CPSC 362
Foundations of Software Engineering

Mustafa Ibrahim
mibrahim@fullerton.edu

Instructor Info

Mustafa Ibrahim

Instructor at California State University Fullerton

mibrahim@Fullerton.edu

Syllabus is located on Canvas

Credits: Thank You to Dr. Mira Kim for providing me her lecture material as a starting point for all my lectures.

About Me

Background

- ~20 Yrs XP within Software Organizations in Bio-technology
- 7 Years teaching at the University Level
- B.S. at Cal Poly, San Luis Obispo
- M.S. at Cal State, Northridge

Expertise

- Building software products used in health care from implantable devices and lab instruments to Cloud Informatics systems.

What about you?

- In the next 10 minutes
  - Form a group of 3-4 students
    - Pick a Presenter and a Recorder
  - Intros and discuss the following
    - Name
    - Favorite food
    - What do you think of Software Engineering?
    - What do you know about Software Engineering?
    - What is your preferred programming language?

Outline

- Introduction to software engineering
- Why is software engineering important?
- Software development activities
- Software development challenges
- Essential attributes of good quality software

Software engineering

Meaning of Software and Engineering

- Software
  - Computer programs and associated documentation, developed for a particular customer or a general market.
- Engineering
  - Application of a systematic approach, based on science and mathematics, toward the production of a structure, machine, product, process or system.

Software Engineering Goals

- Produce a High “Quality” software system in a “Cost-effective” and “Timely” manner

Software engineering

Engineering discipline that is concerned with all aspects of software production from the early stages of system specification through to maintaining the system after it has gone into use.

Engineering discipline

- Using appropriate theories and methods to **solve problems** bearing in mind organizational and financial constraints.

All aspects of software production

- Technical process of development, project management and the development of tools, methods, etc to support **software production**.

Why is software engineering important?

Essential for the functioning of modern societies (government, national & international businesses and institutions)

Individual and societal reliance on advanced software systems is rapidly increasing. We need to be able to produce reliable and trustworthy systems economically and quickly.

It is usually cheaper, in the long run, to use software engineering methods and techniques for software systems. For most types of system, software maintenance (adapting to changes in hardware and requirements) incurs the highest cost than developing it.

Software Crisis

- 40-Year-Old Software Productivity Problem
  - Software has failed to keep up with hardware evolution.
  - No significant advances for the last few decades
  - Difficulty of writing correct, understandable, and verifiable software
- Software Crisis manifested in Several Ways
  - Projects running over-budget and over-time
  - Software with Inefficiency
  - Software with Low Quality
    - Usability, Performance, Maintainability, etc.
  - Software not Meeting requirements
  - Unmanageable Projects

Causes of Software Crisis

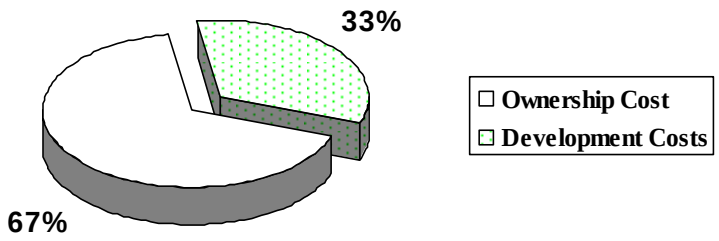
- Increased Software "Complexity"
  - User Demands on Richer Functionality of Software
  - Heterogeneous Hardware Environment
  - Distributed Computing
  - Web-based and Mobile Applications
  - Software embedded in Hardware Systems
  - Context-aware Computing
- Increased Software "Cost"
  - Development Cost
  - Ownership Cost
    - Operation Cost + Maintenance Cost

Software Complexity

- Building Large-scale Software
  - is *not* a simple scalability problem.
- Analogy
  - *Foot* bridge over stream vs. *Road* bridge over river
- Nature of the Problem
  - *Complexity* of the Software

Software Cost (1)

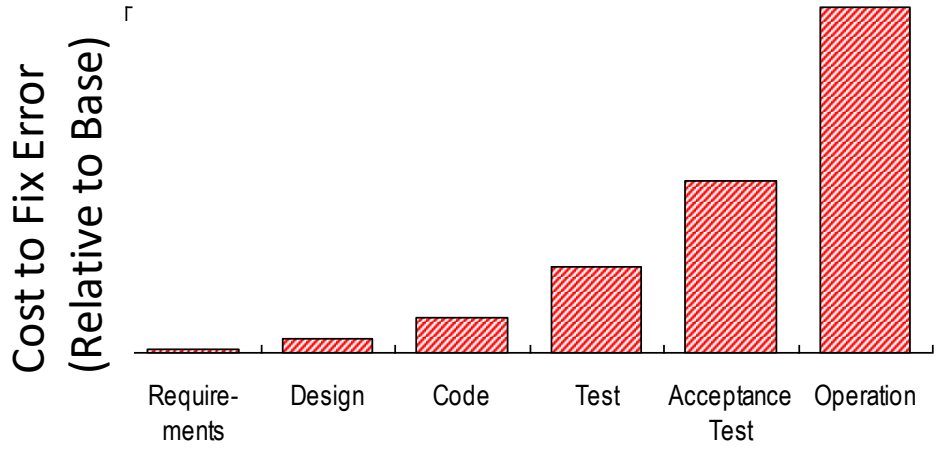
- Development Cost vs. Ownership Cost
  - Software ownership is generally twice as expensive as development.
    - Primarily, the cost of maintenance



- Message
  - Software product is not the final goal.
  - Maintenance becomes a significant issue.

Software Cost (2)

- Costs to Fix Errors
  - **The sooner an error is discovered, the better.**



  - More errors are found by outside testers and users than by developers.
  - More errors are found in the two latest stages.

Software engineering is more than writing computer programs

Problem solving

- Formulate the problem
- Analyze the problem
- Search for solutions
- Decide on the appropriate solution
- Specify the solution

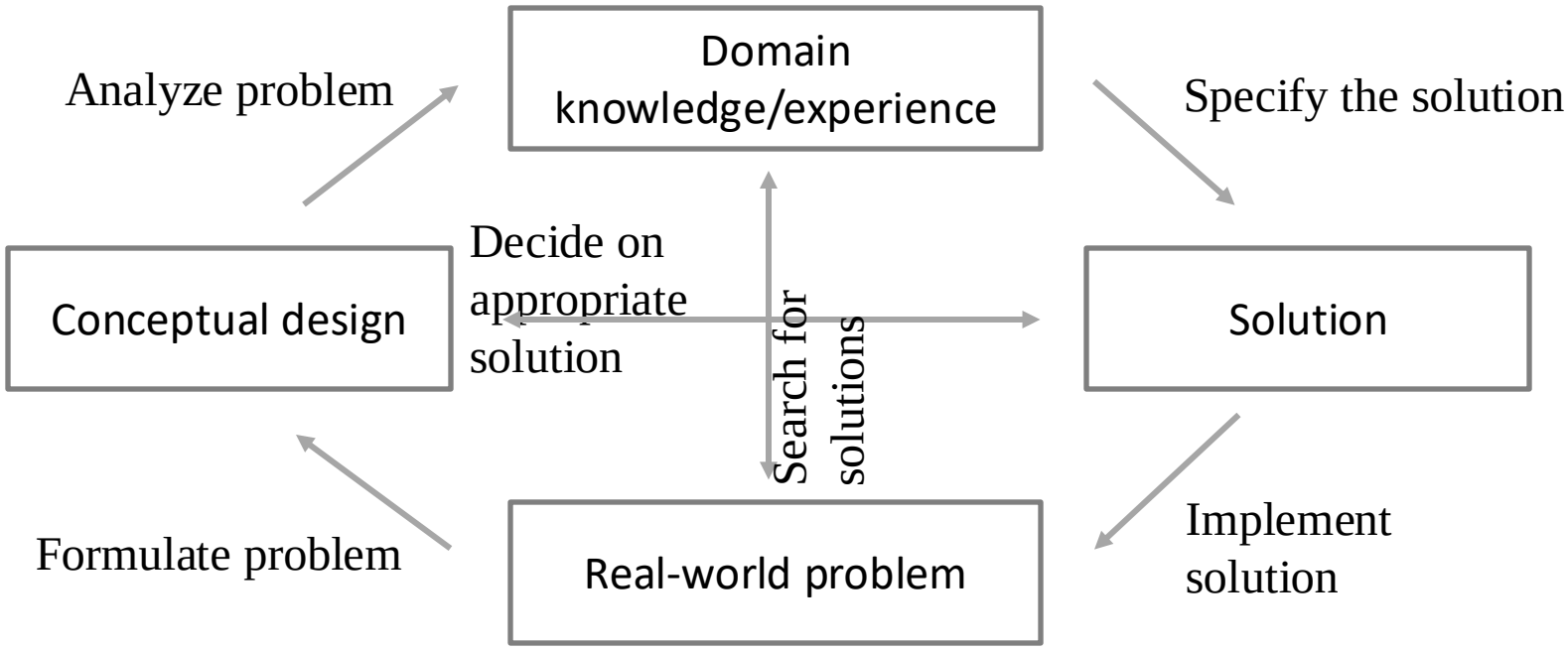
Modeling

- Identify and understand the nature of the problem/system
- Break it down into pieces (decomposition)

Knowledge acquisition

Software engineering – Problem Solving

Formulate the problem → Analyze the problem → Search for solutions → Decide on the appropriate solution → Specify the solution



Software engineering – Modeling activity

Modeling activity

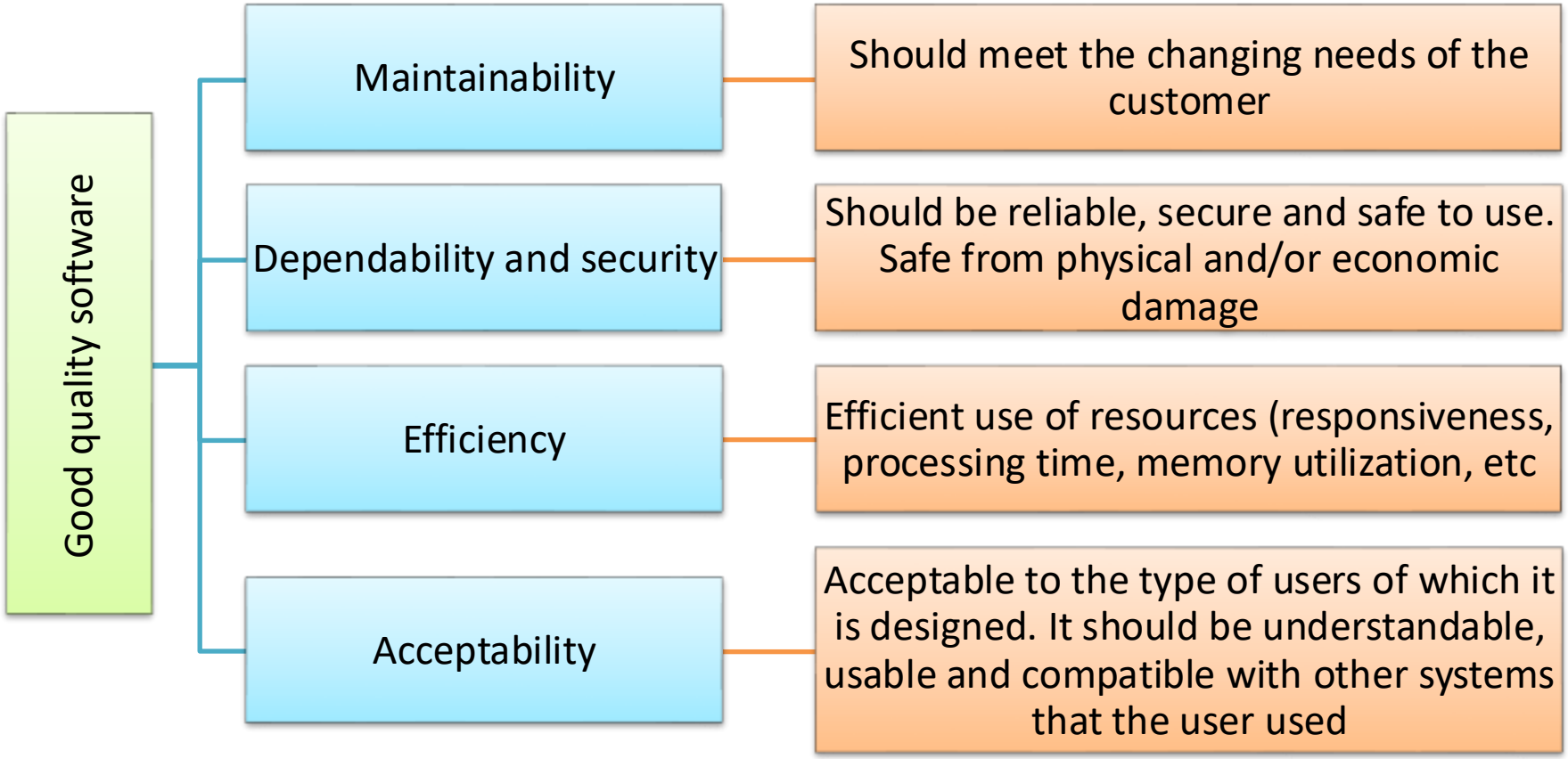
- Software can get complex and is driven by varying needs
- Systems typically consist of many small components with dependent/interrelated relationships
- Over time, the relationships may change (unpredictable) and new relationships may emerge
- Complexity can be dealt by decomposing the problem into smaller pieces and modeling their interactions (one at a time).
- Typically, multiple different models are created by the software engineers

Software engineering – Dealing with complexity

Software complexity

- Decompose larger system into smaller components
- Complexity can be reduced to make a system simpler

Essential attributes of good software

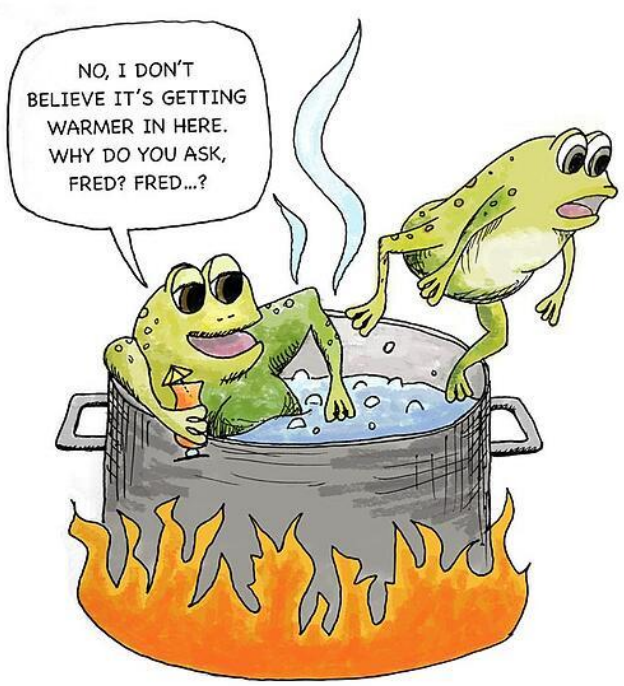


Requirements engineering is important

Omission or misinterpretation of user requirements leads to software disorders/dysfunction

Software rot – frogs in boiling water

- These disorders may not be like super hot water but tepid water which is slowly brought to boiling level



Summary

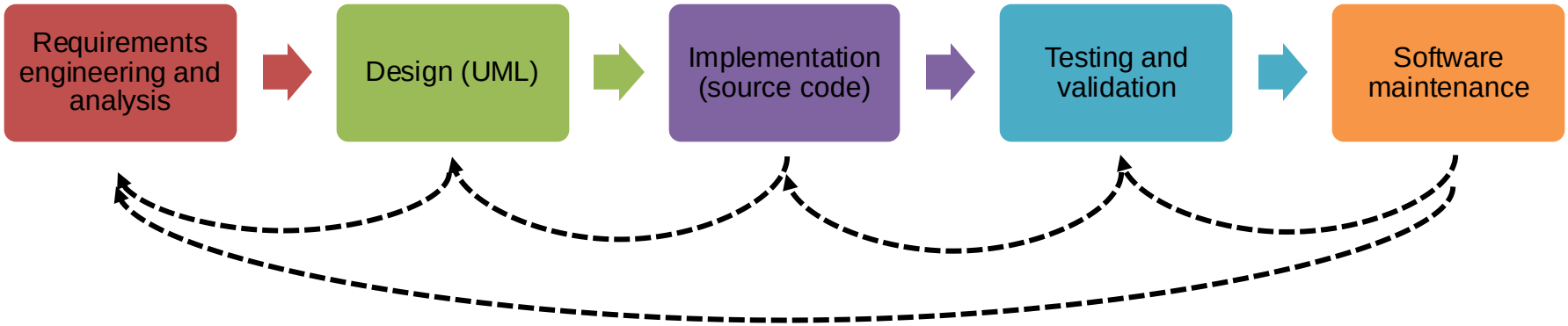
Question	Answer
What is software?	Computer programs and associated documentation. Software products may be developed for a particular customer or may be developed for a general market.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an engineering discipline that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software specification, software development, software validation and software evolution.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.

Outline

- Software engineering, life cycle process
- Principles and best practices of software development
- Software diversity
- Software success and failure stories

Software Development Lifecycle

Set of activities and their relationships to each other to support the development of a software system



Essence of software engineering practice

Prepare: Understand the problem (communication and analysis).

Think: Plan a solution (modeling and software design).

Execute/implement: Carry out the plan (code generation).

Verify/validate: Examine result for accuracy (testing & quality assurance).

Polya, G. (1945). *How to solve it; a new aspect of mathematical method.* Princeton University Press.

Understand the problem

Who has a stake in the solution to the problem?

- That is, who are the stakeholders?

What are the unknowns?

- What data, functions, and features are required to properly solve the problem?

Can the problem be compartmentalized?

- Is it possible to represent smaller problems that may be easier to understand?

Can the problem be represented graphically?

- Can an analysis model be created?

Plan a solution

Have you seen similar problems before?

- Are there patterns that are recognizable in a potential solution? Is there existing software that implements the data, functions, and features that are required?

Has a similar problem been solved?

- If so, are elements of the solution reusable?

Can subproblems be defined?

- If so, are solutions readily apparent for the subproblems?

Can you represent a solution in a manner that leads to effective implementation?

- Can a design model be created?

Carry out the plan

Does the solution conform to the plan?

- Is source code traceable to the design model?

Can we confirm each component is correctly implemented to help solve the problem?

- Has the design and code been reviewed, or better, have correctness proofs been applied to algorithm?

Examine the result

Is it possible to test each component part of the solution?

- Has a reasonable testing strategy been implemented?

Does the solution produce results, that conform to the data, functions, and features that are required?

- Has the software been validated against all stakeholder requirements?

Hooker's general principles

1. The Reason It All Exists – provide value to users. (Agile principle - team)
2. K I S S (Keep It Simple, Stupid!) – design simple as it can be.
3. Maintain the Vision – clear vision is essential.
4. What You Produce, Others Will Consume.
5. Be Open to the Future - do not design yourself into a corner.
6. Plan ahead for Reuse – reduces cost and increases value (next slide).
7. Think! – placing thought before action produce results.

Reusability

A good software design solves a specific problem but is general enough to address future problems (for example, changing requirements)

Experts do not solve every problem from first principles

- They reuse solutions that have worked for them in the past

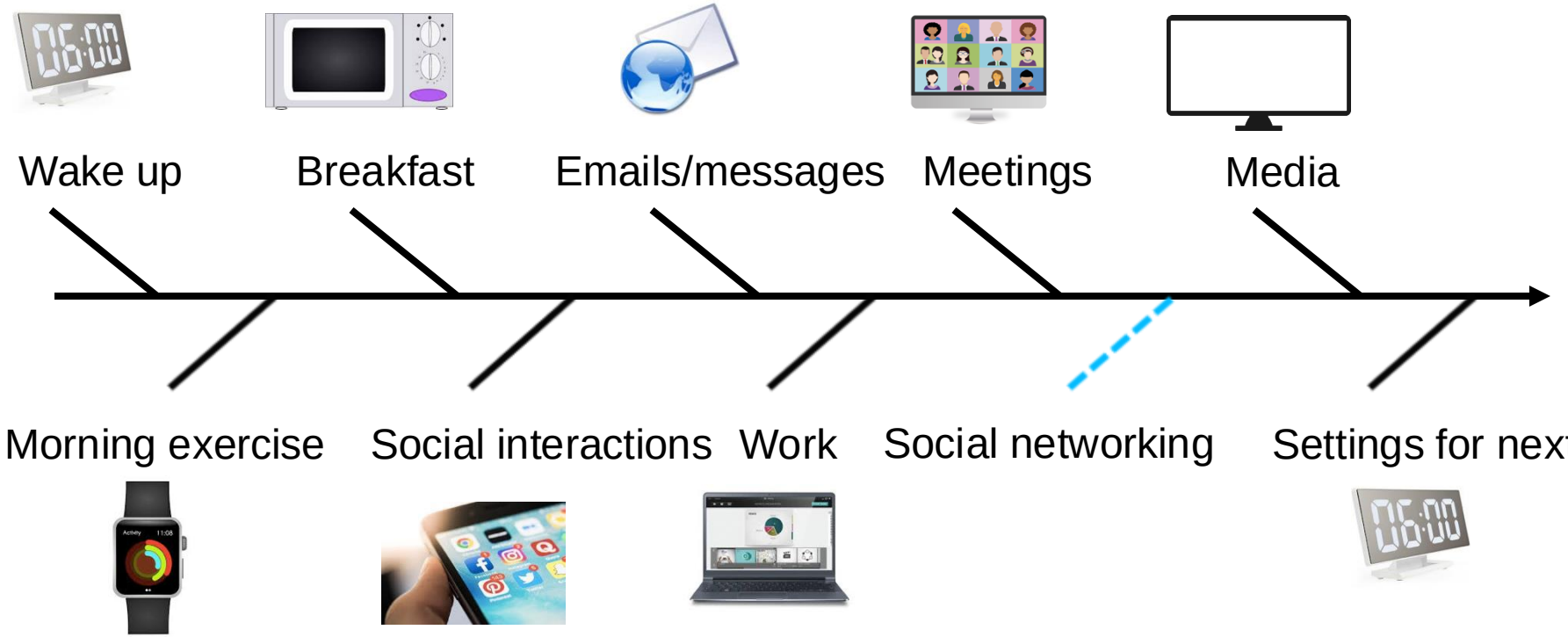
Goal for the software engineer:

- Design the software to be reusable across application domains and designs

How?

- Use design patterns and frameworks whenever possible

Software Diversity – Revisit the role of a software



Software Diversity – Types of software applications

Real time:	• air traffic control
Embedded systems:	• digital camera, GPS
Data processing:	• telephone billing, pensions
Information systems:	• web sites, digital libraries
Sensors:	• weather data
System software:	• operating systems, compilers
Communications:	• routers, mobile telephones
Offices:	• word processing, video conferences
Scientific:	• simulations, weather forecasting
Graphical:	• film making, design

Software failure stories

- Boeing 737 Max's two fatal crashes (346 deaths) - October 2018 & March 2019
- Uber software glitch – software notifications being pushed even after the sign out ($45 million lawsuit) - 2017
- Equifax social security hack - 2017
- Hawaii Emergency Management Agency: statewide false missile strike alarm – 2018
- Crowdstrike software update causing windows OS to crash - 2024

Software success stories

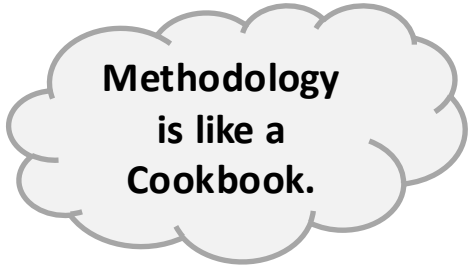
- Instagram: took 8 weeks to complete by two developers, > $100 billion net worth
- Tumblr blogging platform: 2.2 billion-page views per week
- Amazon Web Services: lead cloud computing platform, ~$500 billion net worth
- Microsoft Azure: $36 billion revenue in Q2 2020
- Zoom video conferencing: $72 billion net worth

… and many more

CPSC 362
Software Engineering

Process in Software Engineering

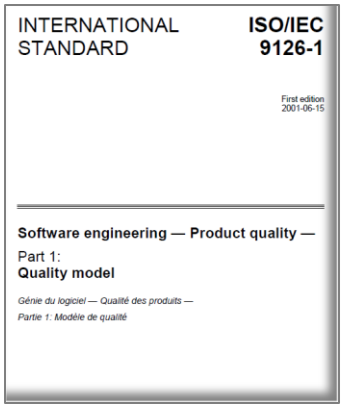
- Definition
  - A methodology consists of a process, a vocabulary, and guidelines.
  - A process defines a set of activities that together accomplish all the goals of the methodology.
- Key Elements of a Process
  - Life-Cycle Model
    - Phases, Activities, Steps
  - For each Activity
    - Input, Output Artifacts, Instruction, Quality Guidelines



Why is the Process essential?

- ISO/IEC 9126
  - Revision in 2001
    - ISO/IEC 9126-1 to 9126-4
  - Replaced by ISO/IEC 25010 in 2011
    - ‘Security’ and ‘Compatibility’ were added

- Part 1
  - Quality Model
- Part 2
  - External Metrics
- Part 3
  - Internal Metrics
- Part 4
  - Quality In Use Metrics


INTERNATIONAL STANDARD

ISO/IEC 9126-1

First edition 2001-06-15

Software engineering — Product quality —

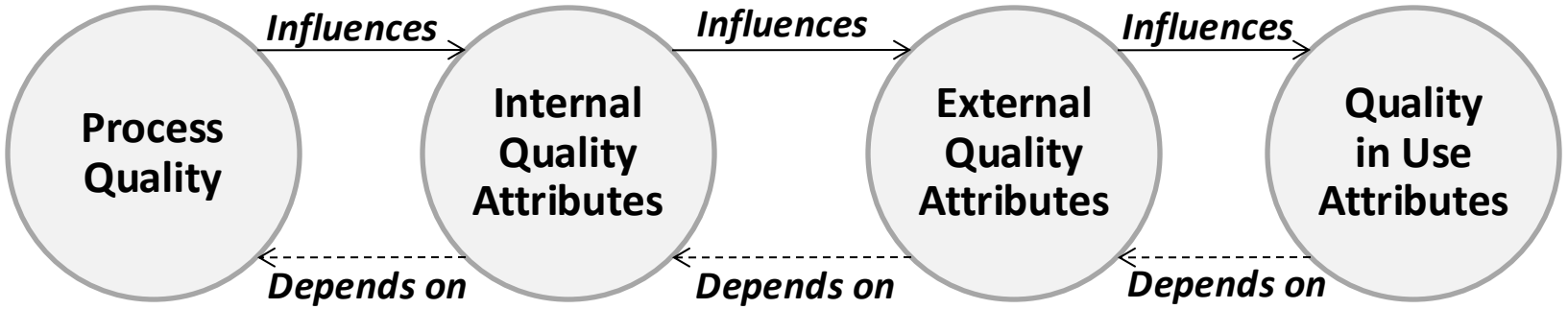
Part 1:
Quality model

Génie du logiciel — Qualité des produits —

Partie 1: Modèle de qualité

Quality Model Framework

- Quality in the Lifecycle



Process *Software Product* *Effect of SW Product*

Quality Model Framework

- **Process** Quality
  - Quality of Life-cycle Process
  - Process quality contributes to improving product quality.
- **Product** Quality
  - Can be evaluated by measuring internal attributes or measuring external attributes.
  - Internal quality
    - is evaluated by the static measure of intermediate products.
    - View at Technical Level
  - External quality
    - is evaluated by measuring the behavior of the code when executed.
    - View of User/Management
  - Product quality contributes to improving *quality in use*.

Quality Model Framework

- Quality In Use
  - User's view of the quality of an environment containing software and is measured from the results of using the software in the environment.
    - Rather than properties of the software itself.
  - User's environment may be different from development environment.

Representative Process Models

- Waterfall Model
- Spiral Model
- Prototyping Model
- Component-Based Development (CBD) Model
- Agile Process
  - eXtreme Programming (XP)
  - Scrum

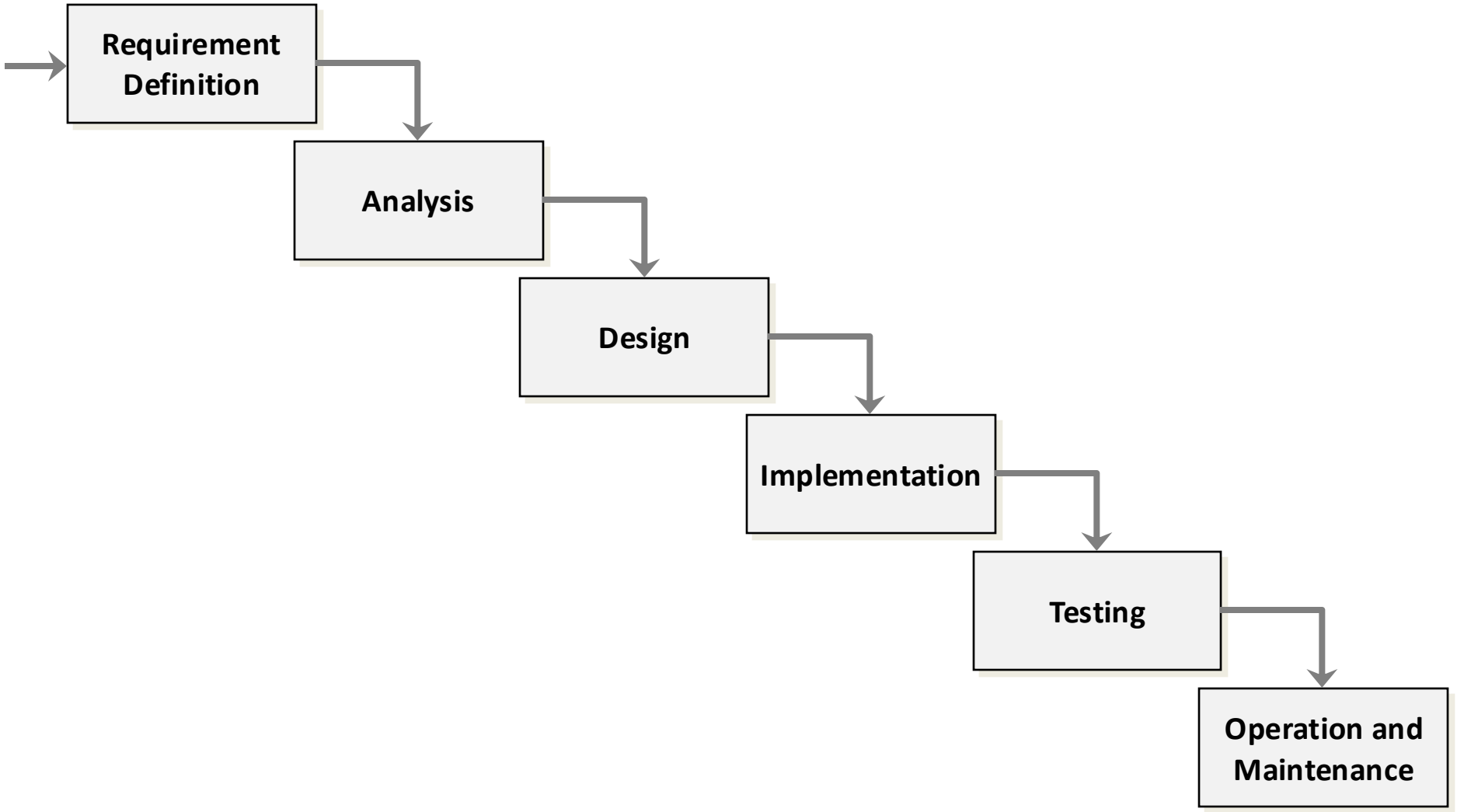
…

Waterfall Model

Waterfall Model (1)

- Oldest paradigm in Software Engineering
- Sequential approach to S/W development
- When to Choose?
  - When requirements of a problem are reasonably well understood.
  - When work flows from communication through deployment in a reasonably linear fashion.
  - When requirements are well-defined and reasonably stable.

