

# Software engineering

Overview and Basic tasks

Dr Hamed Hemeda

# What is software?

- Software is the programs executed by computers. It may be system software such as operating systems or application software such as games, word-processing, database systems and electronic commerce systems
- Software types(according to Lehman)
  - Static/Specified(S-type): well-defined specification and obvious solution. Example: Calculator program
  - Procedural (P-Type): The controlling procedures are well-defined, but the solution is not so obvious. Example: chess playing
  - Evolving(E-Type): Specification and solution are strongly linked to environment. Example: E-Commerce system

# Software engineering projects

- What is software engineering and what is a software engineering project?
  - Software engineering is the developing of software product for specific purposes using scientific principles and methods within the constraints of allowed cost, time and other resources.
  - Software engineering project is activities and tasks done by a team to produce a software product using the principles and methods of software engineering
  - In any engineering project: There should be a plan: What to do? When and how to do it?

# Software engineering basic tasks

- What are the basic tasks done in a software project?
  - **Feasibility study:** Is it possible economic project?
  - **Requirement gathering:** Who are the customers? What they want? What they need?
  - **High-level design:** What are the platforms? On premise or on cloud? What are the modules of the system and how they interact without entering in the details of each module.
  - **Low-level design:** assign modules to team members. Design of each module. Module design refinement until it becomes clear how it is implemented in code.
  - **Development:** implementing the design into code by programming the design
  - **Testing:** test your own code, test different parts or code is working with other parts, test if everything is working after integration of all/new modules
  - **Deploying:** Setup hardware, install software, training
  - **Maintenance:** While the system is running, collect and solve problems and errors

# Software engineering basic tasks: other terms

- **Feasibility** study: Is it possible economic project?
- **Requirements engineering**: customers? What they want? What they need?
- **User interface design**: What will the user see? How will he interact?
- **Architectural design**: which modules and how they interact.
- **Detailed design**: planning how of each module work.
- **Programming**: Implement each module in code
- **System integration**: Collecting modules to build the whole product
- **Validation**: Are we building the right product?
- **Verification** (testing): Are we building this X right?
- **Production**: install, administer, and maintain

# Engineering software development process?

- Why?

- Because software systems is becoming large and complex.
- To get scalable system
- To get cost effective systems in the allowed time
- To deal with dynamic nature of requirements change and system update
- To obtain quality in software product

- How?

- Engineering principles and methods/techniques for doing each of the basic tasks
- Process model: An overall plan or strategy for starting, working on, finishing all the required basic tasks.

# Document management system

- While working as a team on the preceding tasks, we need and produce many documents. A document management system is needed to manage these documents. What are the feature needed by a document management system for software engineering process?
  - People can share documents so that they can all view and edit them.
  - Only one person can edit a document at a given time.
  - You can fetch the most recent version of a document.
  - You can fetch a previous version of a document by specifying either a date or version number.
  - You can search documents for tags, keywords, and anything else in the documents.
  - You can compare two versions of a document to see what changed, who changed it, and when the change occurred. (Ideally, you should also see notes indicating why a change was made, although, that's a less common feature.)
  - You should have the ability to access documents over the Internet or on mobile devices.

# Feasibility study

Whether the project is worthwhile?

Dr. Hamed Hemeda



# Stages of visibility study

- Identifying the problem, requirements, defining the objectives/goals, scope and expected outcomes.
- Search alternatives
- Assessing Technical Feasibility
- Assessing Economic Feasibility
- Assessing Operational Feasibility
- Assessing Legal Feasibility
- **Schedule Feasibility**

# Systems classification

- With respect to feasibility study, software systems can be categorized:
- Replacement or new?
  - A system that replaces an existing computer-based system
  - A brand-new system that replaces or enhances work that is not currently computer-assisted
- General/Special purposes
  - A general-purpose system, such as a word processor or a game. This is written and then sold in the marketplace
  - A tailor-made one-off system for a specific application.
- Develop or Buy
  - Writing the software or buying it off the shelf?

# Technical feasibility

- Is the proposal technically feasible? That is, will the technology work?
- Examples:
  - A system to predict lottery results cannot work.
  - A system to understand natural-language dialog is borderline
  - A system that manage a store is possible

# Cost-benefit analysis

- Comparing:
  - The cost of providing the system- Cost
  - The money saved or created by using the system – the benefit
- Is benefit passes the cost? For more than one way to do the work: which way you choose? The greater benefit/cost ratio
- Period: cost-benefit for which period?
  - 5 years to overcome and distribute the starting cost
- Which cost?
  - cost to buy equipment, principally the hardware
  - Cost to develop the software
  - Cost of training
  - Cost of lost work during switchover
  - cost to maintain the system
  - Cost to repair the equipment in the event of failure
  - Cost of lost work in the event of failure
  - Cost to upgrade, in the event of changed requirements.

# Example Feasibility study of the software for an ATM

- Required:
  - 200 ATM systems for a bank each is complete with its hardware and software
- Technical feasibility
  - It is possible to build ATMs using available hardware components. The software can be specially designed or bought
- Cost-benefit study
  - Hardware cost
    - System parts:
      - Hardware: For each ATM: the processor, the card reader, the display, the screen, the keypad, the printer, a cash dispenser and a modem
      - AT the bank server and other resources
    - Startup cost
      - ATM machine hardware: 10000
      - Server and other resources the bank to respond to ATM requests=2000/ATM-machine
      - Installation in secure way=8000/ATM-machine
    - Running cost:
      - telephone-bill, changing printer paper, r, stocking the ATM with cash.=1000/ATM-machine/year

# Example Feasibility study of the software for an ATM-Continued

- Cost-benefit study
  - Software cost: The ATM software must be robust and reliable because it is used by members of the public
    - System parts:
      - ATM machine software system
      - Bank server system
    - Startup-cost
      - An estimate: two person years for the software=450/Atm-machine
    - Running cost
      - Update and enhancements= 50/Atm-machine/year

# Example Feasibility study of the software for an ATM-Continued

- Cost-benefit study
  - Benefit
    - ATMs are convenient, available 24/7 hours in stations, stores and public buildings. So they attract customers to bank=1000/ATM-machine/year
    - Replacing employees and bank space=40000/ATM-machine/year
  - Cost-benefit summary
  - For five years period
    - Cost/ATM-machine/year= 21000(Hardware)+500(software)=21500
    - Benefit: 40000 ATM-machine/year
  - Result:

Should the ATMs be developed? YES

# Running example: Library management

## System: Description:

It is needed to automate a currently manual managed • library. The library receive visitors for searching for books, read books inside the library. A visitor can ask to be registered as library-member or client. A library member can borrow books to read them at home

## Today task (Report):

Do a feasibility study for the suggested system •



# GIT and GITHUB

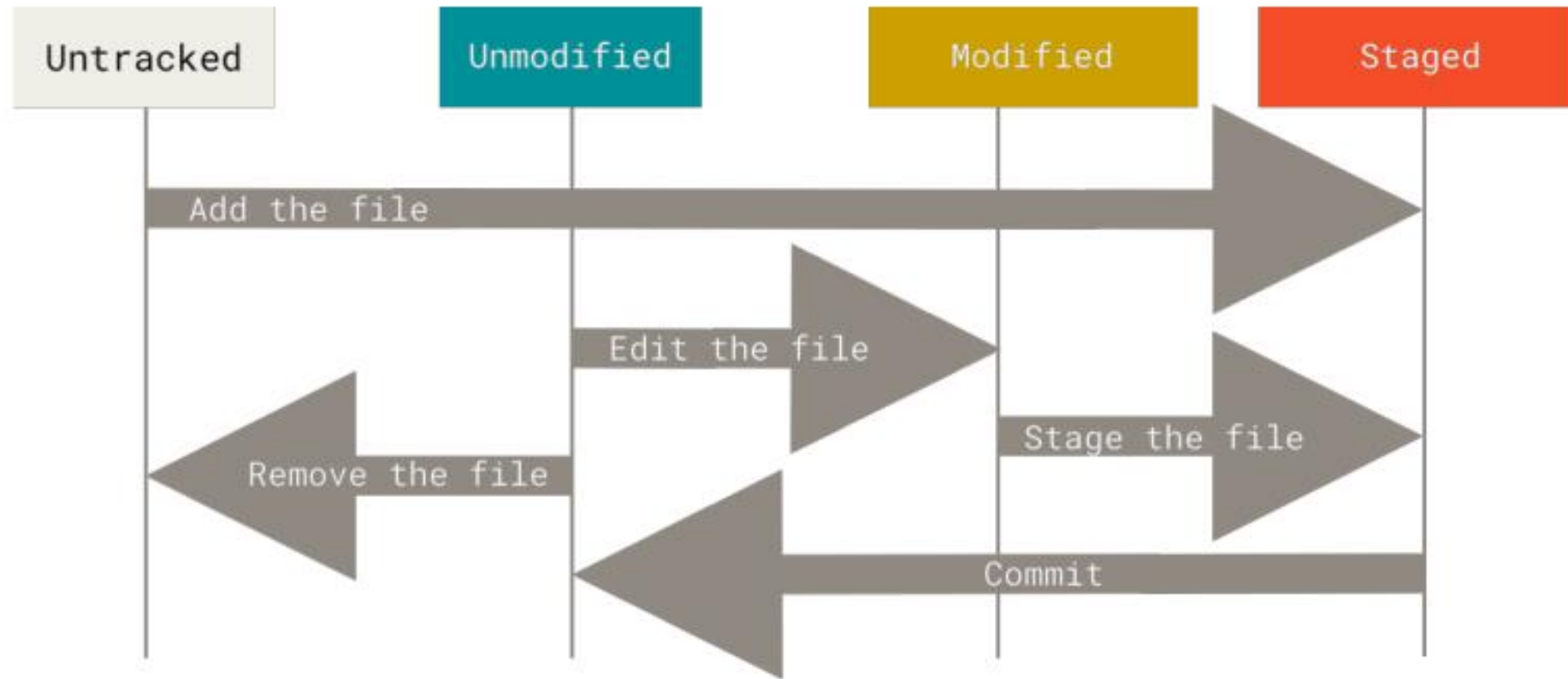
# What is Git?

- Git is a version control system that helps you keep track of changes to your code over time. With Git, you can:
  - Keep a history of all changes made to your code
  - Collaborate with other developers on the same codebase
  - Easily switch between different versions of your code
  - Roll back to previous versions of your code if something goes wrong
  - Branch your code to work on new features or experiments without affecting the main codebase

# Basic Git Workflow

- In Git, there are three main stages for managing changes to your code:
  - Working Directory - This is the directory on your local computer where you make changes to your code.
  - Staging Area - This is where you add changes that you want to commit to the Git repository.
  - Local Repository - This is where Git stores all of the committed changes to your code.

# Git version-controlled file status



# Setting Up Git

- Before we can use git, we need to install it on our system. To do this, we can visit the Git website and download the installer for our operating system. Once we have installed git, we can open up a terminal or command prompt and run the following command:
  - **git --version**
- This command will display the current version of git installed on our system. If we see a version number, we can proceed to using git. If not, we may need to troubleshoot the installation process.
- After installation configure username and mail:
  - **\$ git config --global user.name "John Doe"**
  - **\$ git config --global user.email johndoe@example.com**
  - **\$ git config --global core.editor emacs**
  - **\$ git config --global init.defaultBranch main**

# Initializing a Git Repository

- To start using git, we need to initialize a git repository in our project directory. We can do this by navigating to our project directory in the terminal and running the following command:
  - **git init**
- This will create a new git repository in our project directory. We can now start tracking changes to our code using git.

# Working with Git: Checking Status

- We can use the `git status` command to check the status of our git repository. This command will show us which files have been modified and which files are staged for commit.
  - **`git status`**

# Working with Git: Staging Changes

- Before we can commit changes to our git repository, we need to stage the changes. We can do this by using the git add command. Here are some examples of how to use this command:
  - **git add . // Add all changes**
  - **git add -A // Add all changes**
  - **git add <filename> // Add a specific file**
  - **git add \*.js // Add all files with a .js extension**



# Working with Git: Committing Changes

- Once we have staged our changes, we can commit them to our git repository using the `git commit` command. We need to provide a commit message that describes the changes we have made.
  - **`git commit -m "Add new feature"`**

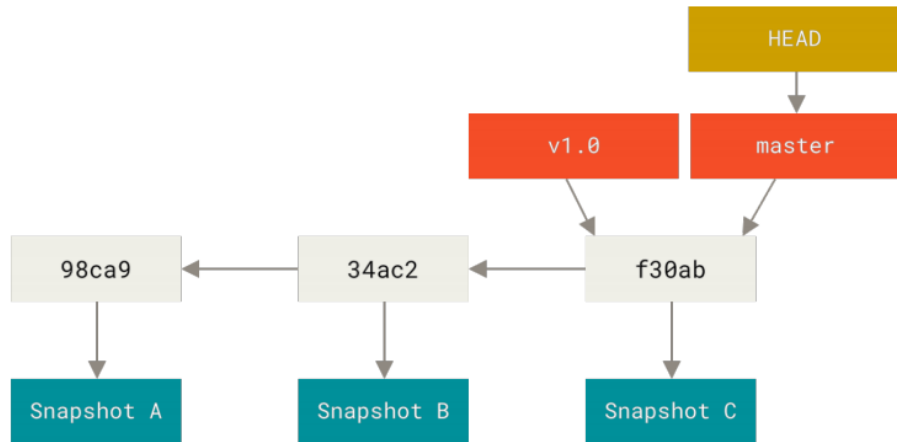
# Working with Git: Undoing Changes

- If we make a mistake and need to undo changes, we can use the `git reset` command. Here are some examples of how to use this command:
  - **`git reset //` Reset changes to the working directory**
  - **`git reset HEAD~ //` Reset changes to the staging area**
  - **`git reset --hard //` Reset changes to the last commit**
  - **`git rm <filename> //` Remove a file from the working directory and stage the removal**

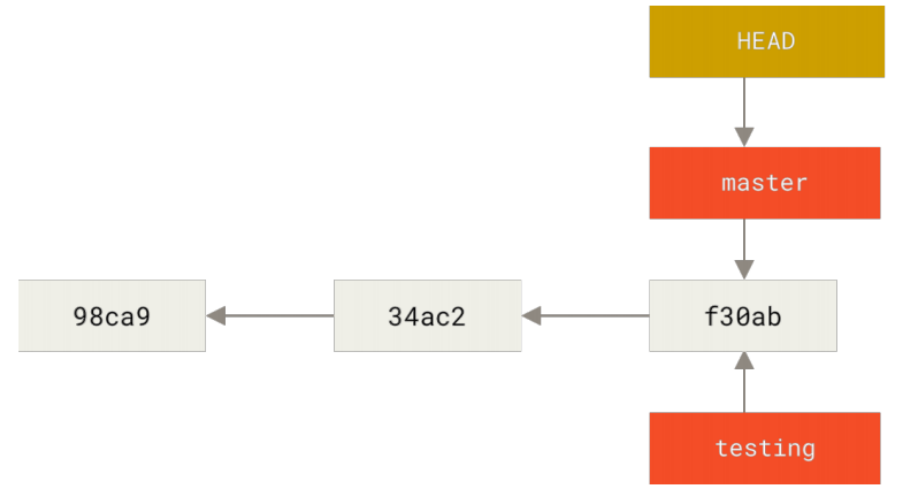
# Working with Git: Branching and Merging

- Branching and merging are powerful features of git that allow us to work on multiple features or versions of our code simultaneously. Here are some examples of how to use these features:
  - **git branch // List all branches**
  - **git branch <branch-name> // Create a new branch**
  - **git checkout <branch-name> // Switch to a different branch**
  - **git merge <branch-name> // Merge changes from another branch**

