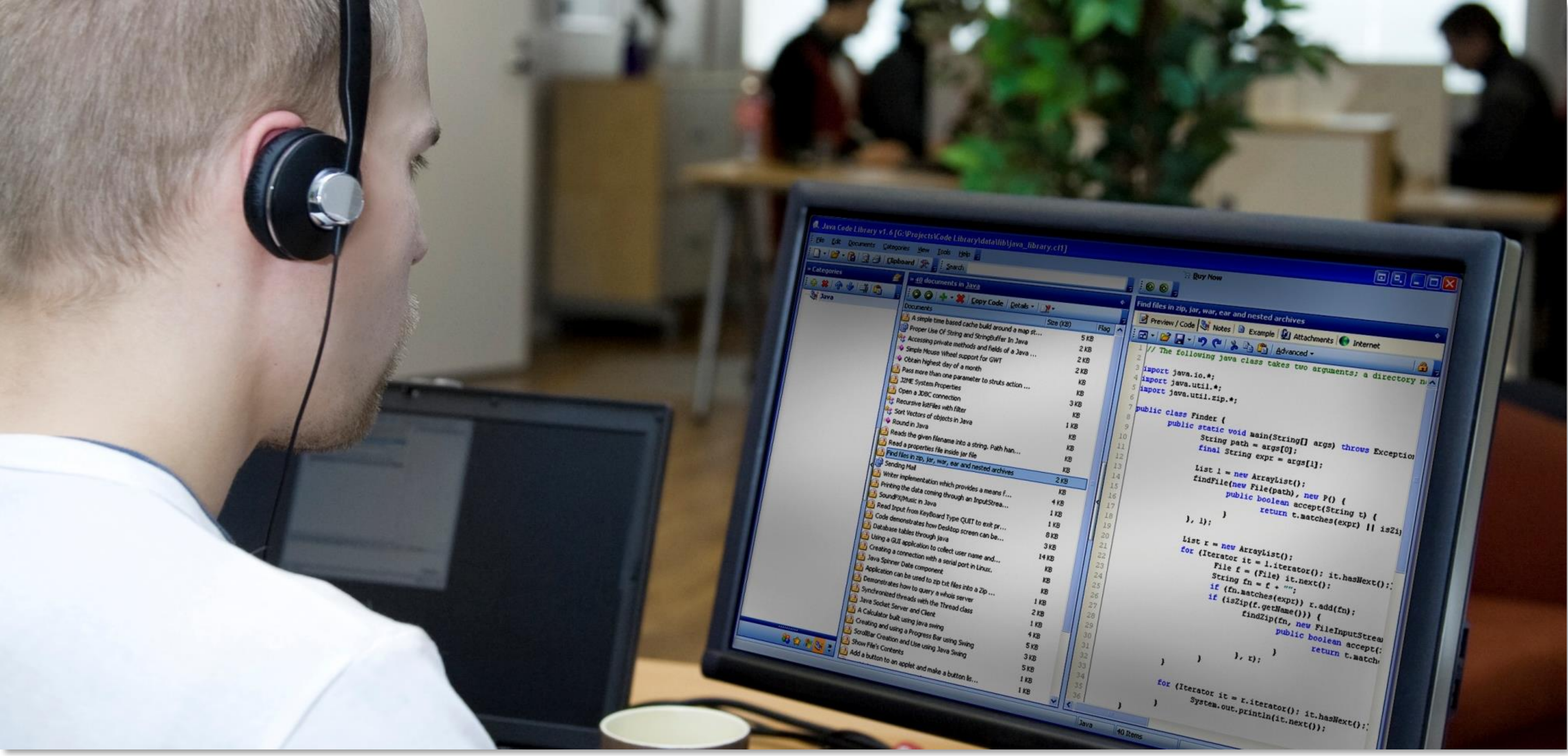


Course Introduction

Software development tools *(Architecture and Design)*

General Information

- Course title – Software development tools

- Lecturer – Zhuldyz Beishenalievna Kalpeyeva
- PhD, Association-professor CE&IS Dept., Room 804,
- E-mail: zh.kalpeyeva@iitu.kz,
- Office hours Wed, Fri 14.00-16.00

- Lectures: 15 hours (1 h/w)
- Laboratory classes: 30 hours (2 h/w)
- Each lab work must be submitted at the end of the lesson
- Each submission is evaluated and marked according to the 100-point system

COURSE OVERVIEW

TO STUDY LARGE SYSTEMS AND HOW THEY WERE PARTITIONED INTO SUBSYSTEMS AND COMPONENTS, AS WELL AS HOW THE STRUCTURING OF THESE ELEMENTS INTO A SOLUTION AND THE INTERFACES USED TO JOIN THEM TOGETHER FACILITATES COMMUNICATION AND CONTROL.