Waterfall Model (2)



Waterfall Model (3)

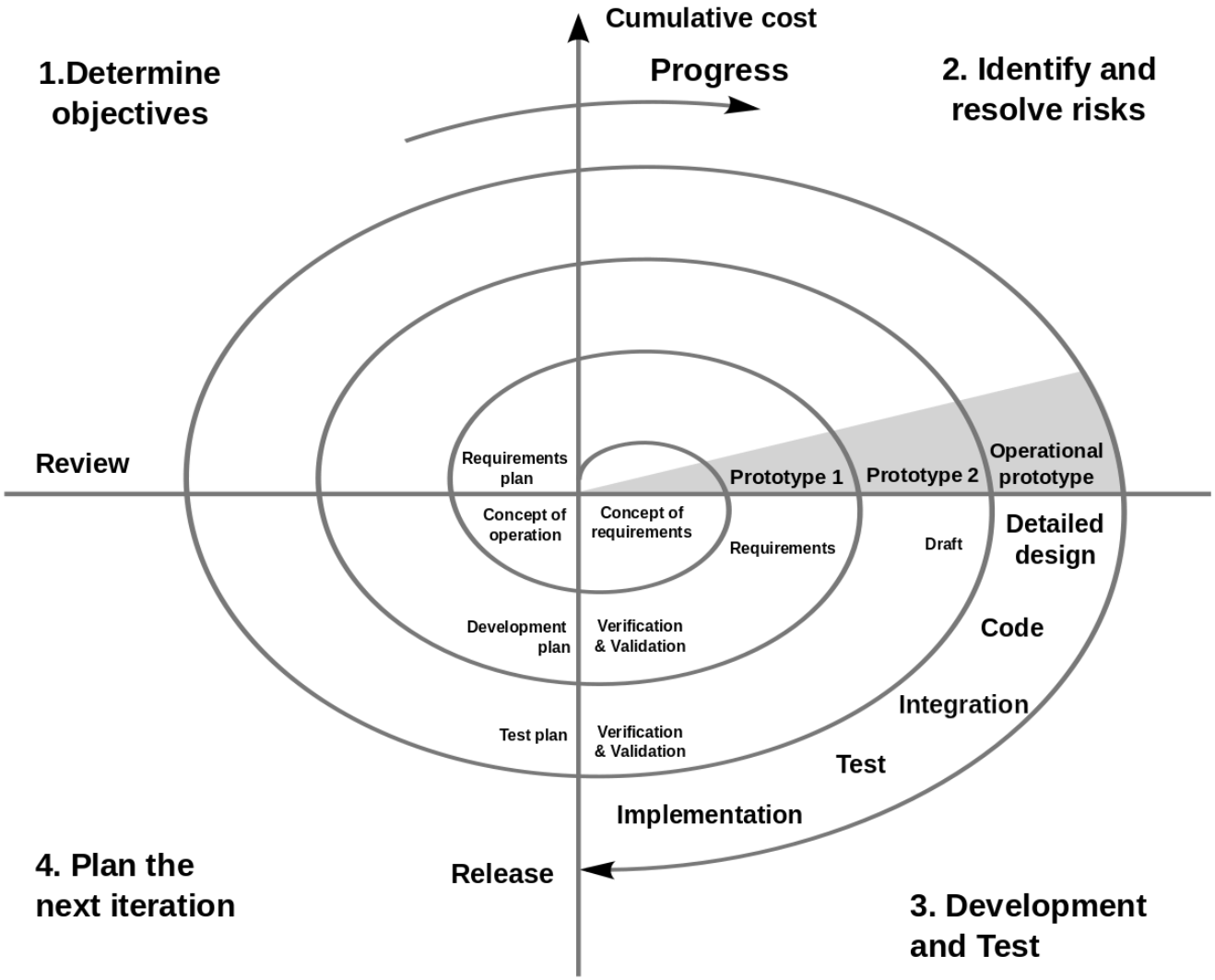
- Problems
  - Real projects rarely follow the sequential flow that the model proposes.
  - It is often difficult to for the customer to state all requirements explicitly.
  - The customer must have patience.
    - A working version of the program will not be available until late in the project time-span.

Spiral Model

Spiral Model (1)

- Adapted to apply throughout the entire life cycle of an application, from concept development and maintenance.
- Couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.
- Software is developed in a series of incremental releases.
  - Early iterations
    - Release might be a paper model prototype.
  - Later iterations
    - More complete versions of the engineered system are produced.

Spiral Model (2)



Spiral Model (3)

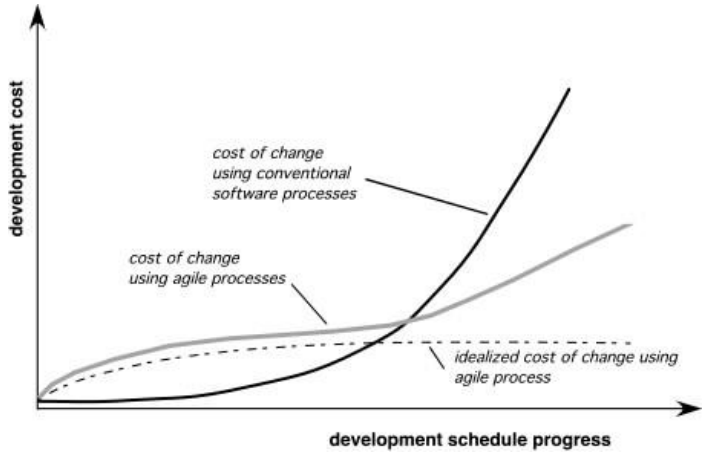
- Advantages
  - Effectively Gathering and Refining Requirements
  - Risk Reduction through Clients' Feedback
  - Realistic Approach for Large-scaled systems
- Drawbacks
  - Hard to Control
    - Iterations and Increments
    - Progress Monitoring

Agile Process

Agile Process (1)

- What is “Agility”?
  - Effective (rapid and adaptive) response to change
  - Effective communication among all stakeholders
  - Drawing the customer onto the team
  - Organizing a team so that it is in control of the work performed

- Agility and Cost of Change



Yielding rapid, incremental delivery of software

Agile Process (2)

- A group of software development methods based on iterative and incremental development, where requirements and solutions evolve through collaboration between self-organizing, cross-functional teams [Wiki]
- To promote;
  - Adaptive Planning
  - Evolutionary development and delivery
  - A time-boxed iterative approach,
  - Rapid and flexible response to change
- Introduced by Agile Manifesto in 2001

12 Agile Principles (1)

- Principle 1. Customer satisfaction by rapid delivery of useful software
  - The highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Principle 2. Welcome changing requirements, even late in development
  - Agile processes harness change for the customer's competitive advantage.
  - Agile teams work to keep the software structure flexible, so requirement change impact is minimal.
- Principle 3. Working software is delivered frequently (weeks rather than months)
  - From a couple of weeks to a couple of months with a preference to the shorter time scale

12 Agile Principles (2)

- Principle 4. Working software is the principal measure of progress
  - Agile projects measure their progress by measuring the amount of working software.
- Principle 5. Sustainable development, able to maintain a constant pace
  - Agile Processes promote sustainable development.
    - The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Principle 6. Close, daily cooperation between people in business and developers
  - For agile projects, there must be significant and frequent interaction between the
    - customers
    - developers
    - stakeholders

12 Agile Principles (3)

- Principle 7. Face-to-face conversation is the best form of communication (co-location)
  - In agile projects, developers talk to each other.
    - The primary mode of communication is conversation.
    - Documents may be created, but there is no attempt to capture all project information in writing.
- Principle 8. Projects are built around motivated individuals, who should be trusted
  - Give them the environment and support they need, and trust them to get the job done.
- Principle 9. Continuous attention to technical excellence and good design
  - The way to go fast is to keep the software as clean and robust as possible.
  - Thus, all agile team-members are committed to producing only the highest quality code they can.

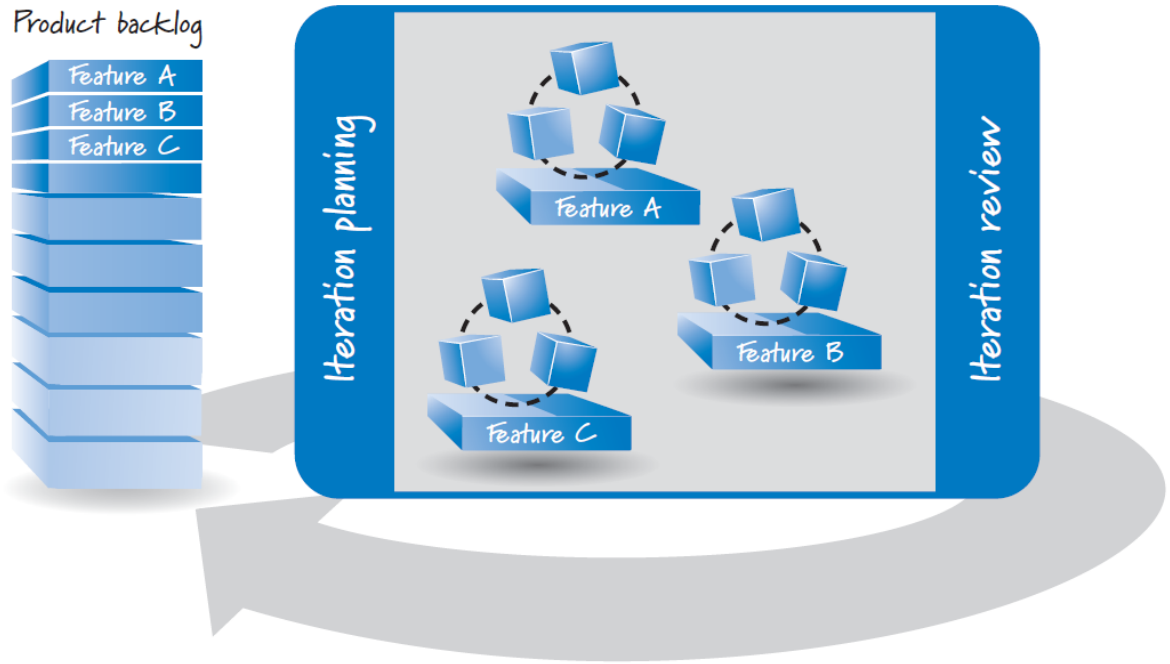
12 Agile Principles (4)

- Principle 10. Simplicity is essential
  - Agile teams take the simplest path that is consistent with their goals.
- Principle 11. Self-organizing teams
  - The best architectures, requirements, and designs emerge from Self-Organizing Teams.
    - Responsibilities are not handed to individual team members from the outside.
    - Responsibilities are communicated to the team as a whole, and the team determines the best way to fulfill them.
- Principle 12. Regular adaptation to changing circumstances
  - At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.
  - An agile team knows that its environment is continuously changing, and knows that they must change with that environment to remain agile.

Agile Process → Scrum

Overview on Scrum (1)

- Agile approach for developing innovative products and services



Overview on Scrum (2)

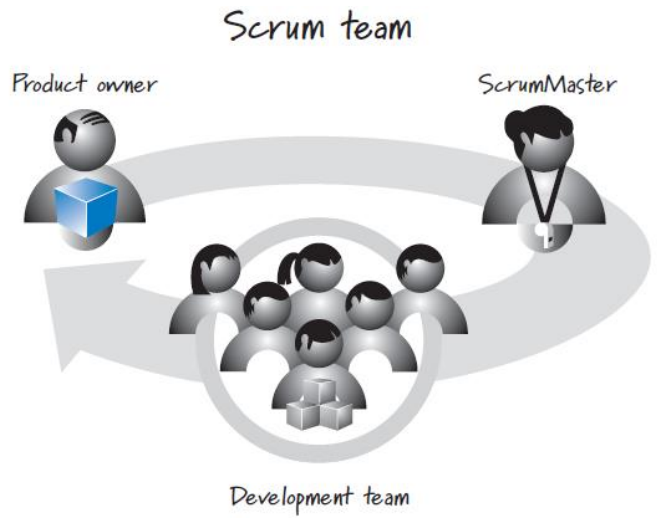
- To begin by creating a product backlog
  - A prioritized list of the product backlog items (PBIs).
- To work on the most important or highest priority items first, guided by the product backlog
- To perform each work in short, time-boxed iterations
  - Ranging from a week to a calendar month in length
  - By a self-organizing, cross-functional team
  - Covering the much greater amount of work than can be completed by a team in one short-duration iteration

Overview on Scrum (3)

- At the start of each iteration
  - To plan which high-priority subset of the product backlog to create in the upcoming iteration
- At the end of the iteration
  - To produce a potentially shippable product or increment of the product
  - To review the completed features with the stakeholders to get their feedback
  - Based on the feedback, to alter both what they plan to work on next and how the team plans to do the work

Scrum Roles

- Scrum Team
  - Consists of one or more members.
- In each Scrum Team
  - To make up of essential three scrum roles;
    - Product Owner
    - ScrumMaster
    - Development Team
  - Can include other roles



Scrum Roles – Product Owner

- The single authority responsible for deciding what features and functionalities will be developed and in what order
- To maintain and communicate to all other participants a clear vision of what the Scrum team is trying to achieve
  - To actively collaborate with the ScrumMaster and development team
  - Must be available to answer questions soon after other participants are posed
- Responsible for the overall success of the solution being developed or maintained

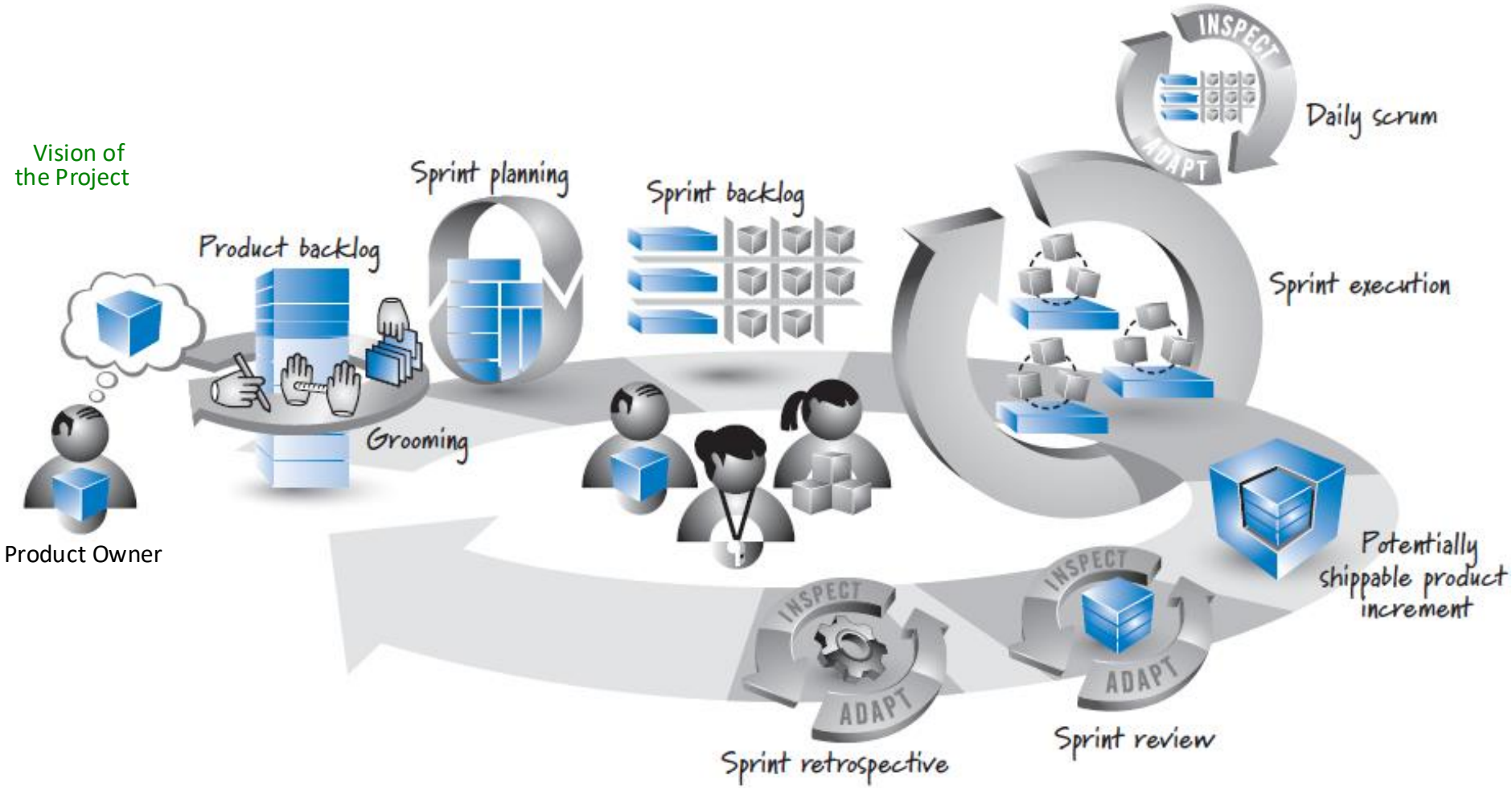
Scrum Roles – ScrumMaster

- Responsible for guiding the team in creating and following its own process based on the broader Scrum framework
  - To help everyone involved understand and embrace the Scrum values, principles, and practices
- To act as a coach
  - To help the Scrum team and the rest of the organization develop their own high performance, organization-specific Scrum approach
- To act as a facilitator
  - To help the team resolve issues and make improvements to its use of Scrum
  - To protect the team from outside interference
- To act as a leader
  - To take a leadership role in removing impediments that inhibit team productivity

Scrum Roles – Development Team

- Responsible for determining how to deliver what the product owner has asked for
- To self-organize to determine the best way to accomplish the goal set out by the product owner
- Typically five to nine people in size
  - Its members must collectively have all of the skills needed to produce good quality, working software.

Scrum Activities and Artifacts (1)



Scrum Activities and Artifacts (2)

- To perform grooming
  - Done by a product owner
  - To break a project vision of the product owner down into a set of features that are collected into a prioritized list (i.e. product backlog)
  - Artifact
    - A product backlog
- To perform sprint planning
  - Mainly done by a development team
  - To determine a subset of the product backlog items it believes it can complete
  - Artifact
    - A sprint backlog
      - Which describes how the team plans to design, build, integrate, and test the selected subset of features from the product backlog during that particular sprint

Scrum Activities and Artifacts (3)

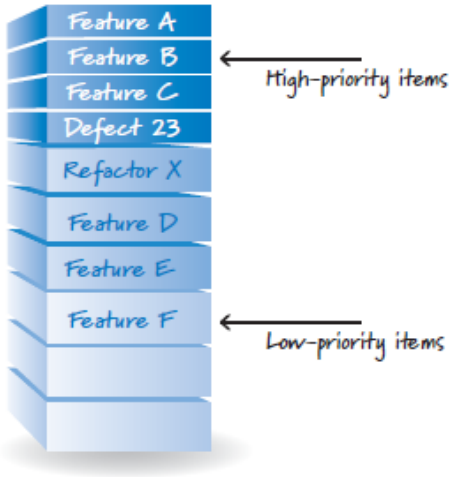
- To perform the sprint execution
  - Done by the development team
  - To perform the tasks necessary to realize the selected features
  - To perform the daily scrum everyday
    - The team members help manage the flow of work by conducting a synchronization, inspection, and adaptive planning activity.
  - Artifact
    - A potentially shippable product increment (PSPI)
      - Which represents some, but not all, of the product owner's vision, at the end of sprint execution

Scrum Activities and Artifacts (4)

- To perform two inspect-and-adapt activities
  - The sprint review
    - The stakeholders and Scrum team inspect the product being built.
  - The sprint retrospective
    - The Scrum team inspects the Scrum process being used to create the product.
  - Artifact
    - Adaptations that will make their way into the product backlog or be included as part of the team's development process

Product Backlog

- A prioritized list of the project work, i.e. Product Backlog Items.
- PBIs are prioritized.
  - Higher-value items appear at the top of the product backlog.
  - Lower-value items appear toward the bottom.
- Initially generated by the product owner
  - With input from the team and stakeholders
- Can be evolved, i.e. Grooming
  - Items can be added, deleted, and revised at the end of each iteration.
  - Mainly due to (1) change on requirement change and (2) team's growing understanding on the product.

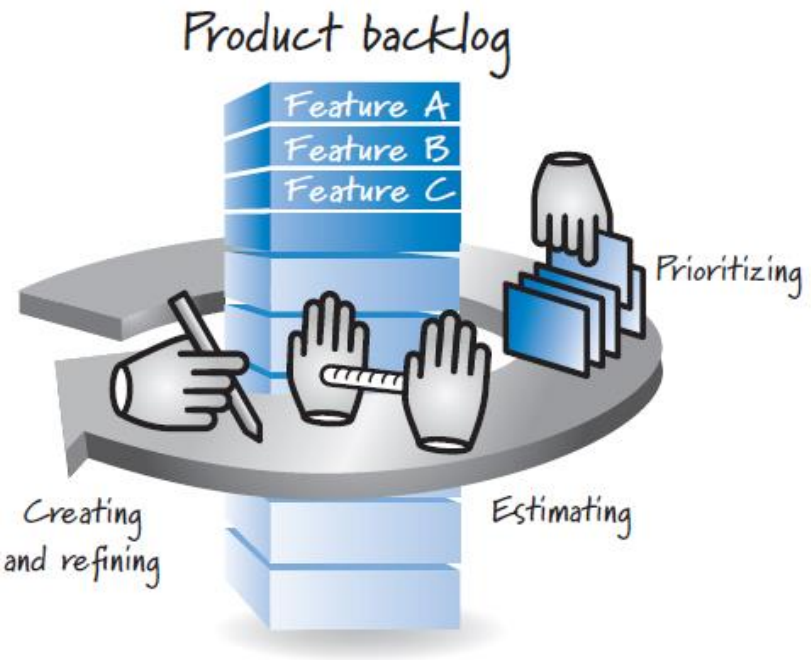


Types of Product Backlog Items

- Feature Type
  - Represents a functional unit to be developed.
  - Example) Develop 'Rental' Component.
- Change Type
  - Represents a modification work on an existing artifact, mainly due to the change on requirement or business process.
  - Example) Modify the 'Check out car' procedure.
- Defect Type
  - Represents a defect removal work on an existing artifact.
  - Example) Fix the defect on *Rental*; computing the fee for a late return.
- Technical Improvement
  - Represents an improvement work on an existing artifact.
  - Example) Upgrade the DMBS with a latest version.
- Knowledge Acquisition Type
  - Represents a knowledge acquisition work needed in the project.
  - Example) Acquire algorithms for multi-feature machine learning.

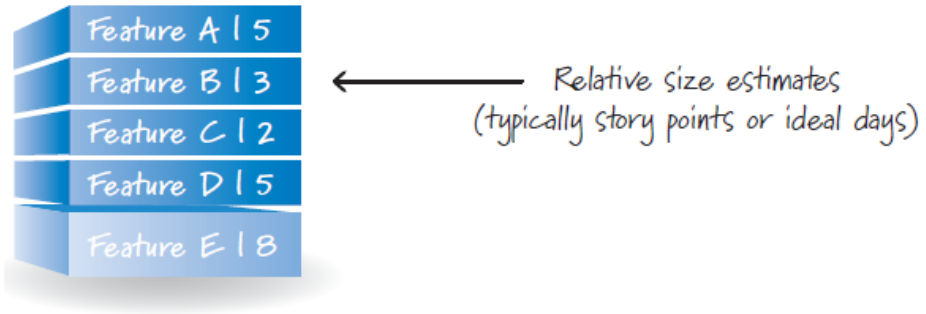
Grooming Product Backlog

- The activity of creating and refining PBIs, estimating them, and prioritizing them
  - Creating and Refining
  - Estimating
  - Prioritizing



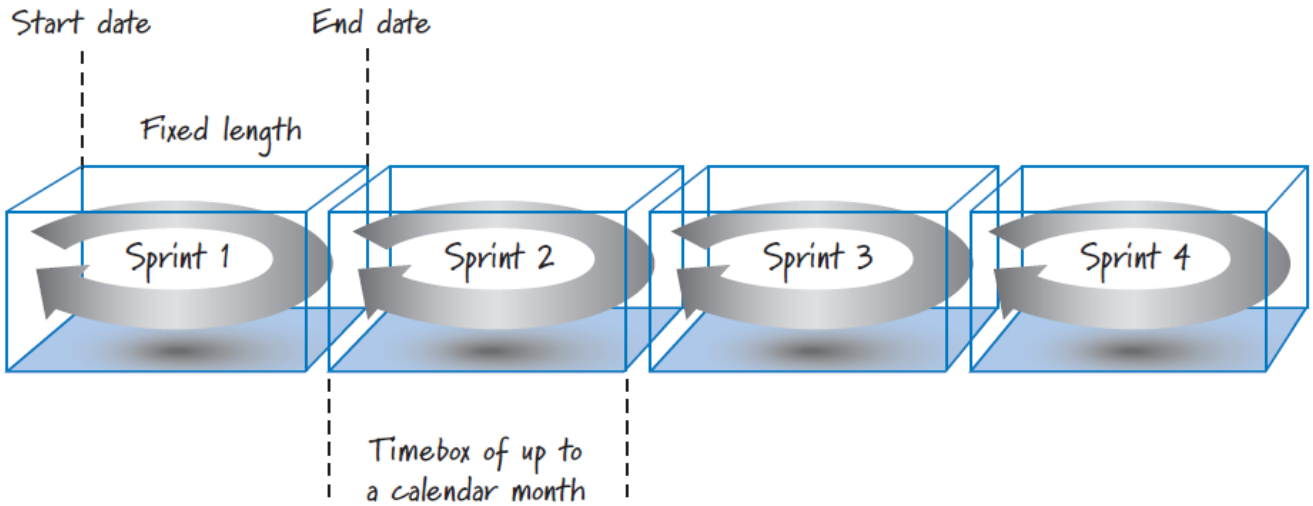
Estimating Cost of PBIs

- Estimating the size/cost of each PBIs
  - Used to properly determine its priority.
  - Before finalizing prioritizing, ordering, or otherwise arranging the product backlog
  - Which size measures to use
    - Relative Size Estimates are typically used



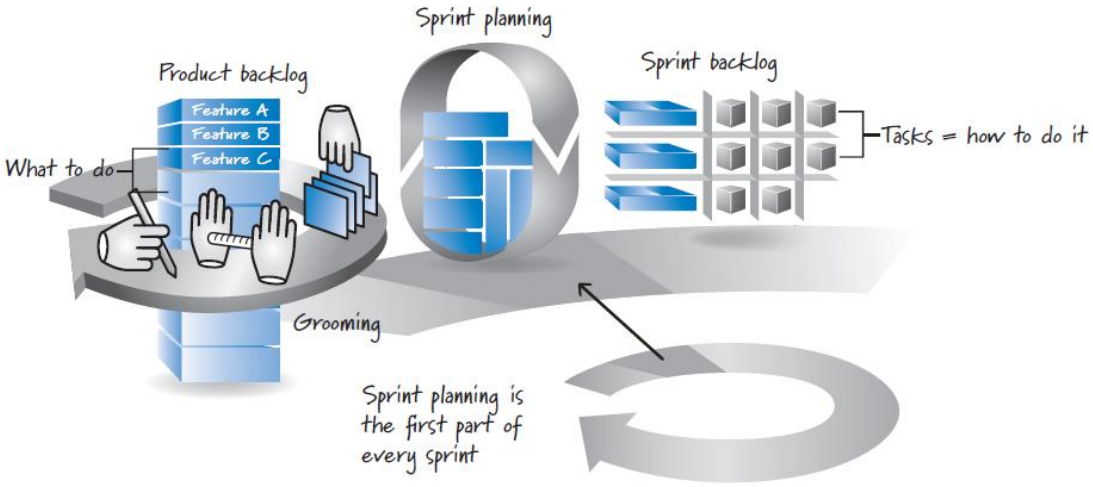
Sprints

- Iterations or cycles of up to a calendar month
- Time-boxed
  - To have a fixed start and end date
  - Generally, to have the same duration



Activity – Sprint Planning (1)

- To determine the most important subset of product backlog items to build in the next sprint
- To spend about four to eight hours in sprint planning
  - For two-week to one-month sprint