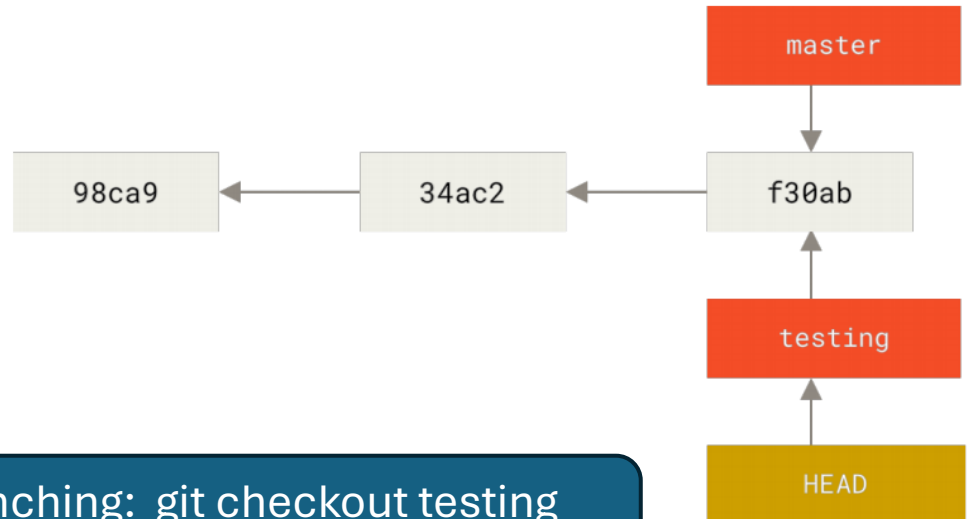
# A branch and its commit history



Before branching

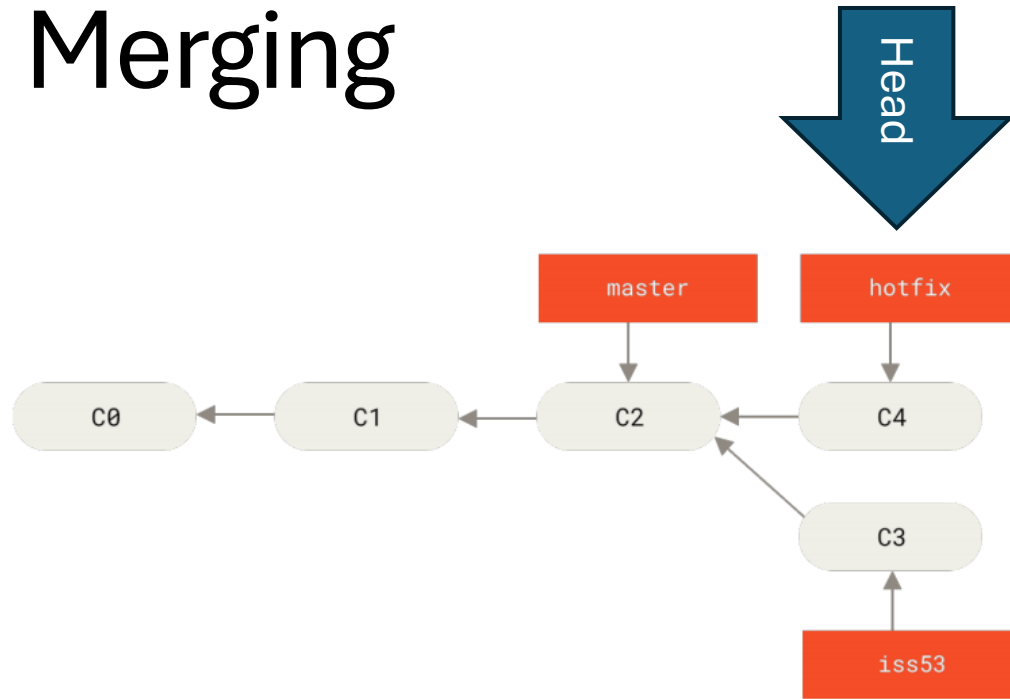


After branching: git branch testing

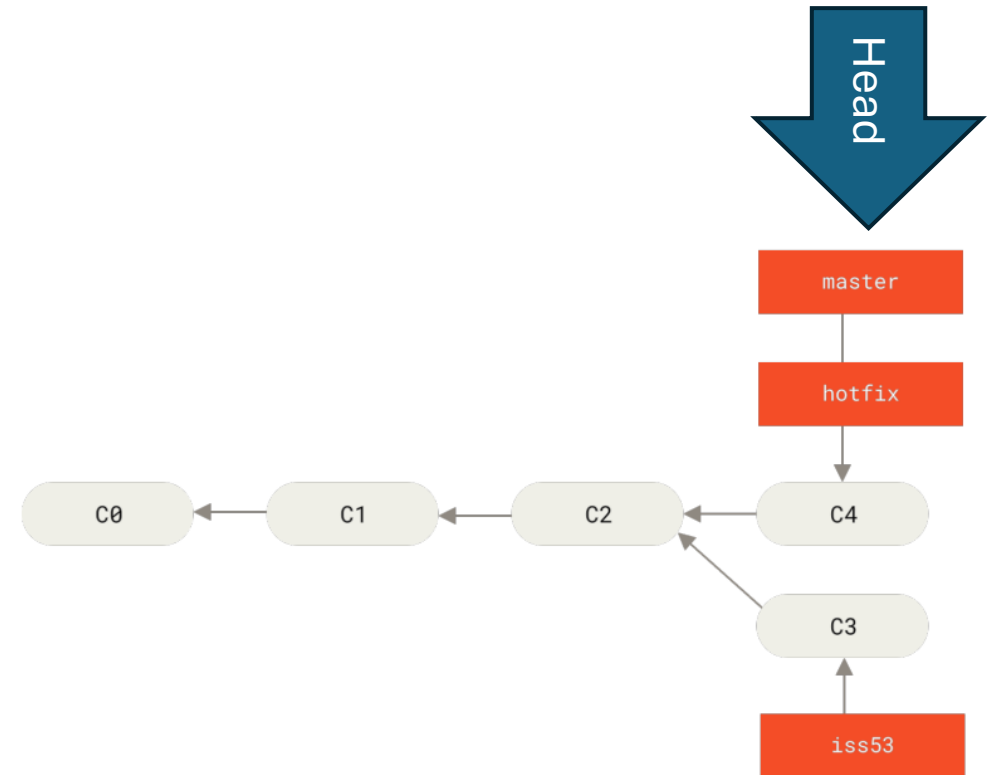


After branching: git checkout testing

# Merging

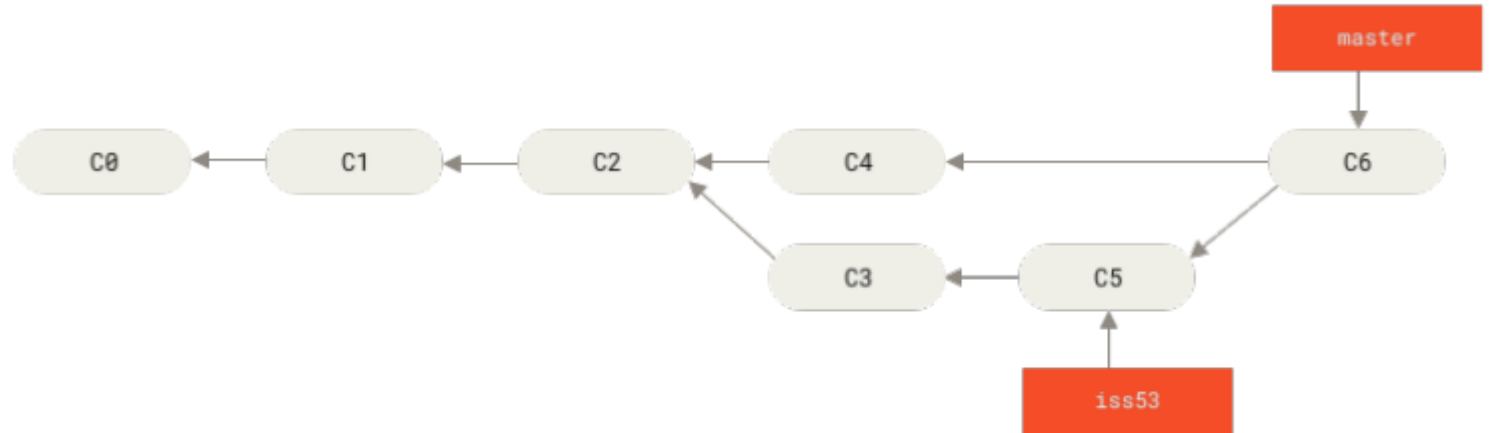
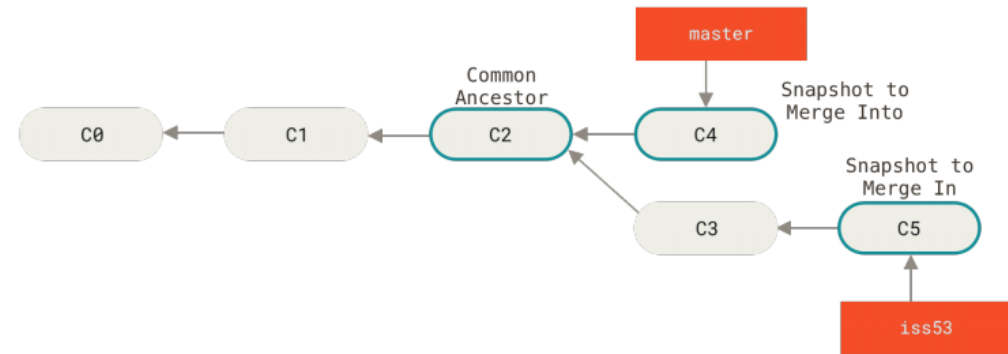


Before merging



After merging:  
\$ git checkout master  
\$ git merge hotfix

- Merging may need to issue commits
- Merging do not delete any branches; you should explicitly delete branches if you no longer need them



# Working with Git: Pushing changes to a remote repository and pulling them

- To push changes to a remote repository, run:
  - **git push**
- This will push your changes to the default branch of the remote repository.
- To pull changes from a remote repository, run:
  - **git pull**
- This will download any changes from the remote repository and merge them with your local copy.

# Working with Git: Cloning an existing repository

- To clone an existing repository, navigate to the directory where you want to store the repository, and run the following command:
- **git clone <repository-url>**
- This will clone the repository and download all of its files to your local machine.



# Working with Git: Pushing and Pulling from a Remote Repository

- In order to collaborate with others using git, we need to push and pull changes from a remote repository. Here are some examples of how to use these commands:
  - **git clone <repository-url> // Clone a remote repository**
  - **git push <remote-name> <branch-name> // Push changes to a remote repository**
  - **git fetch // Fetch changes from a remote repository**
  - **git merge // Merge changes from a remote repository**
  - **git pull // Fetch and merge changes from a remote repository**

# Working with GitHub

- GitHub is a web-based platform for hosting and collaborating on Git repositories. GitHub makes it easy to share your code with other developers, contribute to other projects, and keep track of issues and bugs.
- To get started with GitHub, you'll need to create an account on the GitHub website: <https://github.com/join>
- Once you have an account, you can create a new repository by clicking the "New" button on the main page of your GitHub account. You can give your repository a name and description, and choose whether to make it public or private.
- To connect your local Git repository to your remote GitHub repository, you can use the git remote command:
  - **git remote add origin <https://github.com/username/repository-name.git>**
- This will add a remote repository called origin with the URL of your GitHub repository

# Software engineering process models

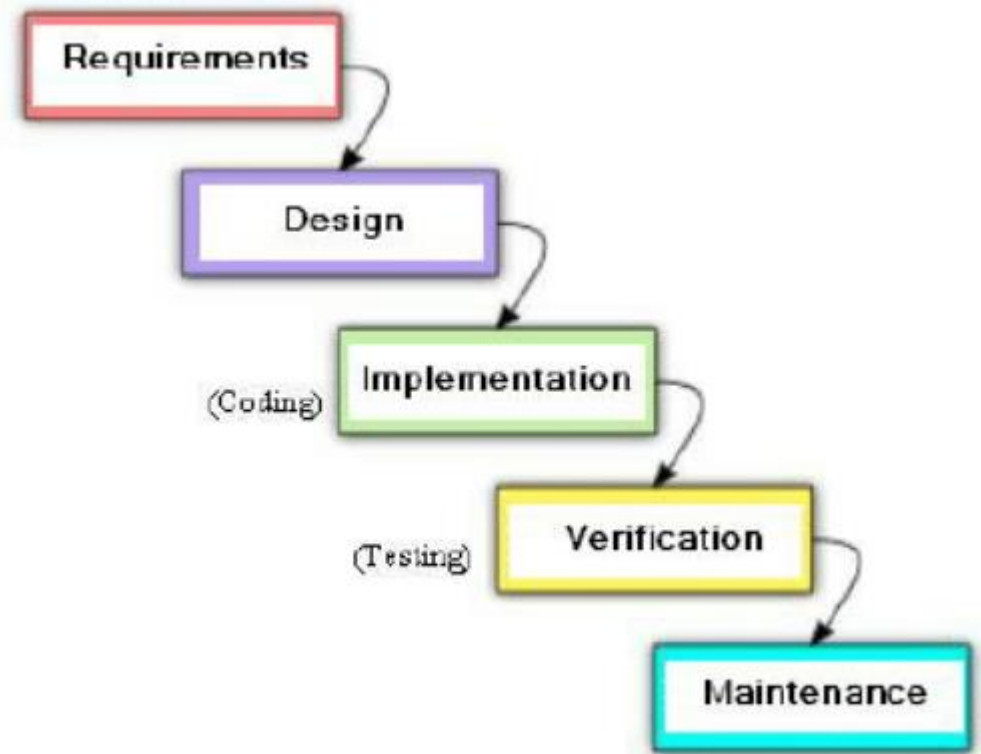
Dr Hamed Hemed

# What is software engineering process

- Software engineering process (Also called life-cycle) definition:
  - A software process is a series of phases of activities performed to construct a software system. Each phase produces some artifacts that are the input to other phases. Each phase has a set of entrance criteria and a set of exit criteria.
- Several models exist for software engineering process. A model is a specific arrangement of phases and activities in the process. A model may be suitable for a given project than another depending on the nature of the project. Existing models are:
  - Water-fall process model
  - Throwaway prototyping process model
  - Evolutionary prototyping process model
  - Spiral process model
  - Rational Unified Process (RUP) model
  - Agile processes model

# Water-fall process model

Is a straight sequence of development activities that begin with requirements analysis, followed by software design, implementation, testing, deployment, and maintenance. It is called a waterfall process because its straight sequence of activities resembles a waterfall when the activities are shown vertically one after another. Suitable for well-specified systems



Most software projects are wicked problem rather tame being=>water-fall may not be suitable

Properties of a Tame Problem	Properties of a Wicked Problem
1) A tame problem can be completely specified.	1) A wicked problem does not have a definite formulation.
2) For a tame problem, the specification and the solution can be separated.	2) For a wicked problem, the specification is the solution, and vice versa.
3) For tame problems, there are stopping rules.	3) There is no stopping rule for a wicked problem—you can always do it better.
4) A solution to a tame problem can be evaluated in terms of correct or wrong.	4) Solutions to wicked problems can only be evaluated in terms of good or bad, and the judgment is subjective.
5) Each step of the problem-solving process has a finite number of possible moves.	5) Each step of the problem-solving process has an infinite number of choices—everything goes as a matter of principle.
6) There is a definite chain of cause-effect reasoning.	6) Cause-effect reasoning is premise-based, leading to varying actions, but it is hard to tell which one is the best.
7) The solution can be tested immediately; once tested, it remains correct forever.	7) The solution cannot be tested immediately and is subject to life-long testing.
8) The solution can be adapted for solving similar problems.	8) Every wicked problem is unique.
9) The solution process is a scientific process.	9) The solution process is a political process.
10) If the problem is not solved, simply try again.	10) The problem solver has no right to be wrong because the consequence is disastrous.

# Throwaway prototyping process model

In some projects, it is probable that there will be a mismatch between the software and users' expectation. As a solution, a prototype of the system is constructed and used to acquire and validate requirements. Prototypes are also used in feasibility studies as well as design validation. A simple prototype shows only the look and feel and a series of screen shots to illustrate how the system would interact with a user. A sophisticated prototype may implement many of the system functions.

A throwaway prototype is constructed quickly and economically—just enough to serve its purpose. A throwaway prototype could be used as a reference implementation to check whether the implementation produces the correct result. Furthermore, it could be used to train users before the system is released.

