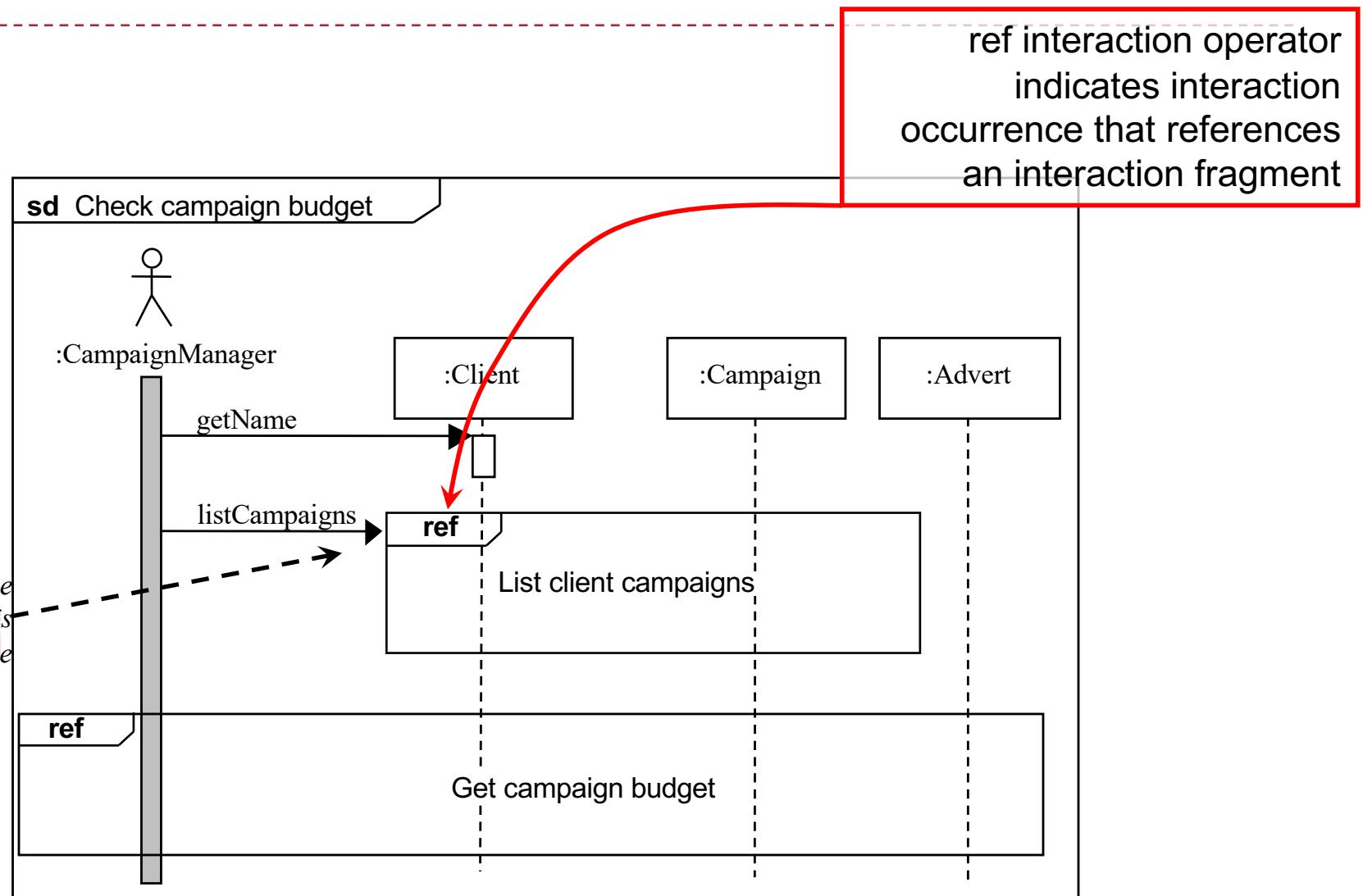


Handling Complexity

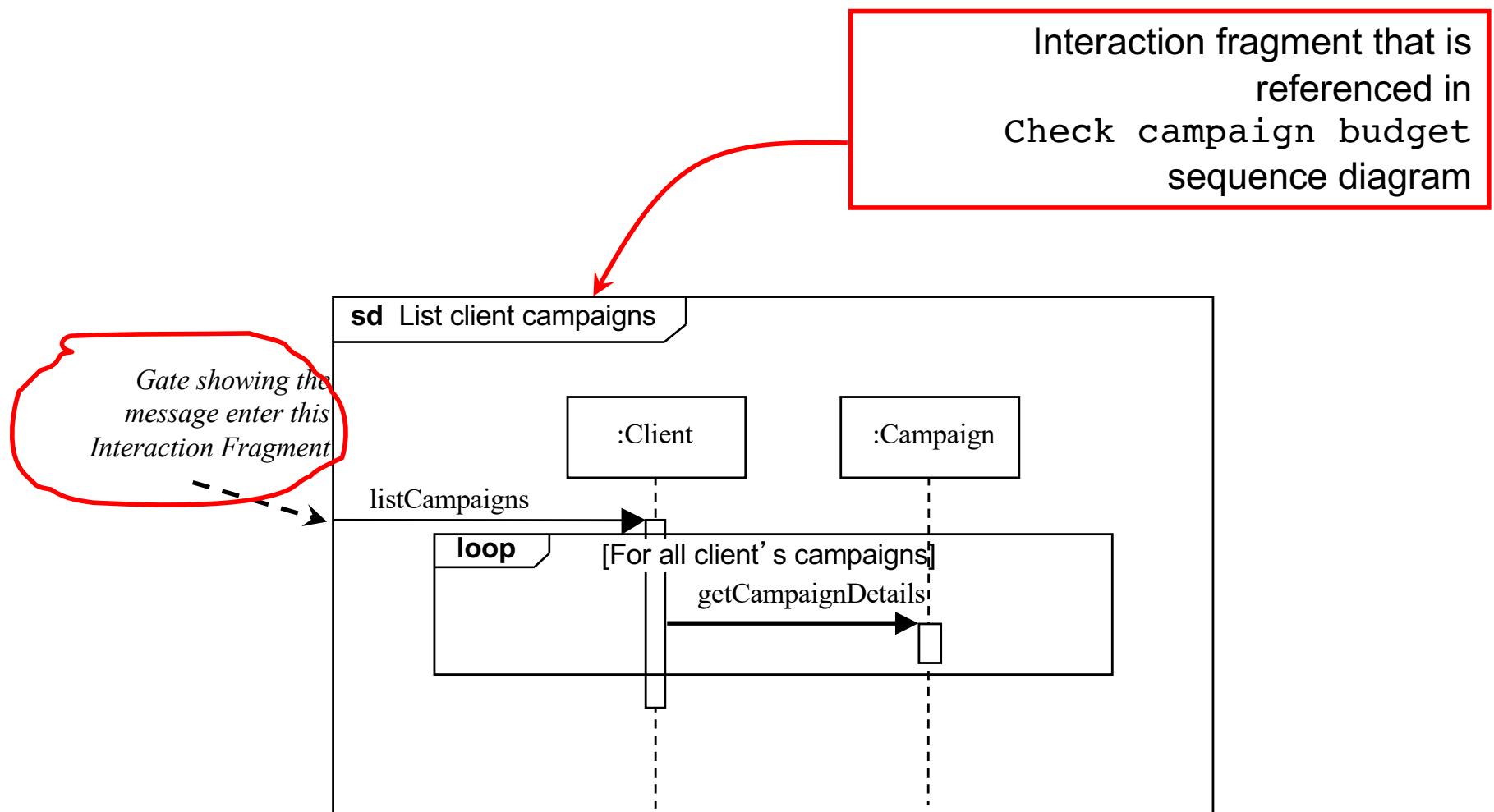
- ▶ Complex interactions can be modelled using various different techniques
 - ▶ Interaction fragments
 - ▶ Lifelines for subsystems or groups of objects
 - ▶ Continuations



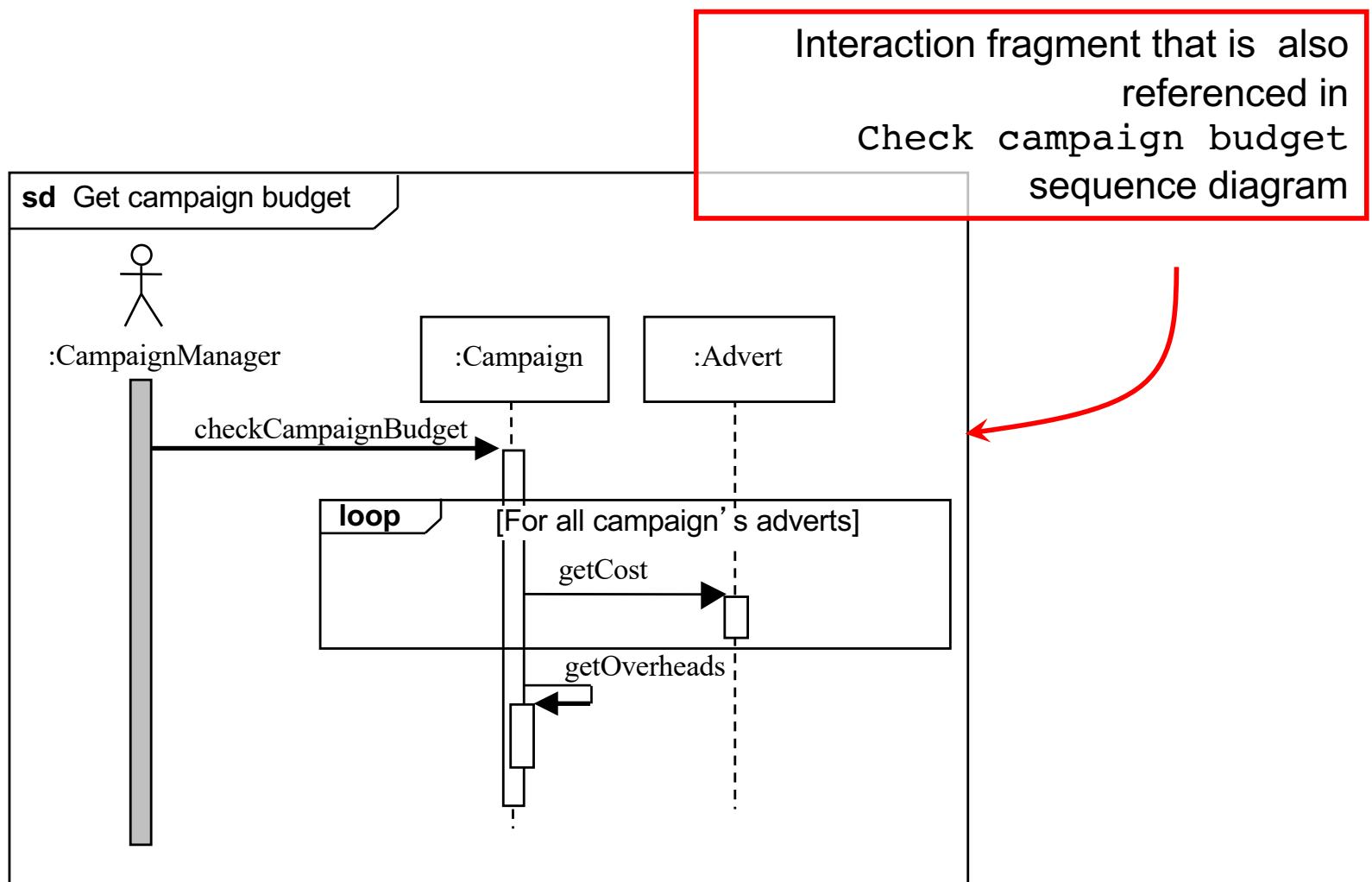
Using Interaction Fragments



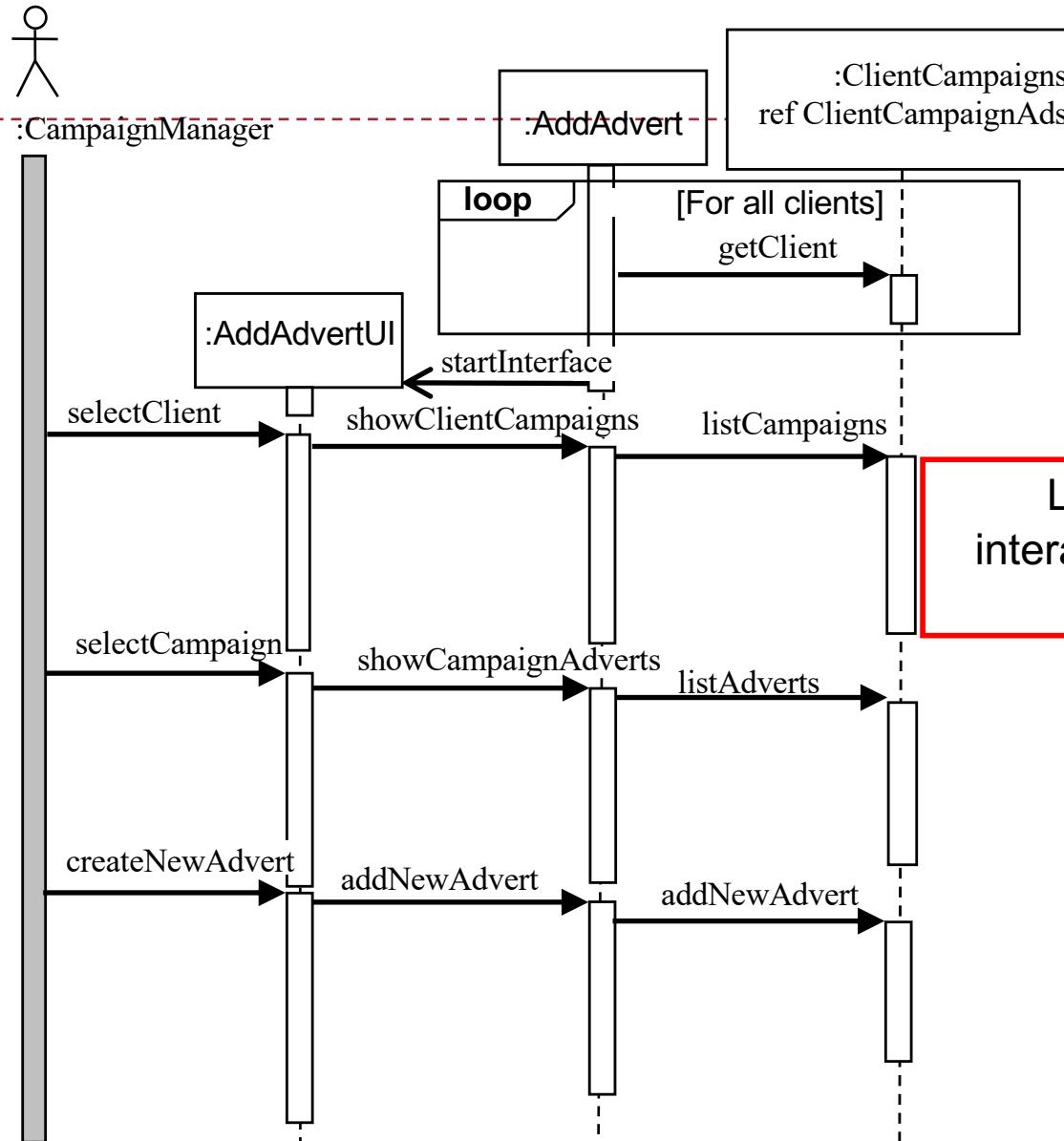
Interaction Fragment



Interaction Fragment



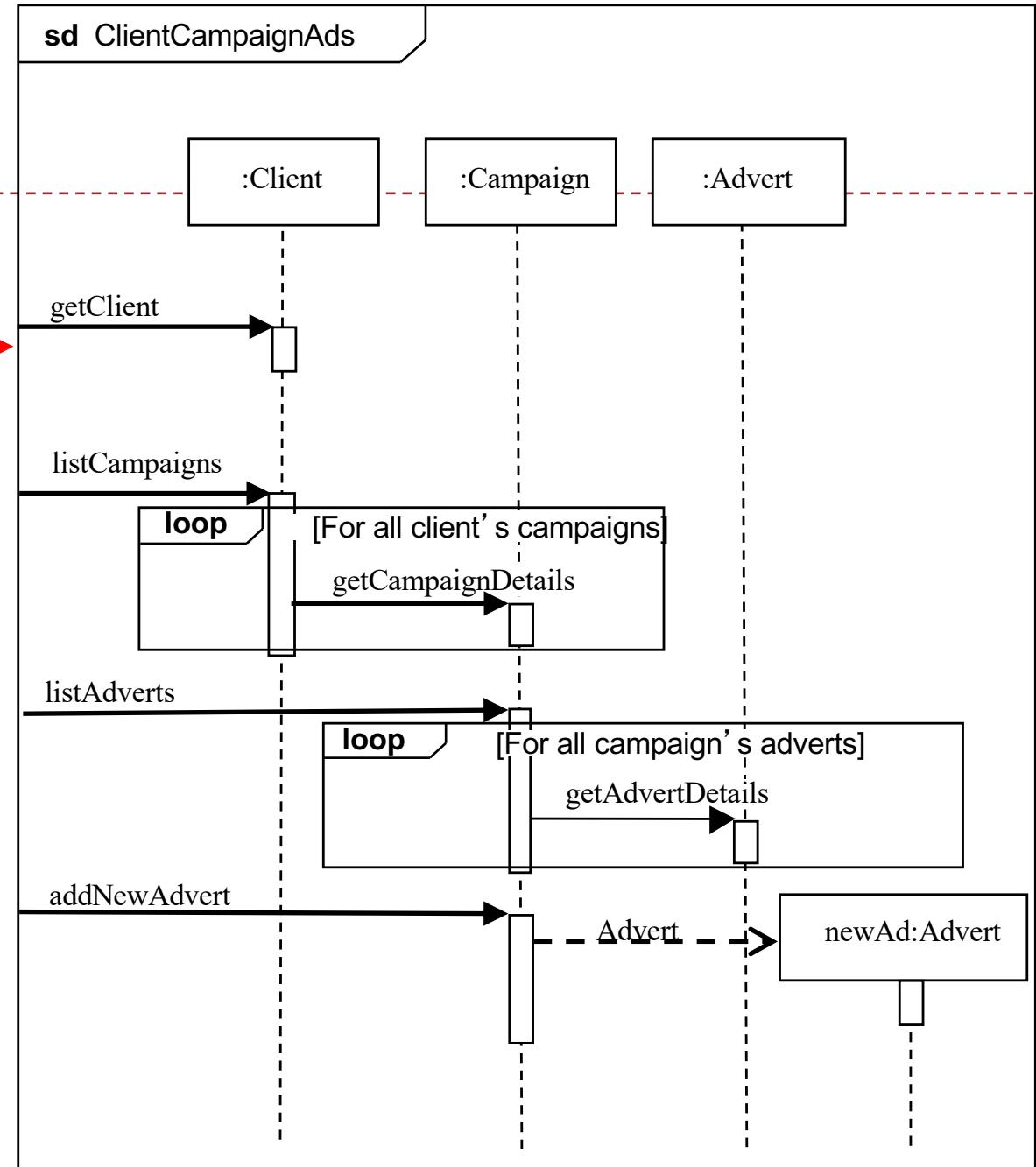
sd Add a new advert to a campaign



Lifeline representing the interaction between a group of objects



Sequence diagram
referenced in the
Add a new advert to a
campaign sequence
diagram



Communication diagrams

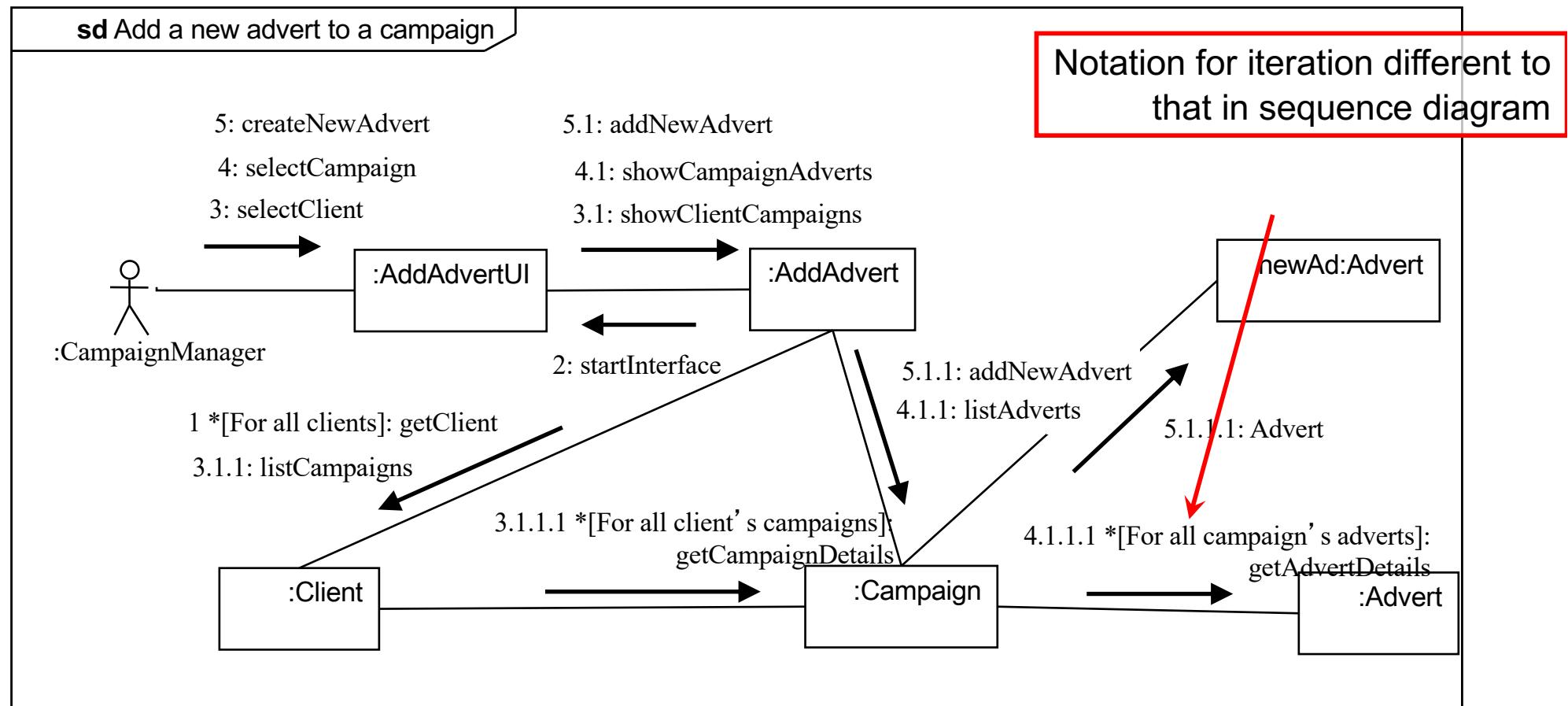
Communication (or Collaboration) Diagrams

- ▶ Hold the same information as sequence diagrams.
- ▶ Show links between objects that participate in the collaboration.
- ▶ No time dimension, sequence is captured with sequence numbers.
- ▶ Sequence numbers are written in a nested style (for example, 3.1 and 3.1.1) to indicate the nesting of control within the interaction that is being modelled.



Communication Diagrams

Compare with Sequence
diagram on slide 17

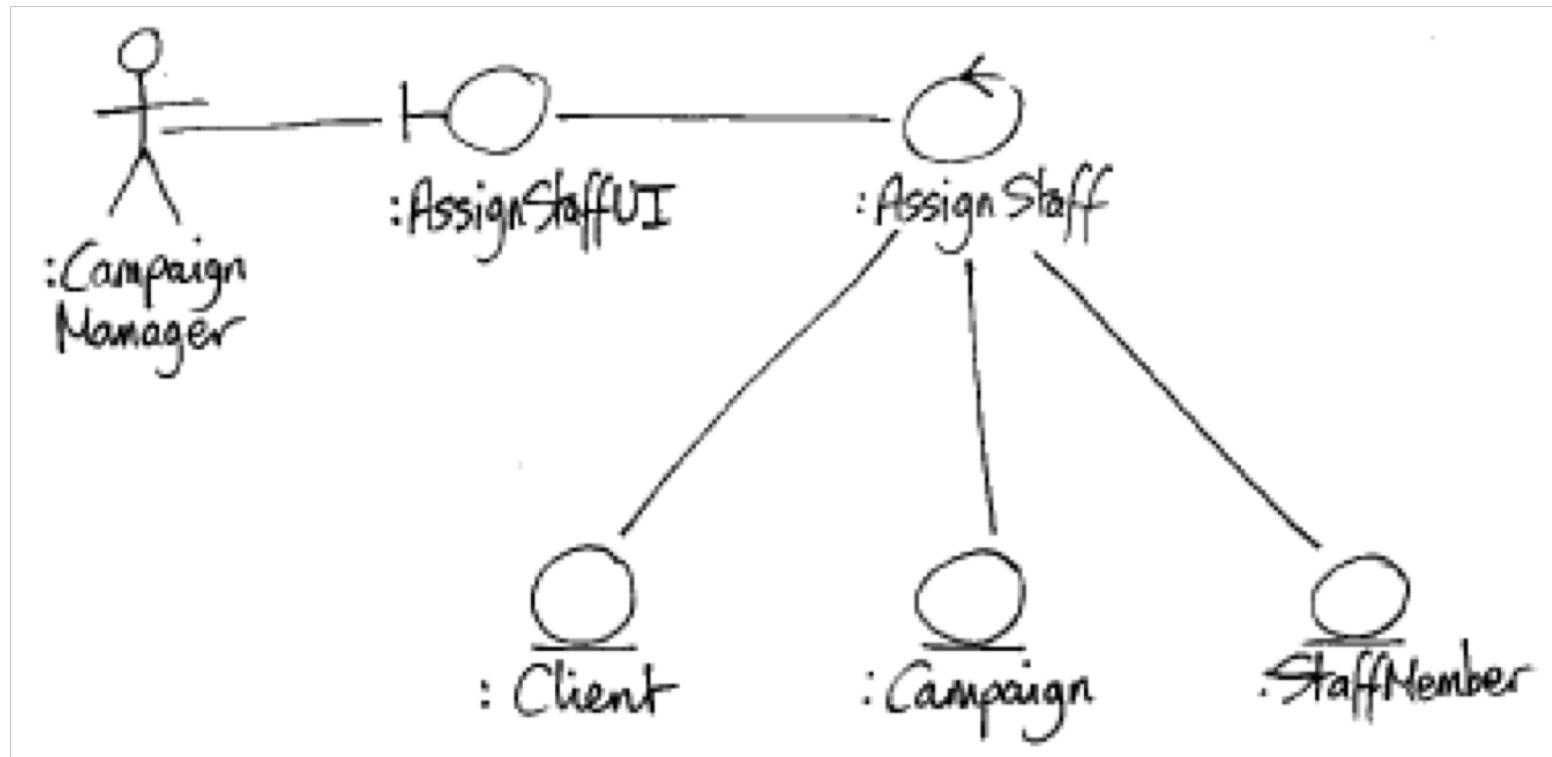


Communication diagram: Message Labels

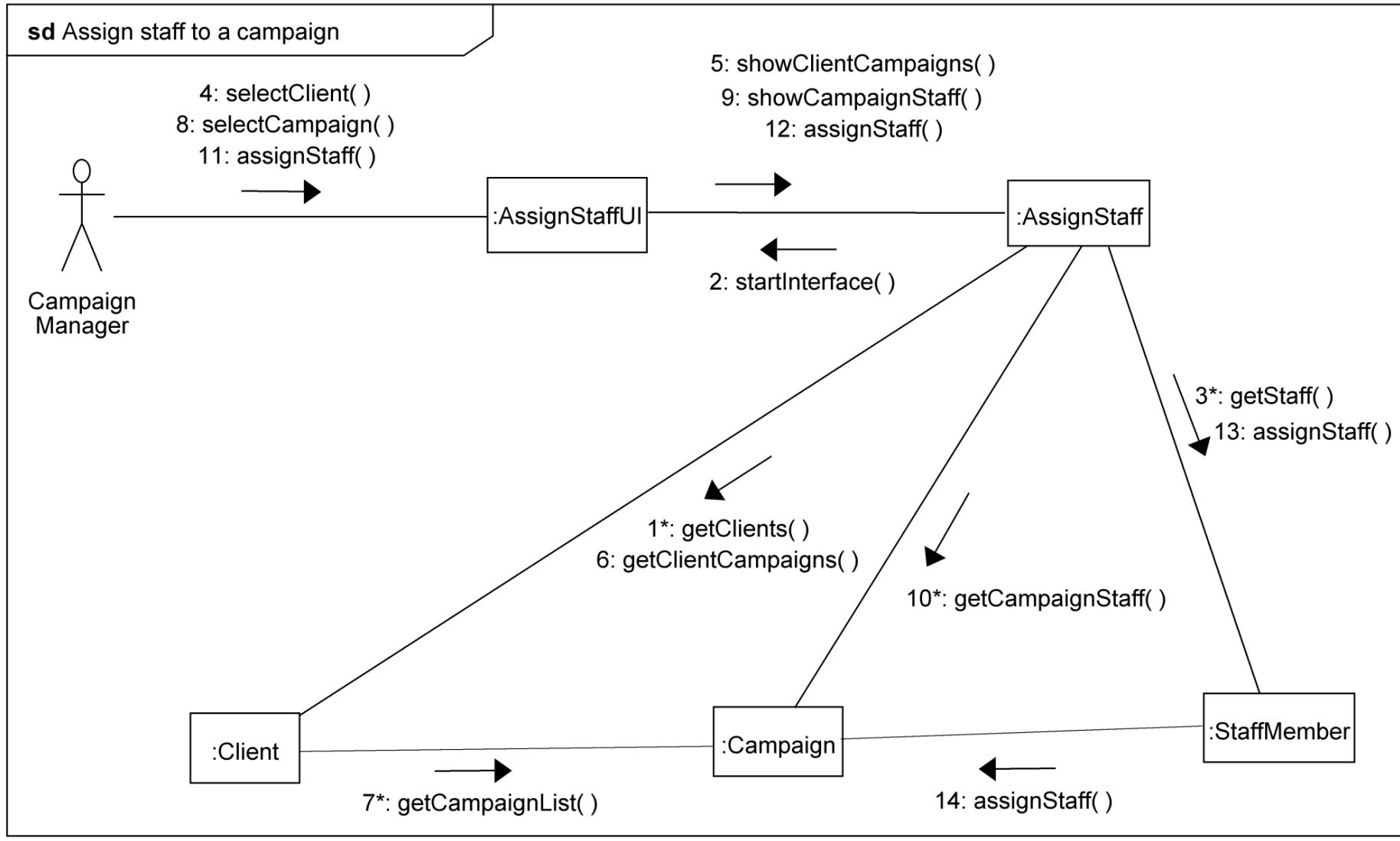
Type of message	Syntax example
Simple message.	4 : addNewAdvert
Nested call with return value. <i>The return value is placed in the variable name.</i>	3.1.2 : name = getName
Conditional message. <i>This message is only sent if the condition [balance > 0] is true.</i>	5 [balance > 0] : debit(amount)
Iteration	4.1 * [For all adverts] : getCost



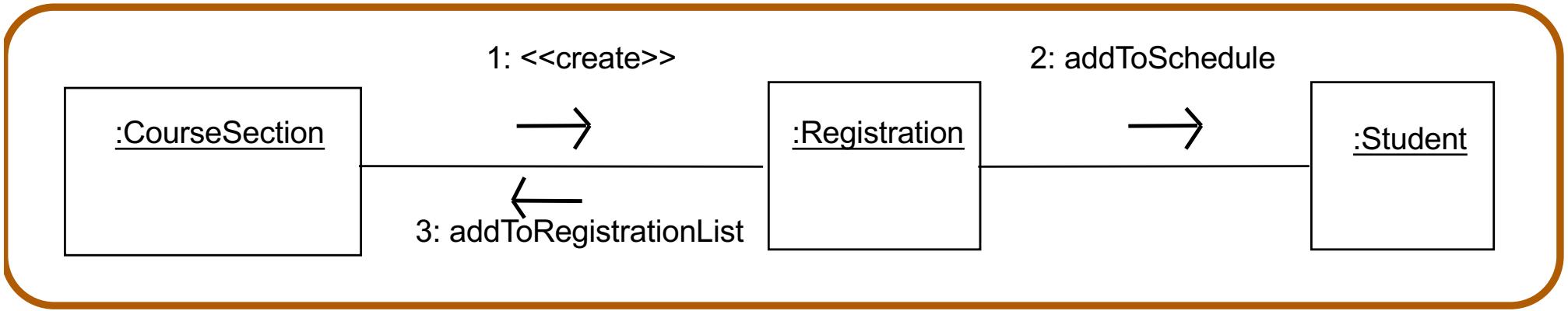
Early Draft Communication Diagram



More Developed Communication Diagram



Collaboration diagrams: possible implementations

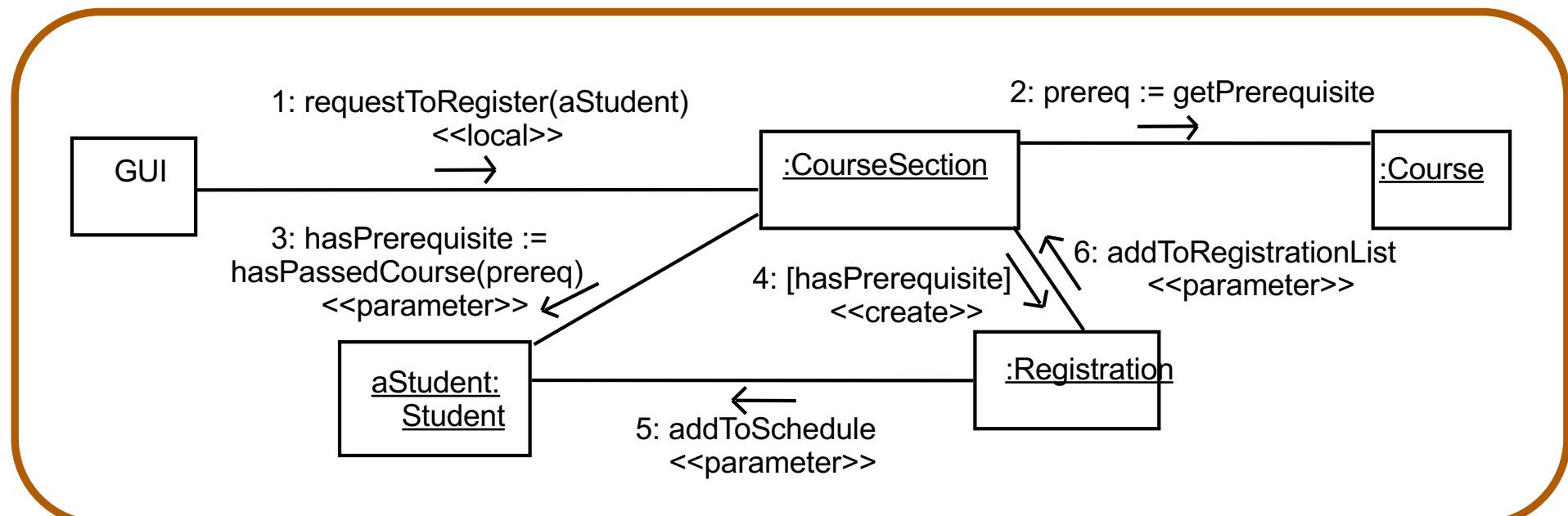
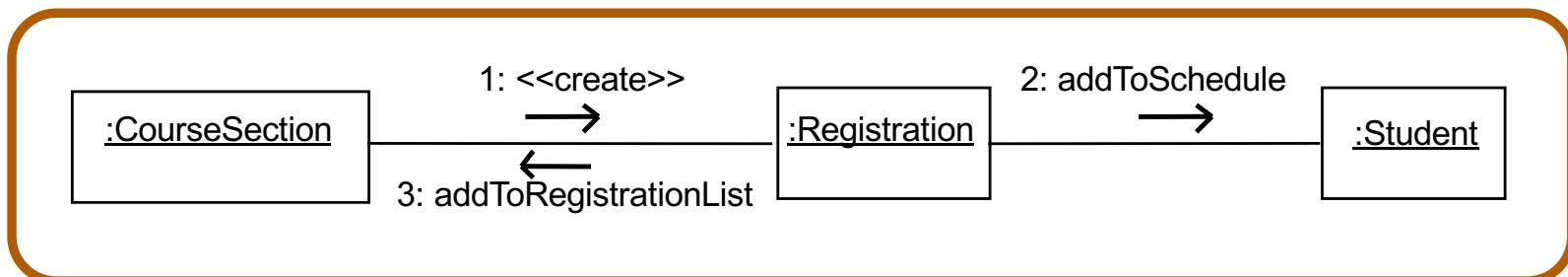


```
public class courseSection {  
    void someMethod(Student s)  
        Registration r = new Registration(this, s);  
        ...  
}
```

```
public class Registration {  
    Registration(c CourseSection, Student s) {  
        student.addToSchedule();  
        c.addToRegistrationList(this);  
        ...  
    }  
}
```

See slides 13-14
on Model
Consistency

Collaboration diagrams: abstraction and refinement



Summary

- ▶ Objects collaborate to provide system functionality
- ▶ Two approaches which model the same thing:
 - ▶ Sequence Diagrams
 - ▶ Collaboration/Communication Diagrams
- ▶ Guidelines for producing Sequence diagrams (will be the same for Collaboration diagrams)
- ▶ Need to check the consistency between diagrams
- ▶ Can be used to create the class diagram
- ▶ Can generate the initial code for each Class based on these diagrams

Exercises

- ▶ Write a set of Java classes and their methods that implement some of the diagrams in this presentation

Further reading

- ▶ Object-Oriented Software Engineering: Practical Software Development using UML and Java, Second Edition, Timothy C. Lethbridge and Robert Laganière, McGraw Hill, 2001
- ▶ See Chapter 9 in Bennett



Class: Requirements / Use Cases

- Use-case diagrams

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Requirements changes should be minimal

- ▶ Why?

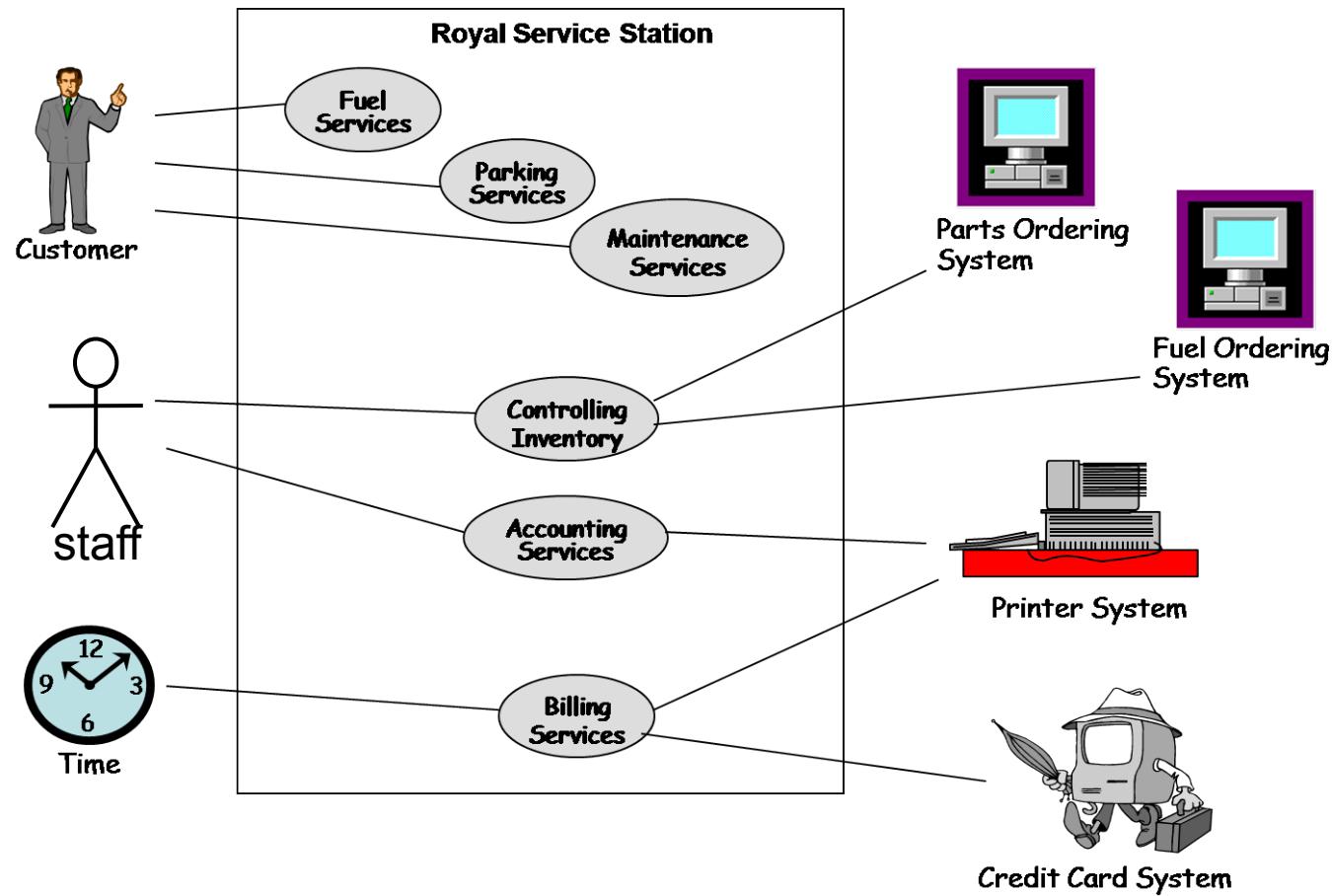




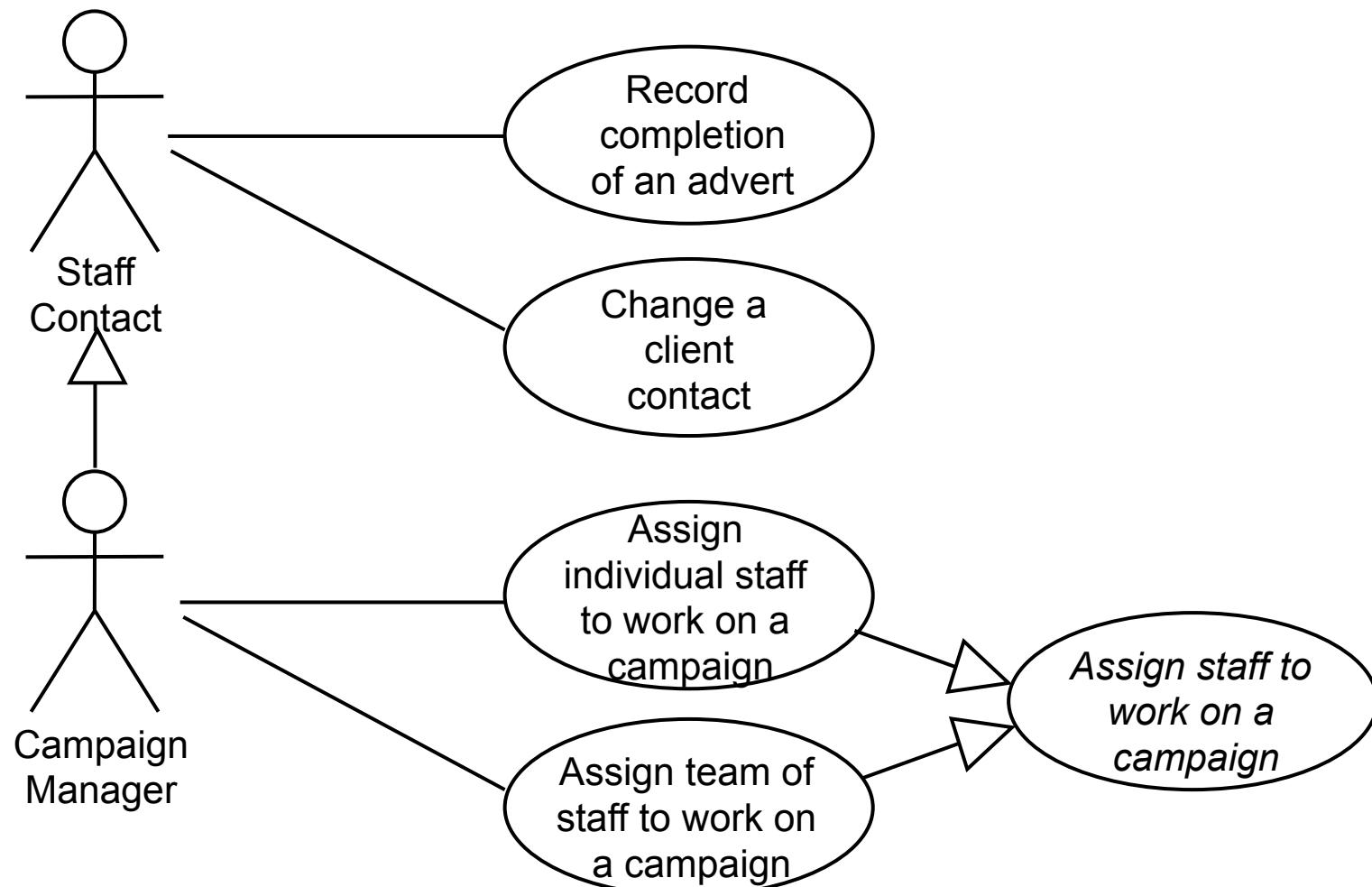
Use-case modelling

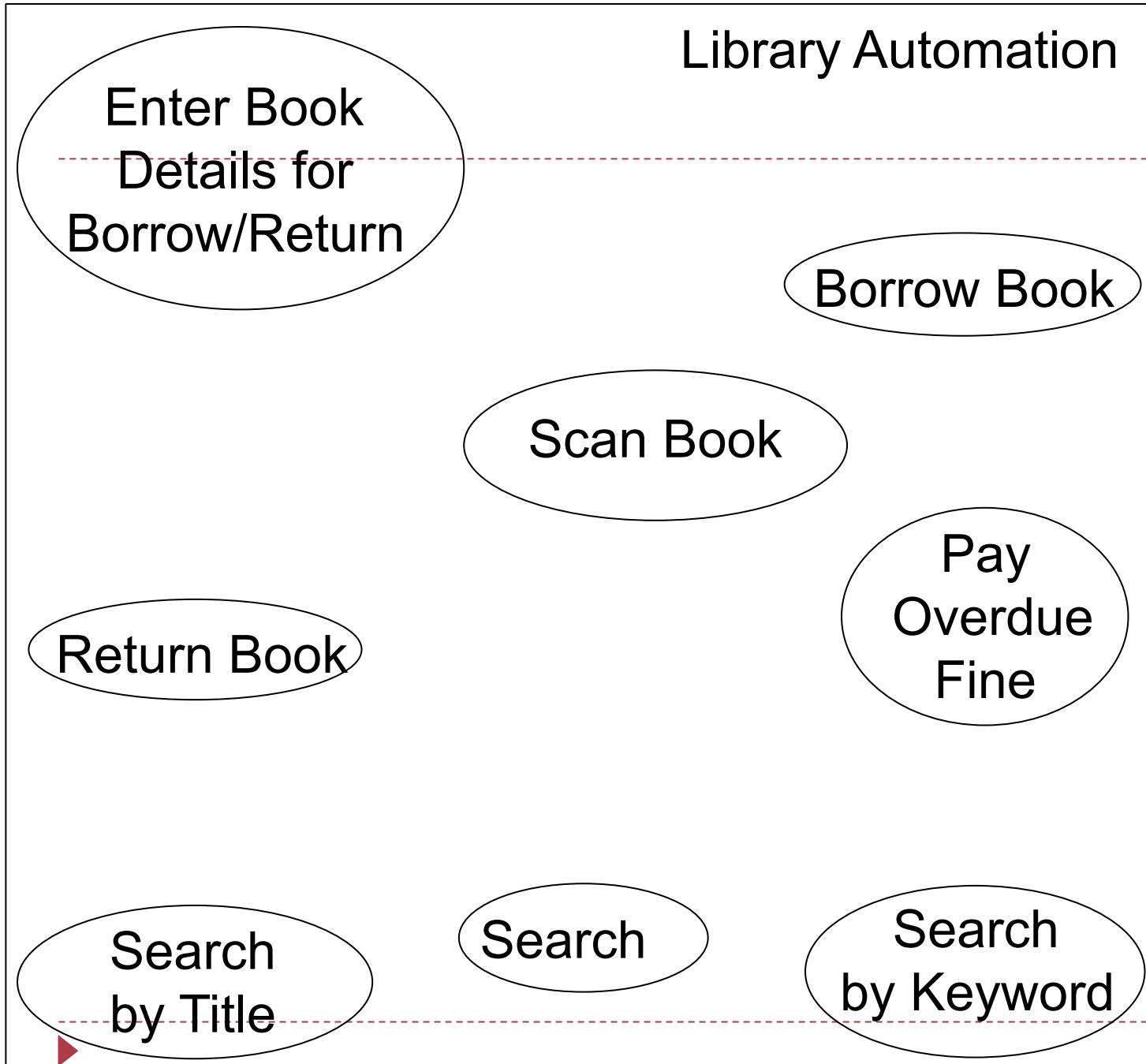


Use-case diagrams example: actors

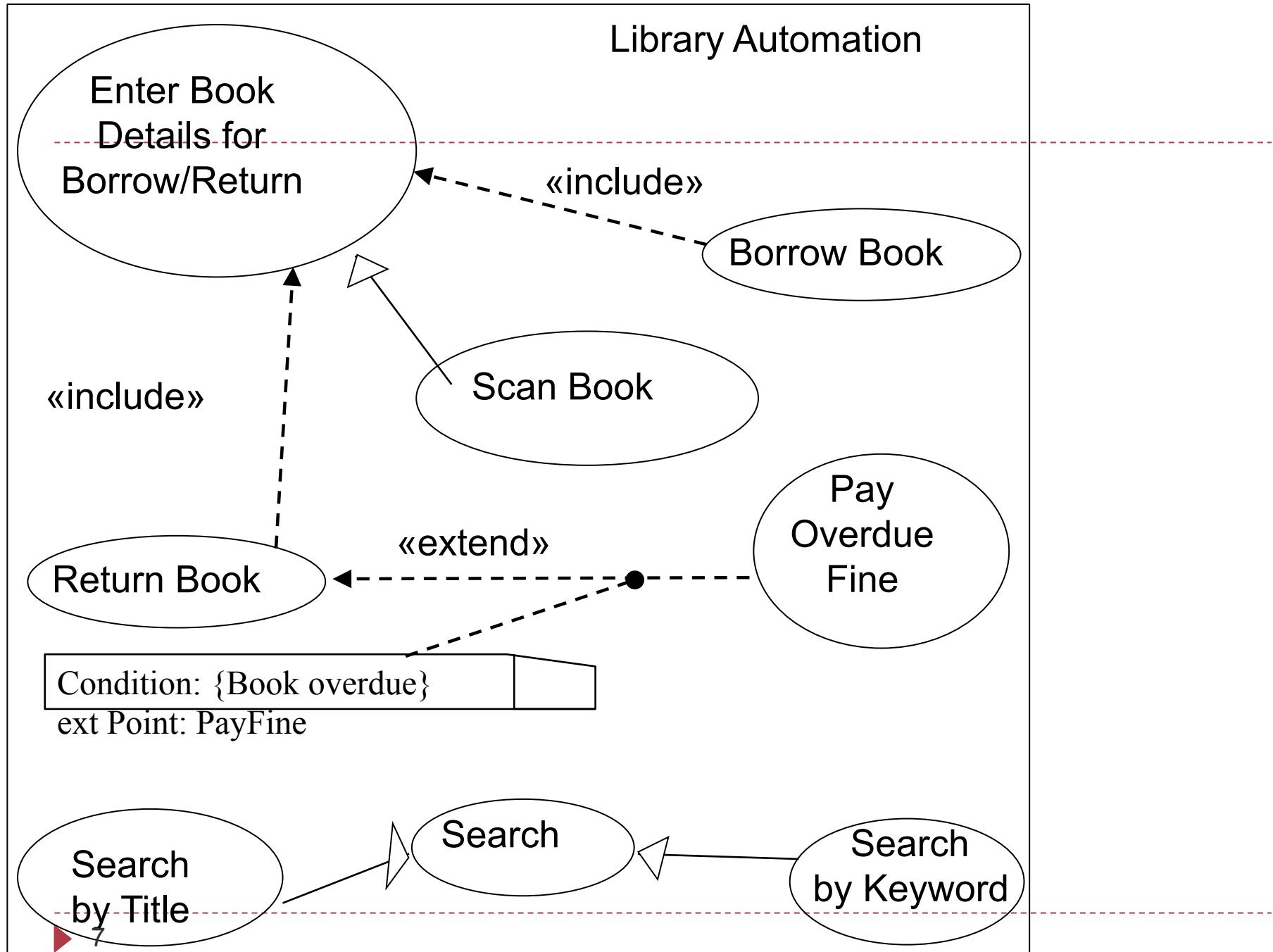


Use-case diagrams example: generalization





Please
fill in
use case
relationships:
<<generalisation>>
«extend»
«include»



Specifying Control Using State Machines

- CE202 Software Engineering
- M. Gardner



In this lecture you will learn:

- ▶ how to identify requirements for control in an application;
- ▶ how to model object life cycles using state machines;
- ▶ Basic state machine features
- ▶ More advanced features
- ▶ how to develop state machine diagrams from interaction diagrams;
- ▶ how to ensure consistency with other UML models.



State

- ▶ The current state of an object is determined by the current value of the object's attributes and the links that it has with other objects.
- ▶ For example the class StaffMember has an attribute startDate which determines whether a StaffMember object is in the probationary state.



State

- ▶ A state describes a particular condition that a modelled element (e.g. object) may occupy for a period of time while it awaits some event or *trigger*.
- ▶ The possible states that an object can occupy are limited by its class.
- ▶ Objects of some classes have only one possible state.
- ▶ Conceptually, an object remains in a state for an interval of time.



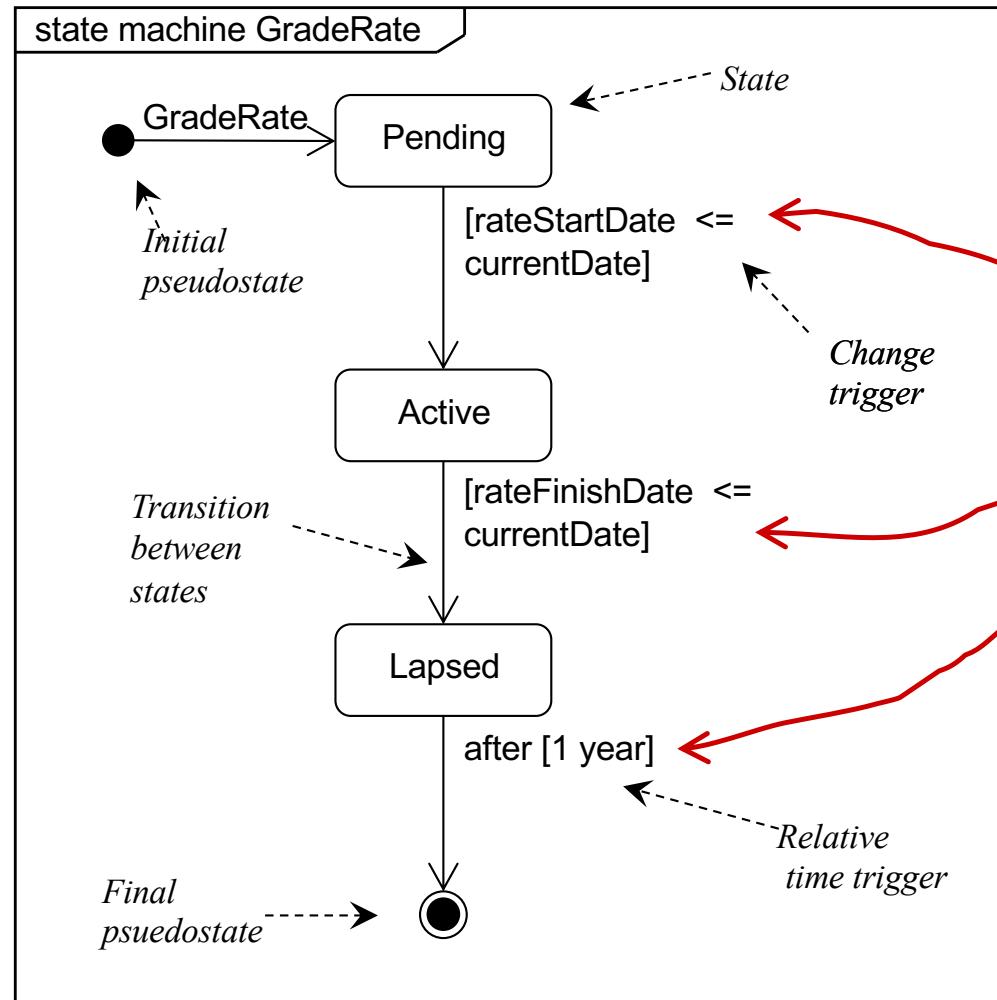
State machine example

- ▶ The current state of a GradeRate object can be determined by the two attributes rateStartDate and rateFinishDate.
- ▶ An enumerated state variable may be used to hold the object state, possible values would be **Pending**, **Active** or **Lapsed**.



State machine example (continued)

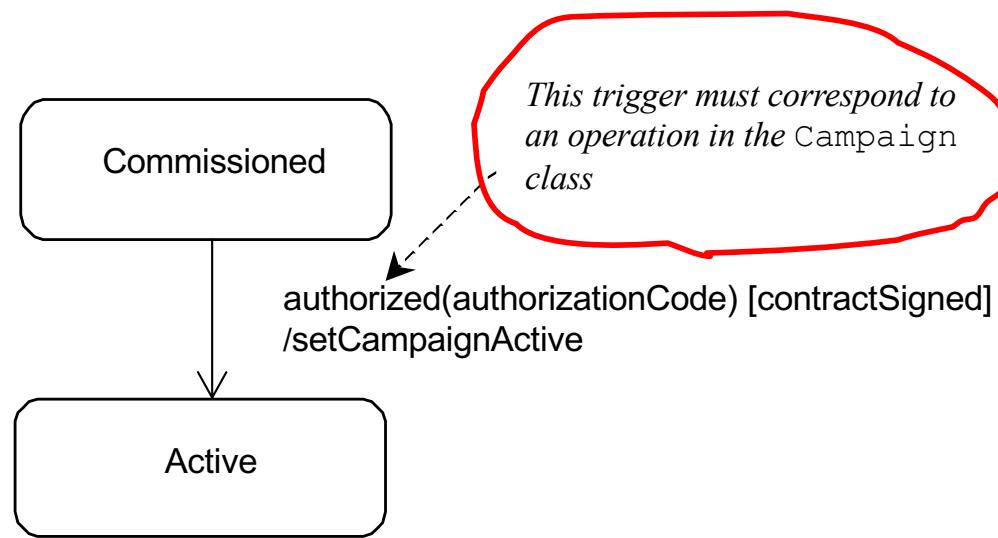
state machine for
the class
`GradeRate`.



Movement from one state to another is dependent upon events that occur with the passage of time. These are triggers

Example Event (with more detailed trigger information)

State machine for a **Campaign** object



Call and Signal Events

- ▶ *trigger-signature* '[' *constraint* ']' '/' *activity-expression*
- ▶ Where *trigger-signature* takes following form
 - ▶ *event-name* '(' *parameter-list* ')'
 - ▶ Where *event-name* may be the call or signal name
 - ▶ Where *parameter-list* contains parameters of the form, separated by commas:
 - ▶ *parameter-name* ':' *type-expression*
 - ▶ Can use single-quotes for literals
- ▶ A constraint is a guard condition. The transition only occurs if the guard condition is true (Boolean)
- ▶ *activity-expression* is executed when a trigger is fired



Activity Expressions

- ▶ May have multiple events
- ▶ Separated by semi-colons

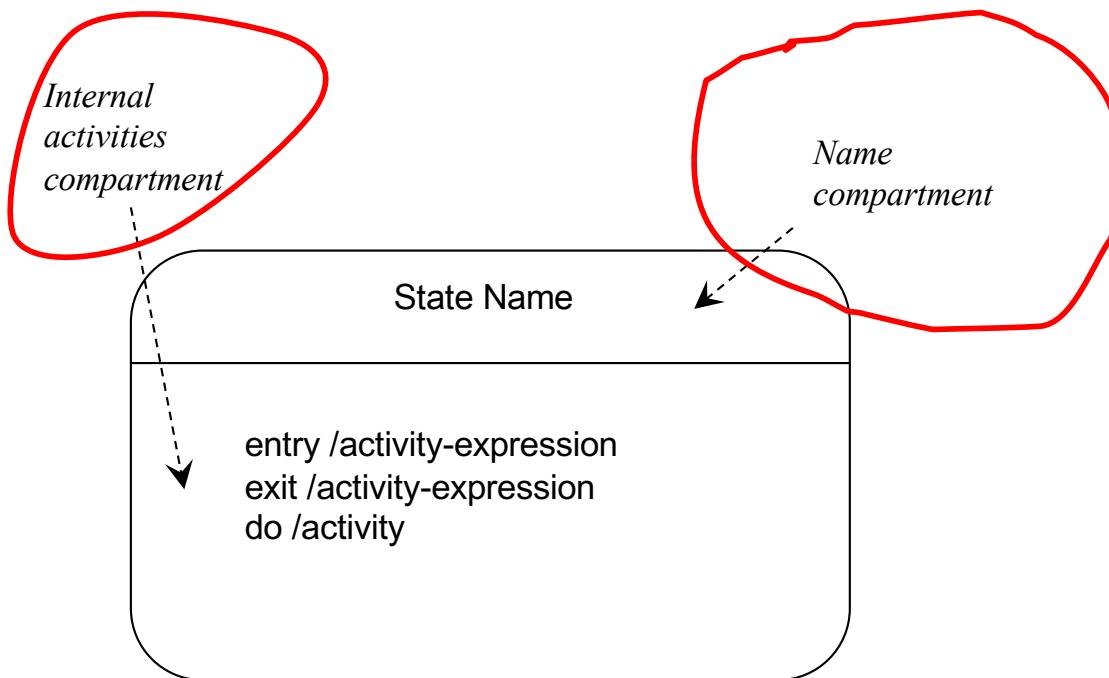
Example:

left-mouse-down(location) [validItemSelected] / menuChoice = pickMenuItem(location); menuChoice.highlight

The SEQUENCE of actions is significant as it determines the order in which they are executed



Internal Activities



Used to model internal activities associated with a state

Three types of internal activity: entry, exit and state activities (do)

Transition into the state causes the entry activity to fire

Transition out of the state causes the exit activity to fire



Internal Activities/Transitions

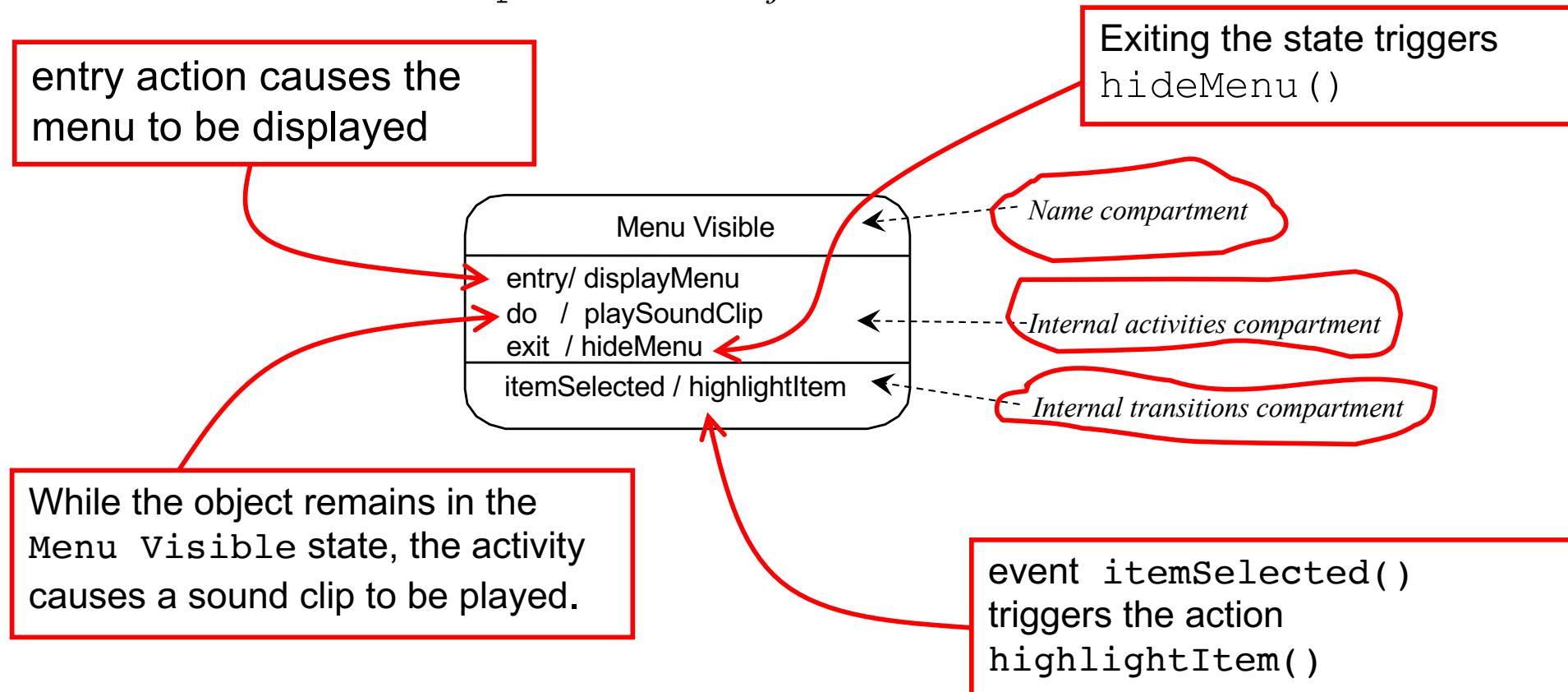
- ▶ Internal activities
 - ▶ ‘entry’ ‘/’ activity-name ‘(‘ parameter-list ‘)’
 - ▶ ‘exit’ ‘/’ activity-name ‘(‘ parameter-list ‘)’
 - ▶ ‘do’ ‘/’ activity-name ‘(‘ parameter-list ‘)’

- ▶ Internal transitions
 - ▶ Same syntax as triggers



An example: ‘Menu Visible’ State

Menu Visible state for a
DropDownMenu object.

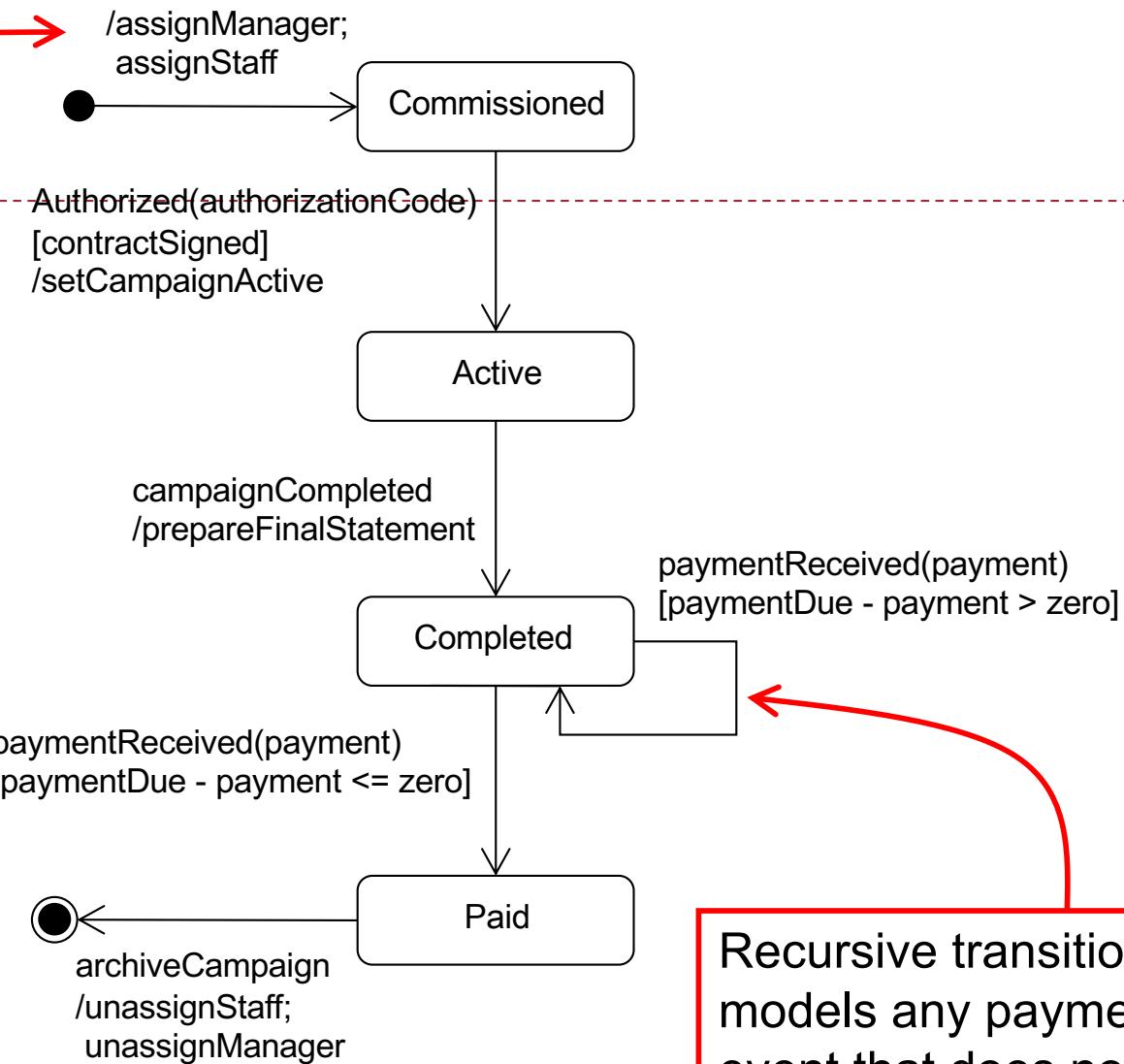


In this example, the entry action causes the menu to be displayed. While the object remains in the Menu Visible state, the activity causes a sound clip to be played and, if the event itemSelected() occurs, the action highlightItem() is invoked. It is important to note that when the event itemSelected() occurs the Menu Visible state is not exited and entered and as a result the exit and entry actions are not invoked. When the state is actually exited the menu is hidden.

Action-expression
assigning manager and
staff on object creation

state machine
for the class
Campaign.