TO EXPLORE WITH VARIOUS NOTATIONS AND FORMALISMS AS THEY LEARN THE RELATIONSHIP BETWEEN THESE STRUCTURES AND KEY QUALITY ATTRIBUTES AND THEIR IMPACT ON SYSTEM IMPLEMENTATION.

DIFFERENCES BETWEEN DETAILED DESIGN AND ARCHITECTURE ARE EXPLORED, AS WELL AS NOTATIONS USED FOR BOTH.

SEVERAL WELL-KNOWN ARCHITECTURAL STYLES ARE ANALYZED .

THE USE OF VARIOUS NOTATIONS IS EXPLORED, WITH A FOCUS ON UML (UNIFIED MODELING LANGUAGE), AND THE ROLE OF ARCHITECTURE AND DETAILED DESIGN SPECIFICATIONS ARE CONSIDERED FROM THE PERSPECTIVE OF RISK MANAGEMENT.

Knowledge and Abilities Required Before the Students Enter the Course

- You are assumed to be proficient programming in Java.
- You should be able to declare and use classes, interfaces, polymorphism, inheritance, arrays, strings, loops and conditional statements, methods, create objects from classes, invoke methods on an object, perform basic text file input and output, and use basic data structures.
- You must have completed Course: Performance, Data Structures, and Algorithms.

HOW TO SUCCEED IN THIS COURSE

- **To Pass the Course**
- You must earn at least a "**C**" in this course. To do this, you must complete the following tasks at a bare minimum:
- **Complete the assigned readings.**
- **Attend and participate in class.**
- **Produce solutions to course projects.**
- **Pass the assessments (exams) include:**
- Laboratory works, weekly assessments on selected reading and lecture material
- Comprehensive assessments at the middle and end of the semester

LEARNING-BY-DOING (LBD), STORY-CENTERED CURRICULUM

- Learning-by-doing activities include individual and small group tasks.
- This environment, called the “back story,” uses an approach called story-centered learning
- you in the role of a **junior intern** at a firm, iCarnegie Consulting, where you will solve problems that resemble those encountered by real companies every day.

LEARNING OUTCOMES. knowledge

- The importance of taking a risk-based approach to software development.
- Why doing so aids an organization in determining how much architecture is enough for a given project.
- Students will be able to identify risks and discuss how to mitigate them.
- How software architecture can be used to ensure quality goals will be met.
- Be able to discuss the purpose for creating different views of software architecture and be able to contribute to discussions in determining which are appropriate for a given project.
- To discuss architectural choices, the short-term and long-term consequences associated with each choice and the rationale for selecting one choice over the others.
- Discuss architecture styles, as well as the advantages and disadvantages of their use.
- Describe how software architecture and design can be used during software maintenance.
- Students will also be able to identify system solution, from requirements to quality attributes and architectural structures to design patterns and detailed design, to implementation, testing, integration, sustainment and future reengineering as required to extend a system’s life.