# Evolutionary prototyping process model

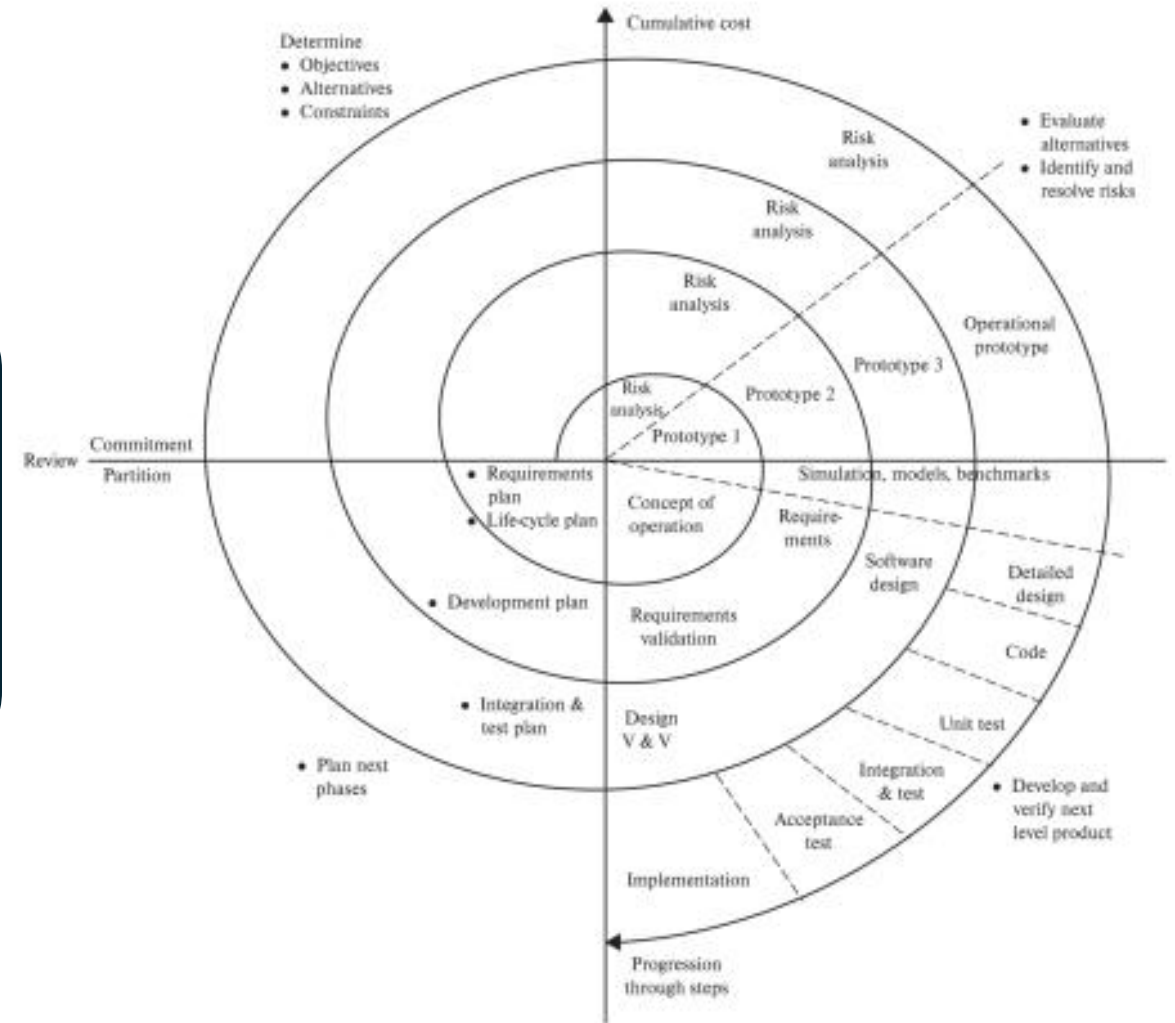
In some projects, it is probable that there will be a mismatch between the software and users' expectation. As a solution, a prototype of the system is constructed and used to acquire and validate requirements. Prototypes are also used in feasibility studies as well as design validation. A simple prototype shows only the look and feel and a series of screen shots to illustrate how the system would interact with a user. A sophisticated prototype may implement many of the system functions.

Throwaway prototypes imply that much effort is wasted. This is true when sophisticated prototypes are needed for feasibility study and design validation of large, real-time embedded systems. The evolutionary process model is aimed at saving the effort by letting the prototype evolve. It lets the users experiment with an initial prototype, constructed according to a set of preliminary requirements. Feedback from the users is used to evolve the prototype. This is repeated until no more extensions are required.



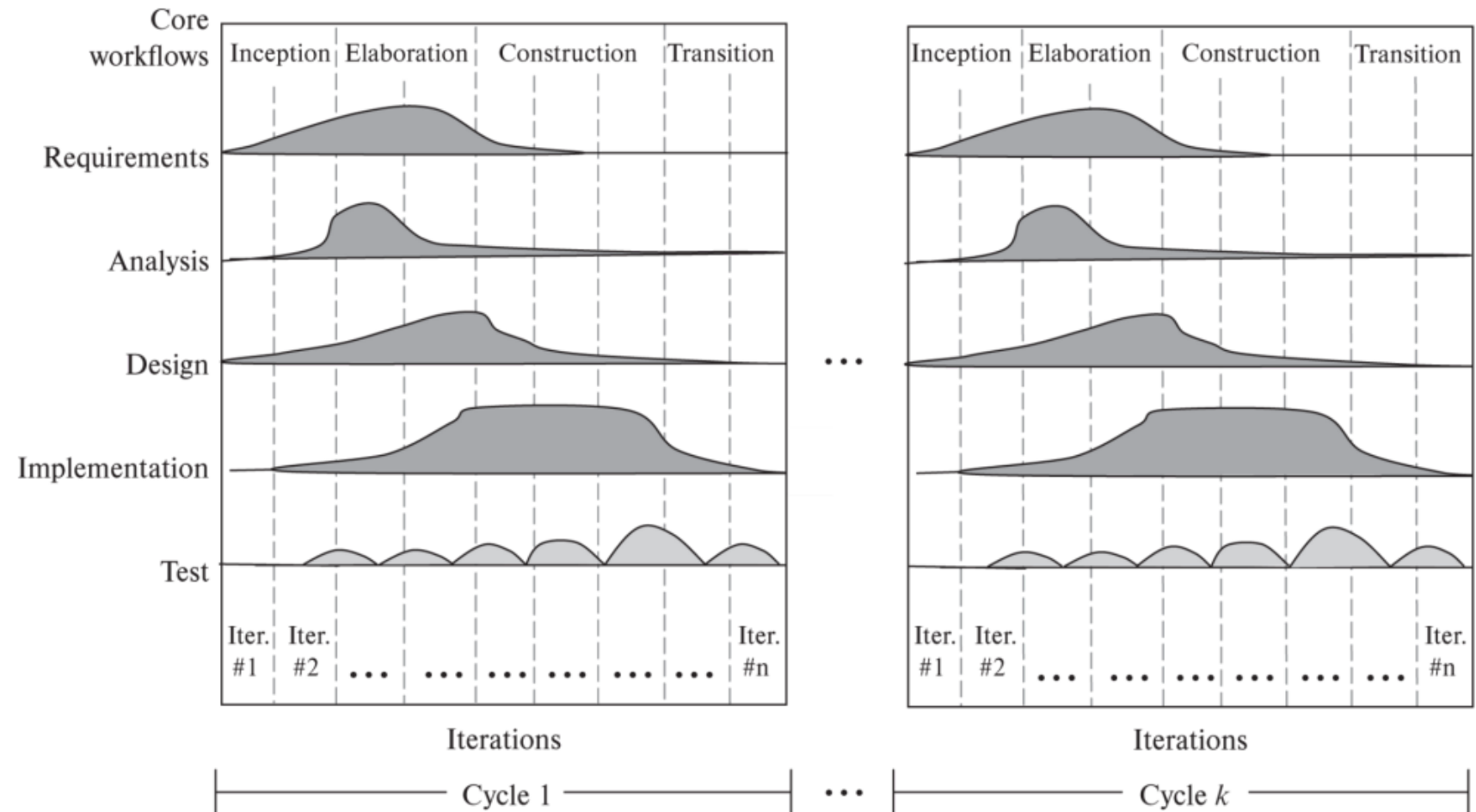
# Spiral process model

It combines iterative models and whater-fall models by dividing the process to cycles each can attach a specific aspect/objectives. Objective and constraints are identified, evaluated. Risk analysis is done then development take place either through prototyping or water-fall which is suitable for the cycle.



# Rational Unified Process (RUP) model

- The Rational Unified Process (RUP) or Unified Process (UP) consists of a series of cycles. Each cycle concludes with a release of the system.
- Each cycle has several iterations.
- The iterations are grouped into four phases: inception, elaboration, construction, and transition. Each phase ends with a major milestone, at which the manager makes an important project decision as to continue or terminate the project.
- Each iteration goes through a series of workflow activities, including requirements analysis, design, implementation, and testing.
- The gray areas are rough indications of the extent of the workflow activities that are carried out during the phases.



# Agile processes model

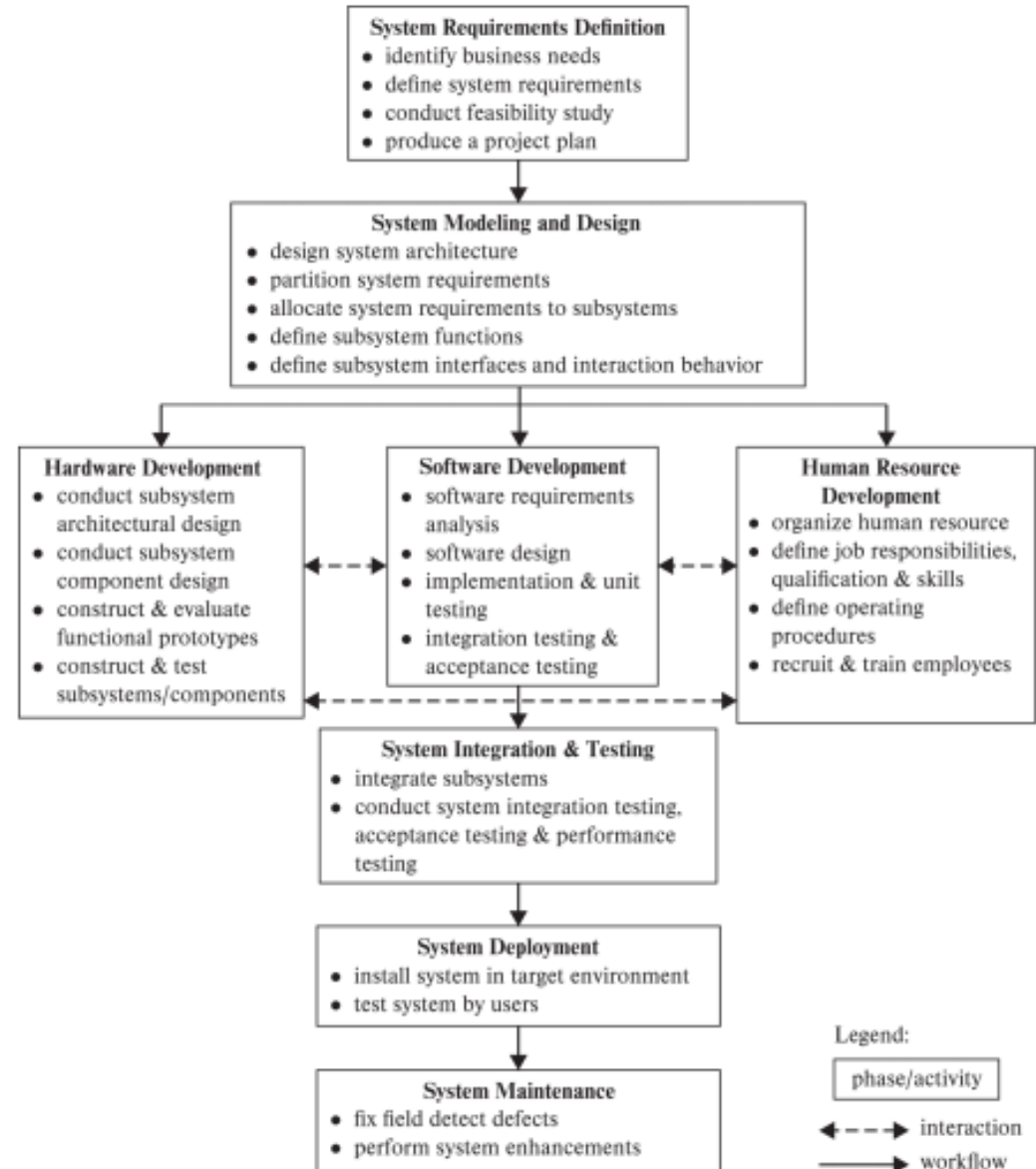
Agile processes emphasize teamwork, joint application development with the users, design for change, and rapid development and frequent delivery of small increments in short iterations. Agile development is guided by agile values, principles, and best practices. All these consider wicked-problem properties.

- Agile development values individuals and interactions over processes and tools.
- Agile development values working software over comprehensive documentation.
- Agile development values customer collaboration over contract negotiation.
- Agile development values responding to change over following a plan.

# Agile processes model principals

- Active user involvement is imperative
- The team must be empowered to make decisions.
- Requirements evolve, but the timescale is fixed. New and changed requirements can replaces low priority requirements.
- Capture requirements at a high level; lightweight and visual: story cards, screenshots,..
- Develop small, incremental releases and iterate.
- Focus on frequent delivery of software products.
- Complete each feature before moving on to the next.
- Apply the 80-20 rule: 80% of the work or result is produced by 20% of the system functionality. Those 20% are of high priority
- Testing is integrated throughout the project life cycle; test early and often.
- A collaborative and cooperative approach between all stakeholders is essential

- System engineering is an engineering discipline that deals with multidisciplinary systems (systems containing software, hardware and human resources).
- Such system development is an interdisciplinary effort. It involves hardware, software, and human resource developments.
- The figure shows a process model for such systems



# Requirement engineering

For a software development project

Dr. Hamed Hemeda

# What is requirement engineering

## Definition:

- What precisely the system user wants?
- What they need? Are there needs match their wants?
- How we can represent user needs using practical scientific engineering models, methods, tools and documentations on which there both developers and uses agree

## Who do Requirement engineering

- Software engineers during the phase of requirement engineering work as system analysts
- Special member of the developments team for system analysis

# Example: Top-view

We wish to develop a computer system for a library that currently does not use computers.

- Investigate and document all the current manual procedures for buying books, cataloging, shelving, loaning, etc.
- Decide which areas of activity are to be computerized (for example, the loans system)
- Thoroughly investigate the loans system, its documentation, interview current responsible in the manual system.
- Represent and document what you get is a scientific engineering way.



# The qualities of a specification

Implementation free – what is needed, not how this is achieved

Complete – there is nothing missing

Consistent – no individual requirement contradicts any other

Unambiguous – each requirement has a single interpretation

Concise – each requirement is stated once only, without duplication

Minimal – there are no unnecessary ingredients

Understandable – by both the clients and the developers

Achievable – the requirement is technically feasible

Testable – it can be demonstrated that the requirements have been met

# Application of the requirements qualities

Requirements: Write a Java program to provide a personal telephone directory. It should implement functions to look up a number and to enter a new telephone number. The program should provide a friendly user interface.

- Negative: “program is to be written in Java” its implementation (How) not needs (What)
- Negative: “A user-friendly interface” is hopelessly vague. Some determination should be added like a normal user should take no more than 5 seconds to reach the desired number
- Negative: Incomplete. No thing said about cost and time

# What is the role of the analyst?

## Role:

- Elicit and clarify the requirements from the users
- Help resolve possible differences of view amongst the users and clients.
- Advise users on what is technically possible and impossible.
- Document the requirements (see the next section).
- Negotiate and to gain the agreement of the users and clients to a (final) requirements specification

## Activities:

- listening (or requirements elicitation)
- thinking (or requirements analysis)
- writing (or requirements definition).

# Requirement specification

The product of requirements elicitation and analysis

Precisely state what the system should do

Is the reference document against which all subsequent development is assessed

Important factors:

- To whom the document is addressed (Users and developers)
- The level of detail (users suitable may not developers suitable)
- The notation used. (natural-language, formal using specific language and formulas, semi-formal)

If necessary, we can make two specifications:

- Requirement for agreement between developers and users
- Technical requirements using formal notation and models for developers

# Check list for the contents of specification

- The contents:
  - The functional requirements
    - Examples (Note verbs):
      - The system will display the titles of all the books written by the specified author.
      - The system will continuously display the temperatures of all the machines
  - The data requirements
    - Input/Output, storing and backup, Data interchange between systems
  - Performance requirements
    - Cost, delivery time, response-time, data-size, reliability, security
  - Constraints
    - System to run on existing hardware, OS, specific technology
  - Guidelines.