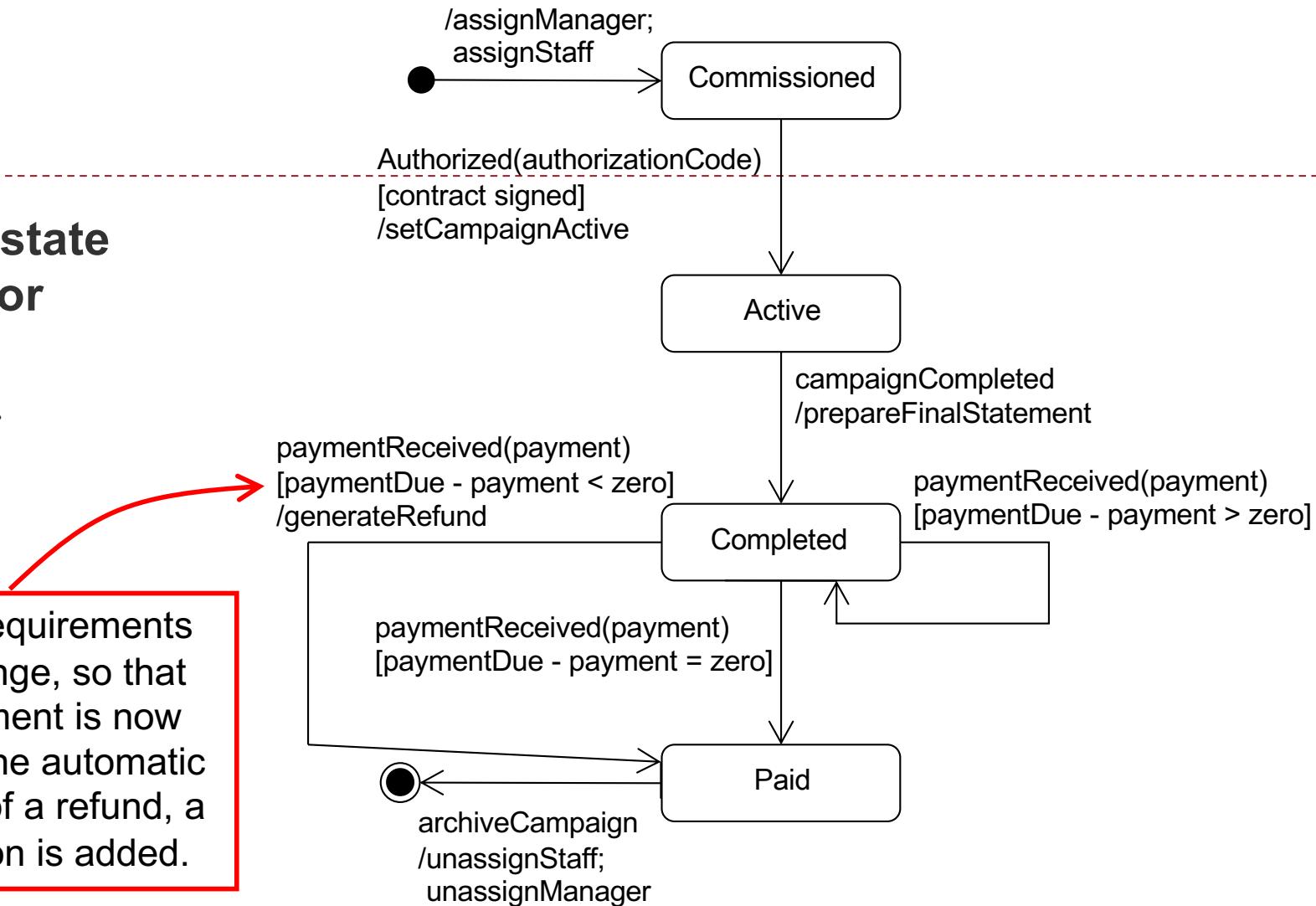
Guard condition ensuring
complete payment
before entering Paid



The recursive transition from the **Completed** state models any payment event that does not reduce the amount due to zero or beyond. Only one of the two transitions from the **Completed** state (one of which is recursive) can be triggered by the `paymentReceived` event since the guard conditions are mutually exclusive. It would be bad practice to construct a state machine where one event can trigger two different transitions from the same state. A life cycle is only unambiguous when all the transitions from each state are mutually exclusive.

Recursive transition
models any payment
event that does not
reduce the amount due
to zero or beyond.

A revised state machine for the class Campaign

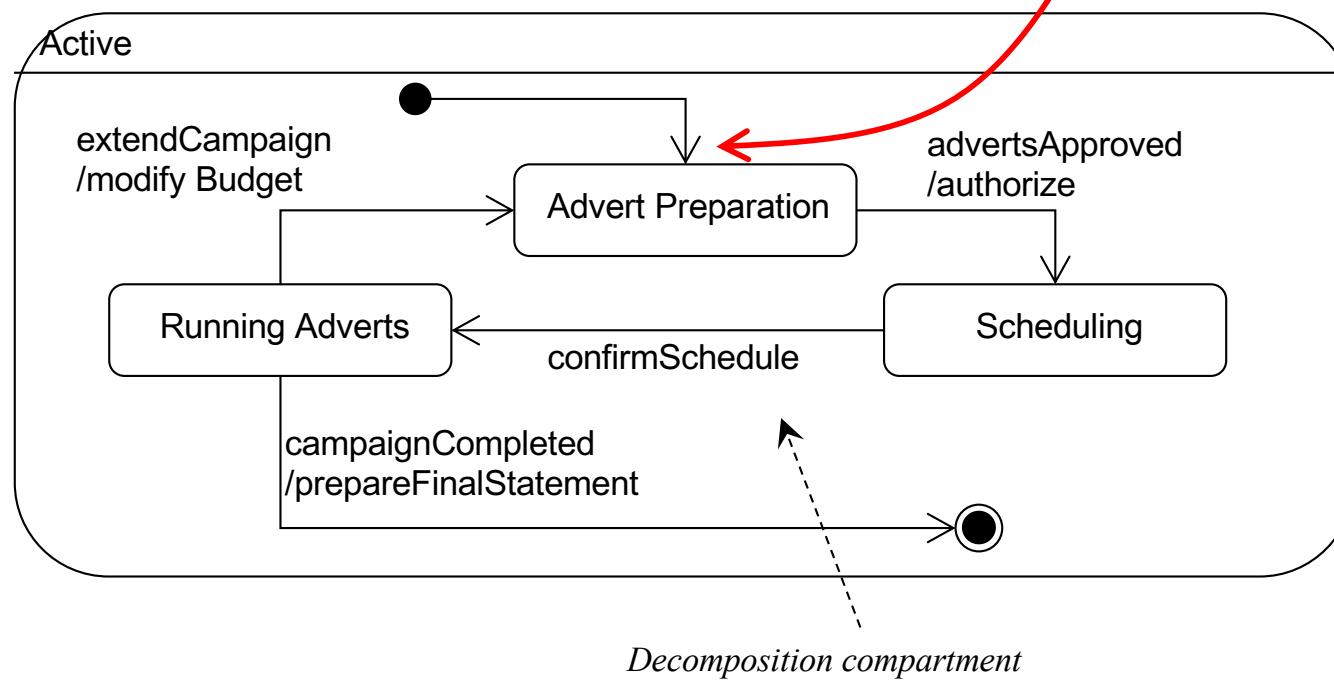


If the user requirements were to change, so that an overpayment is now to result in the automatic generation of a refund, the state machine can be changed as follows. Since the action that results from an overpayment is different from the action that results from a payment that reduces paymentDue to zero, a new transition is needed from the Completed state to the Paid state. The guard conditions from the Completed state must also be modified.

Nested Substates

The Active state of Campaign showing nested substates.

The transition from the initial pseudostate symbol should not be labelled with an event but may be labelled with an action, though it is not required in this example



In the nested state machine within the Active state, there is an initial state symbol with a transition to the first substate that a Campaign object enters when it becomes active. The transition from the initial pseudostate symbol to the first substate (Advert Preparation) should not be labelled with an event but it may be labelled with an action, though it is not required in this example. It is implicitly fired by any transition to the Active state. A final pseudostate symbol may also be shown on a nested state diagram. A transition to the final pseudostate symbol represents the completion of the activity in the enclosing state (i.e. Active) and a transition out of this state triggered by the completion event. This transition may be unlabelled (as long as this does not cause any ambiguity) since the event that triggers it is implied by the completion event.

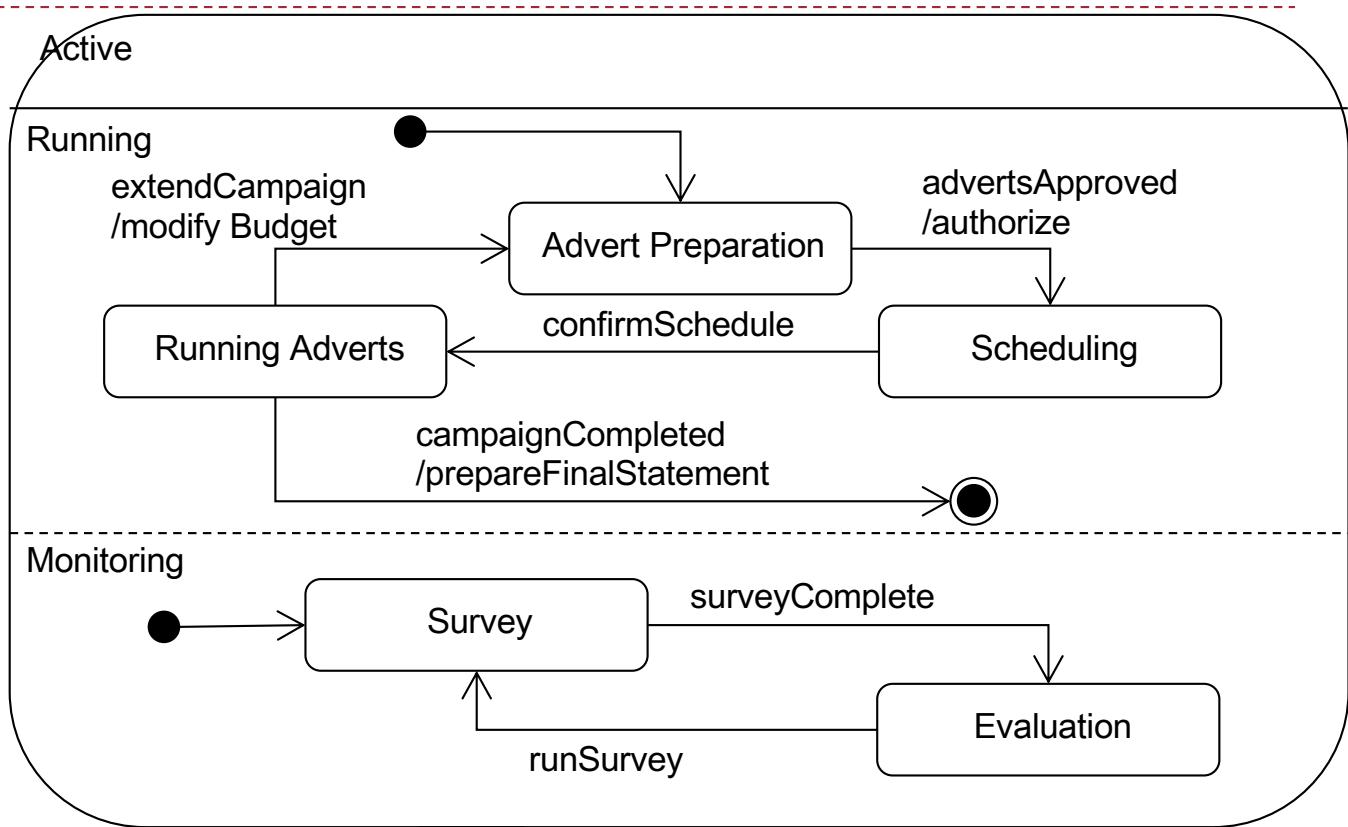
Summary of basic UML state machines

- ▶ What we have covered so far, forms the basis for most of the state machines we will consider in CE202
- ▶ Basic state machines will need:
 - ▶ **Start point** and an **initial starting state** (no trigger required)
 - ▶ Transitions to other states (triggers and **optional** activity expressions)
 - ▶ Each state will require a **sensible state name**
 - ▶ Decide on the level of detail required – **start simple**
 - ▶ Sometimes you can specify internal activities (but not always necessary – depends on the level of detail required)
 - ▶ Sometimes have nested state machines if needed
 - ▶ Sometimes an **exit point** (but not always)

Additional UML state machine features

The next slides describe additional features of UML state machines (some of which will be covered in the class exercises)

The Active state with concurrent substates.



Suppose that further investigation reveals that a campaign is surveyed and evaluated while it is also active. A campaign may occupy either the Survey substate or the Evaluation substate when it is in the Active state. Transitions between these two states are not affected by the campaign's current state in relation to the preparing and running of adverts. We model this by splitting the Active state into two concurrent nested state machines, Running and Monitoring, each in a separate sub-region of the Active state machine. This is shown by dividing the state icon with a dashed line.

Concurrent States

- ▶ A transition to a complex state is equivalent to a simultaneous transition to the initial states of each concurrent state machine.
- ▶ An initial state must be specified in both nested state machines in order to avoid ambiguity about which substate should first be entered in each concurrent region.
- ▶ A transition to the Active state means that the Campaign object simultaneously enters the Advert Preparation and Survey states.

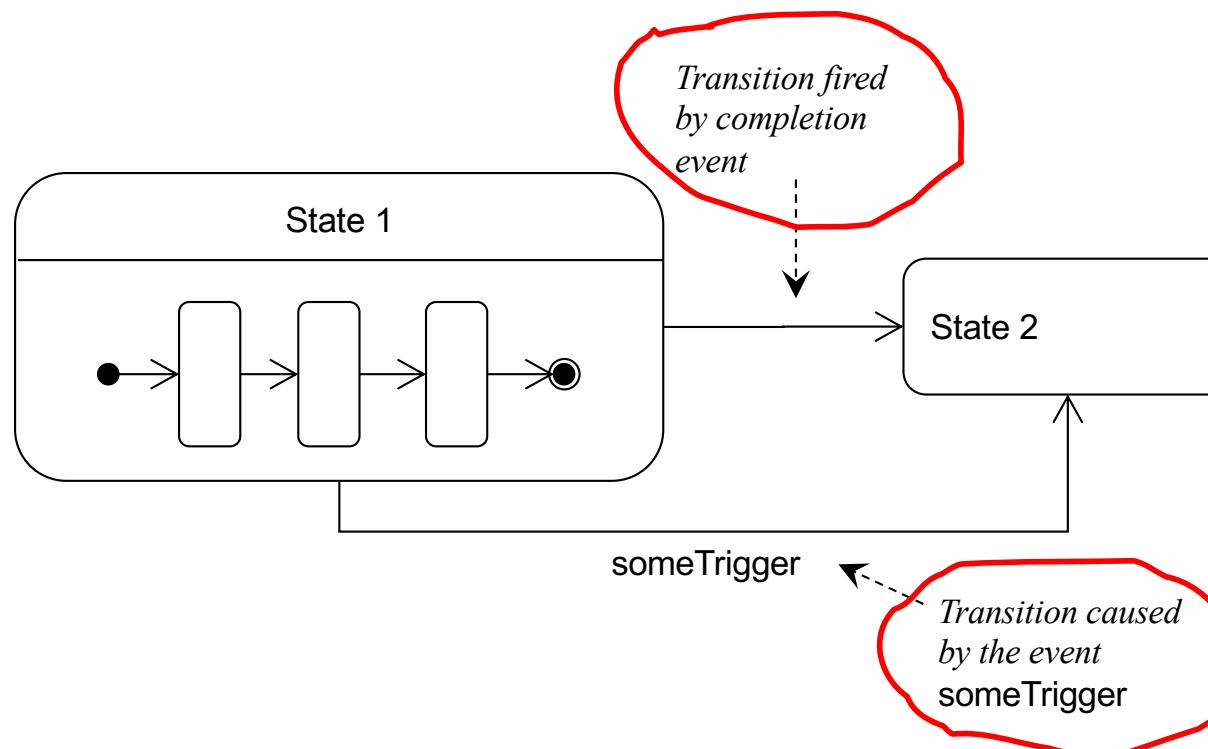


Concurrent States

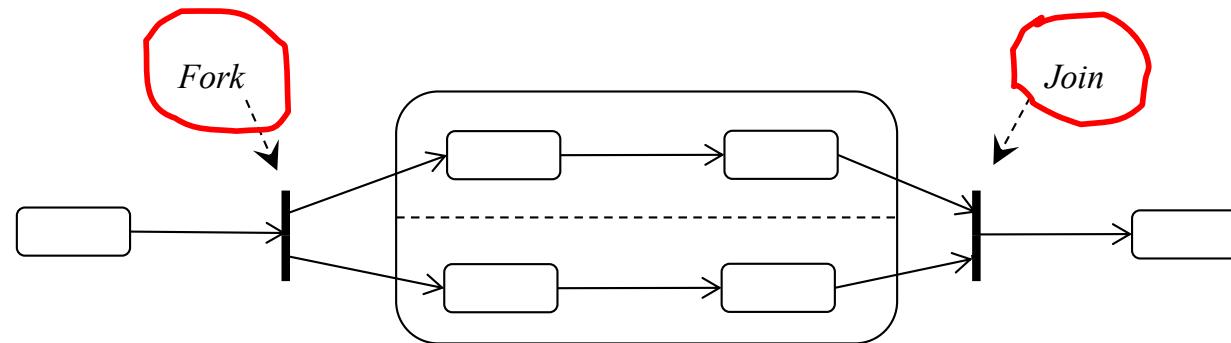
- ▶ Once the composite state is entered a transition may occur within either concurrent region without having any effect on the state in the other concurrent region.
- ▶ A transition out of the `Active` state applies to all its substates (no matter how deeply nested).



Completion Event



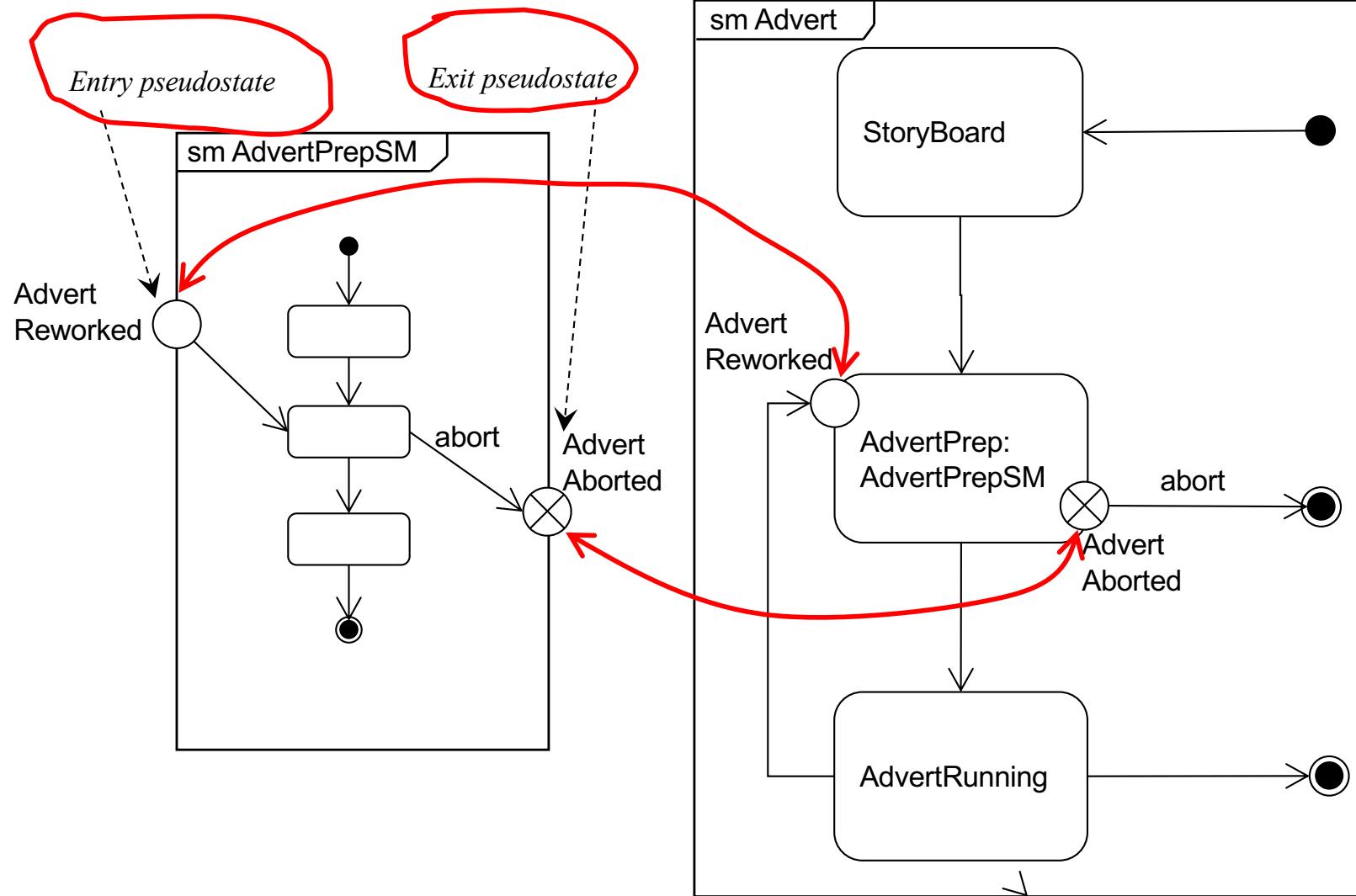
Synchronized Concurrent Threads.



- Explicitly showing how an event triggering a transition to a state with nested concurrent states causes specific concurrent substates to be entered.
- Shows that the composite state is not exited until both concurrent nested state machines are exited.

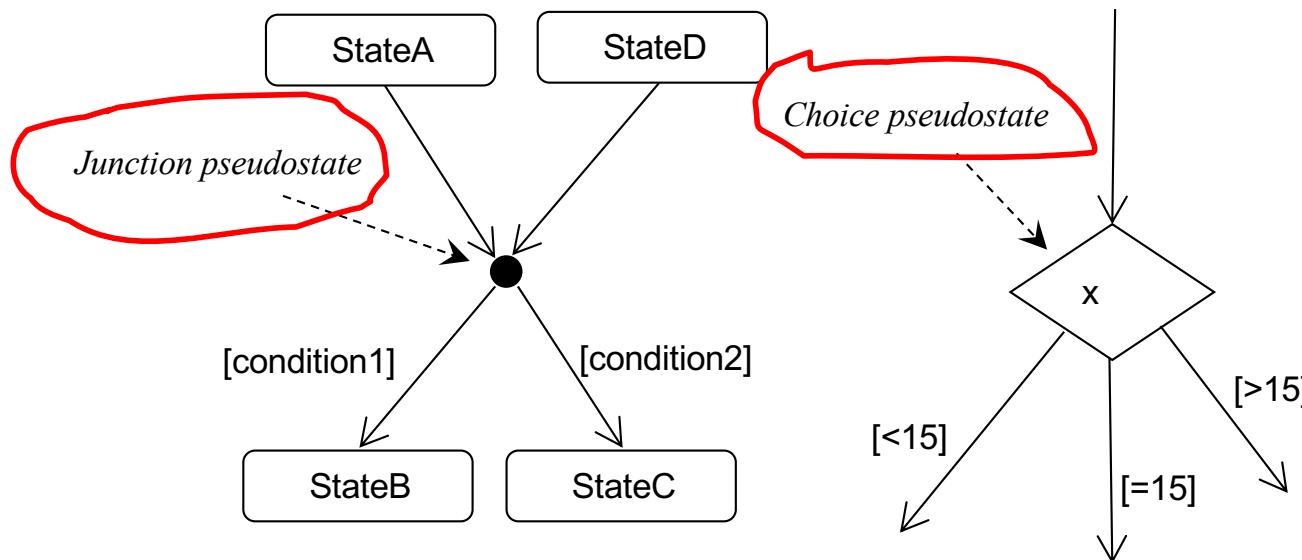


Entry & Exit Pseudostates



Used for modeling exceptional entry to or exit from a submachine state
Can be shown in the frame-boundary OR inside the frame

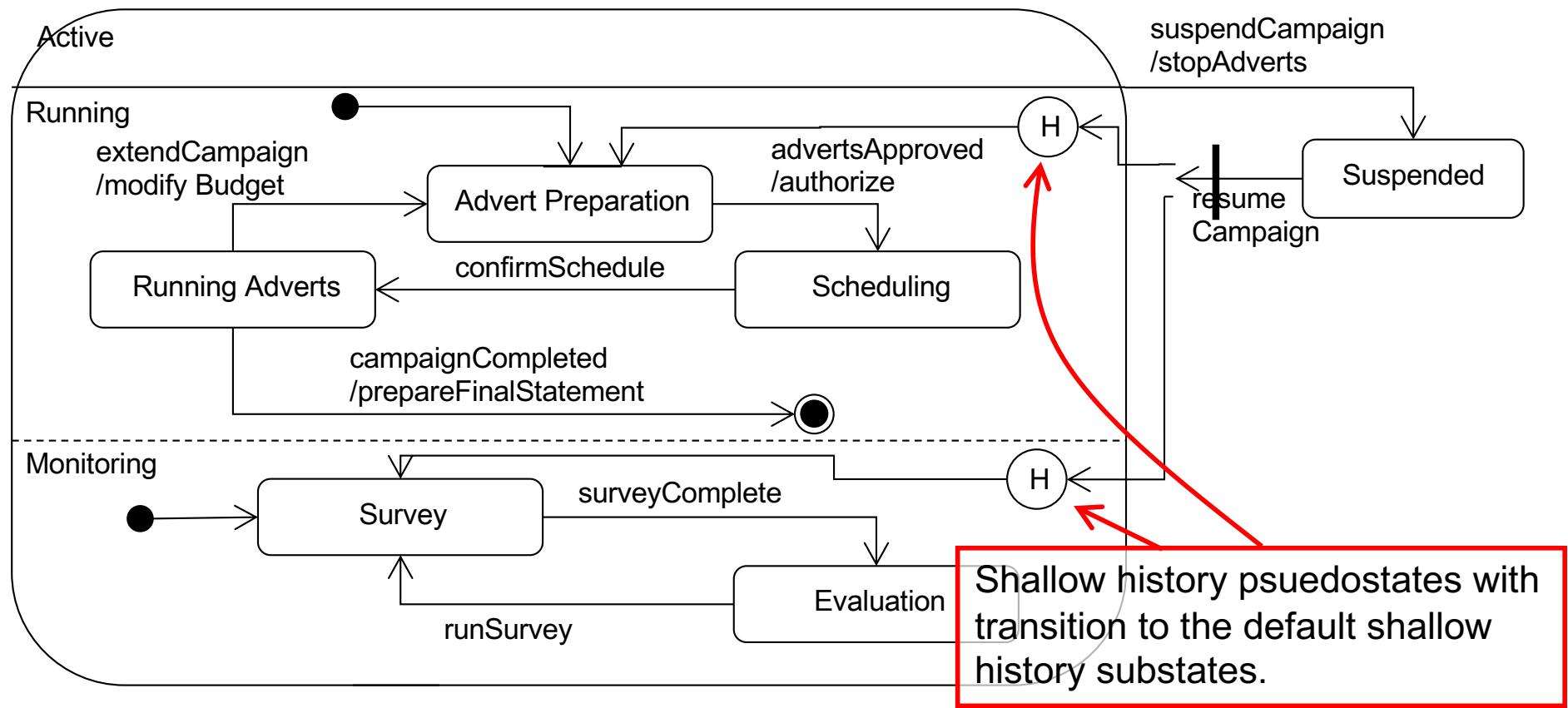
Junction & Choice Pseudostates



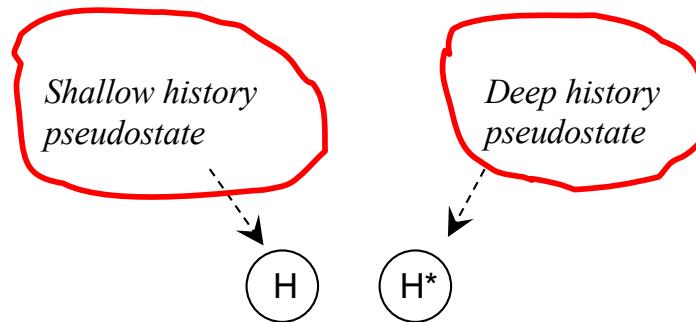
When there are many entry transitions and one exit this is known as a *Merge*

When there are several exit transitions and only one entry transition this is known as a *Static Conditional Branch*

History Pseudostates



History Pseudostates



Use deep history pseudostate if more than 1 substare



Preparing state machines

Behavioural Approach

1. Examine all interaction diagrams that involve each class that has heavy messaging.
2. Identify the incoming messages on each interaction diagram that may correspond to events. Also identify the possible resulting states.
3. Document these events and states on a state machine.
4. Elaborate the state machine as necessary to cater for additional interactions as these become evident, and add any exceptions.



Behavioural Approach

5. Develop any nested state machines (unless this has already been done in an earlier step).
6. Review the state machine to ensure consistency with use cases. In particular, check that any constraints that are implied by the state machine are appropriate.

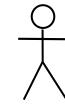


Behavioural Approach

7. Iterate steps 4, 5 and 6 until the state machine captures the necessary level of detail.
8. Check the consistency of the state machine with the class diagram, with interaction diagrams and with any other state machines and models.



sd Record completion of a campaign



:CampaignManager

:CompleteCampaign

:Client

:Campaign

Sequence
Diagram with
States Shown

:CompleteCampaignUI

loop

[For all clients]

getClient

startInterface

selectClient

showClientCampaigns

listCampaigns

loop

[For all client's campaigns]

getCampaignDetails()

Active

Active state

completeCampaign

completeCampaign

completeCampaign

Completed

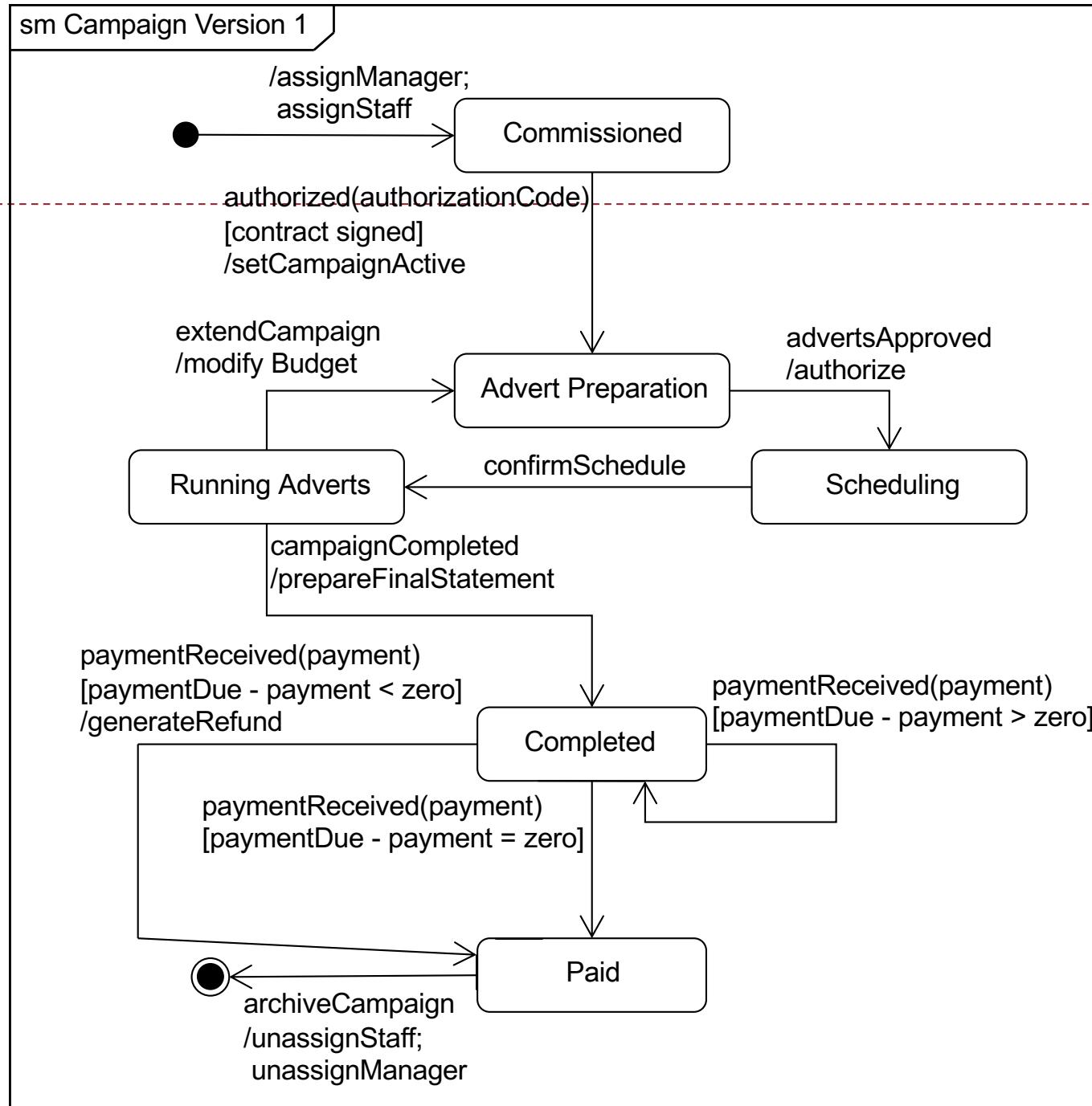
Completed state



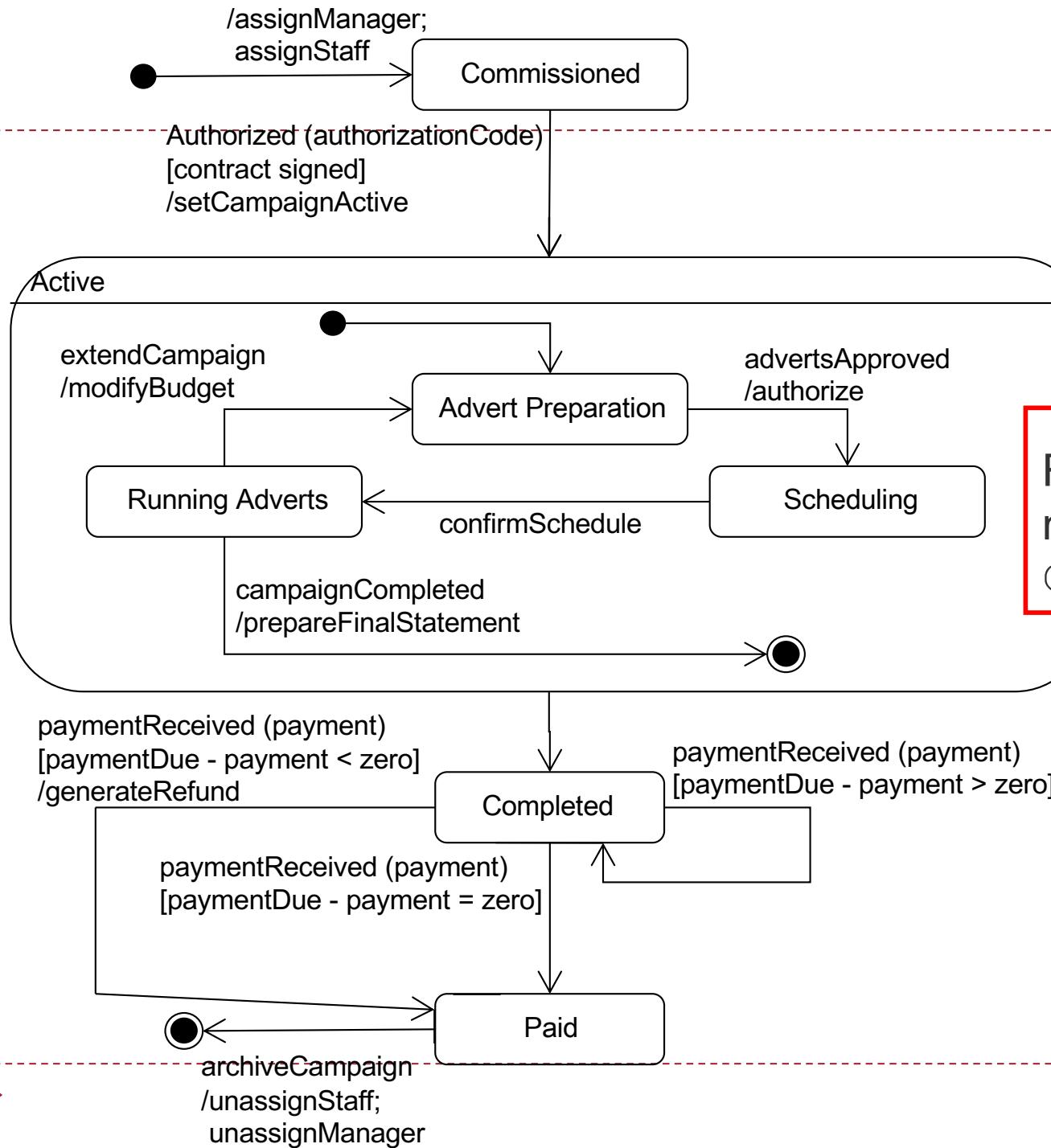
Behavioural approach example

- ▶ The following slides show how a state machine might be developed
- ▶ More detail is added to the state machine as the analysis progresses
- ▶ Always start with an initial simple state machine and then elaborate from there

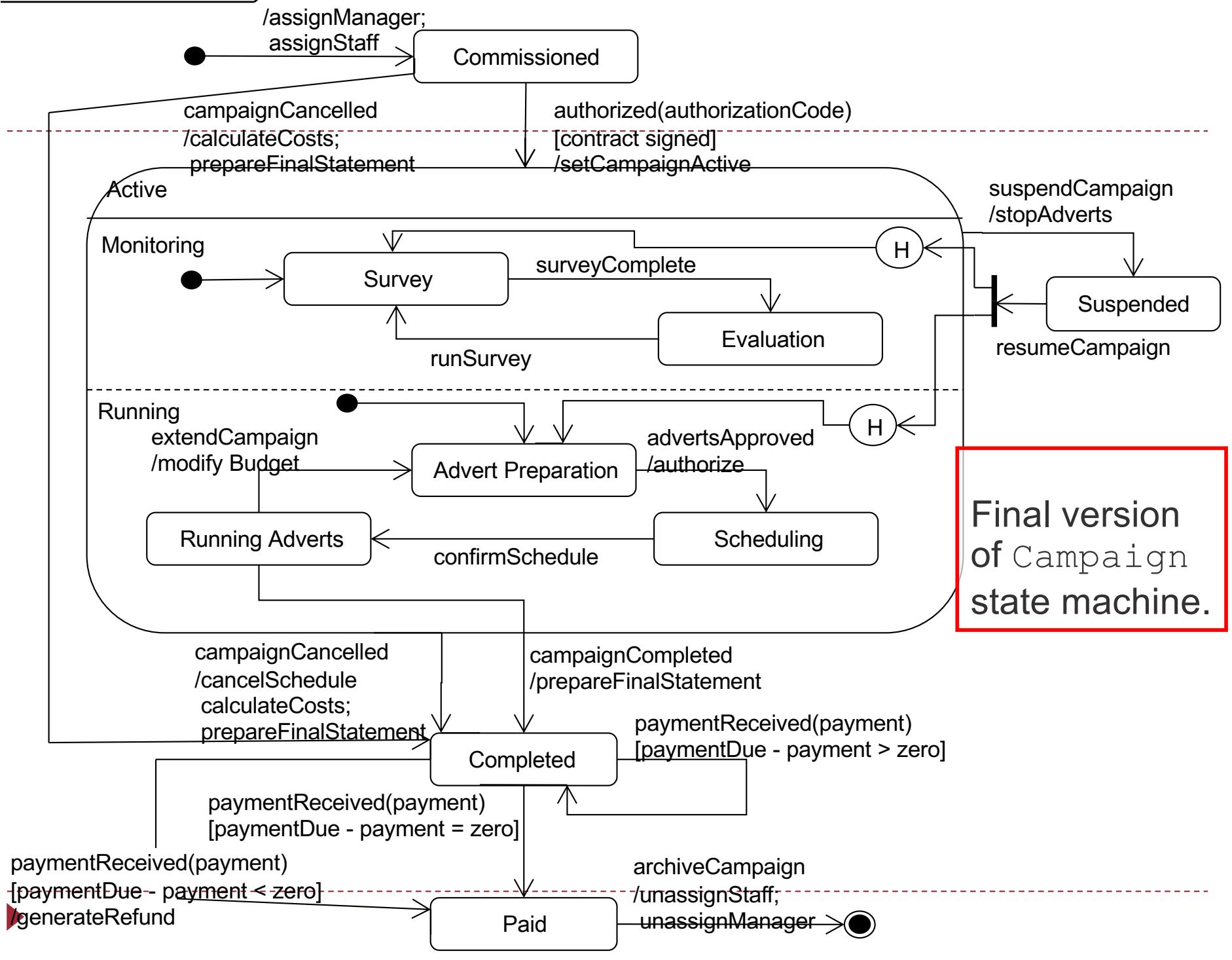
Initial state machine for the Campaign class—a behavioral approach.



sm Campaign Version 2



sm Campaign Version 3



Consistency checking your state machine

- ▶ Every event should appear as an incoming message for the appropriate object on an interaction diagram(s).
- ▶ Every action should correspond to the execution of an operation on the appropriate class, and perhaps also to the dispatch of a message to another object.
- ▶ Every event should correspond to an operation on the appropriate class (but note that not all operations correspond to events).
- ▶ Every outgoing message sent from a state machine must correspond to an operation on another class.



Consistency Checking

- ▶ Consistency checks are an important task in the preparation of a complete set of models.
- ▶ Highlights omissions and errors, and encourages the clarification of any ambiguity or incompleteness in the requirements.



Summary

In this lecture you have learned about:

- ▶ how to identify requirements for control in an application;
- ▶ how to model object life cycles using state machines;
- ▶ how to develop state machine diagrams from interaction diagrams;
- ▶ how to model concurrent behaviour in an object;
- ▶ how to ensure consistency with other UML models.



References

- ▶ Chapter 11 Bennett
- ▶ UML 2.2 Superstructure Specification (OMG, 2009)
- ▶ Douglass B. P. (2004) Real-time UML: Advances in the UML for Real-time Systems (3rd Edition), Addison-Wesley.



UML Tutorial: Collaboration Diagrams

Robert C. Martin

Engineering Notebook Column

Nov/Dec, 97

In this column we will explore UML collaboration diagrams. We will investigate how they are drawn, how they are used, and how they interact with UML class diagrams.

UML 1.1

On the first of September, the three amigos (Grady Booch, Jim Rumbaugh, and Ivar Jacobson) released the UML 1.1 documents. These are the documents that have been submitted to the OMG for approval. If all goes well, the OMG will adopt UML by the end of this year.

The differences between the UML 1.0 and UML 1.1 notation are minimal. The previous article in this series, (September issue) has not been affected by the changes.

Dynamic models

There are three kinds of diagrams in UML that depict dynamic models. State diagrams describe how a system responds to events in a manner that is dependent upon its state. Systems that have a fixed number of states, and that respond to a fixed set of events are called finite state machines (FSM). UML has a rich set of notational tools for describing finite state machines. We'll be investigating them in another column.

The other two kinds of dynamic diagram fall into a category called Interaction diagrams. They both describe the flow of messages between objects. However, sequence diagrams focus on the order in which the messages are sent. They are very useful for describing the procedural flow through many objects. They are also quite useful for finding race conditions in concurrent systems. Collaboration diagram, on the other hand, focus upon the relationships between the objects. They are very useful for visualizing the way several objects collaborate to get a job done and for comparing a dynamic model with a static model

. Collaboration and sequence diagrams describe the same information, and can be transformed into one another without difficulty. The choice between the two depends upon what the designer wants to make visually apparent.

Sequence diagrams will be discussed in a future article. In this article we will be concentrating upon collaboration diagrams.

The interplay between static and dynamic models.

There is a tendency among novice OO designers to put too much emphasis upon static models. Static models depicting classes, inheritance relationships, and aggregation relationships are often the first diagrams that novice engineers think to create. Disastrously, they are sometimes the *only* diagrams that they create.

In fact, a static emphasis on object oriented design is inappropriate. Software design is about behavior; and behavior is dynamic. Object oriented design is a technique used to separate and encapsulate behaviors. Therefore an emphasis upon dynamic models is very important.

More important, however, is the interplay that exists between the static and dynamic models. A static model cannot be proven accurate without associated dynamic models. Dynamic models, on the other hand, do not adequately represent considerations of structure and dependency management. Thus, the designer must iterate between the two kinds of models, driving them to converge on an acceptable solution.

Example: A Cellular Phone.

Consider the software that controls a very simple cellular telephone. Such a phone has buttons for dialing digits, and a “send” button for initiating a call. It has “dialer” hardware and software that gathers the digits to be dialed and emits the appropriate tones. It has a cellular radio that deals with the connection to the cellular network. It has a microphone, a speaker, and a display.

From this simple spec, we might be tempted to create a static model as shown in Figure 1.

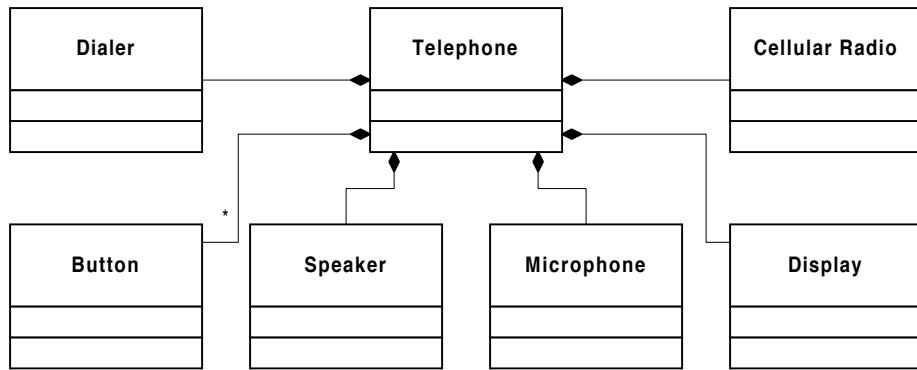


Figure 1: Static model of a Cellular Phone

It is very hard to argue with this static model. The composition relationships reflect, very clearly, the specification above. Indeed, the telephone “has” all the listed components. But is this the correct static model? How would be know?

One criterion is to compare the static model to the real world. Certainly Figure 1 passes this test. In the real world, a cellular phone “has” all the components shown above. However, experienced object oriented designers know that, while this test is essential, it is not sufficient. Figure 1 does not show the *only* static model that matches the real world of the cellular telephone. In order to choose between the many possible static models a more sensitive test is needed.

Specifying Dynamics.

How does the cellular phone work? To keep things simple, lets just look at how a customer might make a phone call. The use case for this interaction looks like this:

- ```
Use case: Make Phone Call
1. User presses the digit buttons to enter the phone number.
2. For each digit, the display is updated to add the digit to the
 phone number.
3. For each digit, the dialer generates the corresponding tone and
 emits it from the speaker.
4. User presses "Send"
5. The "in use" indicator is illuminated on the display
6. The cellular radio establishes a connection to the network.
7. The accumulated digits are sent to the network.
8. The connection is made to the called party.
```

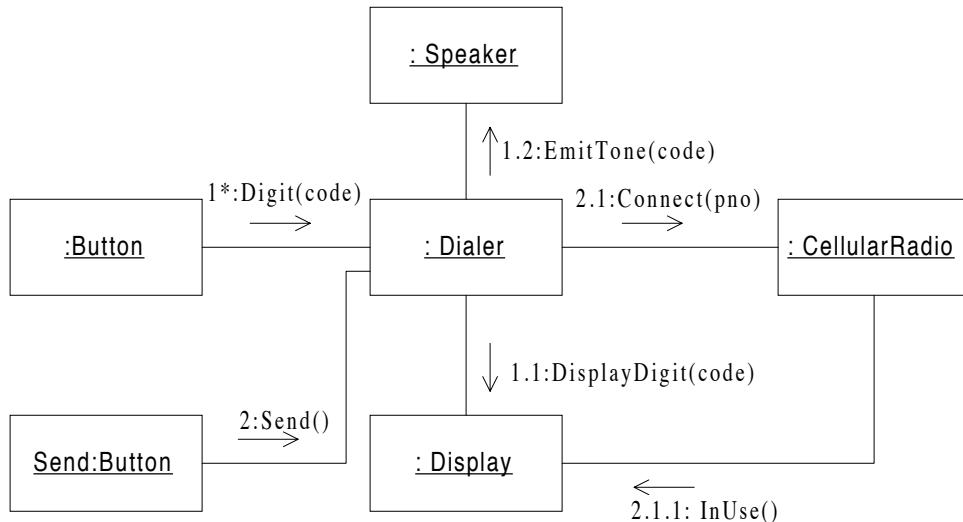
This is simplistic, but adequate for our purposes. The use case makes it clear that there is a procedure involved with making a call. How do the objects in the static model collaborate to execute this procedure?

Let’s trace the process one step at a time. The first thing that happens when this use case is initiated is that the user presses a digit button to begin entering the phone number. How does the software in the phone know that a button has been pushed?

There are a variety of ways that this can be accomplished; but they can all be simplified to having a **Button** object that sends a **digit** message. Which object should receive the digit message? It seems clear that it should be the **Dialer**. The **Dialer** must then tell the **Display** to show the new digit, and must tell the **Speaker** to emit the appropriate tone. The **Dialer** must also remember the digits in the list that accumulates the phone number. Each new button press follows the same procedure until the “Send” button is pressed.

When the “Send” button is pressed, the appropriate **Button** object sends the **Send** message to the **Dialer**. The **Dialer** then sends a **Connect** message to the **CellularRadio** and passes along the accumulated phone number. The **CellularRadio** then tells the **Display** to illuminate the “In Use” indicator.

This simple procedure is depicted in the collaboration diagram in Figure 2.



**Figure 2: Collaboration Diagram of “Make Phone Call” use case.**

## Syntax

First, let’s look at the syntax of the diagram above. The rectangles in this diagram depict objects, not classes. You can tell that they are objects because they are underlined. In UML, something that is underlined is an *instance*; whereas something that is not underlined is a template from which an instance can be created. Notice the object on the lower left entitled **Send:Button**. In UML the full name of an object is a composite that includes the name of its class. The name of the object and the name of the class are separated by a colon. Notice that all the other objects in the diagram are anonymous; they have no name, and their class is shown with a colon in front.

The lines connecting the objects are called links. Links are *instances* of associations. You are not allowed to create a link on a collaboration diagram if there is no corresponding association, (or aggregation, or composition) on a class diagram. Remember this rule, we’ll come back to it later.

The arrows represent messages; and are labeled with their names, sequence numbers, and arguments. The name of a message corresponds to the name of a member function. That member function must exist in the class that the receiving object is instantiated from. The sequence numbers show the order in which the messages occur. The sequence numbers are nested so that you can tell which messages are sent from within other messages.

For example, message **2:Send()** is sent to the **Dialer**. As a result the **Dialer** begins executing a member function. Before that function returns, it sends message **2.1:Connect(pno)** to the

`CellularRadio`. The `CellularRadio` then sends message `2.1.1:InUse()` to the `Display`. Thus, the dot structure of the sequence numbers make it easy to see the procedural nesting of the messages.

Message `1*:Digit(code)` has an asterisk in order to denote that it may occur many times before message 2. UML defines way to use this syntax to represent loops and conditions that are beyond the scope of this article. We'll come back to such details in a subsequent article.

### Reconciling the static model with the dynamic model.

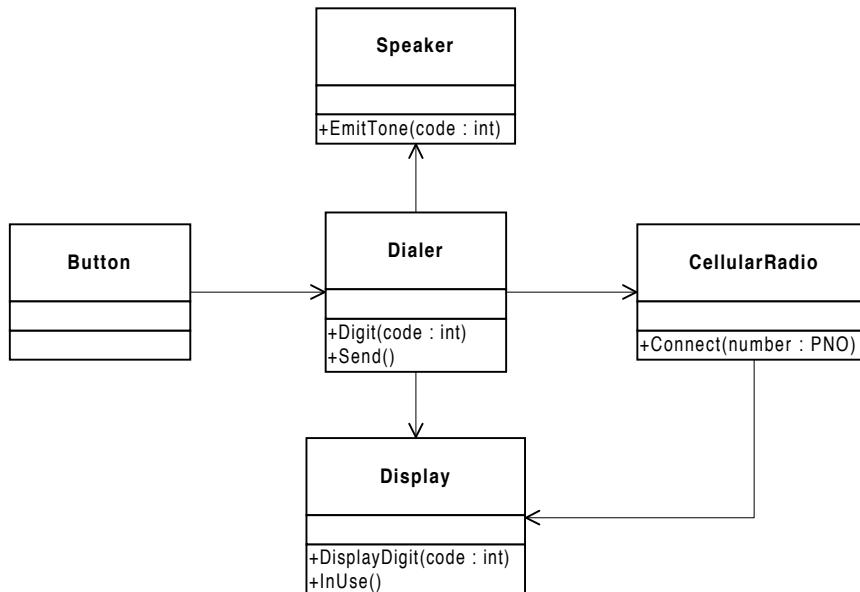
It should be clear that the structure of the objects in the dynamic model (Fig 1) does not look very much like the structure of the classes in the static model (Fig 2). Yet by the rule we talked about above, a link between objects must be represented by a relationship between the classes. Thus, we have a problem.

The problem could be that our dynamic model is incorrect. Perhaps we should force the dynamic model to look like the static model. However, consider what such a dynamic model would look like. There would be a `Telephone` object in the middle that would receive messages from the other objects. The `Telephone` object would respond to each incoming message with outgoing messages of its own.

Such a design would be highly coupled. The `Telephone` object would be the master controller. It would know about all the other objects, and all the other objects would know about it. It would contain all the intelligence in the system, and all the other objects would be correspondingly stupid. This is not desirable because such a “god” object becomes highly interconnected. When any part of it changes, other parts of it may break.

I prefer the dynamic model shown in Figure 2. The concerns are decentralized in a reasonable fashion. Each object has its own little bit of intelligence, and no particular object is in charge of everything. Changes to one part of the model do not necessarily ripple to other parts.

But this means that our static model is inappropriate. It should be changed to look like Figure 3. Notice that I have demoted all the composition relationships to associations. This is because none of the connected classes share a whole/part relationship with the others. The `Dialer` is not part of the `Button` class, The `CellularRadio` is not part of the `Dialer`, etc. Notice also that I have specified the direction of navigation. The dynamic model makes it very clear which class needs to navigate to which other classes. I have also added the member functions into the class icons; again because the dynamic model made them so apparent.



**Figure 3: Reconciled static model**

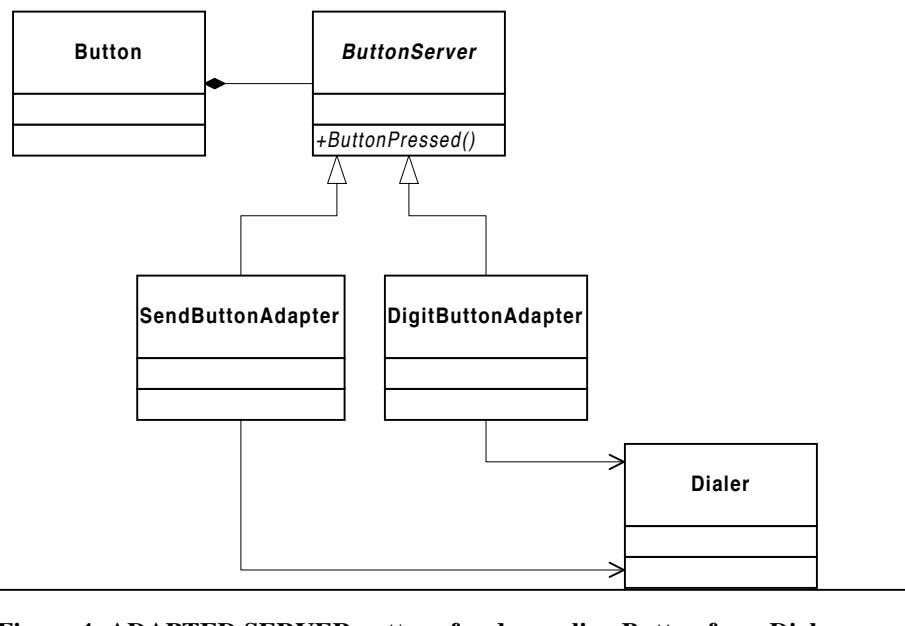
You might feel uncomfortable with this static model because it does not seem to reflect the real world as well as the first. After all, we have lost the notion of the telephone containing the buttons and the display, etc. But that notion was based upon the *physical* components of the telephone, not upon its behavior. Indeed the new static model is based upon the real world behavior of the telephone rather than upon its real world physical makeup.

We also lost a few classes. The **Telephone** and **Microphone** classes played no part in the dynamic model, and so have been removed. It may be that some other dynamic scenario will require them. If that happens, then we will put them back.

This points out the fact that many dynamic models usually accompany a single static model. Each dynamic model explores a different variation of a use case, scenario, or requirement. The links between the objects in those dynamic models imply a set of associations that must be present in a static model. Thus, dynamic models tend to vastly outnumber static models.

## Scrutinizing the static model

Our static model has a few problems. For example, why should a class named **Button** know anything about a class named **Dialer**? Shouldn't the **Button** class be reusable in programs that don't have **Dialers**?



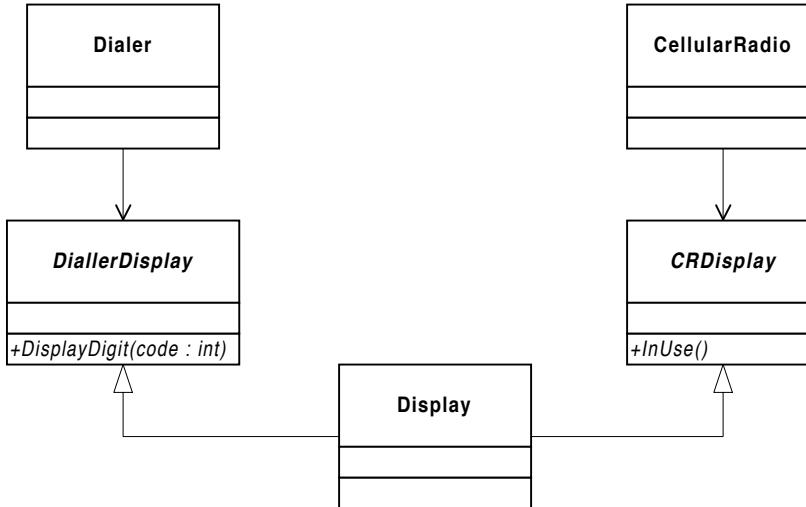
**Figure 4: ADAPTED SERVER pattern for decoupling Button from Dialer**

We can solve this problem by employing the ADAPTED SERVER pattern as shown in Figure 4. Now the **Button** class is completely reusable. Any other class that needs to detect a button press simply derives from **ButtonServer** and implements the pure virtual **ButtonPressed** function. If, like **Dialer**, the class must detect many different **Button** objects, ADAPTERS can be used to catch the **ButtonPressed** messages and translate them.

Another problem with the static model in Figure 3 is the high coupling of the **Display** class. This class will be the target of an association from many different clients. Those of you who recall my column entitled “The Interface Segregation Principle” (ISP) [*C++ Report*, Aug, 1996. *This document is also available in the ‘publications’ section of <http://www.oma.com>*] will understand that an unwarranted dependency exists between the **CellularRadio** class and the **Dialer** class. If one of the methods of **Display** needs to

be altered because of the needs of `Dialer`, then `CellularRadio` will be affected; at very least, by an unwarranted recompile.

To solve this problem, we can segregate the interfaces of the `Display` class as shown in Figure 5.



**Figure 5: Interface Segregation of the Display**

## Iterating the dynamic model

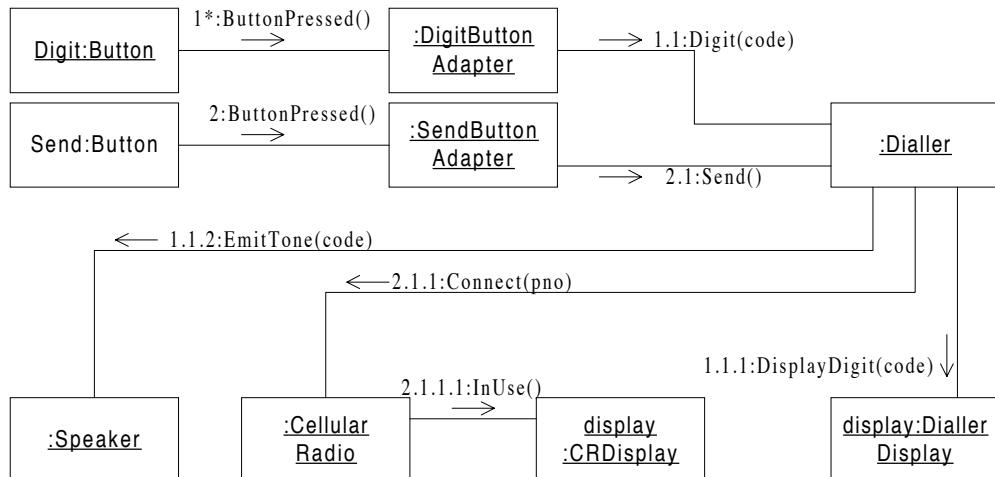
Clearly these perturbation will have an effect upon the dynamic model. Thus, it will have to be changed as shown in Figure 6. This model shows how the adapters translate the `ButtonPressed` messages into something that the `Dialer` can understand. It also shows the segregation of the display interfaces. Note that the object named `display` appears twice, but with different class names. This indicates that `display` is derived from more than one class. Thus, the class name of the object tells the reader which interface the sender is depending upon.

## Conclusion

We have completed two iterations of our static and dynamic model of a cellular phone. The first iteration was simply a guess. In the case of the first static model, the guess was pretty bad. However, after the second iteration we had resolved the disparity between the two models and had begun to explore more subtle design issues. In a real project, this iteration would continue until the designer was satisfied that both models were appropriately tuned.

Static models are necessary but insufficient for complete object oriented designs. A static model that is produced without the benefit of dynamic analysis is bound to be incorrect. The appropriate static relationships are a result of the dynamic needs of the application. UML Collaboration diagrams are a good way to depict dynamic models and compare them to the static models that must support them.

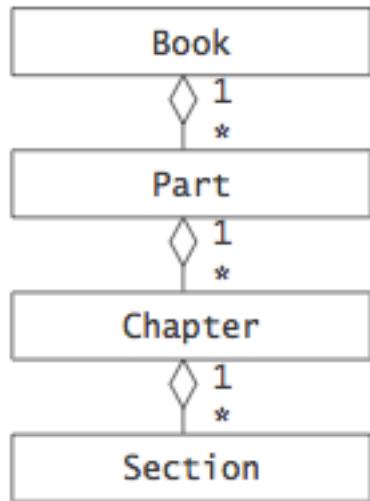
In future columns, we will continue to explore the wiles of UML. Among other things we will discuss UML's rich notation for finite state machines. We will explore how race conditions in concurrent systems can be detected with sequence diagrams. And we will demonstrate the facilities within UML that allow it to be extended.



**Figure 6: Iterated Dynamic Model**

## Week 5 – 01 Exercise Answers.

1. Draw a class diagram representing a book defined by the following statement: "A book is composed of a number of parts, which in turn are composed of a number of chapters. Chapters are composed of sections." Focus only on classes and relationships.



This exercise checks the student's understanding of basic aspects of class diagrams, including:

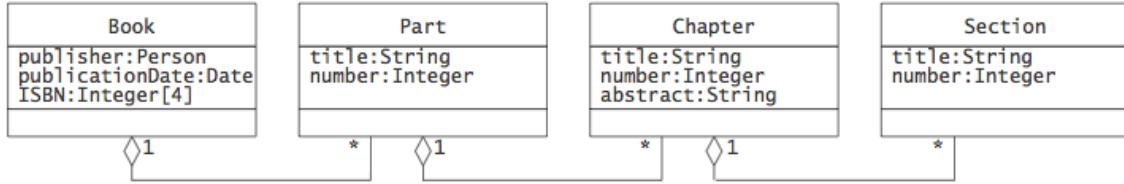
- Classes are represented with rectangles.
- The attribute and operations compartment can be omitted.
- Aggregation relationships are represented with diamonds.
- Class names start with a capital letter and are singular.

2. Add multiplicity to the class diagram you produced

See Figure above. Aggregation does not imply multiplicity, thus the 1..\* multiplicity is necessary.

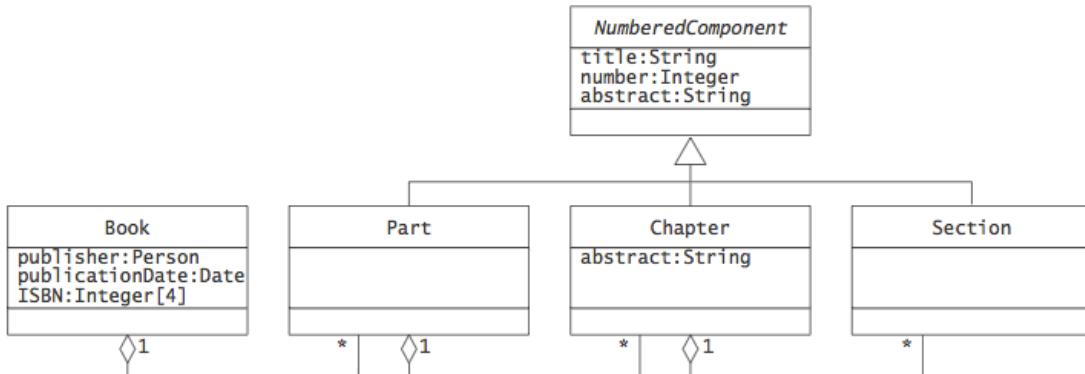
3. Extend the class diagram of Exercise 2–6 to include the following attributes:

- a book includes a publisher, publication date, and an ISBN
- a part includes a title and a number
- a chapter includes a title, a number, and an abstract
- a section includes a title and a number



This exercise checks the student's knowledge of attributes and their representation in UML.

4. Note that the Part, Chapter, and Section classes all include a title and a number attribute. Add an abstract class and a generalization relationship to factor out these two attributes into the abstract class.



This exercise checks the student's knowledge of abstract classes and inheritance.

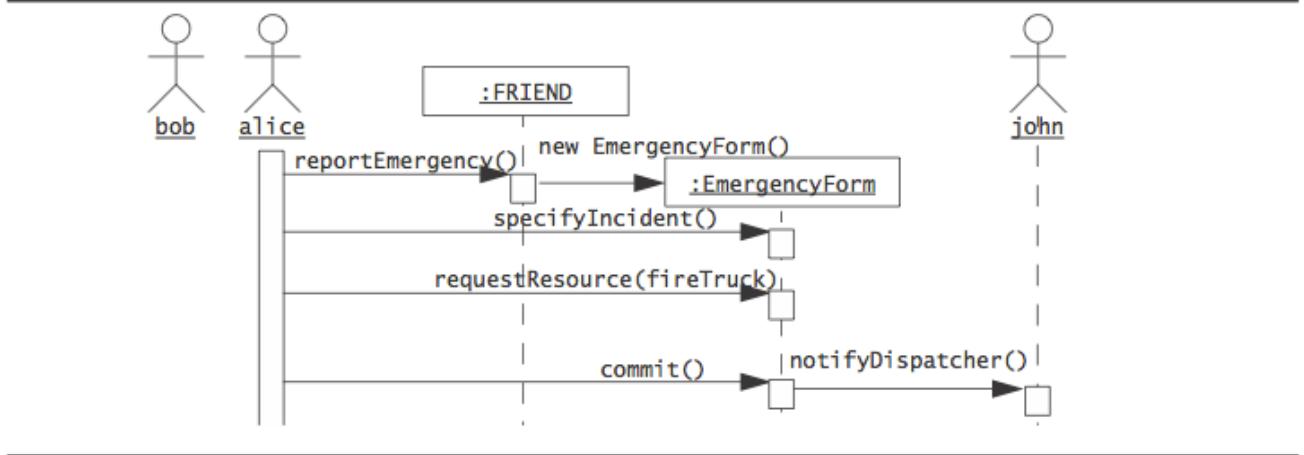
## Week 5 – Class – Answers

### 1A. Exercise: sequence diagram from a use-case scenario

- ▶ *Draw a sequence diagram for the following warehouseOnFire scenario. Include the objects bob, alice, john, FRIEND, and instances of other classes you may need. Draw only the first five message sends.*
- ▶ Scenario name: warehouseOnFire
- ▶ Participating actor instances: bob, alice:FieldOfficer, john:Dispatcher
- ▶ Flow of events:
  - ▶ 1. Bob, driving down the main street in his police car, notices smoke coming out of a warehouse. His partner, Alice, activates the 'Report Emergency' function from her FRIEND laptop.
  - ▶ 2. Alice enters the address of the building, a brief description of its location (ie. northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appears to be relatively busy. She confirms her input and waits for an acknowledgement.
  - ▶ 3. John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and send their estimated arrival time (ETA) to Alice.
  - ▶ 4. Alice receives the acknowledgement and the ETA

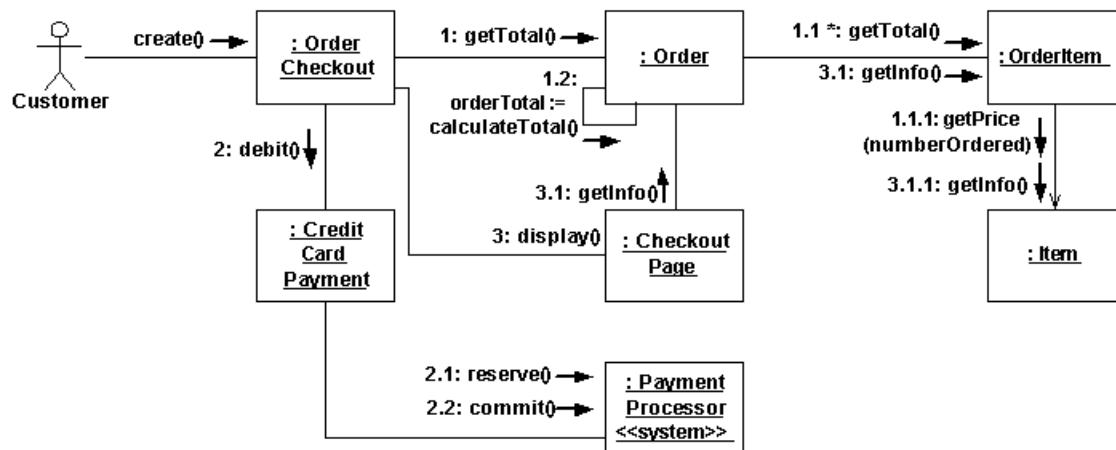
Answer:

This exercise checks the student's knowledge of sequence diagrams when using instances. This exercise can have several correct answers. In addition to the UML rules on sequence diagrams, all correct sequence diagrams for this exercise should include one or more actors on the left of the diagram who initiate the scenario, one or more objects in the center of the diagram which represent the system, and a dispatcher actor on the right of the diagram who is notified of the emergency. All actors and objects should be instances. The following figure depicts a possible answer.