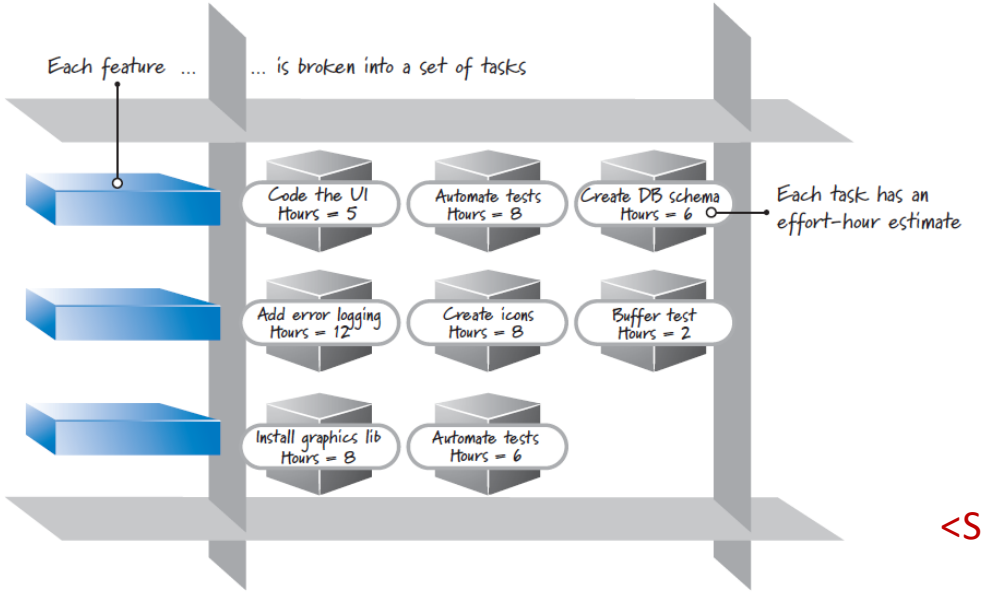


Activity – Sprint Planning (2)

- How to Perform
  - To agree on a sprint goal that defines what the upcoming sprint is supposed to achieve
    - Done by the product owner and development team
  - To review the product backlog and determine the high priority items
    - That the team can realistically accomplish in the upcoming sprint while working at a sustainable pace
    - Done by the development team
- Outcome is the Sprint backlog
  - Digital tracking systems like JIRA, Trello, etc

Activity – Sprint Planning (3)

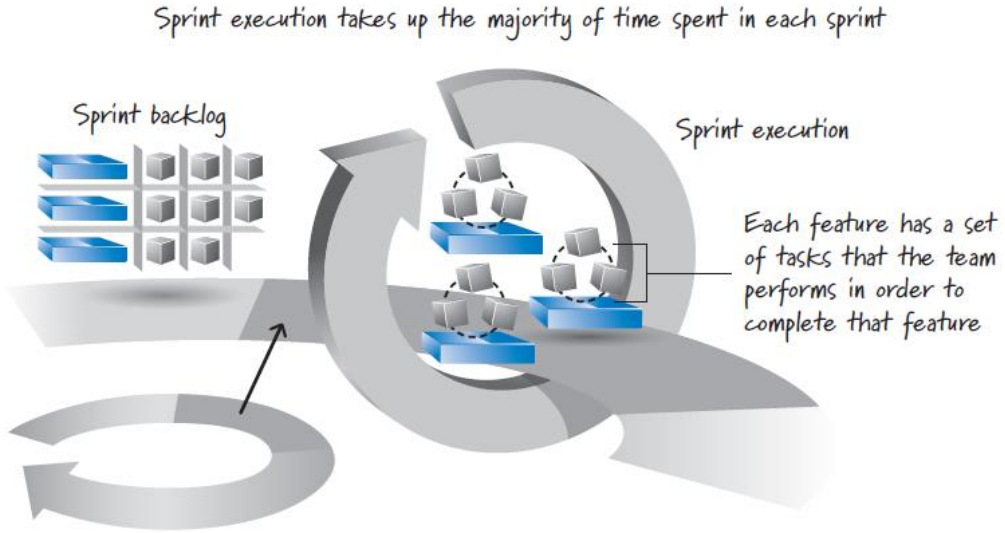
- To break down each targeted feature into a set of tasks, i.e. sprint backlog
- To provide an estimate (typically in hours) of the effort required to complete each task



<Sprint Backlog>

Activity – Sprint Execution (1)

- To perform all of the task-level work necessary to get the features done
  - Done by the development team
  - Guided by the ScrumMaster's coaching



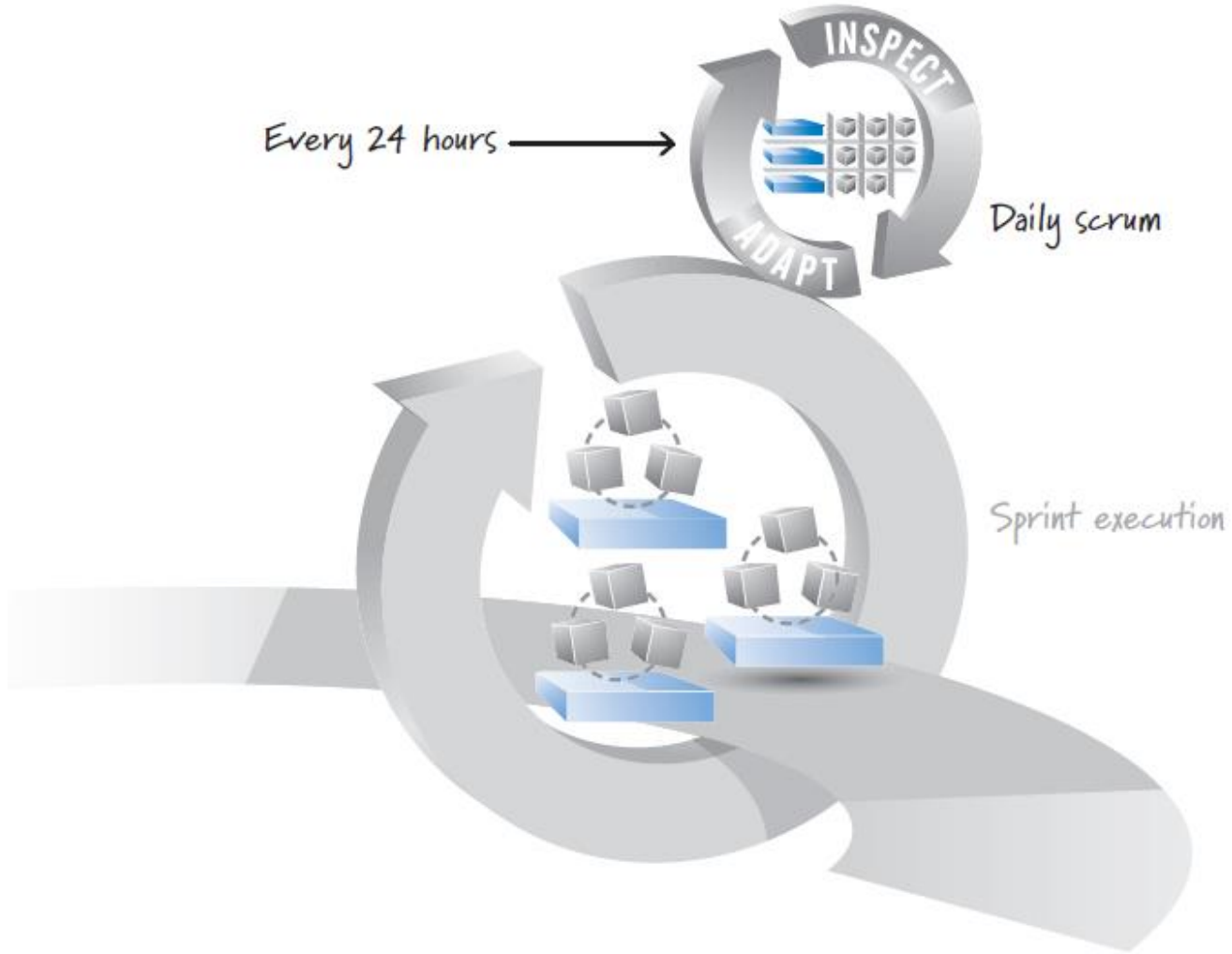
Activity – Sprint Execution (2)

- To define team's task-level works by themselves
- To self-organize in any manner they feel is best for achieving the sprint goal

Activity – Daily Scrum (1)

- To hold a time-boxed (15 minutes or less) inspect-and adapt activity, each day of the sprint
  - An inspection, synchronization, and adaptive daily planning activity that helps a self-organizing team do its job better
- Referred as the daily stand-up
  - Everyone stands up during the meeting to help promote brevity.

Activity – Daily Scrum (2)

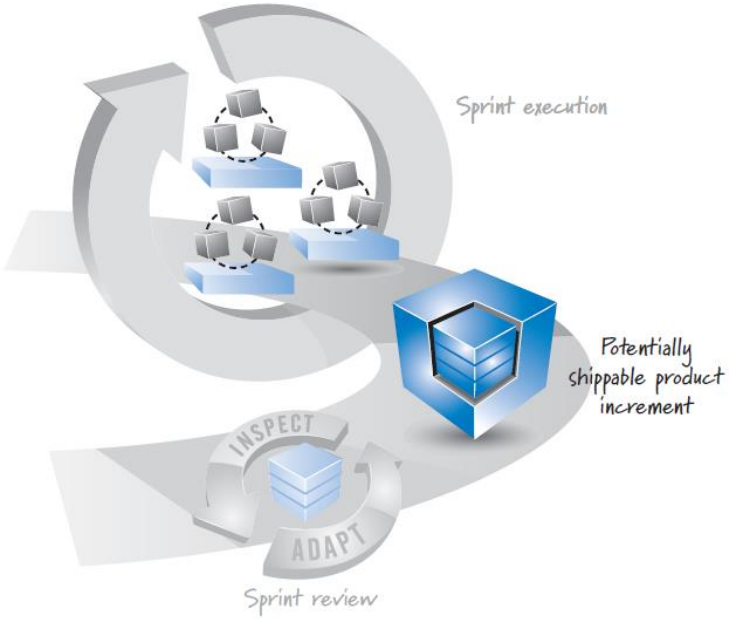


Activity – Daily Scrum (3)

- A Common Approach to Performing the Daily Scrum
  - To have the ScrumMaster facilitating
  - To have each team member taking turns answering three questions
    - What did I accomplish since the last daily scrum?
    - What do I plan to work on by the next daily scrum?
    - What are the obstacles or impediments that are preventing me from making progress?
- Benefits
  - Essential for helping the development team manage the fast, flexible flow of work within a sprint
  - Useful to communicate the status of sprint backlog items among the development team members

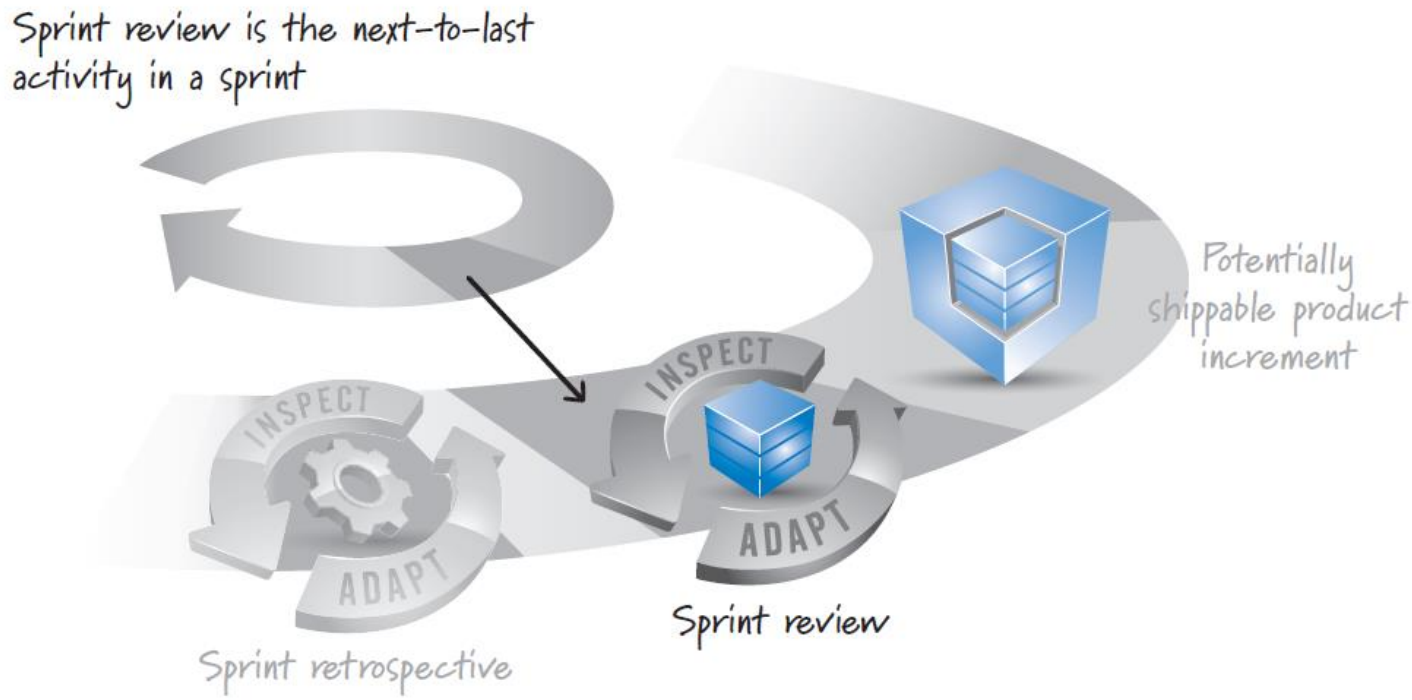
Activity – Done

- To produce potentially shippable product increment
  - Whatever the Scrum team agreed to do is really done according to its agreed-upon definition of done.
  - There isn't materially important undone work that needs to be completed.



Activity – Sprint Review (1)

- To inspect and adapt the product that is being built

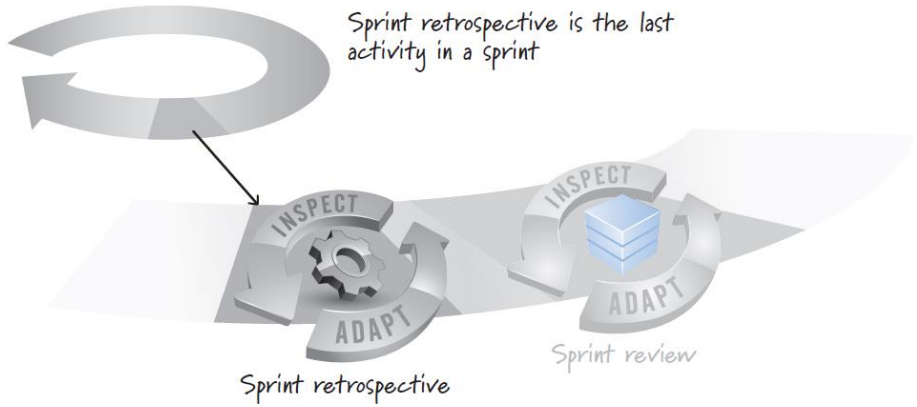


Activity – Sprint Review (2)

- To take place a conversation among its participants
  - Including the Scrum team, stakeholders, sponsors, customers, and interested members of other teams
  - To review the just-completed features in the context of the overall development effort
- Objectives
  - To get clear visibility into what is occurring
  - To have an opportunity to help guide the forthcoming development to ensure that the most business-appropriate solution is created

Activity – Sprint Retrospective

- To inspect and adapt the process
  - The development team, ScrumMaster, and product owner discuss what is and is not working with Scrum and associated technical practices.
- To focus on the continuous process improvement necessary to help a good Scrum team become great



CPSC 362
Software Engineering

Outline

- Requirement Engineering
- Functional and non-functional requirements
- SWOT Analysis

Requirements engineering

- The process of establishing the services that a customer requires from a system and the constraints under which it operates and is developed.
- The system requirements are the descriptions of the system services and constraints that are generated during the requirements engineering process.

What is a requirement?

- It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification.
- This is inevitable as requirements may serve a dual function
  - May be the basis for a bid for a contract - therefore must be open to interpretation;
  - May be the basis for the contract itself - therefore must be defined in detail;
  - Both these statements may be called requirements.

Types of requirement

- User requirements
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints. Written for customers.
- System requirements
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints. Defines what should be implemented so may be part of a contract between client and contractor.

Functional and non-functional requirements

Functional and non-functional requirements

- Functional requirements
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.
- Non-functional requirements
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.
- Domain requirements
  - Constraints on the system from the domain of operation

Functional requirements

- Describe functionality or system services.
- Depend on the type of software, expected users and the type of system where the software is used.
- Functional user requirements may be high-level statements of what the system should do.
- Functional system requirements should describe the system services in detail.

Requirements imprecision

- Problems arise when functional requirements are not precisely stated.
- Ambiguous requirements may be interpreted in different ways by developers and users.
- Consider the term ‘search’ in requirement 1
  - User intention – search for a patient name across all appointments in all clinics;
  - Developer interpretation – search for a patient name in an individual clinic. User chooses clinic then search.

Requirements completeness and consistency

- In principle, requirements should be both complete and consistent.
- Complete
  - They should include descriptions of all facilities required.
- Consistent
  - There should be no conflicts or contradictions in the descriptions of the system facilities.
- In practice, because of system and environmental complexity, it is impossible to produce a complete and consistent requirements document.

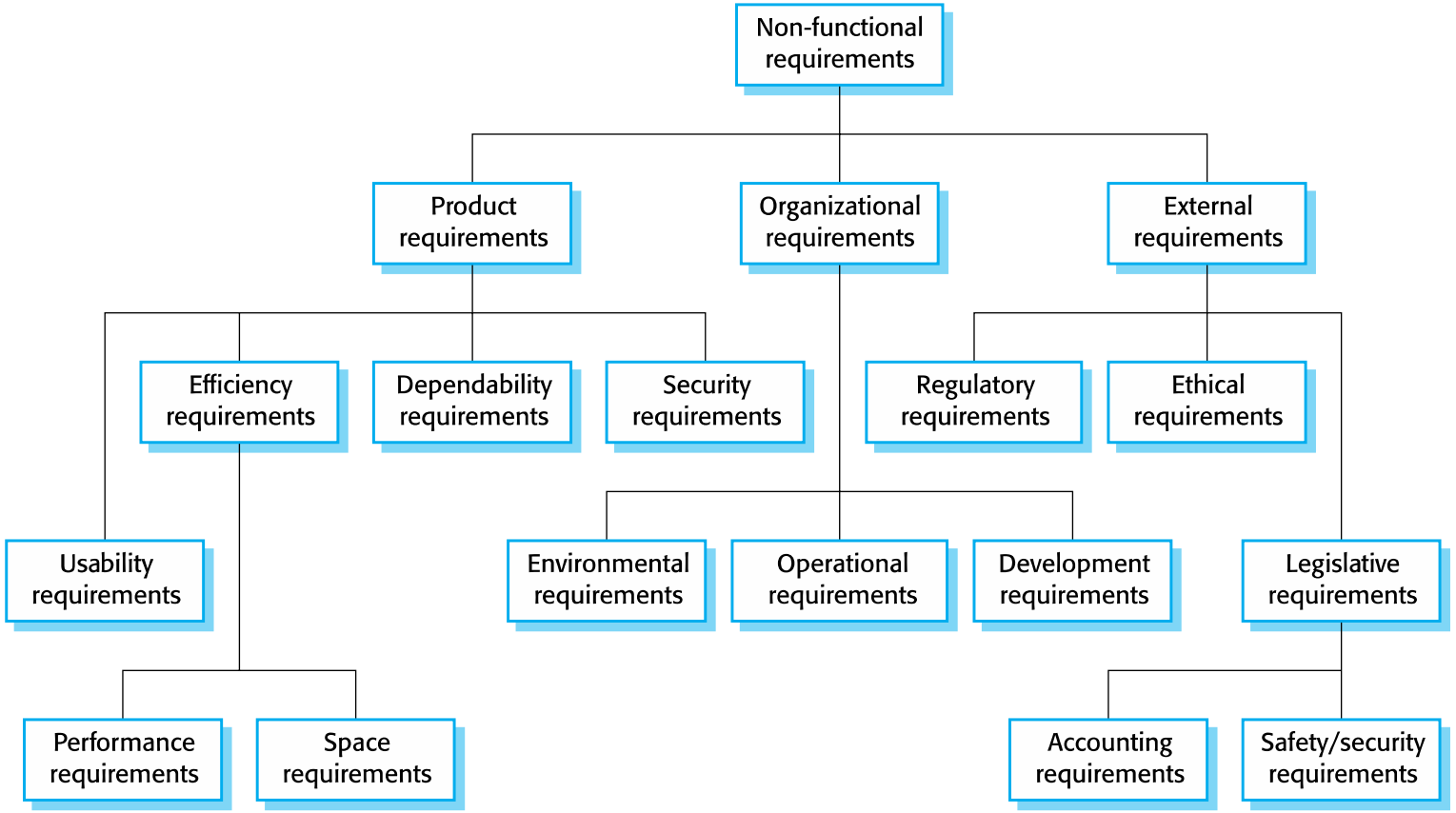
Non-functional requirements

- These define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
- Process requirements may also be specified mandating a particular IDE, programming language or development method.
- Non-functional requirements may be more critical than functional requirements. If these are not met, the system may be useless.

Non-functional requirements implementation

- Non-functional requirements may affect the overall architecture of a system rather than the individual components.
  - For example, to ensure that performance requirements are met, you may have to organize the system to minimize communications between components.
- A single non-functional requirement, such as a security requirement, may generate a number of related functional requirements that define system services that are required.
  - It may also generate requirements that restrict existing requirements.

Types of nonfunctional requirement



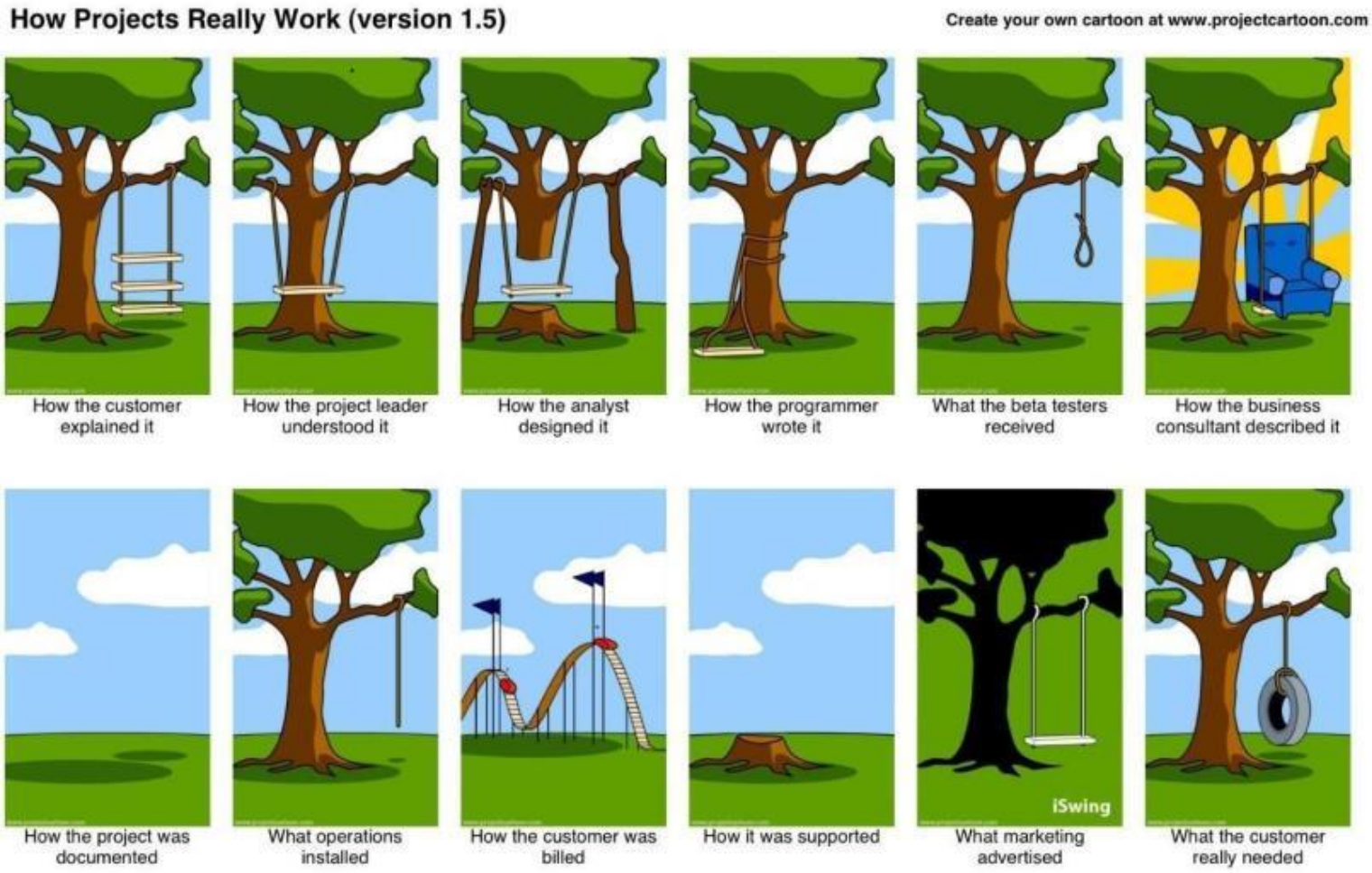
Non-functional classifications

- Product requirements
  - Requirements which specify that the delivered product must behave in a particular way e.g. execution speed, reliability, etc.
- Organizational requirements
  - Requirements which are a consequence of organisational policies and procedures e.g. process standards used, implementation requirements, etc.
- External requirements
  - Requirements which arise from factors which are external to the system and its development process e.g. interoperability requirements, legislative requirements, etc.

Requirements engineering processes

Requirements engineering processes

- The processes used for RE vary widely depending on the application domain, the people involved and the organisation developing the requirements.
- However, there are a number of generic activities common to all processes
  - Requirements elicitation;
  - Requirements analysis;
  - Requirements validation;
  - Requirements management.
- In practice, RE is an iterative activity in which these processes are interleaved.


How Projects Really Work (version 1.5)
Create your own cartoon at www.projectcartoon.com
How the customer explained it
How the project leader understood it
How the analyst designed it
How the programmer wrote it
What the beta testers received
How the business consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
iSwing
What marketing advertised
What the customer really needed

Outline

- Requirement Elicitation
- Requirement Specification and Analysis
- Requirement documents
- Requirement Validation
- Requirement Changes
- Requirement Management

Requirements elicitation

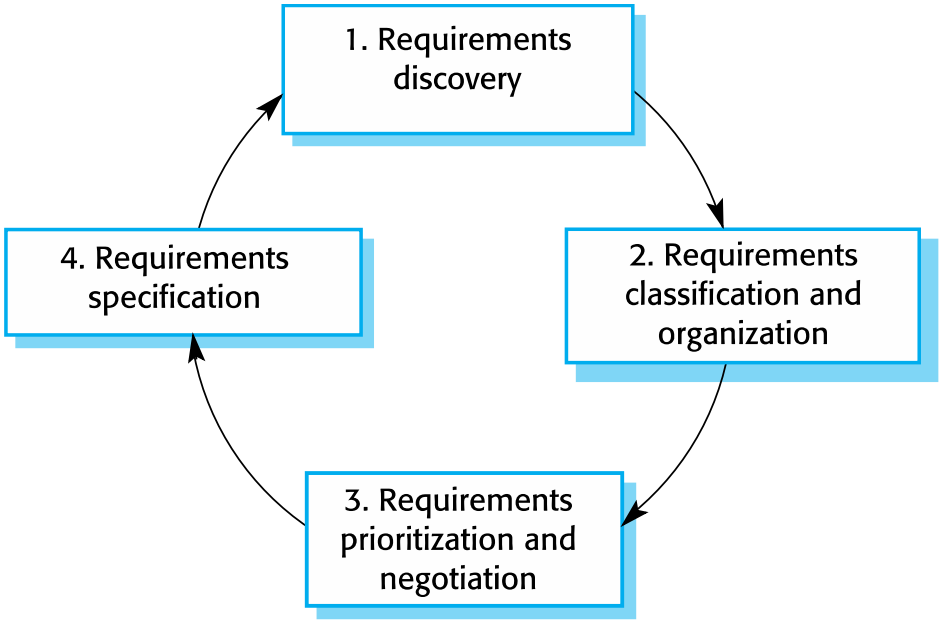
Requirements elicitation and analysis

- Sometimes called requirements elicitation or requirements discovery.
- Involves technical staff working with customers to find out about the application domain, the services that the system should provide and the system's operational constraints.
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders.*

Problems of requirements elicitation

- Stakeholders don’t know what they really want.
- Stakeholders express requirements in their own terms.
- Different stakeholders may have conflicting requirements.
- Organizational and political factors may influence the system requirements.
- The requirements change during the analysis process. New stakeholders may emerge and the business environment may change.

The requirements elicitation and analysis process



Process activities

Requirements discovery

- Interacting with stakeholders to discover their requirements. Domain requirements are also discovered at this stage.

Requirements classification and organisation

- Groups related requirements and organises them into coherent clusters.

Prioritisation and negotiation

- Prioritising requirements and resolving requirements conflicts.

Requirements specification

- Requirements are documented and input into the next round of the spiral.

Requirements specification

Requirements specification

- The process of writing down the user and system requirements in a requirements document.
- User requirements have to be understandable by end-users and customers who do not have a technical background.
- System requirements are more detailed requirements and may include more technical information.
- The requirements may be part of a contract for the system development
  - It is therefore important that these are as complete as possible.

Ways of writing a system requirements specification

Notation	Description
Natural language	The requirements are written using numbered sentences in natural language. Each sentence should express one requirement.
Structured natural language	The requirements are written in natural language on a standard form or template. Each field provides information about an aspect of the requirement.
Design description languages	This approach uses a language like a programming language, but with more abstract features to specify the requirements by defining an operational model of the system. This approach is now rarely used although it can be useful for interface specifications.
Graphical notations	Graphical models, supplemented by text annotations, are used to define the functional requirements for the system; UML use case and sequence diagrams are commonly used.
Mathematical specifications	These notations are based on mathematical concepts such as finite-state machines or sets. Although these unambiguous specifications can reduce the ambiguity in a requirements document, most customers don't understand a formal specification. They cannot check that it represents what they want and are reluctant to accept it as a system contract

Requirements and design

- In principle, requirements should state what the system should do and the design should describe how it does this.
- In practice, requirements and design are inseparable
  - A system architecture may be designed to structure the requirements;
  - The system may inter-operate with other systems that generate design requirements;
  - The use of a specific architecture to satisfy non-functional requirements may be a domain requirement.
  - This may be the consequence of a regulatory requirement.

Natural language specification

- Requirements are written as natural language sentences supplemented by diagrams and tables.
- Used for writing requirements because it is expressive, intuitive and universal. This means that the requirements can be understood by users and customers.

Guidelines for writing requirements

- Invent a standard format and use it for all requirements.
- Use language in a consistent way. Use shall for mandatory requirements, should for desirable requirements.
- Use text highlighting to identify key parts of the requirement.
- Avoid the use of computer jargon.
- Include an explanation (rationale) of why a requirement is necessary.