# Use cases

## What is use cases?

- Widely used approach to documenting requirements
- Textual description for a user (actor) doing a task
- Can be augmented by a UML use case diagram
- A use case both specifies what the user does and what the system does, but says nothing about how the system performs its tasks

## Example ATM

- **withdraw cash.** The user offers up their card. The system prompts for the PIN. The user enters the PIN. The system checks the PIN. If the card and PIN are valid, the system prompts the user for their choice of function. The user selects dispense cash. The user prompts for the amount. The user enters the amount. The system ejects the card. When the user has withdrawn the card, the system dispenses the cash.

# Identifying use-cases

A use case should be a useful task the user can do some work with.

Examples=> In the ATM, for example :

- Is entering and validating the PIN a use case?
  - The answer is no because it is not a useful function from the user's point of view, whereas
- Is withdrawing cash is a use case
  - The answer is yes because it is a useful function from the user's point of view

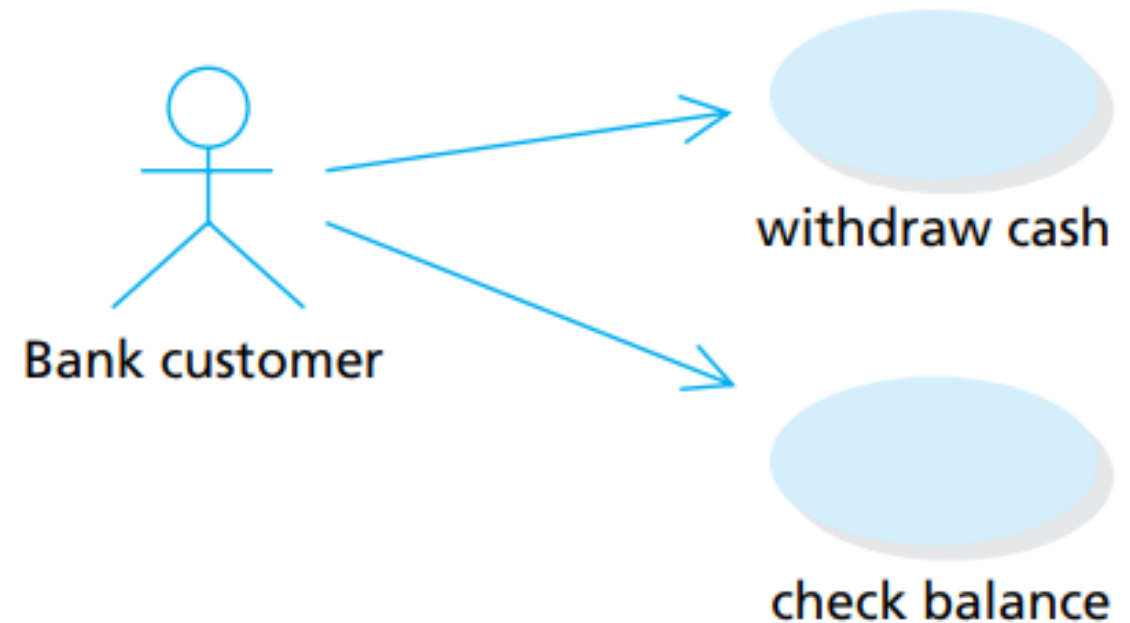
One way to identify distinct use cases is to identify a goal that an actor wishes to accomplish

In large systems, in order to control complexity, use cases are grouped into use case packages. Each package contains a set of related use cases

# Use cases diagram

Use cases can be represent using UML (Unified modeling language) diagrams

A use case diagram does not contain the detail associated with a (textual) use case. However, it does give an overall picture of the actors and the use cases. It is informal diagram





# Information gathering techniques

- Customer presentation
- Literature survey (study similar or related projects)
- Study of existing business procedures and forms
- Stakeholder survey
- User interviewing and questionnaires
- Writing user stories

Modeling is an effective tool to acquire domain knowledge to build requirement specification

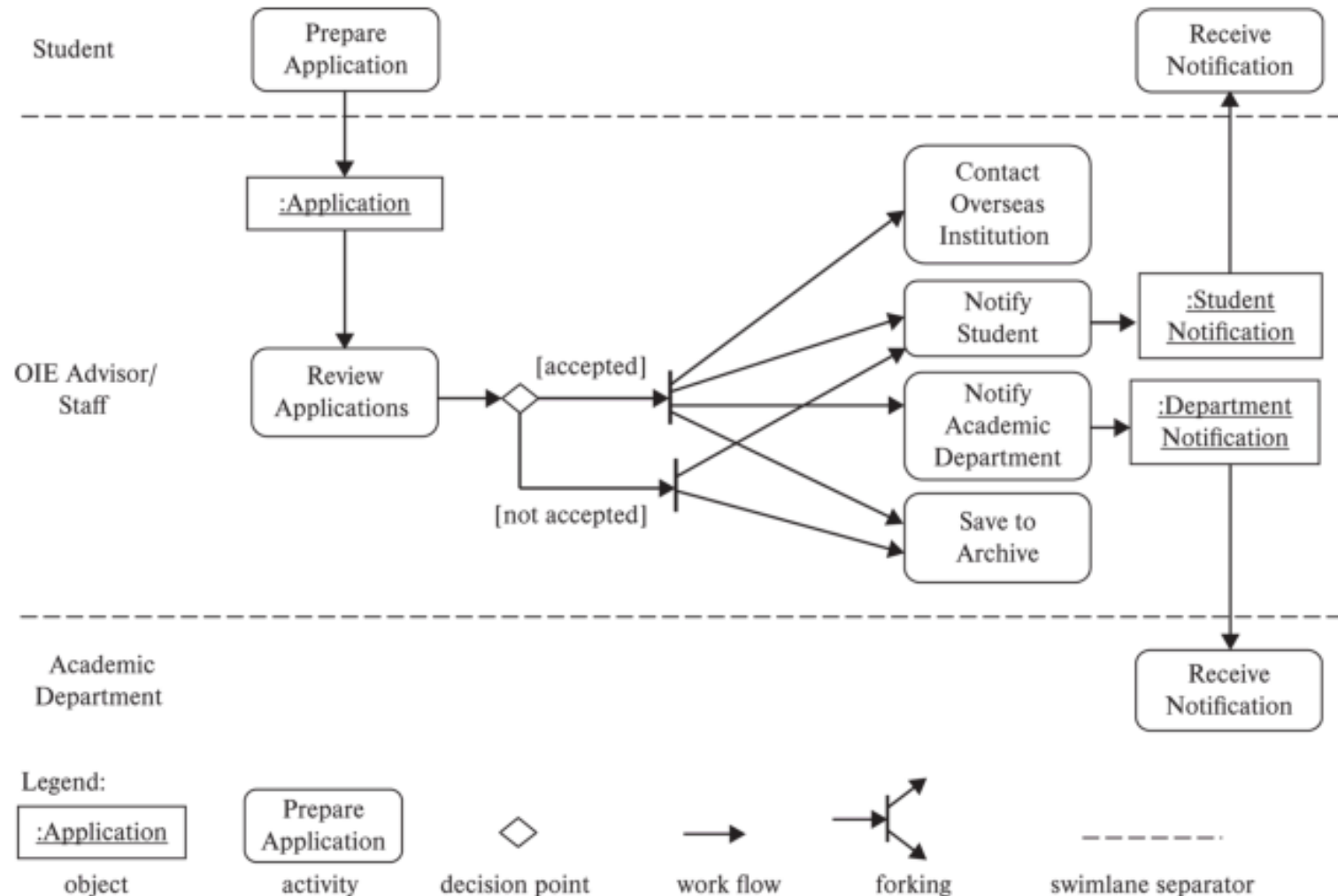
- Modeling is a conceptualization and visualization process that helps the development team understand and analyze the existing application, process or system.
- Several models can be constructed for analysis and representation. The table shows UML models.

UML Diagram	Description	Usefulness in Modeling an Existing System
Class diagram	A directed graph in which the vertexes represent classes and the directed edges represent different types of relationships between the classes. The vertexes also contain information that describes the properties of the classes.	Class diagrams are used to visualize domain models, which help the development team understand and communicate domain concepts and relationships between the domain concepts.
Use case diagram	A graph in which the vertexes represent abstractions of business processes and actors while the edges specify which actors interact with which business processes.	Use case diagrams are used to display an overview of the business processes of an existing application or a subsystem as well as the users that request the services of the business processes. Use case diagrams also show the boundaries of the existing application, system, or subsystems.
Sequence diagram	A directed graph in which the vertexes represent objects and the directed edges represent time-ordered messages or requests between the objects.	Sequence diagrams help the development team understand and analyze the existing business processes; that is, how the business objects process a user request in the existing business environment.
State diagram	A directed graph in which the vertexes represent system states and the directed edges represent state transitions caused by internal or external events.	State diagrams are useful for the modeling of event-driven systems or business activities.
Activity diagram	A directed graph in which the vertexes represent information-processing activities and the directed edges represent data flows and control flows among the activities. The control flows specify that the activities are performed sequentially, concurrently, and/or synchronously.	Activity diagrams are used to model information-processing activities of the existing application in which the activities relate to each other through complex data flow and control flow relationships.

# Example modeling

Activity diagram for an existing business process: how student applications to overseas exchange programs are manually processed.

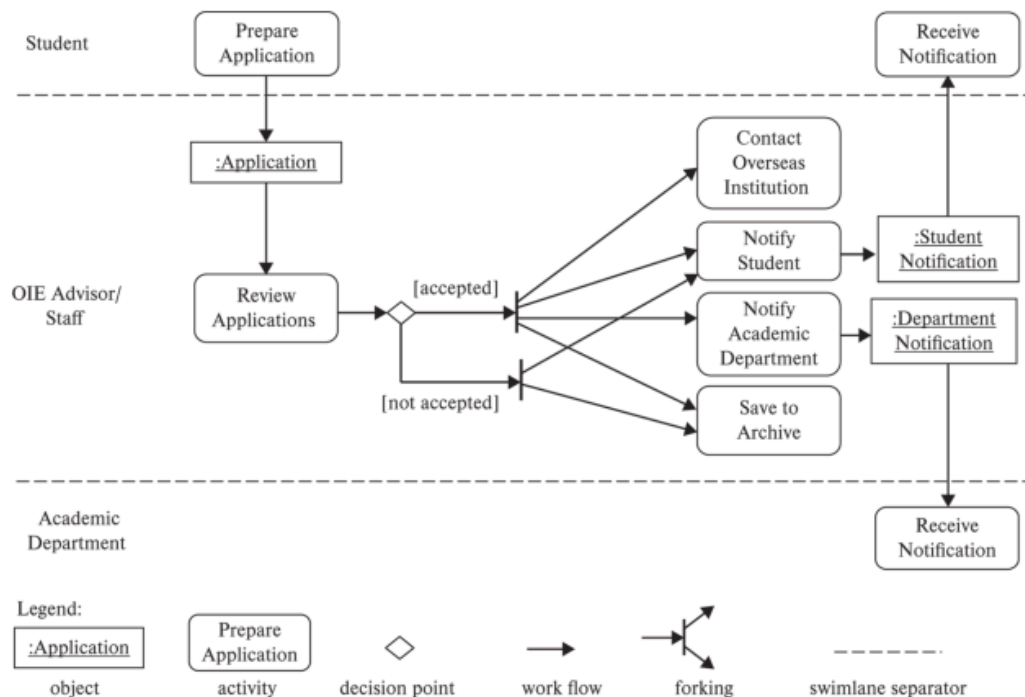
Modeling consumes considerable time and effort. Therefore, construct models only if they greatly help understanding and analysis



# Generating requirements from needs or Wishlist (The proposed system is called SAMS)

- Needs:
  - Students should be able to search for overseas exchange programs using a variety of search criteria.
  - OIE staff members should be able to enter, edit, activate, and deactivate overseas exchange programs.
  - Students should be able to create, edit, and submit applications for overseas exchange programs.
  - OIE advisors should be able to process online applications.
- Sample of requirements (note requirement phrasing)
  - R1. SAMS shall provide a search capability for overseas exchange programs using a variety of search criteria.
  - R2. SAMS shall provide a hierarchical display of the search results to facilitate navigation from a high-level summary to details about an overseas exchange program.
  - R3. SAMS shall allow students to submit online applications for overseas exchange programs

# Generating requirements from analysis models (The proposed system is called SAMS)



- R4. SAMS shall provide an online capability for students to prepare applications to overseas exchange programs.
- R5. SAMS shall facilitate review of applications by OIE advisors.
- R6. SAMS shall contact the overseas institution when an application is accepted by the OIE.
- R7. SAMS shall notify relevant entities when an application is accepted or rejected.
- R7.1. SAMS shall notify the student when an application is accepted or rejected.
- R7.2. SAMS shall notify the academic department when an application is accepted.
- R8. SAMS shall archive accepted and rejected applications.
- R9. SAMS shall allow students to view and print the notification letter concerning the decisions of their applications.
- R10. SAMS shall allow secretaries of the academic departments to view and print notification letters online.

# User stories are widely used by agile methods as a requirements gathering method.

Each user story briefly describes a capability that a user role wishes to have. Each user role may have several user stories.

Each user story briefly describes a capability that a user role wishes to have. Each user role may have several user stories.

1. As a medical assistant, I need to create patient records for new patients.
  2. As a medical assistant, I need to search and retrieve patient records using a patient's birthday and/or the patient's last name, first name.
  3. As a medical assistant, I need to keep track of a patient's medical test records.
- Page 78
4. As a medical doctor, I need to document all conversations with a patient.
  5. As a medical doctor, I frequently need to search conversations with a patient to locate certain information.
  6. As a medical doctor, I need to search a patient's medical test records.
  7. As a medical doctor, I want to access the system from home or from anywhere remotely.
  8. As an information security officer, I want to protect the system from unauthorized access.

# Running example: Library management

## System: Description:

- It is needed to automate a currently manual managed library. The library receive visitors for searching for books, read books inside the library. A visitor can ask to be registered as library-member or client. A library member can borrow books to read them at home

## Today task (Report):

- Do the requirement engineering stage for the library management system.
- Work in a team in which some members play the role of users and other play the role of developers
- Model the existing system, write wishlists, write user stories
- Augment your study with the necessary use cases with drawing

# Modularity

In software engineering project design stage

Dr Hamed Hemed



# What is modularity

Modularity is the decomposition of the software system into components, design and implement each component on its own, and integrate the components to get the whole modular system.

A components is a pieces that are as independent of each other as possible. Ideally, each component should be self-contained and have as few references as possible to other components

# Why modularity?

Managing complexity: Dealing with large complex system as designing, implementing, debugging, testing and maintaining a component is much simpler and easier of dealing with the system as a whole

Reusability: A major software engineering technique is to reuse software components from a library or from an earlier project reducing the time and the cost for a project.