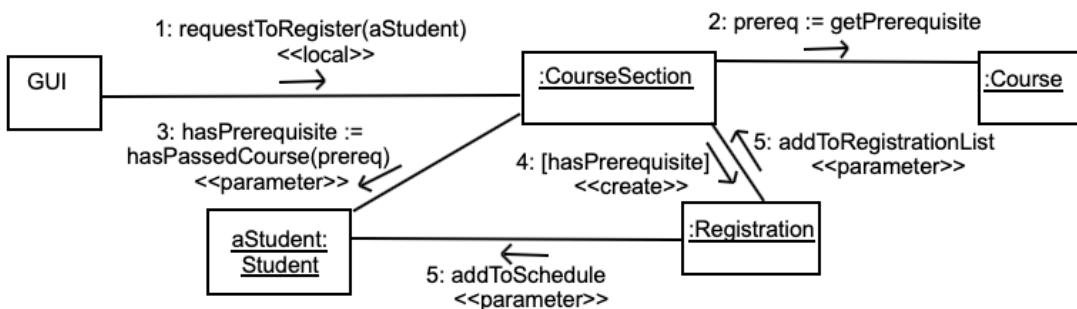


### 1B. Convert to a collaboration diagram

example collaboration diagram:

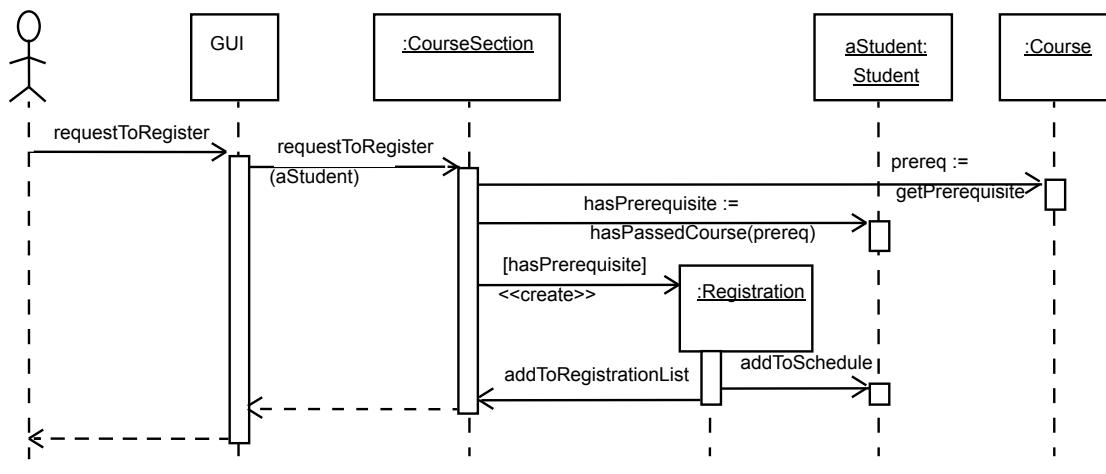


- 3. Draw a sequence diagram that models as much as possible (but not more than that!) of the contents that is being modelled by the following diagram:



## OOD modelling notation: Sequence diagrams

- ▶ Essentially: Same contents as sequence diagram, different representation



### 05c Q2 answer

```
public class CourseSection {
 Course myCourse;
 String prereq;
 Boolean hasPrerequisite;
 Registration myRegistration;

 void requestToRegister(Student aStudent) {
 prereq = myCourse.getPrerequisite;
 hasPrerequisite = aStudent.hasPassedCourse(prereq);
 if (hasPrerequisite) myRegistration = new Registration(this, aStudent);

 }
}
```

```
void addtoRegistrationlist(Registration r) {

}
}
```

---

```
public class Student{
 Boolean hasPassedCourse(String prereq) {

 }
 void addtoSchedule(Registration r) {

 }
}
```

---

```
public class Registration {
 publicRegistration(CourseSection c, Student s) {
 c.addtoRegistrationlist(this);
 s.addtoSchedule(this);

 }
}
```

---

# Collaboration Diagrams

- Class exercise



# Exercise 1A: sequence diagram from a use-case scenario

---

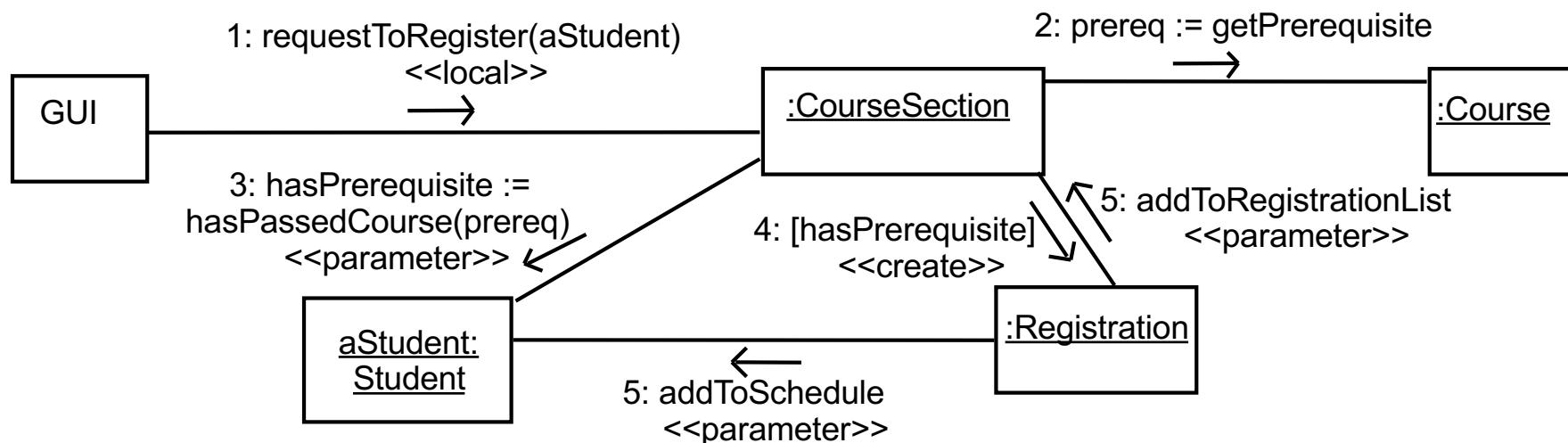
- ▶ Draw a sequence diagram for the following warehouseOnFire scenario. Include the objects bob, alice, john, FRIEND, and instances of other classes you may need. Draw only the first five message sends.
- ▶ Scenario name: warehouseOnFire
- ▶ Participating actor instances: bob, alice:FieldOfficer, john:Dispatcher
- ▶ Flow of events:
  - ▶ 1. Bob, driving down the main street in his police car, notices smoke coming out of a warehouse. His partner, Alice, activates the 'Report Emergency' function from her FRIEND laptop.
  - ▶ 2. Alice enters the address of the building, a brief description of its location (ie. northwest corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appears to be relatively busy. She confirms her input and waits for an acknowledgement.
  - ▶ 3. John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and send their estimated arrival time (ETA) to Alice.
- ▶ 2▶ 4. Alice receives the acknowledgement and the ETA

# Exercise 1B: Convert your sequence diagram into a collaboration diagram

---

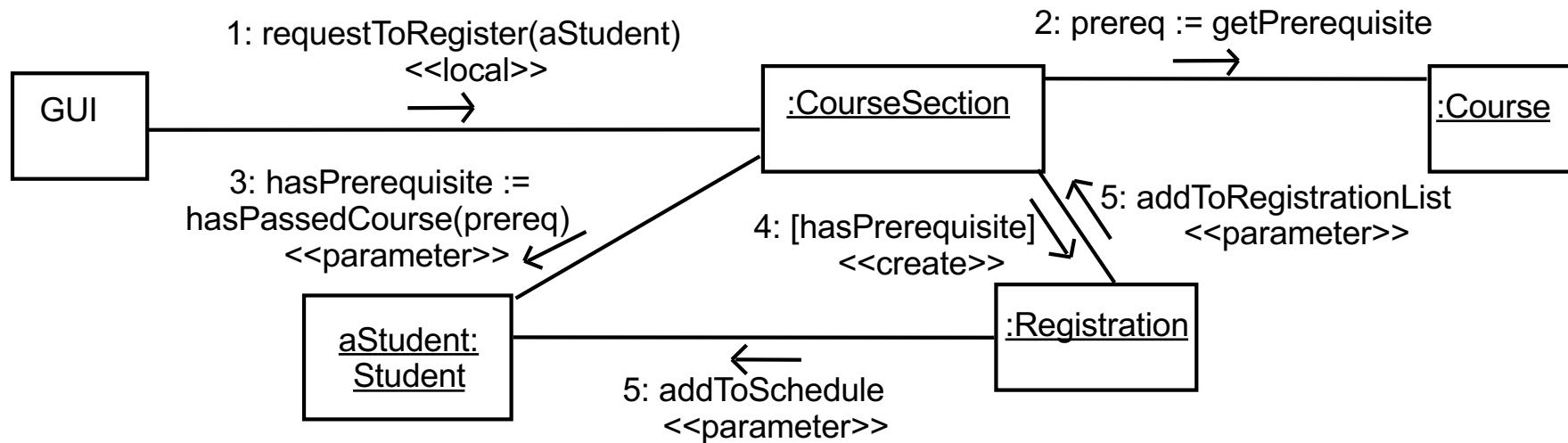
# Exercise 2: code vs. interaction diagram

- ▶ Write a Java implementation that is modelled by the following diagram:



# Exercise 3: sequence vs. interaction diagram

- ▶ Draw a sequence diagram that models as much as possible (but not more than that!) of the contents that is being modelled by the following diagram:

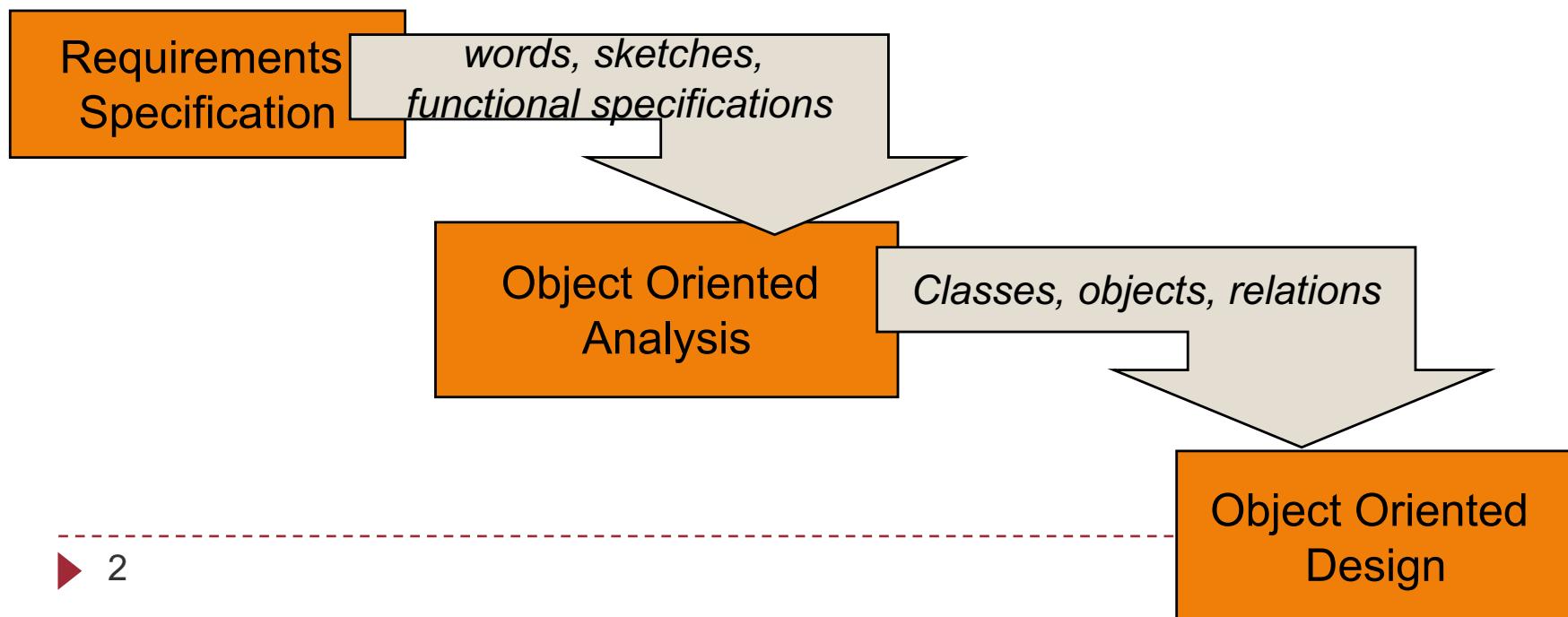


# Object-Oriented Analysis techniques

- OO Analysis and Design - moving into design
- Class Diagrams (again)
- Royal Service Station case study
- CRC Diagrams
- Template Classes
- Exercises

# Object-oriented analysis (OOA)

- ▶ Goal: to fully specify the problem and the application domain **without introducing a bias to any particular implementation**. [Rumbaugh et. al 91]
- ▶ OOA is a transition from specifications to design



# Vocabulary and notations

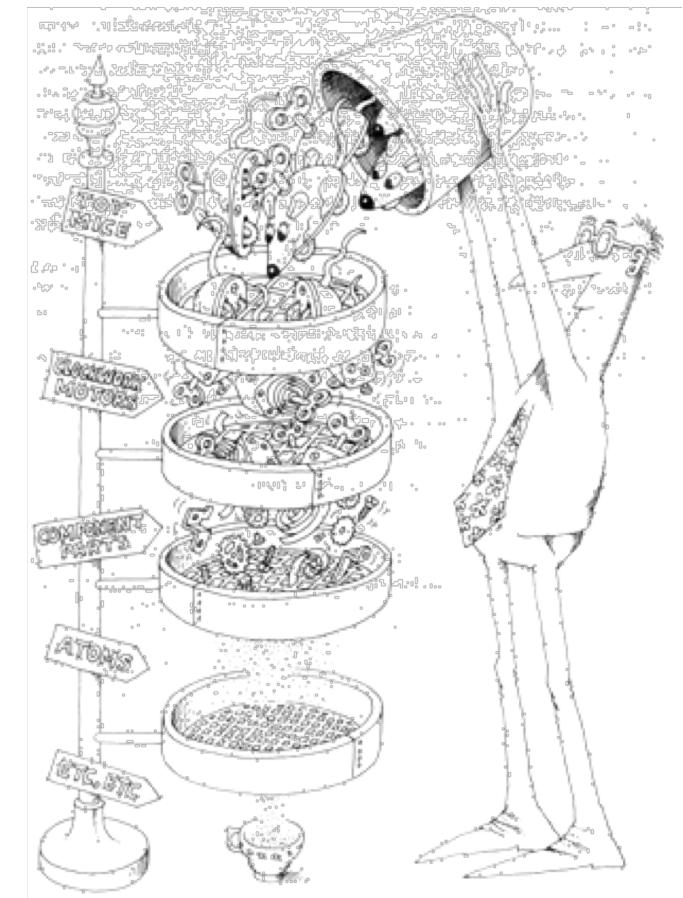
---

- ▶ Vocabulary of OOA: Similar to object-oriented design:
  - ▶ Classes and objects
  - ▶ Generalization/specialization ('inheritance')
- ▶ Notations:
  - ▶ Class diagrams
  - ▶ Sequence diagrams
  - ▶ Interaction diagrams
  - ▶ Type diagrams

# Analysis vs. design

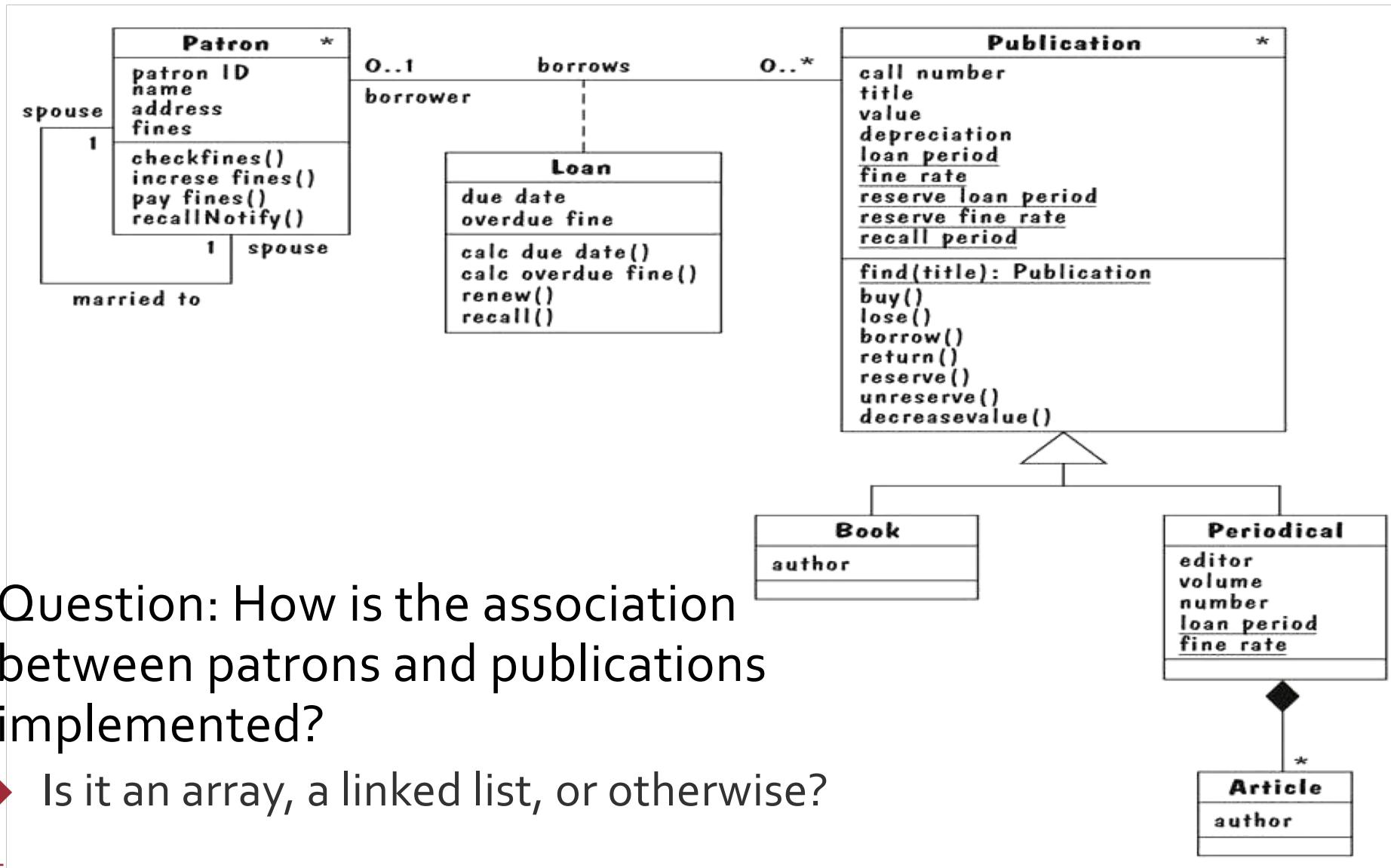
- ▶ Software analysis is more **abstract** than software design in the following senses:

|                   | <i>Analysis</i> | <i>Design</i>  |
|-------------------|-----------------|----------------|
| <b>purpose</b>    | what            | how            |
| <b>activity</b>   | analysis        | synthesis      |
| <b>vocabulary</b> | problem         | solution       |
| <b>subject</b>    | requirements    | implementation |



## More on Class Diagrams

# Class diagrams example: library



- ▶ Question: How is the association between patrons and publications implemented?
  - ▶ Is it an array, a linked list, or otherwise?

# Possible ways the association between Patrons and Publications could be implemented

- ▶ Using either a **Set** or a **List**.
- ▶ If the Loan class contains information about the way a Patron borrows a publication (for example, the due date), you could implement it like this:

```

class Patron
{
 Set<Loan> loans;
}

class Loan
{
 Patron patron;
 Publication publication;
 Date dueDate;
}

class Publication
{
 Set<Loan> loans;
}

```

## ▶ Alternatively: Using a **Map**

```

public class Patron
{
 private Map<Publication, Loan> loansByPublication;
}

public class Loan
{
 //Loan's properties
 Date dueDate;
}

public class Publication
{
 private Map<Patron, Loan> loansByPatron;
}

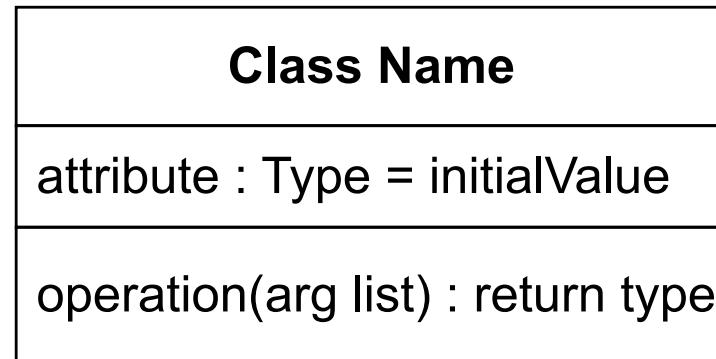
```

# Requirement modelling technique: class diagrams

---

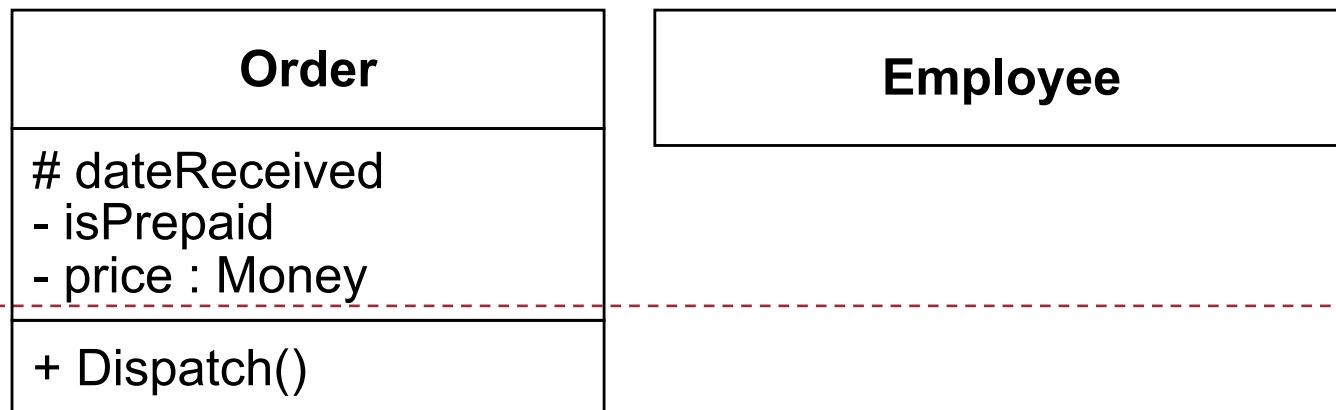
## ► 3 parts to a class

- ▶ name
- ▶ attributes
- ▶ operations



## ► Access Specifications

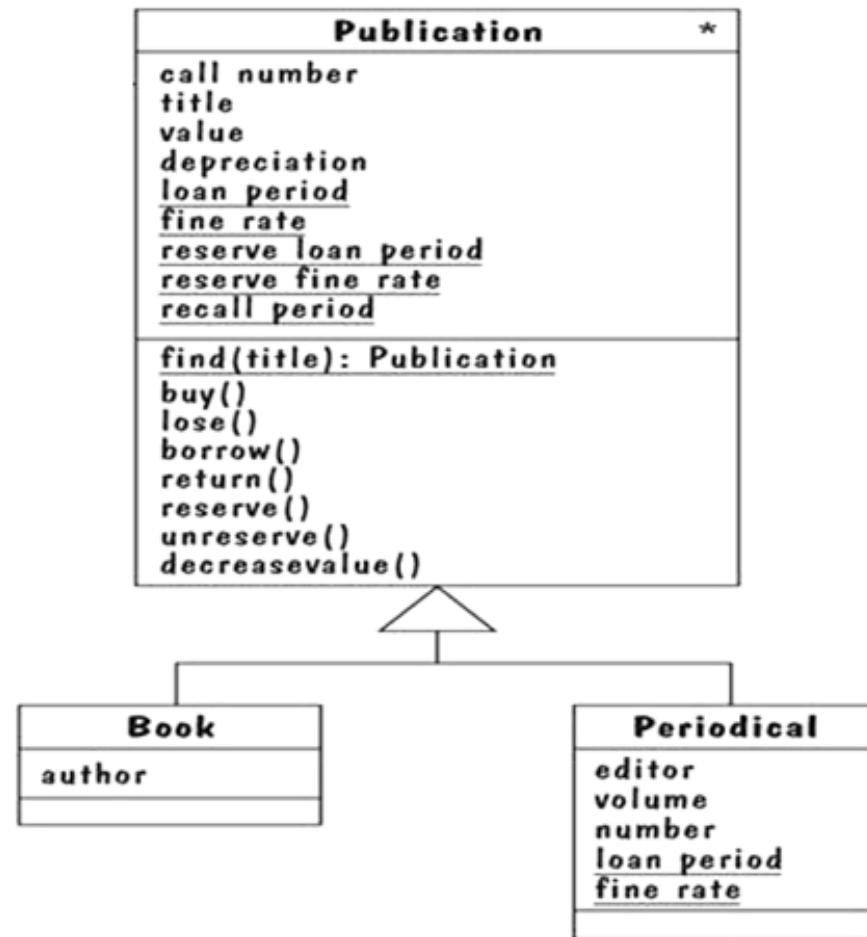
- + Public (Any class that can see the container can also see and use the classes)
- # Protected (Only classes in the same container or a descendent of the container can see and use the classes.)
- Private (only classes in the same container can see and use the classes)
- ~ Package (Only classes within the same package as the container can see and use the classes)



# Class diagrams: inheritance

---

- ▶ Inheritance captures the “is-kind-of” relation
  - ▶ Think of subclasses as subcategories
- ▶ Use it to define classes incrementally



# Adding Generalization Structure

---

- ▶ Add generalization structures when:
  - ▶ Two classes are similar in most details, but differ in some respects
  - ▶ May differ:
    - ▶ In behaviour (operations or methods)
    - ▶ In data (attributes)
    - ▶ In associations with other classes



# Adding Structure

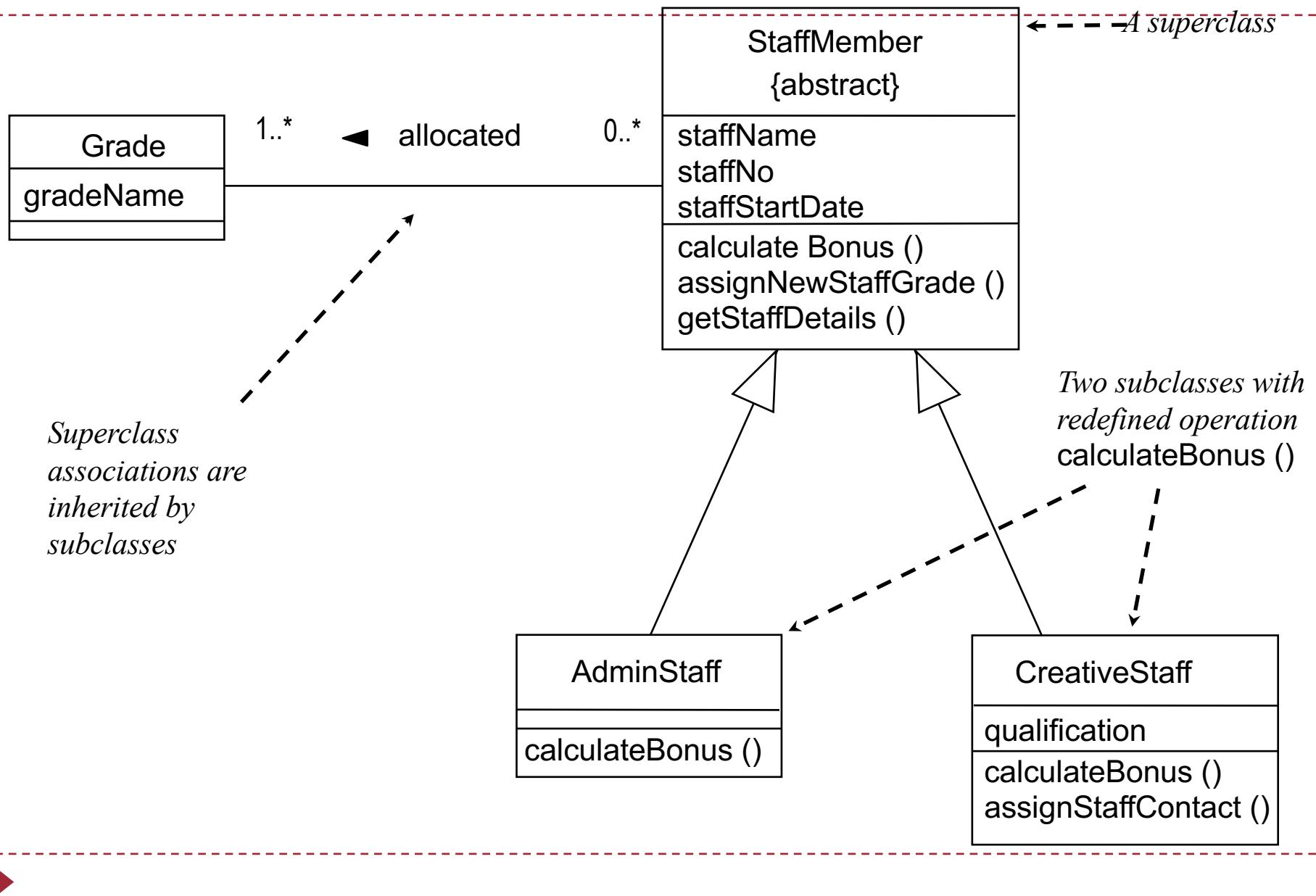
---

- ▶ Two types of staff:

|          |                                                                                                                                   |
|----------|-----------------------------------------------------------------------------------------------------------------------------------|
| Creative | <p>Have qualifications recorded</p> <p>Can be client contact for campaign</p> <p>Bonus based on campaigns they have worked on</p> |
| Admin    | <p>Qualifications are not recorded</p> <p>Not associated with campaigns</p> <p>Bonus not based on campaign profits</p>            |

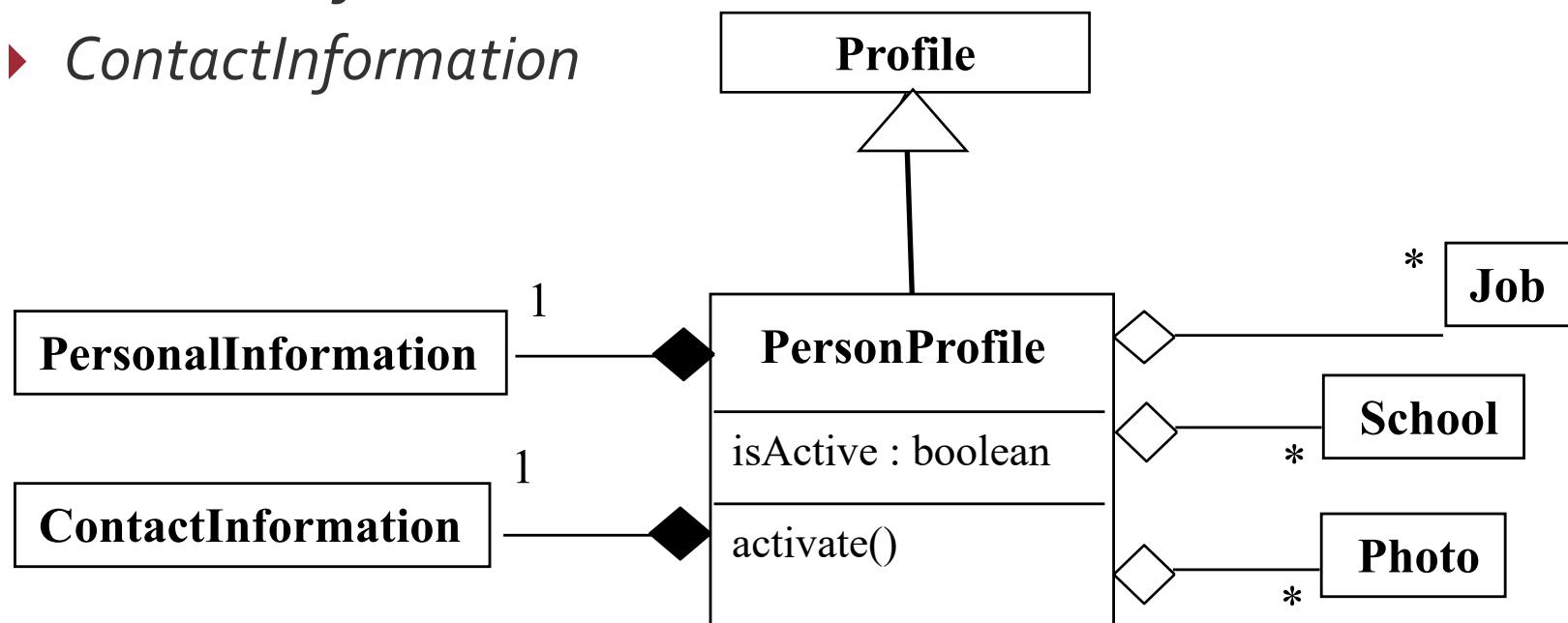


# Adding Structure:



# Class diagrams: Aggregation Vs. Composition

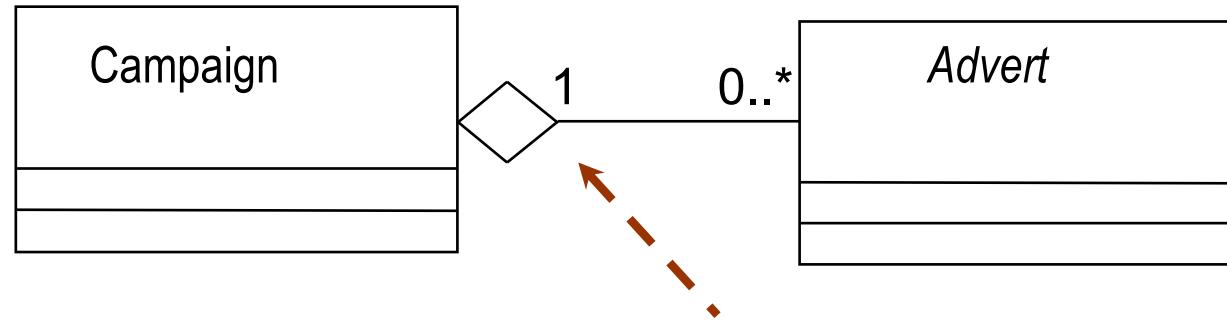
- ▶ Aggregation: Collection of elements
  - ▶ Photos, jobs, schools
- ▶ Composition: Whole-part relation
  - ▶ *PersonalInformation*
  - ▶ *ContactInformation*



# Aggregation and Composition

---

- ▶ Special types of association, both sometimes called whole-part
- ▶ A campaign is made up of adverts:



*Unfilled diamond  
signifies aggregation*



## Aggregation and Composition

---

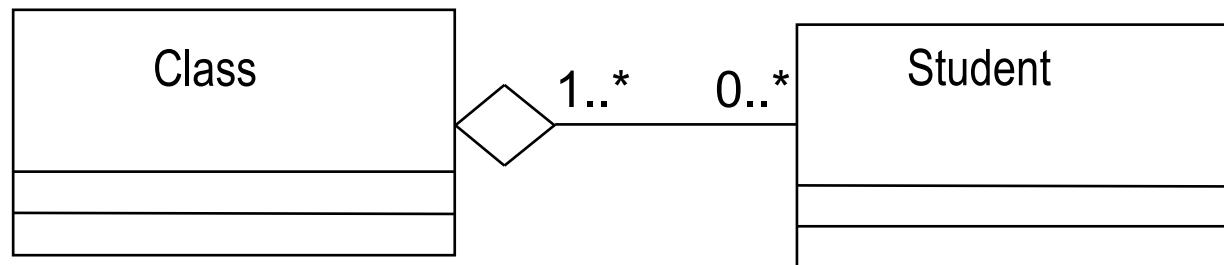
- ▶ Aggregation is essentially any whole-part relationship
- ▶ Semantics can be very imprecise
- ▶ Composition is ‘stronger’ :
  - ▶ Each part may belong to only one whole at a time
  - ▶ When the whole is destroyed, so are all its parts



# Aggregation and Composition

---

- ▶ An everyday example:



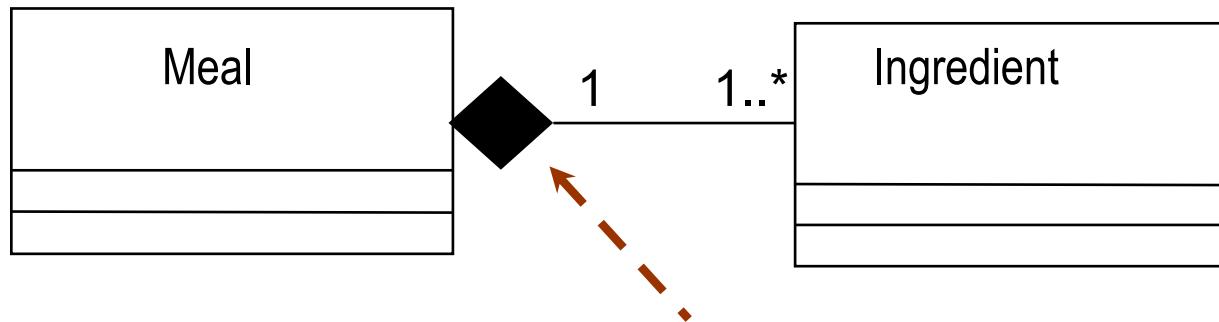
- ▶ Clearly not composition:
  - ▶ Students could be in several classes
  - ▶ If class is cancelled, students are not destroyed!



# Aggregation and Composition

---

- ▶ Another everyday example:



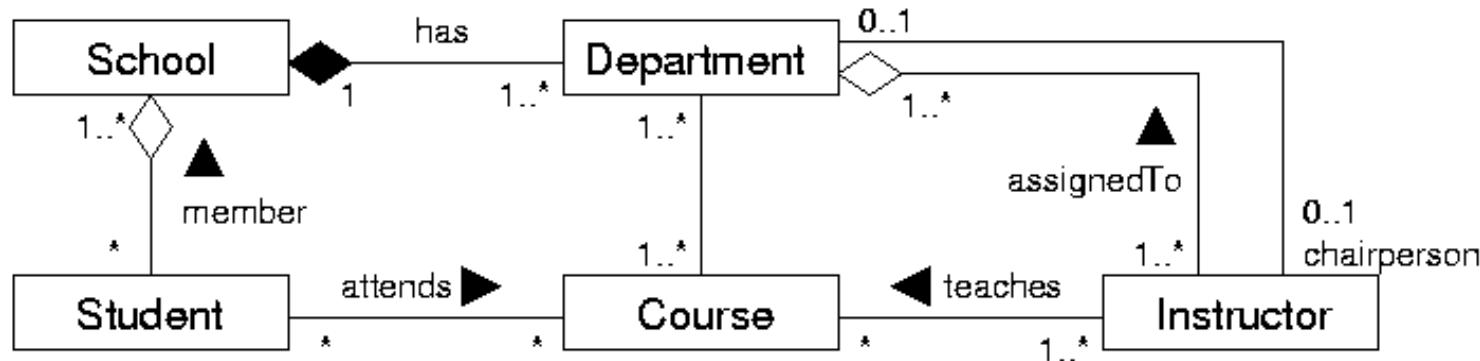
*Filled diamond signifies composition*

- This is (probably) composition:
  - Ingredient is in only one meal at a time
  - If you drop your dinner on the floor, you probably lose the ingredients too



# Class diagrams: cardinality (‘multiplicity’)

---



- ▶ Note that associations can—
  - ▶ be directed
  - ▶ be labelled
  - ▶ have a cardinality on each end



# Bad class diagrams

► What's wrong with this diagram?

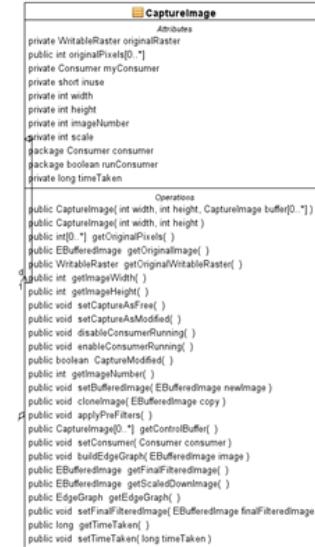
Entities Class diagram

This package is concerned with using the EdgeCollector algorithm on BufferedImage and storing the Edges in an EdgeGraph

EdgePack: Contains rows and columns of Edges

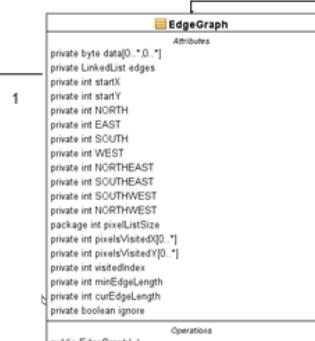
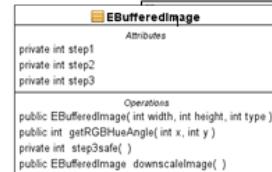


CaptureImage: Contains locks for mutual exclusion, does operations on BufferedImage

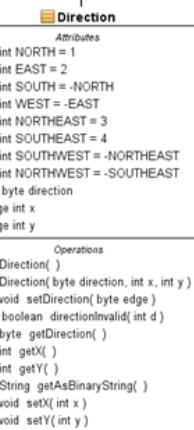


1  
Contains the Edges found in the  
1 BufferedImage

Holds the RGB image from the  
Digital Input Device



1  
Contains info about a single  
pixel and direction  
information

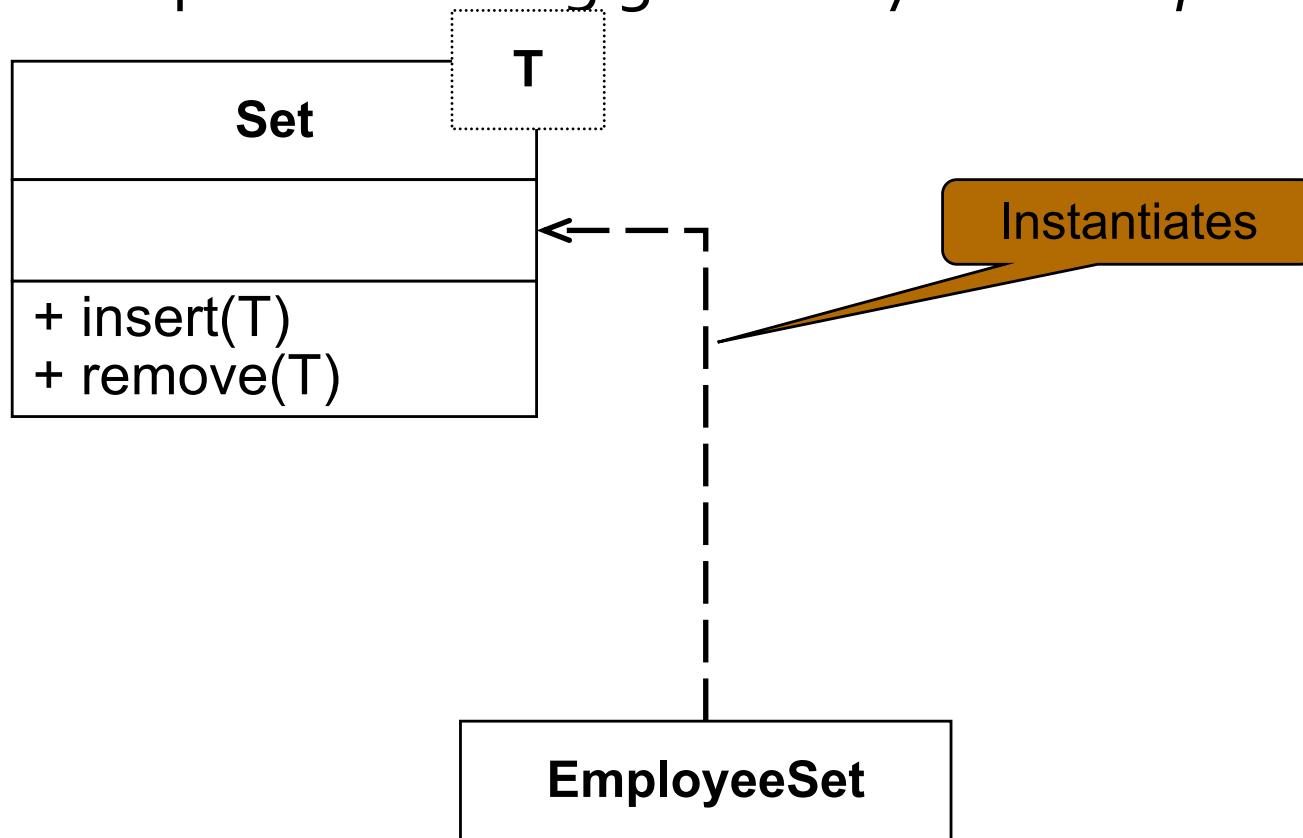


Holds information about an  
Edge, each Direction and how  
many times it is recognised by  
the AISystem



# Class diagrams: modelling genericity

- ▶ Class diagrams are commonly used to specify detailed design
- ▶ For example: modelling genericity and *templates*



# Template Classes

---

- ▶ The Template Parameter specifies the formal parameters which will be substituted by the actual parameters in a binding. A template class defines a family of classes. You may create class with template parameter to form the template class.
- ▶ Benefits:
  - ▶ A template class allows for its functionality to be reused by any bound class.
  - ▶ Templates are an efficient way of allowing one piece of code to operate on many different types. Easy to handle requirement changes.
  - ▶ By creating templates for classes, you benefit from having clear separation of implementation. If any change request has to be done, it can be easily handled by modifying the template, and the changes will be reflected in the implementation for all instances.

# Example

---

- ▶ Suppose that you have an application where you need a list of objects of class Customer.
- ▶ In order to use a template, we must first create a class based on the template, a process commonly referred to as instantiation, and then use the newly created list class for creating list objects.

We might declare a List template as follows:

```
public class List
{
 public List() {}
 public void add (Element e) {}
 public void remove(Element e){}
 public void remove (int index) {}
}
```

```
// instantiate
class CustomerList = new List<Customer>;

// create new object for CustomerList
CustomerList custList = new CustomerList();

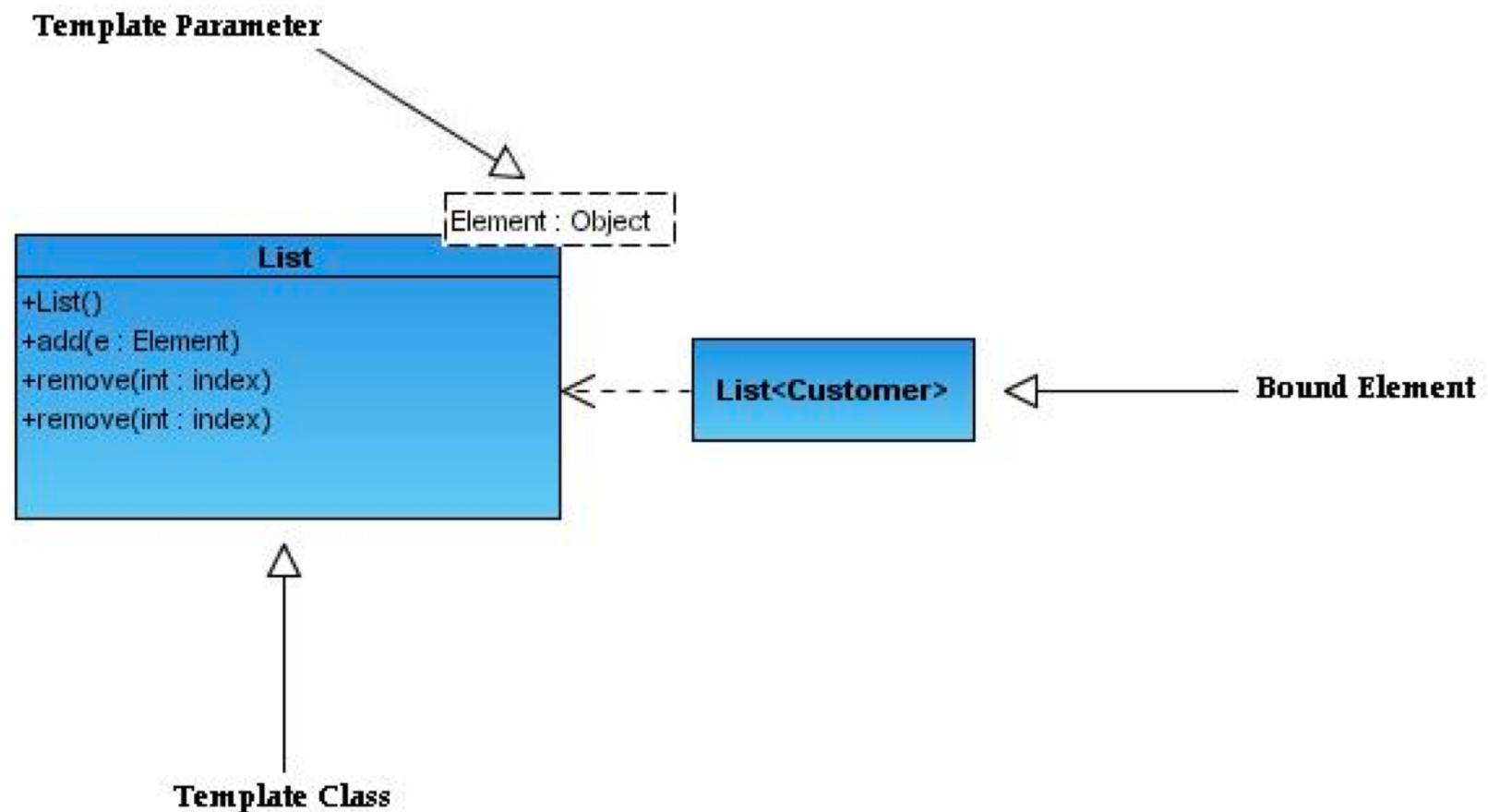
// create a new customer object
Customer c = new Customer();

// put the customer on the list
custList.add(c)
```

The above demonstrates how to use the template.

# Example: Template in UML

Below diagram shows the binding between the model class **Customer** and the **List** template.



# OOA Case Study: Royal Service Station

From text to class diagrams

# Royal Service Station

---

## Requirements

- Royal Service station provides three types of services
- The system must track bills, the product and services
- System to control inventory
- The system to track credit history, and payments overdue
- The system applies only to regular repeat customer
- The system must handle the data requirements for interfacing with other system
- The system must record tax and related information
- The station must be able to review tax record upon demand
- The system will send periodic message to customers
- Customer can rent parking space in the station parking lot
- The system maintain a repository of account information
- The station manager must be able to review accounting information upon demand
- The system can report an analysis of prices and discounts
- The system will automatically notify the owners of dormant accounts
- The system can not be unavailable for more than 24 hours
- The system must protect customer information from unauthorized access



# Identifying Behaviors

---

- Imperative verbs – at the beginning of sentences
- Passive verbs
- Actions
- Membership in
- Management or ownership
- Responsible for
- Services provided by an organization eg. use-cases



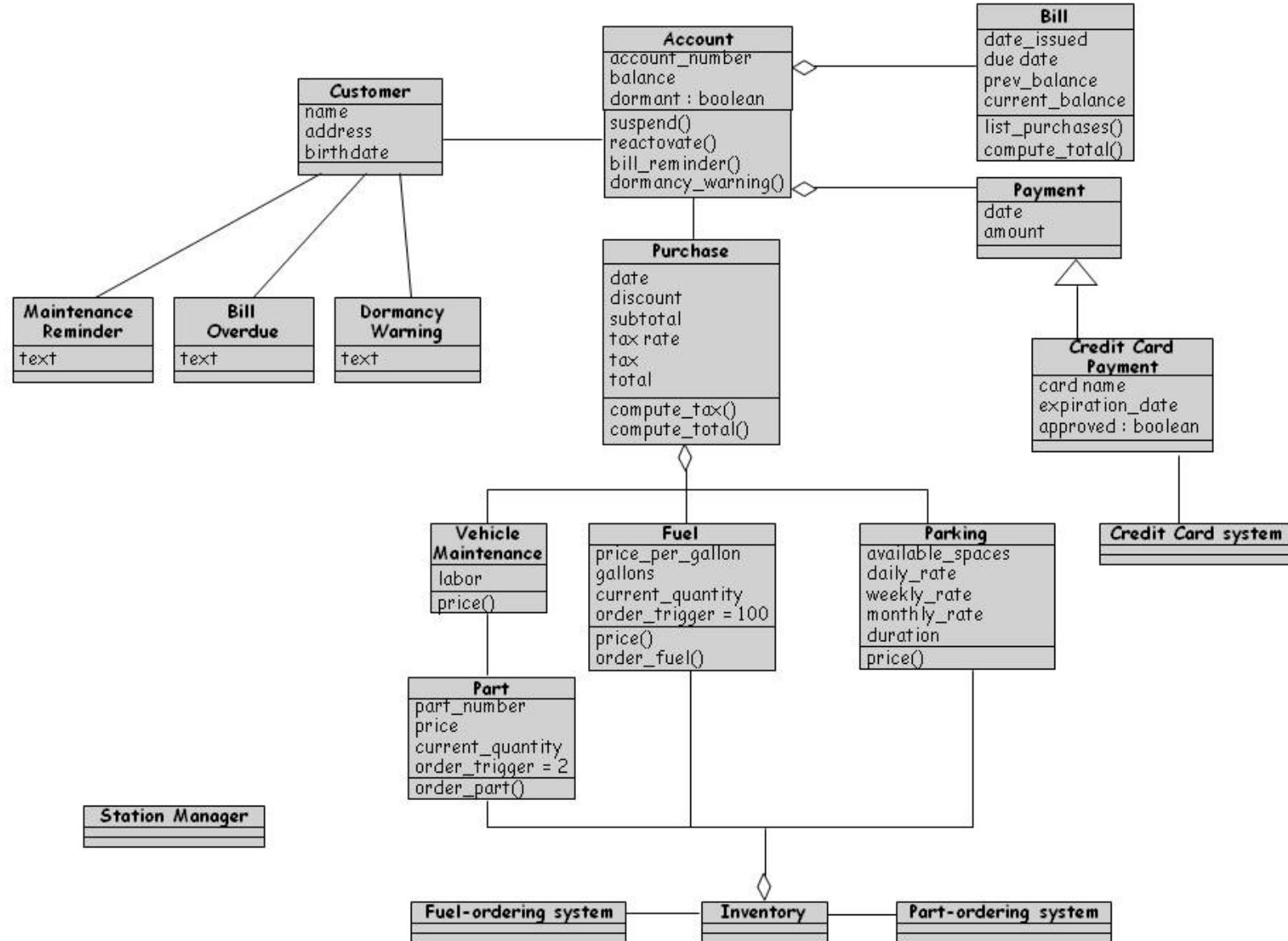
# Grouping of Attributes and Classes

---

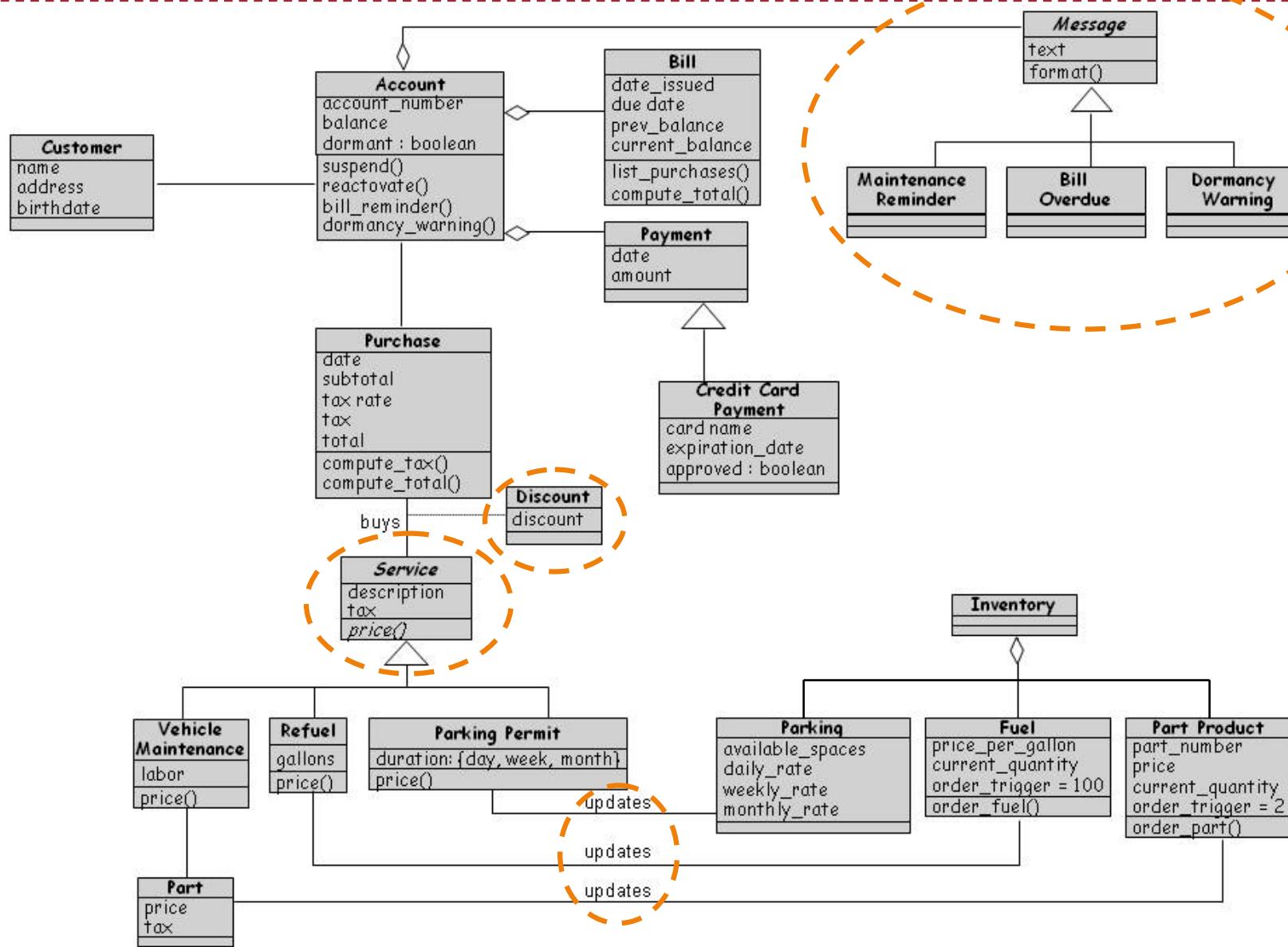
| Attributes                                                                                         | Classes                                                                                                                                                                                                                                                                         |
|----------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Personal check<br>Tax<br>Price<br>Cash<br>Credit card<br>Discounts<br>Name<br>Address<br>Birthdate | Customer<br>Maintenance<br>Services<br>Parking<br>Fuel<br>Bill<br>Purchase<br>Maintenance reminder<br>Station manager<br>Overdue bill letter<br>Dormant account warning<br>Parts<br>Accounts<br>Inventory<br>Credit card system<br>Part ordering system<br>Fuel ordering system |



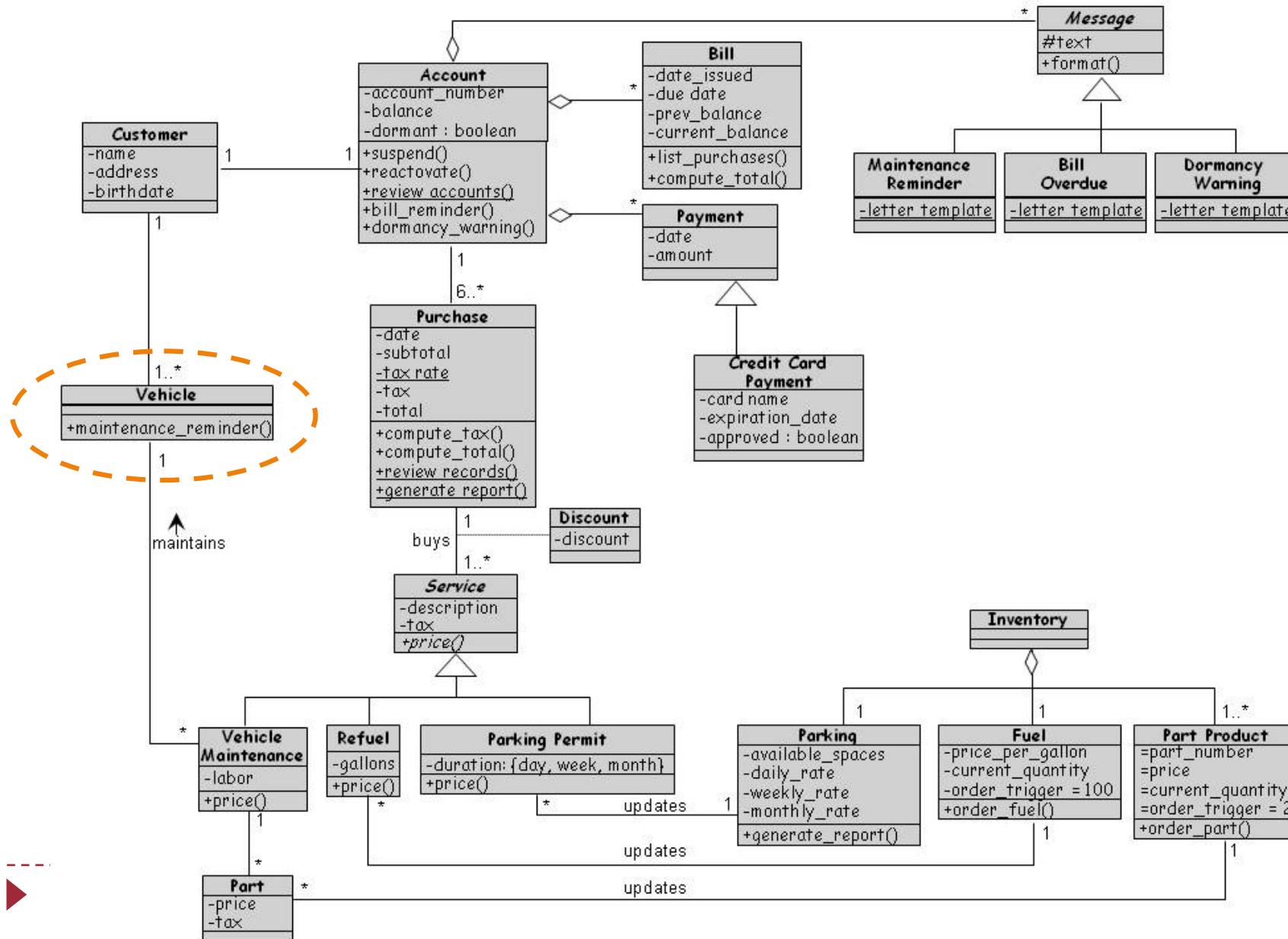
# Class diagram step 1 (based on our knowledge of the domain)



# Class diagram step 2



# Class diagram step 3



## Class / Responsibility / Collaborations

# CRC Cards

---

- ▶ Class–Responsibility–Collaboration cards help to model interaction between objects
- ▶ Used as a way of:
  - ▶ Identifying classes that participate in a scenario
  - ▶ Allocating responsibilities - both operations and attributes (*what can I do?* and *what do I know?*)
- ▶ For a given scenario (or use case):
  - ▶ Brainstorm the objects
  - ▶ Allocate to team members
  - ▶ Role play the interaction



# OOA technique: CRC cards

---

| Class Name:                                                    |                                                                                                                                  |
|----------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------|
| Responsibilities                                               | Collaborations                                                                                                                   |
| <i>Responsibilities of a class are listed in this section.</i> | <i>Collaborations with other classes are listed here, together with a brief description of the purpose of the collaboration.</i> |

# CRC Cards: Example

---

|                                     |                                                                       |
|-------------------------------------|-----------------------------------------------------------------------|
| Class Name                          | <i>Vehicle Traffic Light</i>                                          |
| Responsibilities                    | Collaborations                                                        |
| <i>Receive Open or Close signal</i> | <i>Intersection (send signals)</i><br><i>Road (ingoing, outgoing)</i> |

# From use case to CRC

---

## Example use case description

The campaign manager selects the required campaign for the client concerned and adds a new advert to the existing list of adverts for that campaign. The details of the advert are completed by the campaign manager.

- Identify classes involved
- Work through the scenario to identify the CRC responsibilities and collaborations

# CRC cards: Examples (cont.)

| Class Name                           | <i>Client</i>                              |
|--------------------------------------|--------------------------------------------|
| Responsibilities                     | Collaborations                             |
| <i>Provide client information.</i>   |                                            |
| <i>Provide list of campaigns.</i>    | <i>Campaign provides campaign details.</i> |
| Class Name                           | <i>Campaign</i>                            |
| Responsibilities                     | Collaborations                             |
| <i>Provide campaign information.</i> | <i>Advert provides advert details.</i>     |
| <i>Provide list of adverts.</i>      | <i>Advert constructs new object.</i>       |
| <i>Add a new advert.</i>             |                                            |
| Class Name                           | <i>Advert</i>                              |
| Responsibilities                     | Collaborations                             |
| <i>Provide advert details.</i>       |                                            |
| <i>Construct adverts.</i>            |                                            |



# CRC Cards

---

- ▶ Effective role play depends on an explicit strategy for distributing responsibility among classes
- ▶ For example:
  - ▶ Each role player tries to be lazy
  - ▶ Persuades other players *their* class should accept responsibility for a given task
- ▶ May use ‘Paper CASE’ to document the associations and links



# Summary

---

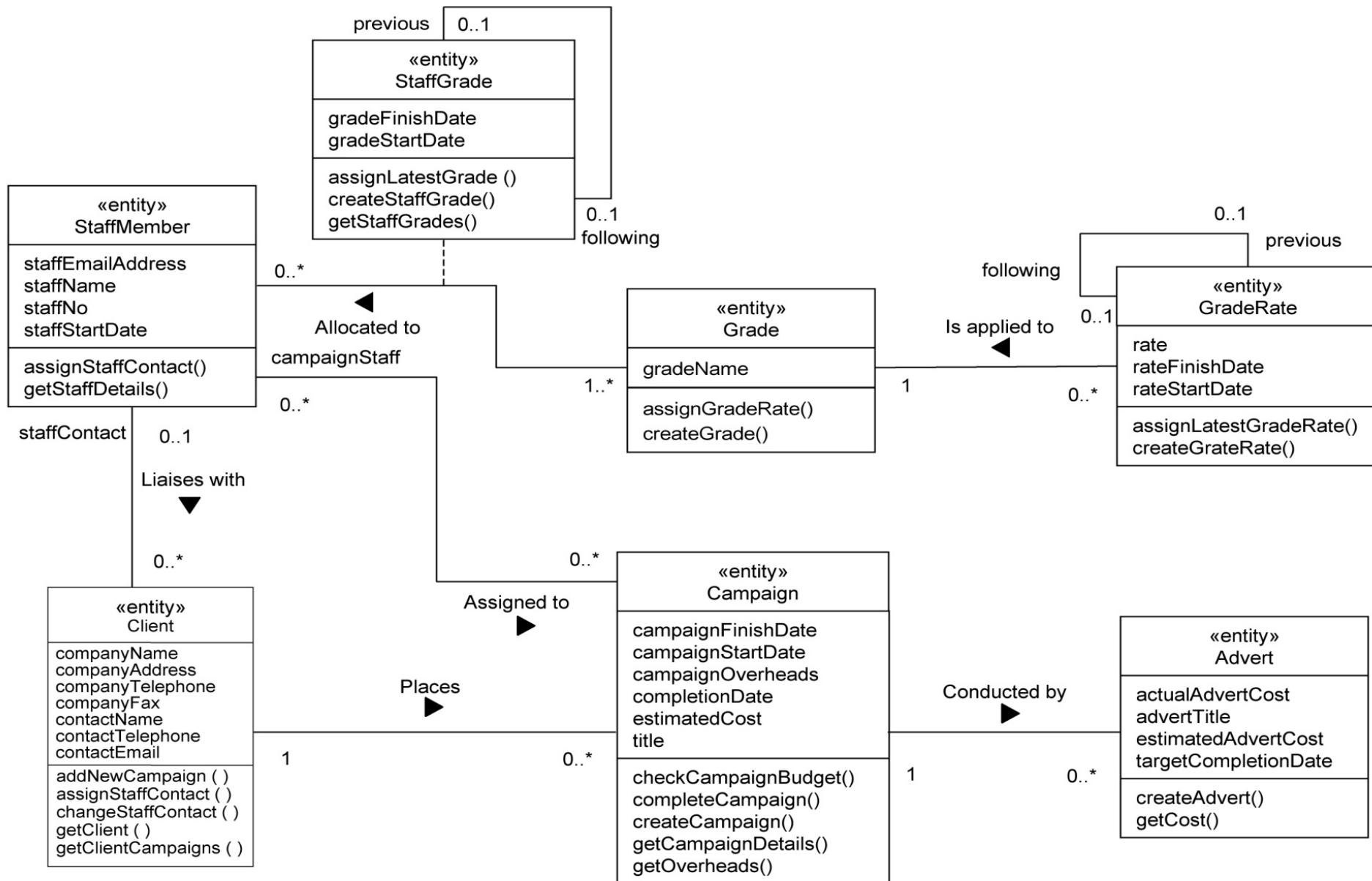
- OO Analysis and Design - moving into design
- Class Diagrams
  - Access specifications
  - Inheritance
  - Associations
  - Composition/Aggregation
  - Cardinality
- Template Classes
- Royal Service Station case study
- CRC Diagrams

## Further reading

---

- ▶ James Rumbaugh (1991). Object-oriented modeling and design. Prentice-Hall.
- ▶ Chapter 8 Bennett – refining the Analysis model
- ▶ Chapter 7 Bennett – CRC diagrams
- ▶ Pfleeger – Section 6.4, Representing OO Designs in the UML (Royal Service Station example)

# Exercise: translate relations to English



## Exercise: reconciling class and collaboration diagrams ( part 1)

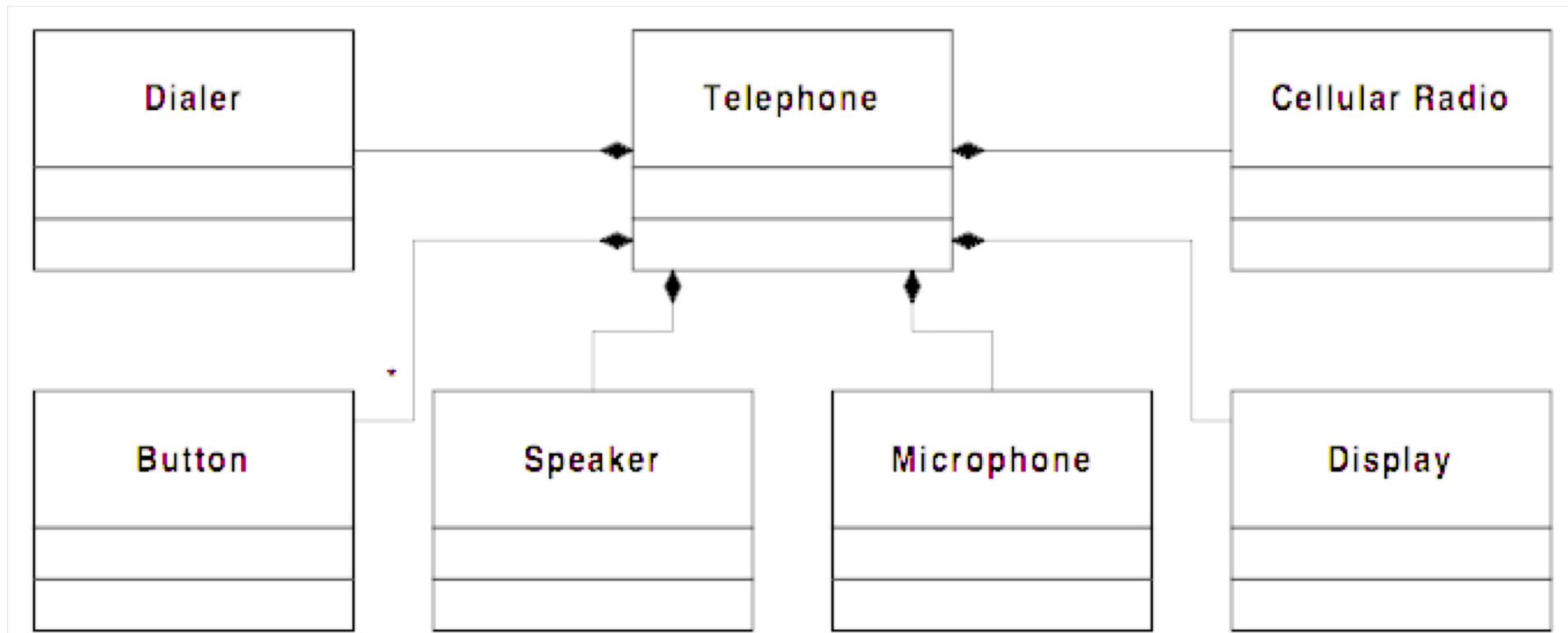
---

- ▶ Consider the software that controls a very simple cellular telephone. Such a phone has buttons for dialing digits, and a “send” button for initiating a call. It has “dialer” hardware and software that gathers the digits to be dialed and emits the appropriate tones. It has a cellular radio that deals with the connection to the cellular network. It has a microphone, a speaker, and a display.
- ▶ Draw a simple class diagram for this based around a simple Telephone class.