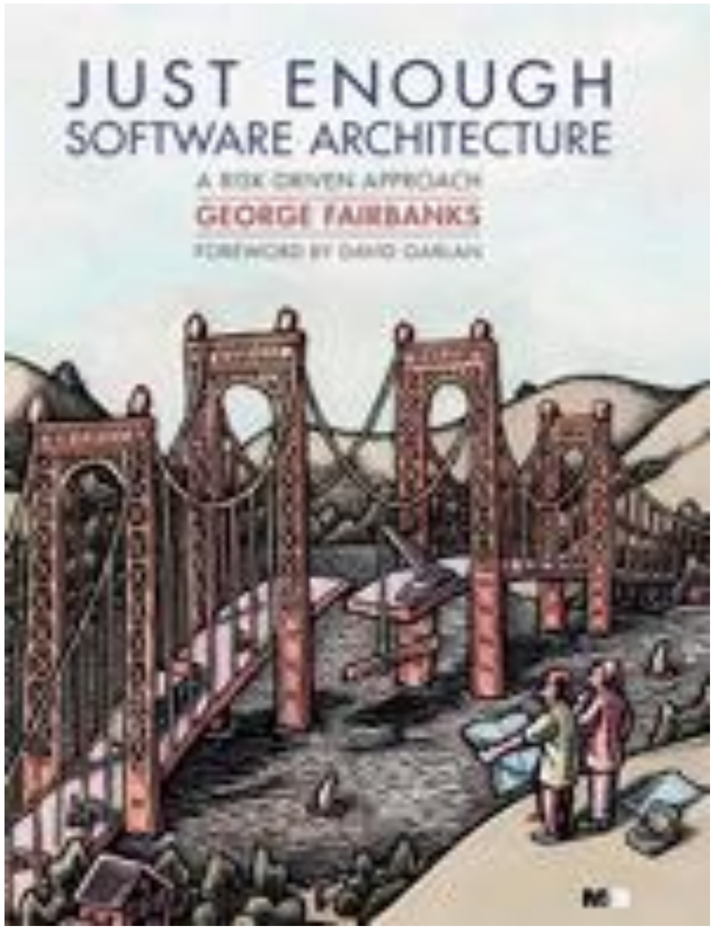
skills

- Use risk management processes, methods and techniques as the basis for deciding what to express and how best to express it, as well as what does not need documenting.
- Draft detailed design documents consistent with a specified architecture for moderate to small systems using UML design notations.
- To design module interfaces to support concurrent design and development by teams.
- Develop draft architecture documents for moderate to small systems employing one or more views.
- Use standard OO and other requirement notions to understand stakeholder requirements and express those.
- Demonstrate understanding design intent required to implement modules, subsystems and systems.
- Create views to capture and communicate key aspects of a design element for a specific and targeted audience
- Use architectural styles and design patterns.
- Use refactoring to improve code without changing its behavior.

MATERIALS

- **Online Resources & References**
- In addition to the textbook, several websites contain required readings. See the course syllabus for more detail. Additional online resources and references may be added as needed.
- **Textbook**
- *Just Enough Software Architecture:*

A Risk-Driven Approach by George Fairbanks, Marshall & Brainerd Publishers, 2010.



COURSE POLICIES AND EXPECTATIONS

Class Rules

- Respect the learning environment:
- Come to class on time and prepared to discuss the assigned reading.
- Do not distract the class with private conversations.
- *Turn off all mobile devices.*
- Use the lab computers in the room for exercises only.
- Bring paper and pens/pencils.

CUSTOMER PROJECTS

- All work in this class will be directed toward developing the architecture and design documents for development of a new software system for an auto supply company. Software requirements for the project will be provided during the first week of class. Your team will be in regular contact with your customer, and each week you will extend the architecture and/or revise previously created design diagrams and supporting documents as needed in response to customer discussions. Each team member will own responsibility for designing some parts of the system. Coordination between team members is achieved mostly through in-class meetings and in-class team exercises.

In the course of the semester, you will:

- Compile and prioritize a list of risks the project faces, along with mitigation strategies and status indicators
- Create one or more views of the architecture, including diagrams and supporting documentation that include at least one pre-existing modules
- Create encapsulations and interfaces for major components
- Create component and class diagrams
- Create a process for maintaining the architecture

Your Assignments

For iCarnegie Customers:

- Project 01. Risk Assessment
- Project 02. Initial Design
- Project 03. Creating Models
- Project 04. Patterns and Styles
- Project 05. Using the Architecture

Lecture 1. Introduction. Requirements Engineering. Basics of Use Cases

A general Introduction to SW Engineering

What is software (SW)?

- SW is
  - not only *programs*
  - but also all *associated documentation*, and
  - *configuration data*

  that make these programs operate correctly.
- More specifically, a SW system consists of
  - *separate programs*
  - *configuration files* setting up these programs
  - *system documentation* describing the structure of the system in good detail
  - *user documentation* explaining how to use and operate the system.

What is SW Engineering (SWE)?

- SW engineering is an *engineering discipline*
  - concerned with *all aspects of SW production* starting from the early stages of system specification through to the maintenance of the system after it has started to be used.
- *engineering discipline*:
  - implies solving a well-defined (or in SW engineering vaguely defined) problem optimally using resources (e.g., time, man power and machine power) and remaining within
    - organizational (i.e., the customer)
    - financial, and
    - other possible
  - constraints.
- *all aspects of SW production*:
  - encompasses
    - not only the technical processes
    - but also deals with project management, development of tools, methods and theories to support SW production.

Software engineering is important for two reasons:

- 1. More and more, individuals and society rely on advanced software systems. We need to be able to produce reliable and trustworthy systems economically and quickly.

- 2. It is usually cheaper, in the long run, to use software engineering methods and techniques for professional software systems rather than just write programs as a personal programming project. Failure to use software engineering method leads to higher costs for testing, quality assurance, and long-term maintenance

What is a SW process?