Problems with natural language

Lack of clarity

- Precision is difficult without making the document difficult to read.

Requirements confusion

- Functional and non-functional requirements tend to be mixed-up.

Requirements amalgamation

- Several different requirements may be expressed together.

Example requirements for the insulin pump software system

3.2 The system shall measure the blood sugar and deliver insulin, if required, every 10 minutes. *(Changes in blood sugar are relatively slow so more frequent measurement is unnecessary; less frequent measurement could lead to unnecessarily high sugar levels.)*

3.6 The system shall run a self-test routine every minute with the conditions to be tested and the associated actions defined in Table 1. *(A self-test routine can discover hardware and software problems and alert the user to the fact the normal operation may be impossible.)*

Structured specifications

- An approach to writing requirements where the freedom of the requirements writer is limited, and requirements are written in a standard way.
- This works well for some types of requirements e.g. requirements for embedded control system but is sometimes too rigid for writing business system requirements.

Form-based specifications

- Definition of the function or entity.
- Description of inputs and where they come from.
- Description of outputs and where they go to.
- Information about the information needed for the computation and other entities used.
- Description of the action to be taken.
- Pre and post conditions (if appropriate).
- The side effects (if any) of the function.

A structured specification of a requirement for an insulin pump

Insulin Pump/Control Software/SRS/3.3.2

Function Compute insulin dose: safe sugar level.

Description

Computes the dose of insulin to be delivered when the current measured sugar level is in the safe zone between 3 and 7 units.

Inputs Current sugar reading (r2); the previous two readings (r0 and r1).

Source Current sugar reading from sensor. Other readings from memory.

Outputs CompDose—the dose in insulin to be delivered.

Destination Main control loop.

A structured specification of a requirement for an insulin pump

Action	
CompDose is zero if the sugar level is stable or falling or if the level is increasing but the rate of increase is decreasing. If the level is increasing and the rate of increase is increasing, then CompDose is computed by dividing the difference between the current sugar level and the previous level by 4 and rounding the result. If the result, is rounded to zero then CompDose is set to the minimum dose that can be delivered.	
Requirements	
Two previous readings so that the rate of change of sugar level can be computed.	
Pre-condition	
The insulin reservoir contains at least the maximum allowed single dose of insulin.	
Post-condition	r0 is replaced by r1 then r1 is replaced by r2.
Side effects	None.

Tabular specification

- Used to supplement natural language.
- Particularly useful when you have to define a number of possible alternative courses of action.
- For example, the insulin pump systems bases its computations on the rate of change of blood sugar level and the tabular specification explains how to calculate the insulin requirement for different scenarios.

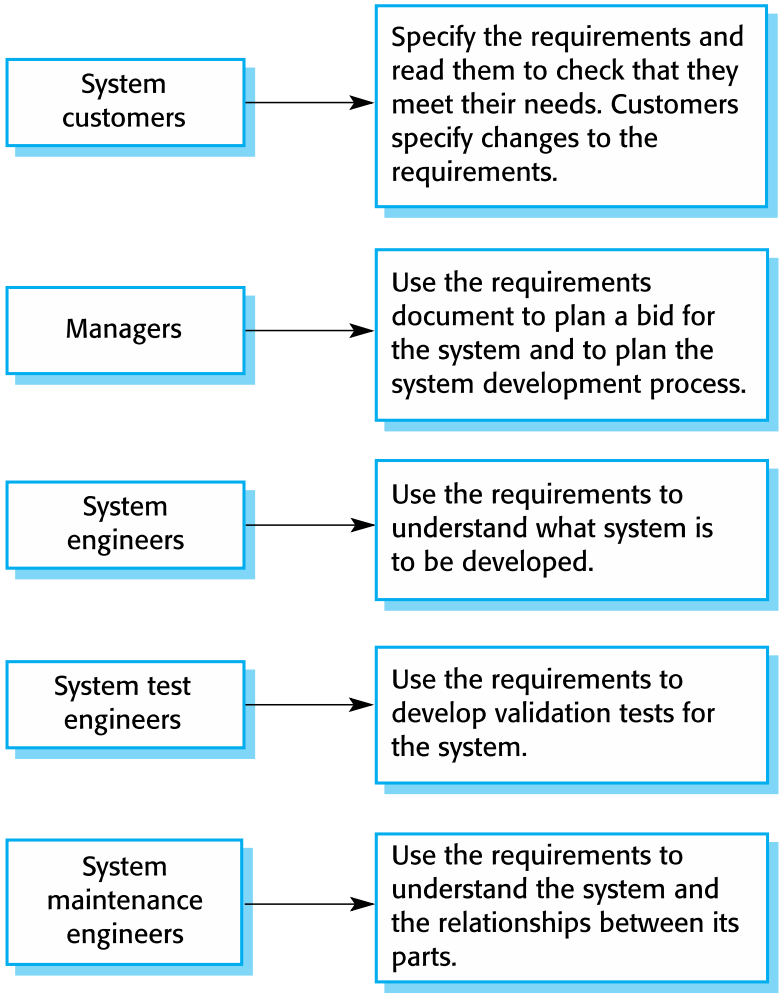
Tabular specification of computation for an insulin pump

Condition	Action
Sugar level falling (r2 < r1)	CompDose = 1
Sugar level stable (r2 = r1)	CompDose = 0
Sugar level increasing and rate of increase decreasing ((r2 – r1) < (r1 – r0))	CompDose = 0
Sugar level increasing and rate of increase stable or increasing ((r2 – r1) ≥ (r1 – r0))	CompDose = round ((r2 – r1)/4) If rounded result = 0 then CompDose = MinimumDose

The software requirements document

- The software requirements document is the official statement of what is required of the system developers.
- Should include both a definition of user requirements and a specification of the system requirements.
- It is NOT a design document. As far as possible, it should set of WHAT the system should do rather than HOW it should do it.

Users of a requirements document



The structure of a requirements document

Chapter	Description
Preface	This should define the expected readership of the document and describe its version history, including a rationale for the creation of a new version and a summary of the changes made in each version.
Introduction	This should describe the need for the system. It should briefly describe the system's functions and explain how it will work with other systems. It should also describe how the system fits into the overall business or strategic objectives of the organization commissioning the software.
Glossary	This should define the technical terms used in the document. You should not make assumptions about the experience or expertise of the reader.
User requirements definition	Here, you describe the services provided for the user. The nonfunctional system requirements should also be described in this section. This description may use natural language, diagrams, or other notations that are understandable to customers. Product and process standards that must be followed should be specified.
System architecture	This chapter should present a high-level overview of the anticipated system architecture, showing the distribution of functions across system modules. Architectural components that are reused should be highlighted.

The structure of a requirements document

Chapter	Description
System requirements specification	This should describe the functional and nonfunctional requirements in more detail. If necessary, further detail may also be added to the nonfunctional requirements. Interfaces to other systems may be defined.
System models	This might include graphical system models showing the relationships between the system components and the system and its environment. Examples of possible models are object models, data-flow models, or semantic data models.
System evolution	This should describe the fundamental assumptions on which the system is based, and any anticipated changes due to hardware evolution, changing user needs, and so on. This section is useful for system designers as it may help them avoid design decisions that would constrain likely future changes to the system.
Appendices	These should provide detailed, specific information that is related to the application being developed; for example, hardware and database descriptions. Hardware requirements define the minimal and optimal configurations for the system. Database requirements define the logical organization of the data used by the system and the relationships between data.
Index	Several indexes to the document may be included. As well as a normal alphabetic index, there may be an index of diagrams, an index of functions, and so on.

Requirements validation

Requirements validation

- Concerned with demonstrating that the requirements define the system that the customer really wants.
- Requirements error costs are high so validation is very important
  - Fixing a requirements error after delivery may cost up to 100 times the cost of fixing an implementation error.

Requirements checking

- Validity. Does the system provide the functions which best support the customer's needs?
- Consistency. Are there any requirements conflicts?
- Completeness. Are all functions required by the customer included?
- Realism. Can the requirements be implemented given available budget and technology
- Verifiability. Can the requirements be checked?

Requirements validation techniques

- Requirements reviews
  - Systematic manual analysis of the requirements.
- Prototyping
  - Using an executable model of the system to check requirements.
- Test-case generation
  - Developing tests for requirements to check testability.

Requirements reviews

- Regular reviews should be held while the requirements definition is being formulated.
- Both client and contractor staff should be involved in reviews.
- Reviews may be formal (with completed documents) or informal. Good communications between developers, customers and users can resolve problems at an early stage.

Review checks

- Verifiability
  - Is the requirement realistically testable?
- Comprehensibility
  - Is the requirement properly understood?
- Traceability
  - Is the origin of the requirement clearly stated?
- Adaptability
  - Can the requirement be changed without a large impact on other requirements?

Requirements change

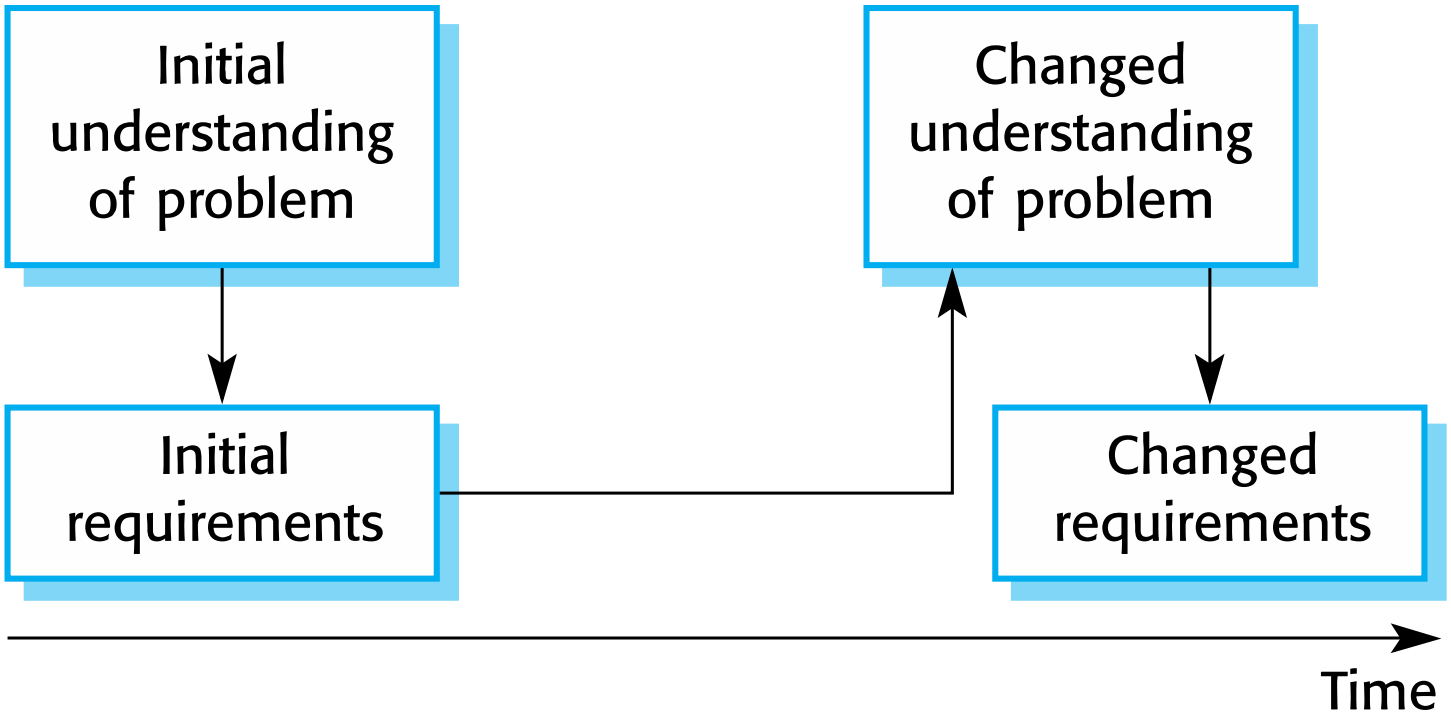
Changing requirements

- The business and technical environment of the system always changes after installation.
  - New hardware may be introduced, it may be necessary to interface the system with other systems, business priorities may change (with consequent changes in the system support required), and new legislation and regulations may be introduced that the system must necessarily abide by.
- The people who pay for a system and the users of that system are rarely the same people.
  - System customers impose requirements because of organizational and budgetary constraints. These may conflict with end-user requirements and, after delivery, new features may have to be added for user support if the system is to meet its goals.

Changing requirements

- Large systems usually have a diverse user community, with many users having different requirements and priorities that may be conflicting or contradictory.
  - The final system requirements are inevitably a compromise between them and, with experience, it is often discovered that the balance of support given to different users has to be changed.

Requirements evolution



Requirements management planning

- Establishes the level of requirements management detail that is required.
- Requirements management decisions:
  - *Requirements identification* - Each requirement must be uniquely identified so that it can be cross-referenced with other requirements.
  - *A change management process* - This is the set of activities that assess the impact and cost of changes.
  - *Traceability policies* - These policies define the relationships between each requirement and between the requirements and the system design that should be recorded.
  - *Tool support* - Tools that may be used range from specialist requirements management systems to spreadsheets and simple database systems.

Requirements change management

- Deciding if a requirements change should be accepted
  - *Problem analysis and change specification*
    - During this stage, the problem or the change proposal is analyzed to check that it is valid. This analysis is fed back to the change requestor who may respond with a more specific requirements change proposal, or decide to withdraw the request.
  - *Change analysis and costing*
    - The effect of the proposed change is assessed using traceability information and general knowledge of the system requirements. Once this analysis is completed, a decision is made whether or not to proceed with the requirements change.
  - Change implementation
    - The requirements document and, where necessary, the system design and implementation, are modified. Ideally, the document should be organized so that changes can be easily implemented.

SWOT Analysis: Strengths, Weaknesses, Opportunities, and Threats

What is a SWOT analysis and why should you use one?

A SWOT analysis guides you to identify the positives and negatives inside your organization (***S***trength & ***W***eakness) and outside of it, in the external environment (***O***pportunity & ***T***hreat). Developing a full awareness of your situation can help with both strategic planning and decision-making.

When do you use SWOT?

You might use it to:

- Explore possibilities to problems.
- Make decisions for your initiative.
- Determine where change is possible.
- Adjust and refine plans mid-course.

What are the elements of a SWOT analysis?

SWOT Analysis Focus

A SWOT analysis focuses on Strengths, Weaknesses, Opportunities, and Threats.

Internal		External	
Strengths	Weaknesses	Opportunities	Threats

SWOT Analysis Focus

If a looser structure helps you brainstorm, you can group positives and negatives to think broadly about your organization and its external environment.

Positives	Negatives
• Strengths • Assets • Resources • Opportunities • Prospects	• Weaknesses • Limitations • Restrictions • Threats • Challenges

SWOT Analysis Focus

This is a third option for structuring your SWOT analysis, which may be appropriate for a larger initiative that requires detailed planning. This "TOWS Matrix" is adapted from Fred David's *Strategic Managemen*t text.

	STRENGTHS 1. 2. 3. 4.	**WEAKNESSES** 1. 2. 3. 4.
OPPORTUNITIES 1. 2. 3. 4.	Opportunity-Strength (OS) Strategies Use the strengths to take advantage of opportunities 1. 2.	Opportunity-Weakness (OW) Strategies Overcome weaknesses by taking advantage of opportunities 1. 2.
THREATS 1. 2. 3. 4.	Threat-Strength (TS) Strategies Use strengths to avoid threats 1. 2.	Threat-Weakness (TW) Strategies Minimize weaknesses and avoid threats 1. 2.

Listing Your Internal Factors: Strengths and Weaknesses (S, W)

General areas to consider

- Human resources - staff, volunteers, board members, target population
- Physical resources - your location, building, equipment
- Financial - grants, funding agencies, other sources of income
- Activities and processes - programs you run, systems you employ
- Past experiences - building blocks for learning and success, your reputation in the community

Listing External Factors: Opportunities and Threats (O, T)

Forces and facts that your group does not control include

- Future trends in your field or the culture
- The economy - local, national, or international
- Funding sources - foundations, donors, legislatures
- Demographics - changes in the age, race, gender, culture of those you serve or in your area
- The physical environment (Is your building in a growing part of town? Is the bus company cutting routes?)
- Legislation (Do new federal requirements make your job harder...or easier?)
- Local, national or international events

Steps for conducting a SWOT analysis

- Designate a leader or group facilitator.
- Designate a recorder to back up the leader if your group is large.
- Introduce the SWOT method and its purpose in your organization.
- Let all participants introduce themselves.
- Have each group designate a recorder; direct them to create a SWOT analysis.
- Reconvene the group at the agreed-upon time to share results.
- Discuss and record the results.
- Prepare a written summary of the SWOT analysis to give to participants.

How do you use your SWOT analysis?

Use it to:

- Identify the issues or problems you intend to change.
- Set or reaffirm goals.
- Create an action plan.

In Summary

A realistic recognition of the weaknesses and threats that exist for your effort is the first step to countering them with a robust set of strategies that build upon strengths and opportunities. A SWOT analysis identifies your strengths, weaknesses, opportunities and threats to assist you in making strategic plans and decisions.

CPSC 362
Software Engineering

Outline

- Abstraction
- United Modeling Language (UML)
- UML diagram types
- Use case diagram
- Internal flow of a use case

Abstraction

- Abstraction allows us to ignore unessential details
- Two definitions for abstraction:
  - Abstraction is a *thought process* where ideas are distanced from objects
    - Abstraction as activity
  - Abstraction is the *resulting idea* of a thought process where an idea has been distanced from an object
    - Abstraction as entity
- Ideas can be expressed by models

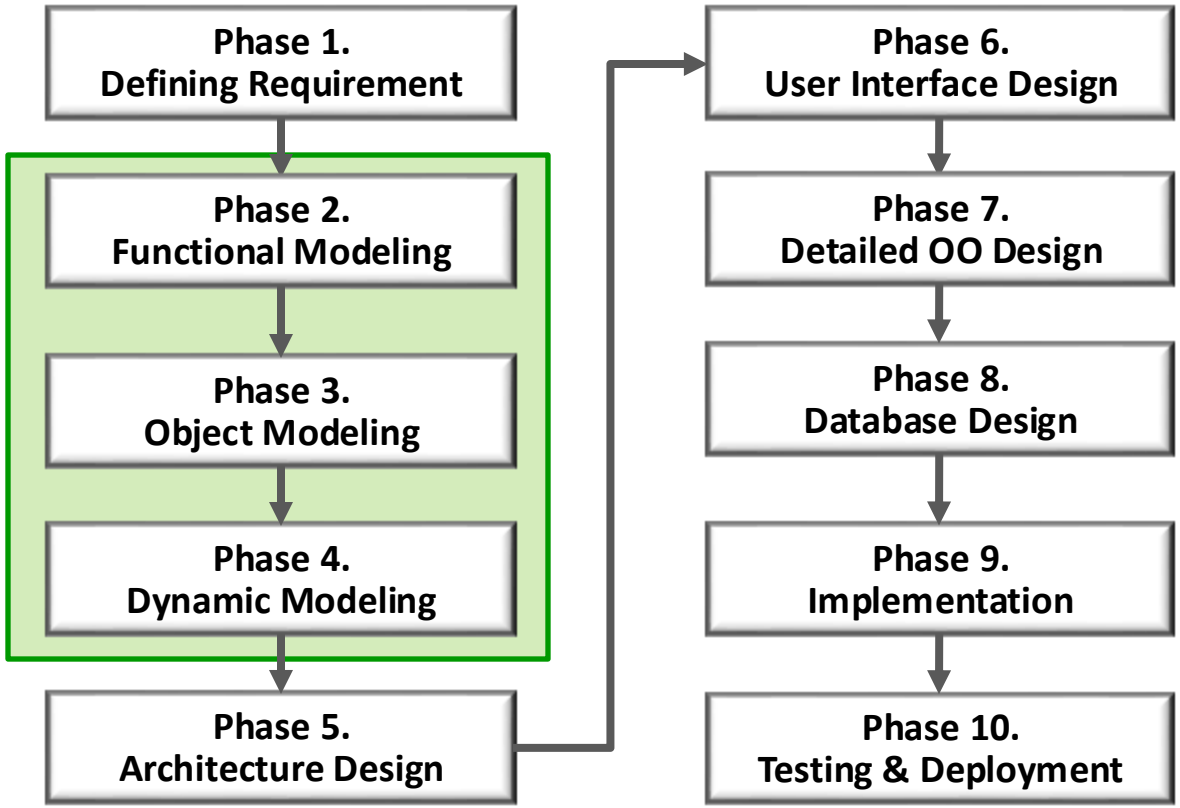
Model

- A model is an abstraction of a system
  - A system that no longer exists
  - An existing system
  - A future system to be built.

We use Models to describe Software Systems

- Object model: What is the structure of the system?
- Functional model: What are the functions of the system?
- Dynamic model: How does the system react to external events?
- System Model: Object model + functional model + dynamic model

A Core Object Oriented Process



Unified Modeling Language (UML)

- Graphical Language for Modeling Object-Oriented Systems
- 14 Diagrams in UML 2.4.1
- Initial Development (1990's)
  - by Grady Booch, Ivar Jacobson and Jim Rumbaugh
  - IBM Rational Software
- Adopted by Object Management Group (OMG) in 1997
- Accepted by ISO as Industry Standard for modeling Software-Intensive Systems in 2000.
- UML models can be exchanged by using XMI.

UML diagram types

Use case diagrams

- show the interactions between a system and its environment.

Class diagrams

- show the object classes in the system and the associations between these classes.

Activity diagrams

- show the activities involved in a process or in data processing .

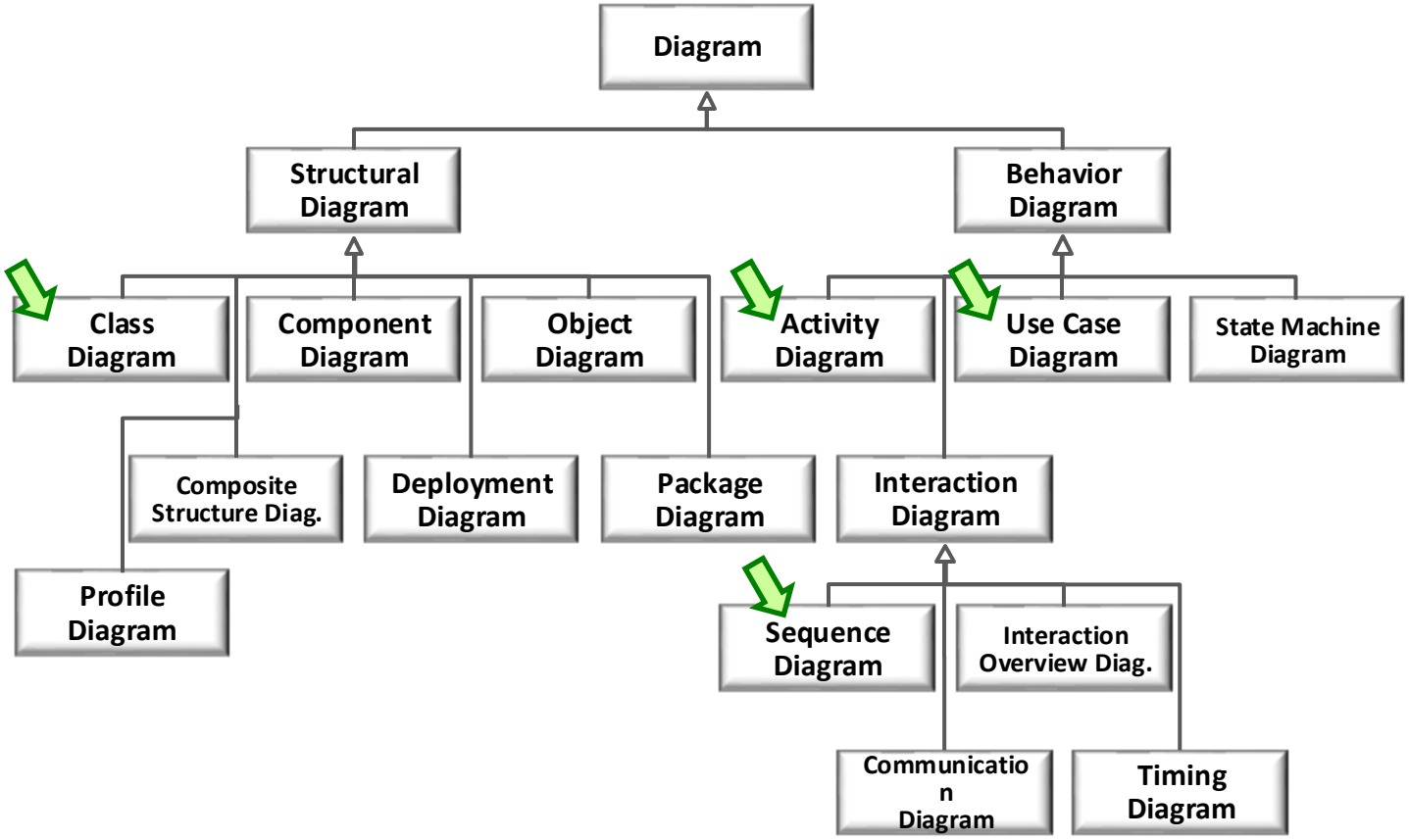
Data flow diagrams

- show the system design and information flow within the system.

Sequence diagrams

- show time-based and/or sequential interactions between actors and the system and between system components.

14 Diagrams of UML



Use of graphical models

As a means of facilitating discussion about an existing or proposed system

- Incomplete and incorrect models are OK as their role is to support discussion.

As a way of documenting an existing system

- Models should be an accurate representation of the system but need not be complete.

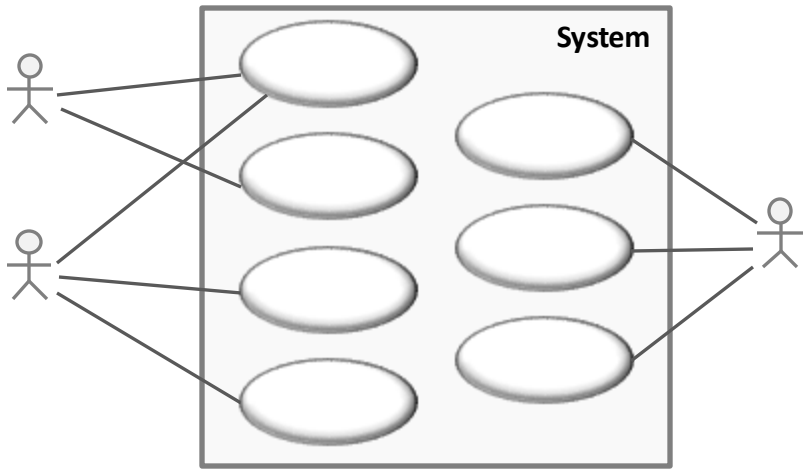
As a detailed system description that can be used to generate a system implementation

- Models have to be both correct and complete.

USE CASE DIAGRAM

Overview of Use Case Diagram

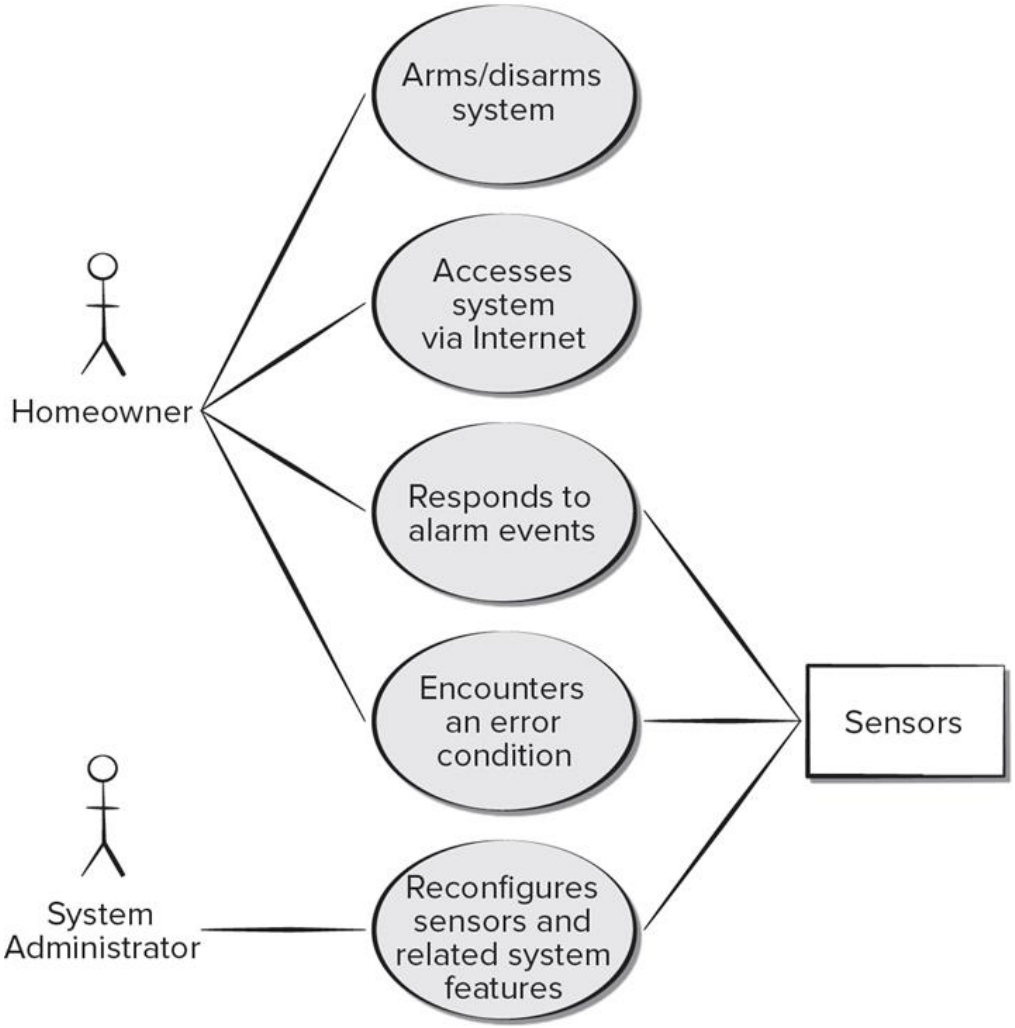
- To describe the externally observable behavior of a system
  - Exposed Functionality
- To describe the main interactions between the system and external entities including users and other systems
  - Interaction between Actors and System



Use case definition

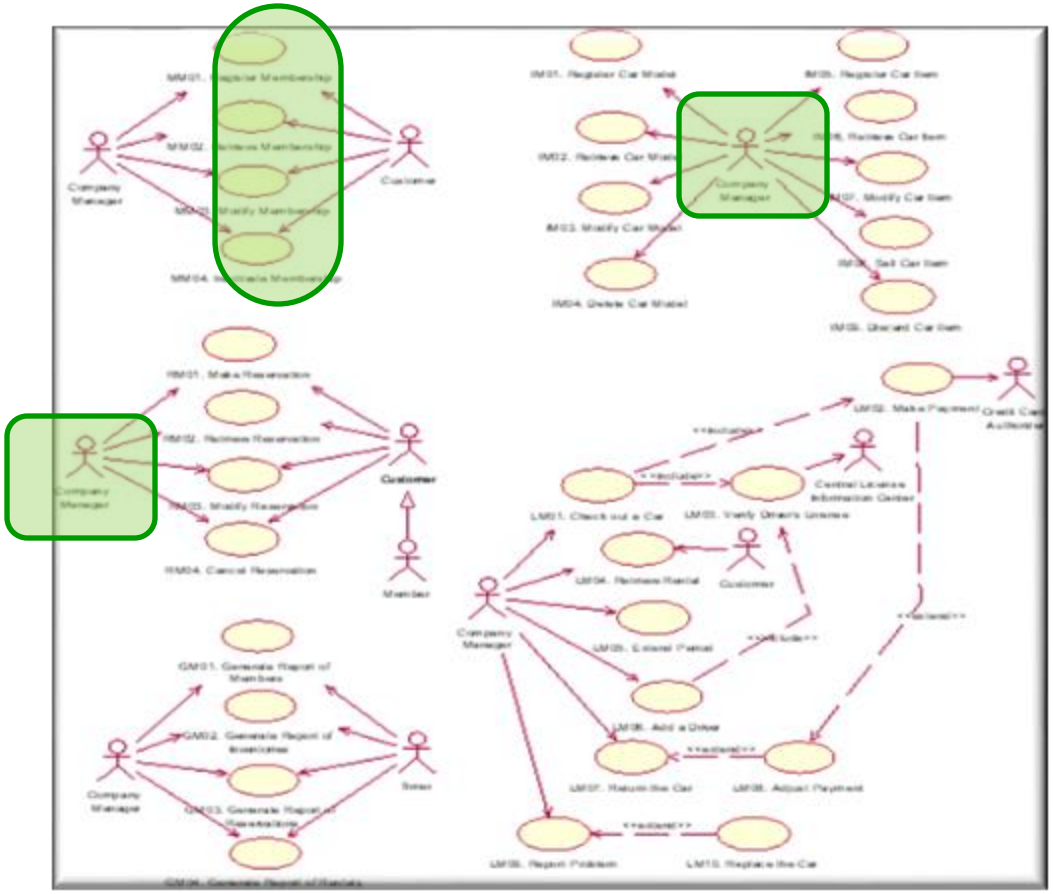
- A collection of user scenarios that describe the thread of usage of a system
- Each scenario is described from the point-of-view of an “actor” - a person or device that interacts with the software in some way
- Each scenario answers the following questions:
  - Who is the primary actor, the secondary actor (s)?
  - What are the actor’s goals?
  - What preconditions should exist before the story begins?
  - What main tasks or functions are performed by the actor?
  - What extensions might be considered as the story is described?
  - What variations in the actor’s interaction are possible?
  - What system information will the actor acquire, produce, or change?
  - Will the actor have to inform the system about changes in the external environment?
  - What information does the actor desire from the system?
  - Does the actor wish to be informed about unexpected changes?