# Possible answer:

---

- ▶ What is wrong with this?



## Exercise: part 2

---

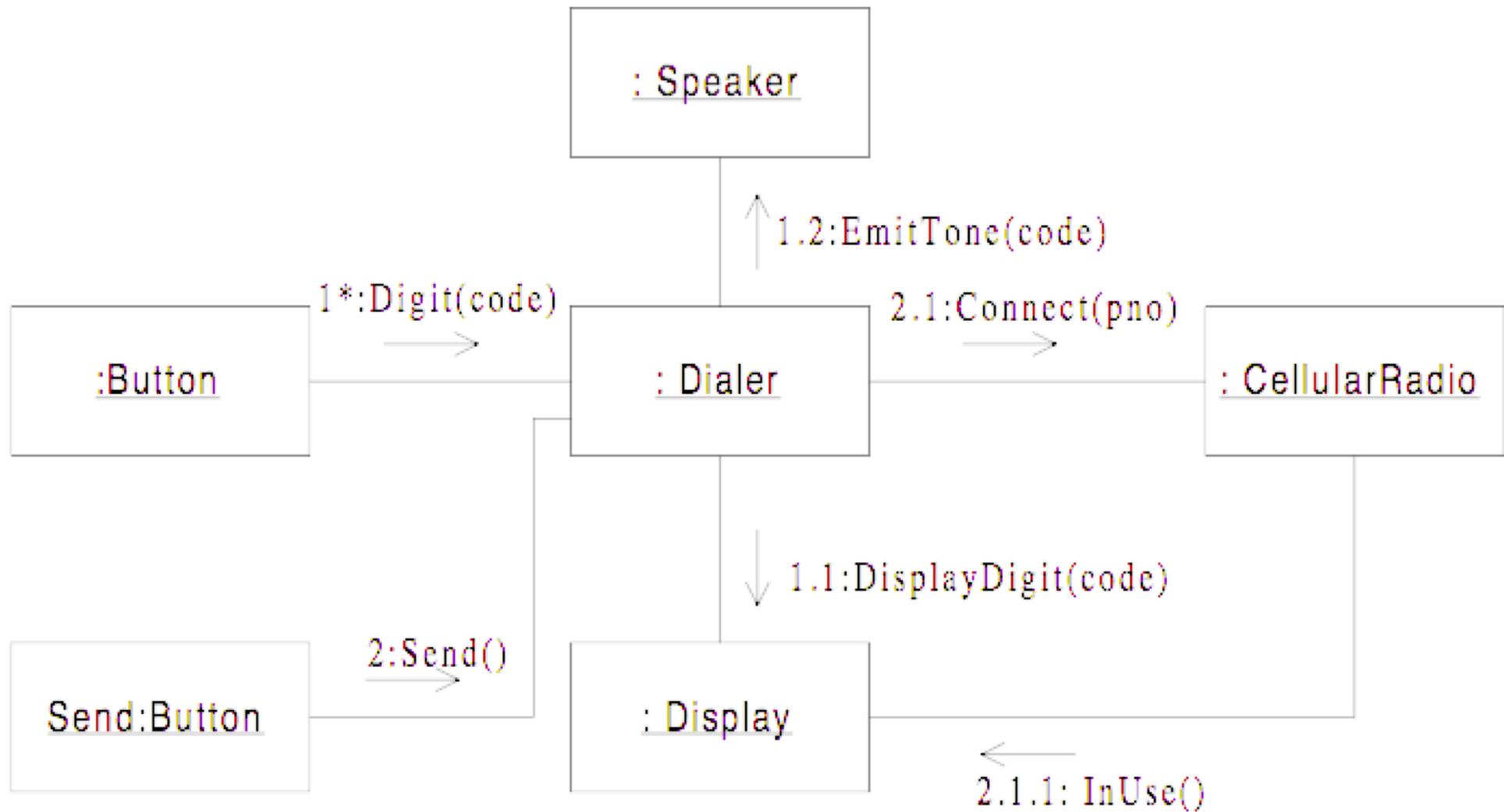
- ▶ Draw a collaboration diagram for the following use case:

Use case: Make Phone Call

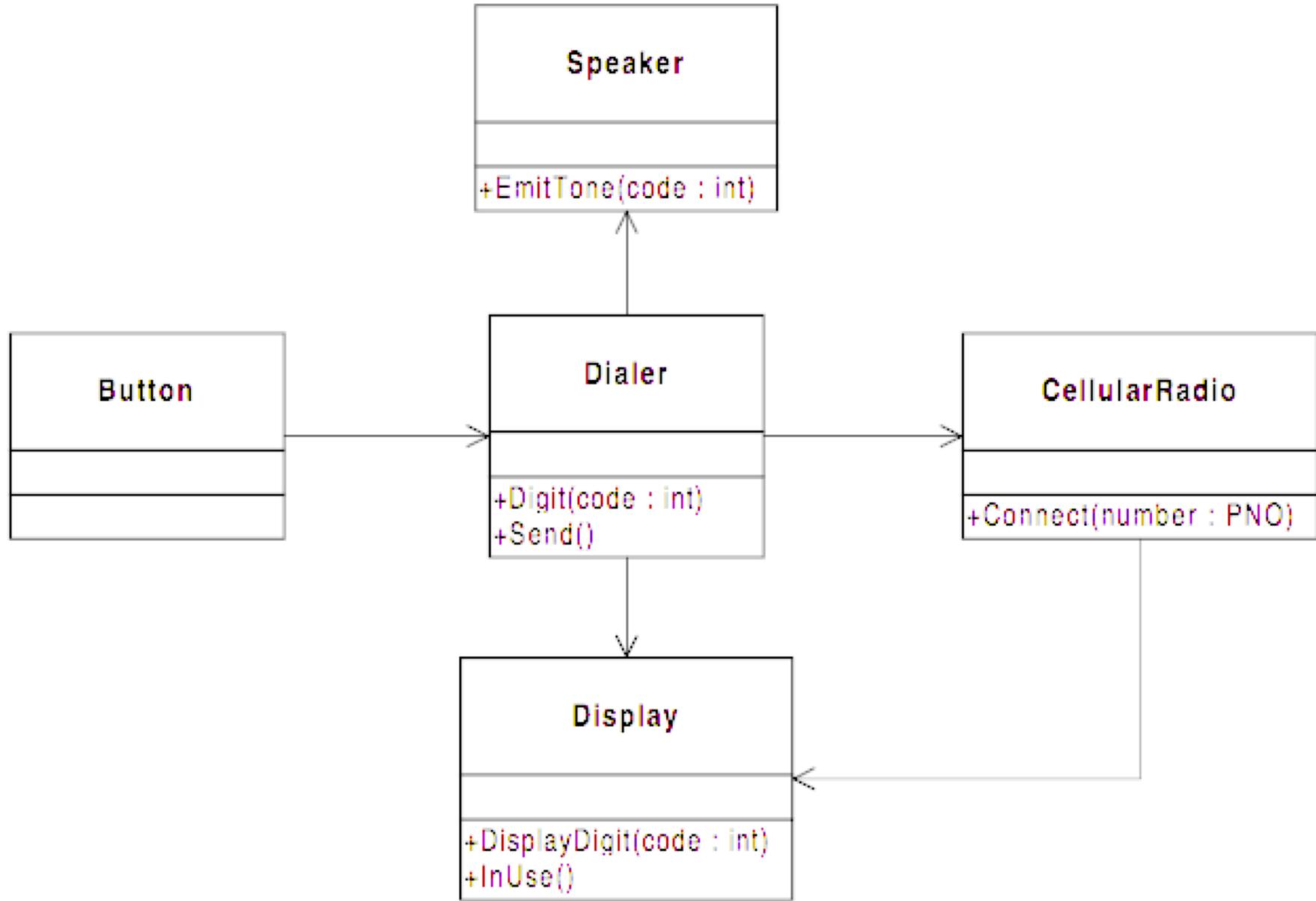
- ▶ 1. User presses the digit buttons to enter the phone number.
- ▶ 2. For each digit, the display is updated to add the digit to the phone number.
- ▶ 3. For each digit, the dialer generates the corresponding tone and emits it from the speaker.
- ▶ 4. User presses "Send"
- ▶ 5. The "in use" indicator is illuminated on the display
- ▶ 6. The cellular radio establishes a connection to the network.
- ▶ 7. The accumulated digits are sent to the network.
- ▶ 8. The connection is made to the called party.
- ▶ Does your dynamic model match to your static model?
- ▶ If not, how can you reconcile this?
- ▶ Reconcile your static model

# Possible answer

- ▶ How do we reconcile this with the class diagram?



# Outline of a solution



## Exercise: Relationships, roles and classes

---

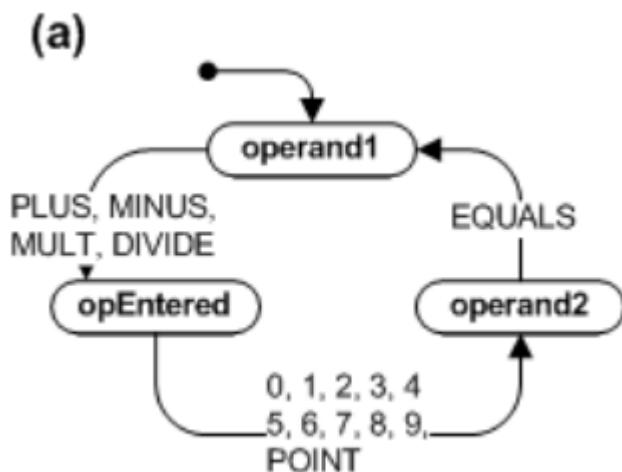
- ▶ Draw a class diagram representing the relationship between parents and children. Take into account that a person can have both a parent and a child. Annotate associations with roles and multiplicities.

## State machines exercise answers

### 1. Design a UML state machine for a simple calculator

Start by designing a simple state machine which handles the sequence of a calculation consisting of an operand, followed by an operator, followed by an operand.

Figure 3-2 shows first steps in elaborating the calculator statechart. In the very first step (panel (a)), the state machine attempts to realize the primary function of the system (the primary use case), which is to compute expressions: operand1 operator operand2 equals... The state machine starts in the "operand1" state, whose function is to ensure that the user can only enter a valid operand. This state obviously needs some internal submachine to accomplish this goal, but we ignore it for now. The criterion for transitioning out of "operand1" is entering an operator (+, -, \*, or /). The statechart then enters the "opEntered" state, in which the calculator waits for the second operand. When the user clicks a digit (0 .. 9) or a decimal point, the state machine transitions to the "operand2" state, which is similar to "operand1." Finally, the user clicks '=', at which point the calculator computes and displays the result. It then transitions back to the "operand1" state to get ready for another computation.

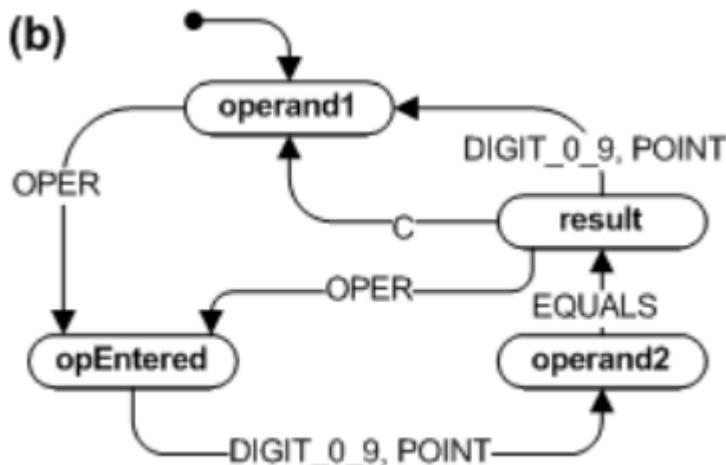


### 2. Modify your state machine so that once a result has been found the user can either:

Click an operator button to use the result as the first operand of a new calculation

Click Cancel © to start a new calculation

Enter a number to start a new calculation



The simple state model from [Figure 3-2\(a\)](#) has a major problem, however. When the user clicks '=' in the last step, the state machine cannot transition directly to "operand1" because this would erase the result from the display (to get ready for the first operand). We need another state "result" in which the calculator pauses to display the result ([Figure 3-2\(b\)](#)). Three things can happen in the "result" state: (1) the user may click an operator button to use the result as the first operand of a new computation (see the recursive production in line 2 of the calculator grammar), (2) the user may click Cancel (C) to start a completely new computation, or (3) the user may enter a number or a decimal point to start entering the first operand.

**TIP:** [Figure 3-2\(b\)](#) illustrates a trick worth remembering: the consolidation of signals PLUS, MINUS, MULTIPLY, and DIVIDE into a higher-level signal OPER (operand). This transformation avoids repetition of the same group of triggers on two transitions (from "operand1" to "opEntered" and from "result" to "opEntered"). Although most events are generated externally to the statechart, in many situations it is still possible to perform simple transformations before dispatching them (e.g., a transformation of raw button presses into the calculator events). Such transformations often simplify designs more than the trickiest state and transition topologies.

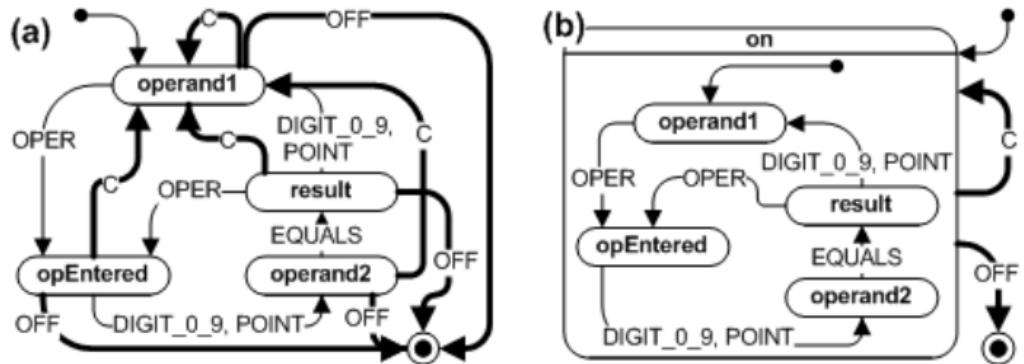
### 3. Expand your state machine to allow for:

The user can cancel and start over at any time ©

The user can turn the calculator off at any time

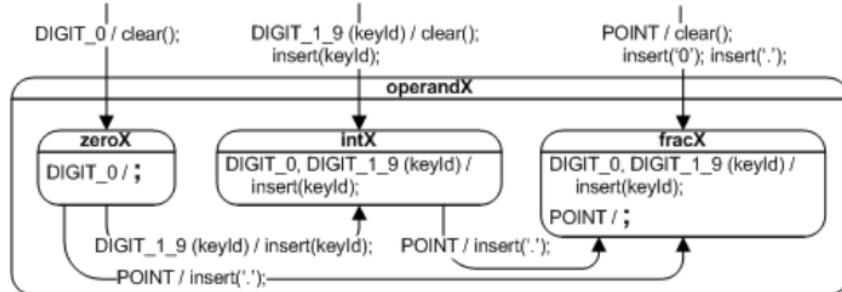
The state machine from [Figure 3-2\(b\)](#) accepts the C (Cancel) command only in the result state. However, the user expects to be able to cancel and start over at any time. Similarly, the user expects to be able to turn the calculator off at any time. Statechart in [Figure 3-3\(a\)](#) adds these features in a naïve way. A better solution is to factor out the common transition into a higher-level state named "on" and let all substates reuse the C (Cancel) and OFF transitions through behavioral inheritance, as shown in [Figure 3-3\(b\)](#).

**Figure 3-3: Applying state nesting to factorize out the common Cancel transition (C).**



#### 4. Create a sub-machine for the operands to handle floating point numbers

**Figure 3-4: Internal submachine of states “operand1” and “operand2.”**



These submachines consist of three substates. The “zero” substate is entered when the user clicks 0. Its function is to ignore additional zeros that the user may try to enter (so that the calculator displays only one 0). Note my notation for explicitly ignoring an event. I use the internal transition (DIGIT\_0 in this case) followed by an explicitly empty list of functions (a semicolon in C).

The function of the “int” substate is to parse integer part of a number. This state is entered either from outside or from the “zero” peer substate (when the user clicks 1 through 9). Finally, the substate “frac” parses the fractional part of the number. It is entered from either outside or both peer substates when the user clicks a decimal point (.). Again, note that the “frac” substate explicitly ignores the decimal point POINT event, so that the user cannot enter multiple decimal points in the fractional part of a number.

# UML Tutorial: Collaboration Diagrams

Robert C. Martin

Engineering Notebook Column

Nov/Dec, 97

In this column we will explore UML collaboration diagrams. We will investigate how they are drawn, how they are used, and how they interact with UML class diagrams.

## **UML 1.1**

On the first of September, the three amigos (Grady Booch, Jim Rumbaugh, and Ivar Jacobson) released the UML 1.1 documents. These are the documents that have been submitted to the OMG for approval. If all goes well, the OMG will adopt UML by the end of this year.

The differences between the UML 1.0 and UML 1.1 notation are minimal. The previous article in this series, (September issue) has not been affected by the changes.

## ***Dynamic models***

There are three kinds of diagrams in UML that depict dynamic models. State diagrams describe how a system responds to events in a manner that is dependent upon its state. Systems that have a fixed number of states, and that respond to a fixed set of events are called finite state machines (FSM). UML has a rich set of notational tools for describing finite state machines. We'll be investigating them in another column.

The other two kinds of dynamic diagram fall into a category called Interaction diagrams. They both describe the flow of messages between objects. However, sequence diagrams focus on the order in which the messages are sent. They are very useful for describing the procedural flow through many objects. They are also quite useful for finding race conditions in concurrent systems. Collaboration diagram, on the other hand, focus upon the relationships between the objects. They are very useful for visualizing the way several objects collaborate to get a job done and for comparing a dynamic model with a static model

. Collaboration and sequence diagrams describe the same information, and can be transformed into one another without difficulty. The choice between the two depends upon what the designer wants to make visually apparent.

Sequence diagrams will be discussed in a future article. In this article we will be concentrating upon collaboration diagrams.

## ***The interplay between static and dynamic models.***

There is a tendency among novice OO designers to put too much emphasis upon static models. Static models depicting classes, inheritance relationships, and aggregation relationships are often the first diagrams that novice engineers think to create. Disastrously, they are sometimes the *only* diagrams that they create.

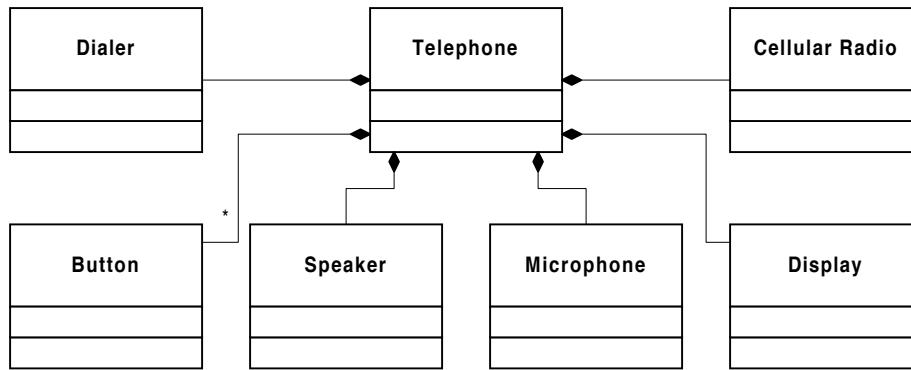
In fact, a static emphasis on object oriented design is inappropriate. Software design is about behavior; and behavior is dynamic. Object oriented design is a technique used to separate and encapsulate behaviors. Therefore an emphasis upon dynamic models is very important.

More important, however, is the interplay that exists between the static and dynamic models. A static model cannot be proven accurate without associated dynamic models. Dynamic models, on the other hand, do not adequately represent considerations of structure and dependency management. Thus, the designer must iterate between the two kinds of models, driving them to converge on an acceptable solution.

## Example: A Cellular Phone.

Consider the software that controls a very simple cellular telephone. Such a phone has buttons for dialing digits, and a “send” button for initiating a call. It has “dialer” hardware and software that gathers the digits to be dialed and emits the appropriate tones. It has a cellular radio that deals with the connection to the cellular network. It has a microphone, a speaker, and a display.

From this simple spec, we might be tempted to create a static model as shown in Figure 1.



**Figure 1: Static model of a Cellular Phone**

It is very hard to argue with this static model. The composition relationships reflect, very clearly, the specification above. Indeed, the telephone “has” all the listed components. But is this the correct static model? How would be know?

One criterion is to compare the static model to the real world. Certainly Figure 1 passes this test. In the real world, a cellular phone “has” all the components shown above. However, experienced object oriented designers know that, while this test is essential, it is not sufficient. Figure 1 does not show the *only* static model that matches the real world of the cellular telephone. In order to choose between the many possible static models a more sensitive test is needed.

## Specifying Dynamics.

How does the cellular phone work? To keep things simple, lets just look at how a customer might make a phone call. The use case for this interaction looks like this:

- ```
Use case: Make Phone Call
1. User presses the digit buttons to enter the phone number.
2. For each digit, the display is updated to add the digit to the
   phone number.
3. For each digit, the dialer generates the corresponding tone and
   emits it from the speaker.
4. User presses "Send"
5. The "in use" indicator is illuminated on the display
6. The cellular radio establishes a connection to the network.
7. The accumulated digits are sent to the network.
8. The connection is made to the called party.
```

This is simplistic, but adequate for our purposes. The use case makes it clear that there is a procedure involved with making a call. How do the objects in the static model collaborate to execute this procedure?

Let’s trace the process one step at a time. The first thing that happens when this use case is initiated is that the user presses a digit button to begin entering the phone number. How does the software in the phone know that a button has been pushed?

There are a variety of ways that this can be accomplished; but they can all be simplified to having a **Button** object that sends a **digit** message. Which object should receive the digit message? It seems clear that it should be the **Dialer**. The **Dialer** must then tell the **Display** to show the new digit, and must tell the **Speaker** to emit the appropriate tone. The **Dialer** must also remember the digits in the list that accumulates the phone number. Each new button press follows the same procedure until the “Send” button is pressed.

When the “Send” button is pressed, the appropriate **Button** object sends the **Send** message to the **Dialer**. The **Dialer** then sends a **Connect** message to the **CellularRadio** and passes along the accumulated phone number. The **CellularRadio** then tells the **Display** to illuminate the “In Use” indicator.

This simple procedure is depicted in the collaboration diagram in Figure 2.

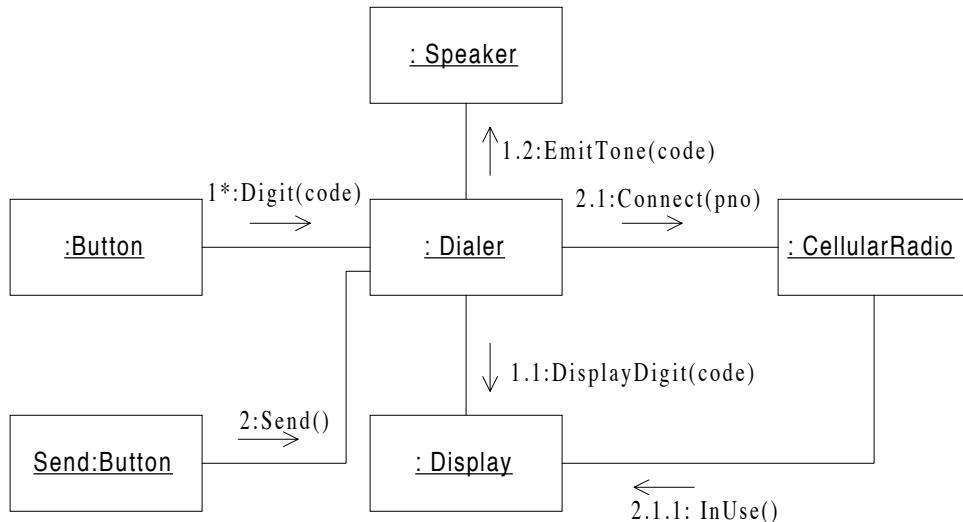


Figure 2: Collaboration Diagram of “Make Phone Call” use case.

Syntax

First, let’s look at the syntax of the diagram above. The rectangles in this diagram depict objects, not classes. You can tell that they are objects because they are underlined. In UML, something that is underlined is an *instance*; whereas something that is not underlined is a template from which an instance can be created. Notice the object on the lower left entitled **Send:Button**. In UML the full name of an object is a composite that includes the name of its class. The name of the object and the name of the class are separated by a colon. Notice that all the other objects in the diagram are anonymous; they have no name, and their class is shown with a colon in front.

The lines connecting the objects are called links. Links are *instances* of associations. You are not allowed to create a link on a collaboration diagram if there is no corresponding association, (or aggregation, or composition) on a class diagram. Remember this rule, we’ll come back to it later.

The arrows represent messages; and are labeled with their names, sequence numbers, and arguments. The name of a message corresponds to the name of a member function. That member function must exist in the class that the receiving object is instantiated from. The sequence numbers show the order in which the messages occur. The sequence numbers are nested so that you can tell which messages are sent from within other messages.

For example, message **2:Send()** is sent to the **Dialer**. As a result the **Dialer** begins executing a member function. Before that function returns, it sends message **2.1:Connect(pno)** to the

`CellularRadio`. The `CellularRadio` then sends message `2.1.1:InUse()` to the `Display`. Thus, the dot structure of the sequence numbers make it easy to see the procedural nesting of the messages.

Message `1*:Digit(code)` has an asterisk in order to denote that it may occur many times before message 2. UML defines way to use this syntax to represent loops and conditions that are beyond the scope of this article. We'll come back to such details in a subsequent article.

Reconciling the static model with the dynamic model.

It should be clear that the structure of the objects in the dynamic model (Fig 1) does not look very much like the structure of the classes in the static model (Fig 2). Yet by the rule we talked about above, a link between objects must be represented by a relationship between the classes. Thus, we have a problem.

The problem could be that our dynamic model is incorrect. Perhaps we should force the dynamic model to look like the static model. However, consider what such a dynamic model would look like. There would be a `Telephone` object in the middle that would receive messages from the other objects. The `Telephone` object would respond to each incoming message with outgoing messages of its own.

Such a design would be highly coupled. The `Telephone` object would be the master controller. It would know about all the other objects, and all the other objects would know about it. It would contain all the intelligence in the system, and all the other objects would be correspondingly stupid. This is not desirable because such a “god” object becomes highly interconnected. When any part of it changes, other parts of it may break.

I prefer the dynamic model shown in Figure 2. The concerns are decentralized in a reasonable fashion. Each object has its own little bit of intelligence, and no particular object is in charge of everything. Changes to one part of the model do not necessarily ripple to other parts.

But this means that our static model is inappropriate. It should be changed to look like Figure 3. Notice that I have demoted all the composition relationships to associations. This is because none of the connected classes share a whole/part relationship with the others. The `Dialer` is not part of the `Button` class, The `CellularRadio` is not part of the `Dialer`, etc. Notice also that I have specified the direction of navigation. The dynamic model makes it very clear which class needs to navigate to which other classes. I have also added the member functions into the class icons; again because the dynamic model made them so apparent.

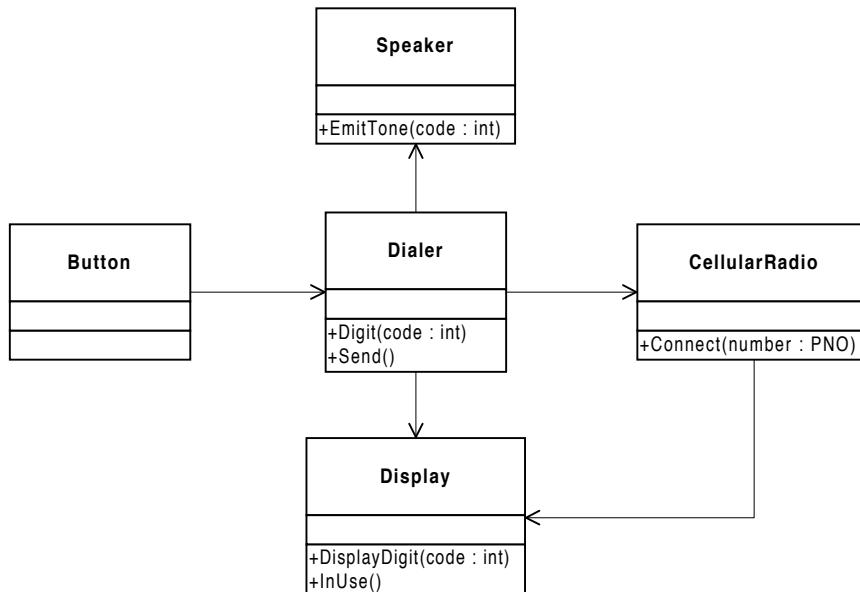


Figure 3: Reconciled static model

You might feel uncomfortable with this static model because it does not seem to reflect the real world as well as the first. After all, we have lost the notion of the telephone containing the buttons and the display, etc. But that notion was based upon the *physical* components of the telephone, not upon its behavior. Indeed the new static model is based upon the real world behavior of the telephone rather than upon its real world physical makeup.

We also lost a few classes. The **Telephone** and **Microphone** classes played no part in the dynamic model, and so have been removed. It may be that some other dynamic scenario will require them. If that happens, then we will put them back.

This points out the fact that many dynamic models usually accompany a single static model. Each dynamic model explores a different variation of a use case, scenario, or requirement. The links between the objects in those dynamic models imply a set of associations that must be present in a static model. Thus, dynamic models tend to vastly outnumber static models.

Scrutinizing the static model

Our static model has a few problems. For example, why should a class named **Button** know anything about a class named **Dialer**? Shouldn't the **Button** class be reusable in programs that don't have **Dialers**?

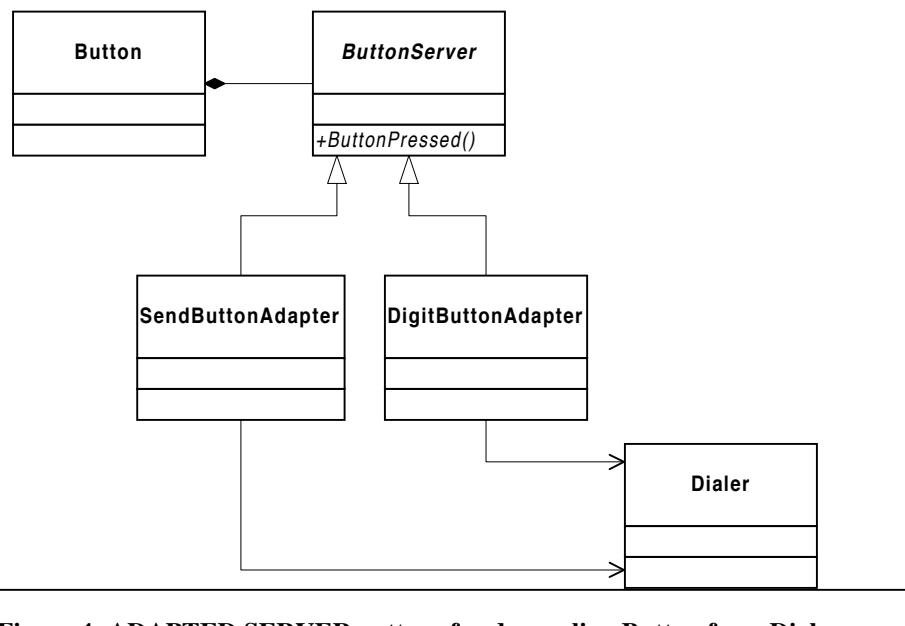


Figure 4: ADAPTED SERVER pattern for decoupling Button from Dialer

We can solve this problem by employing the ADAPTED SERVER pattern as shown in Figure 4. Now the **Button** class is completely reusable. Any other class that needs to detect a button press simply derives from **ButtonServer** and implements the pure virtual **ButtonPressed** function. If, like **Dialer**, the class must detect many different **Button** objects, ADAPTERS can be used to catch the **ButtonPressed** messages and translate them.

Another problem with the static model in Figure 3 is the high coupling of the **Display** class. This class will be the target of an association from many different clients. Those of you who recall my column entitled “The Interface Segregation Principle” (ISP) [*C++ Report*, Aug, 1996. *This document is also available in the ‘publications’ section of <http://www.oma.com>*] will understand that an unwarranted dependency exists between the **CellularRadio** class and the **Dialer** class. If one of the methods of **Display** needs to

be altered because of the needs of `Dialer`, then `CellularRadio` will be affected; at very least, by an unwarranted recompile.

To solve this problem, we can segregate the interfaces of the `Display` class as shown in Figure 5.

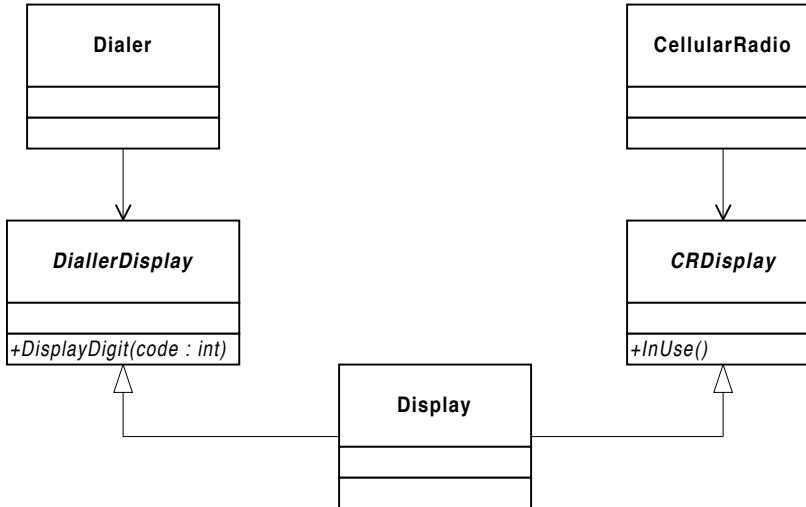


Figure 5: Interface Segregation of the Display

Iterating the dynamic model

Clearly these perturbation will have an effect upon the dynamic model. Thus, it will have to be changed as shown in Figure 6. This model shows how the adapters translate the `ButtonPressed` messages into something that the `Dialer` can understand. It also shows the segregation of the display interfaces. Note that the object named `display` appears twice, but with different class names. This indicates that `display` is derived from more than one class. Thus, the class name of the object tells the reader which interface the sender is depending upon.

Conclusion

We have completed two iterations of our static and dynamic model of a cellular phone. The first iteration was simply a guess. In the case of the first static model, the guess was pretty bad. However, after the second iteration we had resolved the disparity between the two models and had begun to explore more subtle design issues. In a real project, this iteration would continue until the designer was satisfied that both models were appropriately tuned.

Static models are necessary but insufficient for complete object oriented designs. A static model that is produced without the benefit of dynamic analysis is bound to be incorrect. The appropriate static relationships are a result of the dynamic needs of the application. UML Collaboration diagrams are a good way to depict dynamic models and compare them to the static models that must support them.

In future columns, we will continue to explore the wiles of UML. Among other things we will discuss UML's rich notation for finite state machines. We will explore how race conditions in concurrent systems can be detected with sequence diagrams. And we will demonstrate the facilities within UML that allow it to be extended.

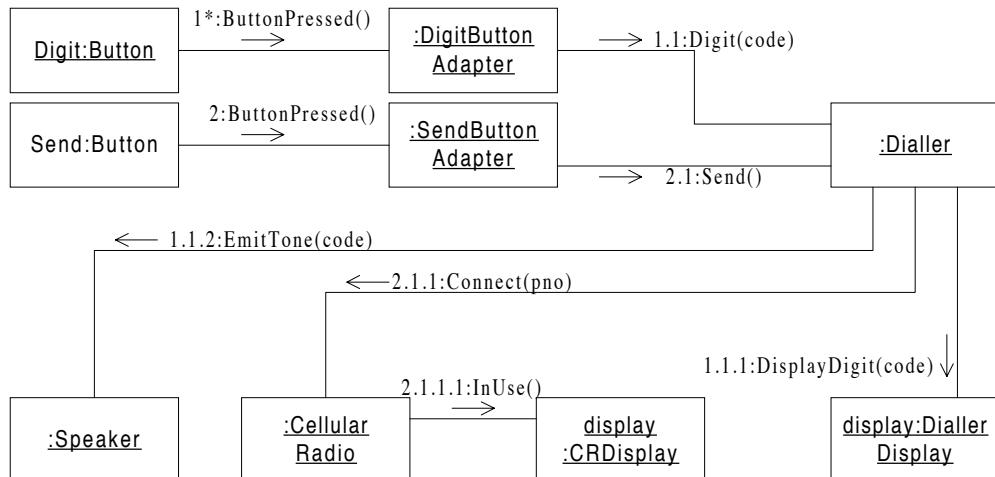


Figure 6: Iterated Dynamic Model



Software design

- Moving from analysis to design
- Design principles
 - Abstraction
 - Modularity
 - Coupling
 - Cohesion

CE202 Software Engineering, Autumn term

In This Lecture You Will Learn:

- ▶ The difference between analysis and design
- ▶ The difference between logical and physical design
- ▶ The difference between system and detailed design
- ▶ The characteristics of a good design
- ▶ The need to make trade-offs in design
- ▶ About some design principles



How is Design Different from Analysis?

- ▶ Design states ‘how the system will be constructed without actually building it’
(Rumbaugh, 1997)
- ▶ Analysis identifies ‘what’ the system must do
- ▶ Design specifies ‘how’ it will do it



How is Design Different from Analysis?

- ▶ The analyst seeks to understand the organization, its requirements and its objectives
- ▶ The designer seeks to specify a system that will fit the organization, provide its requirements effectively and assist it to meet its objectives



How is Design Different from Analysis?

- ▶ As an example, in a Campaign case study:
 - ▶ analysis identifies the fact that the **Campaign** class has a **title** attribute
 - ▶ design determines how this will be entered into the system, displayed on screen and stored in a database, together with all the other attributes of **Campaign** and other classes

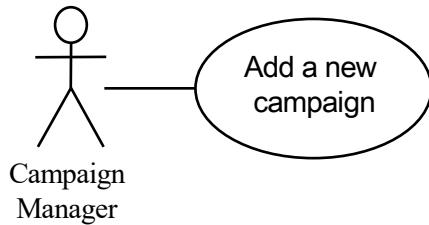


When Does Analysis Stop and Design Start?

- ▶ In a waterfall life cycle there is a clear transition between the two activities
- ▶ In an iterative life cycle the analysis of a particular part of the system will precede its design, but analysis and design may be happening in parallel
- ▶ It is important to distinguish the two activities and the associated mindset
- ▶ We need to know ‘what’ before we decide ‘how’

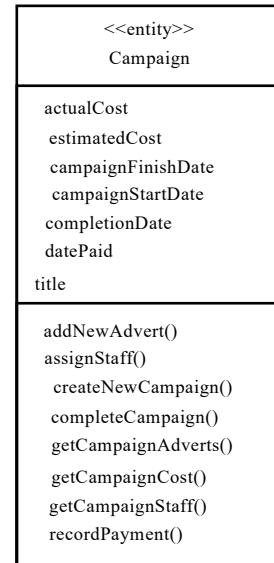
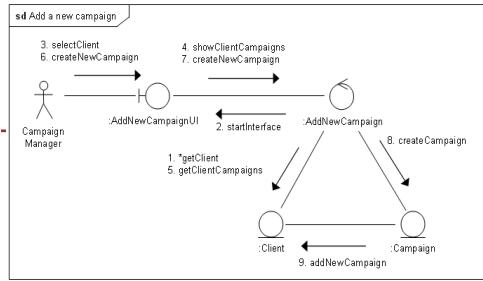


Requirements

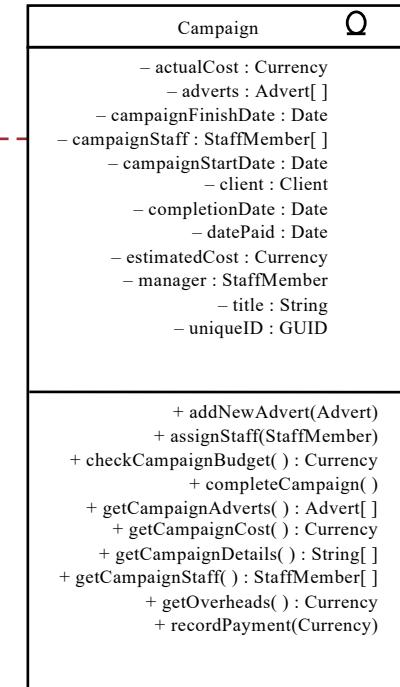


To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion.

Analysis



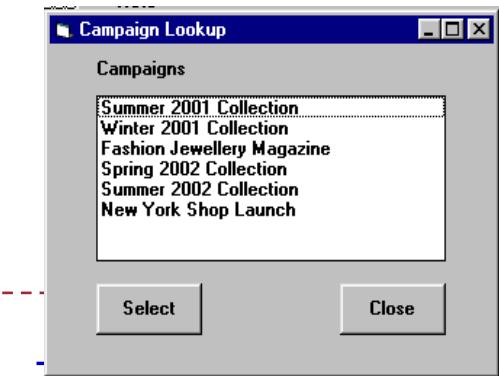
Design



```

CREATE TABLE Campaigns
(VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
 FLOAT actualCost,
 DATE campaignFinishDate,
 DATE campaignStartDate,
 VARCHAR(30) clientID NOT NULL,
 DATE completionDate,
 DATE datePaid,
 FLOAT estimatedCost,
 VARCHAR(30) managerID,
 VARCHAR(50) title);

CREATE INDEX campaign_idx ON Campaigns (clientID, managerID, title);
  
```



Traditional Design

- ▶ Making a clear transition from analysis to design has advantages
 - ▶ project management—is there the right balance of activities?
 - ▶ staff skills—analysis and design may be carried out by different staff
 - ▶ client decisions—the client may want a specification of the ‘what’ before approving spending on design
 - ▶ choice of development environment—may be delayed until the analysis is complete



Design in the Iterative Life Cycle

- ▶ Advantages of the iterative life cycle include
 - ▶ risk mitigation—making it possible to identify risks earlier and to take action
 - ▶ change management—changes to requirements are expected and properly managed
 - ▶ team learning—all the team can be involved from the start of the project
 - ▶ improved quality—testing begins early and is not done as a ‘big bang’ with no time



Seamlessness

- ▶ The same model—the **class model**—is used through the life of the project
- ▶ During design, additional detail is added to the analysis classes, and extra classes are added to provide the supporting functionality for the user interface and data management
- ▶ Other diagrams are also elaborated in design activities



Logical and Physical Design

- ▶ In structured analysis and design a distinction has been made between **logical** and **physical** design
- ▶ **Logical design** is independent of the implementation language and platform
- ▶ **Physical design** is based on the actual implementation platform and the language that will be used



Logical and Physical Design Example

- ▶ Some design of the user interface classes can be done without knowing whether it is to be implemented in Java, C++ or some other language-types of fields, position in windows
- ▶ Some design can only be done when the language has been decided upon — the actual classes for the types of fields, the layout managers available to handle window layout



Logical and Physical Design

- ▶ It is not necessary to separate these into two separate activities
- ▶ It may be useful if the software is to be implemented on different platforms
- ▶ Then it will be an advantage to have a platform-independent design that can be tailored to each platform



System Design and Detailed Design

- ▶ **System design** deals with the high level architecture of the system (see lecture next week)
 - ▶ structure of sub-systems
 - ▶ distribution of sub-systems on processors
 - ▶ communication between sub-systems
 - ▶ standards for screens, reports, help etc.
 - ▶ job design for the people who will use the system



System Design and Detailed Design

- ▶ Traditional **detailed design** consists of four main activities
 - ▶ designing inputs
 - ▶ designing outputs
 - ▶ designing processes
 - ▶ designing files and database structures



System Design and Detailed Design

- ▶ Traditional detailed design tried to **maximise cohesion**
 - ▶ elements of a module of code all contribute to the achievement of a single function
- ▶ Traditional detailed design tried to **minimise coupling**
 - ▶ unnecessary linkages between modules that made them difficult to maintain or use in isolation from other modules
- ▶ Discussed later in the lecture

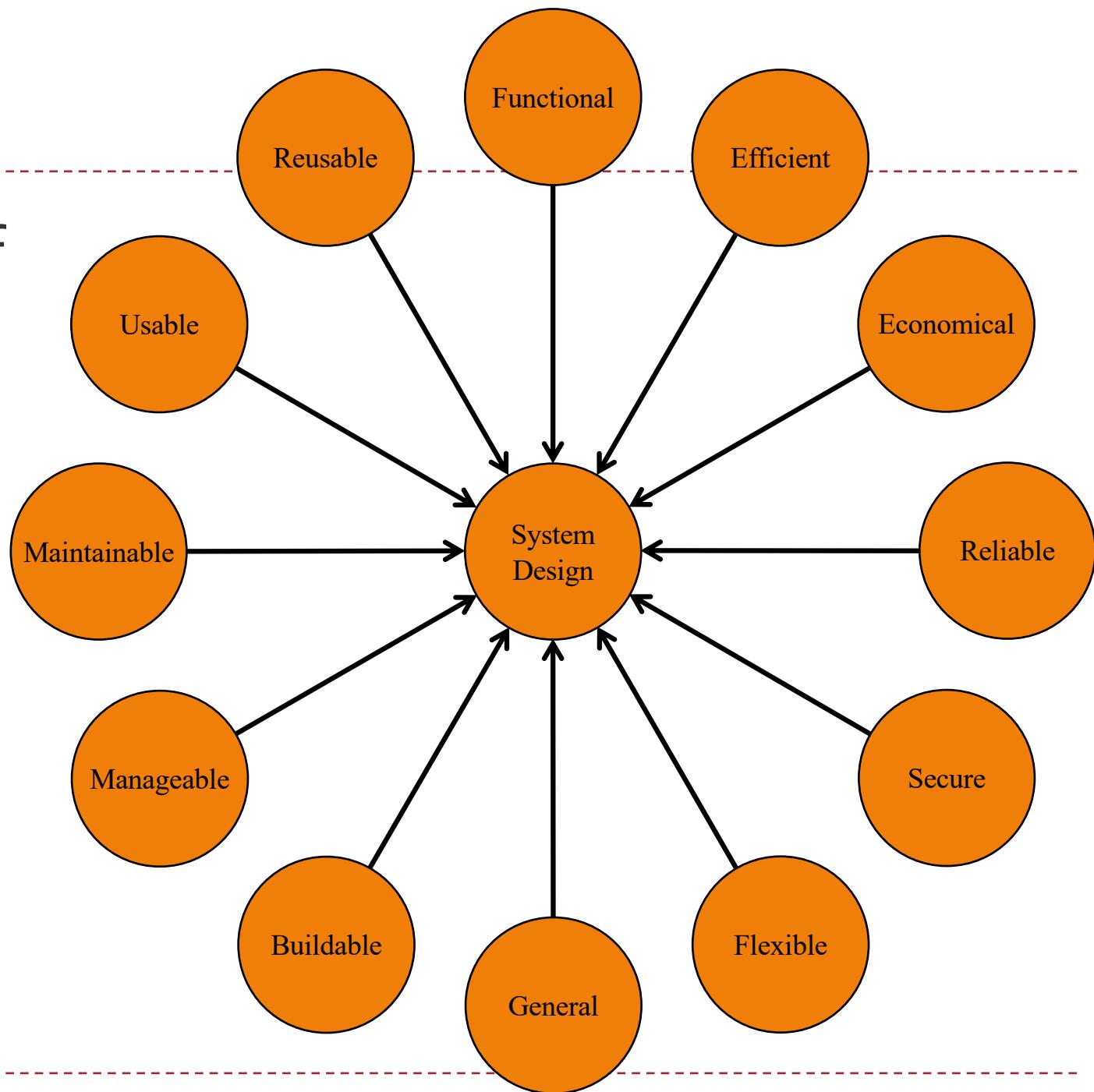


System Design and Detailed Design

- ▶ Object-oriented detailed design adds detail to the analysis model
 - ▶ types of attributes
 - ▶ operation signatures
 - ▶ assigning responsibilities as operations
 - ▶ additional classes to handle user interface
 - ▶ additional classes to handle data management
 - ▶ design of reusable components
 - ▶ assigning classes to packages



Qualities of Design



Qualities of Design (1 of 3)

- ▶ Functional—system will perform the functions that it is required to
- ▶ Efficient—the system performs those functions efficiently in terms of time and resources
- ▶ Economical—running costs of system will not be unnecessarily high
- ▶ Reliable—not prone to hardware or software failure, will deliver the functionality when the users want it



Qualities of Design (2 of 3)

- ▶ Secure—protected against errors, attacks and loss of valuable data
- ▶ Flexible—capable of being adapted to new uses, to run in different countries or to be moved to a different platform
- ▶ General—general-purpose and portable (mainly applies to utility programs)
- ▶ Buildable—Design is not too complex for the developers to be able to implement it



Qualities of Design (3 of 3)

- ▶ Manageable—easy to estimate work involved and to check of progress
- ▶ Maintainable—design makes it possible for the maintenance programmer to understand the designer’s intention
- ▶ Usable—provides users with a satisfying experience (not a source of dissatisfaction)
- ▶ Reusable—elements of the system can be reused in other systems



Prioritizing Design Trade-offs

- ▶ Designer is often faced with design objectives that are mutually incompatible.
- ▶ It is helpful if guidelines are prepared for prioritizing design objectives.
- ▶ If design choice is unclear users should be consulted.



Trade-offs in Design

- ▶ Design to meet all these qualities may produce conflicts
- ▶ Trade-offs have to be applied to resolve these
- ▶ Functionality, reliability and security are likely to conflict with economy
- ▶ Level of reliability, for example, is constrained by the budget available for the development of the system



Measurable Objectives in Design

- ▶ In the requirements phase a set of **non-functional requirements** are described
- ▶ How can we tell whether these have been achieved?
- ▶ Measurable objectives set clear targets for designers
- ▶ Objectives should be quantified so that they can be tested



Measurable Objectives in Design

- ▶ To reduce invoice errors by one-third within a year
- ▶ How would you design for this?



Measurable Objectives in Design

- ▶ To reduce invoice errors by one-third within a year
- ▶ How would you design for this?
 - ▶ sense checks on quantities
 - ▶ comparing invoices with previous ones for the same customer
 - ▶ better feedback to the user about the items ordered



Measurable Objectives in Design

- ▶ To process 50% more orders at peak periods
- ▶ How would you design for this?



Measurable Objectives in Design

- ▶ To process 50% more orders at peak periods
- ▶ How would you design for this?
 - ▶ design for as many fields as possible to be filled with defaults
 - ▶ design for rapid response from database
 - ▶ design system to handle larger number of simultaneous users



Some general design principles

- Abstraction
- Modularity
- Coupling
- Cohesion



Abstraction

▶ Process:

- ▶ “*Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details.*” [Ghezzi et. al 1991]

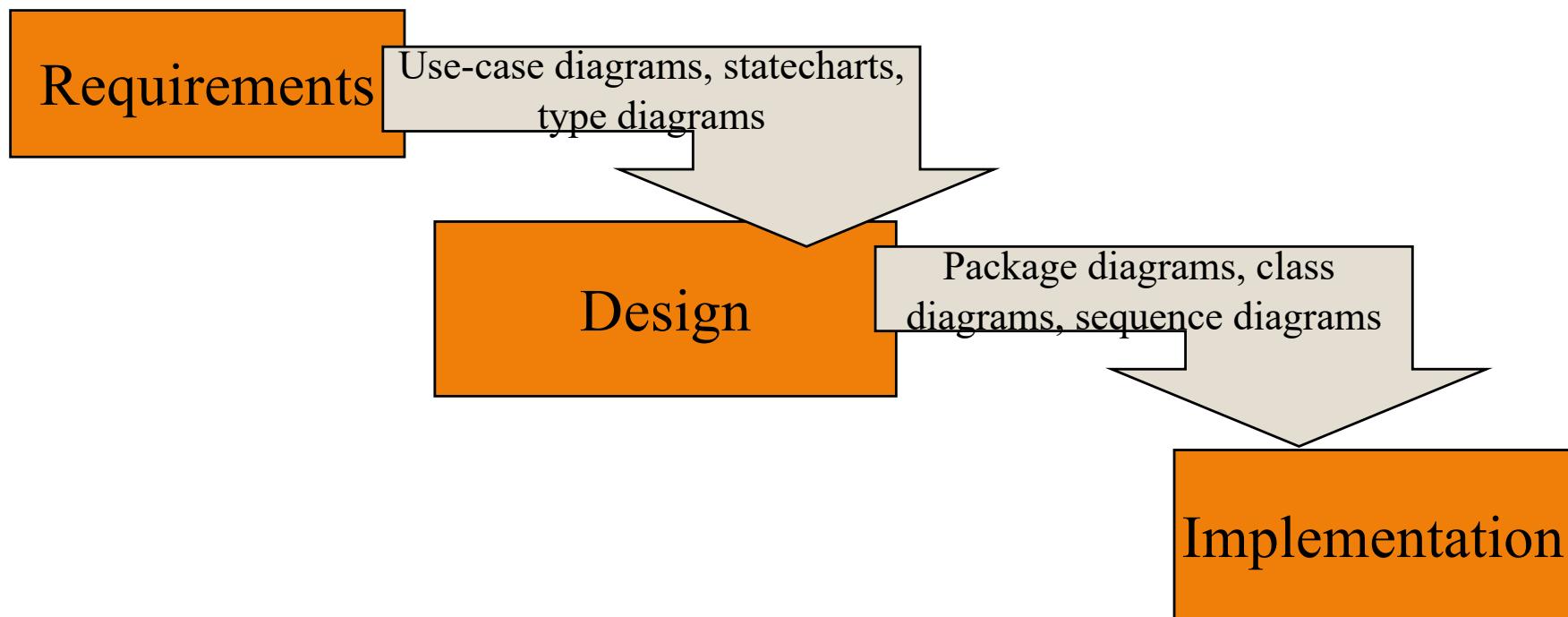
▶ Product:

- ▶ “[A] simplified description, ... that emphasizes some of the system's ... properties while suppressing others.”
- ▶ *A good abstraction is one that emphasizes details that are significant to the reader or user and suppress details that are, at least for the moment, immaterial or diversionary.*” [Shaw 1984]



Software design's level of abstraction

- ▶ A level of abstraction that is between requirements and implementation



Modularity

Kinds of modules
Cohesion
Coupling

Modularity

- ▶ The quality of being divided into modules
- ▶ Quality attributes:
 - ▶ Well-defined: Modules are clearly distinguished
 - ▶ Separation of concerns: each module has one concern, with minimal overlap between modules
 - ▶ Loosely coupled
 - ▶ Cohesive



Why should design be modular?

- ▶ Comprehensibility
- ▶ Parallel development
- ▶ Quality: easier to—
 - ▶ Test (see Unit testing lecture)
 - ▶ Verify and validate
 - ▶ Measure reliability
- ▶ Maintainability: easier to—
 - ▶ Locate faults and correct them
 - ▶ Change
 - ▶ Enhance
- ▶ Reusability



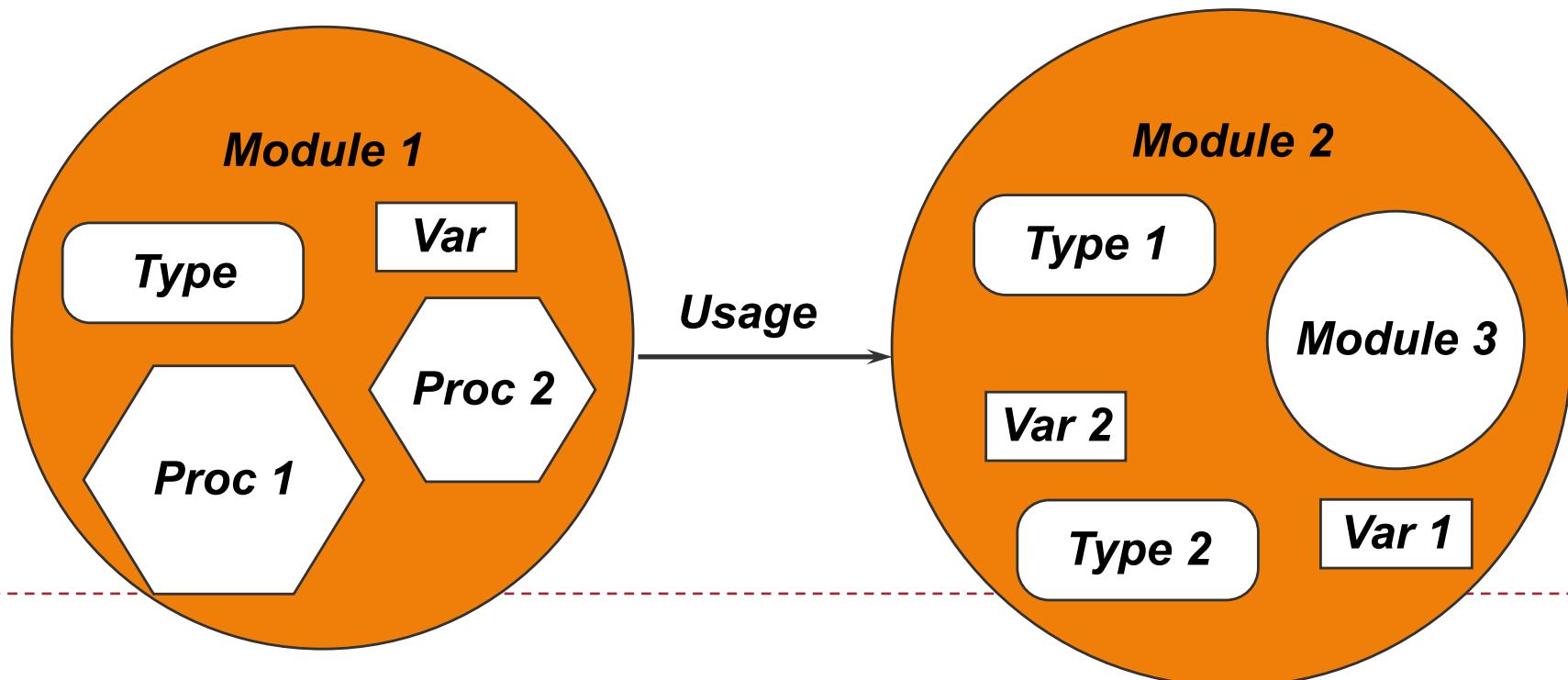
Module: Definition 1 (general)

- ▶ Any kind of an “independent” software unit
 - ▶ Routines, subroutines (in Java: “methods”; in C: “functions”)
 - ▶ Various physical units
 - ▶ Files (as in C, C++)
 - ▶ Library files (.a is the UNIX convention, DLL in Windows, JAR in Java)



Module: Definition 2 (object-based programming)

- ▶ Packaging of “related” variables and procedures
 - ▶ Packages (as in ADA)
 - ▶ Modules (as in Modula)
- ▶ Programming paradigm: “modular programming” or “Object-based programming”



Module: Definition 3 (OOP)

- ▶ “The act of grouping into a single object both data and operations that [directly] affect that data is known as encapsulation. ...” [Wirfs-Brock 90, p.6]
 - ▶ A ‘class’
 - ▶ An ‘object’
- ▶ Also known as: encapsulation, separation of concerns
- ▶ Programming paradigm: “object-oriented programming” or “class-based programming”

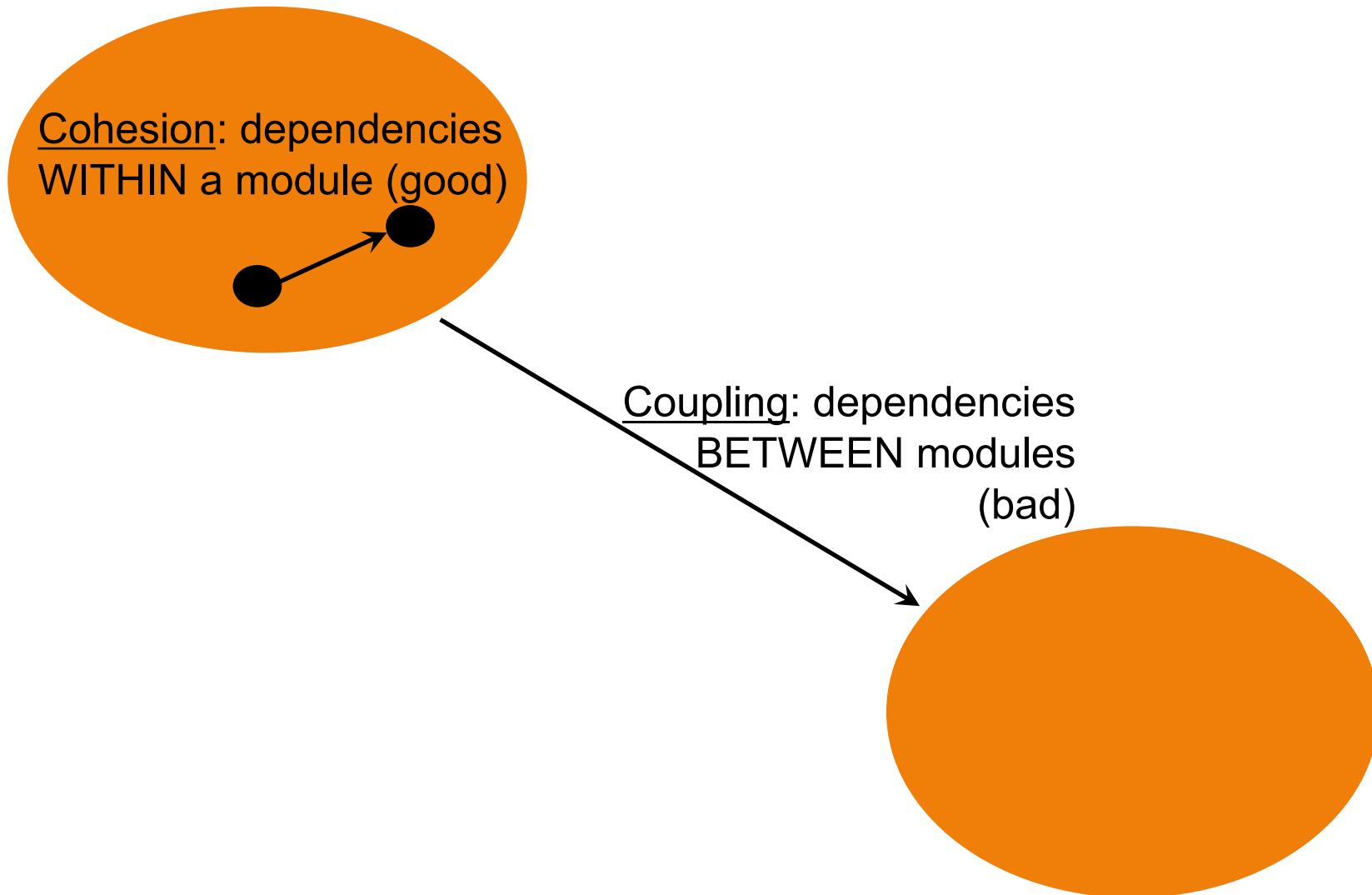




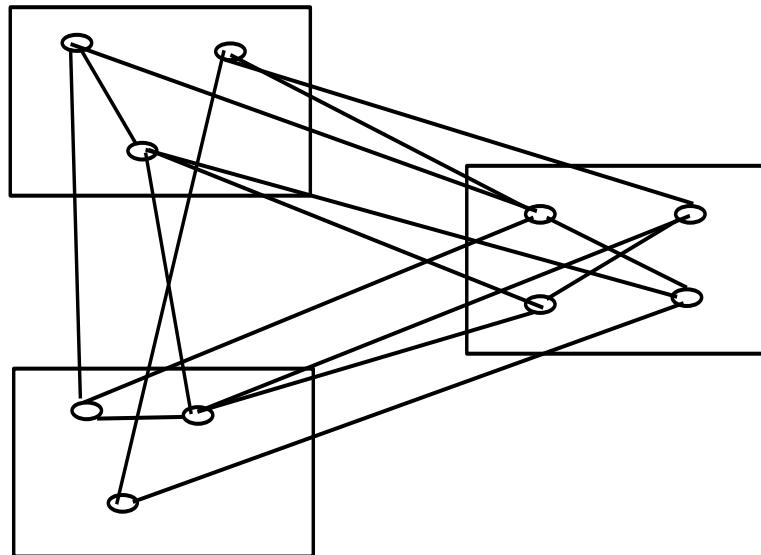
Coupling and cohesion



Cohesion vs. coupling

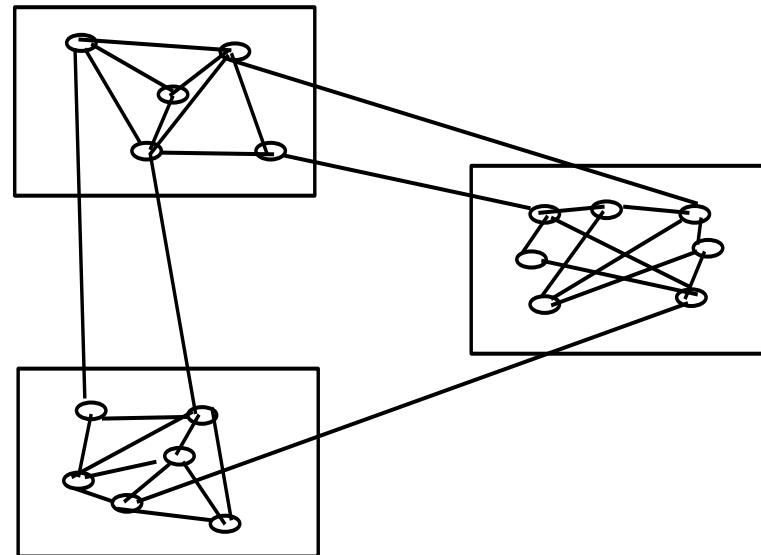


A visual representation



(a)

high coupling



(b)

low coupling



Coupling

- ▶ Coupling is measured by the answer to this question:
How much of one module must be known in order to understand another module?
- ▶ The more that we must know of module B in order to understand module A, the more they are coupled
- ▶ Objective: **loosely coupled systems**



Kinds of Direct Dependencies

[Briand et. al 99]

- ▶ From a class to its --
 - ▶ Super class
 - ▶ Emp → Person
 - ▶ Field (class)
 - ▶ Emp → Date
- ▶ From a method to its --
 - ▶ Parameter types
 - ▶ Emp. hire → Reason
 - ▶ Return type
 - ▶ Emp. hire → SuccessCode
 - ▶ (Class of) local variables
 - ▶ Emp. hire → DB
 - ▶ Invoked method
 - ▶ Emp. hire → DB. getDB

```
class Employee extends Person {  
    Date birthday;  
    SuccessCode hire(Reason why) {  
        why.print();  
        DB theDB = DB.getDB(); ...  
        return SuccessCode.successful; }  
};
```

Briand et al, 1999, Empirical studies of object-oriented artifacts, methods and processes: state of the

art and future direction. Int. J. of Empirical Software Engineering 4(4):

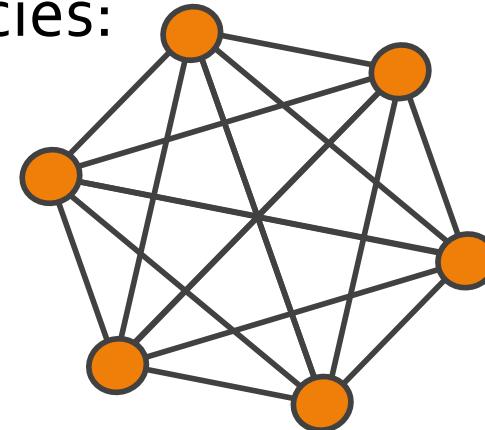


Dependency graphs

- ▶ Maximal number of dependencies:

$$n(n-1)/2$$

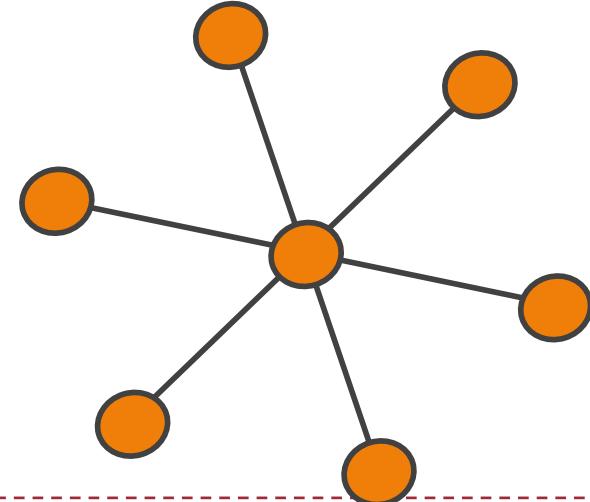
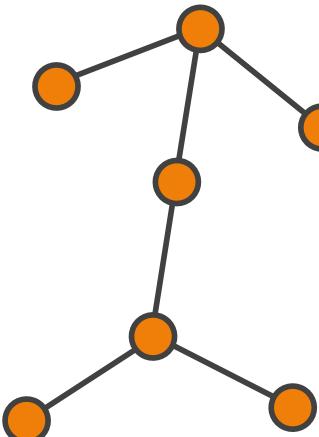
- ▶ Complete Graph



- ▶ Minimal number of interfaces $n-1$

- ▶ Tree

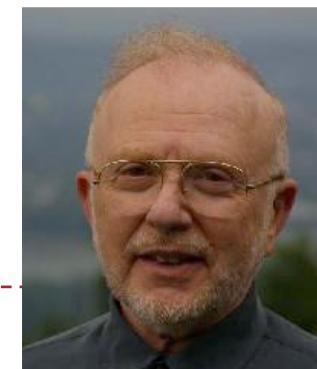
- ▶ Star



How to achieve Low Coupling

- ▶ **Low coupling** can be achieved if a calling class does not need to know anything about the internals of the called class (**Principle of information hiding**, Parnas)
- ▶ Questions to ask:
 - ▶ Does the calling class really have to know any attributes of classes in the lower layers?
 - ▶ Is it possible that the calling class calls only operations of the lower level classes?