- *Four fundamental activities* and associated results which produce a SW product.
  - *SW (requirements) specification*
  - *SW development ...*
    - *Design phase*
    - *Implementation phase*
  - *SW verification and validation (SW V&V)*
  - *SW evolution*

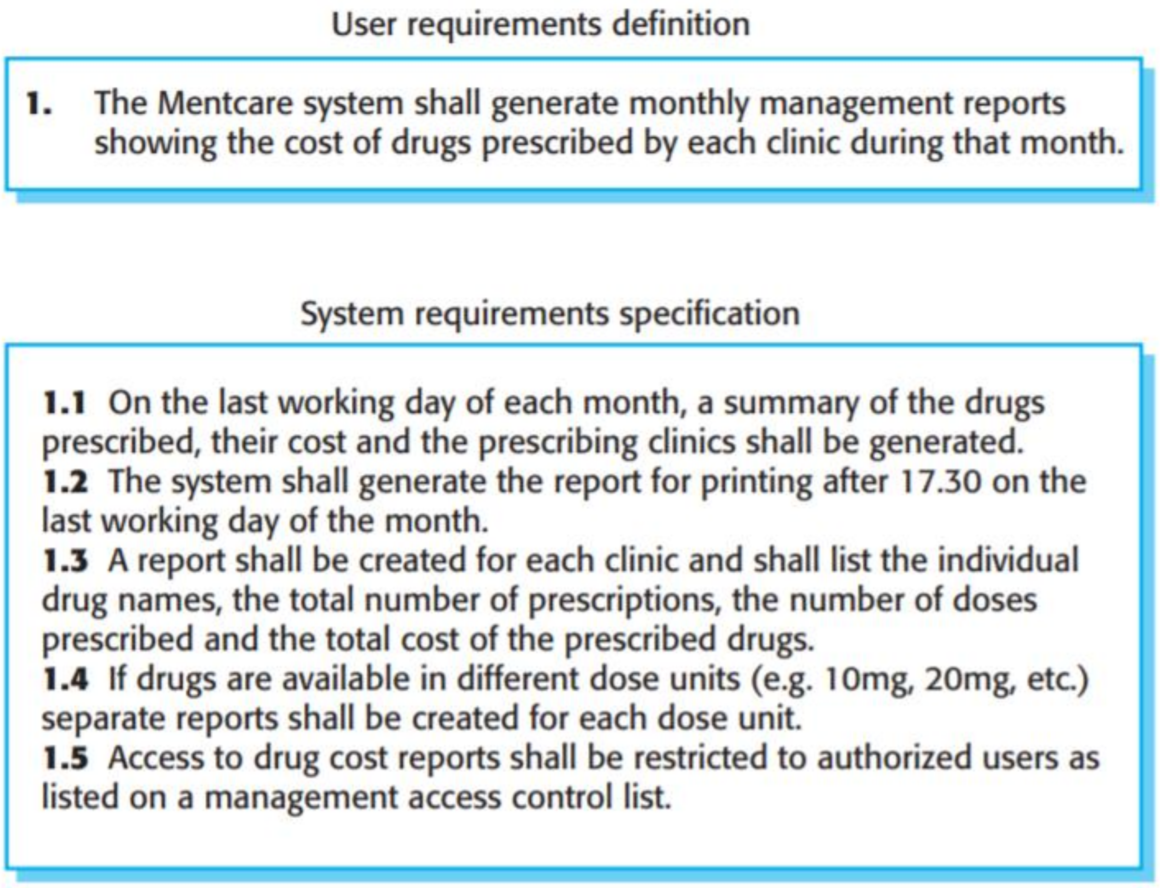
SW (requirements) specification. Requirements Engineering (RE)

- The **requirements** for a system are the descriptions of the services that a system should provide and the constraints on its operation.
- The process of finding out, analyzing, documenting and checking these services and constraints is called **requirements engineering (RE)**.
- Basically, it's the process of determining and establishing the **<u>precise</u>** expectations of the customer about the proposed software system.
- Aim to develop system to meet user needs
- Capture user requirements through:
  - Background reading/research
  - Interviews with users/clients
  - Observation of current practices
  - Sampling of documents
  - Questionnaires

User and system requirements

- User requirements to mean the high-level abstract requirements and system requirements to mean the detailed description of what the system should do.
- User requirements and system requirements may be defined as follows:
- 1. **User requirements** are statements, in a natural language plus diagrams, of what services the system is expected to provide to system users and the constraints under which it must operate.
- 2. **System requirements** are more detailed descriptions of the software system's functions, services, and operational constraints. The system requirements document (sometimes called a functional specification) should define exactly what is to be implemented. It may be part of the contract between the system buyer and the software developers.

User and system requirements



User requirements definition

1. The Mentcare system shall generate monthly management reports showing the cost of drugs prescribed by each clinic during that month.

System requirements specification

1.1 On the last working day of each month, a summary of the drugs prescribed, their cost and the prescribing clinics shall be generated.
1.2 The system shall generate the report for printing after 17.30 on the last working day of the month.
1.3 A report shall be created for each clinic and shall list the individual drug names, the total number of prescriptions, the number of doses prescribed and the total cost of the prescribed drugs.
1.4 If drugs are available in different dose units (e.g. 10mg, 20mg, etc.) separate reports shall be created for each dose unit.
1.5 Access to drug cost reports shall be restricted to authorized users as listed on a management access control list.