Support team working as each member can design, implement, test a component which allow all team members to work in parallel

# The size of a component in a software project

## Component desirable attributes

- Provides a useful service
- Performs a single function
- Has the minimum of connections (ideally no connections) to other components.
- Self contained as global data (shared between components) is harmful

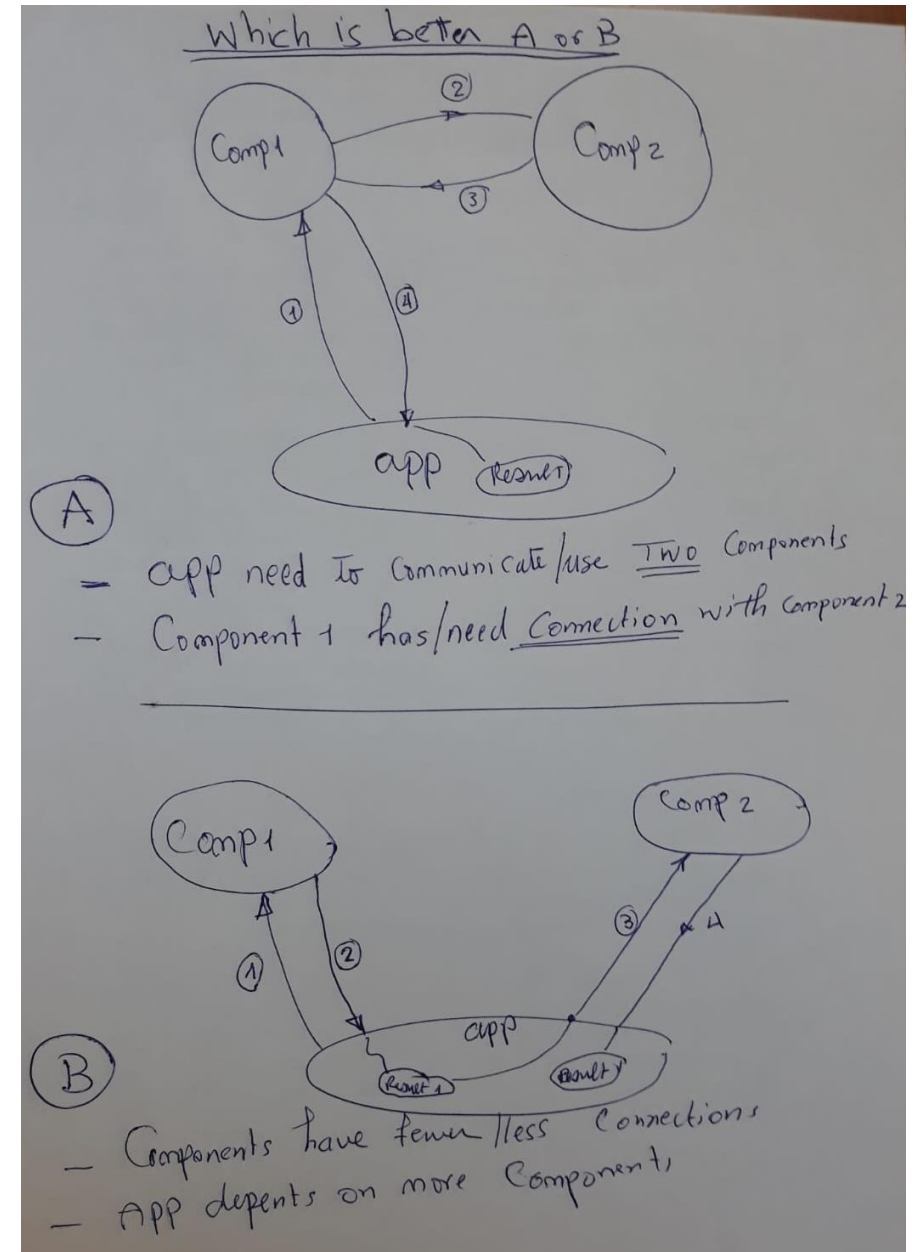
## Component size

- A common point of view is that a component should occupy no more than a page of coding (about 40–50 lines).
- Instead of decomposing the system in too many small components, a hierarchy or few large components each further divided into sub-components is better.

# Few or no connection between components

## Case B is better because it leads to:

- Component interface simplicity: What if component 2 has many ways or options to do its work, how this will affect the interface between component 1 and the application
- The application has more control and flexibility with simpler and more direct interface



# Approaches to structuring software in a highly modular fashion: Information hiding

Information hiding (encapsulation): For an object (example a data structure) , all the following should be parts of the same component:

- The structure itself
- The statements that access the structure
- The statements that modify the structure
- Good encapsulation helps changing implementation details without affecting interaction with the component

Information hiding meets three goals:

- **Changeability:** If a design decision is changed, such as a file structure, changes are confined to as few components as possible and, preferably, to just a single component.
- **Independent development:** Each team member can work on a component
- **Comprehensibility:** understanding individual components independently of others

# Approaches to structuring software in a highly modular fashion: Coupling and cohesion

A piece of software should be constructed from components in such a way that there is a minimum of interaction between components (low coupling) and, conversely, a high degree of interaction within a component (high cohesion)

Promoting less coupling using:

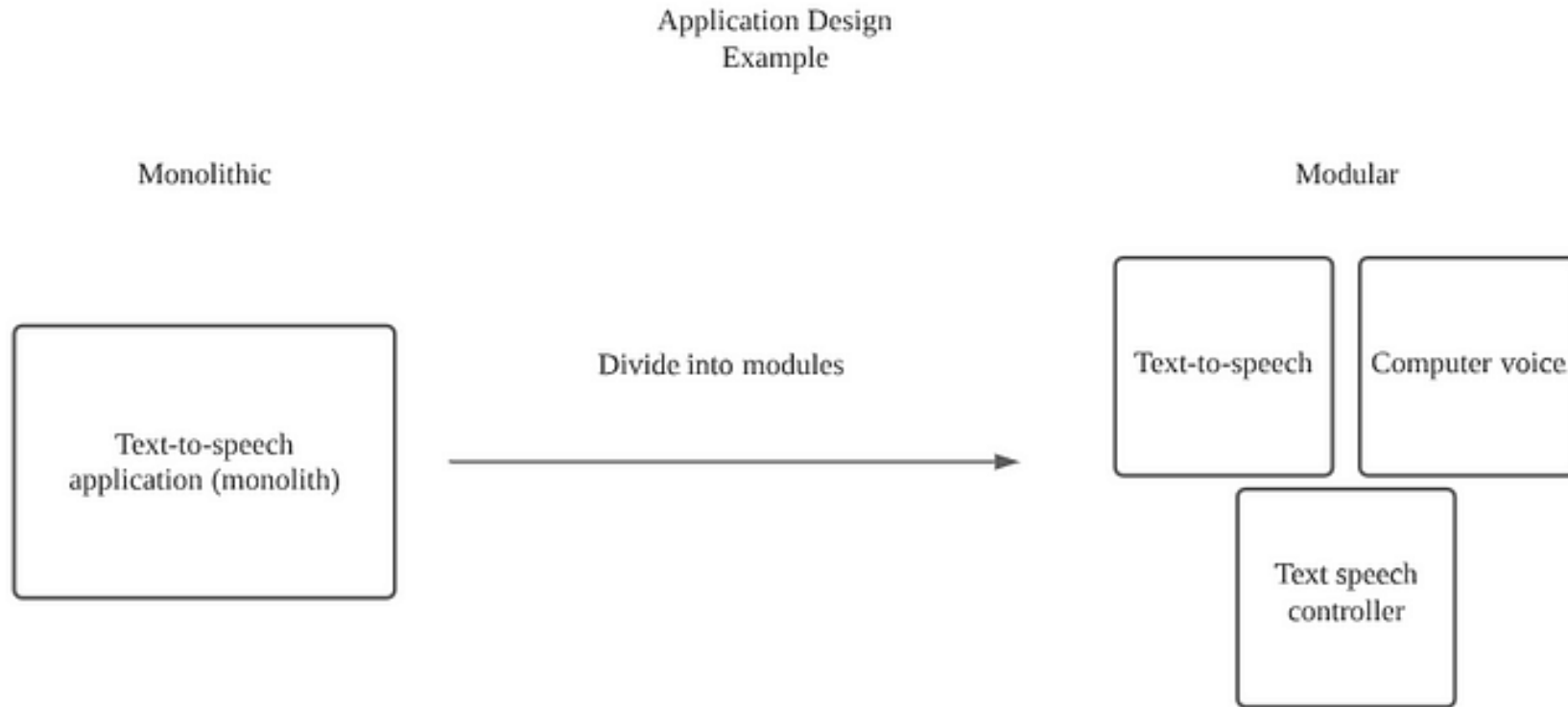
- Using as few parameter as possible when calling between component
- Parameters are just data but not behavior switching ( a switch parameter is a parameter that control behavior. Ex if the first parameter is 0 do X , if 1 do Y. X or Y are done on data provided by the reset of the parameters)

# Example modularization: text-to-speech application

Consider a text-to-speech application that will translate a user's input text into speech and read it out loud. It should be able to:

- Parse a user's input text
- Use a selected computer's voice to read out the text
- Have a controllers that can speed up or slow down the computer's speech if the user chooses

# Example modularization: text-to-speech application





# Text-to-speech application: Module functions

## Text-to-speech:

- Parses the user's text to be read out loud
- Understand the semantics of the text, punctuation to direct for more natural speech

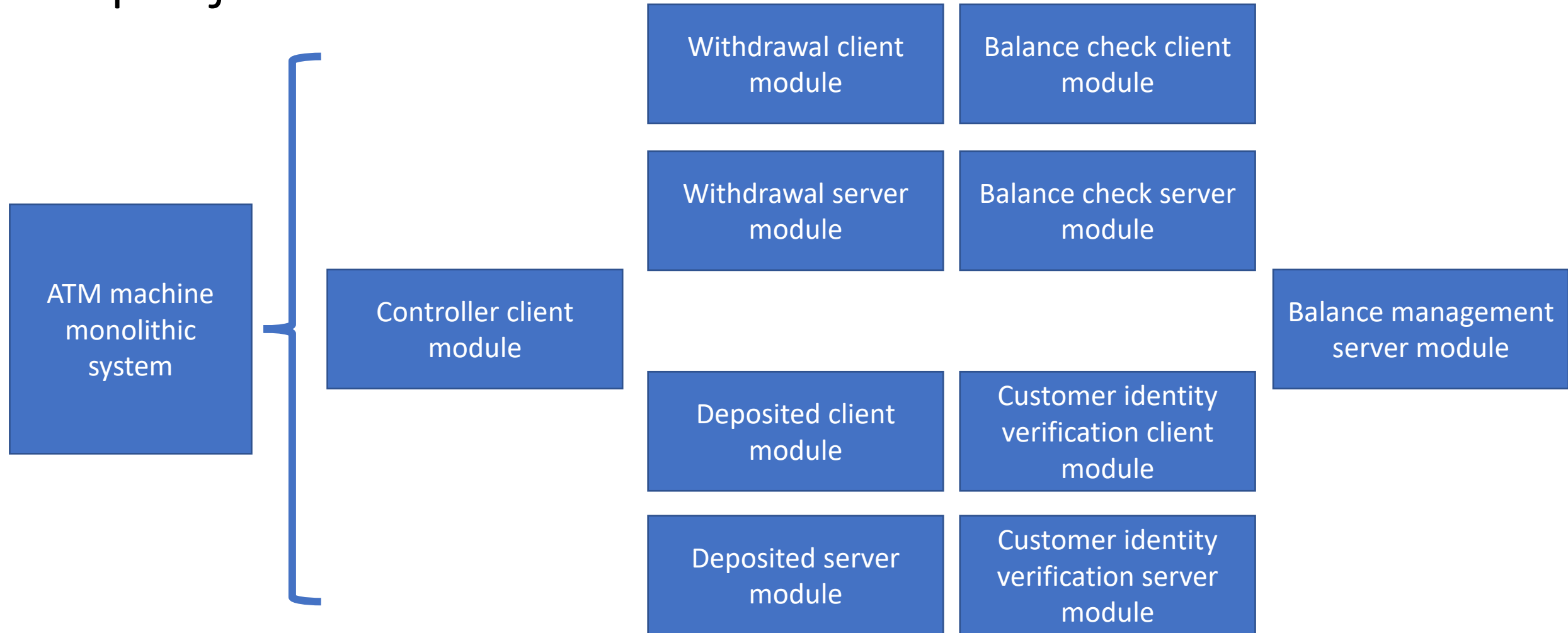
## Computer voice:

- Stores and provides computer voices that the user can choose
- Adding, removing and modifying of computer voices
- Select a computer voice based on some parameters

## Text speech controller:

- Controls that speed of the speech that the user chooses: slowing down, speeding up, women, men, etc.

# Example modularization of the ATM machines project



# Functional decomposition

Software system design technique

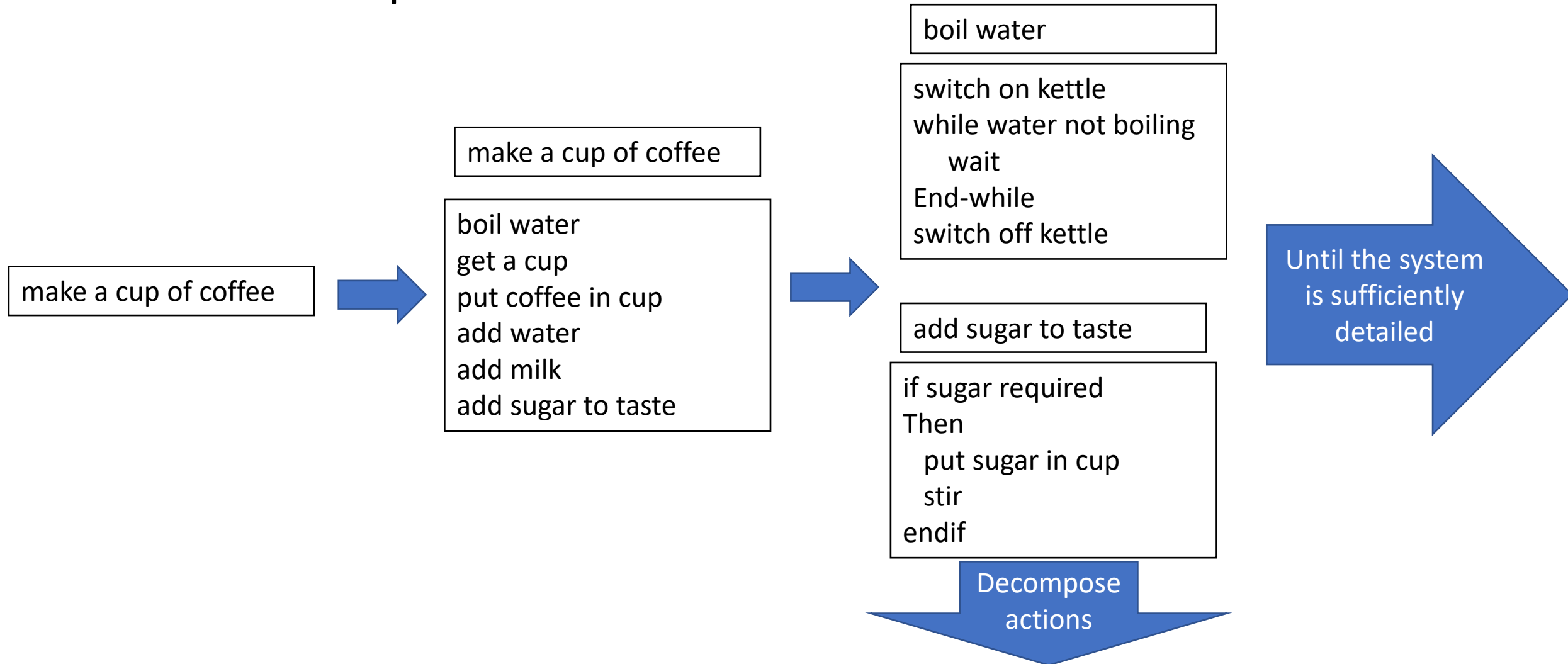
# What is functional decomposition?

Functional decomposition is a technique for designing either the detailed structure or the large-scale structure of a program or module.