► David Parnas, *1941,
Developed the concept of
modularity in design.



Cohesion

- ▶ Coupling between elements within a module
 - co·here v. co·hered, co·her·ing, co·heres. --intr. 1. To stick or hold together in a mass that resists separation. 2. To have internal elements or parts logically connected so that aesthetic consistency results. [American Heritage Dictionary]
- ▶ Module Cohesion is “how tightly bound or related are its internal elements to one another.”
- ▶ A desirable property!
 - ▶ E.g., cohesion within the methods of a class, within classes in a module



Levels of cohesion

[Myers '78]

- ▶ **Coincidental** - Occurs when carelessly trying to satisfy style rules.
 - ▶ print_prompt_and_check_parameters
- ▶ **Logical** - Related logic, but no corresponding relation in control or data.
 - ▶ Library of trigonometric functions, in which there is no relation between the implementation of the functions.
- ▶ **Temporal** - Series of actions related in time.
 - ▶ Initialization module.
 - ▶ Communication- Series of actions related to a step of the processing of a single function
- ▶ **Data item**. May occur in the attempt to avoid control coupling.
 - ▶ clear_window_and_draw_its_frame
- ▶ **Functional** - Execute a single, well-defined function or duty.
- ▶ **Data** - Collection of related operations on the same data.



How to achieve high Cohesion

- ▶ High cohesion can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- ▶ Questions to ask:
 - ▶ Does one subsystem always call another one for a specific service?
 - ▶ Yes: Consider moving them together into the same subsystem.
 - ▶ Which of the subsystems call each other for services?
 - ▶ Can this be avoided by restructuring the subsystems or changing the subsystem interface?
 - ▶ Can the subsystems even be hierarchically ordered (in layers)?



Summary

- ▶ The difference between analysis and design
- ▶ The difference between logical and physical design
- ▶ The difference between system and detailed design
- ▶ The characteristics of a good design
- ▶ The need to make trade-offs in design
- ▶ About some design principles
 - ▶ Abstraction
 - ▶ Modularity
 - ▶ Coupling
 - ▶ Cohesion

Further reading

- ▶ Bennett – Chapter 14, Detailed Design
- ▶ Bruegge – 6.3.3 Coupling and Cohesion
- ▶ Ghezzi 1991 – Fundamentals of Software Engineering, Prentice-Hall
- ▶ Shaw 1984 – Abstraction techniques in modern programming languages, IEEE Software, Vol 1(4)
- ▶ Wirfs-Brock, 1990, Designing object-oriented software, Prentice-Hall
- ▶ Briand et al, 1999, Empirical studies of object-oriented artifacts, methods and processes: state of the art and future direction. Int. J of Empirical Software Engineering 4(4).
- ▶ Myers 1978. Composite/Structured Design. Van Nostrand Reinhold, New York.
- ▶ Hoffman, Daniel M.; Weiss David M. (Eds.): Software Fundamentals – Collected Papers by David L. Parnas, 2001, Addison-Wesley



Exercise: coupling and cohesion (1)

- ▶ Coupling is the unnecessary dependency of one component upon another component's implementation. An example would be if you had a Dog class that should be able to bark and jump. You could write it like this:

```
class Dog
{
    void doAction(int number) {
        if(number==1)
            //Jump code goes here
        if(number==2)
            //Bark code goes here
    }
}
```

- ▶ What is wrong with this?



Exercise: coupling and cohesion (2)

- ▶ What is wrong with this implementation of the Car class?

```
Driver campbell = new Driver();
Car ford = new Car("Ford", "red");
```

...

```
public class Driver {
```

```
    Car myCar;
```

...

```
}
```

```
public void goFaster(int speed) {
```

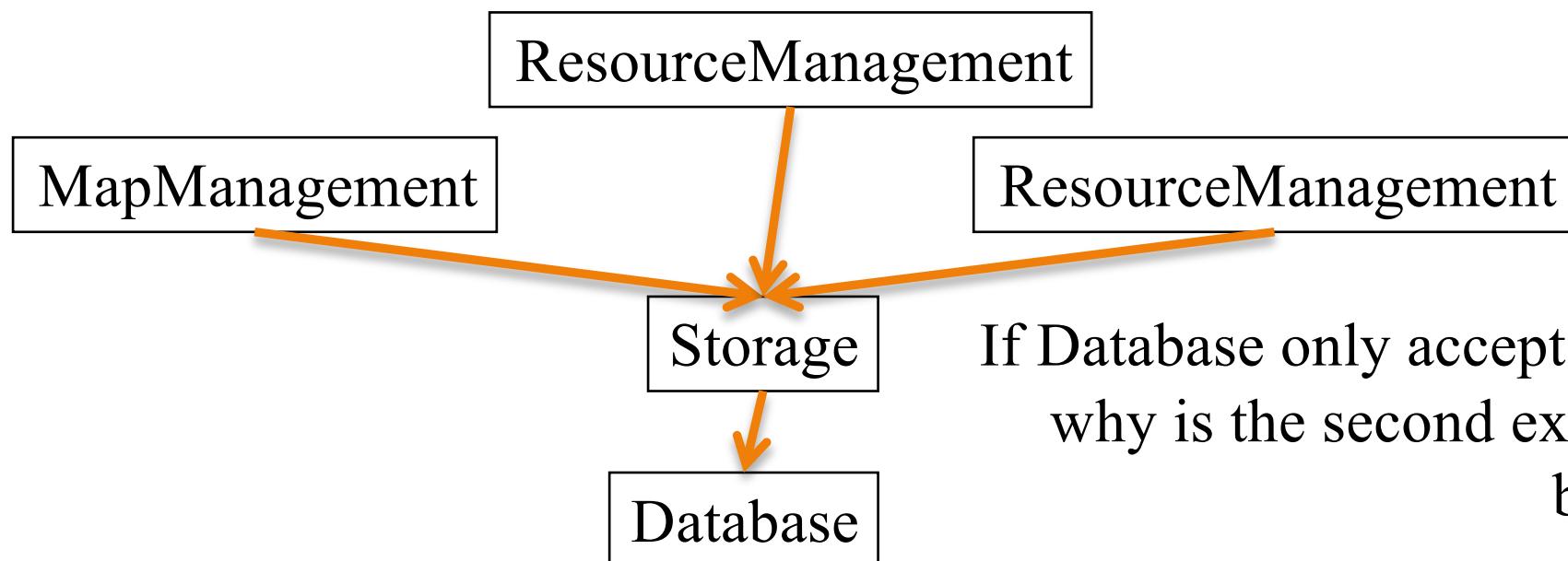
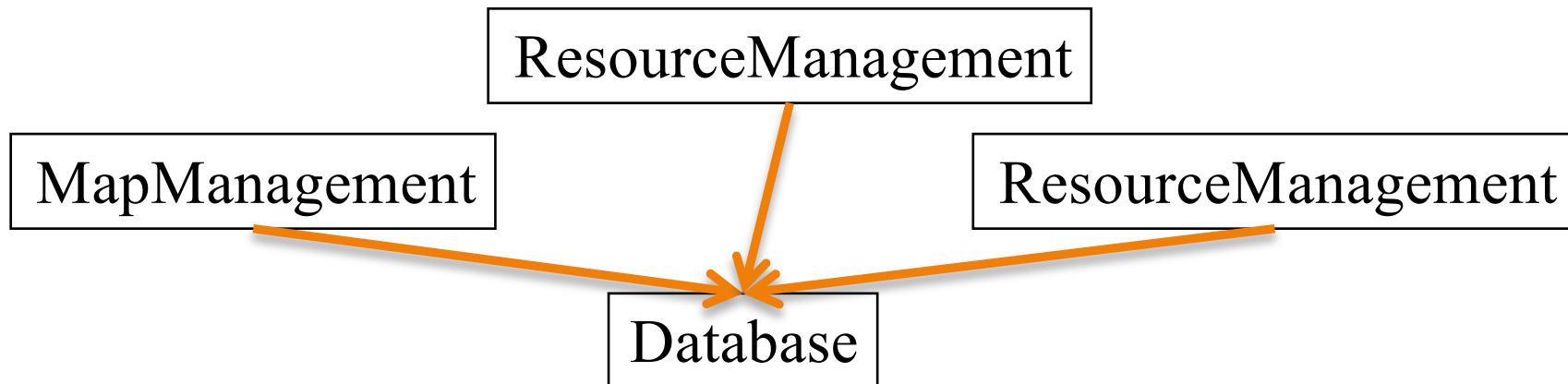
```
    myCar.speed += speed;
```

...

```
}
```

-
- ▶ How can you avoid tight coupling?

Exercise: Reducing coupling by adding complexity



If Database only accepts SQL
why is the second example
better?





Software design: design patterns

- Software development patterns
- Template Method pattern

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Software Development Patterns

A pattern:

- ▶ “describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.”

Alexander et al. (1977)



Software Development Patterns

- ▶ A pattern has:
 - ▶ A *context* = a set of circumstances or preconditions for the problem to occur
 - ▶ *Forces* = the issues that must be addressed
 - ▶ A software configuration that resolves the forces

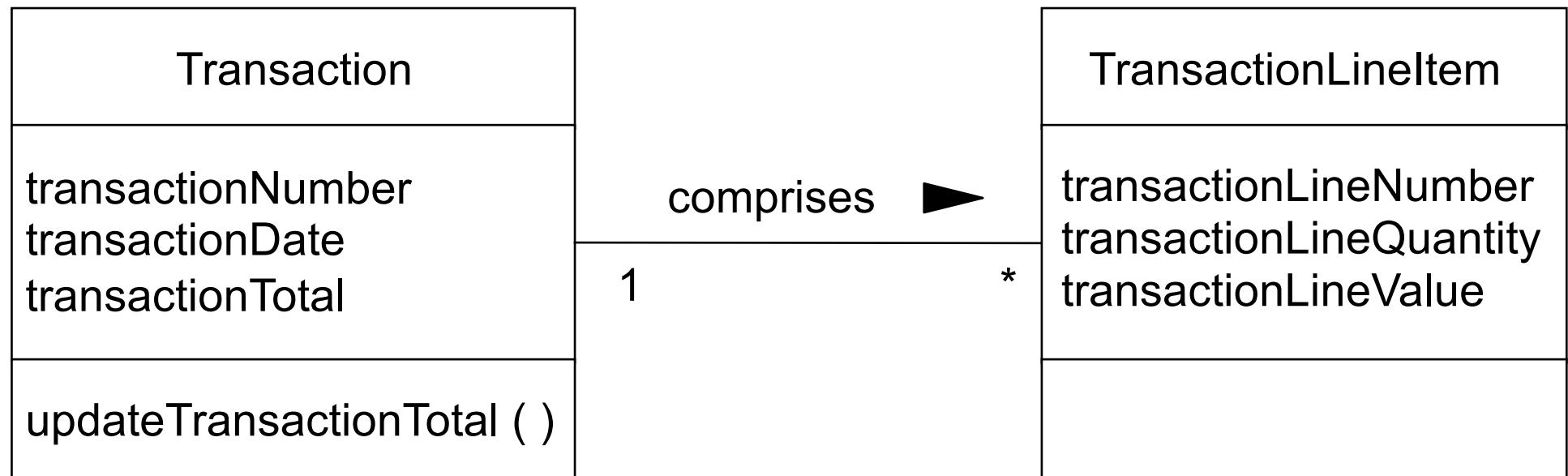


Software Development Patterns

- ▶ Patterns are found at many points in the systems development lifecycle:
 - ▶ **Analysis patterns** are groups of concepts useful in modelling requirements (see lecture on Type diagrams)
 - ▶ **Architectural patterns** describe the structure of major components of a software system (see lecture next week)
 - ▶ **Design patterns** describe the structure and interaction of smaller software components



Simplest Analysis Pattern



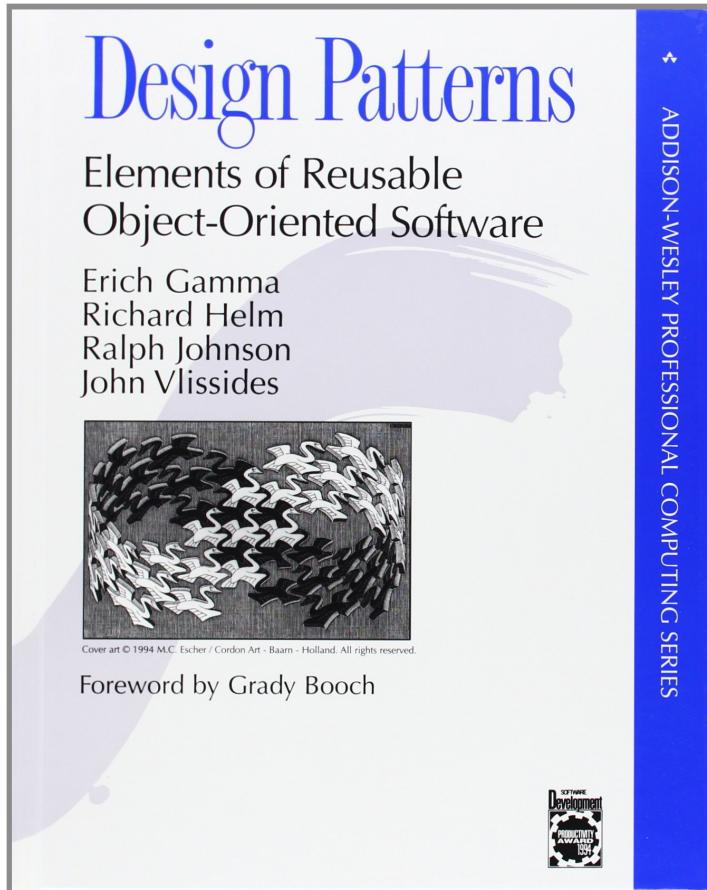
Template Method pattern

The Template Method pattern

- ▶ Intent: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.
 - ▶ Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- ▶ In object-oriented programming, first a class is created that provides the **basic steps** of an algorithm design.
- ▶ These steps are implemented using **abstract** methods.
- ▶ Later on, subclasses change the abstract methods to **implement real actions**. Thus the general algorithm is saved in one place but the concrete steps may be changed by the subclasses.
- ▶ Avoids **duplication** in the code: the general workflow structure is implemented once in the abstract class's algorithm, and necessary variations are implemented in the subclasses.



Design Patterns



- ▶ The template method is one of the twenty-three well-known patterns described in the "Gang of Four" book Design Patterns (1994).

Template Method: Example 1

An abstract class that is common to several games in which players play against the others, but only one is playing at a given time.

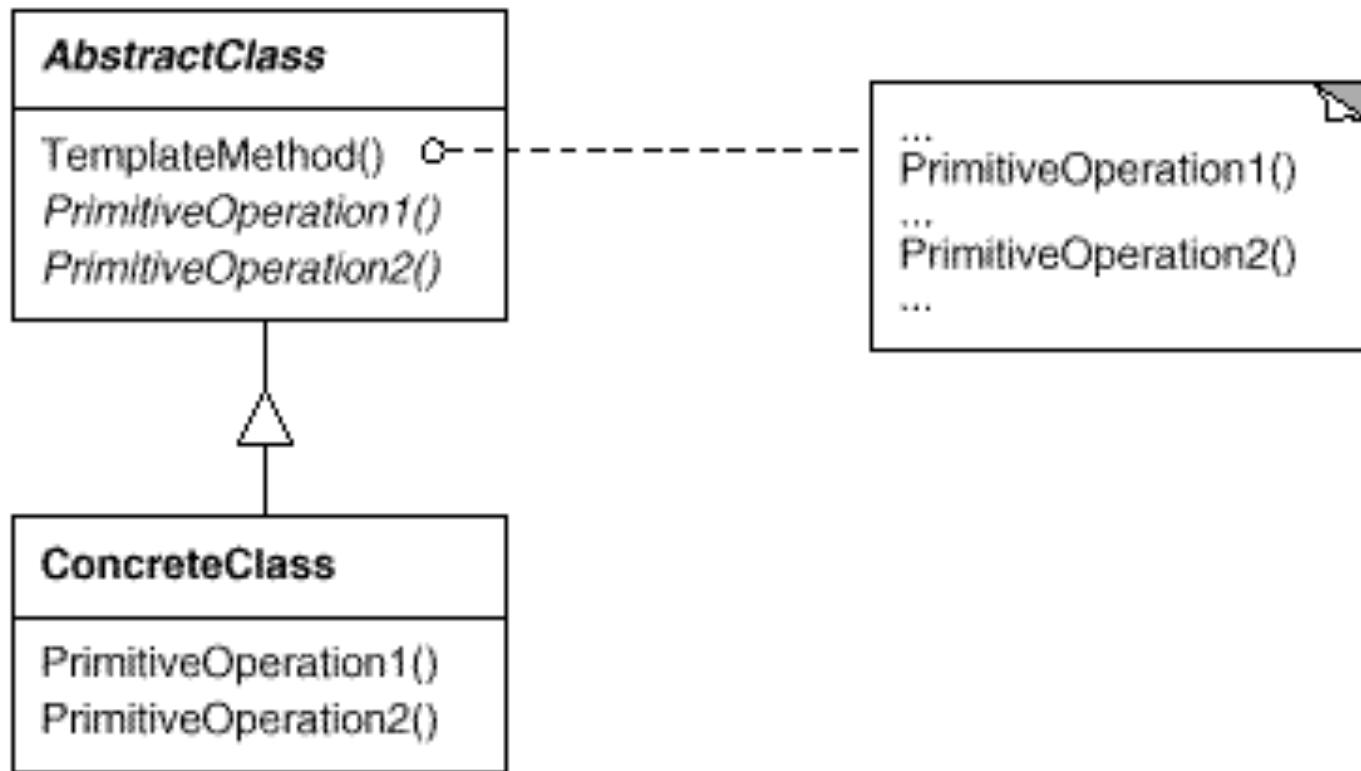
```
abstract class Game {  
  
    protected int playersCount;  
    abstract void initializeGame();  
    abstract void makePlay(int player);  
    abstract boolean endOfGame();  
    abstract void printWinner();  
  
    /* A template method : */  
    public final void playOneGame(int playersCount)  
    {  
        this.playersCount = playersCount;  
        initializeGame();  
        int j = 0;  
        while (!endOfGame()) {  
            makePlay(j);  
            j = (j + 1) % playersCount;  
        }  
        printWinner();  
    }  
}
```

```
class Monopoly extends Game {  
  
    /* Implementation of necessary concrete methods */  
  
    void initializeGame() {  
        // Initialize players  
        // Initialize money  
    }  
    void makePlay(int player) {  
        // Process one turn  
    }  
    boolean endOfGame() {  
        // Return true if game  
        // according to Monopoly rules  
    }  
    void printWinner() {  
        // Display who won  
    }  
    /* Specific declarations for Monopoly */  
    // ...  
}
```

```
class Chess extends Game {  
  
    /* Implementation of necessary concrete methods */  
  
    void initializeGame() {  
        // Initialize players  
        // Put the pieces on the board  
    }  
    void makePlay(int player) {  
        // Process a turn for the player  
    }  
    boolean endOfGame() {  
        // Return true if in Checkmate or  
        // Stalemate has been reached  
    }  
    void printWinner() {  
        // Display the winning player  
    }  
    /* Specific declarations for the chess game. */  
    // ...  
}
```

Now we can extend this class in order to implement actual games

Template Method: *Structure*



Example: J2EE

- ▶ J2EE: A framework for “developing component-based multitier enterprise applications.”
- ▶ Example: class TemplateFilter
 - ▶ The abstract filter dictates the general steps that every filter must complete
 - ▶ The abstract filter leaves the specifics of how to complete that step to each filter subclass



Example: Hook Methods in J2EE II

```
public abstract class TemplateFilter implements javax.servlet.Filter {  
    ...  
    public void doFilter(ServletRequest request,  
        ServletResponse response, FilterChain chain)  
        throws IOException, ServletException {  
        // Common processing for all filters:  
        doPreProcessing(request, response, chain); // Hook 1  
        doMainProcessing(request, response, chain); // Hook 2  
        doPostProcessing(request, response, chain); // Hook 3  
        ...  
    }  
    public void doPreProcessing(ServletRequest request,  
        ServletResponse response, FilterChain chain) {}  
  
    public void doPostProcessing(ServletRequest request,  
        ServletResponse response, FilterChain chain) {}  
  
    public abstract void doMainProcessing(ServletRequest  
        request, ServletResponse response, FilterChain chain);  
}
```



Using J2EE

- ▶ How to use class TemplateFilter?
 - ▶ Extend and override the methods doPreProcessing, doMainProcessing, doPostProcessing

```
public class DebuggingFilter extends TemplateFilter {  
    public void doMainProcessing(ServletRequest req,  
        ServletResponse res, FilterChain chain) {  
        System.out.println("Filtering request:" +  
            req.asString());  
    }  
}
```



The Template Method pattern: Applicability

- ▶ **Applicability:** The Template Method pattern should be used
 - ▶ to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
 - ▶ when common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "**refactoring to generalize**" as described by Opdyke and Johnson [[OJ93](#)]. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
 - ▶ to control subclasses extensions. You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.



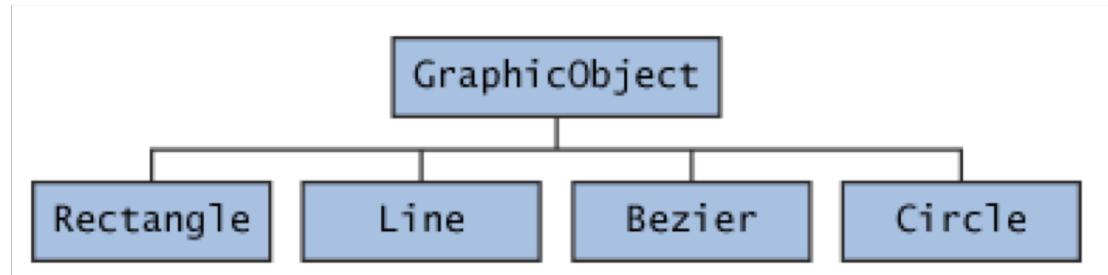
The Template Method pattern: Participants

- ▶ Participants:
 - ▶ AbstractClass (e.g., TemplateFilter)
 - ▶ defines **abstract primitive operations** that concrete subclasses define to implement steps of an algorithm.
 - ▶ implements **a template method** defining the skeleton of an algorithm. The template method calls primitive operations.
 - ▶ ConcreteClass (e.g., DebuggingFilter)
 - ▶ **implements the primitive operations** to carry out subclass-specific steps of the algorithm.



Example Abstract Class

- In an object-oriented drawing application, you can draw circles, rectangles, lines, Bezier curves, and many other graphic objects. These objects all have certain states (for example: position, orientation, line color, fill color) and behaviors (for example: `moveTo`, `rotate`, `resize`, `draw`) in common. Some of these states and behaviors are the same for all graphic objects—for example: position, fill color, and `moveTo`. Others require different implementations—for example, `resize` or `draw`. All `GraphicObjects` must know how to draw or resize themselves; they just differ in how they do it. This is a perfect situation for **an abstract superclass**. You can take advantage of the similarities and declare all the graphic objects to inherit from the same abstract parent object—for example, `GraphicObject`, as shown in the following figure.



Classes Rectangle, Line, Bezier, and Circle inherit from GraphicObject

Example implementation

- ▶ First, you declare an abstract class, `GraphicObject`, to provide member variables and methods that are wholly shared by all subclasses, such as the current position and the `moveTo` method. `GraphicObject` also declares **abstract methods for methods, such as draw or resize**, that need to be implemented by all subclasses but must be implemented in different ways. The `GraphicObject` class can look something like this:

```
abstract class GraphicObject {  
    int x, y;  
    ...  
    void moveTo(int newX, int newY) {  
        ...  
    }  
    abstract void draw();  
    abstract void resize();  
}
```

Continued ...

- ▶ Each non-abstract subclass of `GraphicObject`, such as `Circle` and `Rectangle`, must provide implementations for the `draw` and `resize` methods:

```
class Circle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}  
  
class Rectangle extends GraphicObject {  
    void draw() {  
        ...  
    }  
    void resize() {  
        ...  
    }  
}
```

Summary

- ▶ Looked at how patterns are used in software development in general
- ▶ Explored some common software development patterns
- ▶ For Analysis patterns see the Type diagrams lecture
- ▶ Design patterns
 - ▶ Template method pattern
 - ▶ Can be used in many different contexts
 - ▶ We will look at another design pattern (the Composite pattern) in the class in week 9

Further reading

- ▶ Refactoring and Aggregation (1993), by Ralph E. Johnson , William F. Opdyke. In Object Technologies for Advanced Software, First JSSST International Symposium, volume 742 of Lecture Notes in Computer Science
- ▶ See Bennett Chapter 15 – Design Patterns



Software design: architectural design

- Packages, sub-systems and models
- Architectural styles

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Last week

- ▶ ... we looked at the detailed design of the system
- ▶ This week we will look at **system design**, and specifically the high-level architecture of the system
 - ▶ structure of sub-systems
 - ▶ distribution of sub-systems on processors
 - ▶ communication between sub-systems



Subsystems

- ▶ A subsystem typically groups together elements of the system that share some common properties
- ▶ An object-oriented subsystem encapsulates a coherent set of responsibilities in order to ensure that it has integrity and can be maintained
- ▶ For example, the elements of one subsystem might all deal with the human–computer interface, the elements of another might all deal with data management and the elements of a third may all focus on a particular functional requirement.



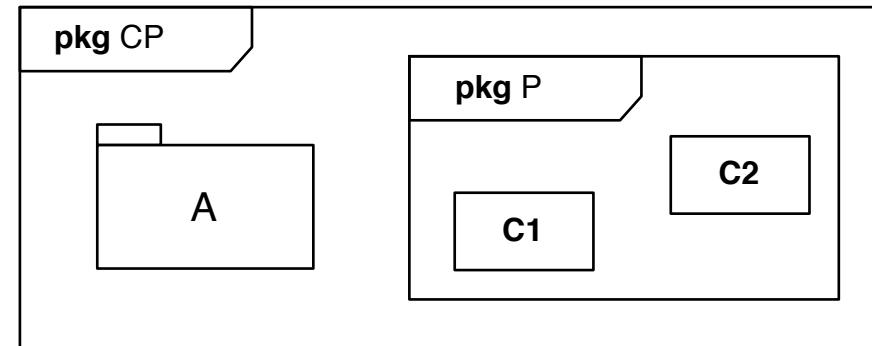
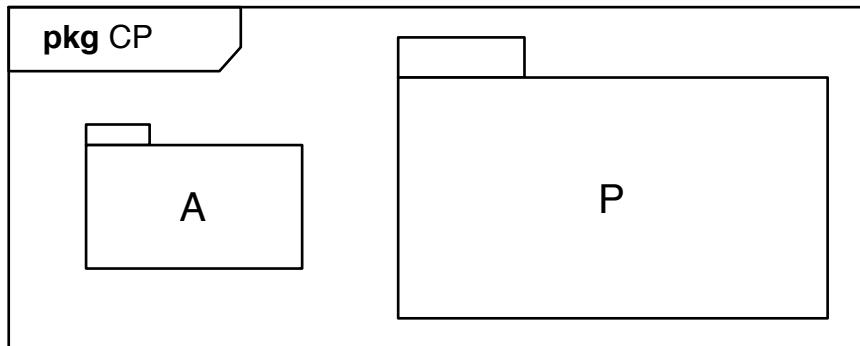
Subsystems

- ▶ The subdivision of an information system into subsystems has the following advantages
 - ▶ It produces smaller units of development
 - ▶ It helps to maximize reuse at the component level
 - ▶ It helps the developers to cope with complexity
 - ▶ It improves maintainability
 - ▶ It aids portability



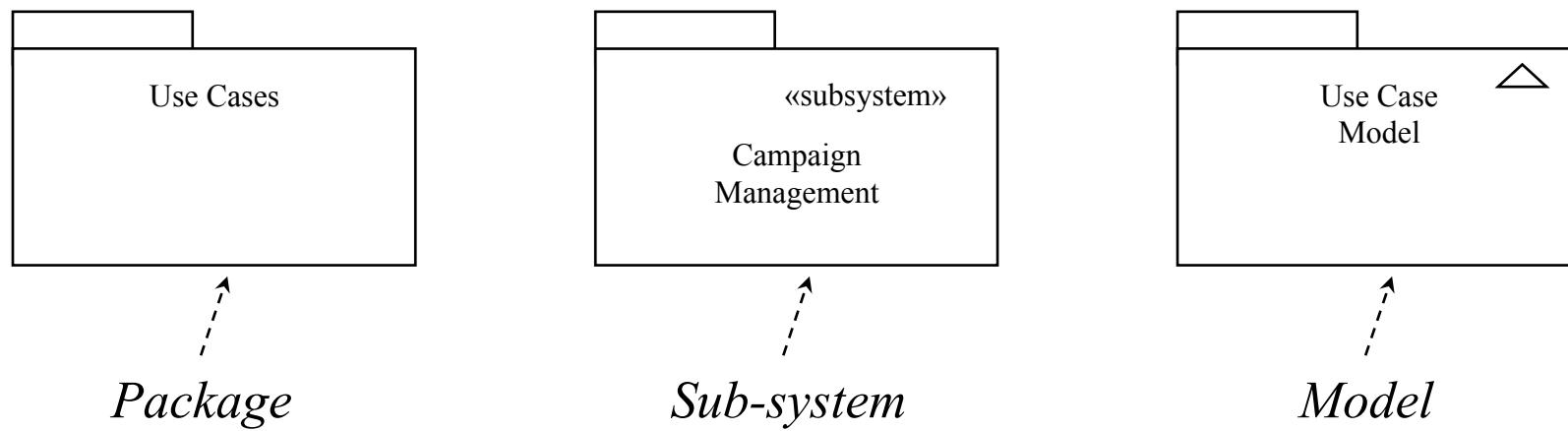
Nested Diagrams

- ▶ UML 2 supports nested diagrams
 - ▶ e.g. an activity diagram inside a class
- ▶ The frame concept visually groups elements that belong to one diagram

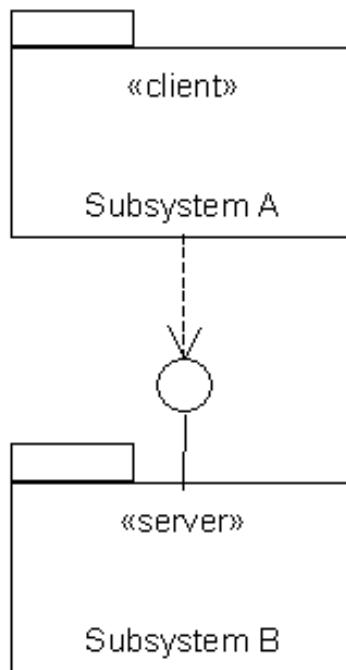


Packages, Sub-systems and Models

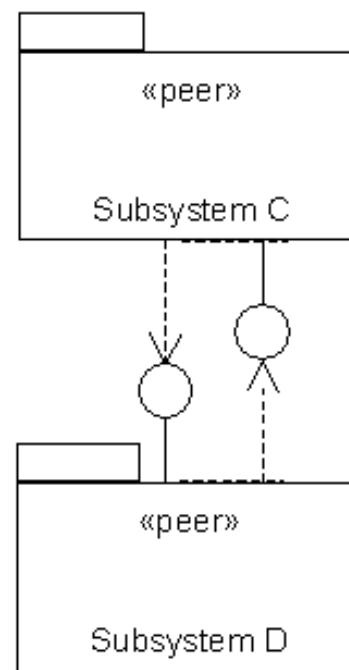
- ▶ UML has notation for showing subsystems and models, and also for packages, which are a mechanism for organising models (e.g. in CASE tools)



Styles of communication between subsystems



The server subsystem does not depend on the client subsystem and is not affected by changes to the client's interface.



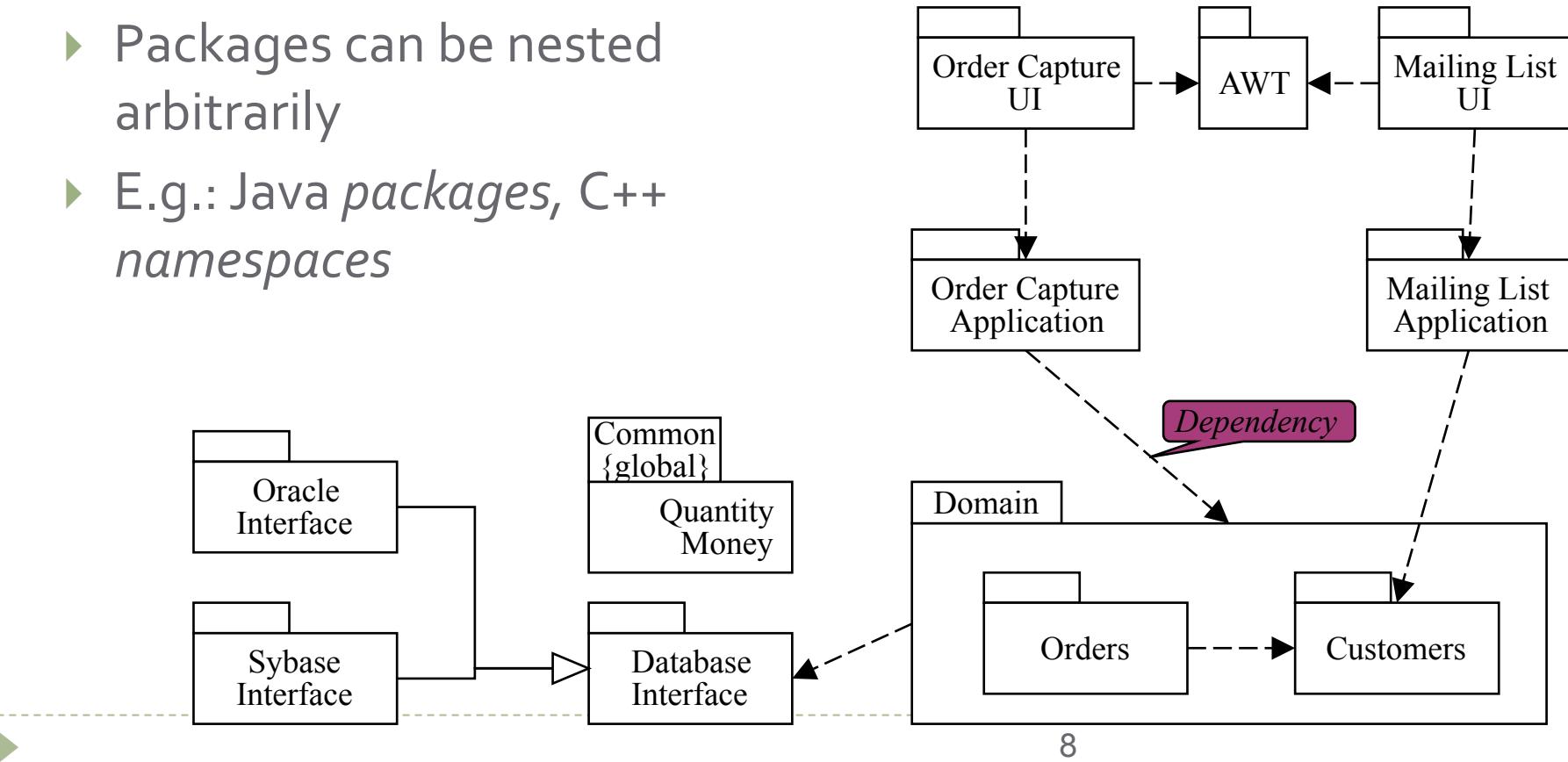
Each peer subsystem depends on the other and each is affected by changes in the other's interface.



System design modelling technique:

Package Diagrams

- ▶ Depicts the dependencies between the packages that make up a model.
- ▶ Package: A module containing any number of classes
 - ▶ Packages can be nested arbitrarily
 - ▶ E.g.: Java *packages*, C++ *namespaces*



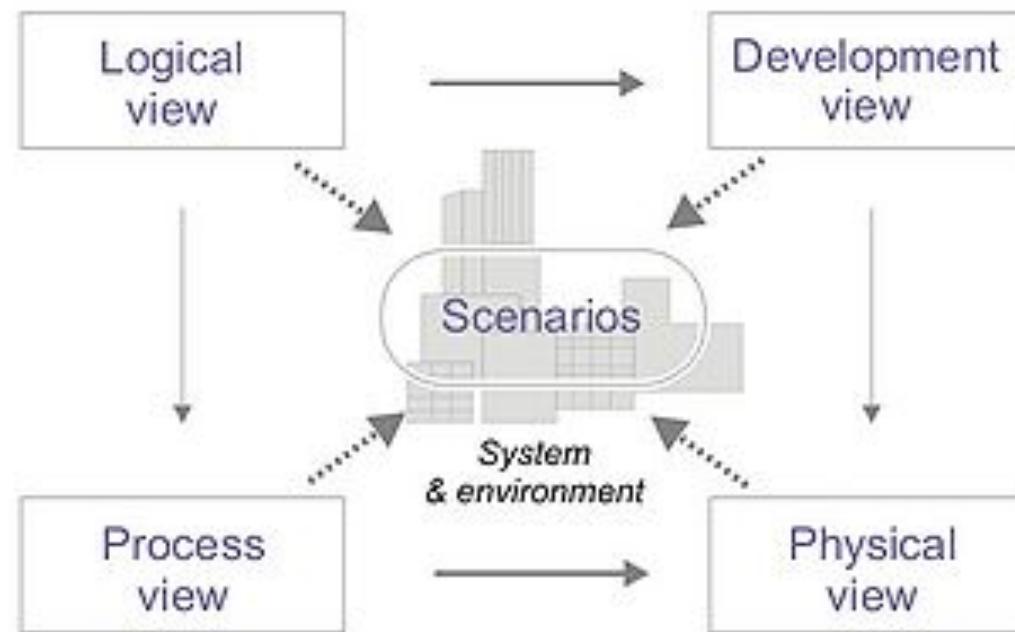
Last week we looked at

- ▶ Difference between **Logical** and **Physical** Design
 - ▶ Difference between **System** Design and **Detailed** Design
-
- ▶ One way of bringing these together to describe the architecture of software-intensive systems is the **4+1 Architectural View** (next slide)

The 4+1 architectural view model

4+1 is a view model for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views" [Kruchten, 1995]

Selected use cases or scenarios are utilized to illustrate the architecture serving as the 'plus one' view



The 4+1 architectural view UML models

- ▶ **Logical view:** concerned with the functionality that the system provides to end-users. UML Diagrams used to represent the logical view include **Class diagram, Communication diagram, Sequence diagram**.
- ▶ **Development view:** illustrates a system from a programmer's perspective and is concerned with software management (also known as the implementation view). It uses UML **Component and Package diagrams**.
- ▶ **Process view:** deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behaviour of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML Diagrams to represent process view include the **Activity diagram**.
- ▶ **Physical view:** depicts the system from a system engineer's point-of-view. It is concerned with the topology of software components on the physical layer, as well as the physical connections between these components (also known as the deployment view). UML Diagrams used to represent physical view include the **Deployment diagram**.
- ▶ **Scenarios :** The description of an architecture is illustrated using a small set of use cases, or scenarios which become a fifth view. The scenarios describe sequences of interactions between objects, and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as **use case view**.

System Architecture in UML

ADL

- ▶ **Architecture Description Language**
 - ▶ UML 2.0 added or changed features to support modeling architecture
 - ▶ Package diagrams (see previous slides)
 - ▶ Component diagrams (not covered)
 - ▶ Composite structure diagrams (not covered)
 - ▶ Deployment diagrams (not covered)



Architectural Styles

Components and connectors
Garlan & Shaw's catalogue

Software architecture: definition

- ▶ [Garlan & Shaw 93]:

“... a level of design concerned with issues ... beyond the algorithms and data structures of the computation; designing and specifying the overall system structure emerges as a new kind of problem.”

“Structural issues include gross organization and global control structure; protocols for communication, synchronization, and data access; assignment of functionality to design elements; physical distribution; composition of design elements; scaling and performance; and selection among design alternatives.”

System Architecture

Key Definitions

- ▶ *System* is a set of components that accomplishes a specific function or set of functions.
- ▶ *Architecture* is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- ▶ *Architectural Description* is a set of products that document the architecture.



System Architecture

Key Definitions

- ▶ *Architectural View* is a representation of a particular system or part of a system from a particular perspective.
- ▶ *Architectural Viewpoint* is a template that describes how to create and use an architectural view. A viewpoint includes a name, stakeholders, concerns addressed by the viewpoint, and the modelling and analytic conventions.

(Garland & Anthony, 2003 & IEEE, 2000)



What is an Architectural Style?

"An architectural style defines a family of systems in terms of a pattern of structural organization." [Shaw & Garlan 93]

Architectural style: describe –

- ▶ Types of **components**
- ▶ Types of **connectors**
- ▶ **Topology** (constraints on possible compositions)

Example: Shared Repository

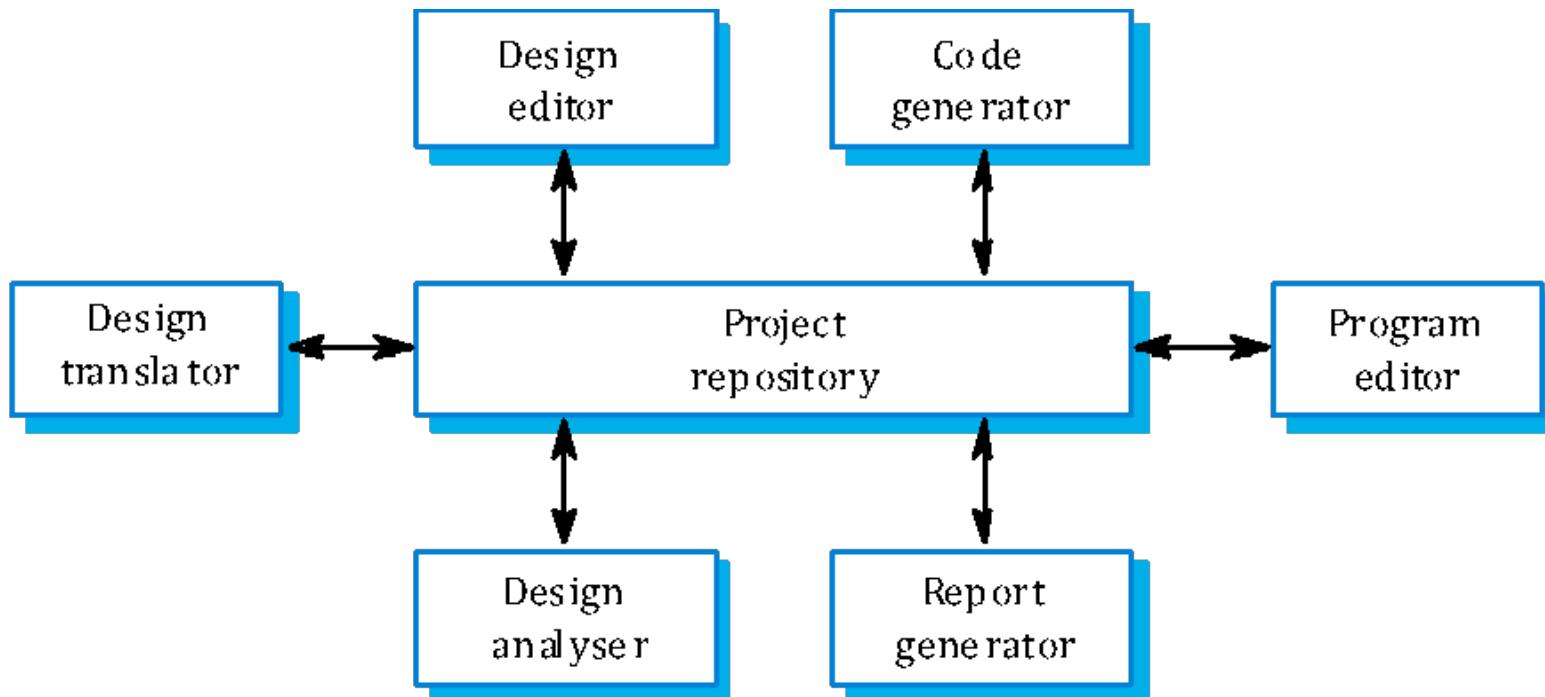
A central database is shared by different subsystems

Applicability:

- ▶ Large, complex body of information
 - ▶ Inter-related records
- ▶ Sophisticated data processing and management
 - ▶ Selection queries, recovery, locking, security ...
- ▶ Distributed systems with shared data

Shared Repository: Example

► CASE Toolset



Shared Repository: Discussion

▶ Pros:

- ▶ Efficient use of resources
- ▶ Consistency maintained
- ▶ Uniform data format
- ▶ Centralized control
 - ▶ Security is simpler

▶ Cons:

- ▶ Evolution is difficult
- ▶ Dependency on data traffic

Client-Server

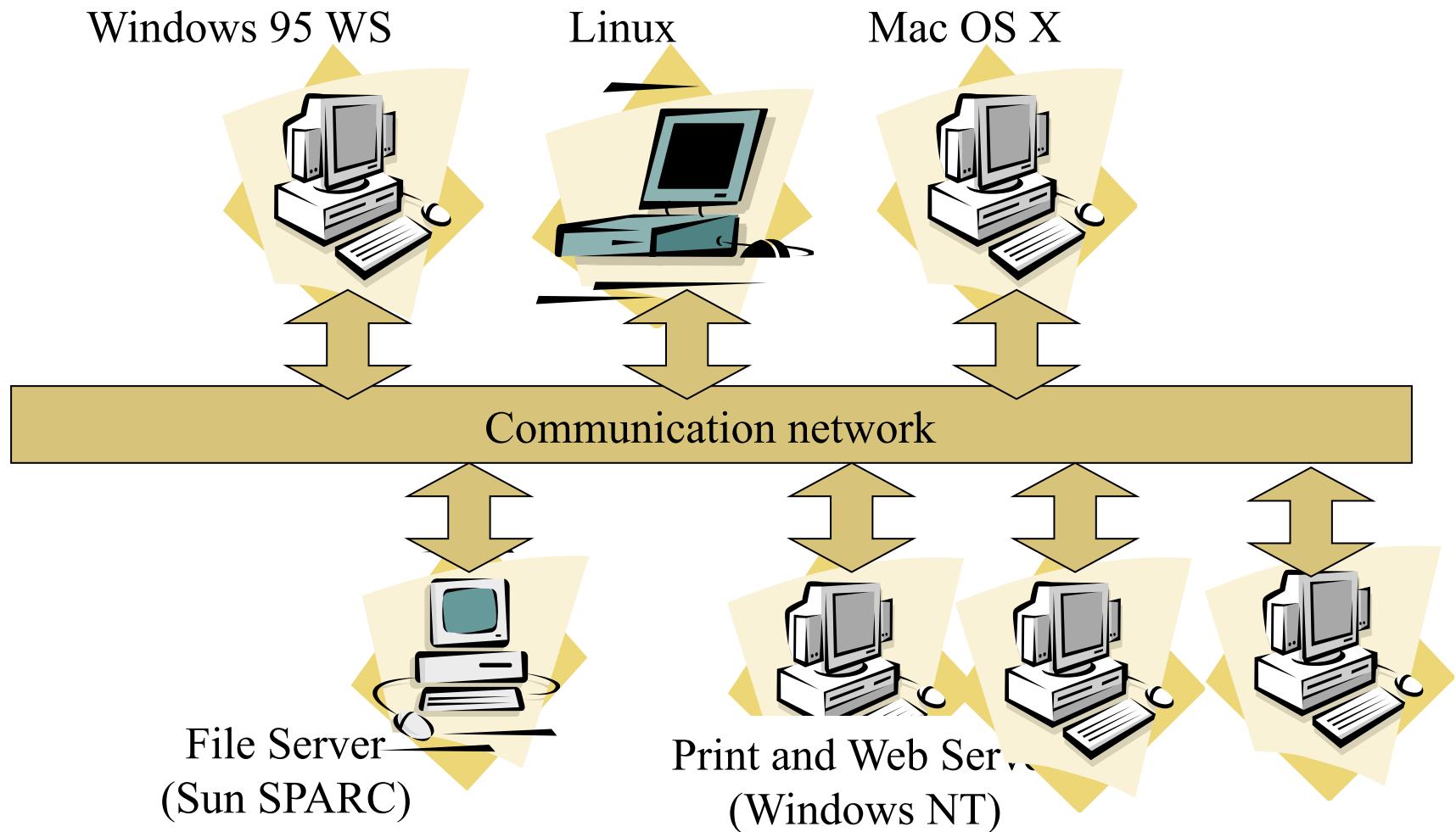
- ▶ A distributed system with --
 - ▶ Server(s): Stand-alone, powerful machines
 - ▶ Clients: Subsystem which employ the services simultaneously
 - ▶ Network: Communicating clients with servers
- ▶ Processing model:
 - ▶ Client initiates request
 - ▶ Request passes through network
 - ▶ Server responds
 - ▶ Response returned via network

Client–server communication

- ▶ Client–server communication requires the client to know the interface of the server subsystem, but the communication is only in one direction
- ▶ The client subsystem requests services from the server subsystem and not vice versa

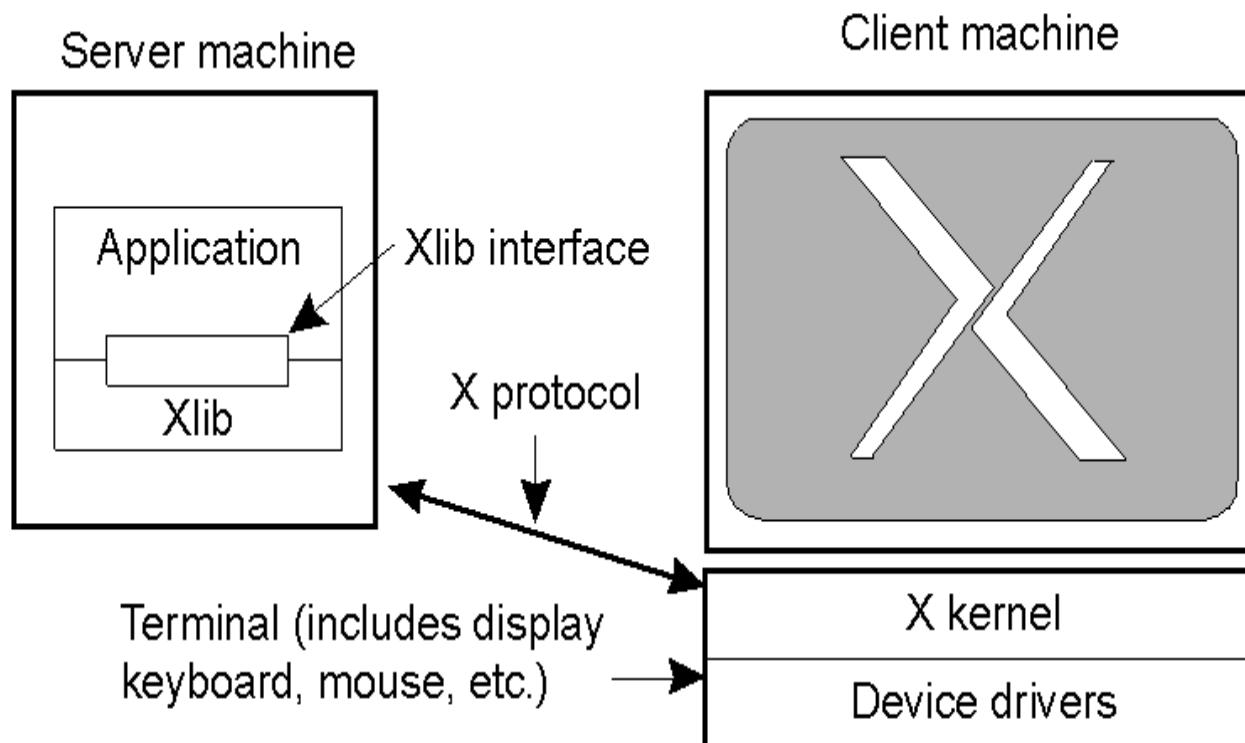


Client Server: Example 1



Client-Server: Example 2

► X-Windows



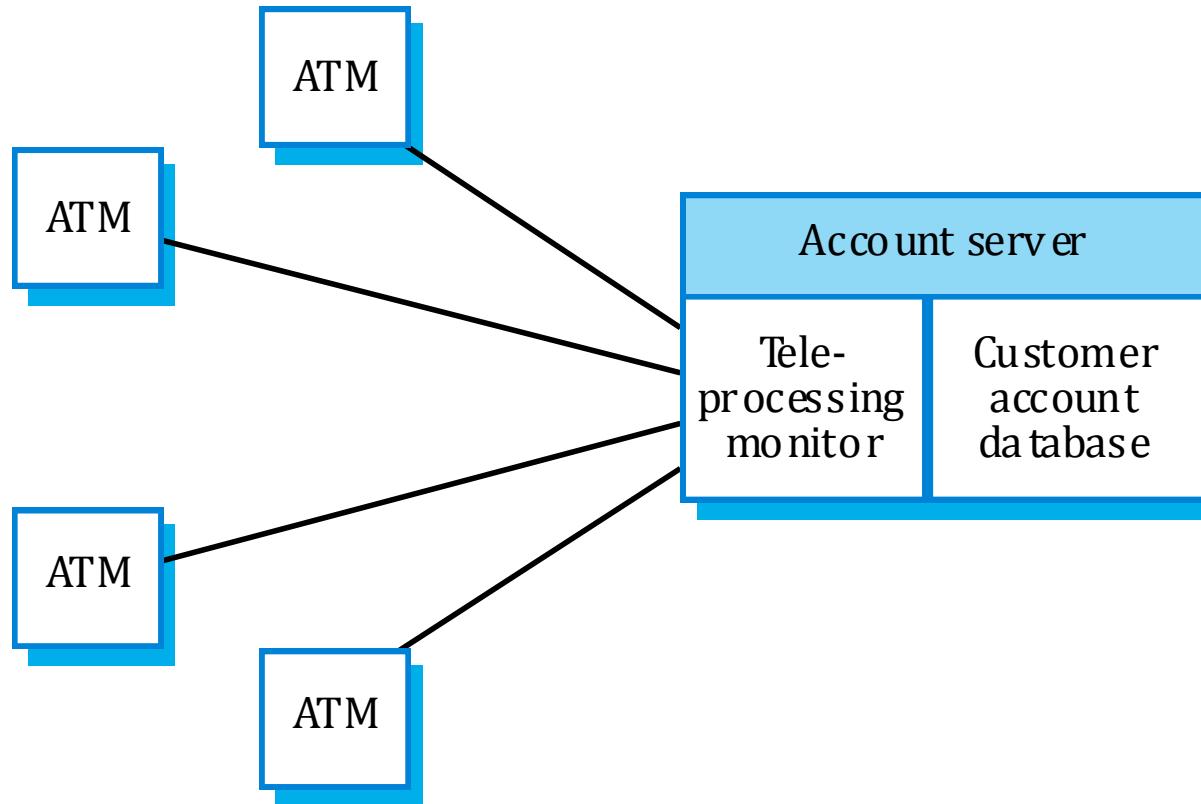
Client-Server: Example 3

► Email

- ▶ Server: MS Exchange Server on exchange3.essex.ac.uk
 - ▶ The domain resolves to more than one machine
- ▶ Clients: Outlook, Thunderbird, Outlook Express, Eudora, ...
- ▶ Protocols
 - ▶ Post Office Protocol Ver. 3 (POP3)
 - ▶ Internet Message Access Protocol (IMAP)
 - ▶ Exchange Server

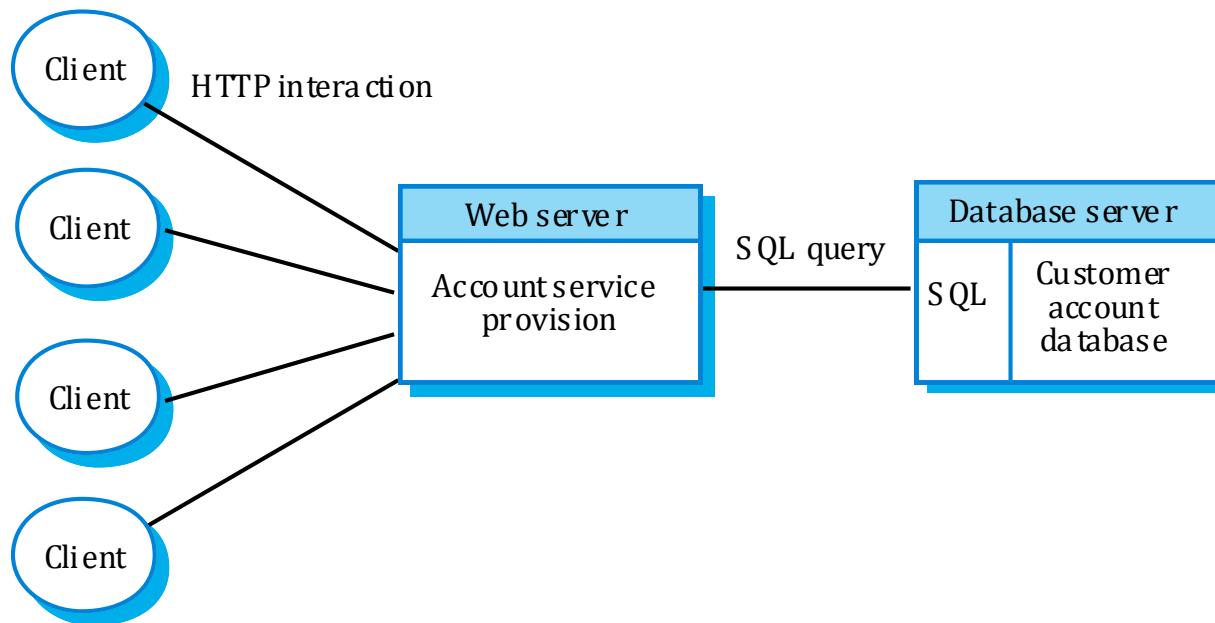
Client-Server: Example 4

► Automated teller machine



Client-Server: Example 5

- ▶ Three-tier client-server architecture
 - ▶ Customary in Web configurations such as Java Server Pages (JSP)



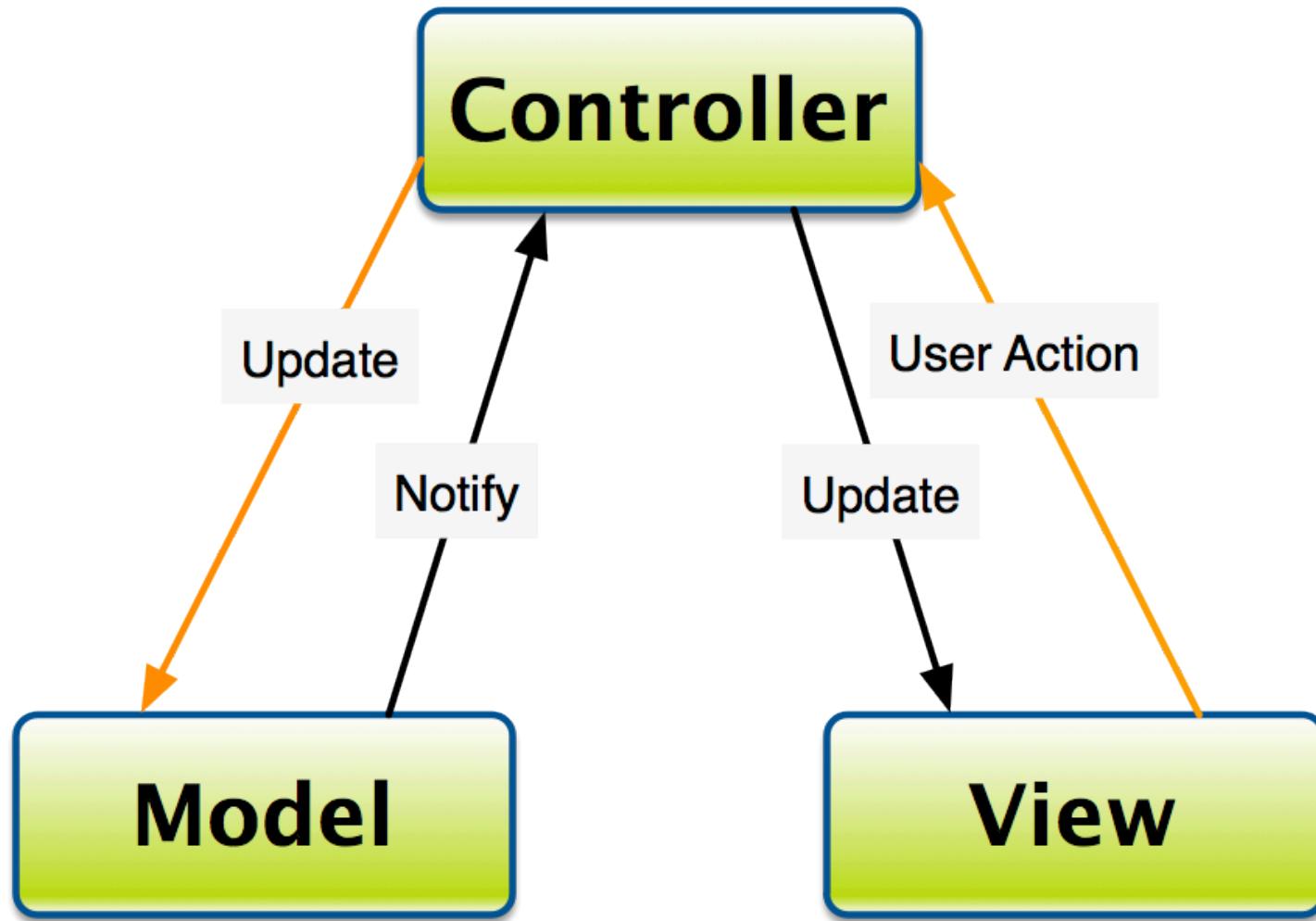
Client-Server: Properties

- ▶ Transparency
 - ▶ Possible distribution of server on different machines without clients' dependency
 - ▶ Clients know servers, not vice versa
- ▶ Heterogeneity
 - ▶ Allowing for different machines, operating systems
- ▶ Redundancy

Side note: Redundancy

- ▶ In engineering, redundancy is the duplication of critical components or functions of a system with the intention of increasing reliability of the system, usually in the case of a backup or fail-safe.
- ▶ There are four major forms of redundancy, these are:
 - ▶ Hardware redundancy, such as DMR and TMR (Dual and Triple Modular Redundant).
 - ▶ -- A machine which is Dual Modular Redundant has duplicated elements which work in parallel to provide one form of redundancy.
 - ▶ Information redundancy, such as error detection and correction methods
 - ▶ Time redundancy, including transient fault detection methods such as Alternate Logic
- ▶ ²⁹ Software redundancy such as N-version programming

Architectural Style: Model-View-Controller



Model-View-Controller

- ▶ **Model**—provides the central functionality of the application and is aware of each of its dependent view and controller components. Similar to UML Entity stereotype
- ▶ **View**—corresponds to a particular style and format of presentation of information to the user. The view retrieves data from the model and updates its presentations when data has been changed in one of the other views. The view creates its associated controller. Similar to UML Boundary stereotype



Model-View-Controller

- ▶ **Controller**—accepts user input in the form of events that trigger the execution of operations within the model. These may cause changes to the information and in turn trigger updates in all the views ensuring that they are all up to date.
Similar to UML Control stereotype
- ▶ Maps to UML stereotypes: Entity, Boundary, Control
- ▶ Maps to 3 tier client-server architecture

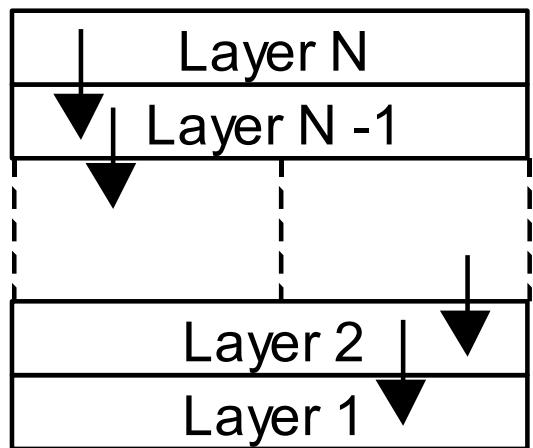


Layering and Partitioning

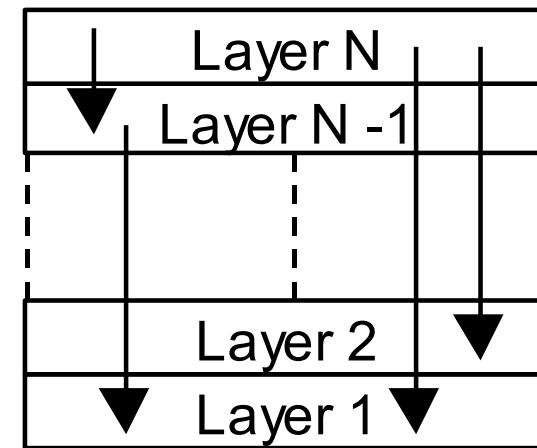
- ▶ Two general approaches to the division of a software system into subsystems
 - ▶ *Layering*—so called because the different subsystems usually represent different levels of abstraction
 - ▶ *Partitioning*, which usually means that each subsystem focuses on a different aspect of the functionality of the system as a whole
- ▶ Both approaches are often used together on one system



Schematic of a Layered Architecture



*Closed architecture—
messages may only be
sent to the adjacent
lower layer.*



*Open architecture—
messages can be sent
to any lower layer.*

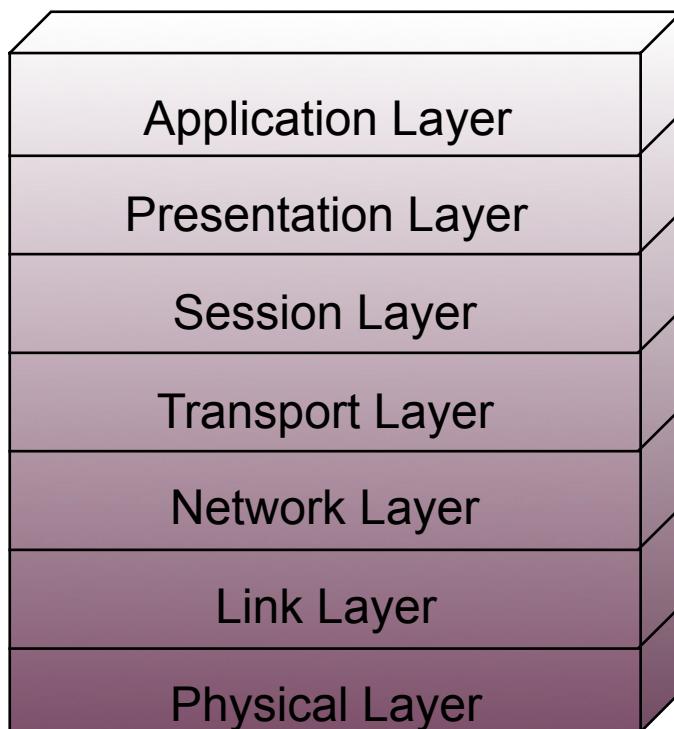
Layered Architecture

- ▶ A closed architecture **minimizes dependencies** between the layers and reduces the impact of a change to the interface of any one layer
- ▶ An open layered architecture produces more compact code, the services of all lower level layers can be accessed directly by any layer above them without the need for extra program code to pass messages through each intervening layer, but this breaks the encapsulation of the layers

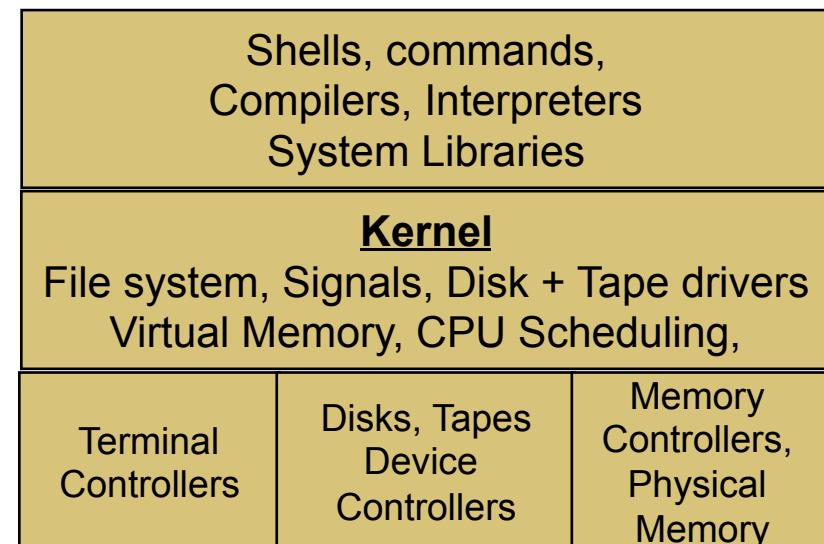


Layered Architecture: examples

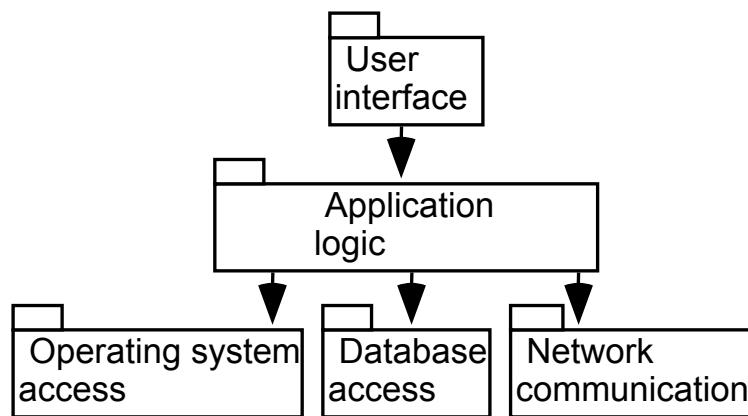
ISO Communication Protocols



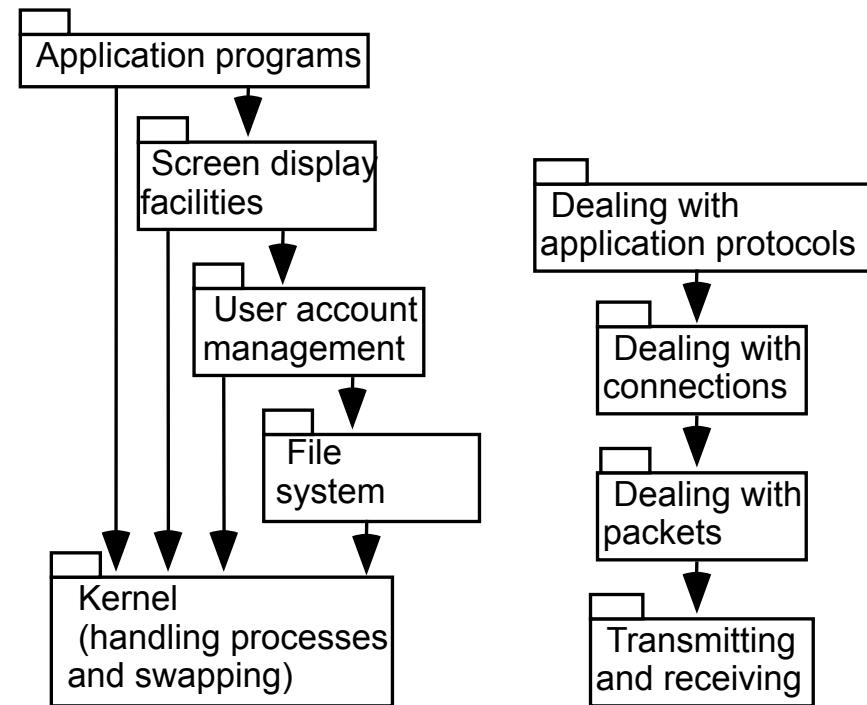
UNIX Operating System



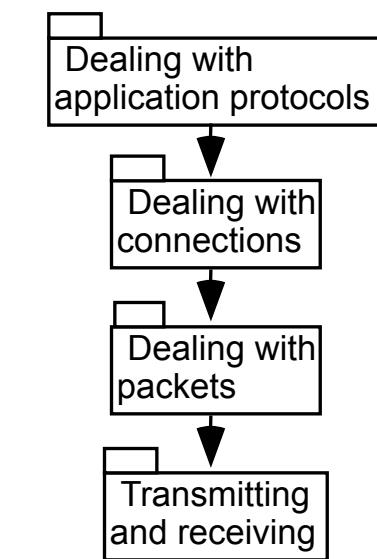
Layered Architecture: examples II



a) Typical layers in an application program

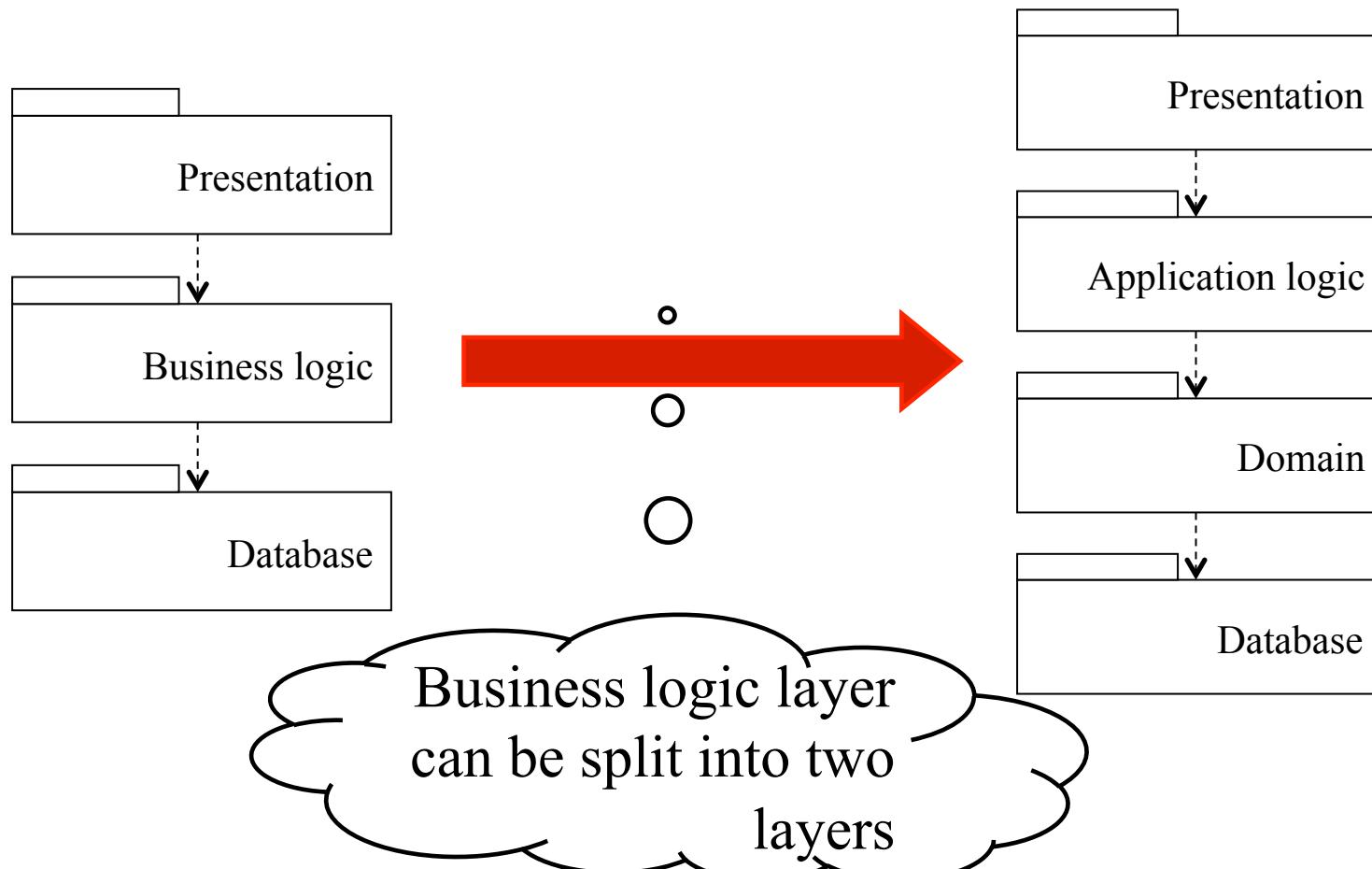


b) Typical layers in an operating system



c) Simplified view of layers in a communication system

Three & Four Layer Architectures.