UML Use Case Diagram



Use Case Diagram

- Car Rental System

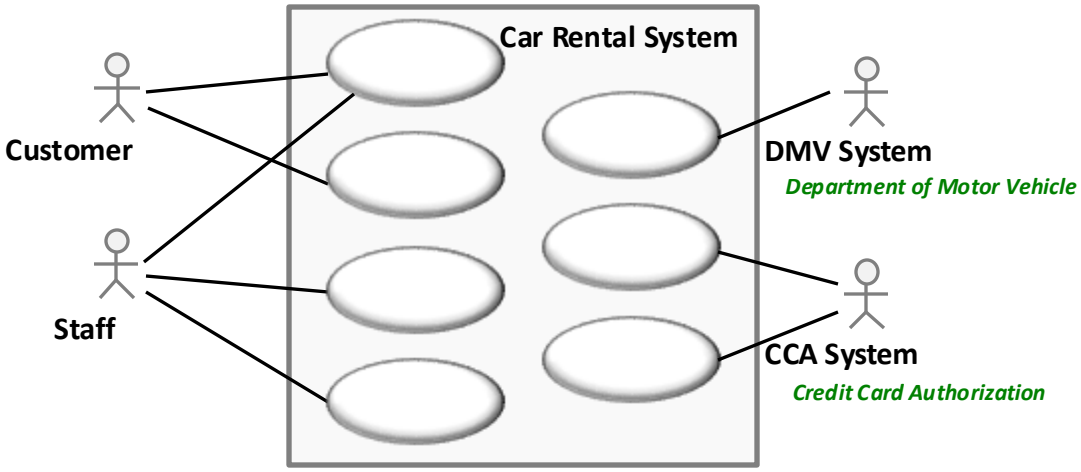


Actor (1)

- A type of role played by an entity that interacts with the subject
- To represent roles played by human users, external hardware, or other subject
- Actor can be;
  - Human User
  - Hardware Device
  - Another System interacting with the system

Actor (2)

- Active Actor
  - Actor can be active as Trigger
- Passive Actor
  - External Systems

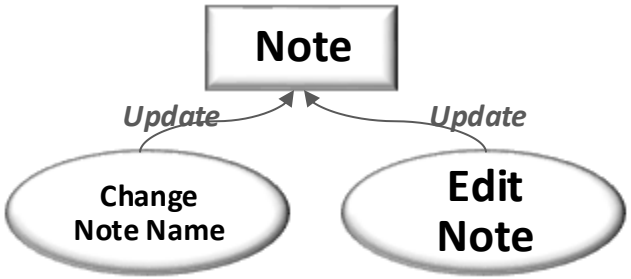


Use Case

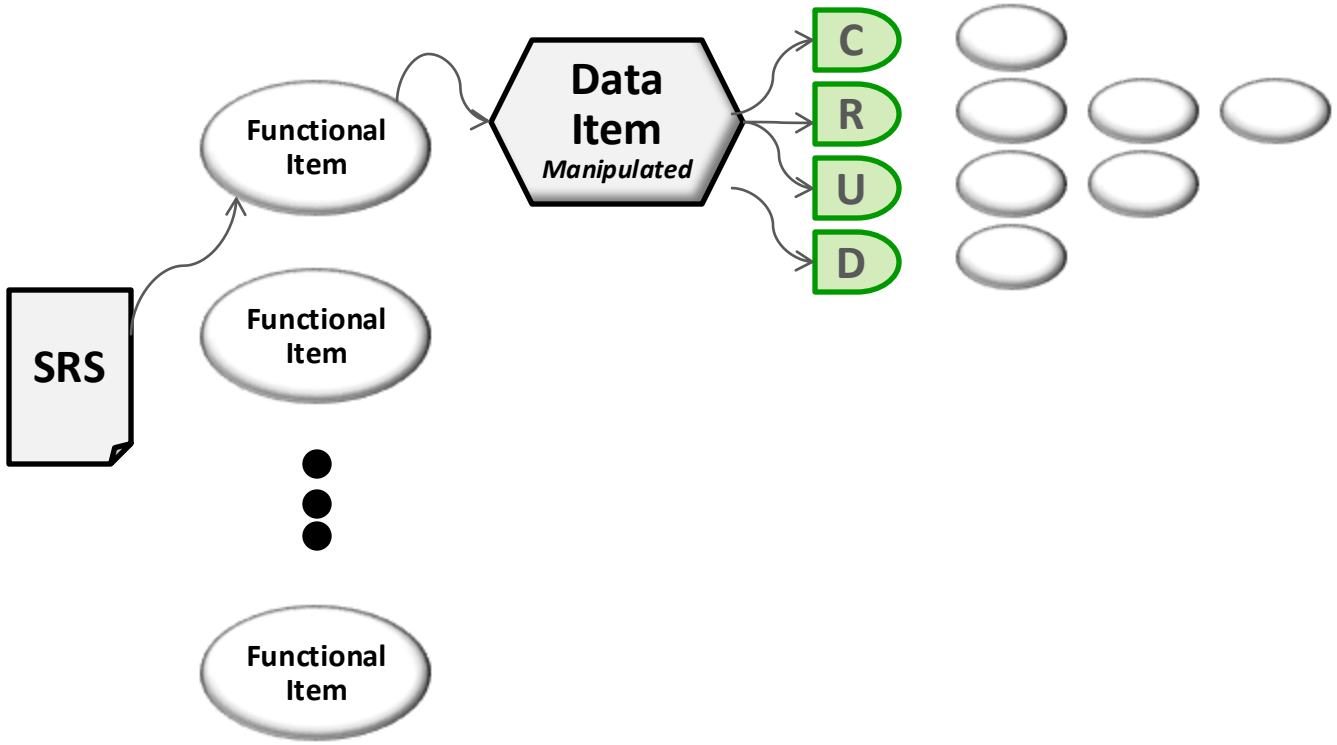
- A use case specifies a cohesive functionality of the system.
  - Unit of Functionality
- Use cases are generally larger-grained than
  - A Function in Procedural Program
  - A Method in Object-Oriented Program
- Notation
- Verb Form

Finding Use Cases (1)

- A use case is a functional unit.
- A function is to manipulate data elements, i.e. objects.
- Consider CRUD-based functions for a target object.
  - Given an object, consider use cases corresponding to CRUD operations on the object.
    - **C** for Create, **R** for Retrieve, **U** for Update, and **D** for Delete
  - There can be more than one use cases for each of CRUD.
    - For 'U' operation on 'Note' object, two use cases are derived.
      - Change Note Name
      - Edit Note

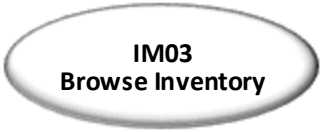


Finding Use Cases (2)



Use Cases for Complex Systems

- Decompose into sub-systems, i.e. packages.
- Apply a numbering scheme.
- Example)
  - 'IM' for Inventory Management



Relationships in Use Case Diagram

Generalization 

- A *Base* use case represents the functionality of several *Derived use cases*.

Include «include» 

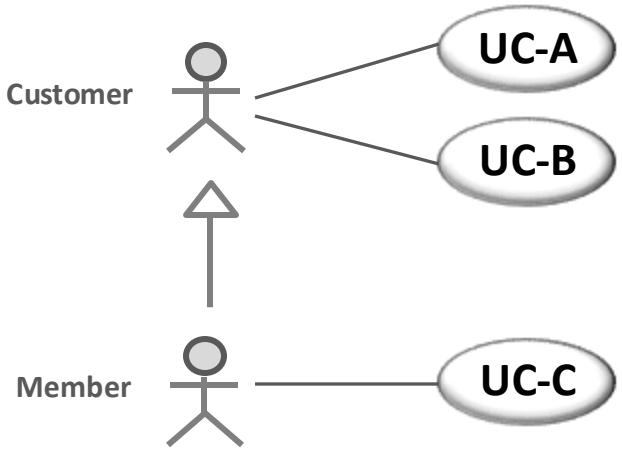
- A *Base* use case always invoke the *Included* use case.

Extend «extend» 

- A *Base* use case invokes the *Extended* use case when the given condition is met.

Relationship – Generalization (1)

- Generalization between Actors
  - A child actor inherits the behavior of its parent actor.
  - Example)



Relationship – Generalization (2)

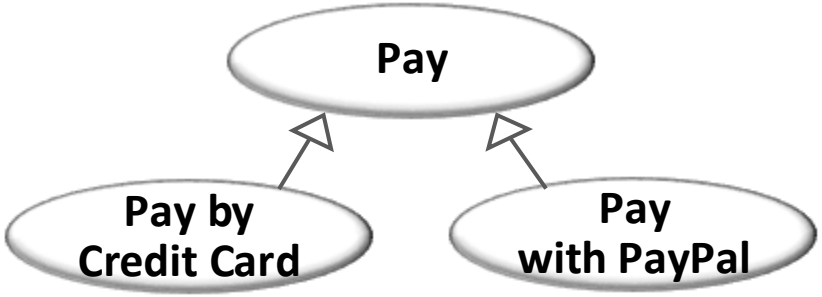
Generalization between Use Cases

- A taxonomic relationship between a base use case and derived use case.
- The base use case captures the common functionality.
- The base use case is *not* implemented.

Substitutability

- At runtime, the base use case is substituted by one of its derived use cases.

Example

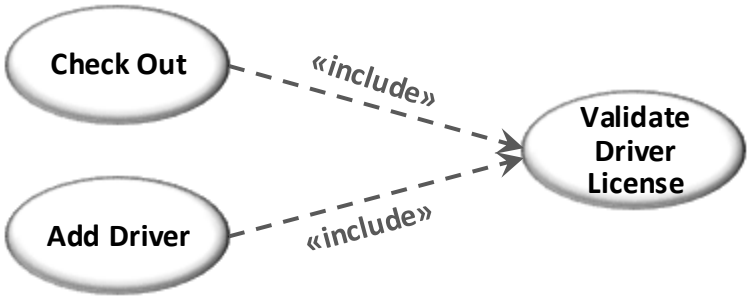


Relationship – Include

Include Relationship

- To define a use case that contains common behavior among multiple *base* use cases.
- A base use case always invokes its included use cases.

Example



Relationship - Extend

Extend Relationship

- To define a use case that contains an optional and extended behavior of the *base* use case.
- Extended use cases are invoked only when certain condition is met.

Example

Use Case Description

- Contents
  - Use Case Name
  - Overview
    - Goal and Scope
  - Pre-condition
    - State what must always be true before a scenario is begun in the use case.
  - Post-condition
    - State what must be true on successful completion of the use case.
  - Flow of Events
    - Main Flow, Alternative Flows, Error Flows
  - Scenarios
    - An Instance of a Use Case
    - Hence, many possible scenarios for a use case may occur.

Internal Flows of a Use Case

Main Flow

- The most *common*, *general*, and *normal* sequence of tasks to deliver the required functionality
- The target use case is fulfilled.

Alternative Flow

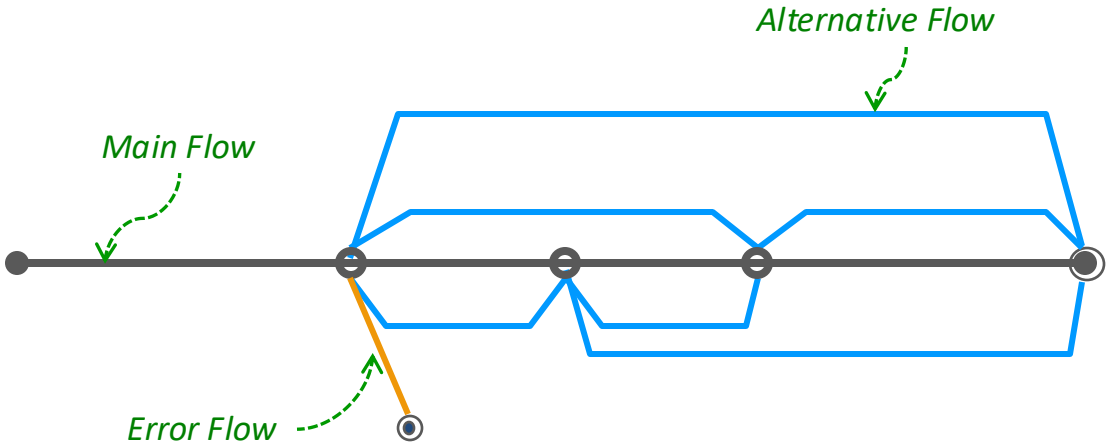
- Not the most common, but an *alternative/exceptional* sequence of tasks to deliver the required functionality
- The functionality of target use case is fulfilled.

Error Flow

- A sequence of tasks which results in an abnormal termination.
- The functionality of target use case is not fulfilled.

Internal Flows of a Use Case

- 3 Types of Flows



Use Case Description

- Main Flow

Actor	System
Enters the information of the customer, the driver license, the car model, and the rental period.	
	Checks the validity of the driver license. If it is invalid, invokes E-1.
	Searches for matching cars. It displays available cars, and asks the user to choose a car. If there is no car matching, invokes A-1.
Chooses a car from the list.	
	Asks user to choose an insurance option; “Comprehensive”, “Liability Only”, “Collision Damage Waiver Only”, or none.
Chooses an insurance option.	
	Calculates and displays the fee for rental and insurance. Asks the user to enter a credit card information.
…	

Use Case Description

- Alternative Flows, A-1

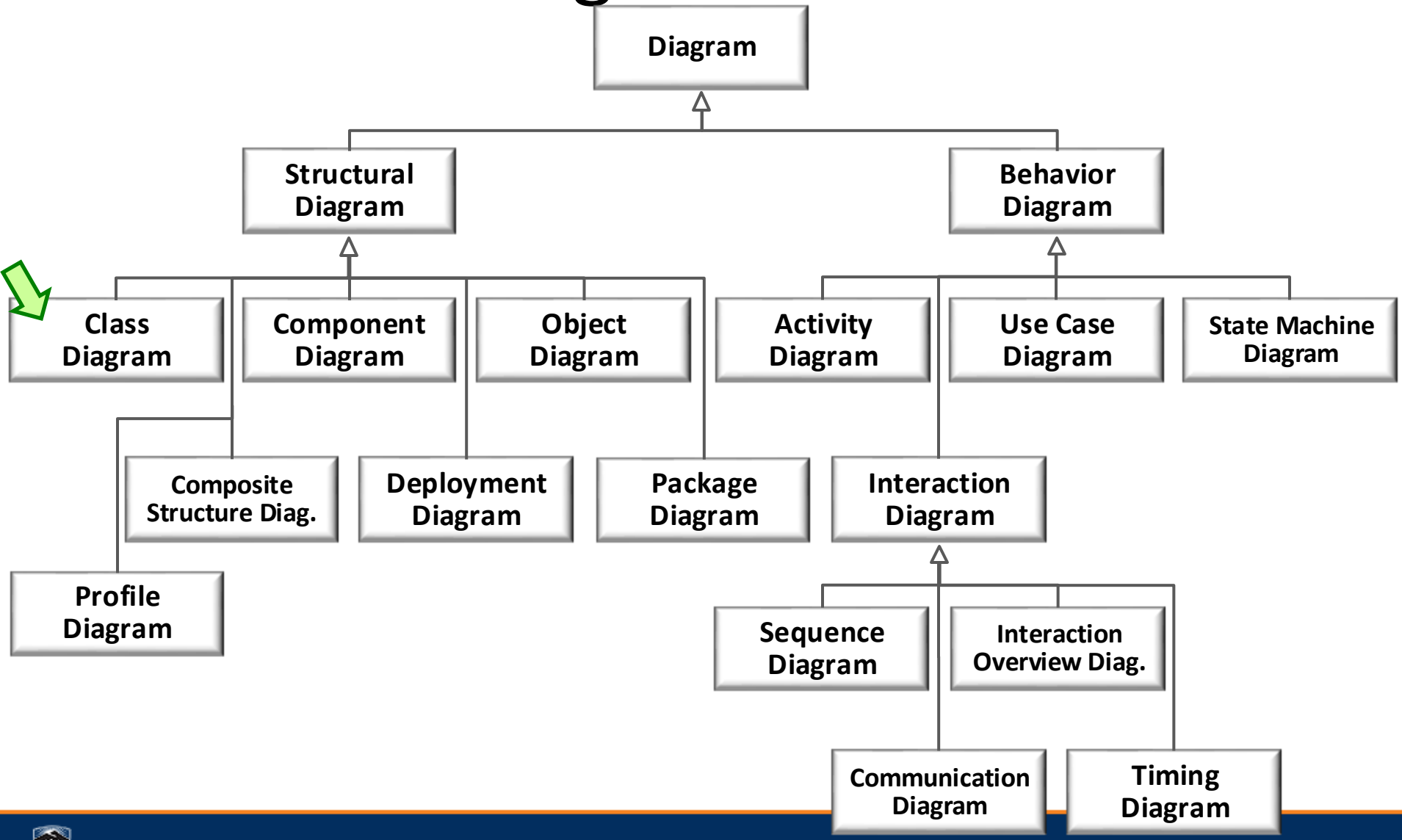
Actor	System
	Displays “No Matching Car.”. Asks the user to choose a different car model.
Enters a different model or ‘abort’.	
	If ‘abort’, terminates the use case. Otherwise, resumes with the different model.

- Error Flows, E-1

Actor	System
	Displays “Invalided License”.
	Terminates the use case.

CPSC 362
Software Engineering

14 Diagrams of UML



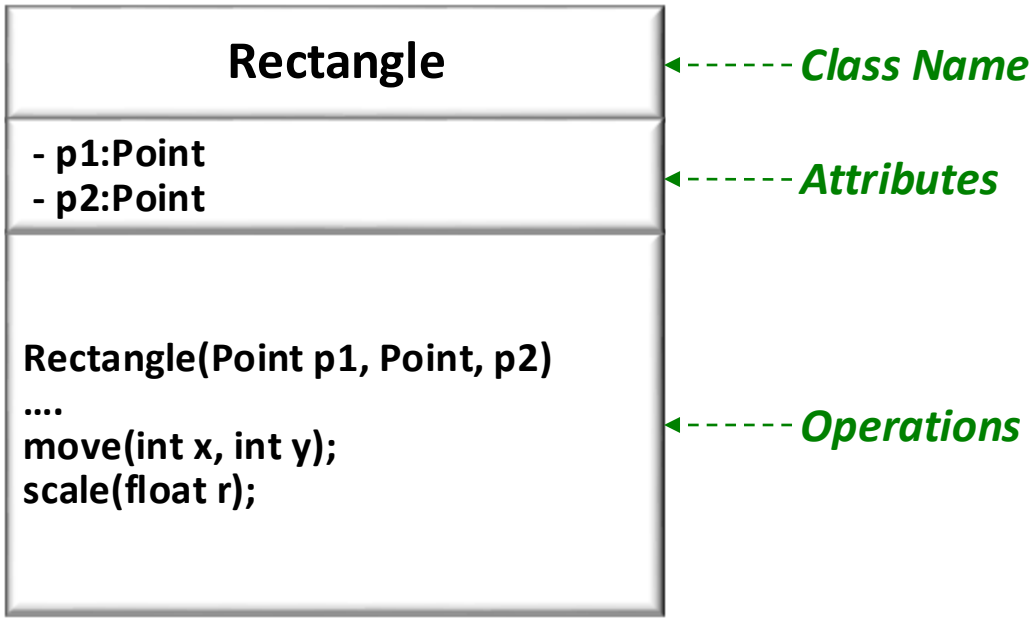
Class Diagram

Class Diagram

- Class
  - How to find Classes
- Attributes
- Operations
- Relationships

Class (1)

- A class specifies related attributes and their operations.
- UML Notation



Class (2)

- Name of Classes
  - Noun Phrase
- Example)
  - Customer
  - Staff
  - Product
  - Rental
  - Reservation

Finding Classes

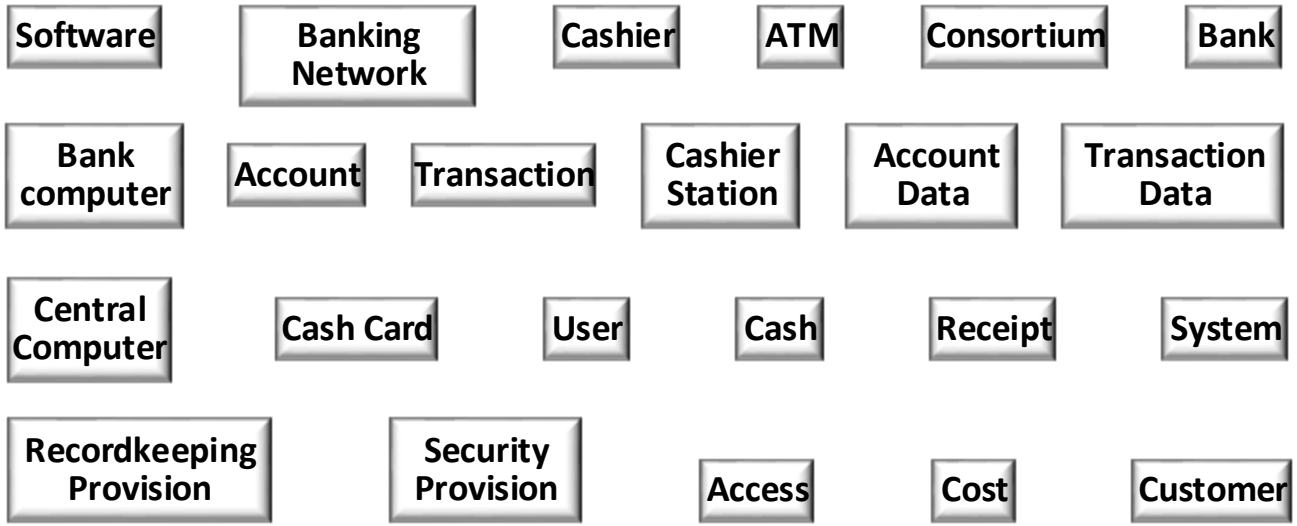
- Object Modeling Technique (OMT)
  - By James Rumbaugh
- List candidate classes found in the SRS. Then, discard bad classes.

Step 1. Underline Nouns

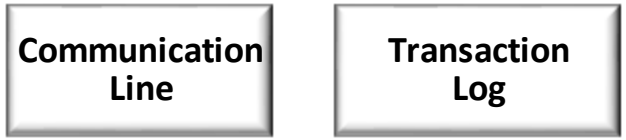
An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate record keeping and security provisions. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computers; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.

Tentative Classes

- From SRS



- From Domain Knowledge



Step 2. Keep the Right Classes (1)

- Discard unnecessary and incorrect classes according to the following criteria.
- Discarding Criteria
  - Redundant Classes
    - If two classes express the same information, the most descriptive name should be kept. ATM example) Customer over User
  - Irrelevant Classes
    - If a class has little or nothing to do with the problem, eliminate it with a careful judgment.
  - Vague Classes
    - A class should be specific.
    - Eliminate classes that have ill-defined boundaries or that are too broad in scope.

Step 2. Keep the Right Classes (2)

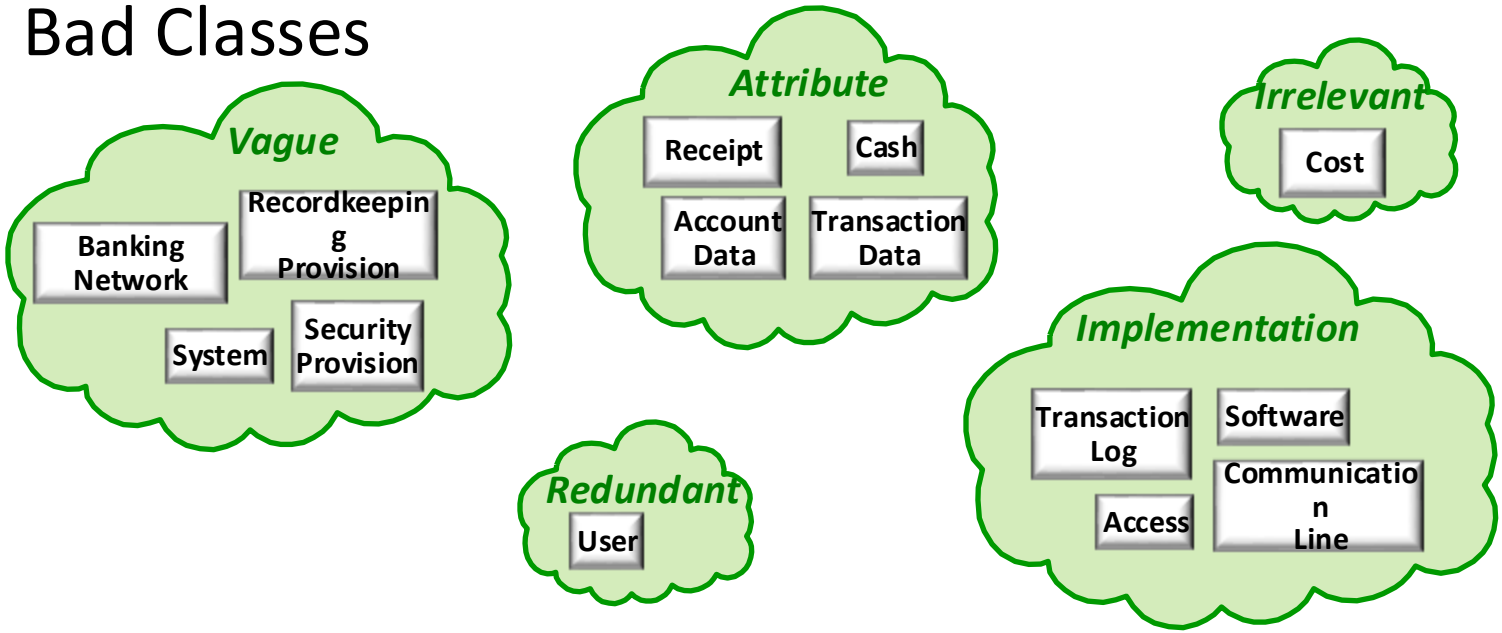
- Attributes
  - Names that primarily describe individual objects should be restated as attributes.
- Operations
  - If a name describes an operation that is applied to objects and not manipulated in its own right, discard it.
- Roles
  - The name of a class should reflect its intrinsic nature and not a role that it plays in an association.
  - Discard the class that represents a certain role.
- Implementation Constructs
  - Constructs extraneous to the real world should be discarded.
  - They may be needed later during design phase.

Step 2. Keep the Right Classes (3)

- Candidate classes indicating value
  - Values are also expresses as nouns.
- Candidate classes indicating Report/Output
  - Report and output are only the results of invoking methods.