Functional and non-functional requirements

- **Functional:** The precise tasks or functions the system is to perform.
  - *e.g.,* details of a flight reservation system
- **Non-functional:** Usually, a constraint of some kind on the system or its construction
  - *e.g.,* expected performance and memory requirements, process model used, implementation language and platform, compatibility with other tools, deadlines, ...

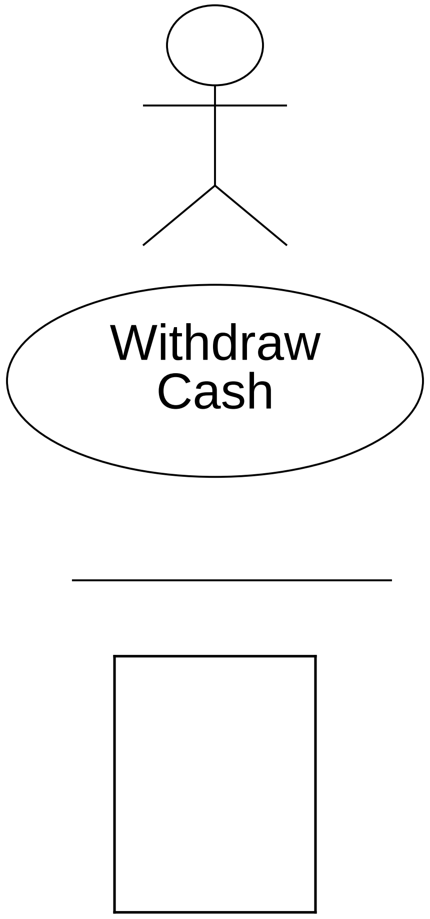
Basics of Use Cases

Use case diagrams are part of the Unified Modeling Language.

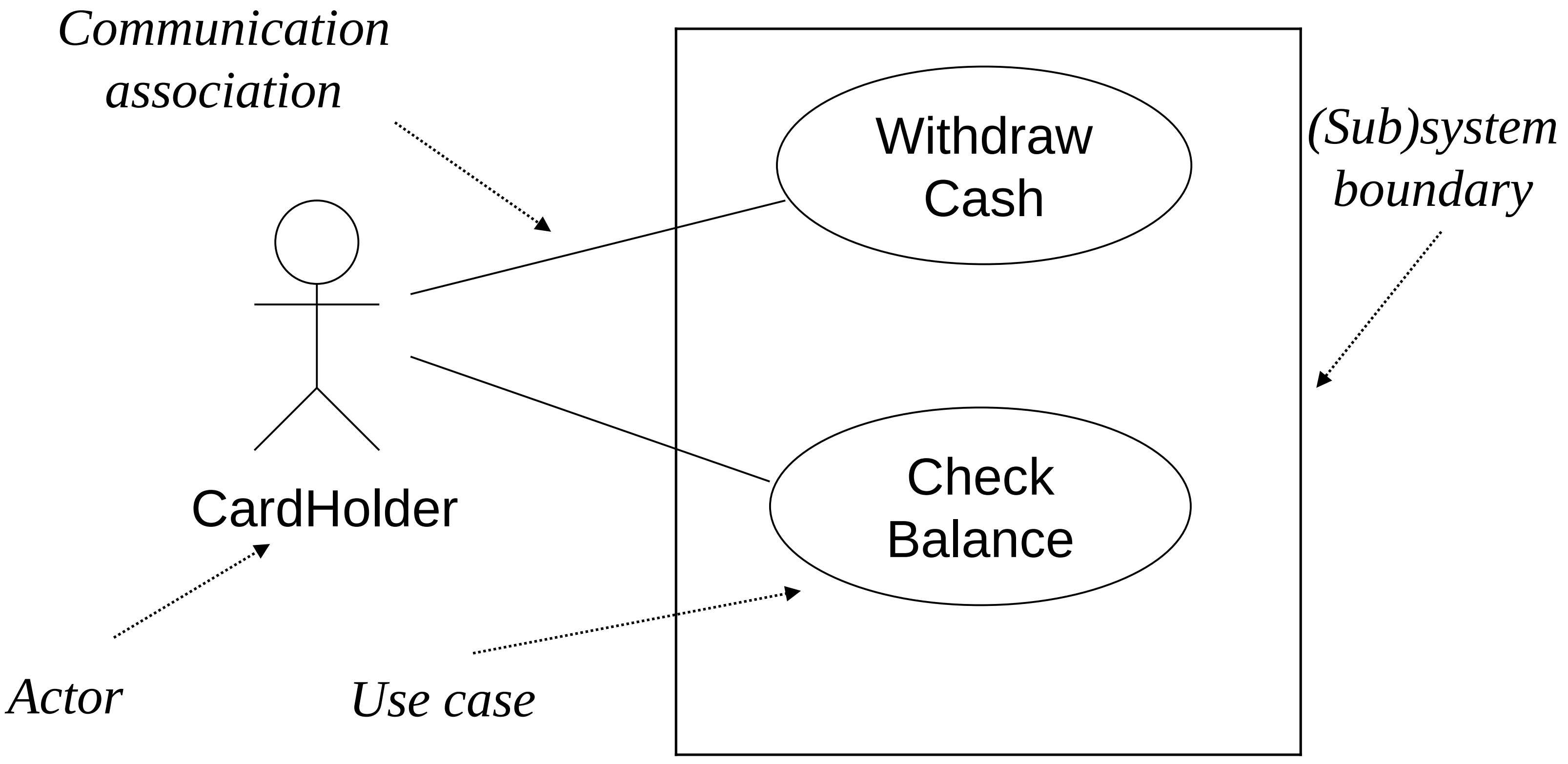
- Use cases concisely describe required functionality including quality related information.
  - What system does, not how
  - Focus on functionality from users' perspective
  - not appropriate for non-functional requirements
- Use cases help developers clarify requirements because they provide view of the system from perspective of users – not software developers – so provide a kind of bridge between the customer and the developers.
- Use Cases are great for stimulating discussions about what the customer really wants the system to do and often reveal new information about how the customer really wants it to behave.
- Use case diagrams are a pictorial representation of a collection of tasks the system should perform and who uses them.
- A "complete" use case diagram would include all tasks, but that is usually not possible and generally unnecessary.
- When identifying use cases, focus on tasks that seem important, will be used a lot, or appear to involve complex interactions.