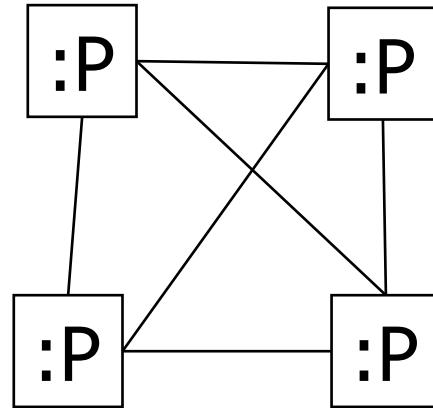
Peer-to-peer (P2P) communication

- ▶ Peer-to-peer communication requires each subsystem to know the interface of the other, thus coupling them more tightly
- ▶ The communication is two way since either peer subsystem may request services from the other

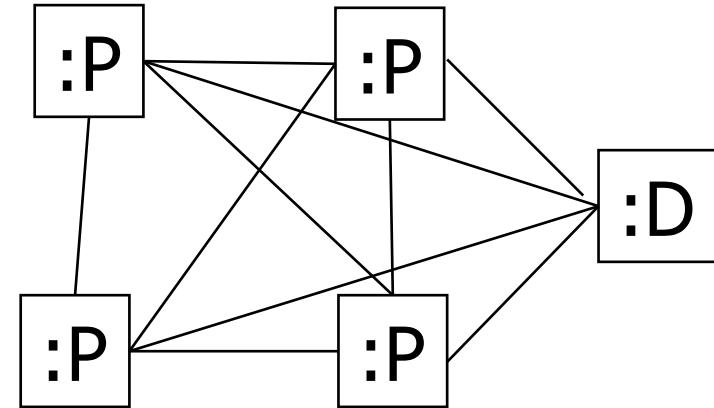


Peer-to-Peer (P2P)

- ▶ System composed of any number of peers
- ▶ Peer: client & server
 - ▶ Computations may be performed by any peer
- ▶ Peers popping in and out of existence



Pure P2P



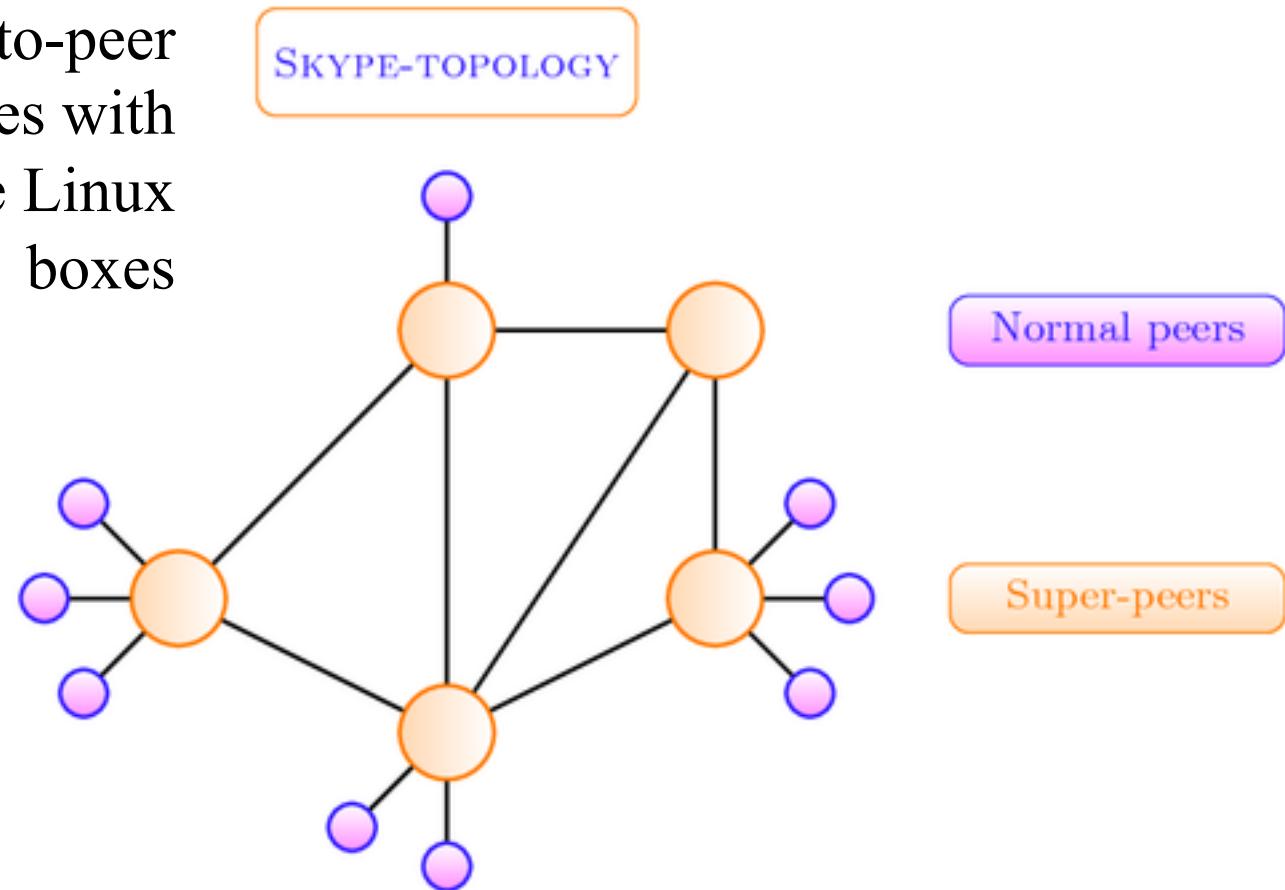
P2P with a discovery server

P2P: Examples I

- ▶ **File sharing**
 - ▶ Napster, Gnutella, Kazaa, torrent applications...
- ▶ **Messenger**
 - ▶ Skype, ICQ, Yahoo Messenger, MSN Messenger ...
 - ▶ NB. Skype is a hybrid peer to peer ie. has server
- ▶ **JXTA**
 - ▶ XML encoding of messages
 - ▶ Resources found via “advertisement”
 - ▶ Requires a “discovery” server

P2P example: Skype

Replacing peer-to-peer client machines with thousands of secure Linux boxes

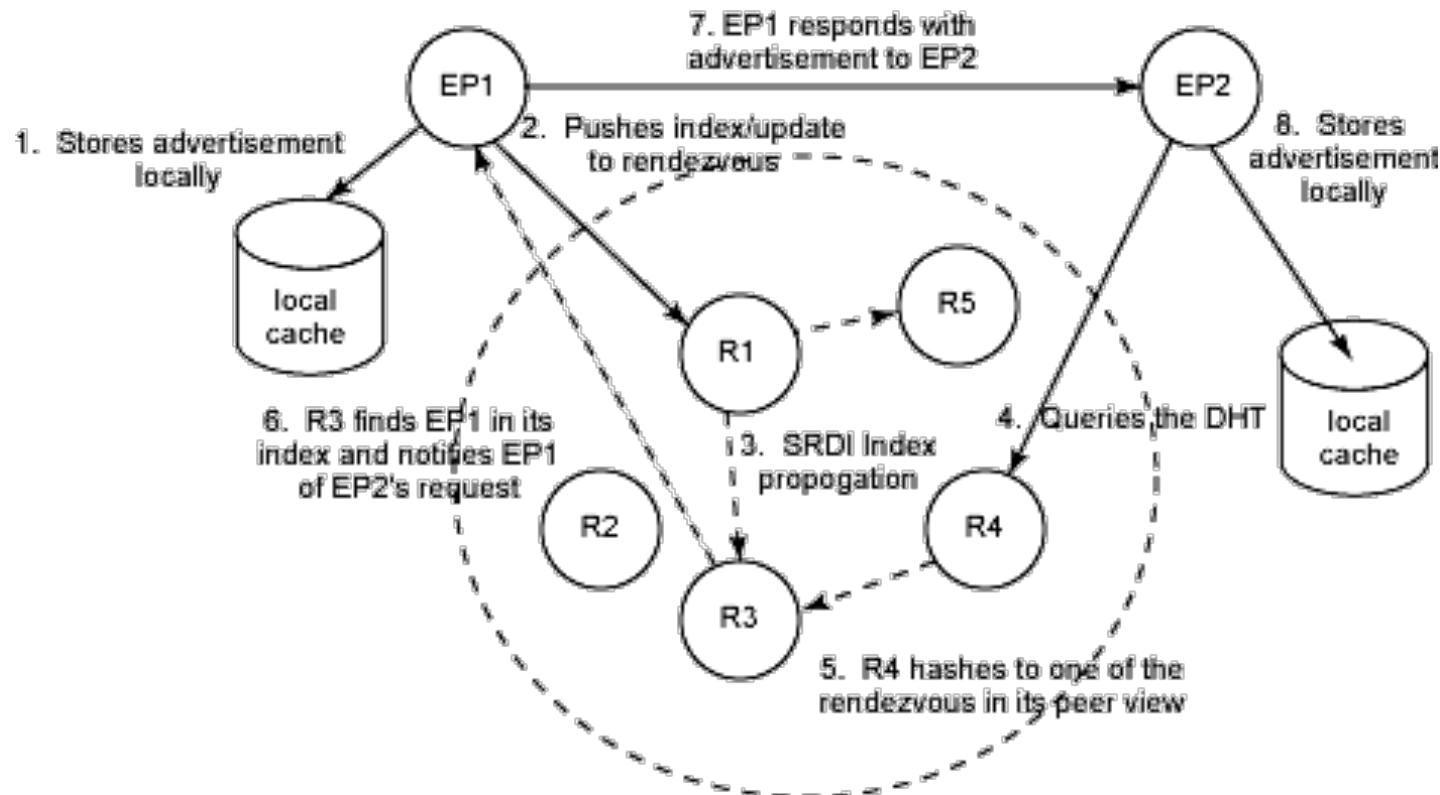


A note on Skype

- ▶ Since its introduction in 2003, the network has consisted of "supernodes" made up of regular users who had sufficient bandwidth, processing power, and other system requirements to qualify. These supernodes then transferred data with other supernodes in a peer-to-peer fashion. At any given time, there were typically a little more than 48,000 clients that operated this way.
- ▶ Skype is now being powered by more than 10,000 supernodes that are all hosted by the company.
- ▶ It's currently not possible for regular users to be promoted to supernode status. What's more, the boxes are running a version of Linux using grsecurity, a collection of patches and configurations designed to make servers more resistant to attacks.
- ▶ The banishment of user-supplied supernodes comes as the number of people simultaneously signed into Skype has mushroomed.
- ▶ According to Skype that number hit 41 million in April 2012, a 37-percent jump from the average number of concurrent users when Microsoft acquired Skype in 2011.

P2P: Advertisement and Discovery

Advertisement & discovery in JXTA2



Steps in previous diagram

- ▶ 1. EP₁ creates a pipe and stores its advertisement locally.
- ▶ 2. The index is updated locally and the changes are pushed to the connected rendezvous (R₁).
- ▶ 3. Through a shared resource distributed (SRDI) index propagation, the rendezvous receiving the update (R₁) replicates the new index information to the selected rendezvous (R₃ and R₅) in the super-peer network
- ▶ 4. Later, EP₂ queries for EP₁'s pipe. This query is sent to its only connected rendezvous -- R₄.
- ▶ 5. R₄ hashes the query's attributes and redirects the request to another rendezvous in the super-peer network -- R₃.
- ▶ 6. R₃, having received EP₁'s index update through R₁, immediately notifies EP₁ of EP₂'s request.
- ▶ 7. EP₁ sends a response directly to EP₂ containing the requested pipe advertisement. At this point, the query is resolved.
- ▶ 8. EP₂ may in turn decided to store the pipe advertisement, and the cycle begins again.

A note on JXTA

- ▶ The JXTA protocols defines a suite of six XML-based protocols that standardize the manner in which peers self-organize into peer groups, publish and discover peer resources, communicate, and monitor each other.
- ▶ The Endpoint Routing Protocol (ERP) is the protocol by which a peer can discover a route (sequence of hops) to send a message to another peer potentially traversing firewalls and NATs.
- ▶ The Rendezvous Protocol (RVP) is used for propagating a message within a peer group.
- ▶ The Peer Resolver Protocol (PRP) is the protocol used to send a generic query to one or more peers, and receive a response (or multiple responses) to the query.
- ▶ The Peer Discovery Protocol (PDP) is used to publish and discover resource advertisements.
- ▶ The Peer Information Protocol (PIP) is the protocol by which a peer may obtain status information about another peers.
- ▶ The Pipe Binding Protocol (PBP) is the protocol by which a peer can establish a virtual communication channel or pipe between one or more peers.
- ▶ The JXTA protocols permit the establishment a virtual network overlay on top of physical networks allowing peers to directly interact and organize independently of their network location and connectivity. The JXTA protocols have been designed to be easily implemented on unidirectional links and asymmetric transports.
- ▶ To create a Distributed Hash Table (DHT), each peer caches advertisements locally, and all locally stored advertisements are indexed.
- ▶ The index is pushed to the rendezvous node (JXTA 2 edge peer connects to only one rendezvous node at any time).
- ▶ The rendezvous super-peer network maintains the DHT containing the amalgamated indices. Queries are always sent to a rendezvous, as illustrated in the previous diagram

P2P: Discovery models

- ▶ Q. How peers are found?
- ▶ A. Discovery models
 - ▶ Some peers offer a directory service
 - ▶ “super peers”
 - ▶ rendezvous peers
 - ▶ Relay peers
 - ▶ Multicast – peers announce their presence on some communication channel
 - ▶ Neighbour model – via other peers

P2P: Discussion

▶ Pros:

- ▶ Very flexible
- ▶ Evolution is with minimal effort
- ▶ Efficient – resource utilization is close to optimal
- ▶ Heterogeneous

▶ Cons:

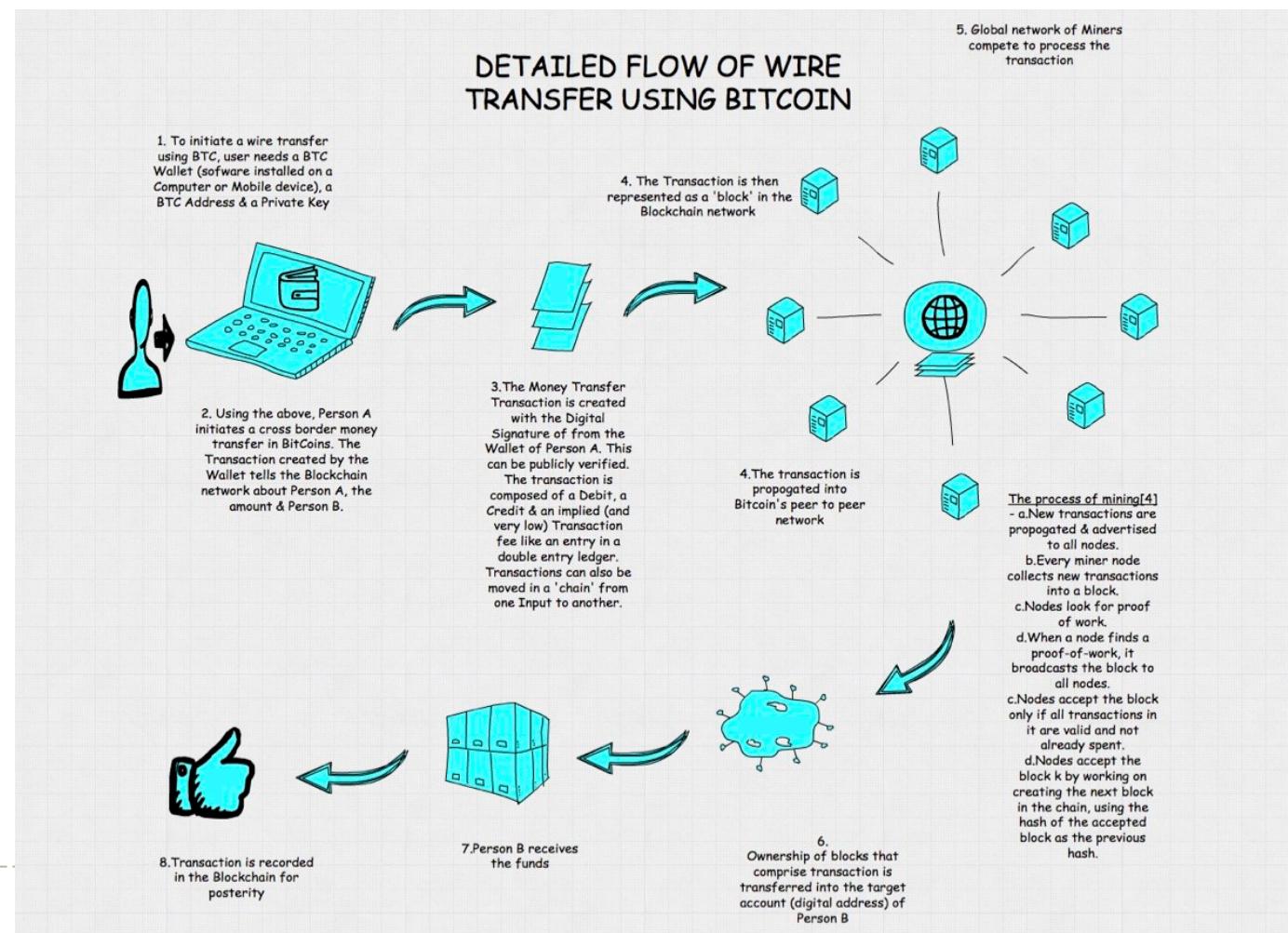
- ▶ Security: authentication is close to impossible (however, see next slide)
- ▶ Duplication
- ▶ Data corruption

Final P2P example: Blockchain

- ▶ Blockchain: *a distributed cryptographic ledger shared amongst all nodes participating in the network, over which every successfully performed transaction is recorded*

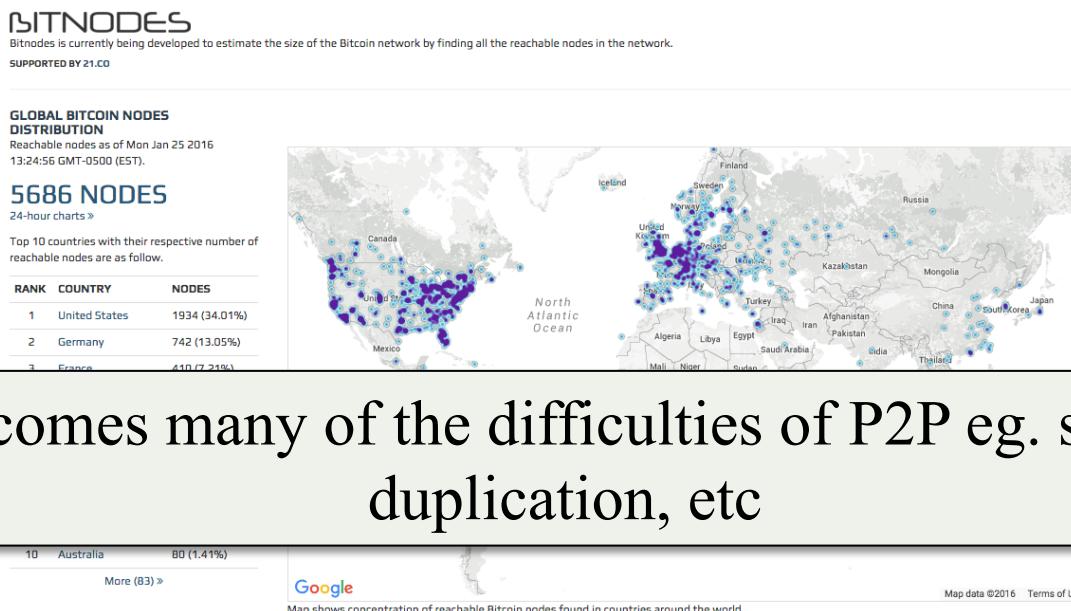
Example:
Bitcoin
A true P2P
distributed
economy

P2P with
strong inbuilt
cryptography



Blockchain P2P nodes

- ▶ The nodes in the BitCoin blockchain play the role of a **Central Bank** or a trusted third party. Every node maintains a **fully replicated copy** of a database that contains the payment history of every bitcoin ever created along with ownership information. As transactions happen using the currency, a **consensus mechanism** essentially dictates how nodes agree on blockchain updates.



Overcomes many of the difficulties of P2P eg. security, duplication, etc

Summary

- ▶ How to model sub-systems using nested diagrams in UML and Package diagrams
- ▶ Modeling the system architecture based on the 4+1 architectural view
- ▶ UML Architectural Description Language (ADL)
- ▶ The system architecture is defined during the system design process
- ▶ Some example architectural styles
 - ▶ Shared repository
 - ▶ Client/Server
 - ▶ Model/View/Controller
 - ▶ Layered architectures
- ▶ 51
▶ P2p

Further reading

- ▶ Bennett, 8.3.4 Packages and dependencies
- ▶ Garlan & Shaw (1993). “**An Introduction to Software Architecture**”. In “An Introduction to Software Architecture,” *Advances in Software Engineering and Knowledge Engineering, Volume I*, edited by V.Ambriola and G.Tortora, World Scientific Publishing Company, New Jersey, 1993.
- ▶ Ian **Sommerville** 2004 Software Engineering, 7th edition.
- ▶ Chapter 9: Architecting and designing software, Lethbridge/Laganière 2001
- ▶ Bennett, Chapter 13, System Design and Architecture
- ▶ JXTA - <http://www.ibm.com/developerworks/java/library/j-jxta2/>
- ▶ Kruchten, Philippe (1995, November). Architectural Blueprints — The “4+1” View Model of Software Architecture. *IEEE Software* 12 (6), pp. 42-50

Exercises

- ▶ Observe the differences between apparently-similar styles: What are the differences between –
 - ▶ Client-Server vs. Broker?
 - ▶ Rendezvous peer vs. broker?
 - ▶ Peer-to-Peer vs. Grid?

Software validation: Introduction

- Basic terminology
 - Static vs. dynamic validation techniques
- Code inspection
- Typing

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

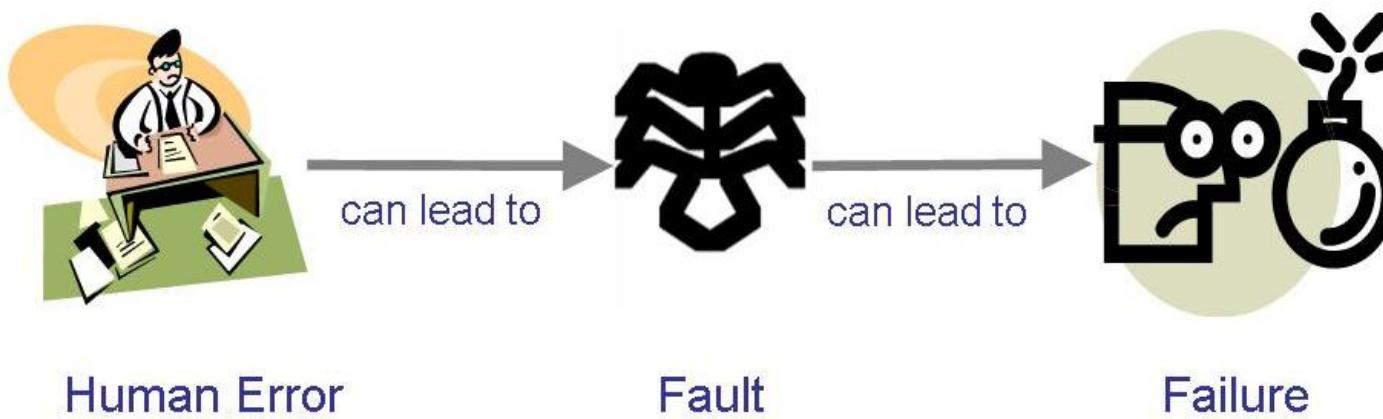


Software verification and validation

- ▶ Checking that the implementation conforms to our expectations
 - ▶ Necessary condition: clear, unambiguous *requirement or product feature specification!*
- ▶ Two variations:
 - ▶ **Verification:** Checking the implementation wrt the specification
 - ▶ **Validation:** Checking the implementation wrt the Client
- ▶ Therefore:
 - ▶ *Validation* is what we “really” want
 - ▶ *Verification* is the 1st step towards *validation*

Terminology: error, fault & failure

- ▶ **Error:** A human mistake in SW development
 - ▶ May lead to one or more *faults*
- ▶ **Fault:** The result of error(s),
 - ▶ May lead to one or more failures
- ▶ **Failure:** The dynamic manifestation of fault(s)
 - ▶ Also: Departure from the required behaviour



Terminology: static vs. dynamic validation

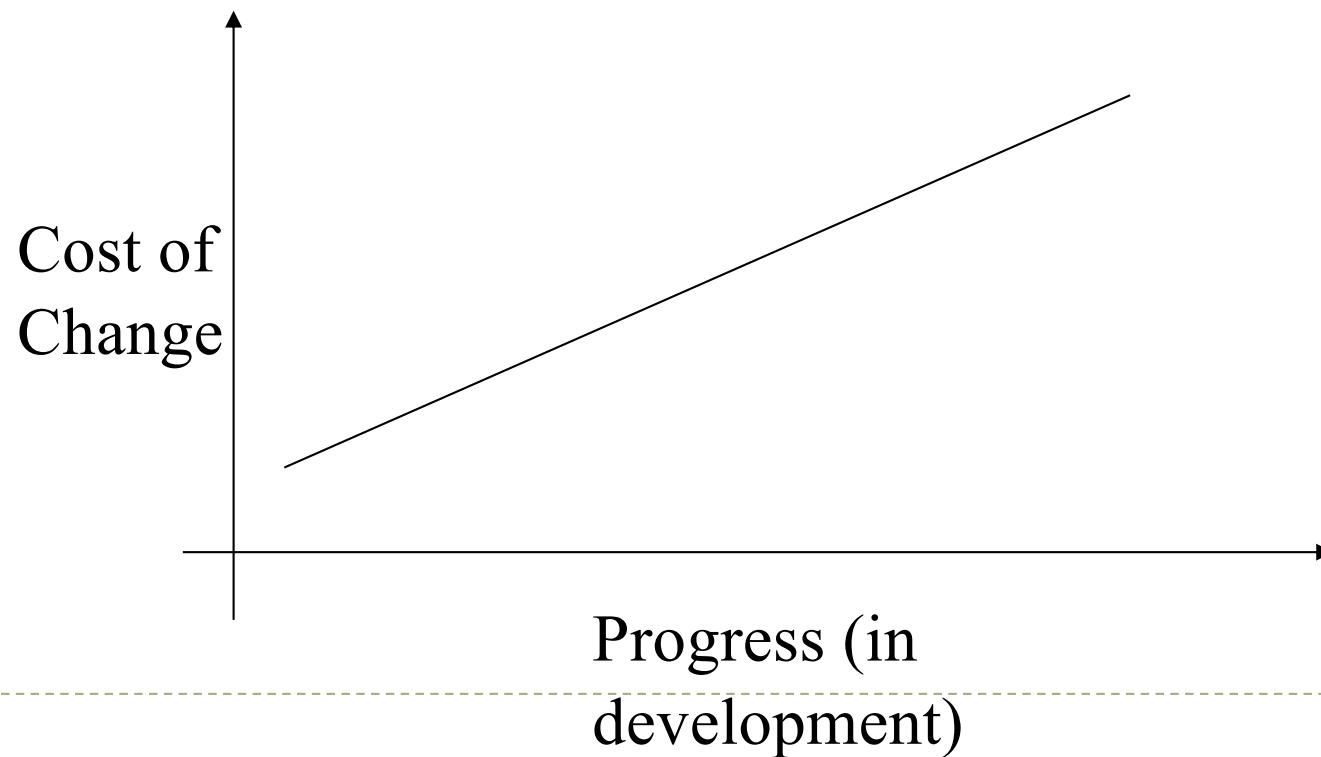
- ▶ **Static validation:** carried out without executing the program
 - ▶ Examples:
 - ▶ software inspection
 - ▶ formal verification
 - ▶ static typing
- ▶ **Dynamic validation:** carried out by executing the program
 - ▶ Examples:
 - ▶ Testing
 - ▶ Defensive programming (design by contract)

Static Vs. Dynamic Techniques

- ▶ Static validation does not require executing the program
- ▶ Pros:
 - ▶ Detection is earlier in the process
 - ▶ The system need not be complete
 - ▶ Correction is easier and cheaper
- ▶ Cons:
 - ▶ Reduced flexibility
 - ▶ Increased “bookkeeping”
 - ▶ Incomplete

Early Detection is Best

- ▶ Earlier changes are cheaper to make
 - ▶ Applies to both corrections and extensions



Software validation: code inspection

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Code Inspection

- ▶ AKA *program inspection*
- ▶ A static validation technique
- ▶ Informal review process carried by peers
 - ▶ Target: Source code
 - ▶ Resources required: Precise specification!
 - ▶ Objective: Detection of faults & anomalies
 - ▶ Local, step by step process
- ▶ Errors discovered:
 - ▶ Use of un-initialized data
 - ▶ Allocation and de-allocation of dynamic memory
 - ▶ Conditional statements that resolve statically
 - ▶ Exception checks
 - ▶ Appropriate handler for each exception
 - ▶ Array bounds
 - ▶ Infinite loops
 - ▶ Method not overridden
 - ▶ ...

Tools supporting inspection

- ▶ Support a “checklist” of faults
- ▶ Examples: Java™ Compiler
 - ▶ Interface errors
 - ▶ typing errors (discussed separately)
- ▶ Example: C/C++ Lint
 - ▶ Variables declared but not used
 - ▶ Use of un-initialized variables
 - ▶ Unreachable code (“code coverage”)
 - ▶ Entry and exit points in loops
 - ▶ ...

LINT Output Sample

```
138% more lint_ex.c
#include <stdio.h>
printarray (int Anarray) {
    printf("%d",Anarray);
}

main () {
    int Anarray[5]; int i; char c;
    printarray (Anarray, i, c);
    printarray (Anarray) ;
}
```

```
139% cc lint_ex.c
140% lint lint_ex.c
```

```
lint_ex.c(10): warning: c may be used before set
lint_ex.c(10): warning: i may be used before set
printarray: variable # of args. lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(10)
printarray, arg. 1 used inconsistently lint_ex.c(4) :: lint_ex.c(11)
printf returns value which is always ignored
```

Software validation: typing

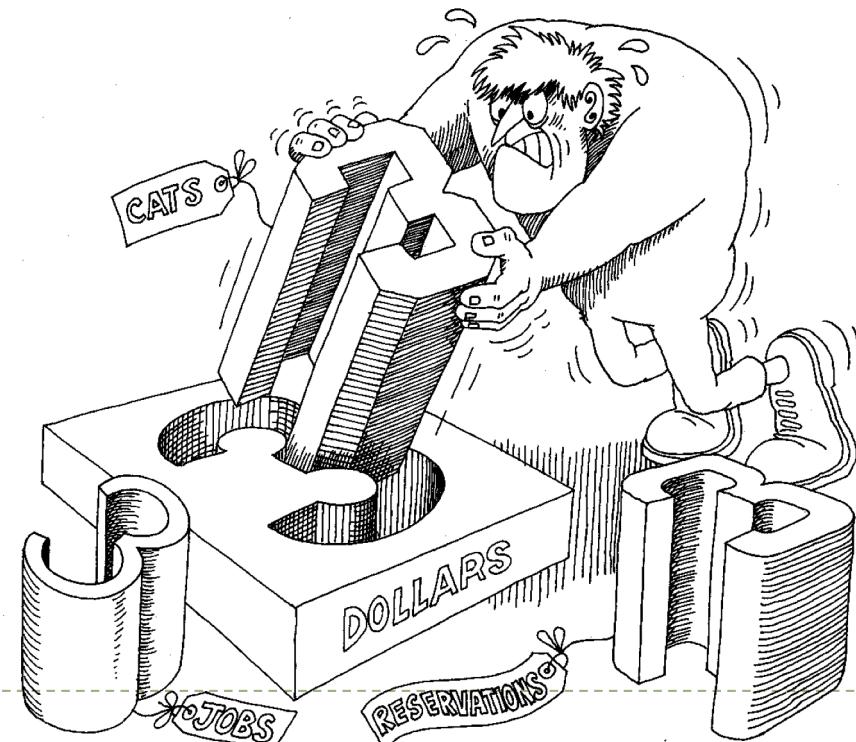
CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



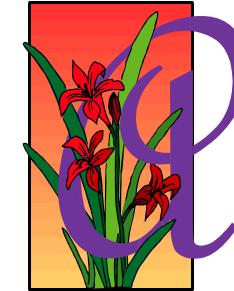
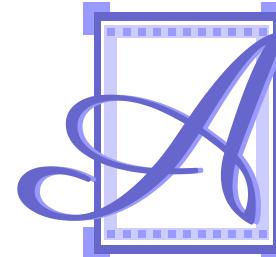
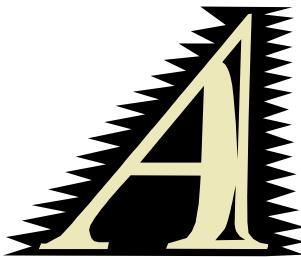
Technique: Typing

- ▶ Supported by the programming language (e.g., Pascal, Ada, Java, C++ – partially)
 - ▶ Reduce the scope for errors
- ▶ Typing techniques:
 - ▶ Static/Dynamic typing
 - ▶ Strong/Weak typing
- ▶ Trade-offs
 - ▶ Safety Vs. Flexibility



Type

- ▶ A set of operations allowed over a group
- ▶ The interpretation of an area in memory
 - ▶ The sequence 0100 means different things if it is a signed integer, unsigned integer, an address, etc.



What is Typed?

- ▶ Anything that has a value
 - ▶ Variables
 - ▶ Expressions
 - ▶ Constants



4

IV

Four

Example: *Type* in C

▶ **short**

- ▶ Representation: Two bytes
- ▶ Operations: All integer operations

▶ **short ***

- ▶ Representation: Address size
- ▶ Operations: All pointer operations

Static Vs. Dynamic Typing

- ▶ ***Static typing:*** *Type checking performed statically*
 - ▶ Done by the compiler
 - ▶ Requires type declaration for every name
 - ▶ Safer
 - ▶ Examples: All strongly-typed languages, C, C++, Eiffel, Java™
- ▶ ***Dynamic typing:*** *Type checking performed dynamically*
 - ▶ Requires *dynamic binding* between the object and the type!
 - ▶ More flexible
 - ▶ Examples: Smalltalk, Lisp
 - ▶ Java: Cast operations are checked dynamically

Example: Static typing in Java

```
class IntStack {  
    public void push(int) { ... }  
    public int pop() {  
        int result;  
        ...  
        return result; // OK: result is of type int  
    }  
...  
}
```

```
class UseStack {  
    public String use(IntStack is, int k) {  
        is.push(k); // OK  
        is.push("3"); // Typing error  
        k.push(3); // Typing error  
        return "3"; // OK  
        return 3; // Typing error  
    }  
}
```

Static typing in Java (Cont.)

- ▶ In OOP: superclass is supertype

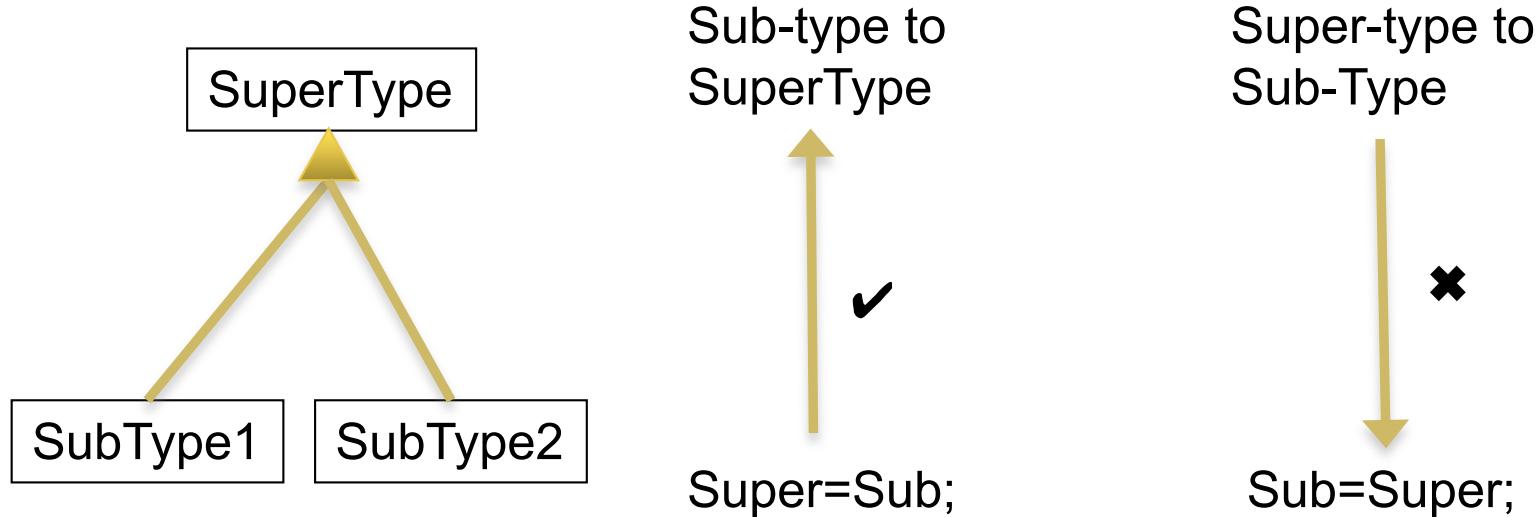
```
class UseStack {  
    public String use(IntStack is) {  
        int sz = is.pop();          // OK  
        is.toString();             // OK  
        Object obj = is;           // OK: Conversion to supertype  
        obj.toString();            // OK  
        obj.push(5);               // Typing error  
        IntStack is2;  
        is2 = obj;                 // Error  
        is2 = (IntStack)obj;       // OK: Casting(will be checked dynamically)  
        String str=(String)obj;   // OK: Casting(will fail in runtime)  
    }  
}
```

Additional notes

- ▶ Remember *obj* is actually an instance of **InstStack**, as it is a pointer to the *is* object
- ▶ *is* is declared to be of class **IntStack**, so *obj* (which is actually an instance of **IntStack**) can be cast to **IntStack** dynamically
- ▶ Therefore, it is not possible for an instance of **IntStack** to be cast to a **String**.
- ▶ To understand this at a fundamental level, think about the *is* object as it would exist in the heap. It has a certain memory layout, with space allocated for the various fields and so on. *is* is a pointer to that memory structure, and then *obj* is created as another pointer to that structure. When you create *is2* as a pointer to that memory structure, the Java VM has to check your cast: so it ensures that the memory structure is type-compatible with an **IntStack** (which it is). However, when you create *str* as a pointer to that memory structure, the VM throws an exception (probably a `ClassCastException`?) to indicate that the memory structure is not compatible with an instance of a **String**.

Converting to the supertype

- ▶ It is permissible to convert from a sub-type to a super-type with automatic coercion
- ▶ It is not permissible to convert from a super-type to a sub-type without explicit casting



Type Coercion

- ▶ Relaxing *type compatibility*:
 - ▶ Allowed by most languages to different extents
- ▶ Two kinds of *type coercion*:
 - ▶ Implicit type conversions (AKA *standard conversion*): weakly typed languages, automatic type conversion by the compiler

```
int pi = 3.14159;
```

C

```
float x = '0';
```

```
unsigned int age = -1;
```

- ▶ Explicit type conversion (AKA *cast*):

```
int pi = (int)3.14159;
```

C

float to **int** causes truncation, i.e. removal of the fractional part.

double to **float** causes rounding of digit

long int to **int** causes dropping of excess higher order bits.

float x = '0' => **x** is definitely a **float**, it is assigned to **48.00f** -- the float value of the character '**0**'

Coercion II

► What coercion is expected?

```
class LinkedList extends List {                                Java
    void method(List aList) {
        LinkedList aLinkedList = aList;          // Error: Requires cast
        LinkedList aLinkedList = (LinkedList)(aList); // OK, type checked
        dynamically
        aList = aLinkedList;                      // OK (standard
        conversion)
    }
}
```

Strong Vs. Weak Typing

- ▶ **Weak typing:** support either implicit type, ad-hoc polymorphism (also known as overloading) or both.
- ▶ **Strong typing:** Coercions allowed only if value is preserved!
 - ▶ Other coercion operations: Cast (explicit coercion) is required
 - ▶ Strong typing implies static typing
- ▶ Pascal and Ada are strongly typed

```
type grade = 0..100; // Subtype of integer Ada
...
i : integer;
g : grade;
...
i := g; // Coercion OK, value always preserved
g := i; // Compilation error: Cast required
g := 1 + 1; // Compilation error: Cast required
g := grade(i); // Cast; Run-time checking will ensure that 0 ≤ i ≤ 100
g := grade(0); // Cast; Run-time checking will ensure that 0 is in range
```

Strong Vs. Weak Typing

	Weak Typing	Strong Typing
Pseudocode	<pre>a = 2 b = "2" concatenate(a, b) # Returns "22" add(a, b) # Returns 4</pre>	<pre>a = 2 b = "2" concatenate(a, b) # Type Error add(a, b) # Type Error concatenate(str(a), b) # Returns "22" add(a, int(b)) # Returns 4</pre>
Languages	<p>BASIC, Perl, PHP, Rexx (is language dependent)</p>	ActionScript 3, C++, C#, Java, Python, OCaml

Reminder: Static Vs. Dynamic Typing

- ▶ ***Static typing:*** *Type checking performed statically*
 - ▶ Done by the compiler
 - ▶ Requires type declaration for every name
 - ▶ Safer
 - ▶ Examples: All strongly-typed languages, C, C++, Eiffel, Java™
- ▶ ***Dynamic typing:*** *Type checking performed dynamically*
 - ▶ Requires *dynamic binding* between the object and the type!
 - ▶ More flexible
 - ▶ Examples: Smalltalk, Lisp
 - ▶ Java: Cast operations are checked dynamically

Example: dynamic typing in Smalltalk

- ▶ Smalltalk lacks static types
- ▶ Each object's class is known dynamically
- ▶ Type checks are only done dynamically

Smalltalk

```
StackUser>>use
| aStack |
aStack := Stack new;
aStack push;
aStack standOnYourHead;
```

"object type not known! "
"object (dynamic) type is now set"
"OK: object's class recognizes message"
"Failure: object's class does not recognize
message"

"Method will compile successfully!"

Summary

- ▶ Software verification & validation - checking that the implementation conforms to our expectations
- ▶ The differences between Static vs Dynamic validation
- ▶ Validation through code inspection
- ▶ The use of Typing in programming languages
 - ▶ Static/Dynamic typing
 - ▶ Strong/Weak typing
 - ▶ Type coercion

Exercise: strong and weak typing

Which of the following is an example of a strong typing and which is an example of weak typing? Explain why?

```
1. /* Python code */  
2. >>> foo = "x"  
3. >>> foo + 2  
4. Traceback (most recent call last):  
5.   File "<pyshell#3>", line 1, in ?  
6.     foo = foo + 2  
7. TypeError: cannot concatenate 'str' and 'int' objects  
8. >>>
```

Example 1

```
1. /* PHP code */  
2. <?php  
3. $foo = "x";  
4. $foo = $foo + 2; // not an error  
5. echo $foo;  
6. ?>
```

Example 2

Exercise: Strong and Weak Typing (2)

- ▶ What is good about dynamic typing?
- ▶ The following is an example from a dynamically typed language (Python). Why in this example, is dynamic typing a problem?

```
1. /* Python code */
2. my_variable = 10
3. while my_variable > 0:
4.     i = foo(my_variable)
5.     if i < 100:
6.         my_variable++
7.     else
8.         my_varaible = (my_variable + i) / 10 // spelling error intentional
```

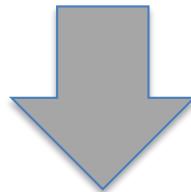
End

Review of diagrams

Week 7 class
CE202 Software Engineering
M. Gardner

Waterfall Lifecycle and Main Outputs (reminder)

Problem Definition
Viewpoints/Scope



Functional &
Non-
Functional
requirements
Scenarios

Use-case
diagrams
Use-case
descriptions
Type Diagrams
Activity Diagrams
Prototypes
CRC Cards
Class Diagram
Interaction
Diagrams
State Machines

Prototypes
Class Diagram
Interaction Diagrams
Package diagrams
Architectural design

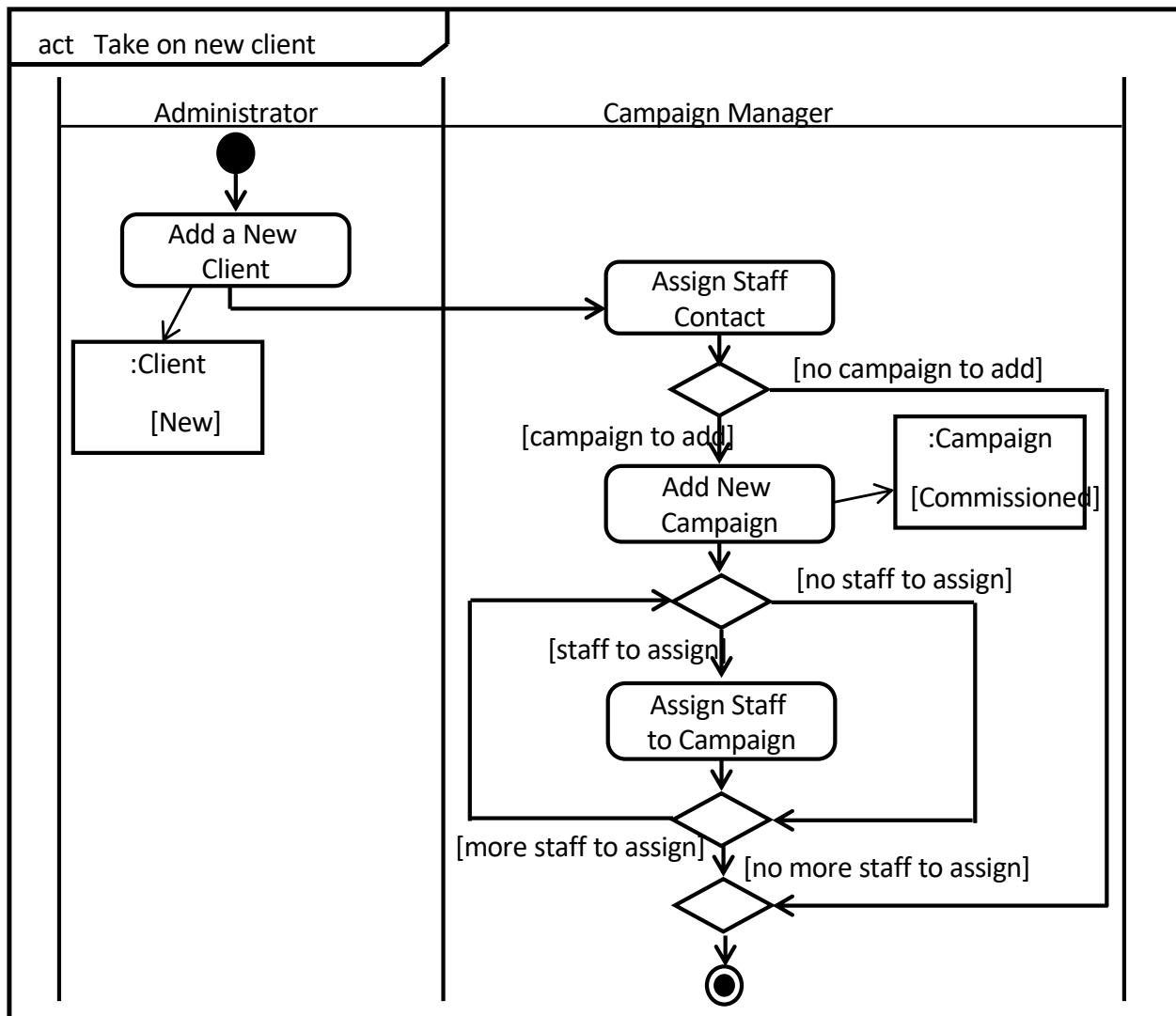
Coding
Unit testing
Version control

Integration testing
Unit testing

Deployment testing
User evaluation



Activity diagrams



Can be used at any stage in the lifecycle. Examples:

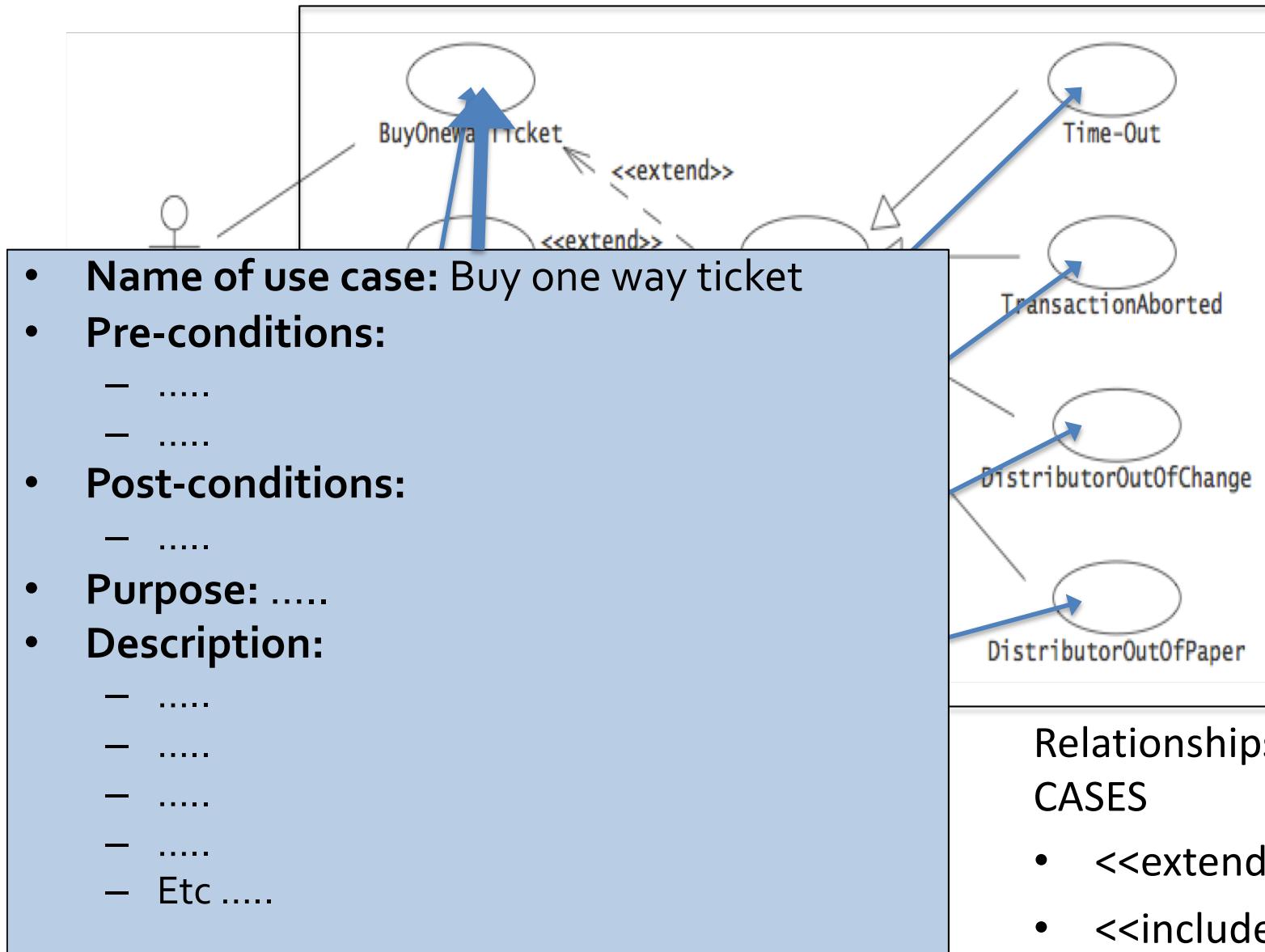
- Model a process derived from the requirements
- Model logic flow for a use case
- Model logic flow for code
- Etc

Need to ensure consistency with other diagrams. Eg:

- Use of objects
- Use-cases (if activities are based on them)

The decision about whether to use this diagram is dependent on your particular problem. Eg. whether you need to model a process/activity flow?

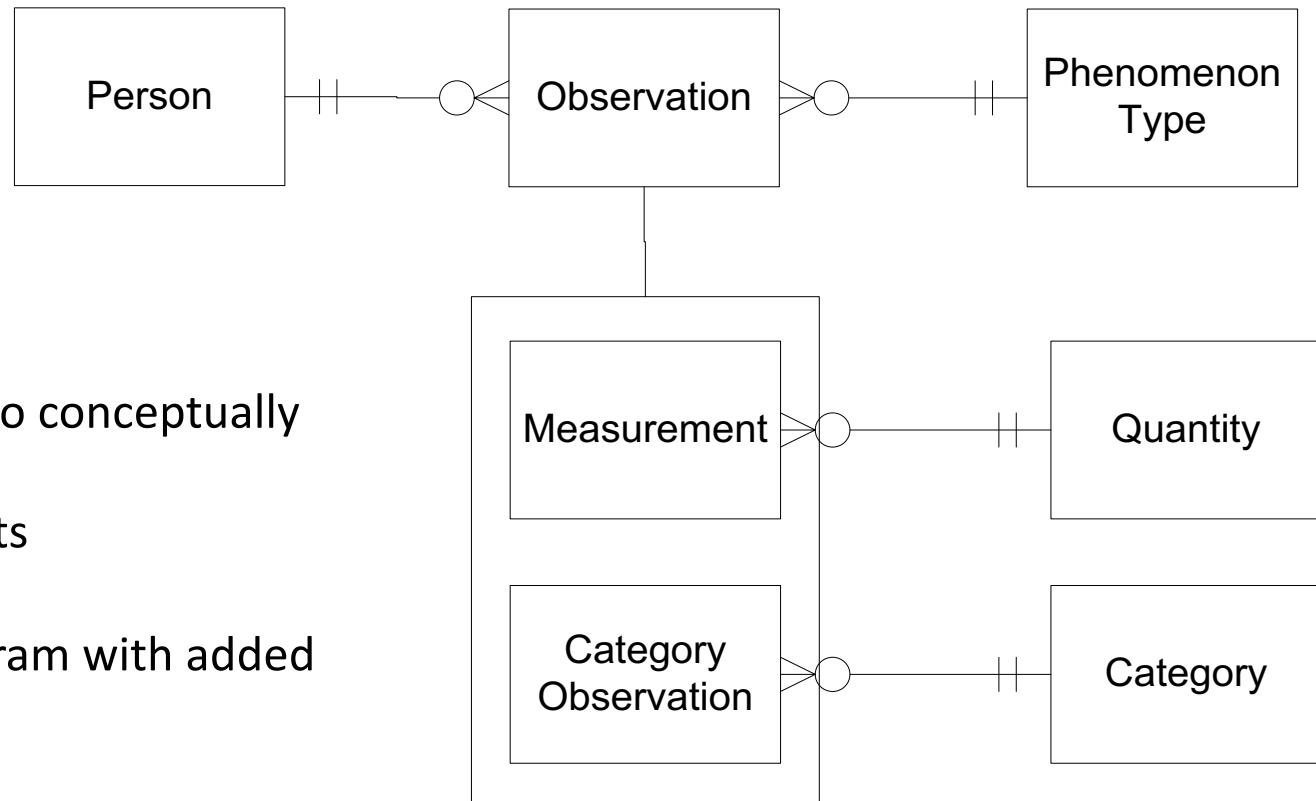
Use case diagram



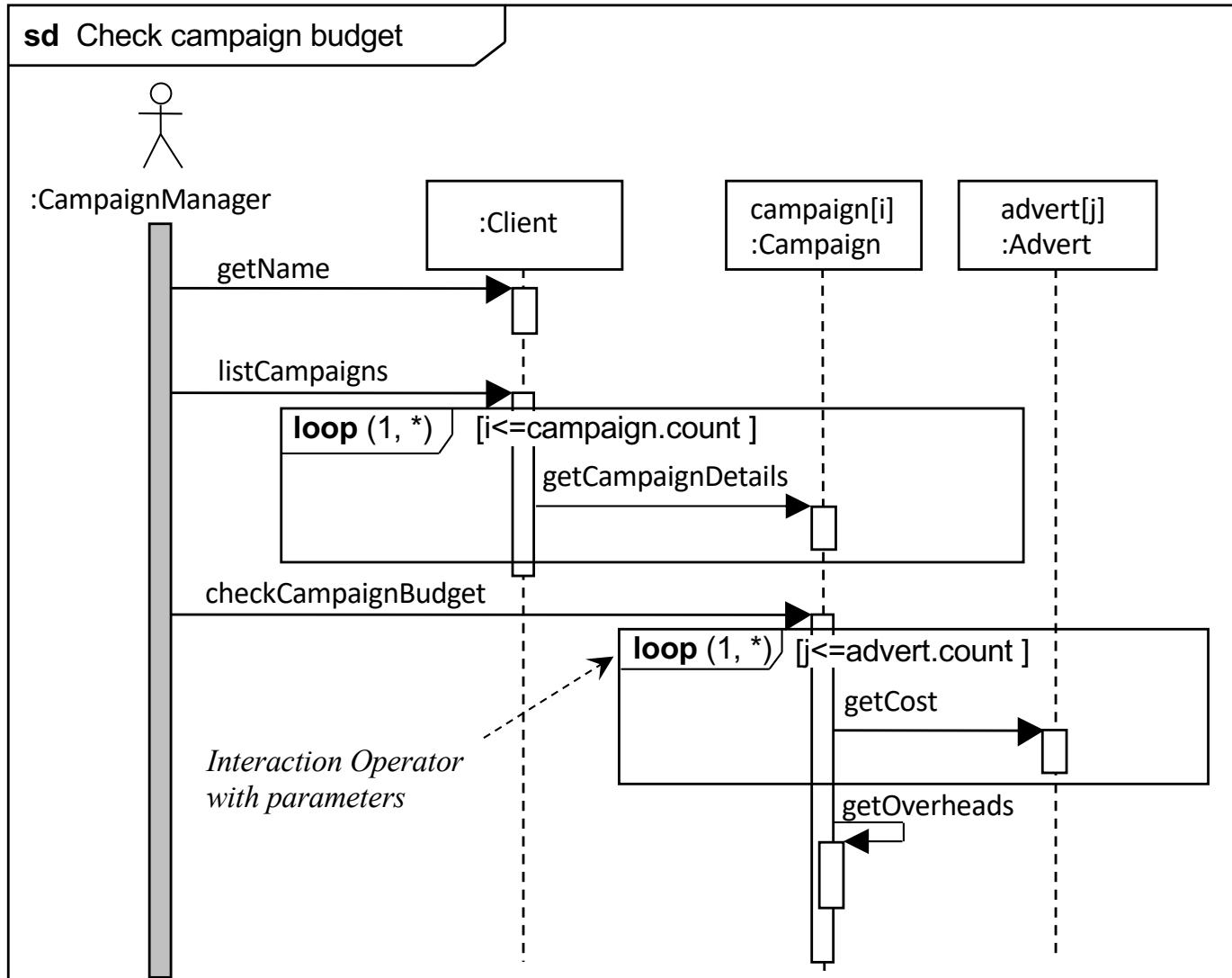
- Do not show logic flow

Type diagrams

- Used during analysis to conceptually model a problem
- Based on requirements
- Only use if needed
- Essentially an ER diagram with added ‘Types’
- Not part of UML
- You may not need to use it!