Functional decomposition is a method that focuses on the functions, or actions, that the software has to carry out

To do Functional decomposition: write down, in English, a single-line statement of what the software has to do. If the software performs several functions (as most GUI-driven software does) write down a single line for each function. Then decompose each action into simpler action, decompose the resulting actions into simpler actions, and so on until we have sufficiently detailed steps for each action

# Example: write a program to direct a robot to make a cup of instant coffee



# How to do the decomposition?

## What to decompose?

- Any action that has some complexity
- Until the decomposition is operations allowed by the programming language/system

## How to decompose?

- Using English concise statement
- We can use structured statements:
  - If then endif
  - If elseif elseif ... else endif
  - While ... end-while
- We can use parameters for actions

# Example: Alien, defender game design

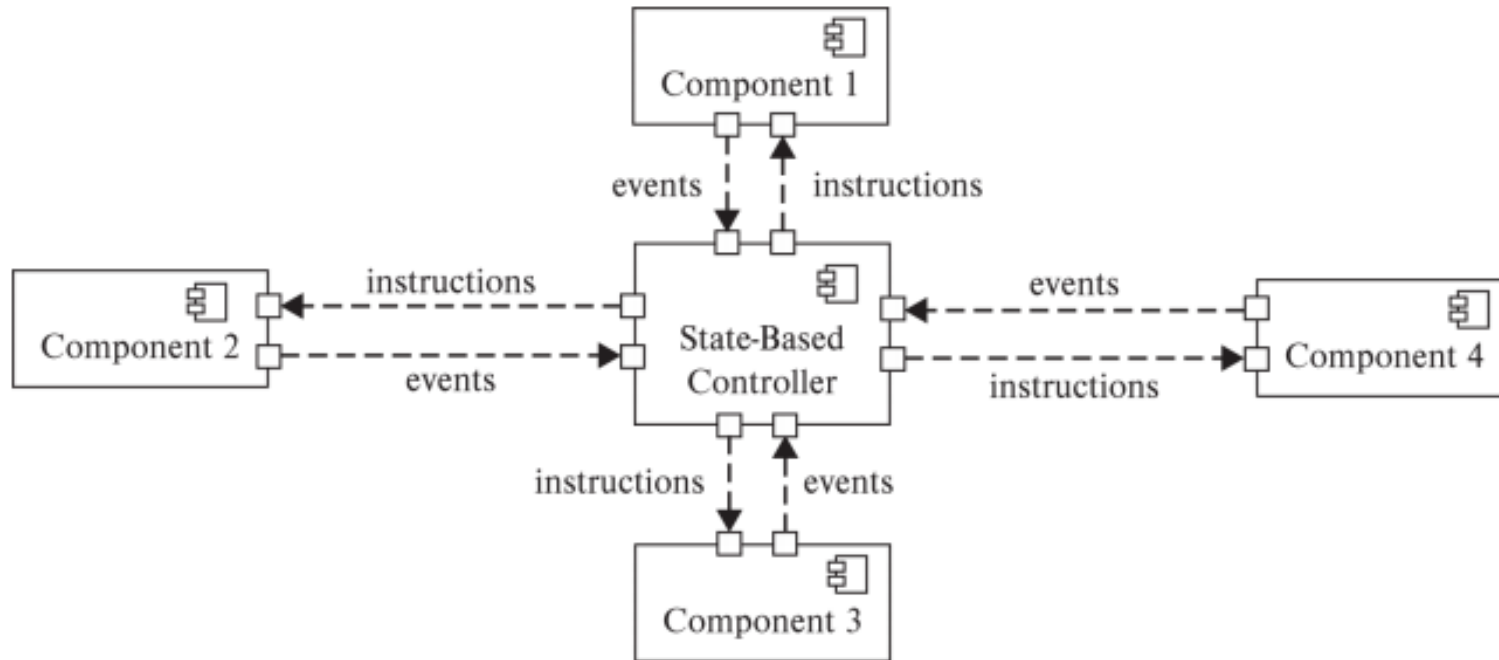
## Program specification

A window displays a defender and an alien. The alien moves sideways. When it hits a wall, it reverses its direction. The alien randomly launches a bomb that moves vertically downwards. If a bomb hits the defender, the user loses, and the game is over. The defender moves left or right according to mouse movements. When the mouse is clicked, the defender launches a laser that moves upwards. If a laser hits the alien, the user wins and the game is over.

A button is provided to start a new game.



# Event driven system architecture



- System objects: Alien, Defender, Bomb, Leaser, timer
- System states: Game-On, Game-Off, Bomb-Exists, Leaser-Exists
- System events: mouse-click(firing leaser), mouse-move(move-defender), Fire-Bomb(Randomly), User Won, System-Won



# Example: Alien, defender game ...continued

This is an event-driven program then we identify actions done on each events

Start button event

create defender  
create alien  
start timer

timer event

move auto-moved objects:  
-Alien  
-Laser if any  
-Bomb if any  
  
display the objects  
check hits

Mouse click event

create laser

Mouse moved event

move defender

# Example: Alien, defender game ...continued

Decompose

move auto-moved objects:

move alien  
move bomb  
move laser  
check boundaries

Display all objects

display background  
display defender  
display alien  
display laser  
display bomb

Check hits

```
if collides(laser, alien) then
    endOfGame(userWins)
else if collides(bomb, defender) then
    endOfGame(alienWins)
else continue-game
endif
```

# Example: Alien, defender game ...continued

Decompose

Collides (a,b)

```
if (a.x > b.x  
    and a.y < b.y + b.height  
    and a.x + a.width < b.x + b.width  
    and a.y + a.height > b.y ) then  
    return true  
Else  
    return false  
endif
```

# Problems with functional decomposition

Data plays second fiddle to the actions that must be taken

Data is shared between actions and no data-protection could be assured

Following the functional decomposition, we can get several design, the method itself gives no guidance as to which design is the best

# Use cases

# Key points

- A use case is a business process. It begins with an actor, ends with the actor, and accomplishes a business task for the actor.
- Use cases are derived from requirements and satisfy the requirements.
- Use cases and subsystems development and deployment are planned to meet the customer's business needs and priorities.
- Use cases specify how the system will interact with an **actor** to deliver the capability but not involve background technical object interaction

# What is an actor? What is a use case?

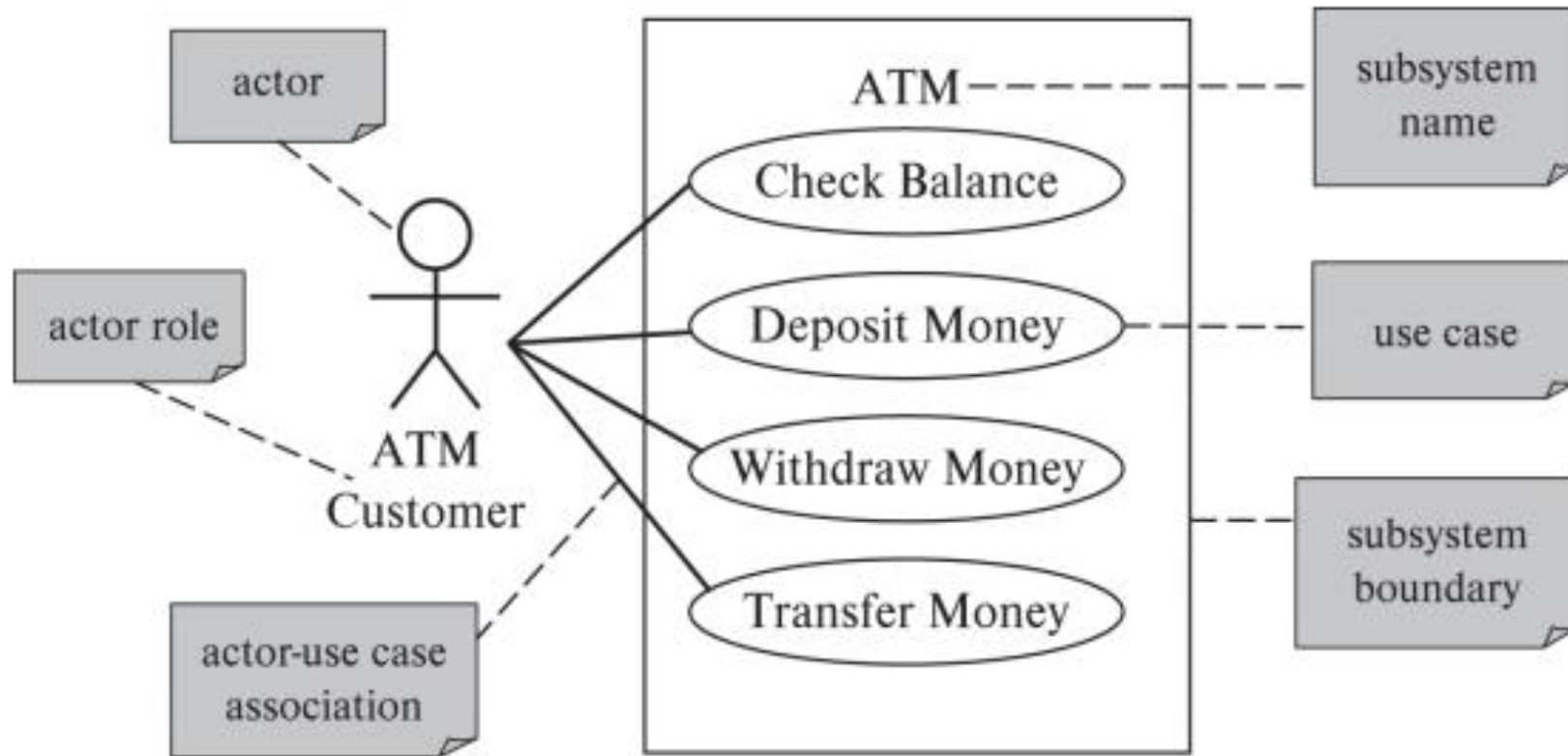
- An actor is a (business) role played by and on behalf of a set of (business) entities or stakeholders that are external to the system and interact with the system. It is often a user but may be a hardware device or subsystem/entity external to the software system.
- A use case is a business process. It **begins with an actor, ends with the actor**, and accomplishes a **business** task for the actor.

# Example uses cases

- Consider an ATM application that allows a customer to deposit money, check balance, withdraw money, and transfer money between bank accounts.
- We have 4 use cases each begin and end with an actor (user)
  - Deposit money.
  - Check balance.
  - Withdraw money.
  - Transfer money between bank accounts



# Example use cases diagram



# BUSINESS PROCESS, OPERATION, AND ACTION

- Consider the ATM application
- Get Balance from Database: Is not a use case:
  - What is the business process?
  - Which actor begin it?
  - It can be an operation is a use case such as **check balance**
- Enter password is not a use case
  - What is the business process?
  - It may be an action in a use case
- An operation is a series of actions or instructions to accomplish a step of a business process.
- An action is an indivisible act, movement, or instruction that is performed during the performance of an operation.

# BUSINESS PROCESS, OPERATION, AND ACTION

ATM	Deposit Money / Withdraw Money	Start	Insert card.
		Authenticate	Enter password, press Enter key.
		Do transaction	Select transaction type, enter deposit/withdraw amount, insert cash/take cash, take deposit/withdraw slip.
		Finish	Press Exit key, take ejected card.

Class Diagram Editor	Edit Diagram	Open Diagram	Click File, select Open, navigate to directory, select file, click OK button.
		Make Changes	
		Add class	Right click in canvas, select Add Class, fill in class information, click OK.
		Delete class	Click a class to select it, press Delete key, press OK button to confirm.
		Modify class	Double click a class, change class information in the pop-up dialog, click OK.
		Save Diagram	Click File, select Save.
		Exit Editor	Click File, select Exit.

# Example: Extract use cases

Consider the design and implementation of a graphical editor for drawing UML class diagrams. The editor allows the user to create new diagrams, save diagrams, delete diagrams, and edit diagrams. Which of the following can be considered use cases?

- Create new diagram file
- Save diagram
- Edit a diagram.

Create new diagram file and save diagram are not use cases because they are not business process. Because incremental delivery allows the team to deliver some use cases before the other, let us assume that “create new diagram” or “save a diagram” is delivered first. In this case, the user won’t be able to do anything that is meaningful, such as editing the diagram. Edit a diagram need to create or open a diagram file, create or modify object and save a drawing and express a business process so it is a use case

# Deriving use cases

- **Step 1. domain specific verb-noun phrases** representing business processes are identified from the requirements (usually found in top-level requirements). Also derived are actors that use the use cases and subsystems that contain the use cases.
- **Step 2.** Constructing use case diagrams. In this step, the use cases, actors, subsystems, and their relationships are visualized using UML use case diagrams.
- **Step 3.** Specifying use case scopes. In this step, the scope of each use case is specified, producing a high-level use case, which specifies when and where the use case begins and when it ends.
- **Step 4.** Producing a Requirement-Use Case Traceability Matrix. In this step, a Requirement-Use Case Traceability Matrix is produced to show the correspondence/relevance between the use cases and the requirements.

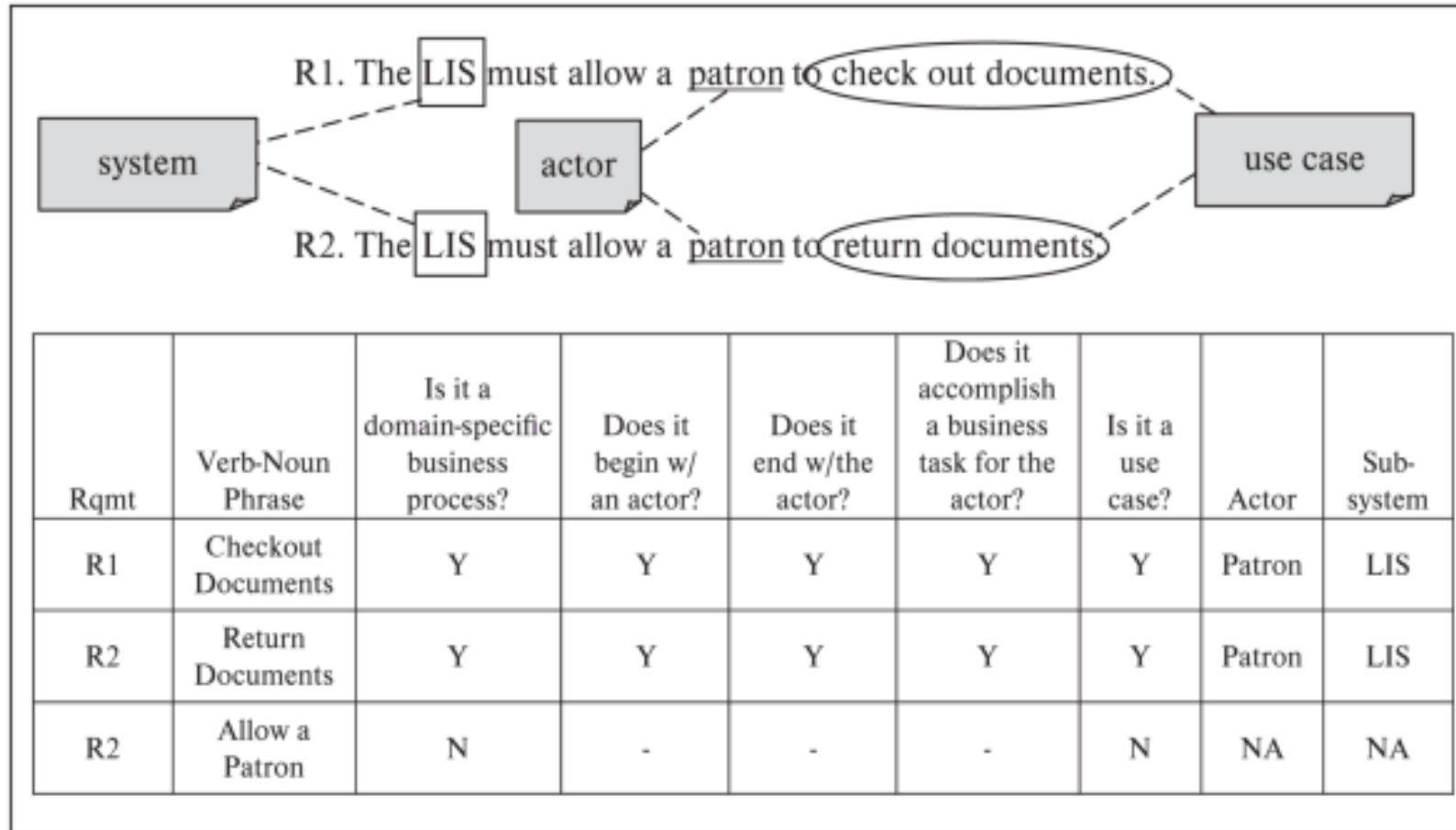
# Deriving use cases continued

- **Step 5.** Reviewing use case specifications. The use case diagrams, use case scopes, requirement–use case traceability matrix, and use case–iteration allocation matrix are reviewed.
- **Step 6.** Allocating use cases to iterations. The use cases are assigned to iterations according to use case priorities, their dependences and the team's capacity.

# Example

- Derive use cases from the following requirements for a library information system (LIS):
  - R1. The LIS must allow a patron to check out documents.
  - R2. The LIS must allow a patron to return documents.