Use case Notation

- Use case diagrams depict:
  - **Actors**: people or other systems interacting with system being modelled. Actors are drawn as stick figures.
  - **Use cases**: represent sequences of actions carried out by the system.
  - **Communication(Association):** between actors and use cases are indicated in use case diagrams by solid lines.
  - **System boundary boxes (optional)**. You can draw a rectangle around the use cases, called the system boundary box, to indicates the scope of your system.



UML Use Case Diagram



Simple Use Case Description

- ATM system example:

Withdraw Cash:

The card holder selects the withdraw cash menu, which is displayed by the system. The card holder selects an amount of cash. The system debits the user’s account, returns the user’s card and issues the requested money.

Elaborated Use-Case Description

	Actor	*System*
1	User selects cash withdrawal option	System displays cash withdrawal menu
2	User selects cash amount	System checks cash is available; returns card
3	User takes card	System dispenses cash
4	User takes cash	System returns to start menu

Use case Scenarios

- Use case represents a *generic* case of interaction
- A particular course that a use-case instance might take is called as *scenario*
- For example, there may be many Withdraw Cash scenarios where no cash is withdrawn!
    - Card holder has insufficient funds on account
    - ATM cannot connect to bank’s system
    - ATM does not contain enough cash
    - Customer cancels the transaction for some reason