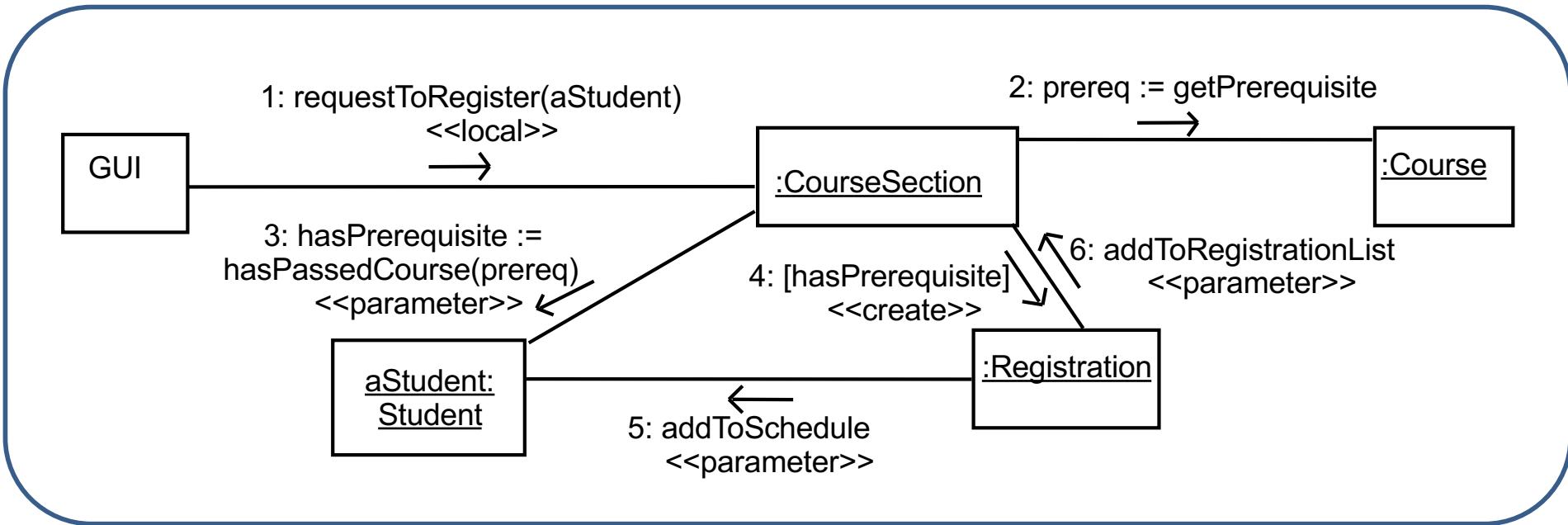


Sequence diagrams



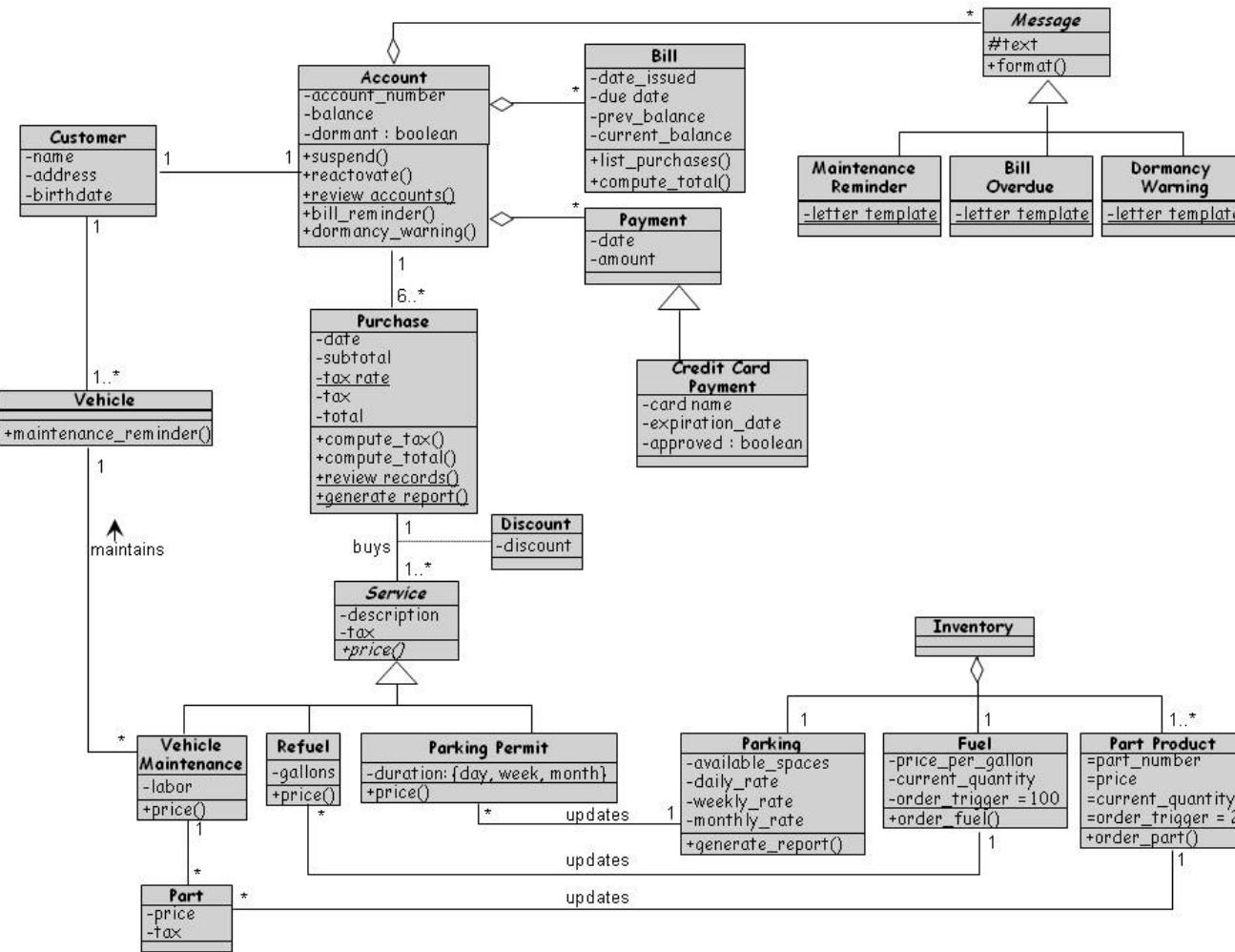
- Based on **use case descriptions**
- Models the flow of messages between objects involved in a use case
- Often provide a bridge between use cases and the class diagram
- Need to ensure consistency with **class diagram** (objects, associations and operations), and any **state machines**
- Beforehand you will need to decide on your **candidate classes/objects**

Collaboration diagrams



- An alternative to the sequence diagram
- Shows exactly the same information (just numbers messages instead to show the order)
- Governed by the same rules as the sequence diagram
- Same consistency checking as sequence diagrams
- Can be easier to map to a class diagram
- Both can be used to generate template code for your classes

Class diagram



Associations between CLASSES

- Named association with cardinality and direction (can be recursive, can have association classes)
- Inheritance or
- Whole/Part relationships:

— Aggregation or
— Composition or
(can be included in a named association)

- The most important diagram
- Bridges between analysis and design
- Based on:
 - Analysis of requirements
 - Interaction diagrams
 - Needs to be consistent
- Check consistency with **interaction diagrams** and state machines

Relationships in diagrams

Use-case diagrams

Between USE CASES

- <<extend>>
- <<include>>
- <<generalise>> or 

Class diagrams

Between CLASSES

- Named association with cardinality and direction
(can be recursive, can have association classes)
- Inheritance or 
- Whole/Part relationships:
 - Aggregation or 
 - Composition or 

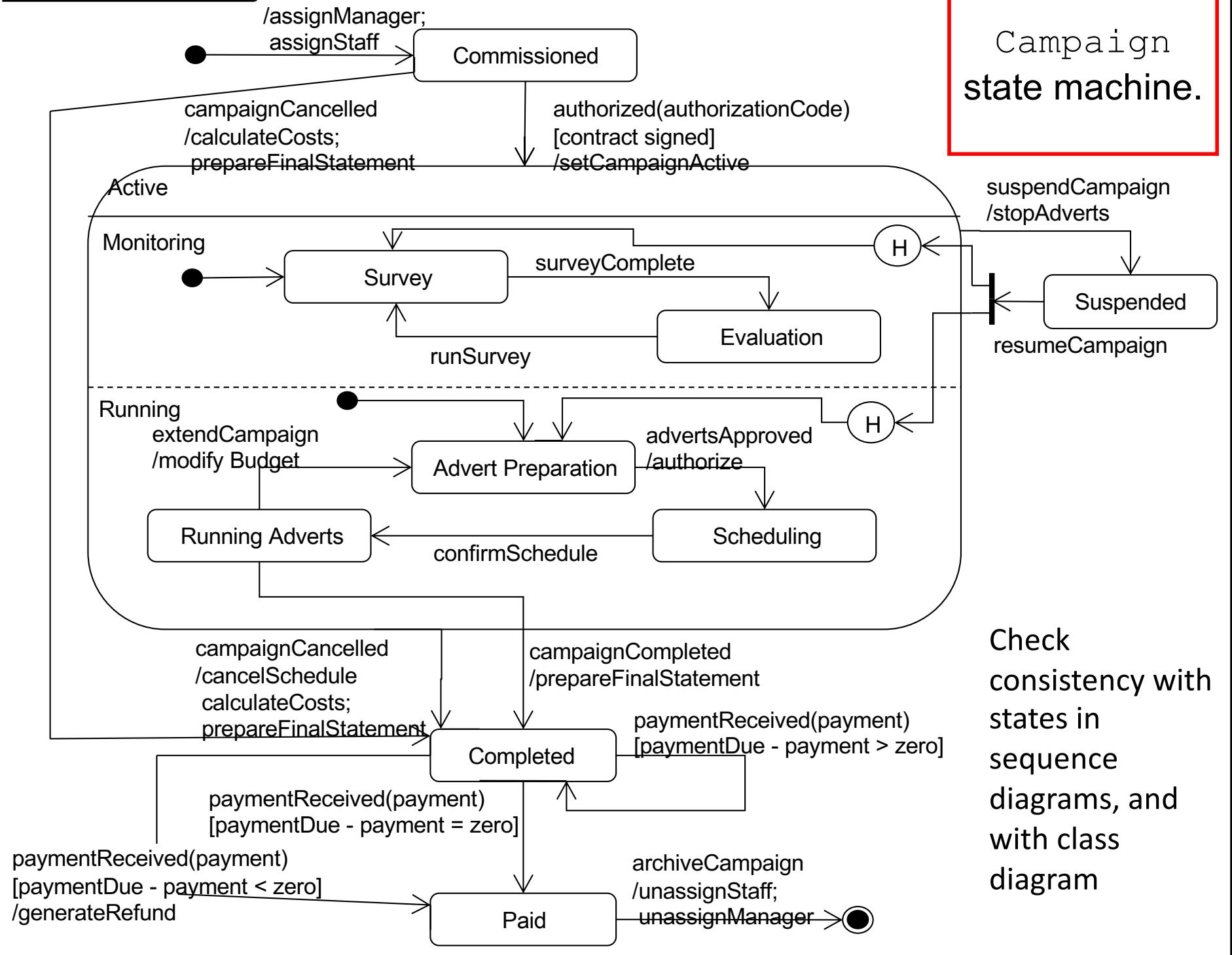
(can be included in a named association)

CRC cards

Class Name	<i>Client</i>
Responsibilities	Collaborations
<i>Provide client information.</i>	
<i>Provide list of campaigns.</i>	<i>Campaign provides campaign details.</i>
Class Name	<i>Campaign</i>
Responsibilities	Collaborations
<i>Provide campaign information.</i>	<i>Advert provides advert details.</i>
<i>Provide list of adverts.</i>	<i>Advert constructs new object.</i>
<i>Add a new advert.</i>	
Class Name	<i>Advert</i>
Responsibilities	Collaborations
<i>Provide advert details.</i>	
<i>Construct adverts.</i>	

- One CRC card per Class
- Responsibilities will need to be supported by the Class **attributes** and **operations**.
- Collaborations must match with Class diagram **associations**.

sm Campaign Version 3



Consistency checking your state machine

- Every **event** should appear as an incoming message for the appropriate object on an **interaction diagram**(s).
- Every **action** should correspond to the execution of an **operation** on the appropriate **class**, and perhaps also to the dispatch of a message to another object.
- Every **event** should correspond to an **operation** on the appropriate **class** (but note that not all operations correspond to events).
- Every **outgoing message** sent from a state machine must correspond to an **operation** on another **class**.

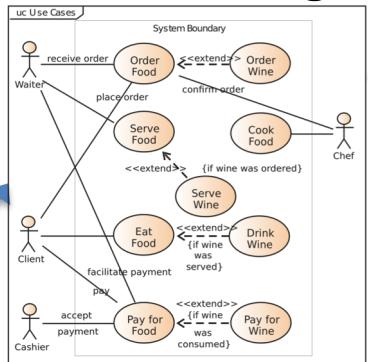
Analysis Flow in UML (reminder)

1 Requirements

Problem
Definition

Viewpoints/Scope
Functional &
Non-
Functional
requirements
Scenarios

2 Use-case diagrams



Actors

A sequence diagram for each use-case description

Names in bubbles
match with use-case description names

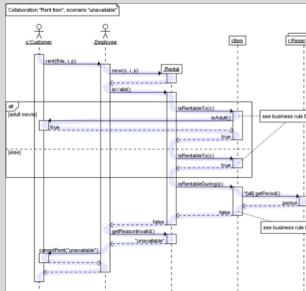
Name	The Use Case name. Typically the name is of the format <action> + <object>.
ID	An identifier that is unique to the Use Case.
Description	A brief sentence that states what the user wants to be able to do and what benefit he will derive.
Actors	The type of user who interacts with the system to accomplish the task. Actors are identified by role name.
Organizational Benefits	The value the organization expects to receive from having the functionality described. Ideally this is a link directly to a Business Objective.
Frequency of Use	How often the Use Case is executed.
Triggers	Concrete actions made by the user within the system to start the Use Case.
Preconditions	Any states that the system must be in or conditions that must be met before the Use Case is started.
Postconditions	Any states that the system must be in or conditions that must be met after the Use Case is completed successfully. These will be met if the Main Course or any Alternate Courses are followed. Some Exceptions may result in failure to meet the Postconditions.
Main Course	The most common path of interactions between the user and the system. 1. Step 1 2. Step 2
Alternate Courses	Alternative paths through the system. AC1: <condition for the alternate to be called> 1. Step 1 2. Step 2 AC2: <condition for the alternate to be called> 1. Step 1
Exceptions	Exception handling by the system. EX1: <condition for the exception to be called> 1. Step 1 2. Step 2 EX2: <condition for the exception to be called> 1. Step 1

3 Use-case descriptions

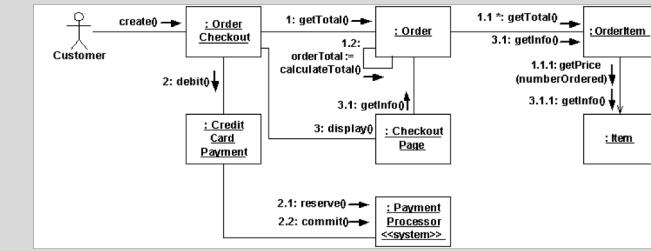
List of Candidate Classes:

Campaign
Order
Customer

Sequence diagrams



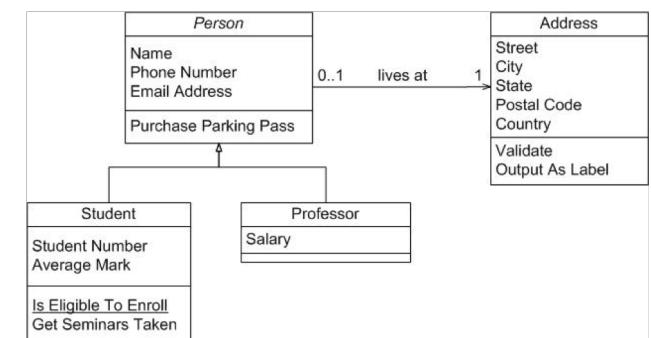
Collaboration diagrams



Class Names
Messages/Associations

Class names

6 Class diagram



End

mgardner@essex.ac.uk

Software validation: testing I

- Basic terminology
- Phases in the testing process

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Software Testing

- ▶ Who tests?
 - ▶ Ideally specialist test teams with access to software to build and execute test scripts
 - ▶ Often the analysts who have carried out the initial requirements gathering and analysis
 - ▶ In eXtreme Programming (XP) programmers are expected to write test harnesses for classes before they write the code
 - ▶ Users of the system, who will test against requirements and do user acceptance testing



Basic testing terminology

- ▶ **Testing:** Executing a program, to detect the differences between specified and observed behaviour.
- ▶ **Validation vs. Defect Testing:**
 - ▶ **Validation testing:** The goal is to ensure the system meets the client's expectations
 - ▶ **Defect testing:** The goal is to detect defects (reveal problems).
- ▶ **Black vs. White Box Testing:**
 - ▶ **Black-box:** From “outside”, just the public methods
 - ▶ **White-box:** From “inside”, considering all aspects (including private methods and fields)

Examples of Faults and Errors

- ▶ **Faults in the Interface specification**

- ▶ Mismatch between what the client needs and what the server offers
- ▶ Mismatch between requirements and implementation

- ▶ **Algorithmic Faults**

- ▶ Missing initialization
- ▶ Incorrect branching condition
- ▶ Missing test for null

- ▶ **Mechanical Faults (very hard to find)**

- ▶ Operating temperature outside of equipment specification

- ▶ **Errors**

- ▶ Null reference errors
- ▶ Concurrency errors
- ▶ Exceptions.

Purpose of Testing

- ▶ The purpose of testing is to try **find errors, not to prove the software is correct**
- ▶ Test data should test the software at its limits and test business rules
 - ▶ extreme values (very large numbers, long strings)
 - ▶ borderline values (0, -1, 0.999)
 - ▶ invalid combinations of values (age = 3, marital status = married)
 - ▶ nonsensical values (negative order line quantities)
 - ▶ heavy loads (are performance requirements met?)
 - ▶ See week 10 class on selecting test conditions (equivalence classes) and choosing boundary values



Caveat



- ▶ Testing can show the presence of bugs *but not their absence* [Dijkstra, 1972]

Levels of Testing

► Level 1

- ▶ Test modules (classes), then programs (use cases) then suites (application)

► Level 2 (Alpha Testing or Verification)

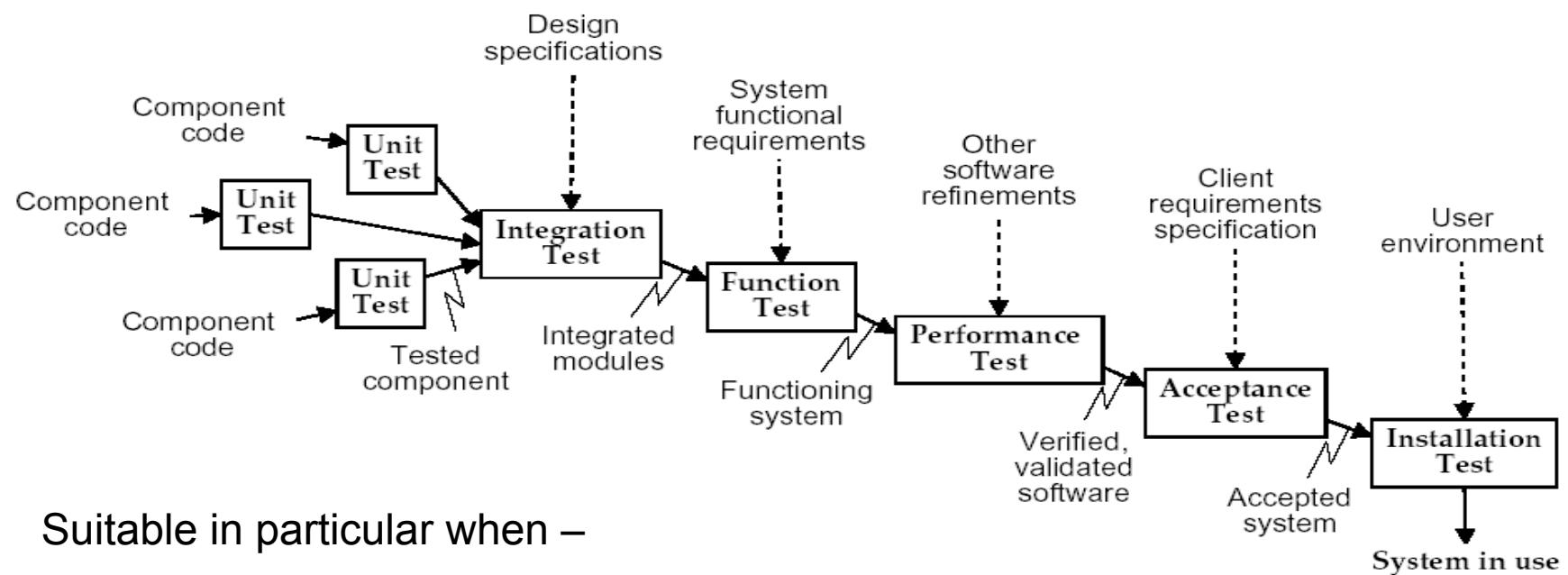
- ▶ Execute programs in a simulated environment and test inputs and outputs

► Level 3 (Beta Testing or Validation)

- ▶ Test in a live user environment and test for response times, performance under load and recovery from failure



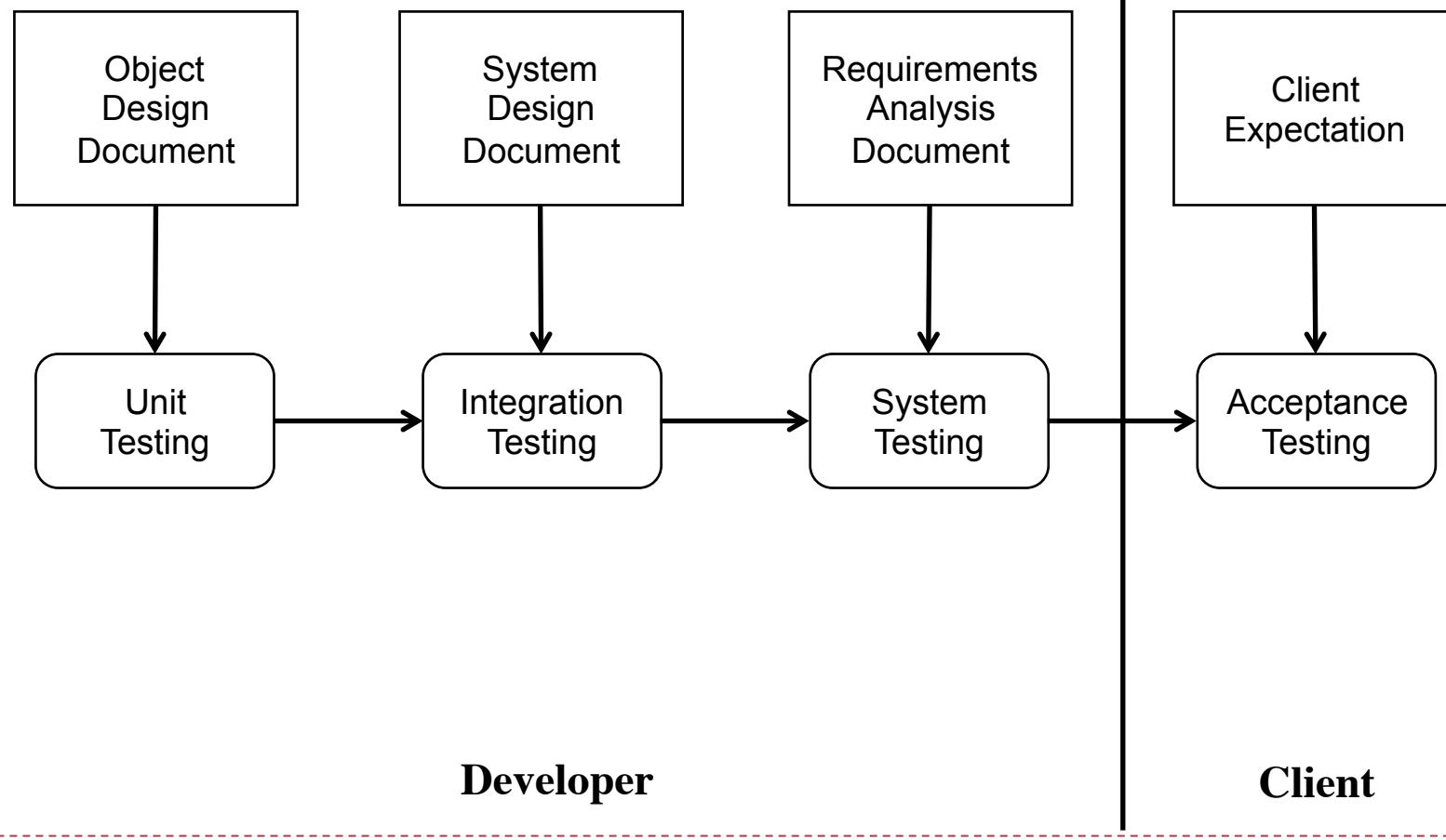
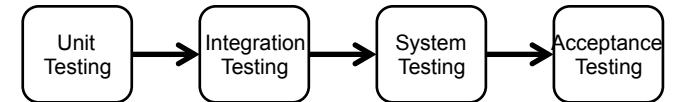
The testing hierarchy



Suitable in particular when –

1. Project is large enough
2. Process model has progressively larger deliverables

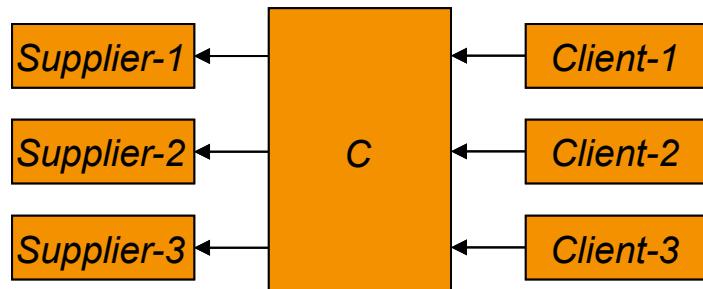
Testing Activities



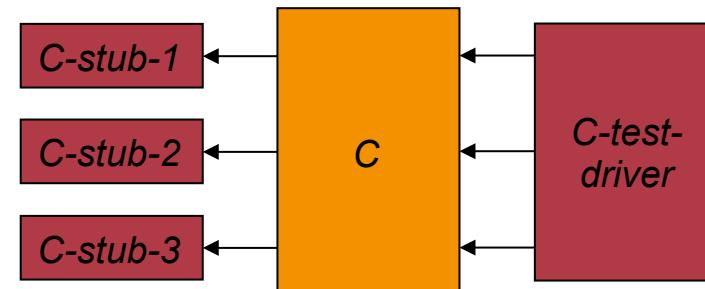
Unit testing

- ▶ Isolate the component under test *C* from the rest of the program:
 - ▶ *Test driver* calls *C*
 - ▶ *Test stubs* whenever *C* calls other components

Planned deployment of *C*:



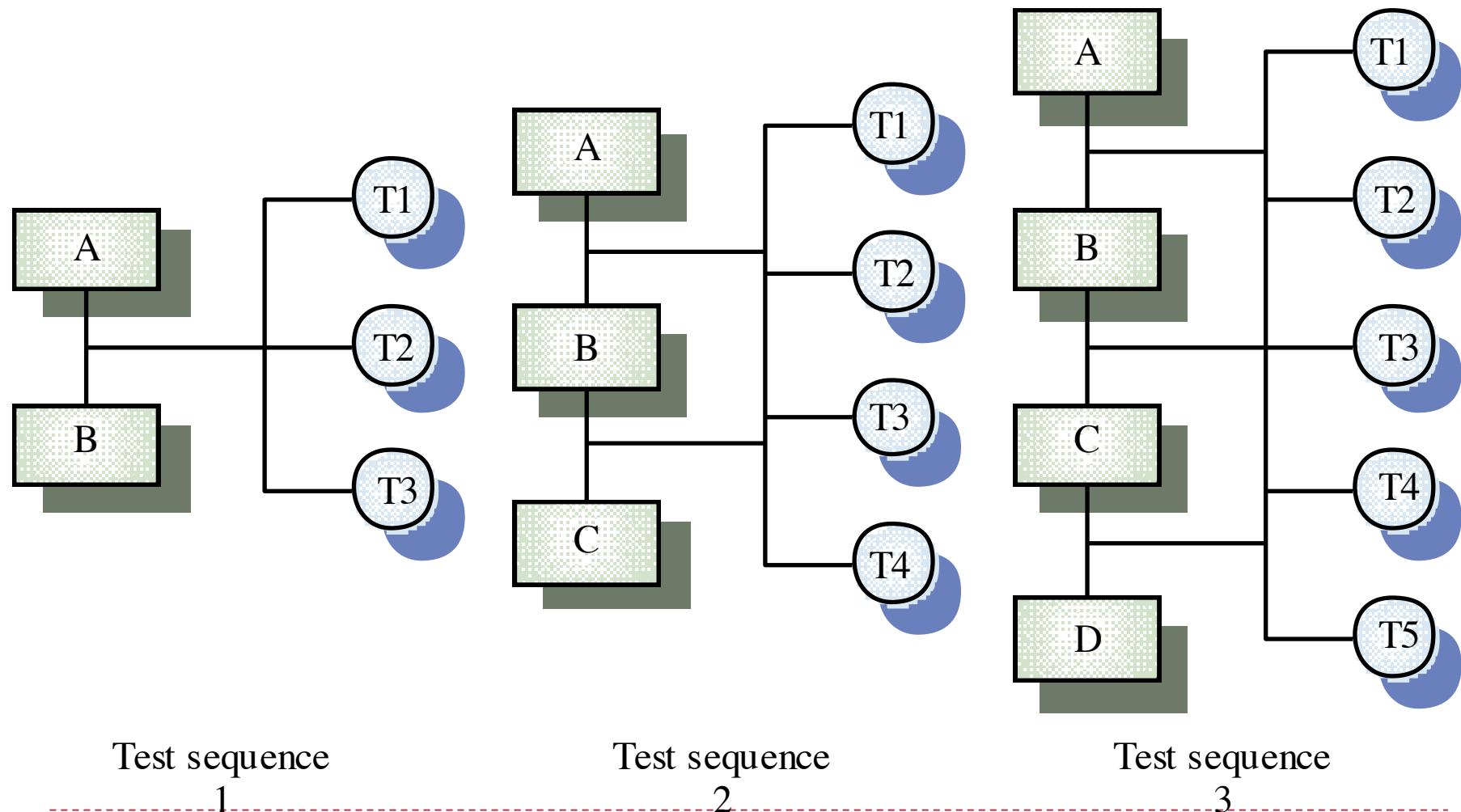
Unit testing of *C*:



Integration testing

- ▶ Tests a system consisting of integrated components
 - ▶ Can be any set of packages or components, up to the entire program
- ▶ Should be black-box testing
 - ▶ Tests derived from the specification
- ▶ **Problem:** Difficult to establish where the fault occurred
 - ▶ Not the problem with Unit testing
- ▶ **Solution:** Incremental integration testing

Incremental integration testing



Test sequence
1

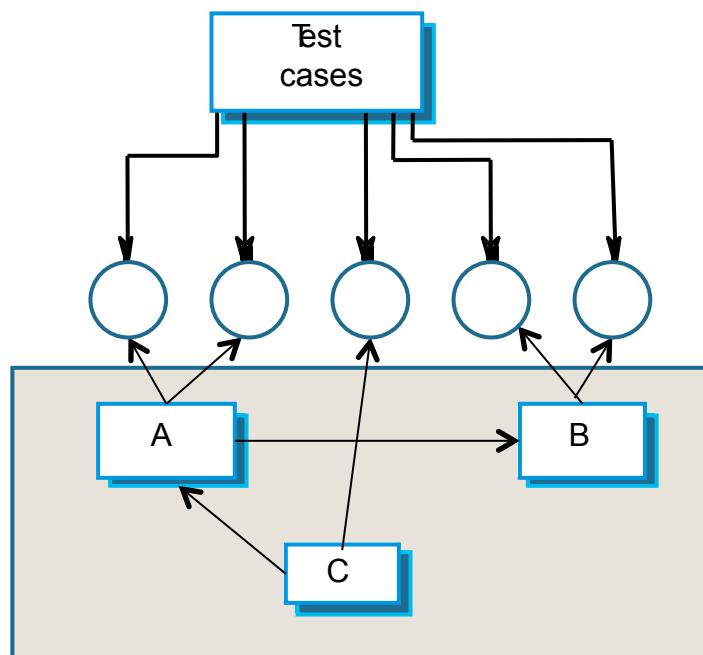
Test sequence
2

Test sequence
3

Approaches to integration testing

- ▶ Top-down testing
 - ▶ Start with high-level system and integrate from the top-down replacing individual components by stubs where appropriate
- ▶ Bottom-up testing
 - ▶ Integrate individual components in levels until the complete system is created
- ▶ In practice, most integration involves a combination of these strategies

Functional/System testing



- ▶ Conducted against the functional specifications
- ▶ Test function, not structure or performance
 - ▶ Test through interface
 - ▶ Ignore implementation detail
- ▶ Examples:
 - ▶ Test that a functional requirement is implemented properly

Performance testing

- ▶ Testing performance requirements
 - ▶ Usually: Test the programs use of resources such as time + space
- ▶ In contrast with: Functional or black-box testing
- ▶ Very important phase for real-time and critical systems!
- ▶ Test against the non-functional requirements

Example: Stress testing

- ▶ Exercises the system beyond its maximum design load
 - ▶ Often causes defects to come to light
- ▶ Testing the system's recovery
 - ▶ Systems should not fail catastrophically
 - ▶ Must not crash
 - ▶ Must not corrupt the data (e.g., leave 'dangling pointers')
 - ▶ Recovery should be swift

Regression testing

- ▶ Re-run all tests after applying changes
 - ▶ Tests lead to the discovery of faults
 - ▶ After changing the program: Must be re-tested!
- ▶ Applicable for every “version” of the program

Acceptance testing

- ▶ One step before last
 - ▶ Test against client's requirements
 - ▶ Involve the client in the testing
 - ▶ Use a test facility (not the client's environment)

Installation (a.k.a. release) testing

- ▶ Last step in testing
 - ▶ Execute in the target environment
 - ▶ Configure the target environment
 - ▶ Re-run all previous tests (unit, acceptance, etc.)
- ▶ usually black-box or functional testing
 - ▶ Based on the system specification only;
 - ▶ Ideally the testers do not have knowledge of the system implementation.

Summary

- ▶ Testing terminology eg. black/white box, validation & defect testing
- ▶ Purpose of testing – can't prove there are no bugs
- ▶ Levels of testing – classes/modules/applications, alpha & beta testing
- ▶ The testing hierarchy
 - ▶ Unit testing
 - ▶ Integration/regression testing
 - ▶ Functional testing
 - ▶ Performance/stress testing
 - ▶ Acceptance testing
- ▶²⁰ Installation testing

Further reading

- ▶ 1972 Turing Award lecture The Humble Programmer by Edsger Dijkstra
- ▶ Chapter 11 – Bruegge – Testing
- ▶ Chapter 8 – Pfleeger – Testing the Programs
- ▶ Chapter 9 – Pfleeger – Testing the System

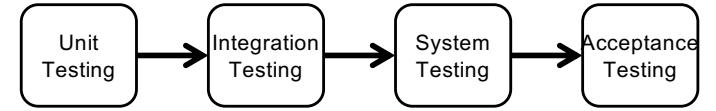
Software validation: unit testing

- The difference between Unit and Integration testing
- Static and Dynamic unit testing
- Black and White box testing approaches
- Code-coverage tests (White-box)
- Unit testing
 - Selecting test cases
 - Components of unit testing and examples: Test driver, tested unit, stubs, oracle
 - JUNIT
- Test Driven Development approach

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex





Reminder: Types of Testing

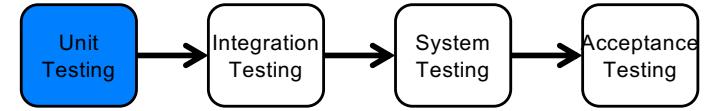
▶ Unit Testing

- ▶ Individual component (class or subsystem)
- ▶ Carried out by developers
- ▶ Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality

▶ Integration Testing

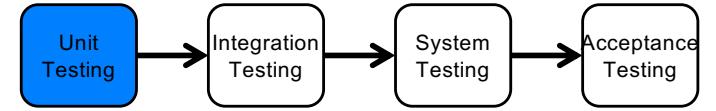
- ▶ Groups of subsystems (collection of subsystems) and eventually the entire system
- ▶ Carried out by developers
- ▶ Goal: Test the interfaces among the subsystems.





Unit Testing

- ▶ Static Testing (at compile time)
 - ▶ Static Analysis
 - ▶ Review
 - ▶ Walk-through (informal)
 - ▶ Code inspection (formal)
- ▶ Dynamic Testing (at run time)
 - ▶ Black-box testing (Test the input/output behavior)
 - ▶ White-box testing (Test the internal logic of the subsystem or object)
 - ▶ Data-structure based testing (Data types determine test cases)



Black-box testing

- ▶ Focus: I/O behavior
 - ▶ If for any given input, we can predict the output, then the component passes the test
 - ▶ Requires a test oracle (to see if a test passes or not)
 - ▶ Almost always impossible to generate all possible inputs ("test cases")
- ▶ Goal: Reduce number of test cases by equivalence partitioning (see week 10 class):
 - ▶ Divide input conditions into equivalence classes
 - ▶ Choose test cases for each equivalence class.
 - ▶ (Example: If an object is supposed to accept a negative number, testing one negative number is enough)

Black-box testing: Test case selection

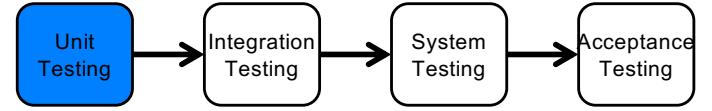
a) Input is valid across range of values

- ▶ Developer selects test cases from 3 equivalence classes:
 - ▶ Below the range
 - ▶ Within the range
 - ▶ Above the range

b) Input is only valid, if it is a member of a discrete set

- ▶ Developer selects test cases from 2 equivalence classes:
 - ▶ Valid discrete values
 - ▶ Invalid discrete values
- ▶ No rules, only guidelines.





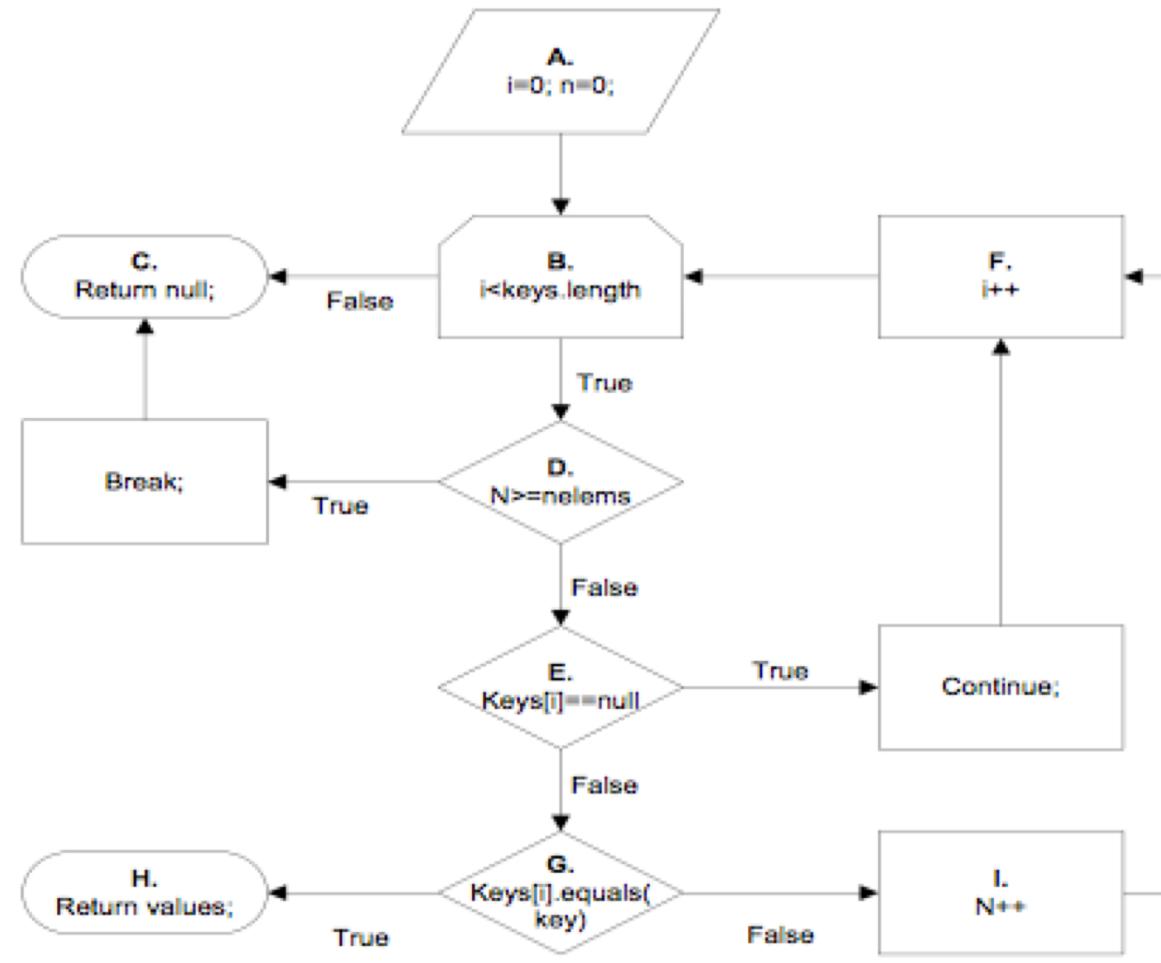
White-box testing overview

- ▶ Code coverage
- ▶ Branch coverage
- ▶ Condition/statement coverage
- ▶ Path coverage
- ▶ The differences between these approaches is discussed in the week 10 class



Code coverage test selection

GET METHOD



Flow-chart for a method

Paths:

1. ABC
2. ABDC
3. ABDEGH
4. ABDEF(B)
5. ABDEGIF(B)

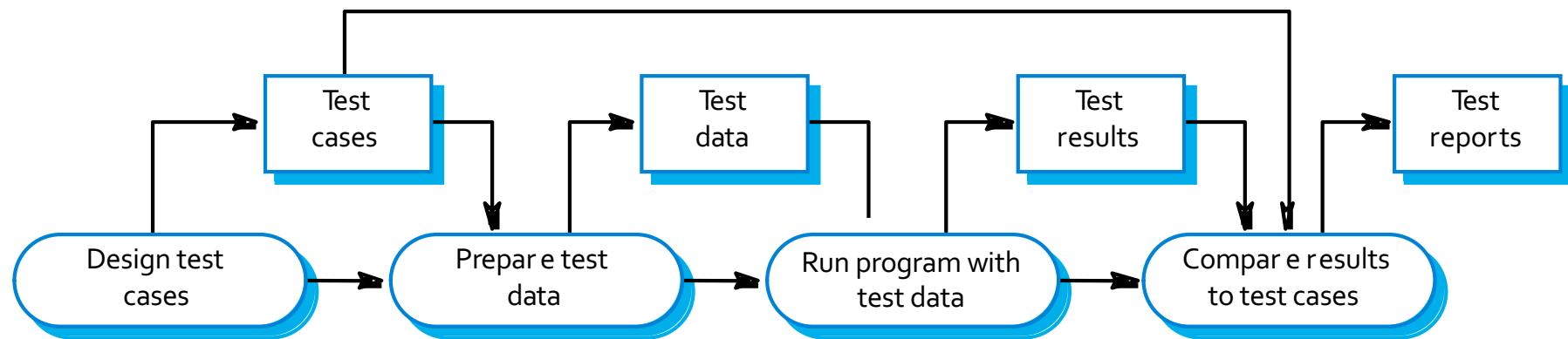
Unit Testing Heuristics

1. Create unit tests when object design is completed
 - ▶ Black-box test: Test the functional model
 - ▶ White-box test: Test the dynamic model
2. Develop the test cases
 - ▶ Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
 - ▶ Don't waste your time!

4. Desk check your source code
 - ▶ Sometimes reduces testing time
5. Create a test harness
 - ▶ Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
 - ▶ Often the result of the first successfully executed test
7. Execute the test cases
 - ▶ Re-execute test whenever a change is made (“regression testing”)
8. Compare the results of the test with the test oracle
 - ▶ Automate this if possible.

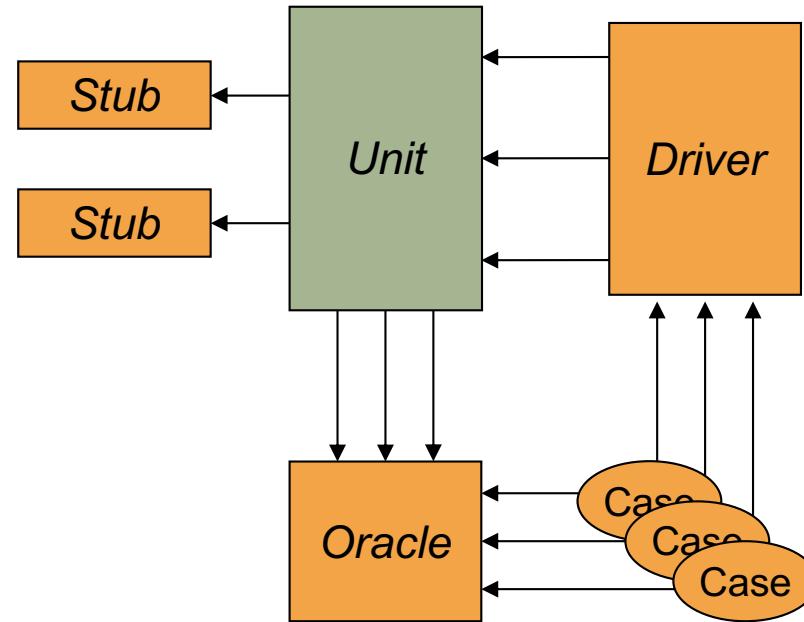
Don't forget regression testing - Re-execute test cases every time a change is made.

Designing and using test cases



Elements in unit testing

- ▶ Cases
- ▶ Driver
- ▶ Oracle
- ▶ Stubs



Terminology

- ▶ **Test case:** A set of input data and expected results
 - ▶ Exercises a component with the purpose of causing failures and detecting faults
- ▶ **Test driver:** A program that simulates the part of the system calling the unit under test (“subject”)
 - ▶ Needed for all tests except for tests for the complete system (e.g., *integration test*)
- ▶ **Test oracle:** Declares either that the system passed or failed
 - ▶ WRT a collection of test cases
- ▶ **Test stub:** isolates the component under test from the rest of the program

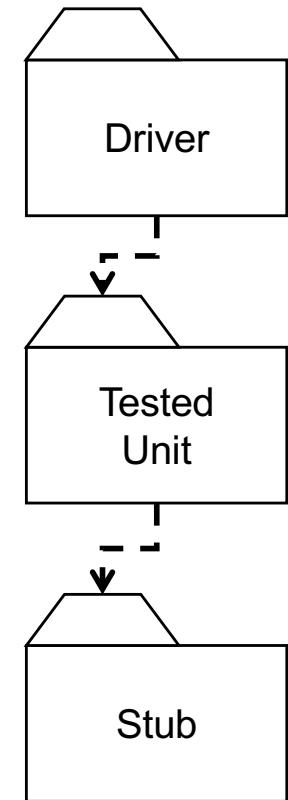
Stubs and drivers

- ▶ **Driver:**

- ▶ A component, that calls the TestedUnit
- ▶ Controls the test cases

- ▶ **Stub:**

- ▶ A component, the TestedUnit depends on
- ▶ Partial implementation
- ▶ Returns fake values.

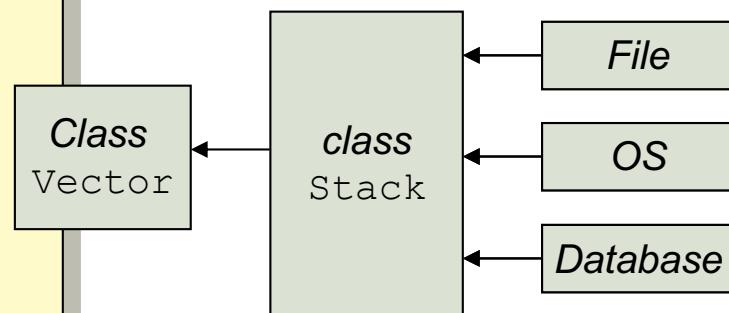


Example: Unit testing for Stack

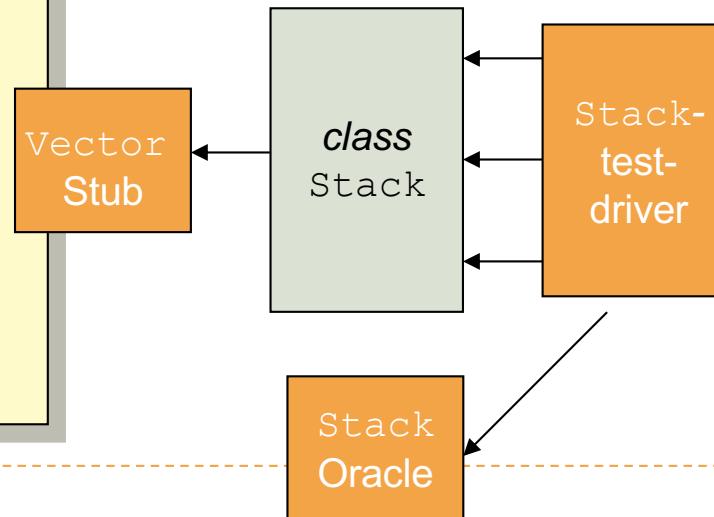
- ▶ Isolate class Stack

```
public class Stack {  
    public void push(Object obj)  
        throws Exception {  
        if (! imp.add(obj))  
            throw new Exception; }  
    ...  
}  
  
public Object pop()  
    throws Exception {  
    if (imp.size()==0)  
        throw new Exception;  
    return imp.removeElementAt(size-1); }  
...  
  
private Vector imp;  
}
```

Planned deployment of Stack:



Unit testing of Stack:



Example: Test driver for Stack

- ▶ A trivial example

- ▶ Missing:

- ▶ Use larger input/output sets
- ▶ Select inputs wisely
- ▶ Need to stub Vector

```
package StackTest;

public class Driver {

    public static boolean test_case1() {
        Stack s = new Stack();
        String input = new String("testing");
        s.push(input);
        String output = s.pop();
        return
            oracle.report_case1(input, output);
    }

    public static boolean test_case2()
        {...}

    public static StackTestOracle oracle;
}
```

Example: Test Oracle for Stack

- ▶ A trivial example

- ▶ Suitable only for `test_case1`
- ▶ Returns **true** if fault occurred (test failed)

- ▶ Missing:

- ▶ Report where fault occurred

```
package StackTest;

public class StackTestOracle {

    public static boolean report_case1(
        String input, String output) {
        // Return false if no fault detected
        return (input != output);
    }
    ...
}
```

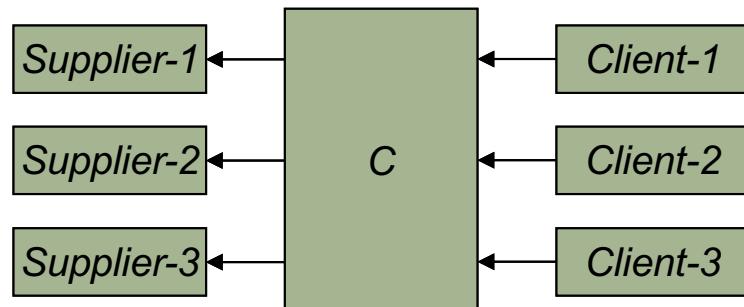
How to write a good test driver/oracle

- ▶ Driver: Read input from a file
 - ▶ E.g.: 4, 9, 16, 25, ...
- ▶ Oracle: Read expected output from another file and compare with input
 - ▶ E.g.: 2, 3, 4, 5, ...

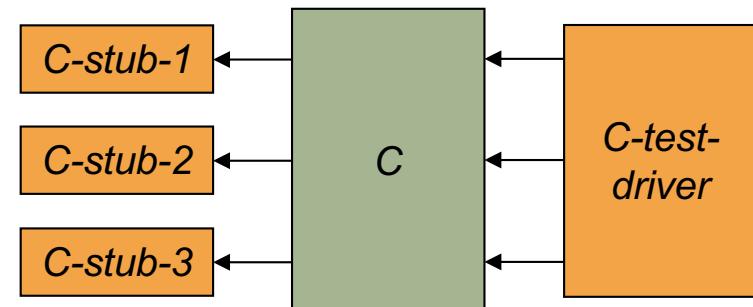
Test stubs

- ▶ In unit testing we isolate the component under test C from the rest of the program:
 - ▶ *Test driver* calls C
 - ▶ *Test stubs* whenever C calls other components
- ▶ A **test stub** simulates units that are called by the subject
 - ▶ Needed for all tests except for tests for the complete system (e.g., *integration test*)

Planned deployment of C:



Unit testing of C:



How to write a stub?

- ▶ Stubs should be kept as simple as possible
 - ▶ Otherwise: They need to be tested themselves...
- ▶ Possible alternatives:
 - ▶ Write a trivial stub
 - ▶ Use code from a reliable source
 - ▶ For example: Replace StackStubs/Vector with java/util/LinkedList
 - ▶ Replace one implementation with another

```
package StackTest;

// an Adapter for LinkedList
public class Vector extends java.util.LinkedList {
    // public void size()          -same as in LinkedList
    // public boolean add(obj)    -same as in LinkedList
    public void removeElementAt(int loc) {
        return this.remove();
    }
}
```

Example: Test stub for Stack

- ▶ Isolate Stack: Replace service suppliers with stubs!
 - ▶ Otherwise: Failures can occur in the suppliers!
- ▶ Replace Vector with a “Stub”
 - ▶ Can be trivial

```
package StackTestCase1;

public class Vector {
    public void size() { return 1; }
    public boolean add(obj) { return true; } // do nothing
    public String removeElementAt(int loc) {
        return new String("testing");
}
```

```
public class Stack {
    public void push(Object obj)
        throws Exception {
        if (! imp.add(obj))
            throw new Exception();
    }

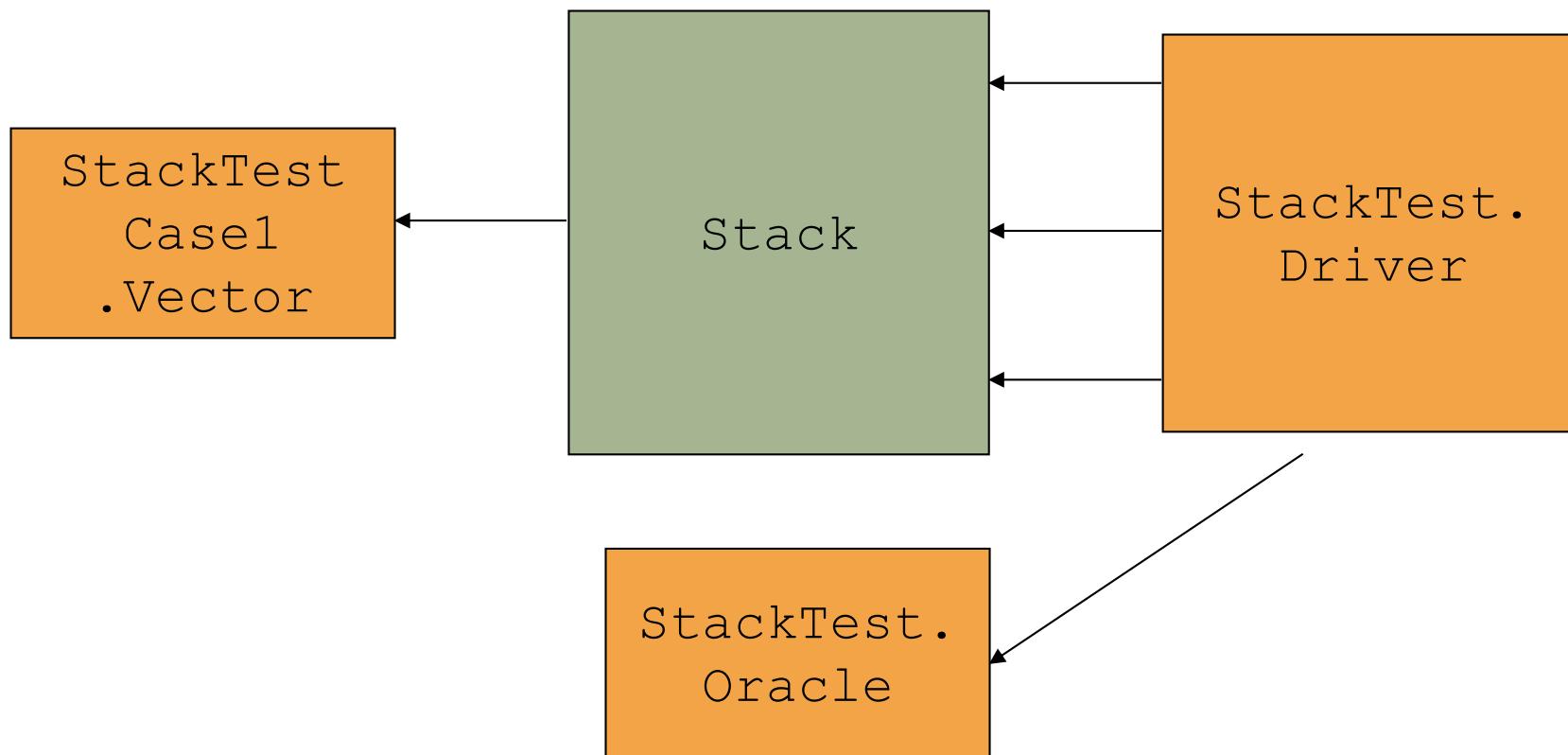
    public Object pop() throws Exception {
        if (imp.size()==0)
            throw new Exception();
        return imp.removeElementAt(size-1);
    }

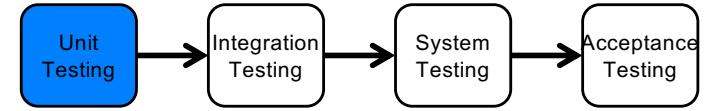
    ...
}

private Vector imp;
```

Summary: Example

Unit testing of Stack:





JUnit: Overview

- ▶ A Java framework for writing and running unit tests
 - ▶ Test cases and fixtures
 - ▶ Test suites
 - ▶ Test runner
- ▶ Implements the same principles eg. provides test driver and ability to know if a test has passed or failed
- ▶ Written by Kent Beck and Erich Gamma
- ▶ Written with “test first” and pattern-based development in mind
 - ▶ Tests written before code
 - ▶ Allows for regression testing
 - ▶ Facilitates refactoring
- ▶ JUnit is Open Source
 - ▶ www.junit.org
 - ▶ JUnit Version 5, released September 2017

JUnit testing

- ▶ A JUnit *test* is a method contained in a class which is only used for testing. This is called a *Test class*. To define that a certain method is a test method, annotate it with the `@Test` annotation.
 - ▶ Essentially this is the **Test Driver**
- ▶ This method executes the code under test. You use an *assert* method, provided by JUnit or another assert framework, to check an expected result versus the actual result. These method calls are typically called *asserts* or *assert statements*.
 - ▶ This carries out the functionality of the **Oracle**
 - ▶ There are various types of assertions like Boolean, Null, Identical etc.
- ▶ Still may need to isolate your unit under test using **Stubs**

JUnit (simple) example

- ▶ This test assumes that the MyClass class exists and has a multiply(int, int) method.

```
import static org.junit.jupiter.api.Assertions.assertEquals;  
import org.junit.jupiter.api.Test;  
public class MyTests {  
    @Test  
    public void multiplicationOfZeroIntegersShouldReturnZero() {  
        MyClass tester = new MyClass(); // MyClass is tested  
  
        // assert statements  
        assertEquals(0, tester.multiply(10, 0), "10 x 0 must be 0");  
        assertEquals(0, tester.multiply(0, 10), "0 x 10 must be 0");  
        assertEquals(0, tester.multiply(0, 0), "0 x 0 must be 0");  
    }  
}
```

Other JUnit features

- ▶ **Textual and GUI interface**
 - ▶ Displays status of tests
 - ▶ Displays stack trace when tests fail
- ▶ **Integrated with Maven and Continuous Integration**
 - ▶ <http://maven.apache.org>
 - ▶ Build and Release Management Tool
 - ▶ <https://continuum.apache.org>
 - ▶ Continous integration server for Java programs
 - ▶ All tests are run before release (regression tests)
 - ▶ Test results are advertised as a project report
- ▶ **Many specialized variants**
 - ▶ Unit testing of web applications
 - ▶ J2EE applications
 - ▶ Etc



The Test Driven Development philosophy

- ▶ The basic tenants are developing and implementing unit tests before writing a line of code
- ▶ Unit tests will and must fail up front
Code is developed after the test is developed.
- ▶ A unique idea that is still foreign to many developers

TDD steps

- ▶ Quickly add a test just enough code to fail test
- ▶ Run testsuite to ensure test fails (may choose to run a subset of suite)
- ▶ Update your functional code to ensure new test passes
- ▶ Rerun test suite and keep updating functional code until test passes
- ▶ Refactor and move on

Benefits of TDD

- ▶ Shortens the programming feedback
- ▶ Provides detailed (executable) specifications
- ▶ Promotes development of highquality code
- ▶ Provides concrete evidence that your code works
- ▶ Requires developers to prove it with code
- ▶ Provides finelygrained, concrete feedback (in mins)
- ▶ Ensures that your design is clean by focusing on creation of operations that are callable and testable
- ▶ Supports evolutionary development

TDD and Agile

- ▶ TDD implies agile
- ▶ Strong emphasis on testing
- ▶ Tests should span entire breadth of codebase
- ▶ Once all software is ready for delivery, all tests should pass
- ▶ Seen as a unique way to address modern challenges in software development

Principles of Agile development (a reminder)

- ▶ Continuous delivery
- ▶ Welcome changing requirements
- ▶ Deliver working software frequently
- ▶ Involve the business and developers throughout the project
- ▶ Build projects around motivated people
- ▶ Communication should be face-to-face
- ▶ Primary metric of progress is working software
- ▶ All participants should maintain a constant pace
- ▶ Continuous attention to technical excellence & good design
- ▶ Simplicity is essential
- ▶ Self organizing teams
- ▶ Periodic retrospective reviews

Testing in a more traditional (waterfall) environment

- ▶ Mostly implies a waterfall/bigbang process
- ▶ Should follow a structured approach ie.
 - ▶ unit -> integration -> system -> acceptance testing
- ▶ Very little emphasis on unit testing by developers
- ▶ Tests are sometimes developed as an afterthought
- ▶ Tests are mostly manual ie. not TDD
- ▶ Huge emphasis on QA team
- ▶ One view is that delivering quality software on time and within budget is almost accidental

Summary

- ▶ The difference between Unit and Integration testing
- ▶ Static and Dynamic unit testing
- ▶ Black and White box testing approaches
- ▶ Code-coverage tests (White-box)
- ▶ Unit testing
 - ▶ Selecting test cases
 - ▶ Components of unit testing and examples: Test driver, tested unit, stubs, oracle
 - ▶ JUNIT
- ▶ Test Driven Development approach
- ▶ Agile and waterfall approaches

Further reading

- ▶ Chapter 11 – Bruegge – Testing
- ▶ Chapter 8 – Pfleeger – Testing the Programs
- ▶ Introduction to Test Driven Development by Scott Ambler (<http://agiledata.org/essays/tdd.html>)



Software Evolution

- Maintenance vs. evolution
- Architectural erosion
- Laws of Software Evolution

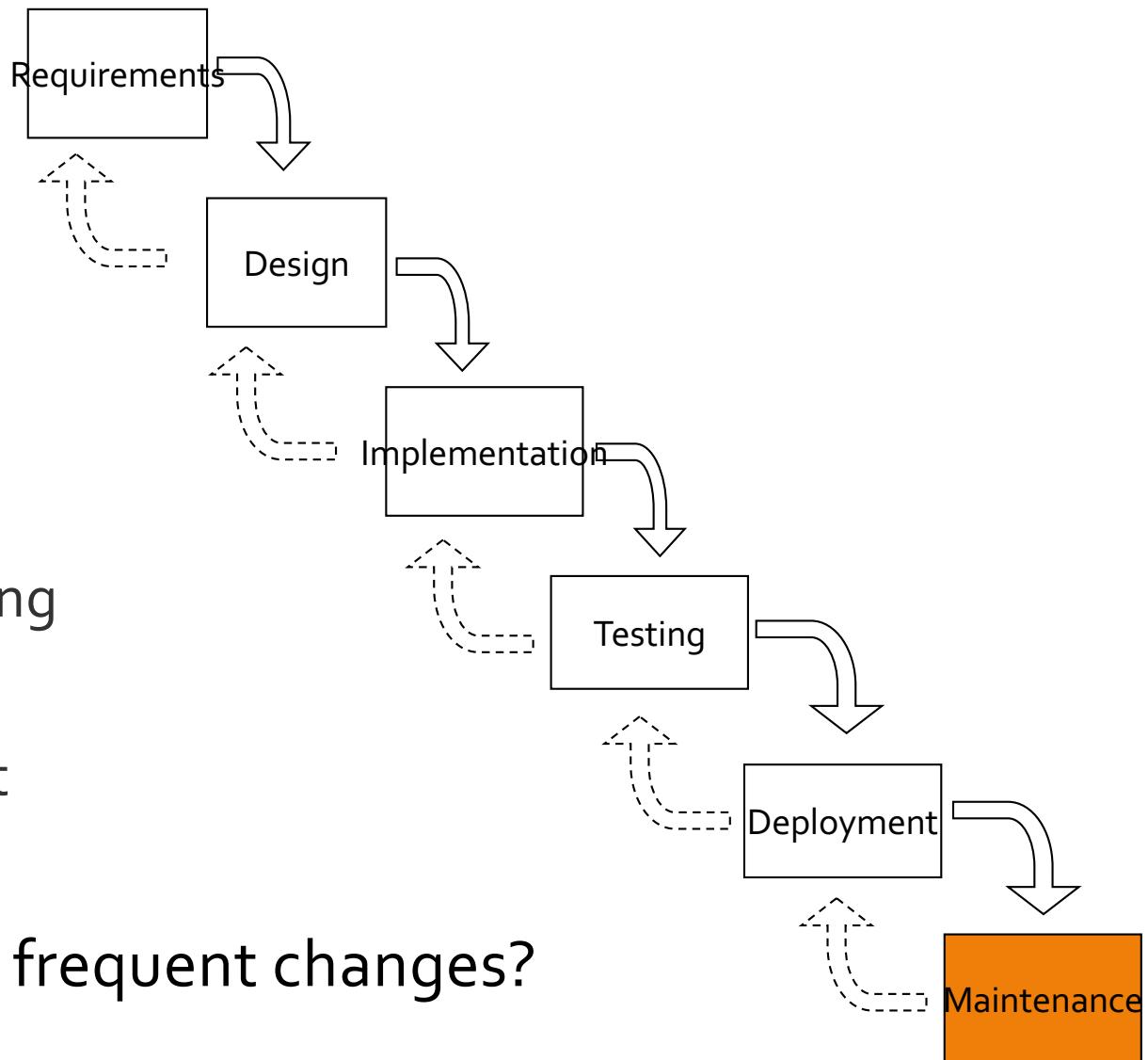
CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



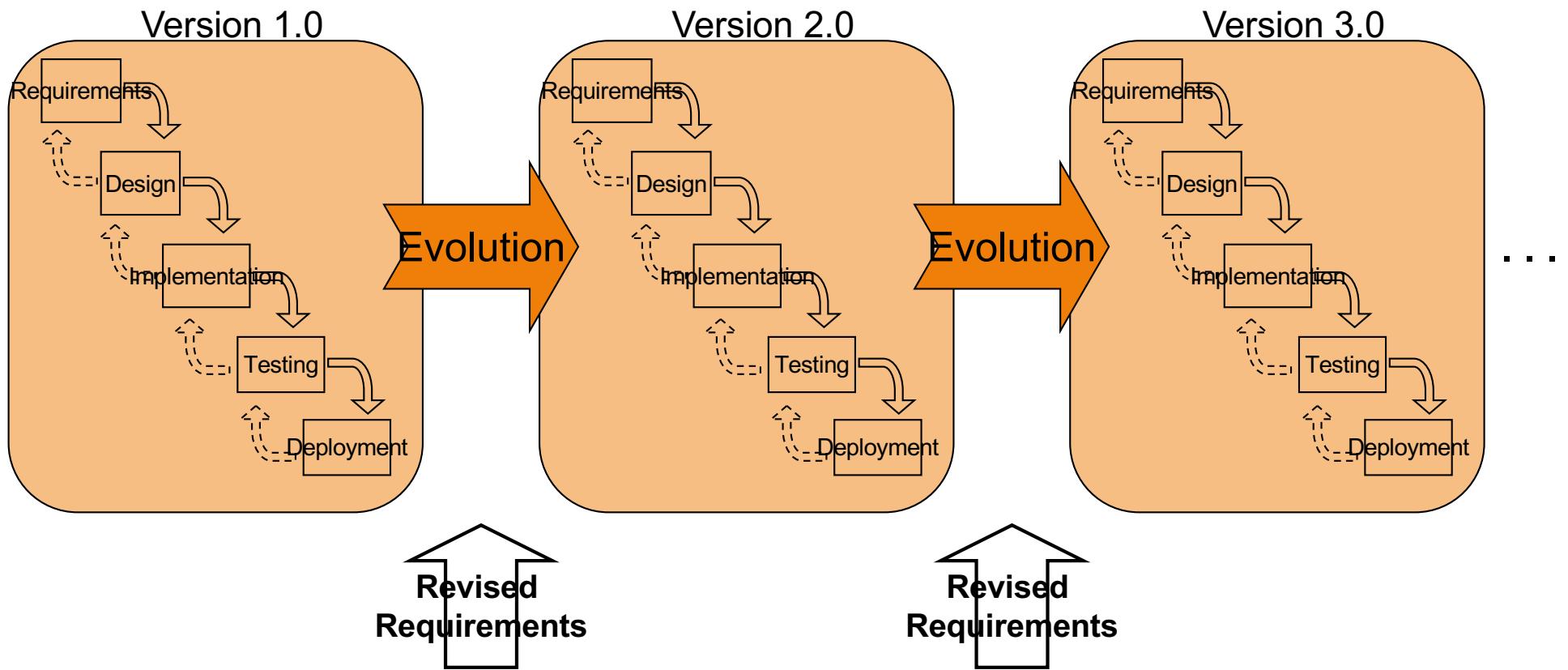
Reminder: Non-evolutionary lifecycle model: Waterfall Model

- ▶ Assumption: the entire project can be done in broad phases
 - ▶ Specify everything
 - ▶ Design everything
 - ▶ Implement everything
 - ▶ Test everything
 - ▶ Deploy final product
 - ▶ “**Maintain**”
- ▶ What about radical, frequent changes?



Reminder: Maintenance or evolution?

“Evolutionary” model: Reflects reality better!



Software maintenance

- ▶ A phase in the traditional waterfall model:
 - ▶ The “last” stage in the process
 - ▶ “Maintenance”: Modifying a program after it has been put into use
 - ▶ Not geared towards major changes
 - ▶ E.g.: Changes in the system’s architecture

→ Conclusion:

- The waterfall model (and the term “maintenance”) is suitable for small, “fixed” projects
- The waterfall model is also suited to very large projects which require rigorous project management



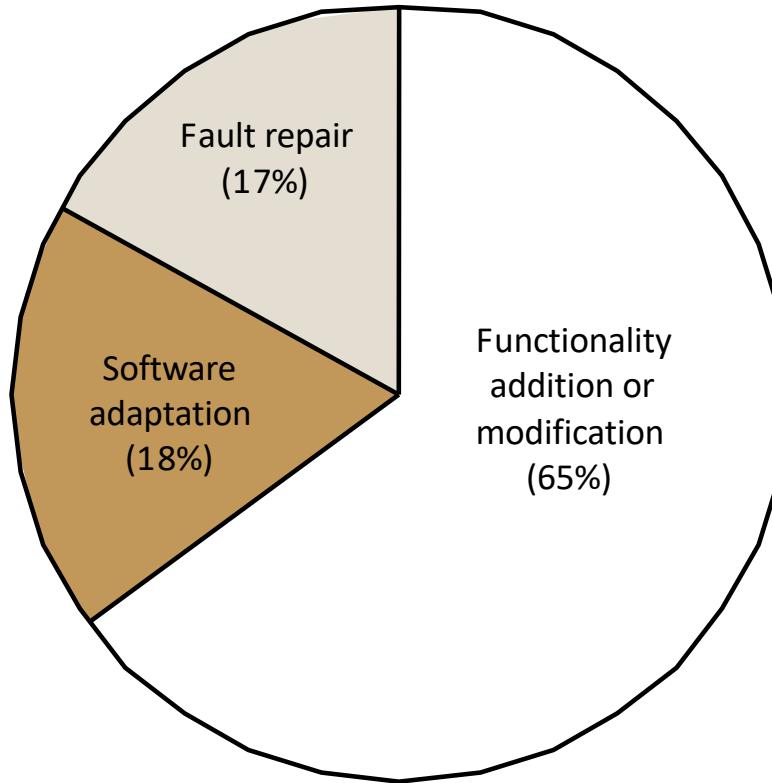
Types of changes

- ▶ **Bug fixing:** Faults discovered & must be fixed
- ▶ **Changes in the “operational environment”**
 - ▶ New hardware (computers, networks) added
 - ▶ Other components and software change
 - ▶ E.g., new operating system
- ▶ **Changes in customer’s requirements**
 - ▶ The business environment changes
 - ▶ Example for new requirement: Streaming video to mobile phones
 - ▶ Clients’ expectations increase
 - ▶ Performance, reliability & safety requirements may change
 - ▶ Competition: Change or lose!



Distribution of maintenance effort

From:
Sommerville, 2004



→ Conclusion: Software change is frequent, unexpected, and very expensive



Why is evolution so expensive?

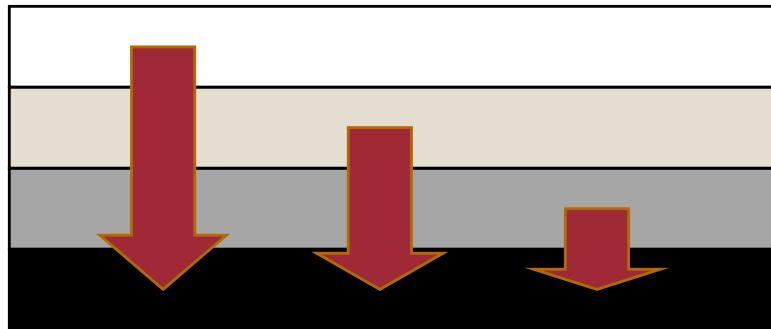
- ▶ Change prediction is very difficult
 - ▶ Which operating system will be used tomorrow?
 - ▶ Which hardware will replace the existing? How is it going to be different?
- ▶ Teams unstable
 - ▶ Short-term contracts with programmers
 - ▶ New programmers need be trained [Brooks 1975]
- ▶ Architectural erosion/drift [Perry & Wolf 92]
 - ▶ As programs age, their structure is degraded and they become harder to understand and change.
- ▶ Contractual responsibility
 - ▶ SW suppliers (usually) have no contractual responsibility for maintenance – no incentive to design for future change.