Step 2. Keep the Right Classes (4)

- Bad Classes



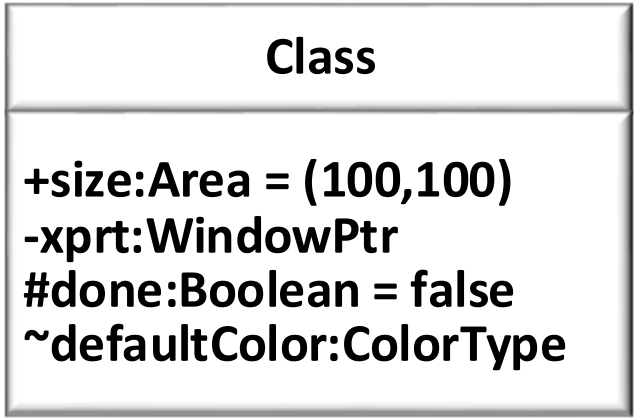
- Good Classes

Account | ATM | Bank | Bank computer | Cash Card | Cashier

Cashier Station | Central Computer | Consortium | Customer | Transaction

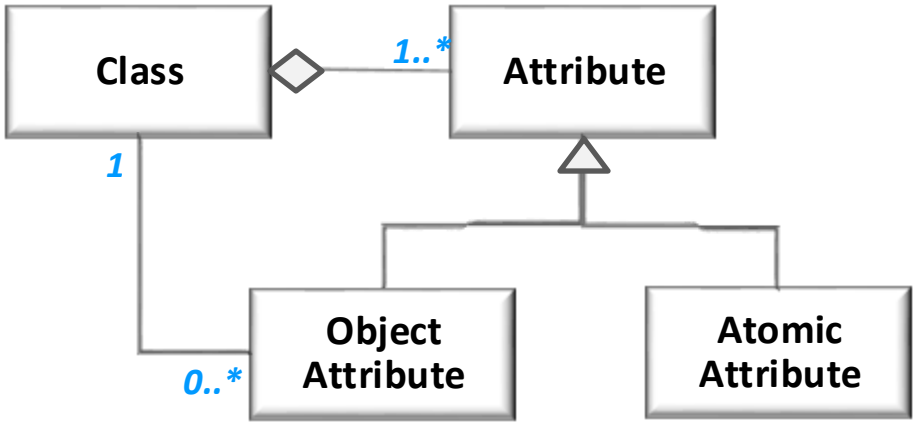
Attributes (1)

- What are attributes ?
  - Data Properties of Objects
  - Abstract Storage of the Data
- Syntax
  - Visibility name : type-expression [multiplicity ordering] =initial-value {property-string}
- Visibility
  - + public visibility
  - - private visibility
  - # protected visibility
  - ~package visibility



Attributes (2)

- Finding Attributes
  - Attributes correspond to nouns followed by *possessive phrases*.
  - Attributes are less likely to be fully described in the problem statement.
- Class vs. Attribute
  - Class is a composite information, consisting of attributes.
  - Attribute is an atomic data item.
  - Attribute can be an object.

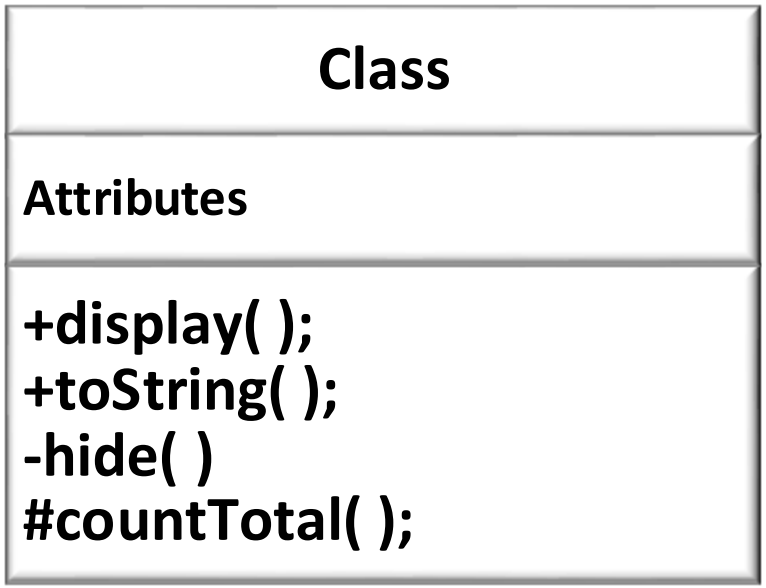


Operations (1)

- An operation provides a cohesive function for a class.
  - Relatively fine-grained unit
- A set of all operations constitutes the whole functionality of a class.
- An operation is specified with an operation signature.
  - Operation Name, Parameters, Return Type
- UML Notation
  - visibility Name (parameters) : return-type [multiplicity] {property-string}

Operations (2)

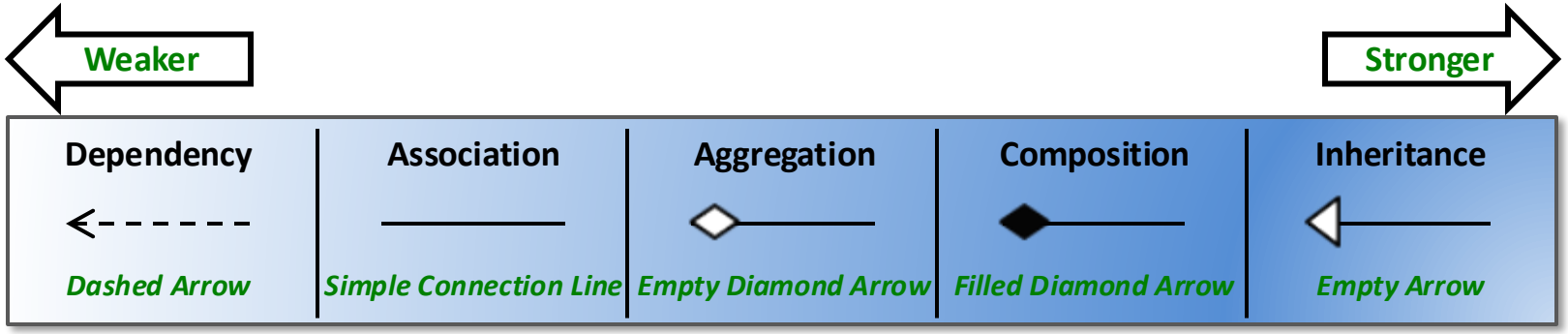
- Example



Relationship between Classes (1)

- Dependency
  - A temporal link between instances
- Association
  - A persistent link between instances
- Aggregation
  - A part-of relationship between instances
- Composition
  - A strong aggregation
- Inheritance
  - A parent-of relationship

Relationship between Classes (2)



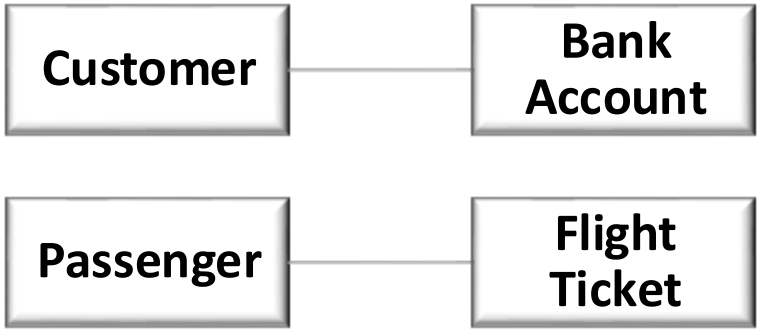
Dependency

- Definition
  - A casual, temporal link between instances
  - Use «invoke».
- Example
  - *Customer* inquires the *US Exchange Rate.*

- Often, the dependency is omitted in class diagrams.

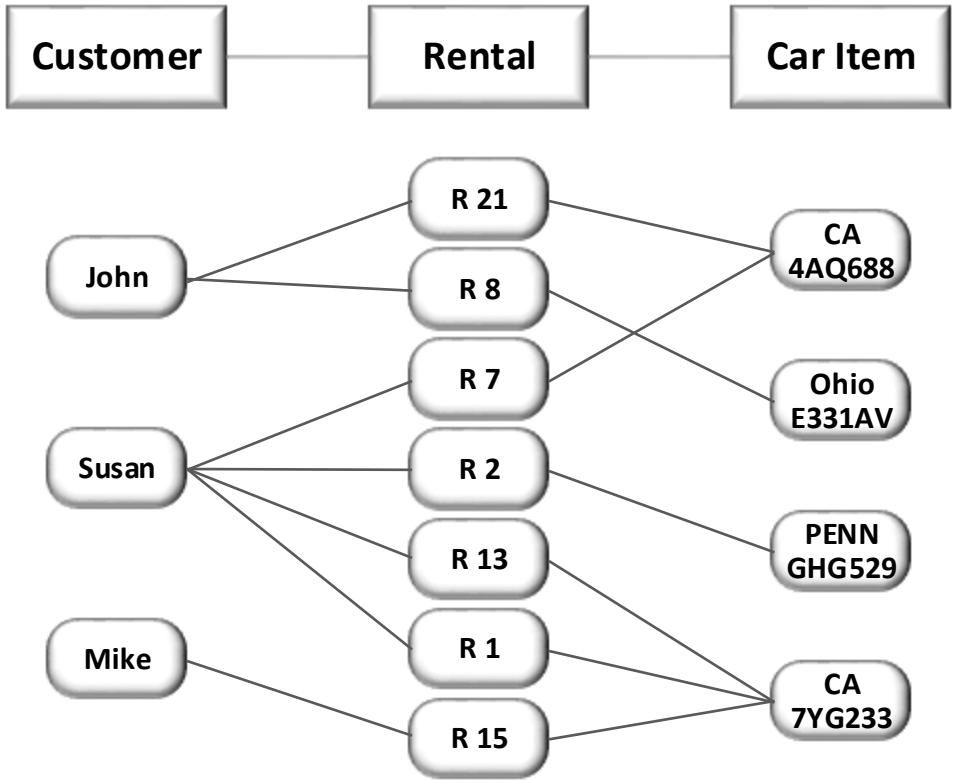
Association

- Definition
  - An association specifies a persistent link between instances.
    - Long-lasting or Permanent Links
  - Use a secondary storage to record all the links between instances.
    - Typically in a Database Table



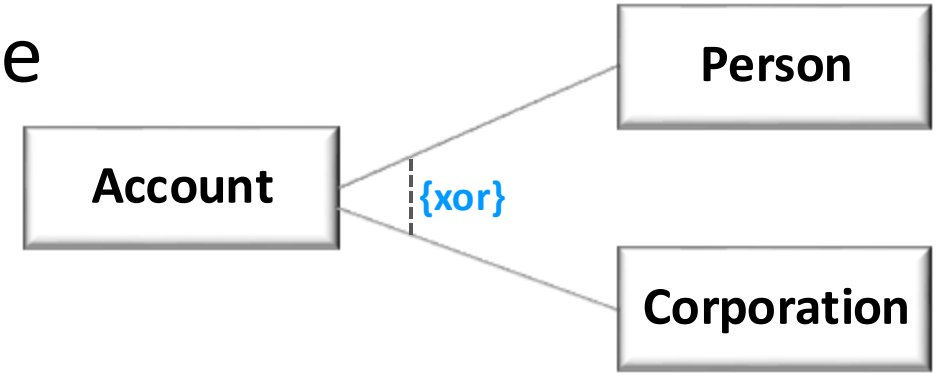
Association and Link

- Association is between Classes.
- Link is between Instances.



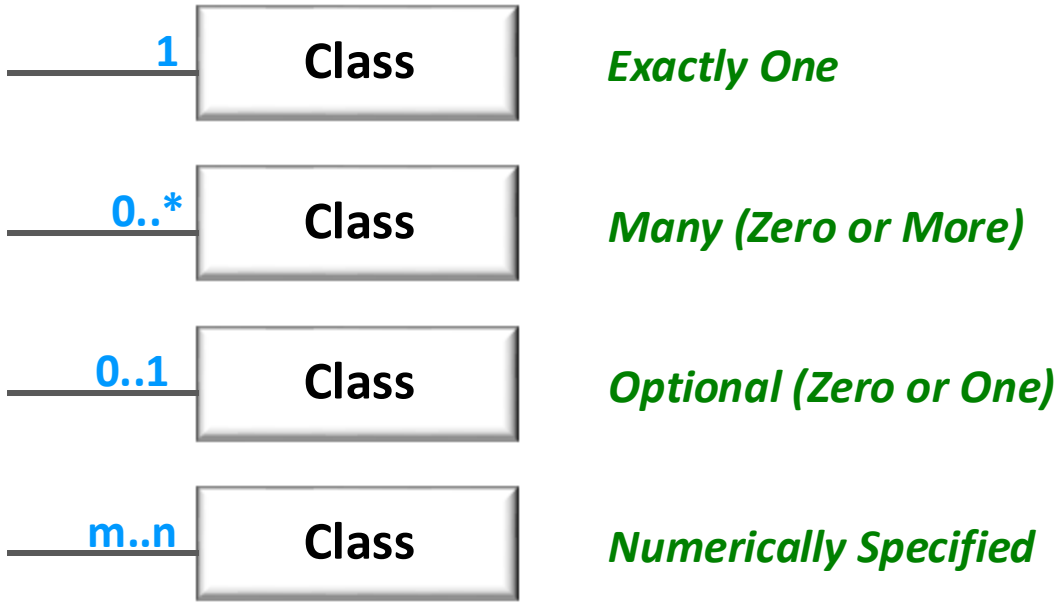
{xor} Constraint

- Usage
  - Only one of several potential associations can be applied to each instance.
- Example



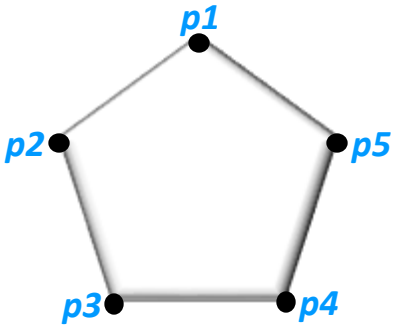
Cardinality (1)

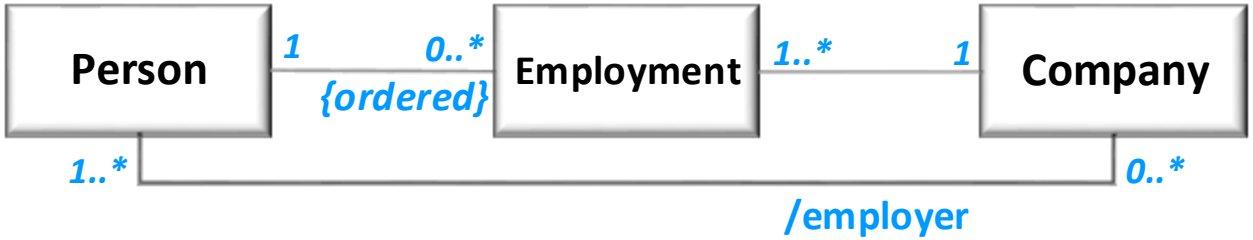
- Semantics
  - To specify the number of links that can exist between instances
- Notation

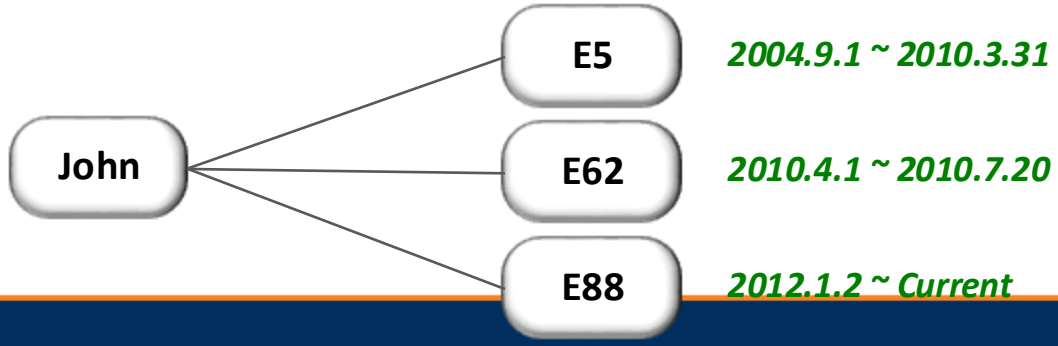


Cardinality (2)

- {ordered}
  - To show that the end represents an ordered set

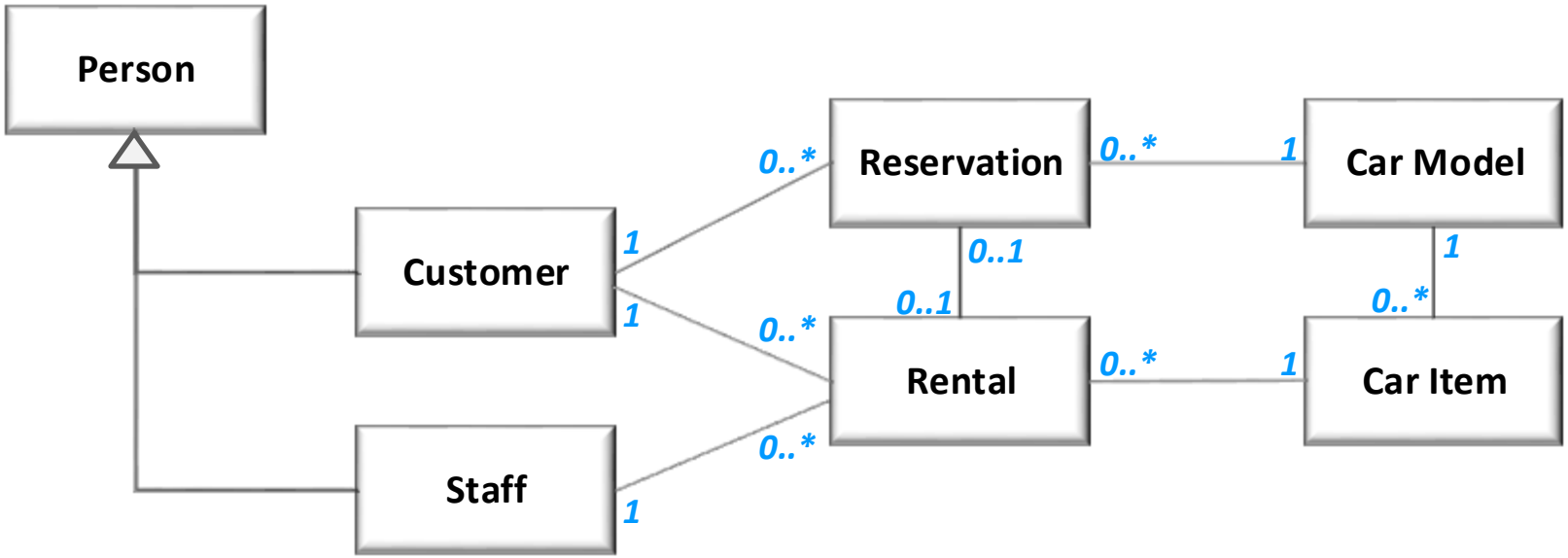






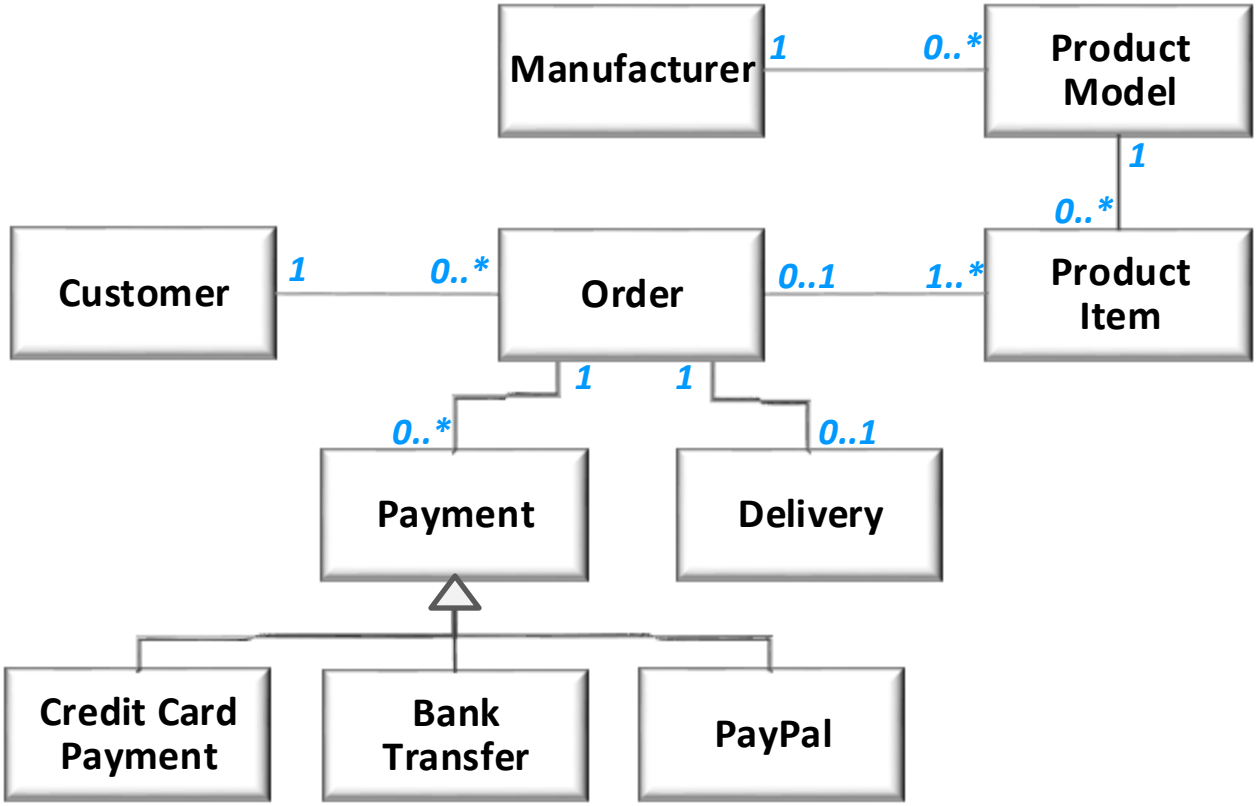
Cardinality (3)

- Car Rental System



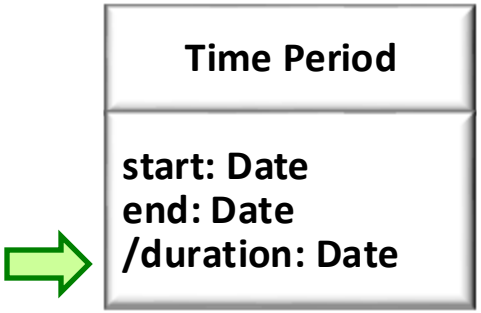
Cardinality (4)

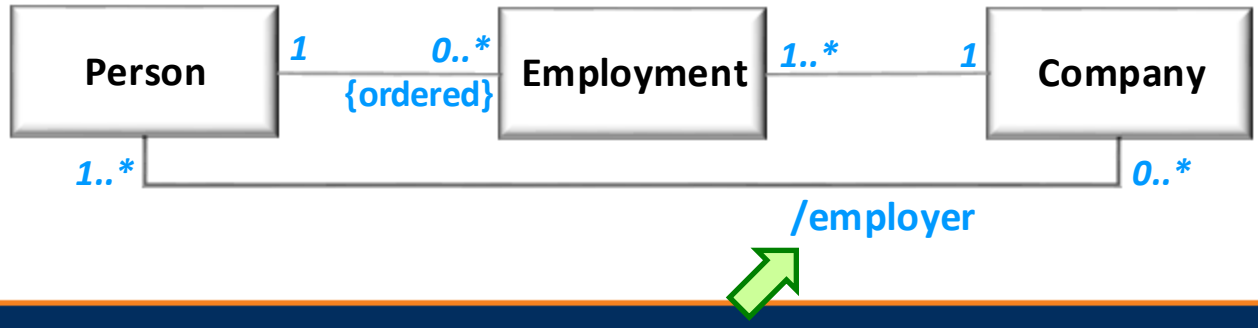
- eCommerce System



Derived Associations and Attributes

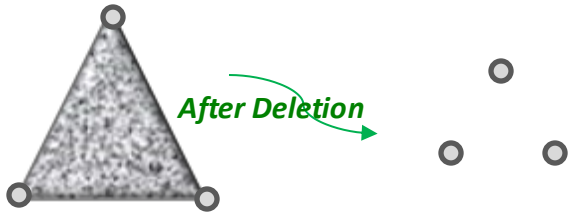
- Definition
  - Derived Attribute
  - Derived Association
- Use a slash, /
- Consideration
  - Redundancy
  - Efficiency

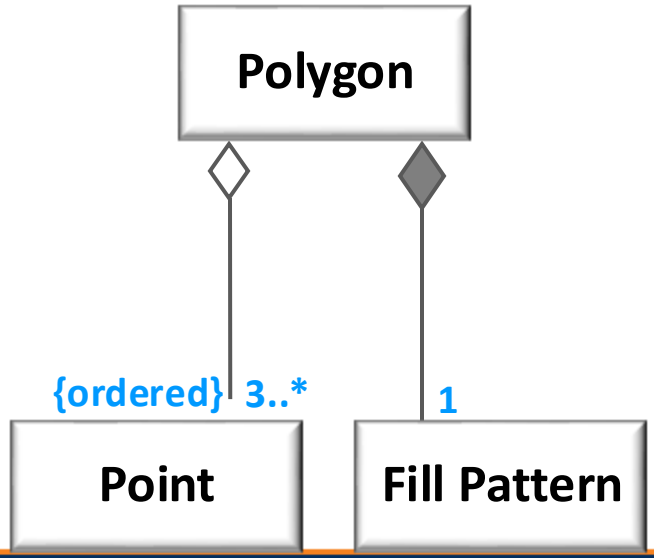




Aggregation and Composition

- Aggregation
  - Has, Part_of, Consist_of Relationship
- Composition
  - A stronger form of aggregation
  - A part object may belong to only one whole object.

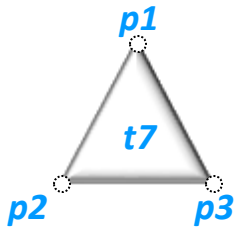




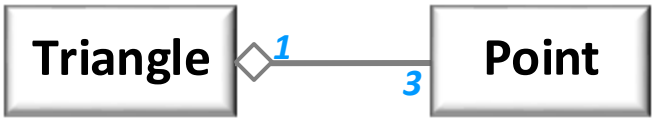
Association, Aggregation, Composition

- Association

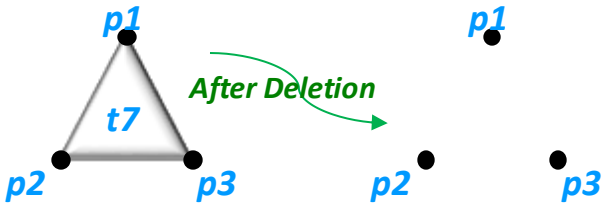
  - Independent Existence



- Aggregation

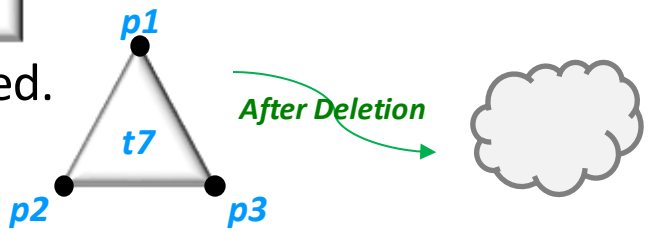


  - Whole part exists with parts.



- Composition

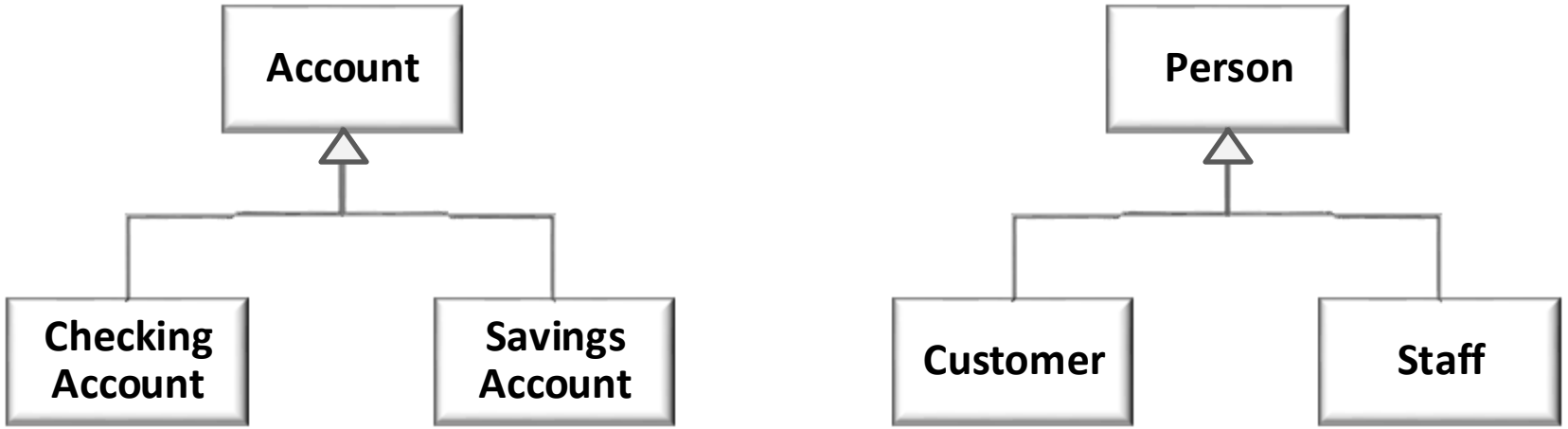
  - Parts are deleted when the whole is deleted.



- Their implementations are different.

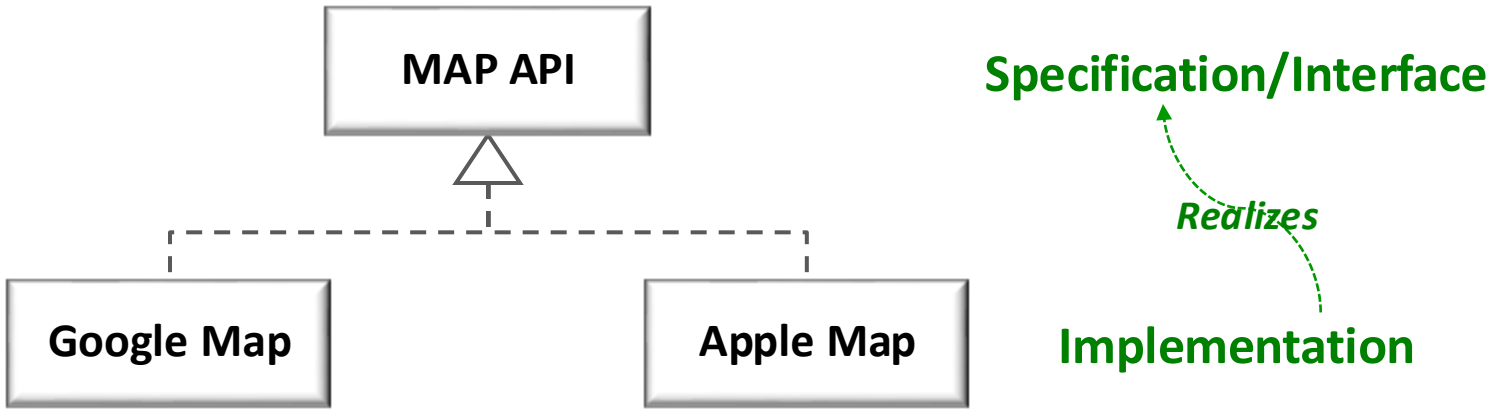
Inheritance

- Definition
  - A relationship between a *general* class and a *specific* class
    - Superclass, Subclass
  - A subclass inherits the features of superclass.