# Example solution





# Actor-System Interaction Modeling (ASIM) for use cases

# Stages for working out use cases

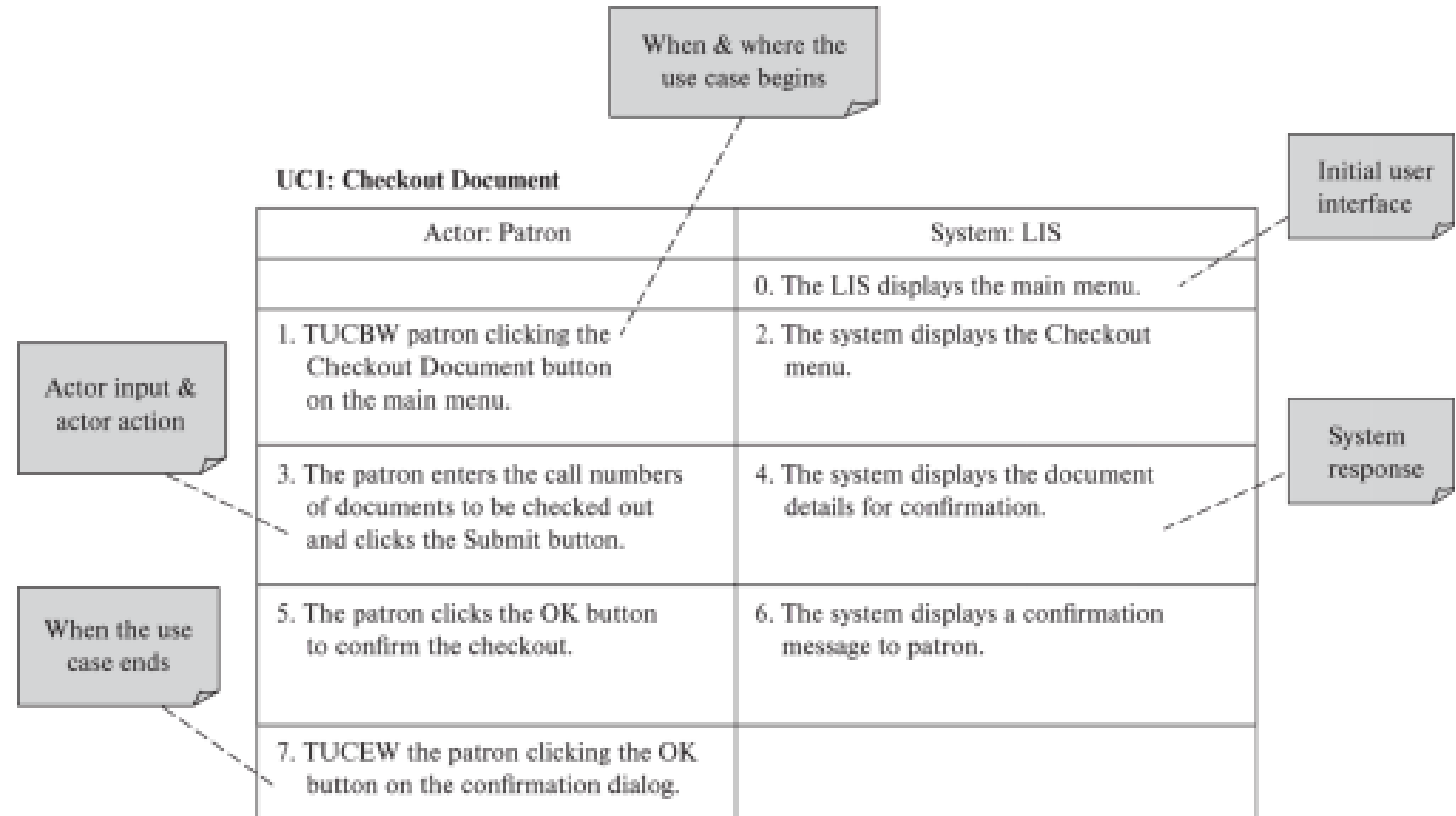
**Abstract Use Case.** An abstract use case is a verb-noun phrase that highlights what the use case will accomplish for the actor.

**High-Level Use Case.** A high-level use case consists of a TUCBW clause that specifies when and where a use case begins and a TUCEW clause that specifies when it ends. TUCBW and TUCEW stand for “this use case begins with” and “this use case ends with,” respectively. In other words, a high-level use case specifies the scope of the use case. UML use cases diagram model use case at this level.

**Expanded Use Case.** An expanded use case specifies how an actor will interact with the system. It is a continuation and refinement of a high-level use case. It specifies the interaction using a two-column table. The left column specifies the actor input and/or actor actions, and the right column specifies the system responses

# Actor-System Interaction Modeling (ASIM) for a use case: Example 1

ASIM: is a table description for the actor-system interaction, the left column specifies the actor input and/or actor actions, and the right column specifies the corresponding system responses.



# Importance of ASIM

It specifies the actor–system interaction or system’s interactive behavior that the subsequent design, implementation, and testing can follow.

It can be used to communicate to future users about the actor–system interaction behavior. It is useful for acquiring user feedback.

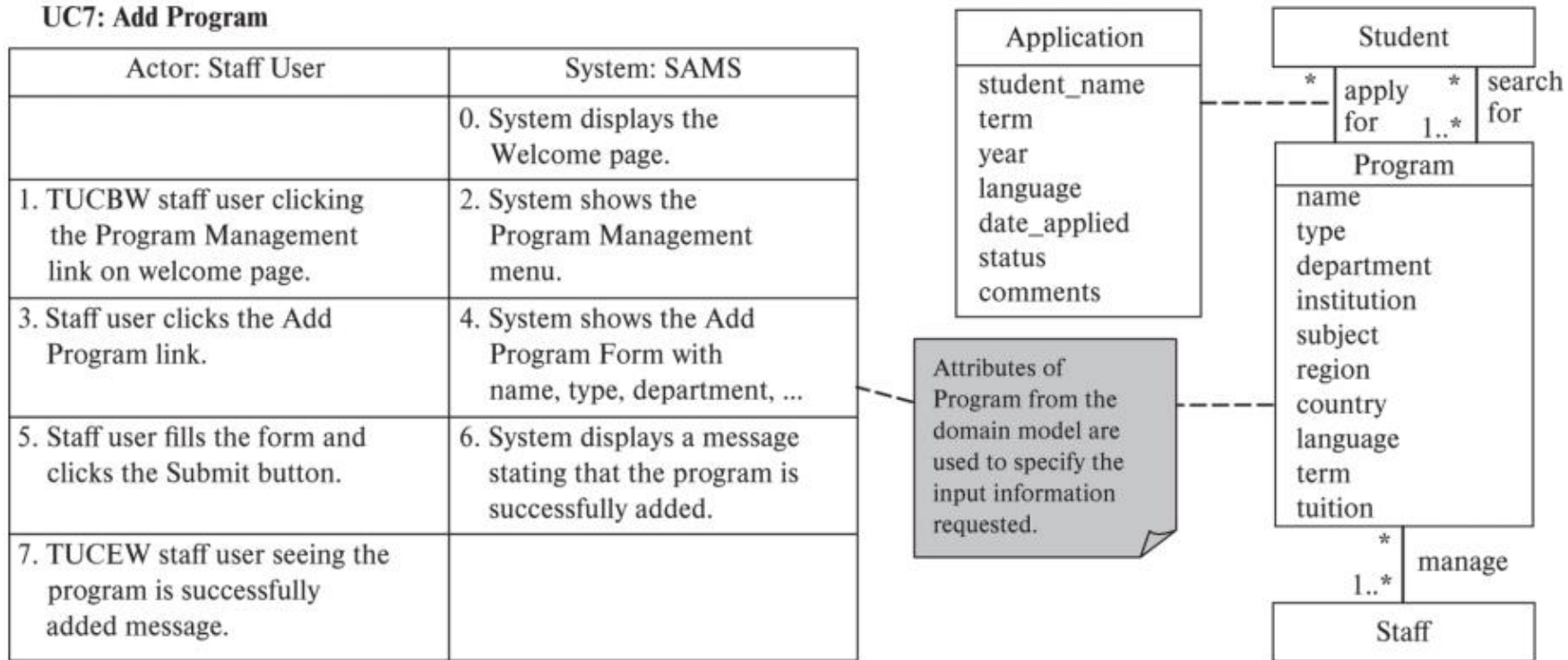
It can be used to generate a preliminary user’s manual. This is because the expanded use case describes exactly how the user will use the system to accomplish a business task. The preliminary user’s manual facilitates a potential user to experiment with a prototype of the system.

If updated timely according to changes in subsequent design and implementation phases, the expanded use case specification can be used to generate the as-built user’s manual. This reduces the increment or system deployment effort, cost, and time.

It can be used to generate use case–based test cases or test scripts. This will be described in

# Actor-System Interaction Modeling (ASIM) for a use case: Example 2

Note: Use case refers to domain modeling. The two must stay in sync, so one can update the other at any time of the design



# ASIM and alternate flow

UC7: Add Program	
Actor: Staff User	System: SAMS
1. TUCBW staff user clicking the Program Management link on welcome page.	2. System shows the Program Management menu.
3. Staff user clicks the Add Program link.	4. System shows a dialog with two options: (a) fill a form (b) upload a file
5. Staff user selects (a) fill a form, or (b) upload a file	6. System displays (a) an Add Program form, or (b) a dialog for uploading a file
7. Staff user either (a) fills the Add Program form and clicks the Submit button, or (b) locates the program file and clicks the Upload button	8. System displays a message stating that the program is successfully added.
9. TUCEW staff user seeing the program successfully added message.	

Here there are twp flows starting from stage 4 in the usecase

Note: Only normal flow is modeled in ASIM. Exceptional flow such as handling exception and system error are not modeled in ASIM and are considered implementation details not design aspects

# ASIM: Including other used cases

UC3: Transfer Funds	
Actor: Investor	System: Fund Manager
	(0) System displays Fund Management page.
(1) TUCBW investor clicking the Transfer Funds link on the Fund Management page.	(2) System displays fund transfer options: (a) Transfer Funds Online. (b) Transfer Funds with a Check. (c) Transfer Funds with Transfer Form.
(3) Investor selects one of the fund transfer options: (a) Transfer Funds Online. (b) Transfer Funds with a Check. (c) Transfer Funds with Transfer Form.	(4) TUCCW the selected fund transfer use case: (a) UC4: Transfer Funds Online. (b) UC5: Transfer Funds with a Check. (c) UC6: Transfer Funds with a Fund Transfer Form.
(5) TUCEW investor seeing: (a) Online fund transfer is completed message. (b) Fund Transfer slip. (c) Fund Transfer Form to print, fill, sign and mail.	

TUCCW: This use case continue with, complete the flow by another use case.

# ASIM: commonly seen problems

	Actor: Student	System: SAMS	
(a) No specification of which page.	1. TUCBW the student clicking the "Register Button."	2. System shows a registration screen.	(b) Should not contain a blank entry during interaction.
(c) Steps 3 and 4 should be merged into one step.	3. Student fills the registration form.		
	4. Student clicks the "Submit" button.	5. System retrieves student record from database, validates the information provided by the student.	(d) Should not specify background processing.
(e) Should not contain a blank entry during interaction.		6. System sends an email to the address with a password.	(f) Should not specify background processing.
		7. System displays a "Registration is successful and the password is sent via email" message.	
(g) Missing TUCEW clause here.			



# Guidelines

- Keep it simple:
  - An expanded use case should contain only a few steps. Moreover, each step should be simple, clear, and straightforward. If an expanded use case has more than half a dozen steps , then there are two possibilities:
    - It is likely that lower-level operations are treated as steps of the process. The solution is to hide lower-level operation.
    - The use case is in fact a concatenation of two use cases. The solution is to split the use case into two or more use cases
- Use alternative flows with care because it complicate the design
- Do not show background processing and exception handling in expanded use cases.
- Good enough is enough. Quickly move on to software implementation.

# Object Interaction Modeling (OIM)

Modeling background processing of use cases

# ASIM and OIM for use cases

Object interaction modeling (OIM) develops high-level algorithms specifying how objects interact with each other to produce system responses from actor input.

Object interaction behaviors are specified using UML sequence diagrams.

OIM deals with background processing of use cases while actor–system interaction modeling deals with foreground processing.

# Object-Oriented Paradigm for software

The basic building blocks are objects.

The real world as well as the software system is viewed as consisting of objects relating to and interacting with each other.

The objects relate to each other through inheritance, aggregation, and association relationships.

They interact with each other by requesting services from, or performing actions on, other objects.


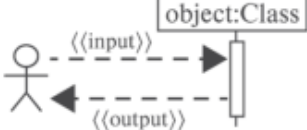
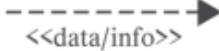

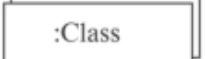
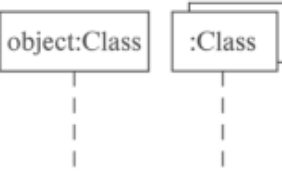

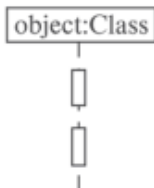

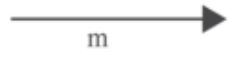
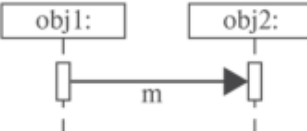

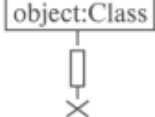
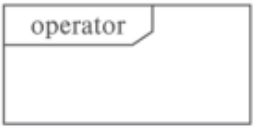
The objects must not interact with each other arbitrarily. They must interact in some way to accomplish the business processes of the use cases

# OIM for analysis and for design

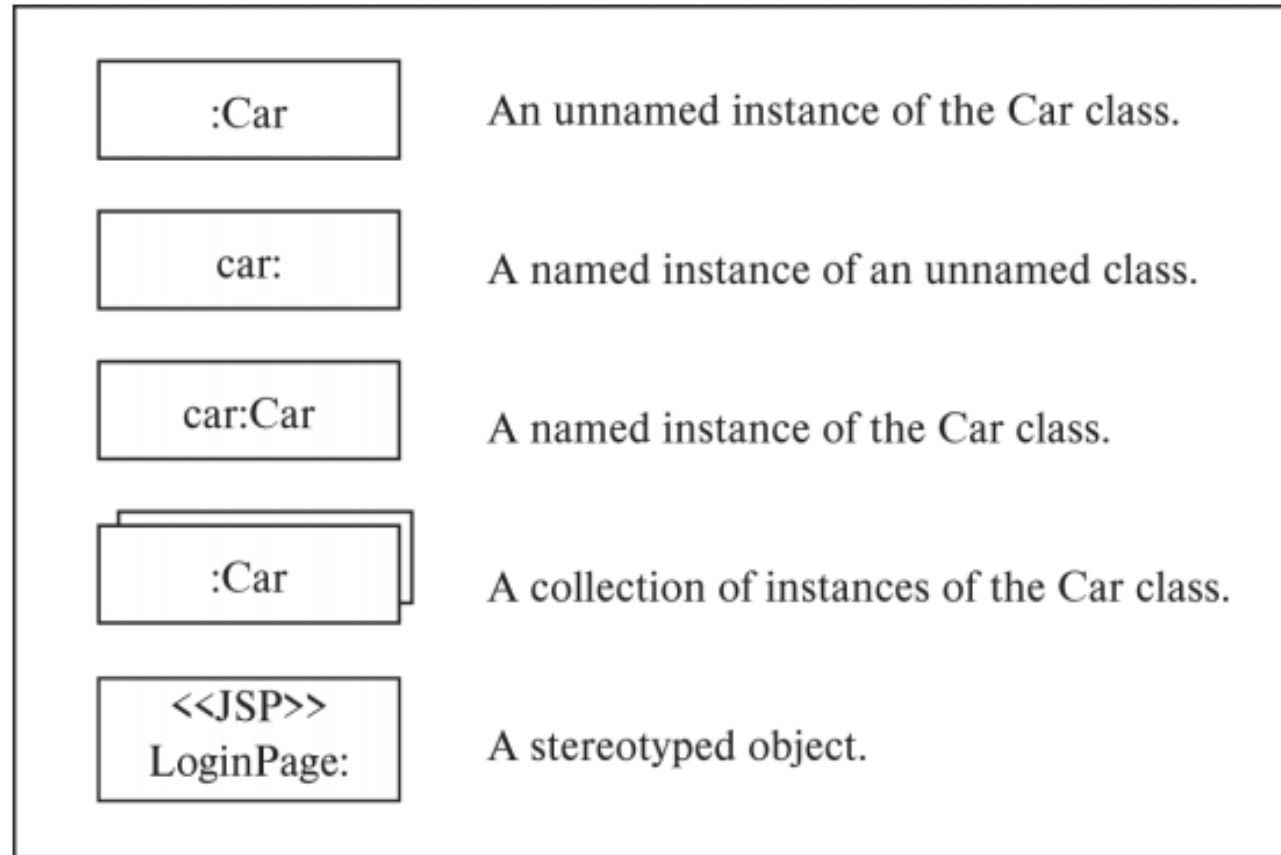
- OIM is used for
  - Analysis (for existing system either manual or computerized):
    - Help understand how objects interact in the existing system or existing business processes or existing software system to be enhanced.
    - Help identify problems and limitations of the existing system or existing business processes.
  - Design
    - Design and specify high-level algorithms that describe how objects interact in the proposed software system to process the use cases
      - Design of object interaction behavior: sequence of messages between objects
      - Object interfacing: signature and return types of messages
      - Design for better: apply known to be effective design patterns/guidelines such as low coupling, high cohesion, proper assignment of responsibilities to objects, and easy adaptation to changes in requirements

# OIM diagram

- OIM diagrams
  - Sequence diagram (Studied here)
  - Communication diagram (Not studied here)

Notion	Notation	Semantics	Connectivity
Actor		A role played by a set of entities or stakeholders that are outside of the system and interact with the system.	
Stereotyped message		A stereotype, enclosed in a pair of double-angle brackets, “<<” and “>>,” lets the modeler introduce a modeling concept not in UML, such as a design pattern or a database. A stereotyped message is a message between an actor and an object that has an application-specific interpretation.	
Object Collection	(a)  (b) 	(a) An object of a class. (b) A collection of objects of a class. These are placed on top of the lifeline. The colon indicates an object, or a collection of objects. The object name is placed before the colon and the class name after the colon.	
Lifeline		Indicating that the object exists in the system but it is not executing a method.	
Method execution		Indicating, by the rectangular shape, that the object is executing one of its methods.	
Message passing		A message m is sent from one object to another object, i.e., one object calls a function m of another object.	
Object destruction		The cross at the tip of the lifeline indicates that the object is destroyed or ceases to exist.	
Combined fragment		A combined fragment expresses repetition or conditional execution of a portion of a sequence diagram, where operator can be <b>loop</b> , <b>opt</b> or <b>alt</b> ; opt mean if-then and alt means alternate such as if-then-else.	