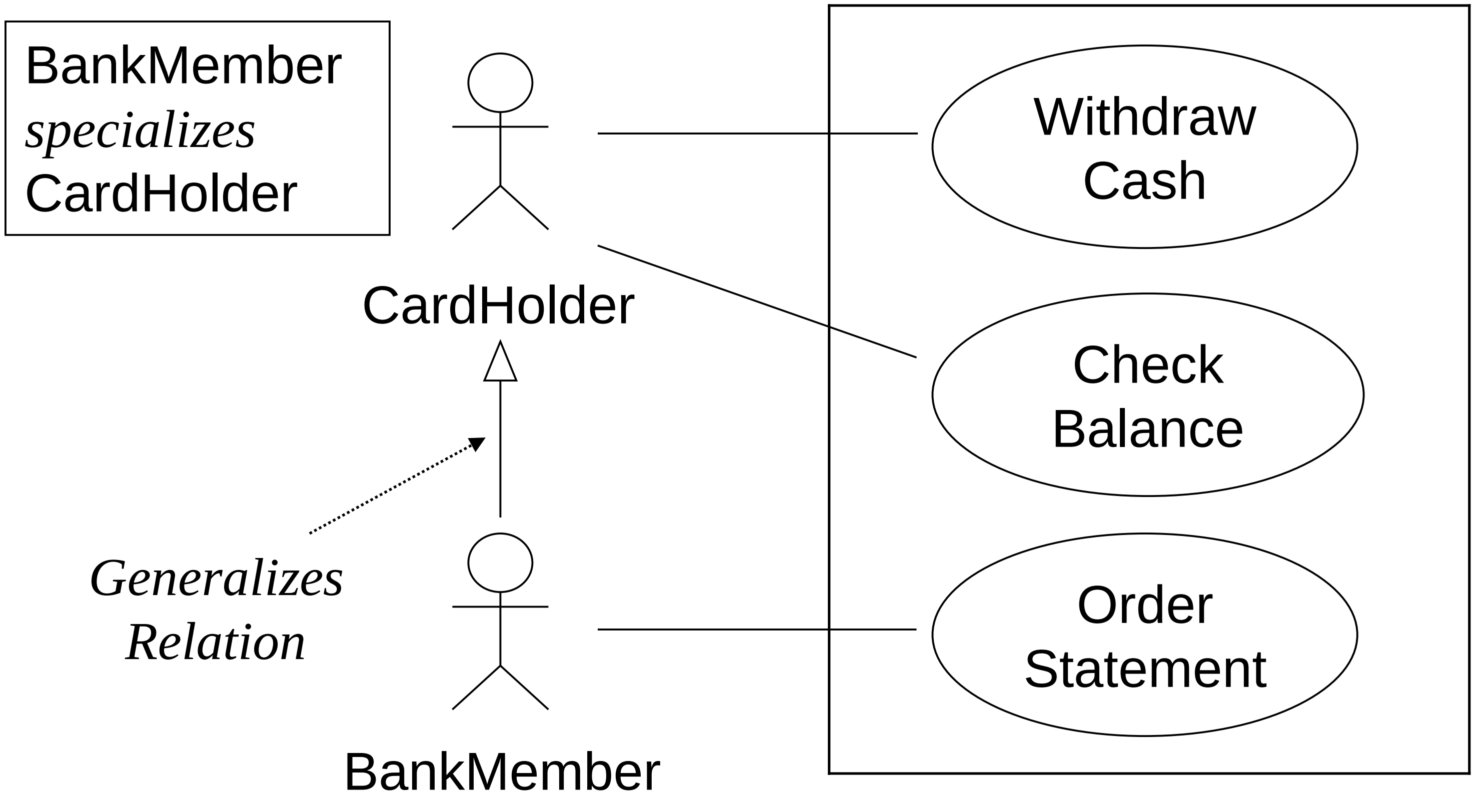
Example Scenarios

- CardHolder Mary Jones selects the cash withdrawal menu. She changes her mind, cancels the transaction and takes her card
- CardHolder Peter Smith selects the cash withdrawal menu. He selects a cash amount, but is refused because he has insufficient funds on his account. His card is returned

Types of Relationship on Use case Diagram

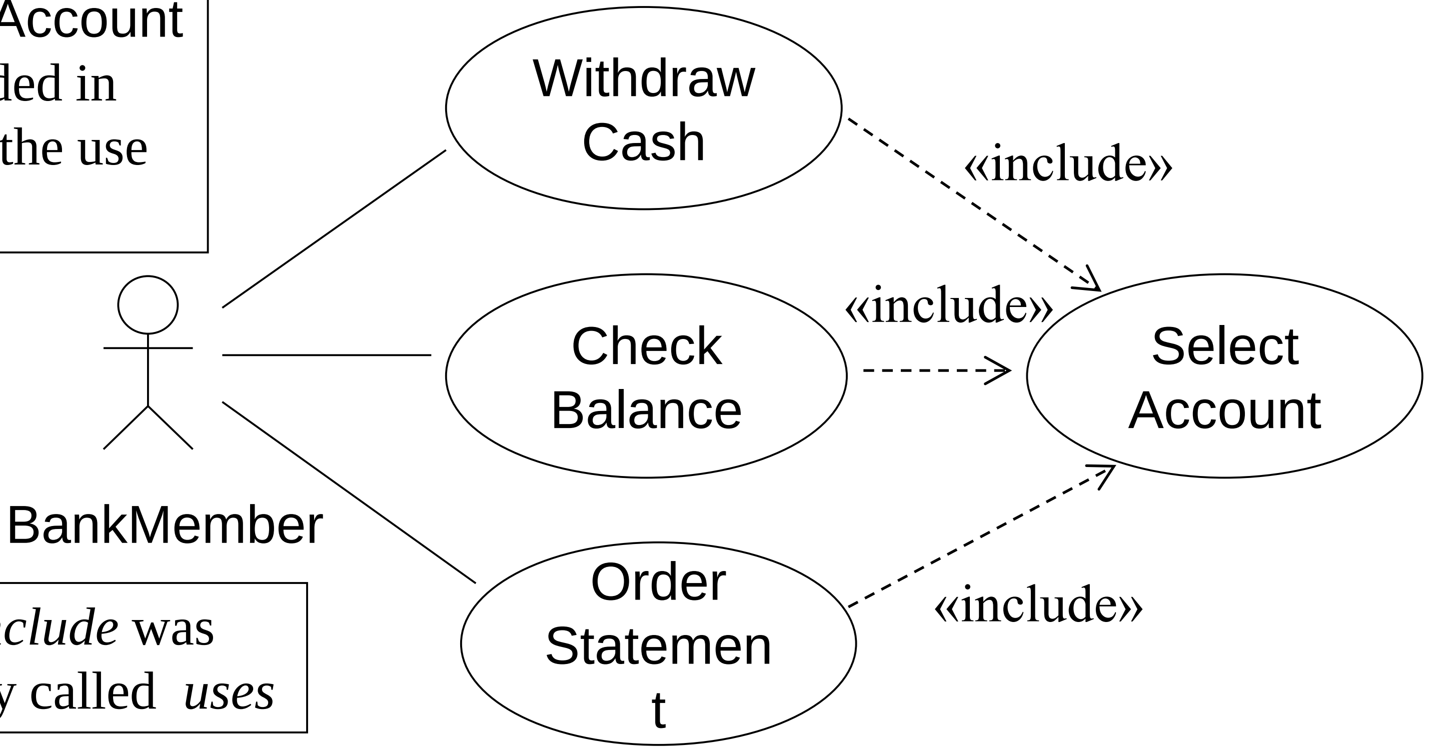
- Generalizes:
  - Permits actors/use cases to inherit properties of more general actors/use cases
- Include:
  - Permits use case to include functionality of another use case
- Extend:
  - Allows for optional extensions of use case functionality