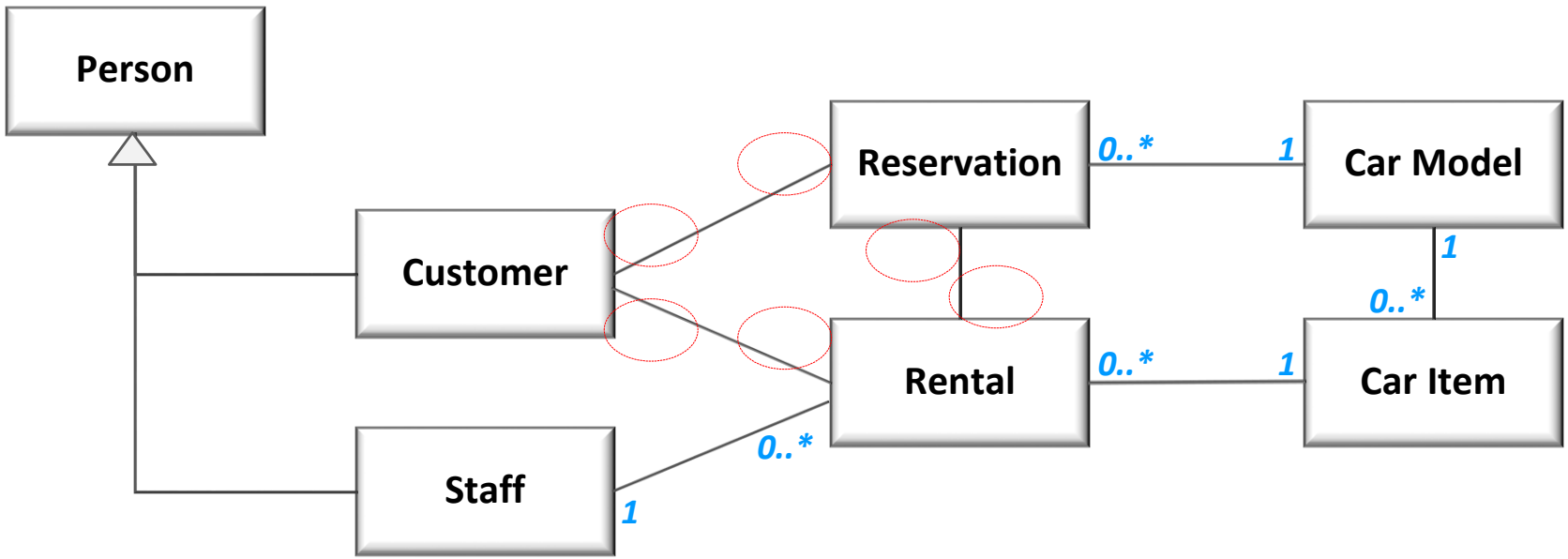
Realization

- Definition
  - A relationship between a specification and an implementation
  - Used for specifying stepwise *Refinement.*
- Notation
  - Dashed line with a triangular arrowhead



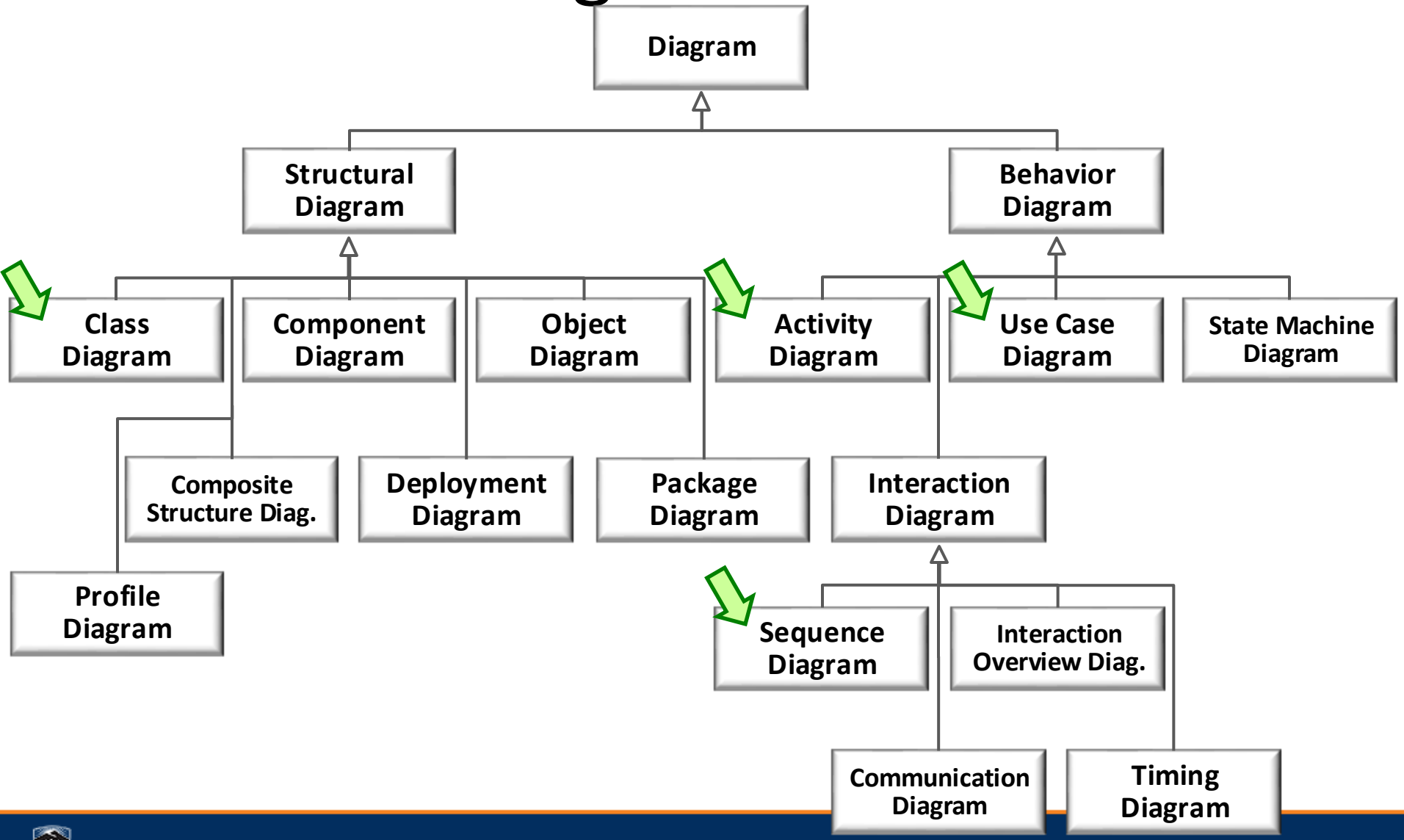
In-class exercise

Define the cardinalities on the associations in the class diagram for Car Rental Syste m. There are 6 places to fill in.



CPSC 362
Software Engineering

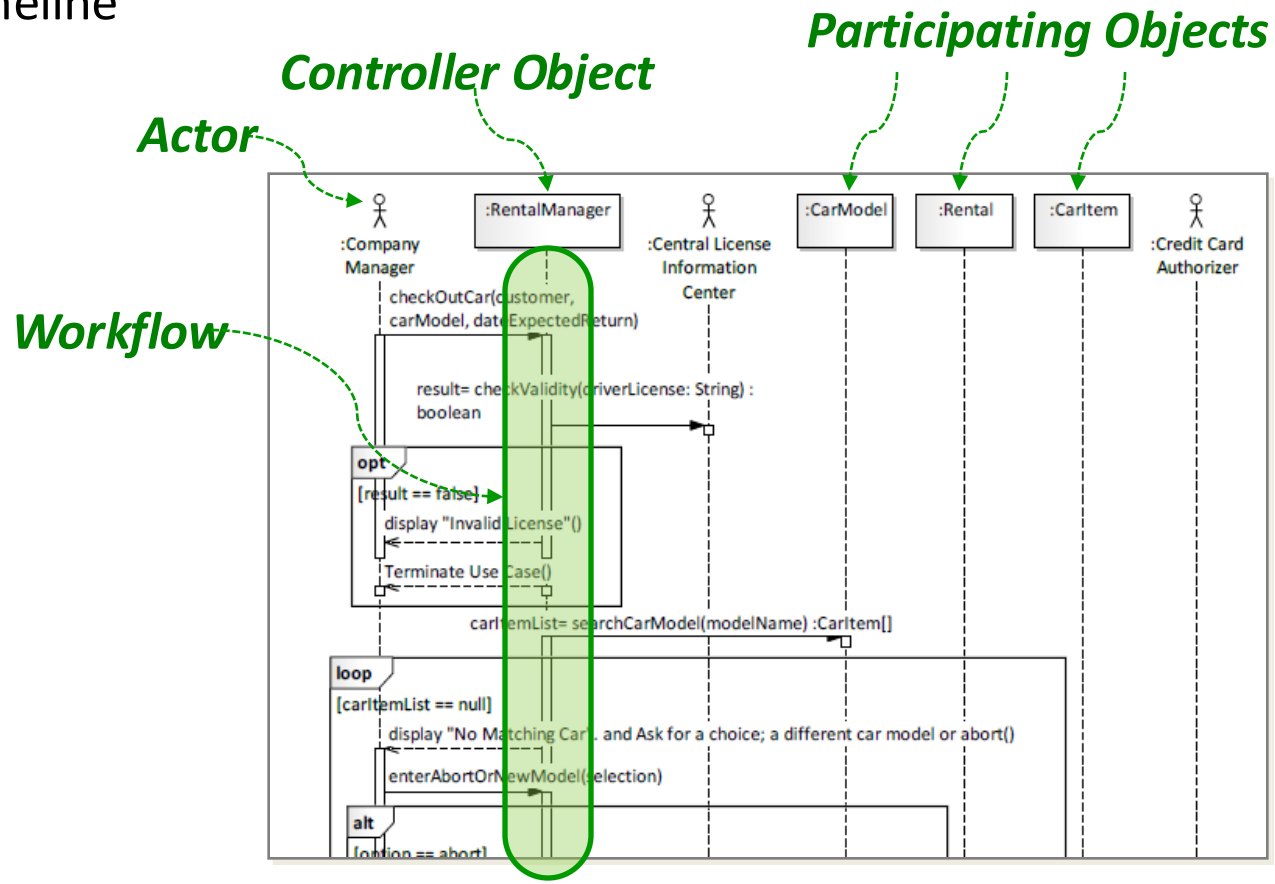
14 Diagrams of UML



Sequence Diagram

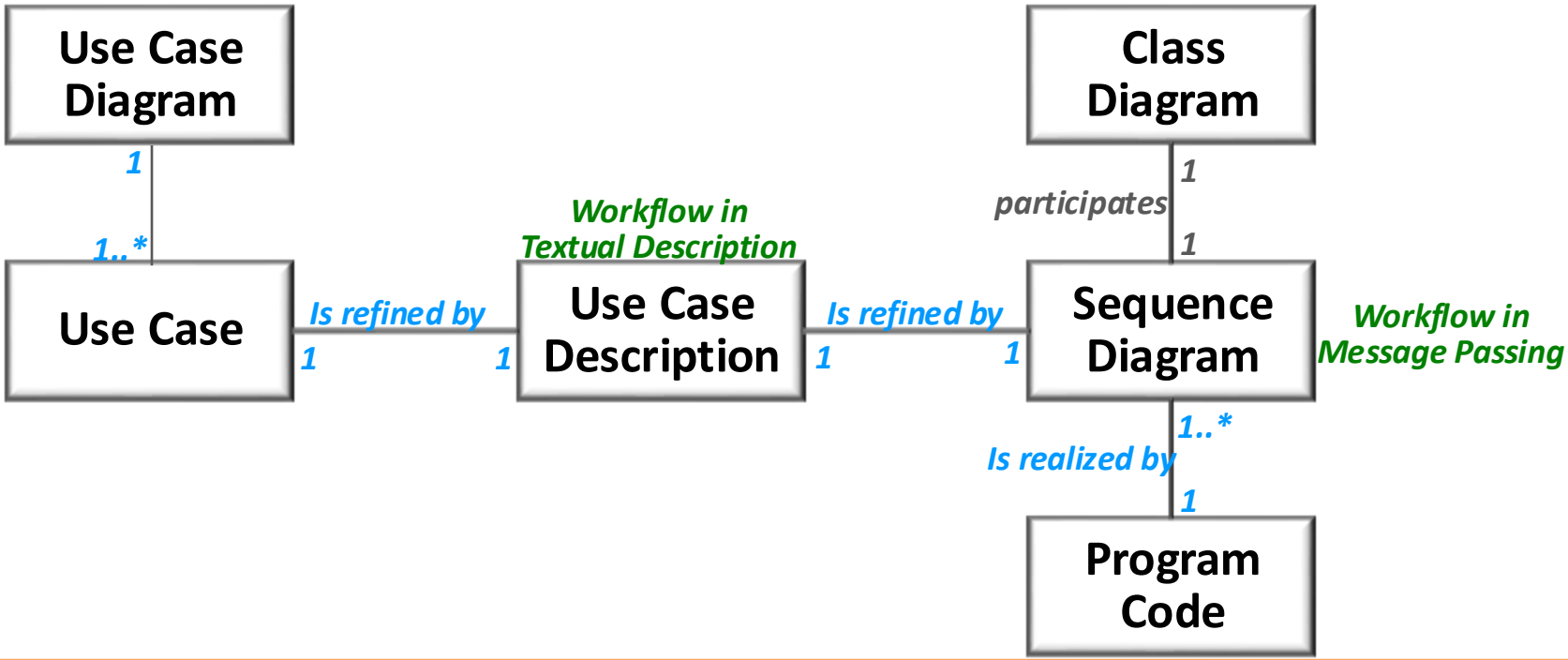
Sequence Diagram

- To specify the sequences of message exchanges among objects using timeline



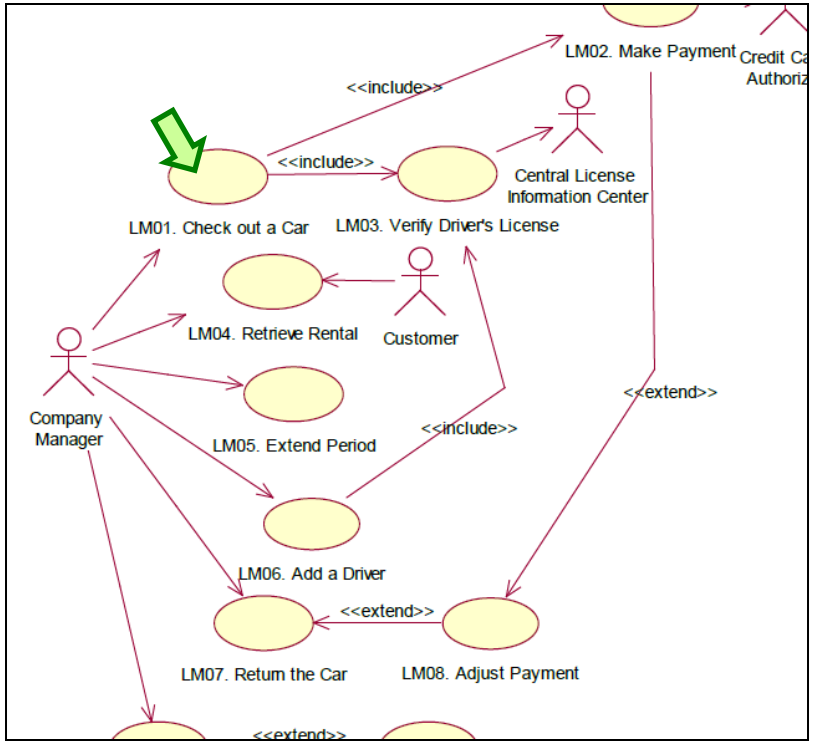
Traceability Engineering

- 'is refined by' Relationship
- 'conforms to' Relationship



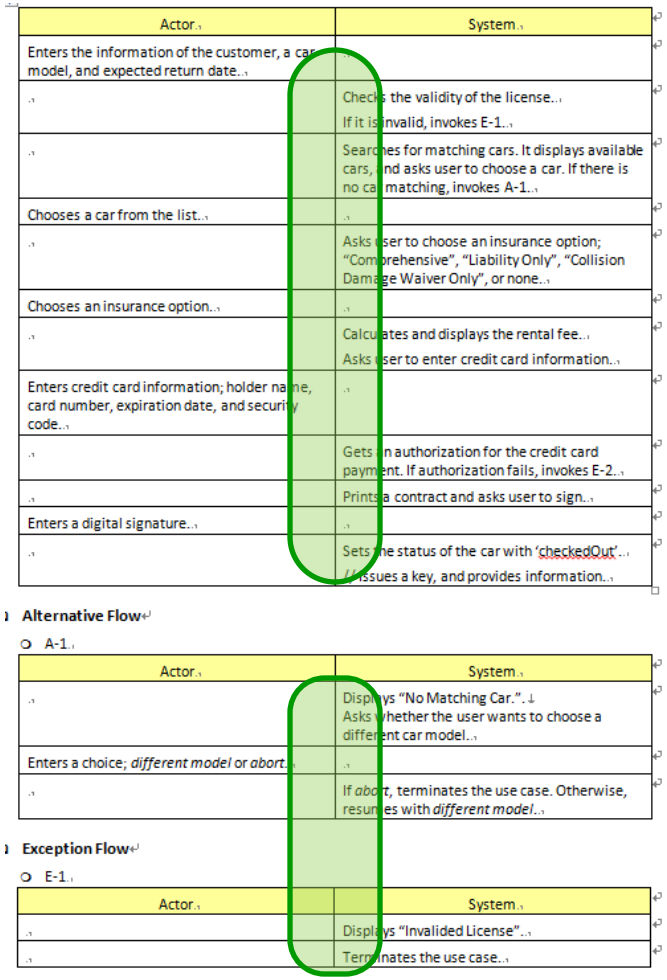
Traceability Case Study (1)

- Car Rental System
- Use Case ‘Check out’



Traceability Case Study (2)

- UC Description for ‘Check Out’
  - Textual Form
  - Natural Language



Actor	System
Enters the information of the customer, a car model, and expected return date.	
	Checks the validity of the license. If it is invalid, invokes E-1.
	Searches for matching cars. It displays available cars, and asks user to choose a car. If there is no car matching, invokes A-1.
Chooses a car from the list.	
	Asks user to choose an insurance option; “Comprehensive”, “Liability Only”, “Collision Damage Waiver Only”, or none.
Chooses an insurance option.	
	Calculates and displays the rental fee. Asks user to enter credit card information.
Enters credit card information; holder name, card number, expiration date, and security code.	
	Gets an authorization for the credit card payment. If authorization fails, invokes E-2.
	Prints a contract and asks user to sign.
Enters a digital signature.	
	Sets the status of the car with ‘checkedOut’. // Issues a key, and provides information.

Alternative Flow

- A-1

Actor	System
	Displays “No Matching Car.”. Asks whether the user wants to choose a different car model.
Enters a choice; *different model* or *abort*.	
	If *abort*, terminates the use case. Otherwise, resumes with *different model*.

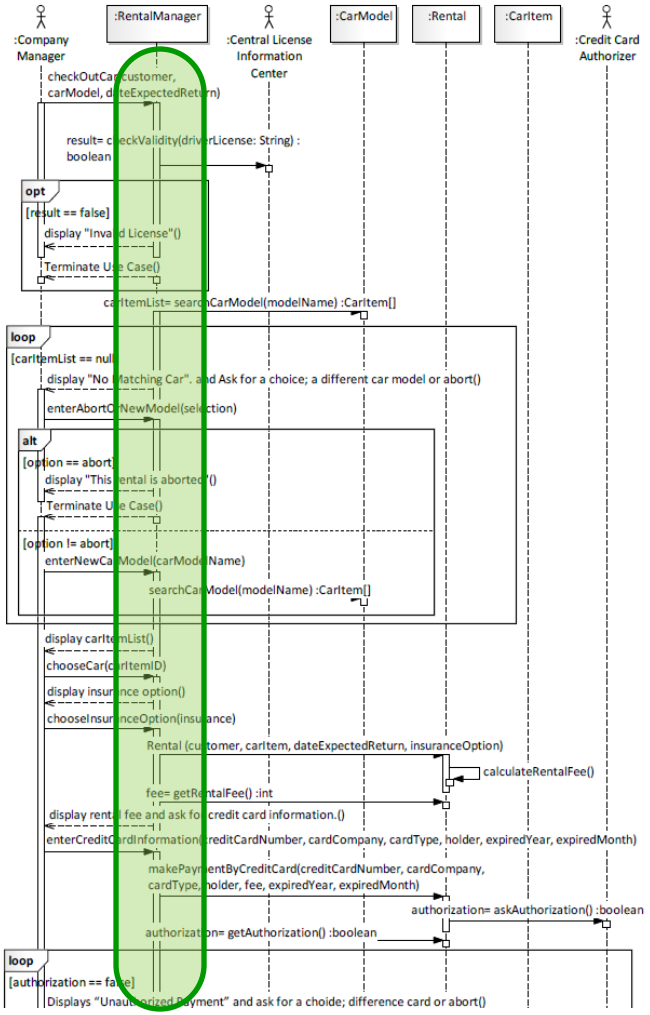
Exception Flow

- E-1

Actor	System
	Displays “Invalided License”.
	Terminates the use case.

Traceability Case Study (3)

- Sequence Diagram for ‘Check Out’
  - Graphical Form
  - Message-Driven



Traceability Case Study (4)

- UC Description derives the workflow in Sequence Diagram.

Actor	System
Enters the information of the customer, a car model, and expected return date.	
	Checks the validity of the license. If it is invalid, invokes E-1.
	Searches for matching cars. It displays available cars, and asks user to choose a car. If there is no car matching, invokes A-1.
Chooses a car from the list.	
	Asks user to choose an insurance option; "Comprehensive", "Liability Only", "Collision Damage Waiver Only", or none.
Chooses an insurance option.	
	Calculates and displays the rental fee. Asks user to enter credit card information.
Enters credit card information; holder name, card number, expiration date, and security code.	
	Gets an authorization for the credit card payment. If authorization fails, invokes E-2.
	Prints a contract and asks user to sign.
Enters a digital signature.	
	Sets the status of the car with 'checkedOut'. // Issues a key, and provides information.

Alternative Flow

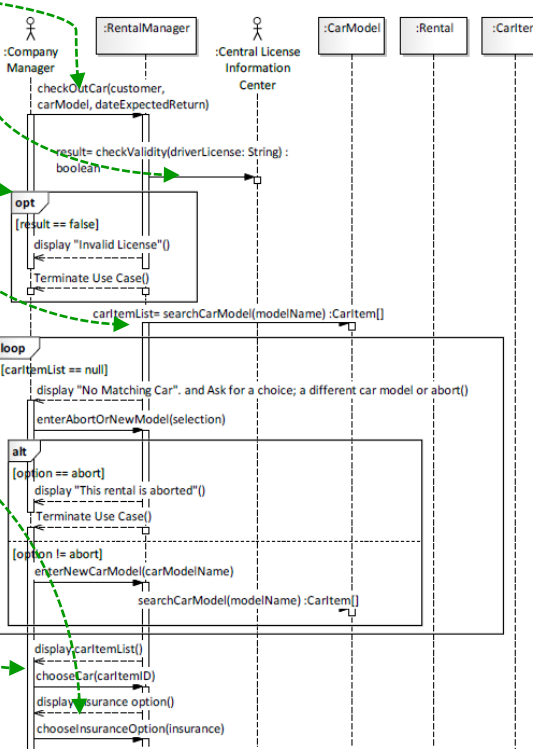
A-1.

Actor	System
	Displays "No Matching Car.". Asks whether the user wants to choose a different car model.
Enters a choice; *different model* or *abort*.	
	If *abort*, terminates the use case. Otherwise, resumes with *different model*.

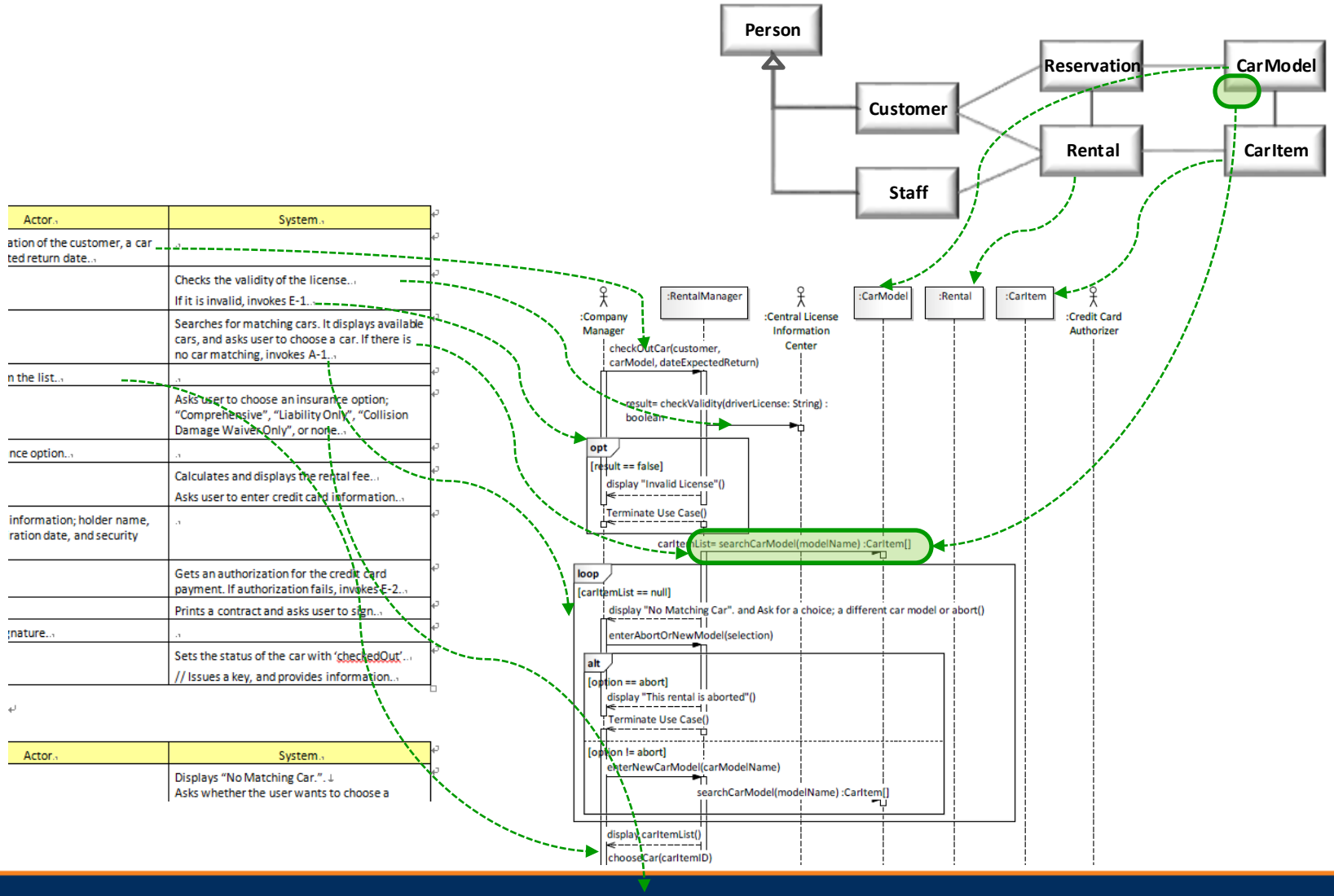
Exception Flow

E-1.

Actor	System
	Displays "Invalided License".
	Terminates the use case.