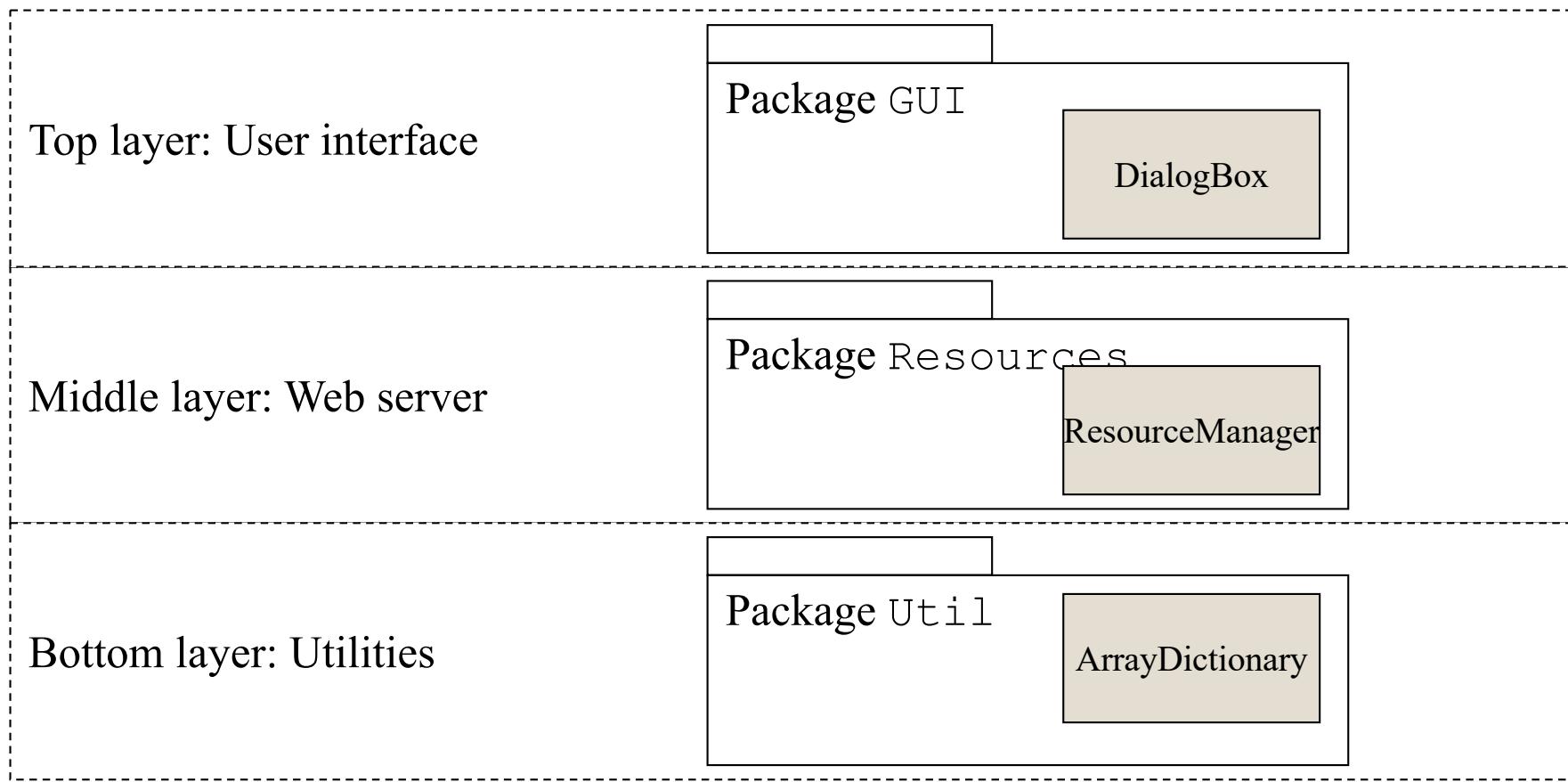
Example:

Architectural erosion in a layered architecture

- ▶ Principle: Prevent cycles of dependencies between every part of the system
 - ▶ Each module belongs to a “layer”
 - ▶ And
 - ▶ Each layer may access only “lower” layers



Architectural erosion: example (cont.)



Architectural erosion: example (cont.)

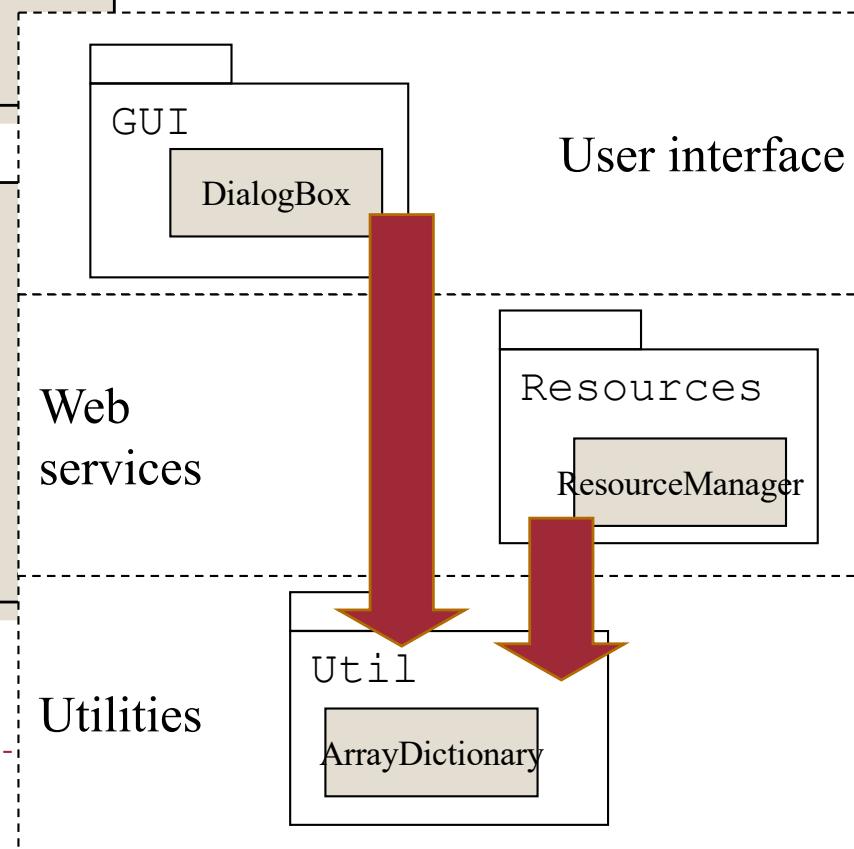
- ▶ Correct dependency: From upper to lower layer

```
package org.w3c.resource;

public class ResourceManager {
    private ArrayDictionary allResources;
    ...
}
```

```
package org.w3c.GUI;

public class DialogueBox {
    private ArrayDictionary
        Labels_to_Widgets;
    ...
}
```



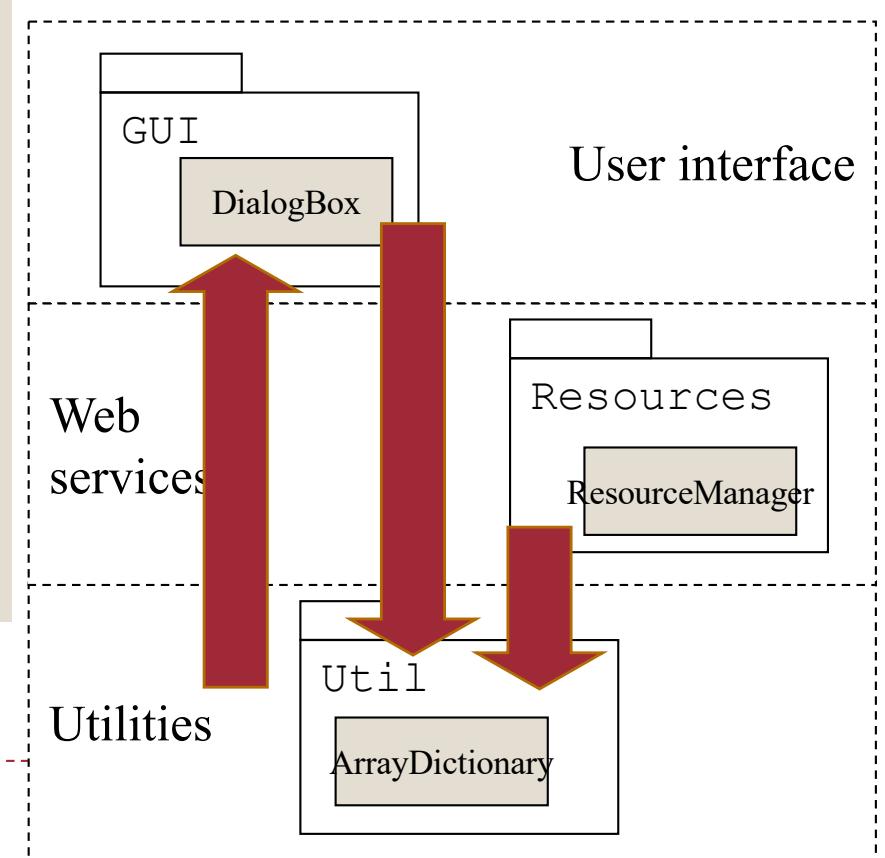
Architectural erosion: example (cont.)

- ▶ Change creating an incorrect dependency: From lower to upper layer

```
package org.w3c.util;

public class ArrayDictionary {
    public Object get(Object key) {
        if (cannot-find-object)
            // Report error:

        org.w3c.GUI.DialogBox.report("error...");
    }
    ...
}
```



Outcomes of architectural erosion

- ▶ The “architectural model” is incorrect
 - ▶ Maintenance is difficult: Programmers cannot find their way in the program
- ▶ The Program is “brittle”
 - ▶ Breaks with the simplest change
 - ▶ The “domino effect”



The Laws of Software Evolution

- ▶ **Manny Lehman**, the “Father of Software Evolution”, wrote many papers from the mid 70s onwards, proposing “Laws of Software Evolution” for “E-type systems”.
- ▶ Systems classified into:
 - ▶ S-type: formally specified and verified; static by definition
 - ▶ E-type: real-world system
- ▶ “Laws”: Reflect the cooperative activity of many individual and organisational behaviours.
 - ▶ Influenced by thermodynamics
 - ▶ Reflect established observations and empirical evidence
- ▶ “An emerging theory of software process and software evolution.”
 - ▶ i.e., work in progress



Lehman's laws

I. Continuing Change

- An E-type system must be continually adapted else it becomes progressively less satisfactory in use

II. Increasing Complexity

- As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it

III. Self regulation

- Global E-type system evolution processes are self-regulating and supports the ability to react to change.
(eg. an E-type systems growth inevitably slows as it grows older)

IV. Conservation of Organisational Stability

- Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime. Managers have no power?

V. Conservation of Familiarity

- In order for E-type systems to continue to evolve, its maintainers must possess and maintain a mastery of its subject matter and implementation.

VI. Continuing Growth

- The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over the system lifetime

VII. Declining Quality

- Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type system will appear to be declining as it is evolved

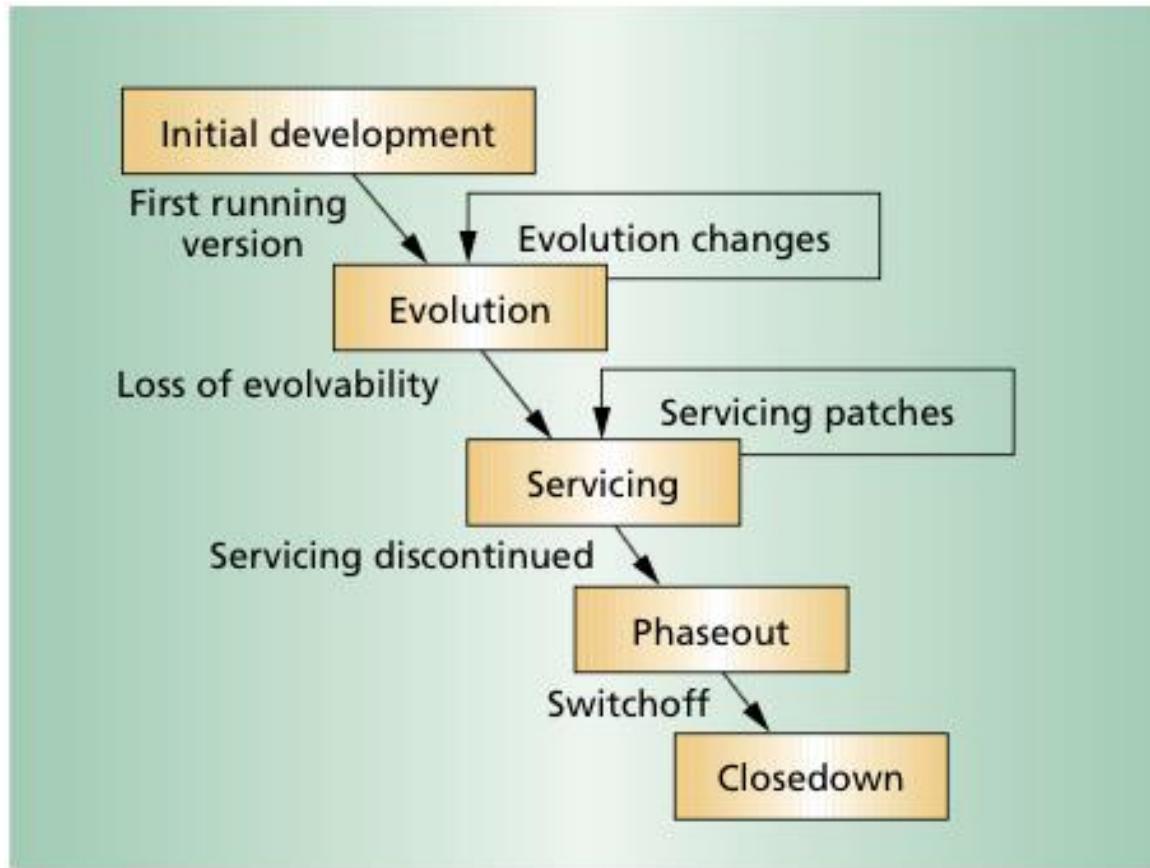
VIII. Feedback System

- In order to sustain continuous change, or evolution of a system, there must be a means to monitor the performance it will have.



I	Continuing Change	An E-type system must be continually adapted else it becomes progressively less satisfactory in use
II	Increasing Complexity	As an E-type system is evolved its complexity increases unless work is done to maintain or reduce it
III	Self regulation	Global E-type system evolution processes are self-regulating and supports the ability to react to change. (eg. an E-type systems growth inevitably slows as it grows older)
IV	Conservation of Organisational Stability	Average activity rate in an E-type process tends to remain constant over system lifetime or segments of that lifetime. Managers have no power?
V	Conservation of Familiarity	In order for E-type systems to continue to evolve, its maintainers must possess and maintain a mastery of its subject matter and implementation.
VI	Continuing Growth	The functional capability of E-type systems must be continually enhanced to maintain user satisfaction over the system lifetime
VII	Declining Quality	Unless rigorously adapted to take into account changes in the operational environment, the quality of an E-type system will appear to be declining as it is evolved
VIII	Feedback System	In order to sustain continuous change, or evolution of a system, there must be a means to monitor the performance it will have.

Staged model of software evolution*



* Bennett, K. H.; Rajlich, V. T.; Mazrul, R. Mohamad (1995). "Legacy System: Coping with success". *IEEE Software*. pp. 19–23.



Summary

- ▶ Maintenance as part of the waterfall process
- ▶ Evolutionary models reflect reality better
- ▶ Software change is frequent, unexpected, and very expensive
- ▶ Problems caused by architectural erosion
- ▶ The laws of software evolution
- ▶ The staged model of software evolution



Class exercises



Law no. I: *Continuing Change*

An E-type program that is used must be continually adapted else it becomes progressively less satisfactory

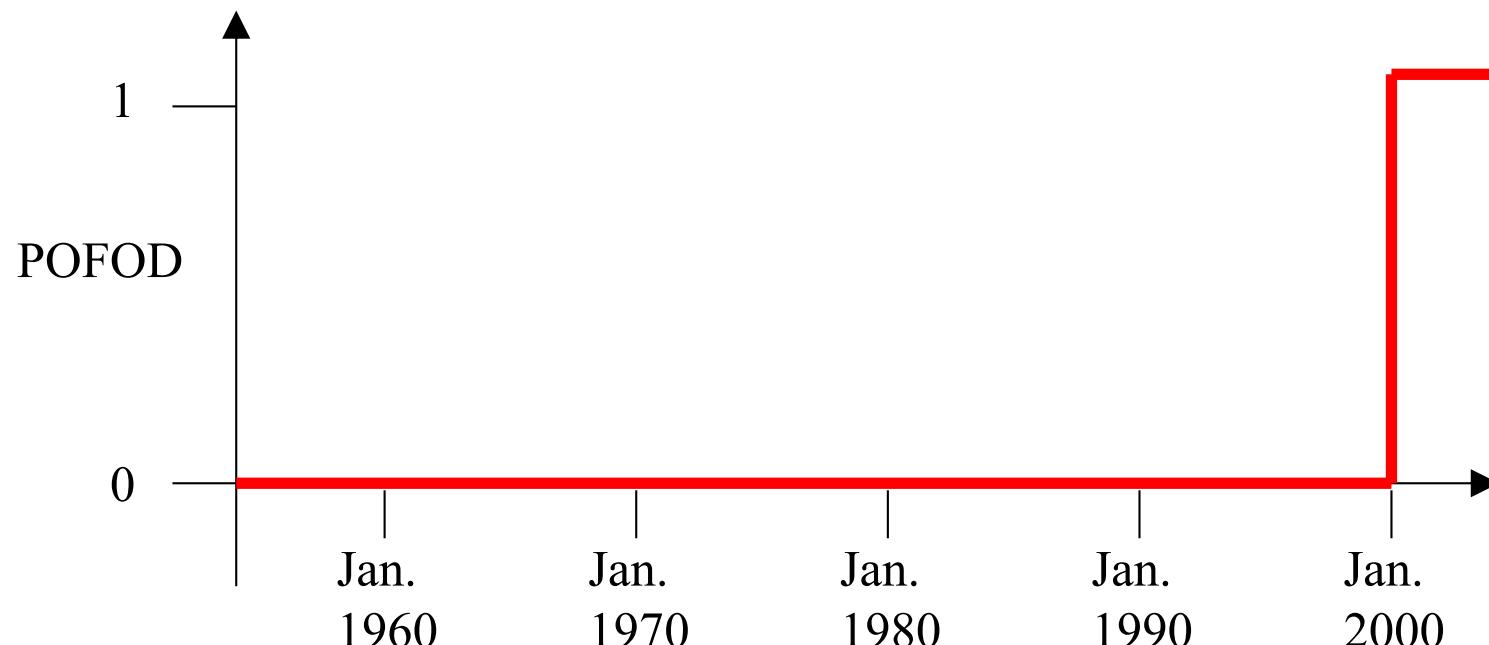
What are the reasons for this?

- ▶ Growth of the environment (“operational domain”)
- ▶ Hence, increasing mismatch between the system and its environment



Exercise: *Continuing Change*

- ▶ Sketch the reliability of a program that is not Y2K-compliant as a function of time before and after January 2000:



Law no. II - *Increasing Complexity*

As a program is evolved its complexity increases unless work is done to maintain or reduce it.

What are the reasons for this?

- ▶ Small changes are applied in a step-wise process
 - ▶ Each ‘patch’ makes sense locally, not globally
- ▶ Effort needed to reconcile accumulated changes with the overall performance goals



Law no. VI: *Continuing Growth*

Functional content of a program must be continually increased to maintain user satisfaction over its lifetime

- ▶ Implied by the first law, except with focus on *functional requirements*
- ▶ What is the motivation for this?
 - ▶ Competition
 - ▶ Customer demand
 - ▶ User dissatisfaction
 - ▶ “Omitted attributes will become the bottlenecks and irritants in usage as the user has to replace automated operation with human intervention. Hence they also lead to demand for change”



Exercise

- ▶ Find a solution to the architectural erosion problem in slide 11
 - ▶ How can a dialogue box be created without introducing a dependency of class util.ArrayDictionary on class gui.DialogBox?



Bibliography

- ▶ M. Lehman. "Laws of Software Evolution Revisited". Lecture Notes in Computer Science 1149 (Proc. 5th European Workshop on Software Process Technology), pp. 108–124. Berlin: Springer-Verlag, 1996.
- ▶ F. P. Brooks. The Mythical Man-Month. Reading: Addison-Wesley, 1975.
- ▶ D. E. Perry, A. L. Wolf. "Foundation for the Study of Software Architecture". ACM SIGSOFT Software Engineering Notes, Vol. 17, No. 4 (1992), pp. 40–52.



Bibliography II

- ▶ Ch. 32 'Software Maintenance' in: I. Sommerville.
"Software Engineering" 7th Edition. Reading: Addison-Wesley, 2004.
- ▶ Tom Mens, Lecture Notes in Software Evolution,
Université de Mons-Hainaut, Service de Génie Logiciel



Configuration and Build Management

- Purpose of Software Configuration Management (SCM)
- Software Configuration Management Activities
- Outline of a Software Configuration Management Plan
- SVN and GIT in more detail

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



Why Software Configuration Management ?

- ▶ The problem:
 - ▶ Multiple people have to work on software that is changing
 - ▶ More than one version of the software has to be supported:
 - ▶ Released systems
 - ▶ Custom configured systems (different functionality)
 - ▶ System(s) under development
 - ▶ Software on different machines & operating systems
- ▷ *Need for coordination*
- ▶ Software Configuration Management
 - ▶ manages evolving software systems
 - ▶ controls the costs involved in making changes to a system.



What is Software Configuration Management?

- ▶ Definition Software Configuration Management:
 - ▶ A set of management disciplines within a software engineering process to develop a baseline
 - ▶ Software Configuration Management encompasses the disciplines and techniques of initiating, evaluating and controlling change to software products during and after a software project
- ▶ Standards (approved by ANSI)
 - ▶ IEEE 828: Software Configuration Management Plans
 - ▶ IEEE 1042: Guide to Software Configuration Management.

Baseline: “*A specification or product that has been formally reviewed and agreed to by responsible management, that thereafter serves as the basis for further development, and can be changed only through formal change control procedures.*”

Configuration Management Activities (1)

► Software Configuration Management

Activities:

- ▶ Configuration item identification
- ▶ Promotion management
- ▶ Release management
- ▶ Branch management
- ▶ Variant management
- ▶ Change management

► No fixed order:

- ▶ These activities are usually performed in different ways (formally, informally) depending on the project type and life-cycle phase (research, development, maintenance).



Possible Selection of Configuration Items

- ▶ Problem Statement
- ▶ Software Project Management Plan (SPMP)
- ▶ Requirements Analysis Document (RAD)
- ▶ System Design Document (SDD)
- ▶ Project Agreement
- ▶ Object Design Document (ODD)
- ▶ Dynamic Model
- ▶ Object model
- ▶ Functional Model
- ▶ Unit tests
- ▶ Integration test strategy
- ▶ Source code
- ▶ API Specification
- ▶ Input data and data bases
- ▶ Test plan
- ▶ Test data
- ▶ Support software (part of the product)
- ▶ Support software (not part of the product)
- ▶ User manual
- ▶ Administrator manual

Configuration Management Roles

- ▶ Configuration Manager
 - ▶ Responsible for identifying configuration items
 - ▶ Also often responsible for defining the procedures for creating promotions and releases
- ▶ Change Control Board Member
 - ▶ Responsible for approving or rejecting change requests
- ▶ Developer
 - ▶ Creates promotions triggered by change requests or the normal activities of development. The developer checks in changes and resolves conflicts
- ▶ Auditor
 - ▶ Responsible for the selection and evaluation of promotions for release and for ensuring the consistency and completeness of this release.

Software Configuration Management Planning

- ▶ Software configuration management planning starts during the early phases of a project
- ▶ The outcome of the SCM planning phase is the *Software Configuration Management Plan (SCMP)* which might be extended or revised during the rest of the project
- ▶ The SCMP can either follow a public standard like the IEEE 828, or an internal (e.g. company specific) standard.

Outline of a Software Configuration Management Plan (SCMP, IEEE 828-2005)

1. Introduction

- ▶ Describes the Plan's purpose, scope of application, key terms, and references

2. SCM management (WHO?)

- ▶ Identifies the responsibilities and authorities for managing and accomplishing the planned SCM activities

3. SCM activities (WHAT?)

- ▶ Identifies all activities to be performed in applying to the project

4. SCM schedule (WHEN?)

- ▶ Establishes required coordination of SCM activities with other activities in the project

5. SCM resources (HOW?)

- ▶ Identifies tools and physical and human resources required for the execution of the Plan

6. SCM plan maintenance

- ▶ Identifies how the Plan will be kept current while in effect



Software Configuration Management Tools

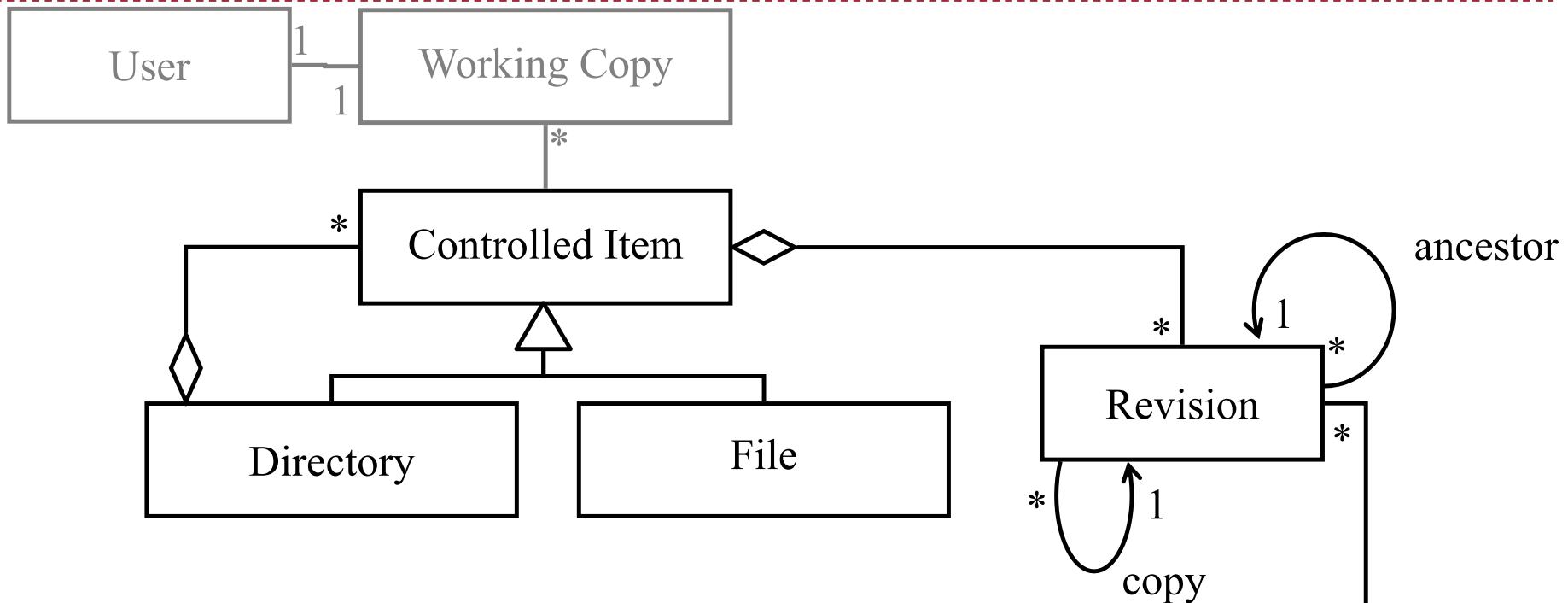
- ▶ RCS: The first on the block [Tichy 1975]
- ▶ CVS (Concurrent Version Control)
 - ▶ based on RCS, allows concurrency without locking
 - ▶ <http://www.nongnu.org/cvs/Subversion>
 - ▶ Based on CVS
 - ▶ Open Source Project (<http://subversion.apache.org>)
- ▶ Perforce
 - ▶ Repository server, keeps track of developer's activities
 - ▶ <http://www.perforce.com>
- ▶ ClearCase
 - ▶ Multiple servers, process modeling, policy check mechanisms
 - ▶ <https://www.ibm.com/uk-en/marketplace/rational-clearcase>
- ▶ GIT
 - ▶

Subversion

- ▶ Open Source Project <http://subversion.tigris.org/>
- ▶ Based on CVS
 - ▶ Subversion interface and features similar to CVS
 - ▶ Commands: checkout, add, delete, commit, diff
- ▶ Differences to CVS
 - ▶ Version controlled moving, renaming, copying of files and directories
 - ▶ Version controlled metadata of files and directories
- ▶ Server Options
 - ▶ Standalone installation
 - ▶ Integrated into the Apache webserver



Subversion Model (UML Class Diagram)



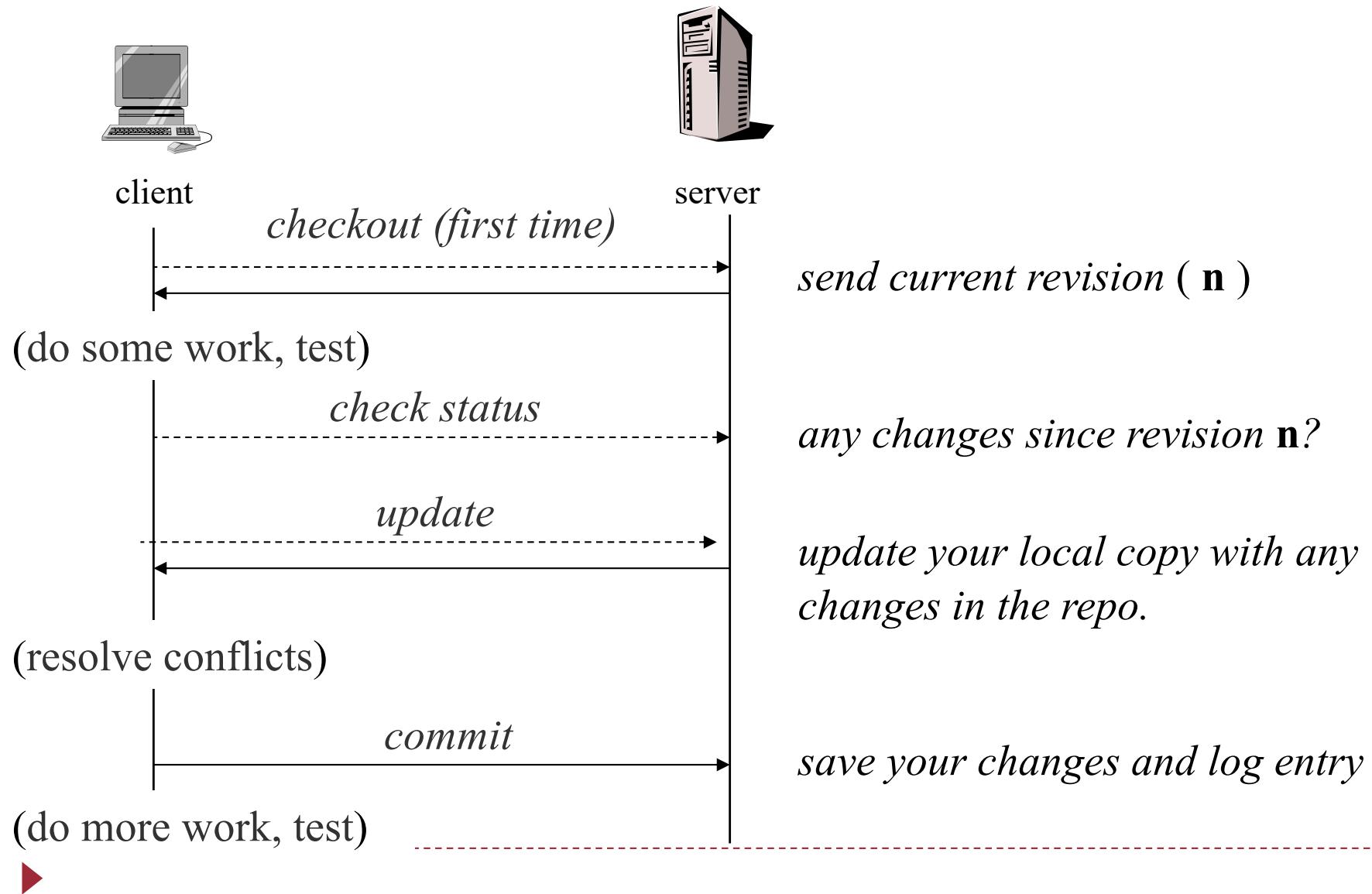
Similar functionality to CVS
File renaming/moving is controlled
Global version numbers
Branches and tags are copies
(can you spot the composite design pattern in this diagram?)



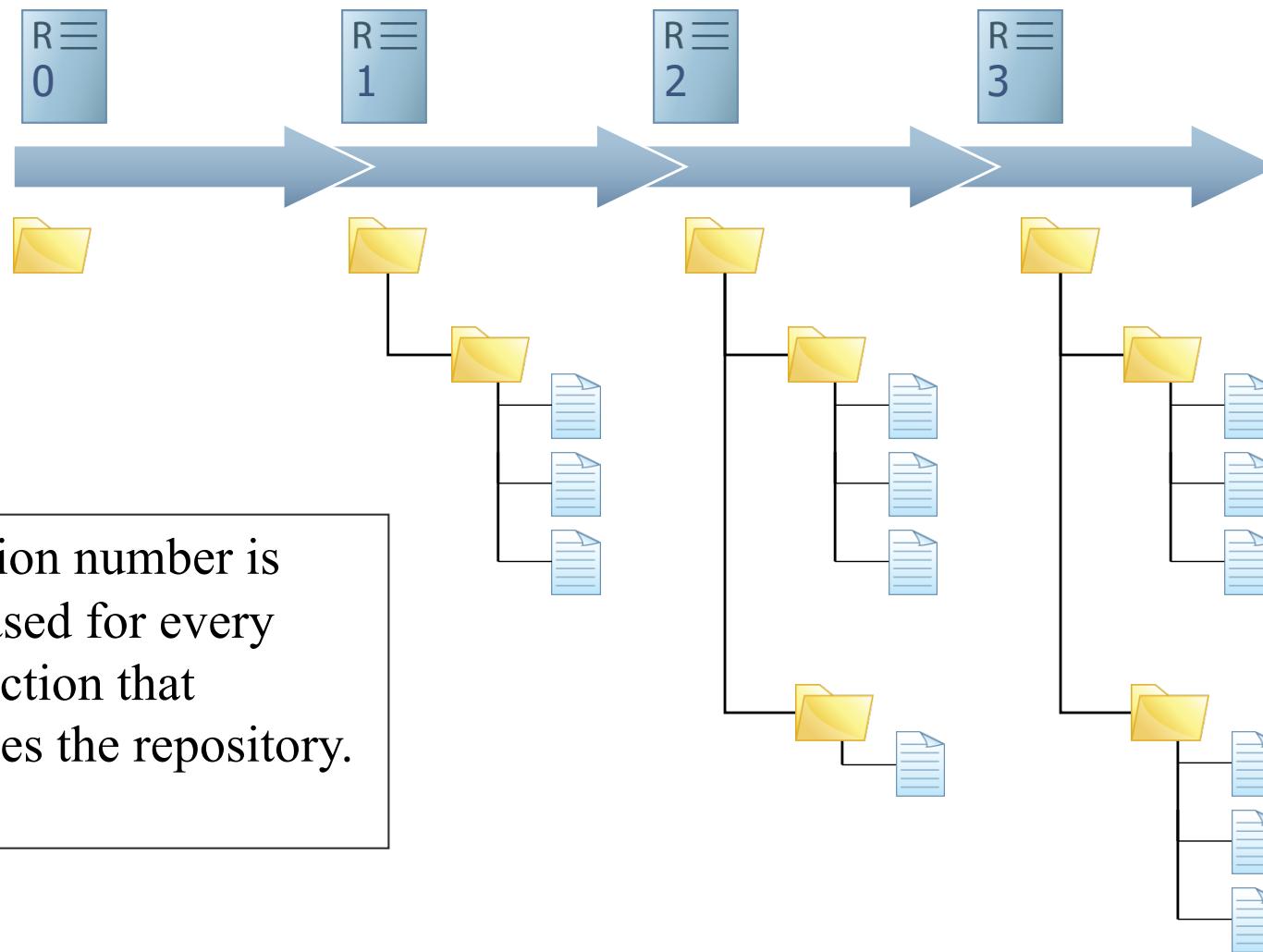
What is version control in SVN?

- ▶ manage documents *over time*
 - ▶ keep a history of all changes - multiple versions of every file
 - ▶ coordinate work of multiple authors
 - ▶ avoid conflicts ...and help *resolve* them
 - ▶ permissions: authenticate and control access to files
 - ▶ show differences between versions of a file
 - ▶ document changes -- reason for change
-

How to Use Version Control



Revision numbers

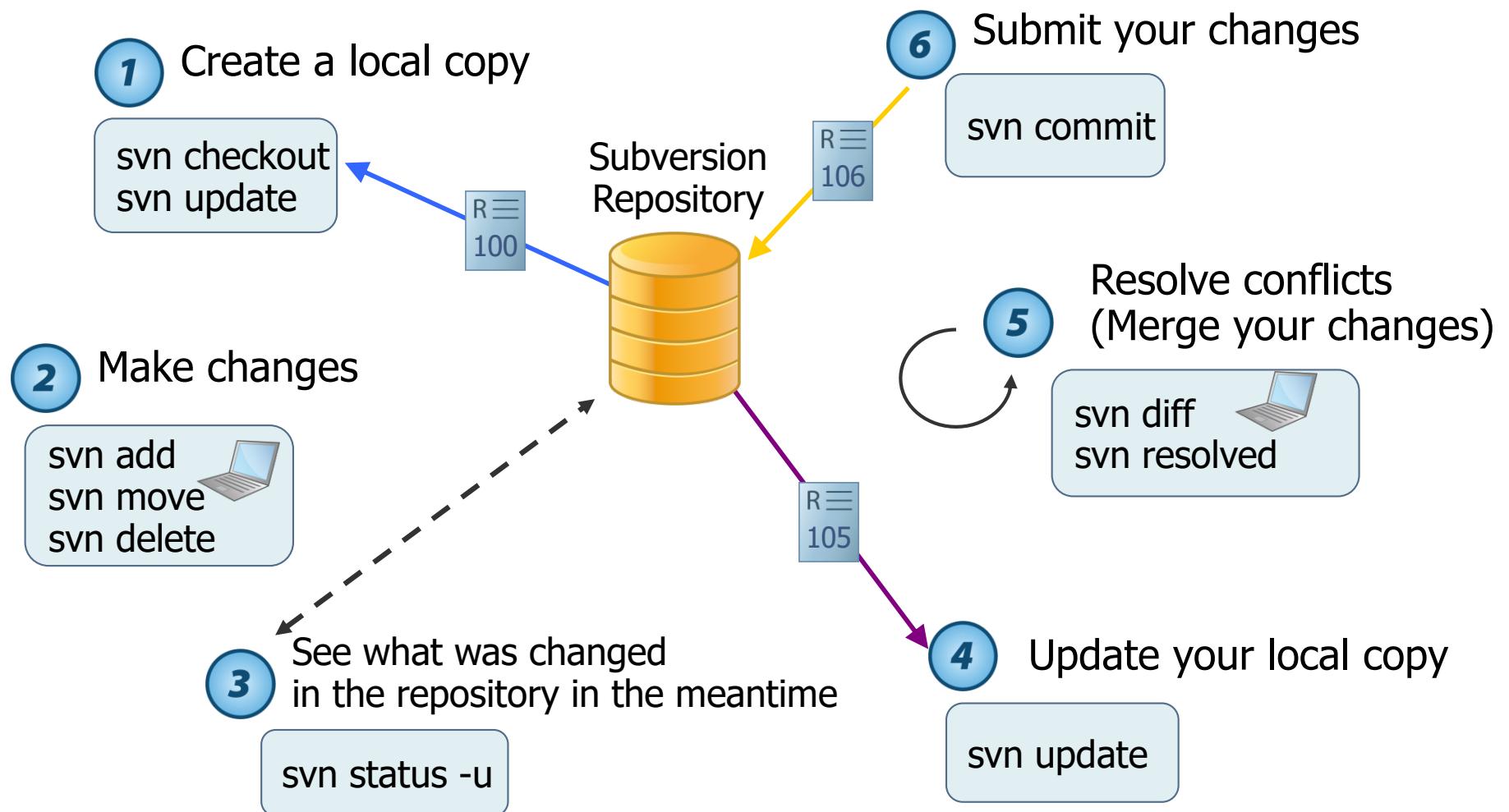


Properties of a Repository

- ▶ History of *all changes to files and directories.*
 - ▶ you can recover any previous version of a file
 - ▶ remembers "moved" and "deleted" files
- ▶ Access Control
 - ▶ Read / write permission for users and groups
 - ▶ Permissions can apply to repo, directory, or file
- ▶ Logging
 - ▶ author of the change
 - ▶ date of the change
 - ▶ reason for the change



The Work Cycle



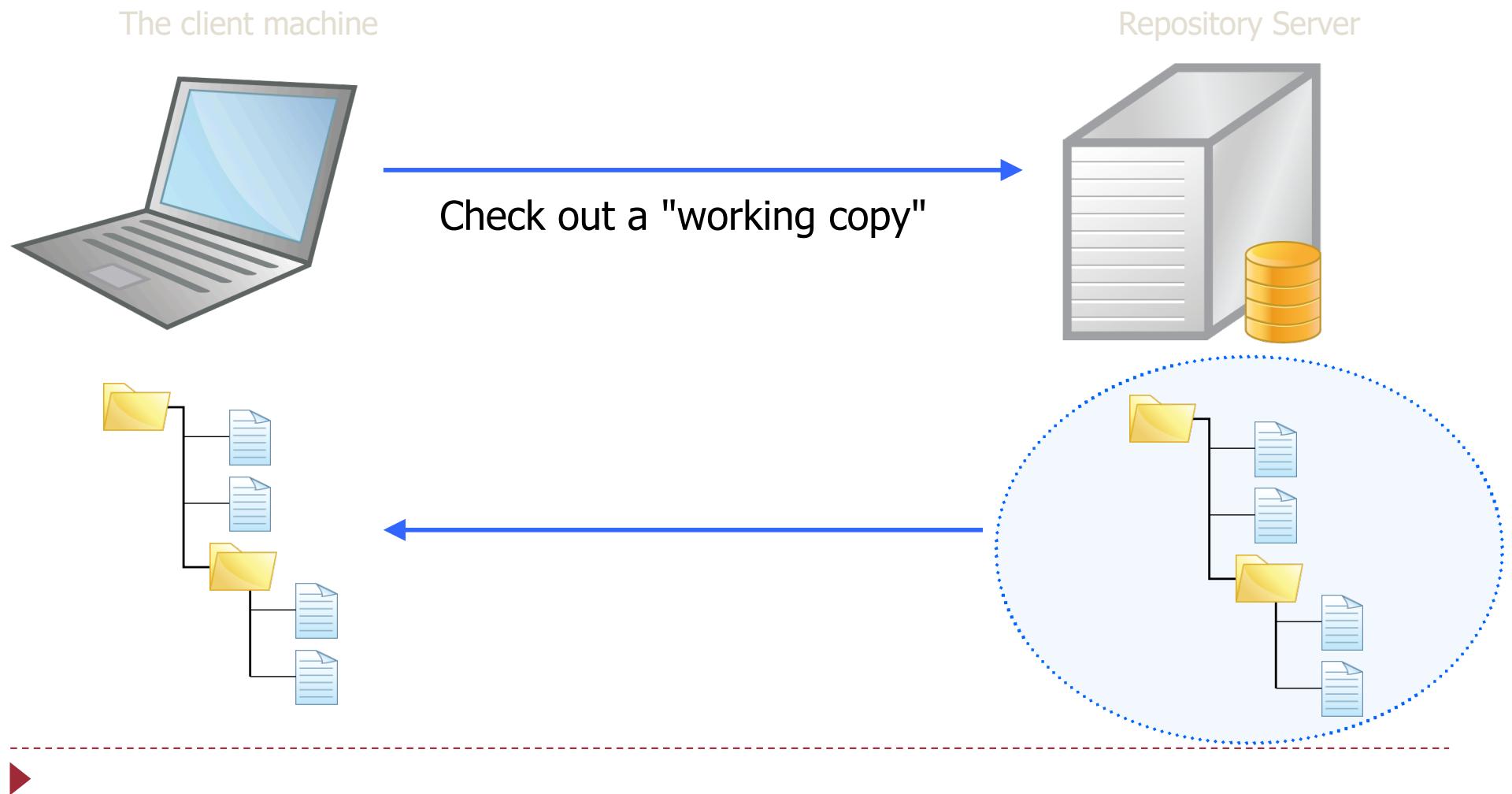
Logging a Revision

Content	what has changed?
Date	when did it change?
Author	who changed it?
Reason	why has it changed?

The diagram consists of a table with four rows. The first two rows ('Content' and 'Date') are grouped by a curly brace on the right side labeled 'SVN does this'. The last two rows ('Author' and 'Reason') are grouped by another curly brace on the right side labeled 'you enter this'.



(1) Check Out and the "Working Copy"



(2) Work Cycle: Edit Files

1. Edit files using anything you like.

2. Test Your Work.

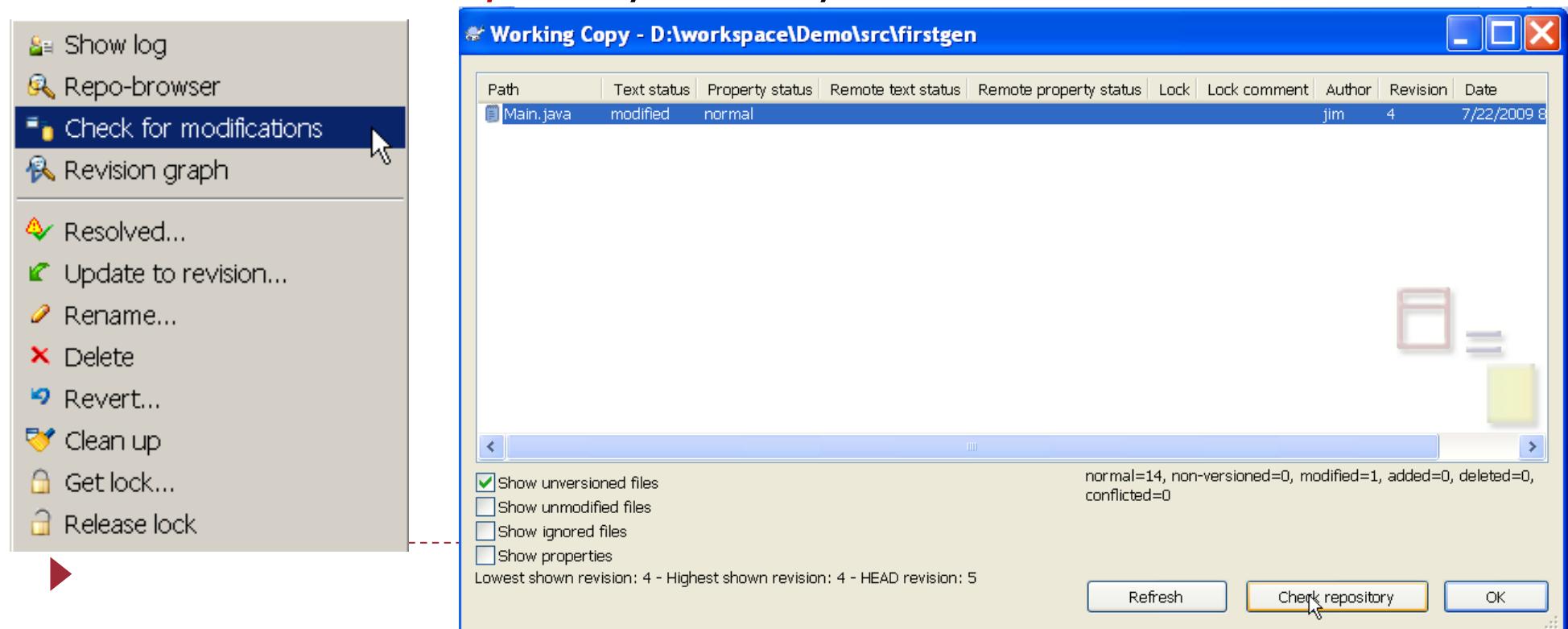
- ▶ Don't commit buggy code to the repository.

Then go to next step...



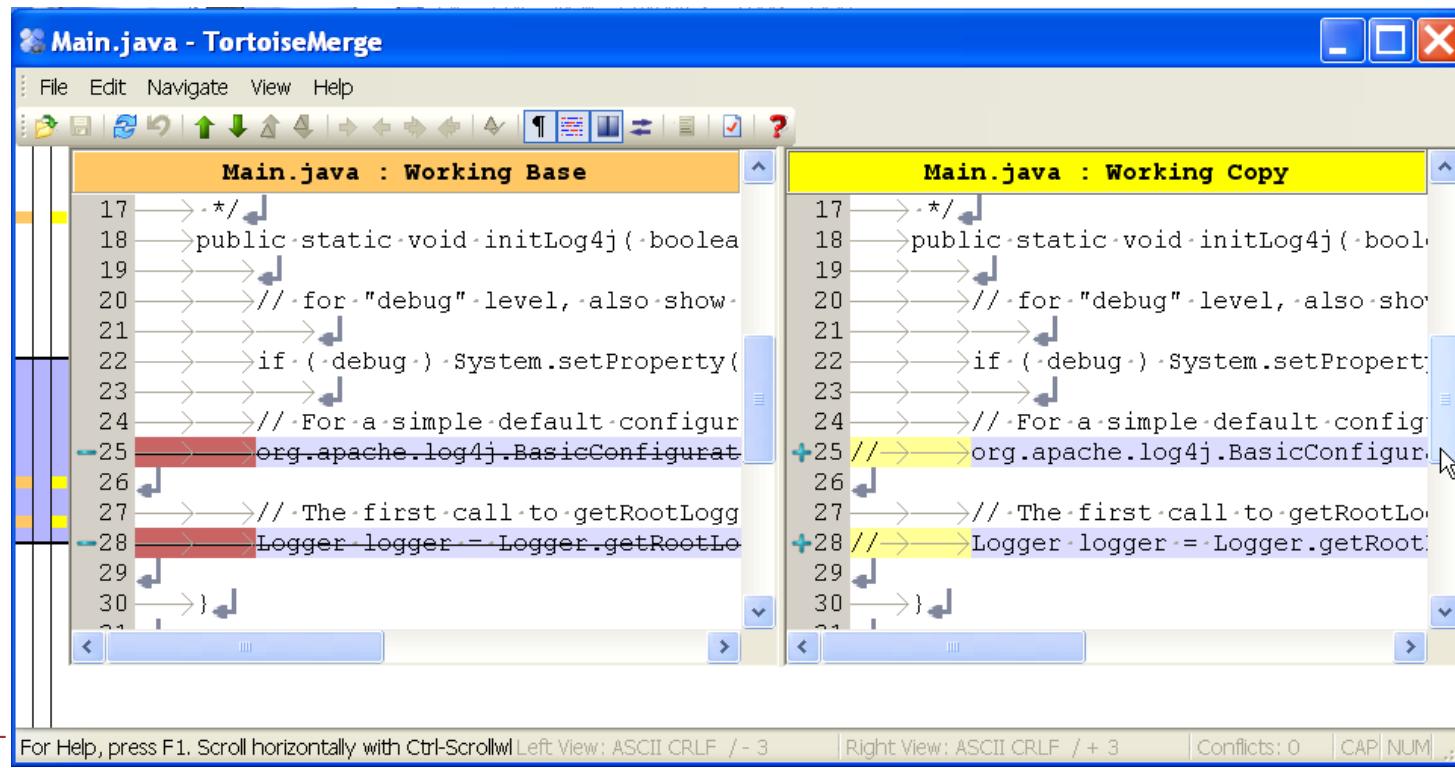
(3) Check for Updates

- ▶ Before "committing" your work, **check for updates** in the repository.
- ▶ Something might have changed while you were working.
- ▶ Subversion *requires* you to synchronize before commit.



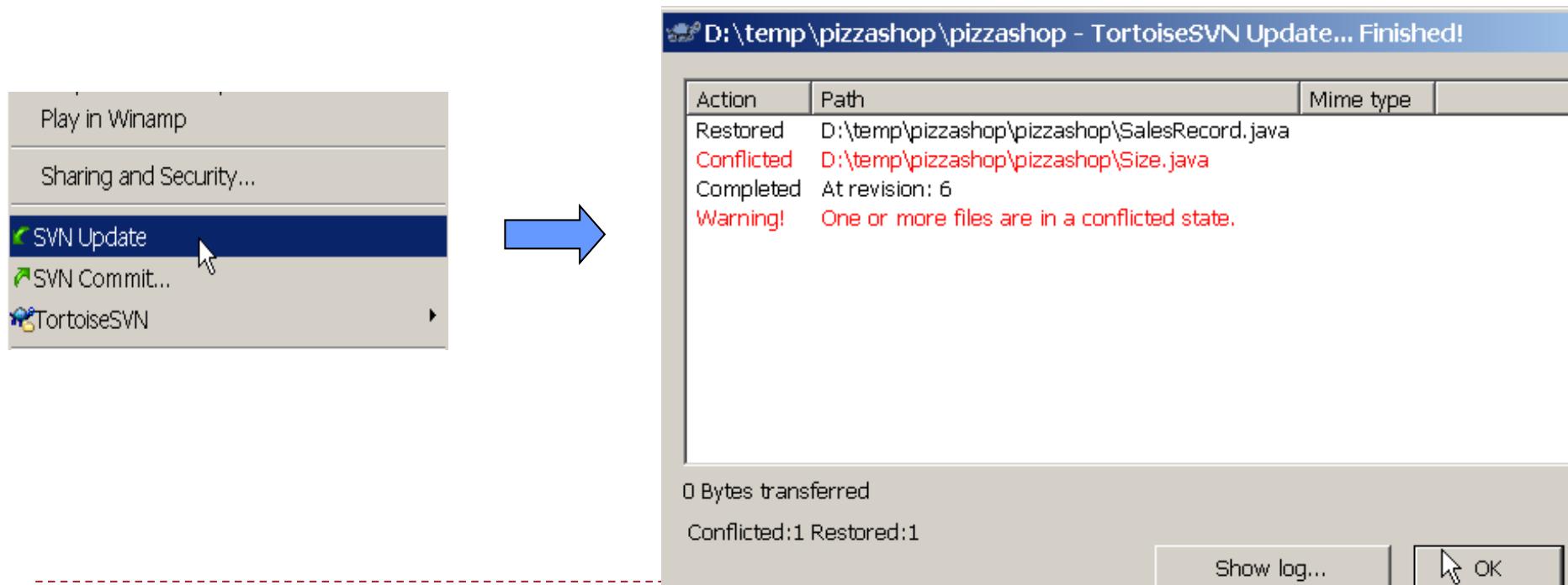
View Differences

- ▶ You can compare your version and the base or repo version.
- ▶ Select file, right-click => **Compare with base**



(4) Work Cycle: Update working copy

- ▶ If there are any changes on the server, then you should "update" your working copy before "commit"-ing your changes.



(5) Resolve Conflicts

- "Conflict" means you have made changes to a file, and the version in the repository has been changed, too.
- So there is a "conflict" between your work and work already in the repository.



Resolving Conflicts

The choices are:

- (1) merge local & remote changes into one file.
- (2) accept remote, discard your local changes.
- (3) override remote, use your local changes.

- ▶ After resolving all conflicts, mark the file as "resolved".
- ▶ Subversion will delete the extra copies.



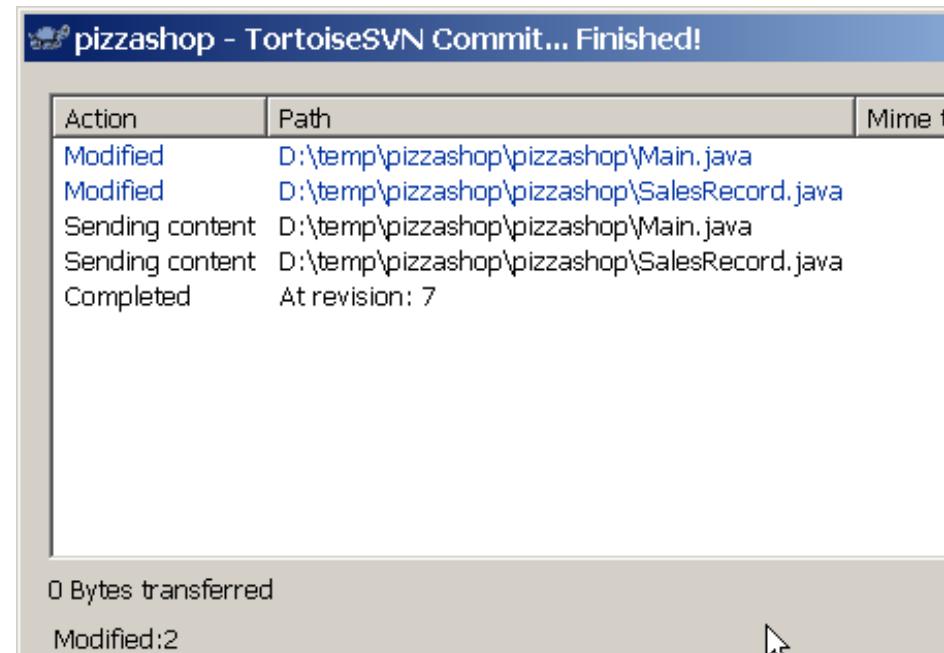
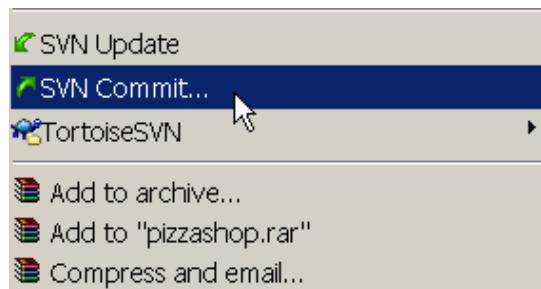
(6) Work Cycle: Commit

- ▶ After "update" and resolving conflicts, commit your work.

- ▶ Command line:

```
svn commit -m "description of your changes"
```

- ▶ TortoiseSVN:



Git vs SVN

- ▶ **Git is *much* faster than SVN**
 - ▶ Coded in C, which allows for a greater amount of optimization
 - ▶ Accomplishes much of the logic client side, thereby reducing time needed for communication
 - ▶ Developed to work on the Linux kernel, so that large project manipulation is at the forefront of the benchmarks



Git vs SVN

- ▶ Git is significantly smaller than SVN
 - ▶ All files are contained in a small decentralized .git file
 - ▶ In the case of Mozilla's projects, a Git repository was 30 times smaller than an identical SVN repository
 - ▶ Entire Linux kernel with 5 years of versioning contained in a single 1 GB .git file
 - ▶ SVN carries two complete copies of each file, while Git maintains a simple and separate 100 bytes of data per file, noting changes and supporting operations



Git vs SVN

- ▶ Git is more secure than SVN
 - ▶ All commits are uniquely hashed for both security and indexing purposes
 - ▶ Commits can be authenticated through numerous means
 - ▶ In the case of SSH commits, a key may be provided by both the client and server to guarantee authenticity and prevent against unauthorized access



Git vs SVN

- ▶ Git is decentralized
 - ▶ Each user contains an individual repository and can check commits against itself, allowing for detailed local revisioning
 - ▶ Being decentralized allows for easy replication and deployment
 - ▶ In this case, SVN relies on a single centralized repository and is unusable without



Git vs SVN

- ▶ **Git is flexible**
 - ▶ Due to it's decentralized nature, Git commits can be stored locally, or committed through HTTP, SSH, FTP, or even by Email
 - ▶ No need for a centralized repository
 - ▶ Developed as a command line utility, which allows a large amount of features to be built and customized on top of it



Git vs SVN

▶ Data Assurance

- ▶ A checksum is performed on both upload and download to ensure sure that the file hasn't been corrupted.
- ▶ Commit IDs are generated upon each commit
 - ▶ Linked list style of commits
 - ▶ Each commit is linked to the next, so that if something in the history was changed, each following commit will be rebranded to indicate the modification



Git vs SVN

► Branching

- ▶ Git allows the usage of advanced branching mechanisms and procedures
- ▶ Individual divisions of the code can be separated and developed separately within separate branches of the code
- ▶ Branches can allow for the separation of work between developers, or even for disposable experimentation
- ▶ Branching is a precursor and a component of the merging process



Git vs SVN

► Merging

- ▶ Content of the files is tracked rather than the file itself
 - ▶ This allows for a greater element of tracking and a smarter and more automated process of merging
 - ▶ SVN is unable to accomplish this, and will throw a conflict if a file name is changed and differs from the name in the central repository
 - ▶ Git is able to solve this problem with its use of managing a local repository and tracking individual changes to the code



What is GitLab

- ▶ A web interface for Git
- ▶ Provides additional features on top of a Git repository
- ▶ Developed as a Github clone for self-hosting
- ▶ Allows for access to the repository from a web browser
- ▶ Issue and milestone tracking implemented
- ▶ Support for attachments and code snippets
- ▶ Integration of a wiki and wall for project documentation



Use of GitLab

▶ Issue Tracking

- ▶ Can assign small parts of the project to team members through GitLab
- ▶ This allows the group to know exactly what needs worked on
- ▶ Bugs can also be submitted as an issue, and assigned to a particular developer to address
- ▶ Issues can be closed in a ticketing like system to show which parts of the project have been completed
- ▶ Milestones can be created with issues assigned to them, and a chosen due date applied



Use of GitLab

▶ Backup

- ▶ “A file does not exist unless it is present in multiple geographical locations”
- ▶ With a local repository on each developer machine, development server, and GitLab server, can have a great amount of data redundancy in the event of a failure, either software or hardware
- ▶ The repository can be stored on the cloud in addition to using a local server
- ▶ Able to pull old versions of the project as well



Summary

- Purpose of Software Configuration Management (SCM)
 - An essential activity in any project with more than 1 developer
- Software Configuration Management Activities
 - ▶ Configuration item identification
 - ▶ Promotion management
 - ▶ Release management
 - ▶ Branch management
 - ▶ Variant management
 - ▶ Change management
- Outline of a Software Configuration Management Plan
- Outline of using SVN and GIT

CE202 Module overview reminder

Autumn term

- o. Introduction
- 1. Modelling notations
 - ▶ UML diagrams: use-case, class, interaction diagrams; type diagrams; statecharts
- 2. Requirements engineering
 - ▶ Use-case analysis
 - ▶ Object-oriented analysis
- 3. Software design
 - ▶ Principles of software design
 - ▶ Object-oriented design
 - ▶ Design patterns
- 4. Software validation
 - ▶ Software architecture
 - ▶ Inspection
 - ▶ Typing
 - ▶ Testing
 - ▶ Design by Contract
- 5. Software reliability
 - ▶ Probability
 - ▶ Metrics
- 6. Software evolution
 - ▶ Laws of software evolution
 - ▶ Configuration management

Assignment no. 1: requirements and design



Progress test

That completes the material for CE202!

- ▶ No lecture or class next week (week 11)
- ▶ Aiming to get the marks for assignment 1 completed by by Friday 13th December
- ▶ The progress test is on Wednesday 11th December, 12pm 5.300 B, EBS.2.2, LTBo2, LTBo4
- ▶ In the revision lecture next April I will be giving out hints for what to focus on for the exam - please ensure you attend!
- ▶ Have a good break!

Progress Test

- ▶ **Wednesday 11th December 2019**
- ▶ 12pm in 5.300 B, EBS.2.2, LTBo2, LTBo4 (check your schedule)
- ▶ 40 minutes, 20 multi-choice questions, 1 correct answer per question
- ▶ Worth 20%
- ▶ Please arrive at 11.50pm so we can start promptly
- ▶ Late arrivers will not be given extra time!
- ▶ Bring pencils and a rubber
- ▶ If you finish early remain seated and quiet until the end
- ▶ Covers everything we have covered this term



Progress Test: example question

Which of the following statements best describes the goal of 'Validation testing':

[A] To detect defects (reveal problems)

[B] To ensure the system meets the client's expectations

[C] To ensure that the system handles extreme values correctly

[D] To prove that there are no bugs in the system

The answer in this case is [B]



End

- ▶ See Bruegge chapter 13 for more detail on the topic of this lecture



Software reliability: probability

- Calculating probability of orthogonal events
- Elementary probability
- Calculating the Probability of Failure on Demand (POFOD)

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex



What is probability?

- ▶ A real number in the range 0..1
- ▶ Represents the chance that a certain event will occur
- ▶ Examples:
 - ▶ The probability of “heads” in a coin toss: 0.5
 - ▶ Represents a 50%-50% chance that it be “heads”
 - ▶ The probability for rain tomorrow: 0.14
 - ▶ Based on statistical analysis, meter/map readings...
 - ▶ The probability for a student to drop out before graduation is 0.4
 - ▶ Based on numbers of students who dropped out before graduation

How is probability calculated?

- ▶ Difficulty: Prediction under uncertainty
 - ▶ How can we tell if it will rain tomorrow?
 - ▶ How can we tell if the coin toss will result in “heads”?
 - ▶ How can we tell if ‘A Smith’ will drop-out before graduation?
- ▶ We cannot!
- ▶ All we can do is: “predict” chances based on history

Calculating probability of equiprobable discrete events

- ▶ When all events have equal probability:

$$\frac{\text{\#suitable events}}{\text{\#possible events}}$$

Prob. of discrete (equiprobable) events: example

- ▶ Probability for “head” in one coin toss:

$$\frac{H}{H/T} = \frac{1}{2} = 0.5$$

- ▶ Probability for all “heads” in two coin tosses:

- ▶ Possible events:

- ▶ head, head
 - ▶ Head, tail
 - ▶ Tail, head
 - ▶ Tail, tail

$$\frac{HH}{HH/HT/TH/TT} = \frac{1}{4} = 0.25$$

Context: Probability of Failure on Demand

- ▶ **POFOD (probability of failure on demand)**
- ▶ This is the probability that the system will respond correctly when a request is made for service at a given point in time. This metric is used for systems where demands for service are intermittent and relatively infrequent over the lifetime of the system.
- ▶ Example - The likelihood that the system will fail when a user requests service. A biometric authentication device that fails to correctly identify or reject users on an average of **once out of a hundred** times has a POFOD of **1%**.



Probability of Failure on Demand

- ▶ **POFOD (probability of failure on demand)**
 - ▶ Measured using the history
 - ▶ Number of failed tests divided by the total number of tests

$$\frac{\text{\#failed tests}}{\text{\#tests}} = \frac{2}{20} = 0.1$$



Probability of complementary events

- ▶ If p_e is the probability that an event e occurs, what is the probability that e will NOT occur?

$$1 - p_e$$

NOT probability: examples

- ▶ Probability for “head” NOT to appear in one coin toss:

$$1 - p_{\text{head}} = 1 - \frac{1}{2} = 0.5$$

- ▶ Probability for anything OTHER THAN two “heads” in two coin tosses:

$$1 - p_{\text{head,head}} = 1 - \frac{1}{4} = 0.75$$

- ▶ Probability for no failure during execution:

$$1 - p_{\text{failure}} = 1 - \frac{2}{20} = 0.9$$

Probability: lottery example

- ▶ No. of lottery tickets: 2,000
- ▶ No. of prizes: 20
- ▶ Probability of winning the lottery:

$$\frac{20}{2,000} = 0.01$$

- ▶ Probability of NOT winning in the lottery:

- ▶ Calculation 1:

$$1 - p_{win} = 0.99$$

- ▶ Calculation 2:

$$\frac{1,980}{2,000} = \frac{99}{100} = 0.99$$

AND Probability of orthogonal (independent) events

- ▶ *Probability of event P1 AND P2 happening*
- ▶ P_1
 - ▶ probability of event #1
- ▶ P_2
 - ▶ probability of event #2
- ▶ $P_1 \times P_2$
 - ▶ probability of event #1 AND event #2

AND probability: examples

- ▶ Probability of getting the number 6 in tossing a die: $\frac{1}{6}$
- ▶ Probability of getting the numbers 6, 6 in tossing two dice:
$$\frac{1}{6} \times \frac{1}{6} = \frac{1}{36}$$
- ▶ Probability of NOT winning the lottery: 0.99
- ▶ Probability of NOT winning the lottery twice: 0.99x0.99

Example: software failure

- ▶ A PRINT operation from MS Outlook will succeed if both software components do not fail:
 - ▶ MS Outlook (failure probability: 0.1)
 - ▶ Printer driver (failure probability: 0.2)
- ▶ What is the probability that PRINT succeeds?

$$\begin{aligned} p_{\text{success}} &= p_{\text{succOutlook}} \times p_{\text{succPDF}} = \\ &= (1 - 0.1) \times (1 - 0.2) = 0.72 \end{aligned}$$

- ▶ What is the probability that PRINT fails?

$$p_{\text{fail}} = 1 - p_{\text{success}} = 0.28$$

OR probability of mutually exclusive events

- ▶ *Probability of event P1 OR event P2*
- ▶ P_1
 - ▶ probability of event #1
- ▶ P_2
 - ▶ probability of event #2
- ▶ Events are mutually exclusive
- ▶ $P_1 + P_2$
 - ▶ probability of event #1 OR event #2

OR probability: examples

- ▶ Probability for “head” OR tail to appear in one coin toss:
 - ▶ $0.5 + 0.5 = 1$
- ▶ Probability for anything OTHER THAN two “heads” in two coin tosses:
 - ▶ $P_{TT} + P_{TH} + P_{HT} = 0.25 + 0.25 + 0.25 = 0.75$
- ▶ Also can be calculated using the NOT rule:
 - ▶ Same as the probability of NOT getting two “heads”
 - ▶ $1 - P_{HH} = 1 - 0.25 = 0.75$

Summary

- ▶ Probability of (equiprobable, discrete) event:

$$\frac{\text{\#suitable events}}{\text{\#possible events}}$$

- ▶ NOT probability (complement):

$$1 - p_e$$

- ▶ AND probability (orthogonal events):

$$p_1 \times p_2$$

- ▶ OR probability (mutually exclusive events):

$$p_1 + p_2$$

- ▶ POFOD

$$\frac{\text{\#failed tests}}{\text{\#tests}} = \frac{2}{20} = 0.1$$

- ▶ likelihood that the system will fail when a user requests service

Exercise

- ▶ The probability of an error for each cell (mobile phone network unit) is 0.1. Your mobile phone needs only one cell to connect to the network, and it is situated within the range of three cells. You start your phone and check connection (which takes about a second). What is the probability that this operation fails?