Generalizes Relationship example



Include Relationship example

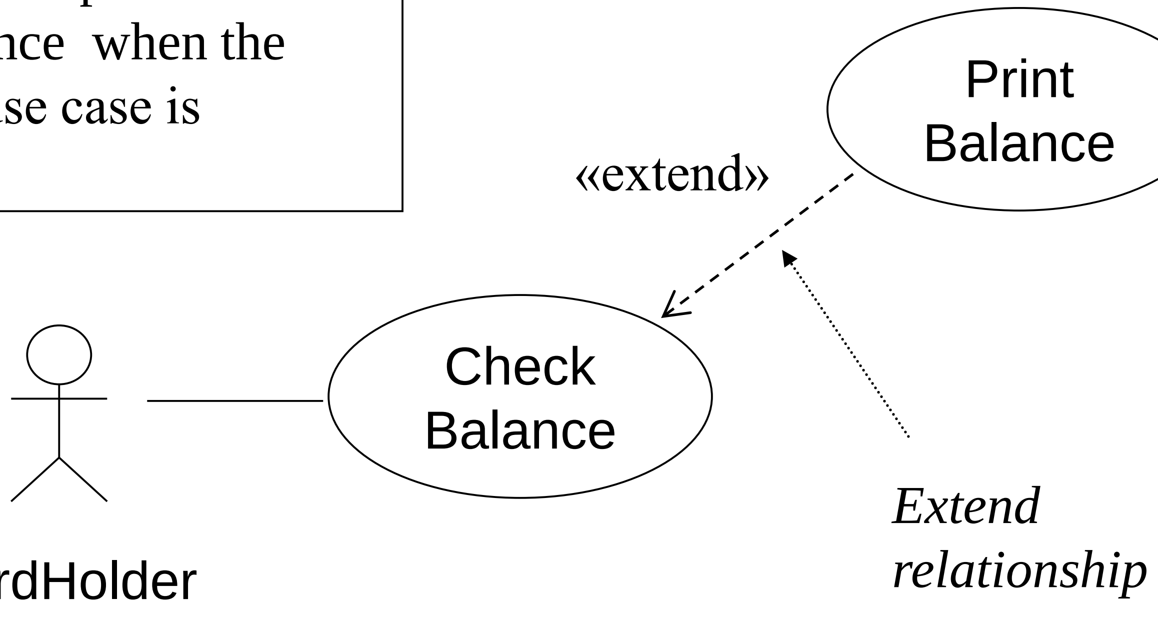
Select Account is included in each of the use cases



Note: *include* was formerly called *uses*

Extend Relationship example

CardHolder has the option of printing out a balance when the Check Balance use case is invoked.



Summary

- Use case analysis often a first step in system development:
  - provide high-level view of system functionality (what rather than how) and its users
  - Model *generic* activities
  - Particular instances of use-cases are termed *scenarios*
- UML use case diagrams
  - Contain *actors*, *use cases* and *associations*
  - supported by *behaviour specifications* (e.g. *use-case descriptions)*

Thank you for the attention!