# Representing objects, collections in sequence diagram



# Sequence diagram example: The scenario

1- Web user submits uid and password to LoginPage.

2- LoginPage verifies uid and password with LoginController.

3- LoginController gets user (object) from the database manager (DBMgr) using uid.

4- DBMgr returns user (object) to LoginController.

5- LoginController verifies password with user (object).

6- User (object) returns result to LoginController.

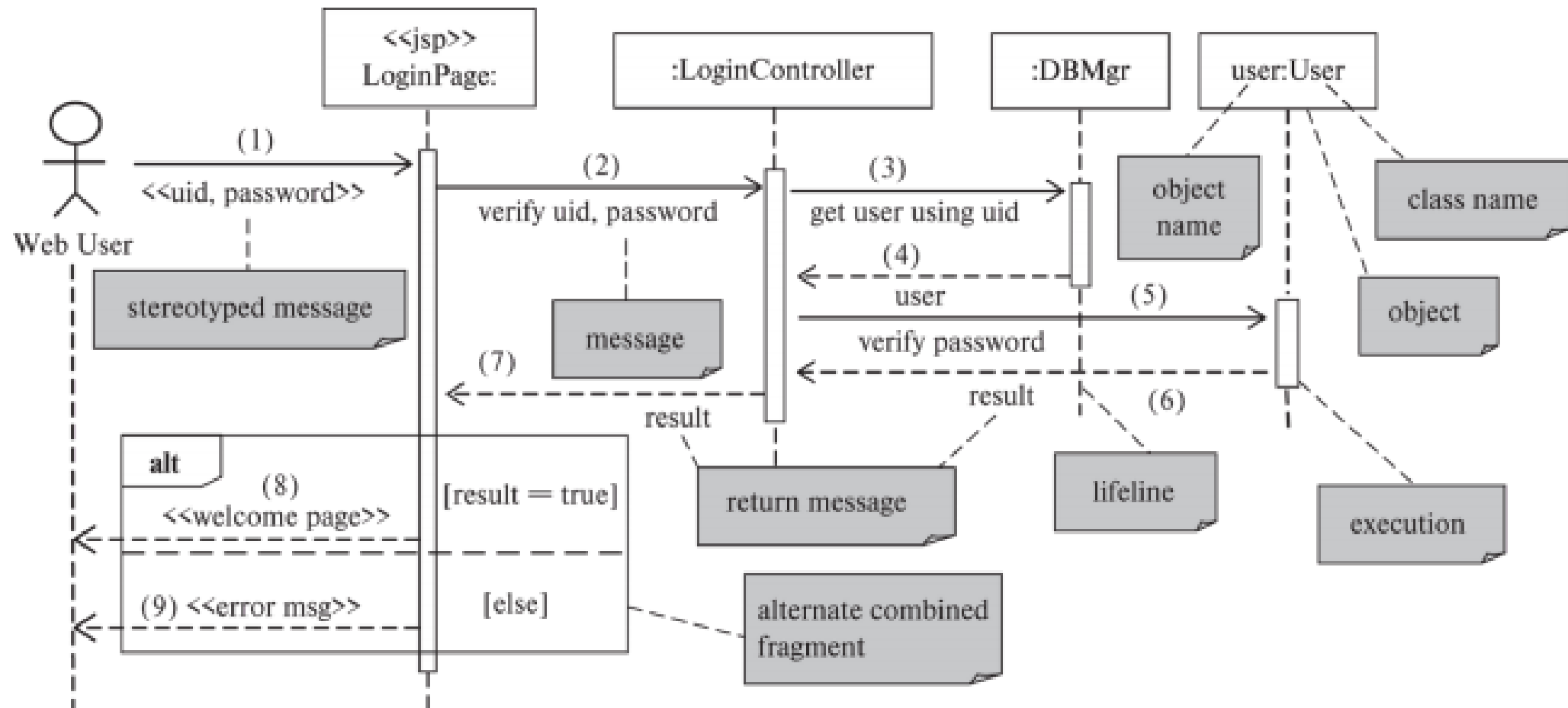
7- LoginController returns result to LoginPage.

8- If result is true, LoginPage shows the WelcomePage.

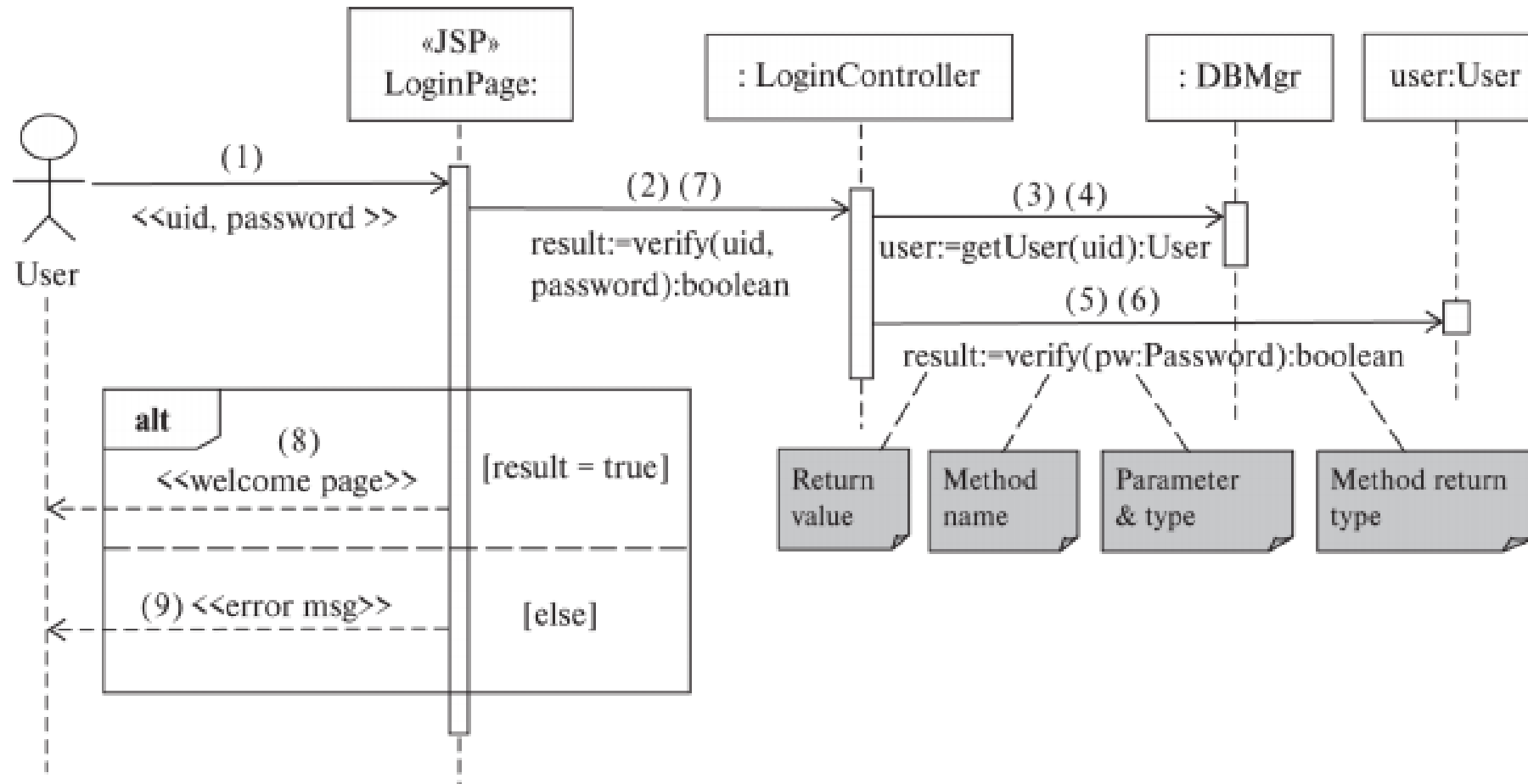
9 Else, LoginPage shows an error message.



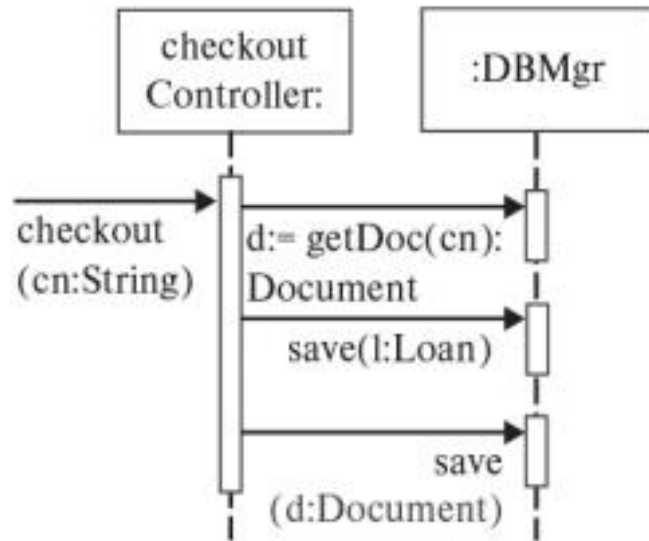
# Informal sequence diagram example uses for analysis or early design



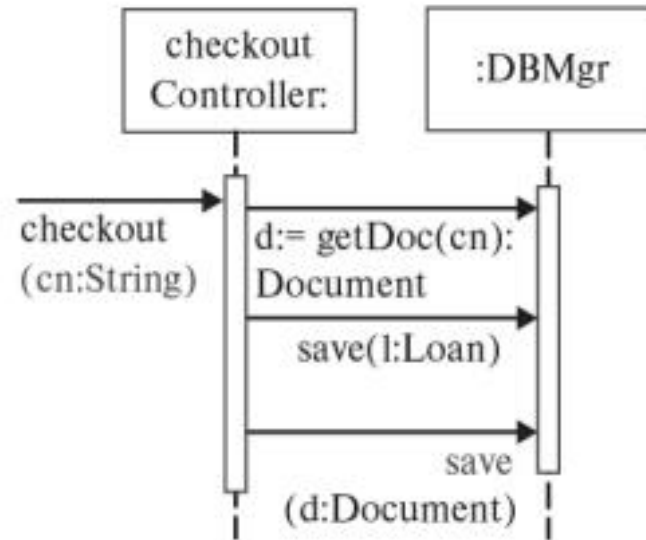
# Informal sequence diagram example uses for analysis or early design (messages=>function2)



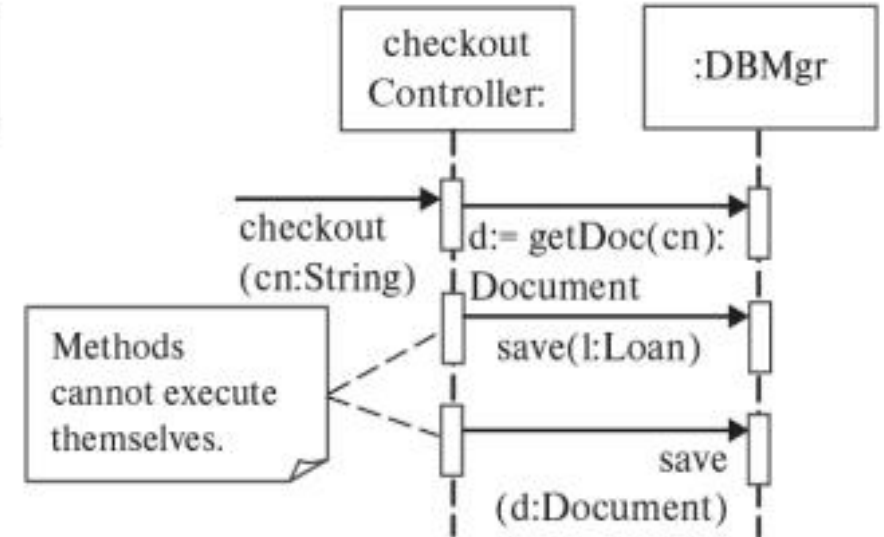
# Using the notation correctly (1)



(a) Correct: during the execution of `checkout(...)`, three separate calls to `DBMgr` are made.

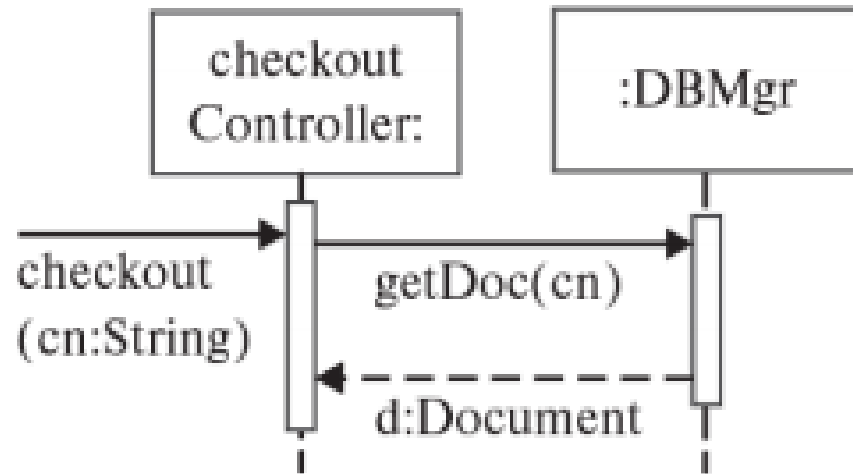


(b) Incorrect: the long rectangle beneath `DBMgr` should split into three as in (a)

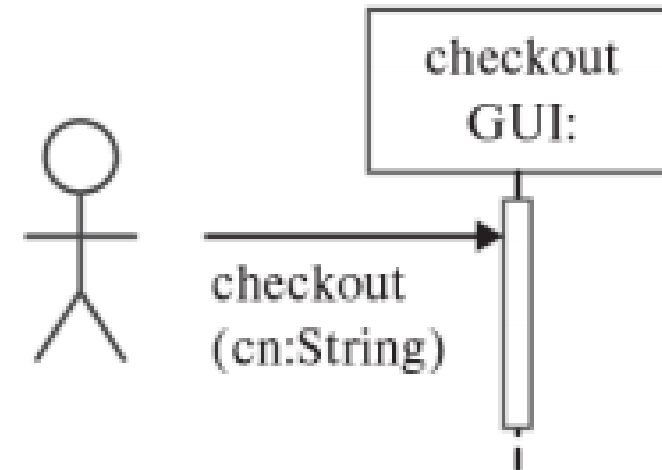


(c) Incorrect: methods must be called to execute.

# Using the notation correctly (2)



(d) Not preferred: the back dashed arrow line can be interpreted differently. (a) is preferred.



(e) Incorrect: an actor cannot call a function of an object; should use a dashed line and stereotype message.

# Steps for OIM

Step 1. Collect information about existing business processes.

Step 2. Identify nontrivial steps from the expanded use cases assigned to the current iteration.

Step 3. Write object-interaction scenarios for the nontrivial steps.

Step 4. Convert the scenarios into scenario tables. This step is optional.

Step 5. Convert scenarios or scenario tables into sequence diagrams.

Step 6. Review the object interaction models.