Traceability Case Study (5)

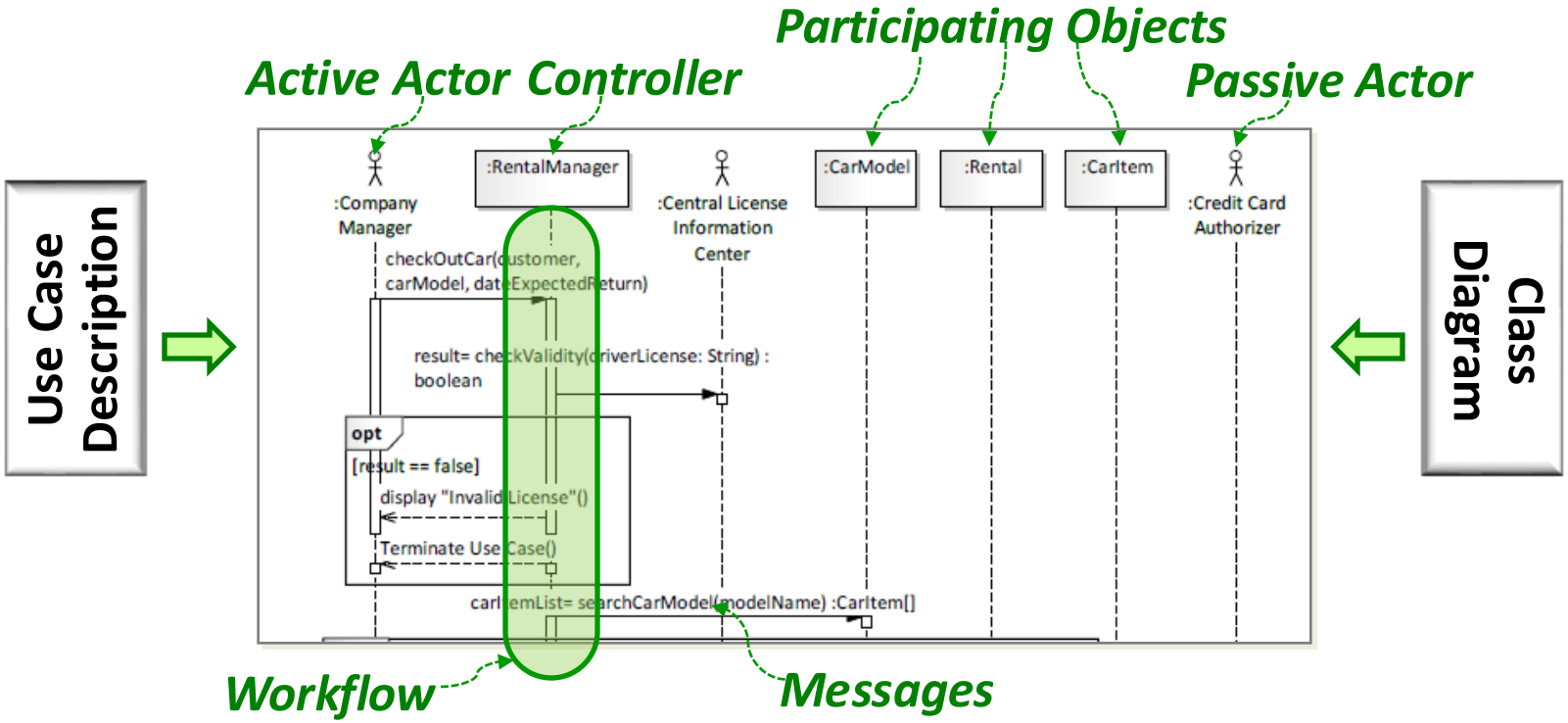


Instruction for Sequence Diagram (1)

- Specify an Active Actor in column #1.
- Define a Controller object in column #2.
  - The Controller does not appear in class diagrams.
    Hence, create a controller with a meaningful name.
- Add Participating Objects in the next columns.
  - Participating Object Types (i.e. classes) must exist in the class diagram.
- Add any Passive Actor in appropriate columns.
  - Typically next to an object interacting with this actor.
  - Example) Credit Card Authorization (CCA) system

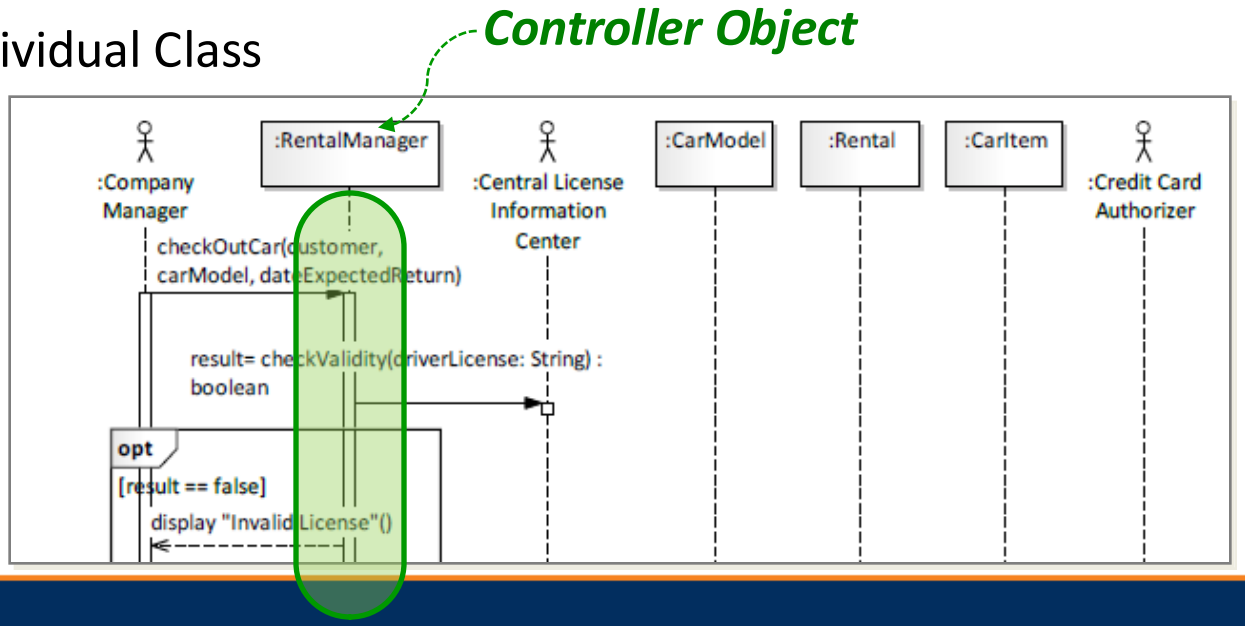
Instruction for Sequence Diagram (2)

- Specify the Workflow in message passing form.
  - Conformance to use case description
  - Messages should correspond to the operations of the object.



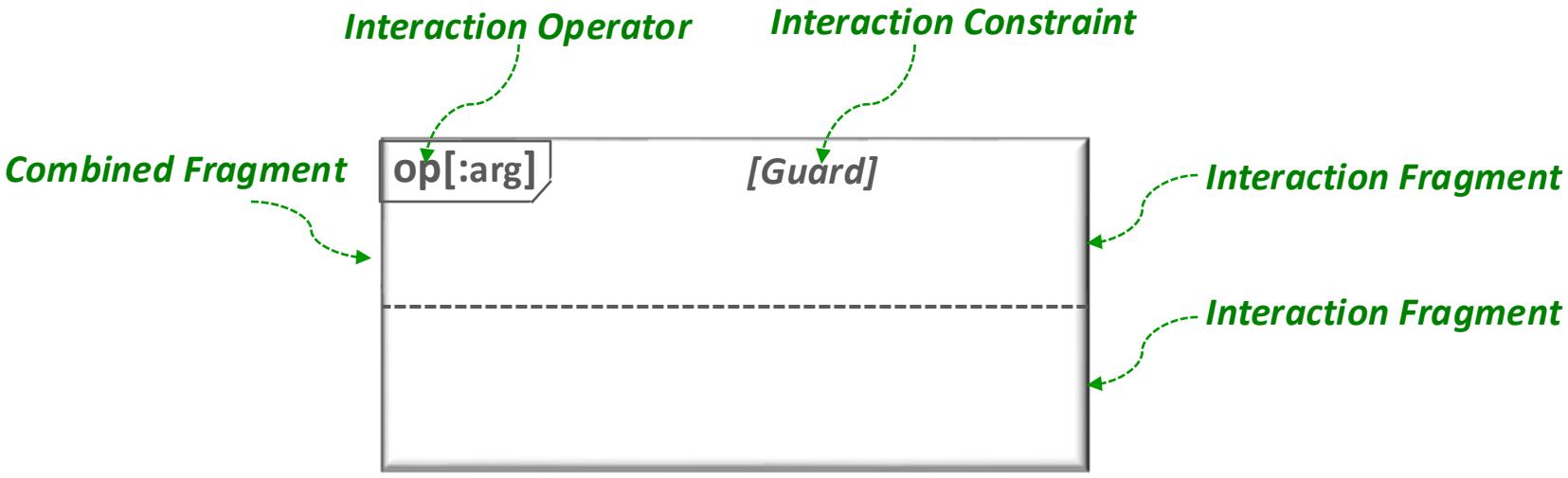
Why include ‘Controller’ ?

- Controller takes the whole responsibility of executing the workflow.
- Participating objects are not aware of the workflow.
- Good for Separation of Concerns
  - Concern of Workflow
  - Concern of Individual Class



Combined Fragment

- *Combined Fragment* is a group of interaction fragments.
  - *Interaction Fragment* is a unit of an interaction, i.e. sequence of messages.
- *Combined Fragment* has an interaction operator and corresponding interaction operands.
- Interaction Operators
  - Alt, opt, loop, break, par, critical, strict, seq, ignore, consider, assert, neg

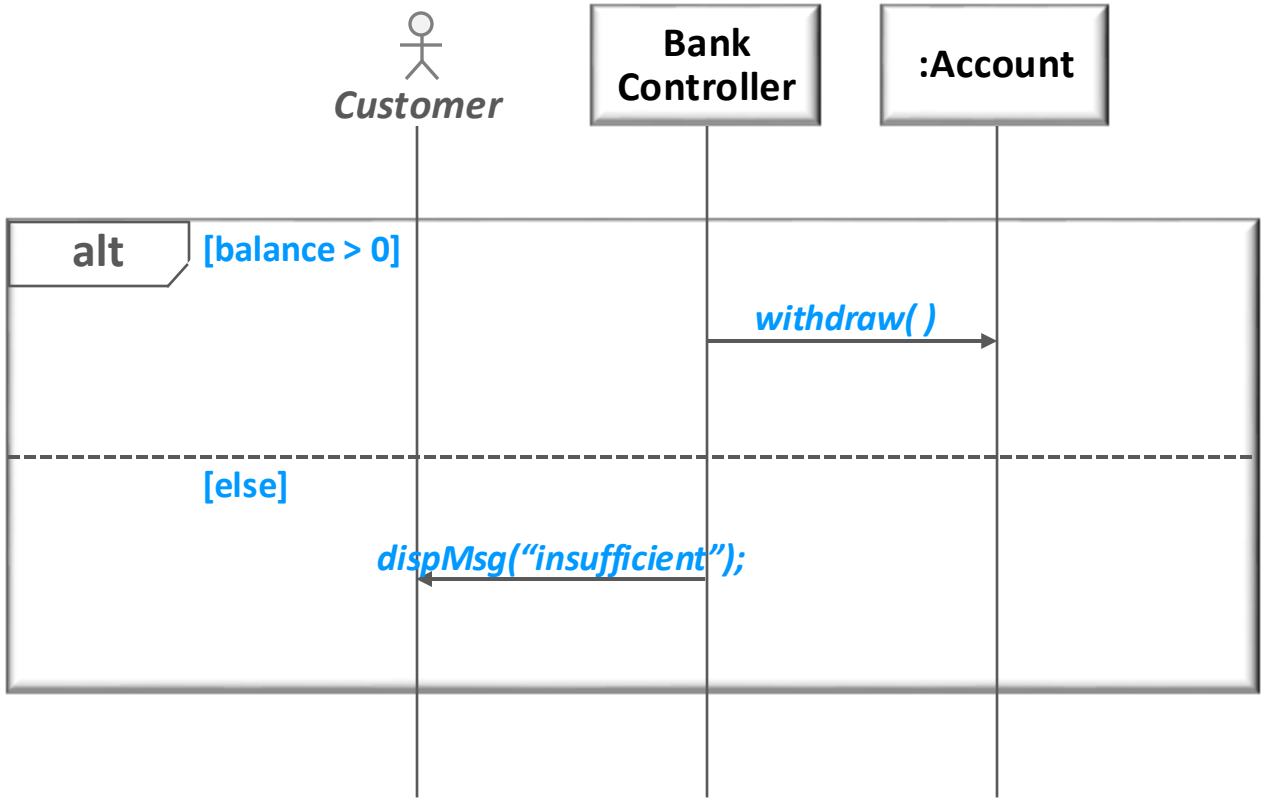


‘alt’ for Alternatives (1)

- To represent a choice of behavior
  - Like *if-then-else*, *switch-cases*
- At most one of the operands will be chosen.
- An operand must have an explicit or implicit guard expression.
- Operand with No Guard
  - True Condition is assumed.
- Operand with [else]
  - Represents a guard that is the negation of the disjunction of all other guards.

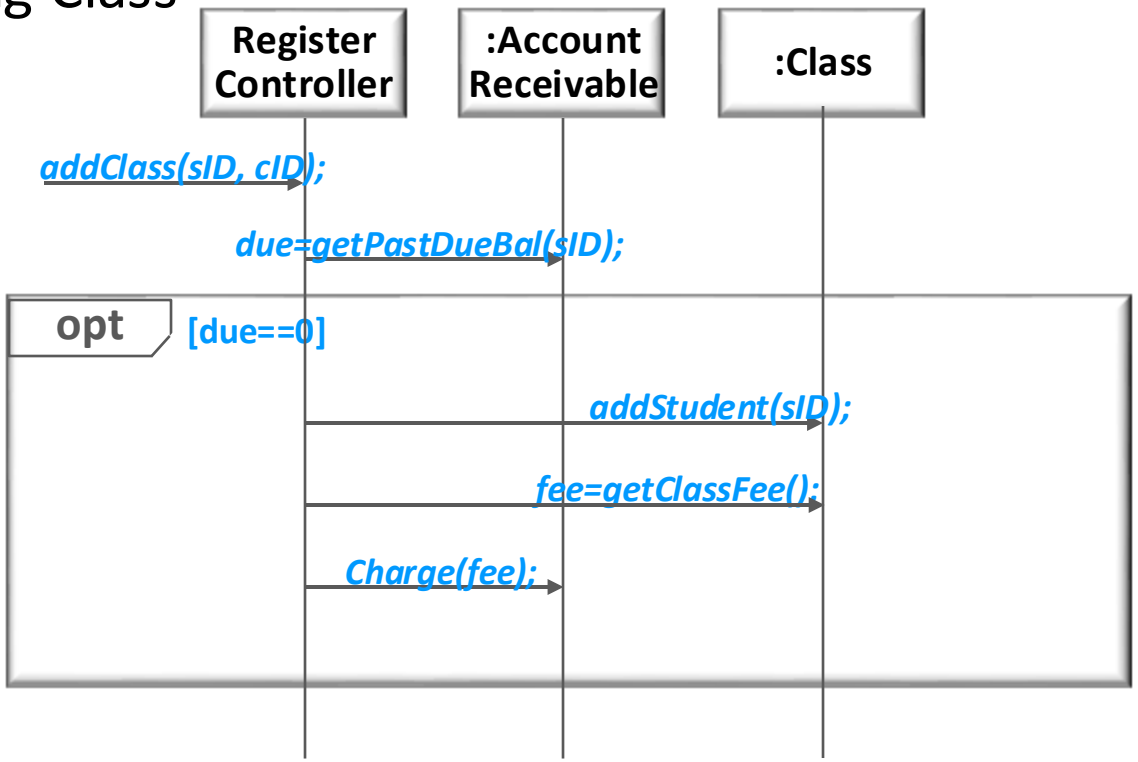
‘alt’ for Alternatives (2)

- Example: Withdraw for Account



‘opt’ for Option

- To represent an optional behavior.
  - Like *if-then*
- Example: Registering Class

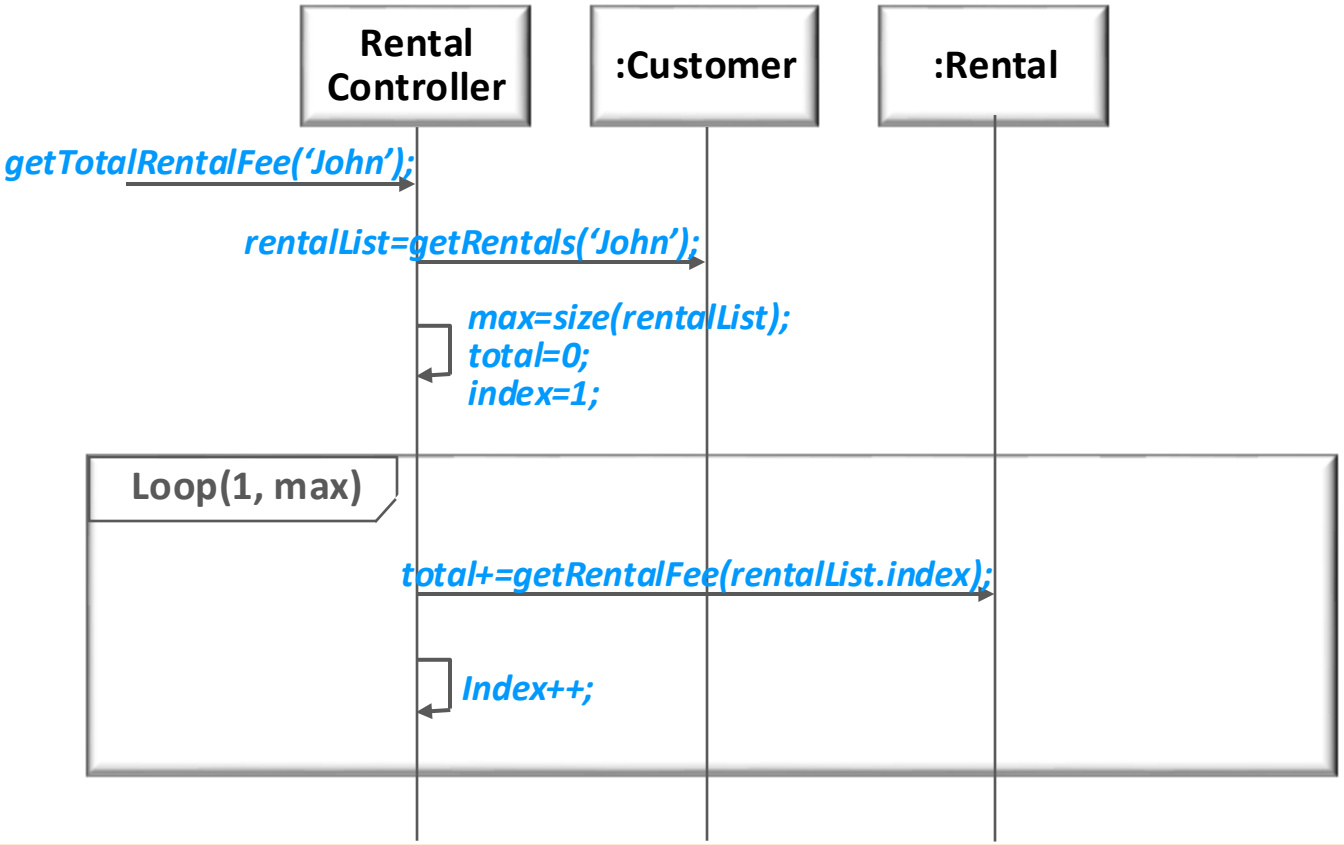


‘loop’ for Loop (1)

- To represent a loop of an interaction a number of times
- Notation
  - <Loop Operand> ::= *‘loop’* ['(' <min-int> [',' <max-int>] ')']
  - <min-int> ::= non-negative-integer
    - The loop repeats at least this number of times.
  - <max-int> ::= positive-integer | '*'
    - The loop does not repeat more than this number of times.
  - [guard]
    - After the <min-int> iterations, the condition is tested before each additional loop iteration.
    - If the condition is false, then the loop is abandoned.

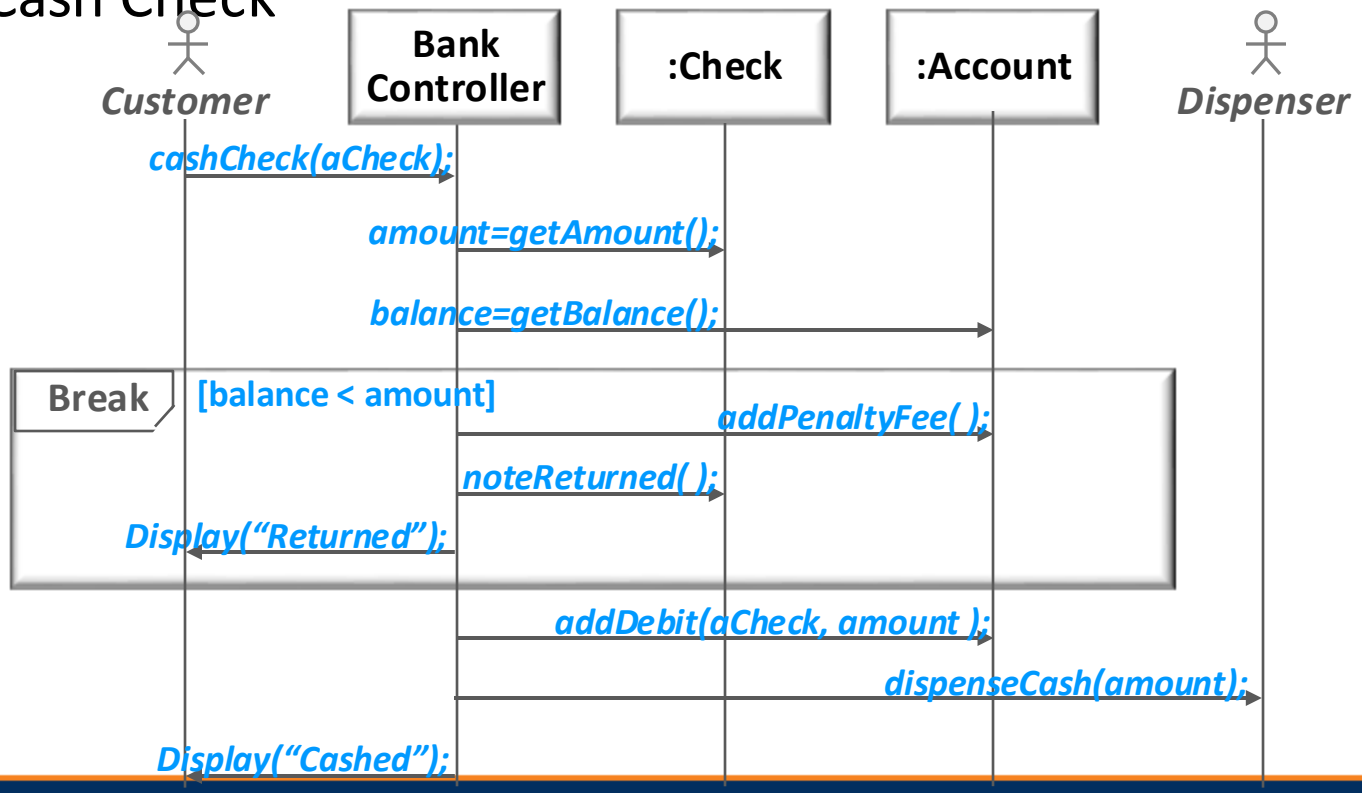
‘loop’ for Loop (2)

- Example: Compute Average Rental Fee



‘break’ for Break

- To represent a breaking scenario that is performed instead of the remainder of the Interaction fragment
- Example: Cash Check

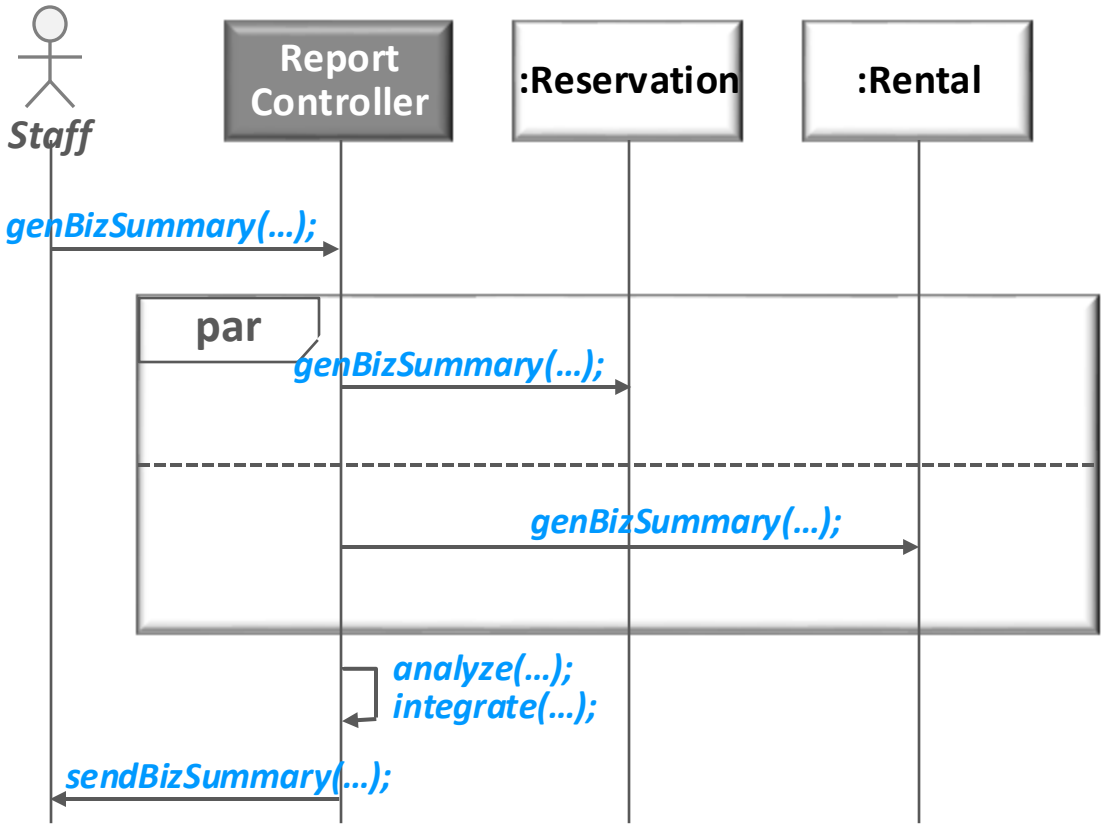


‘par’ for Parallel (1)

- To represent a parallel processing
  - A task is split into sub-tasks that execute simultaneously.
- Granularity of Task
  - Thread level
  - Process level
  - Processor Core level
  - Processor level
  - Node level
- Useful for processing Big Data/Contexts

‘par’ for Parallel (2)

- Example) Generate Business Summary



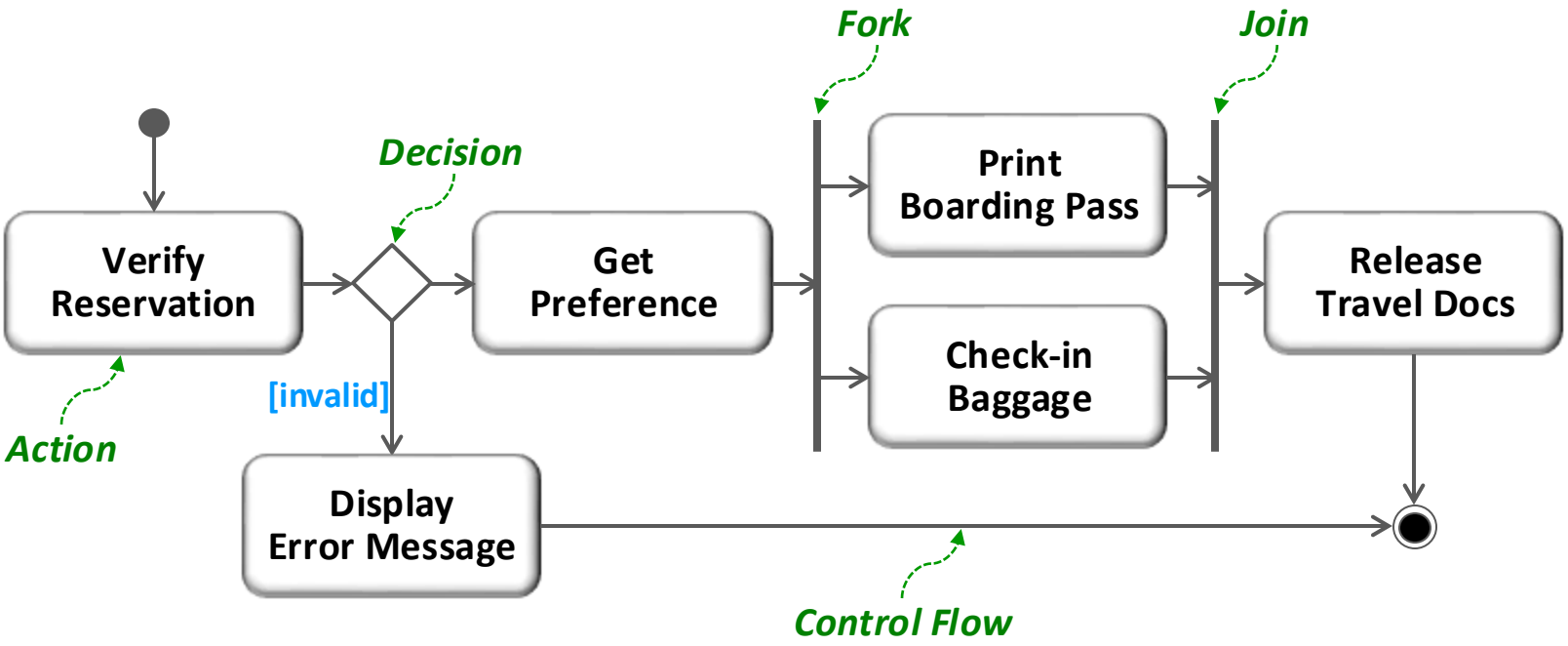
Activity Diagram

Overview of Activity Diagram (1)

- To model the dynamic aspect of the system, focusing on the activities among objects
- Usages
  - To model the flow of a use case in details
    - Similar to Sequence Diagram
  - To model the flow among use cases
  - To model the details of an algorithm or an operation

Overview of Activity Diagram (2)

- Example: Check In for Flight

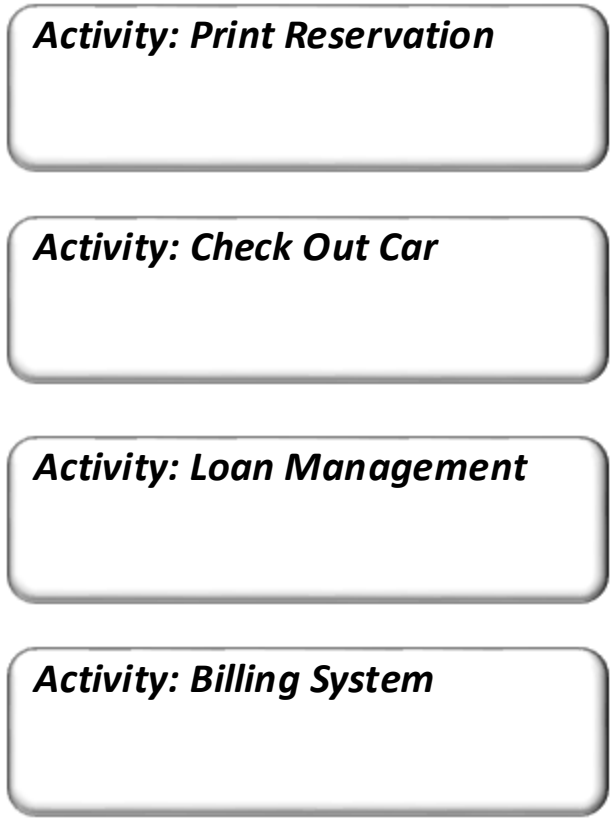


Key Elements of Activity Diagram

- Activity
- Action
- Initial Node, Final Node
- Control Flow
- Object, Object Flow
- Data Store
- Decision and Merge
- Fork and Join
- Exception Handler
- Partition
  - Swimlane

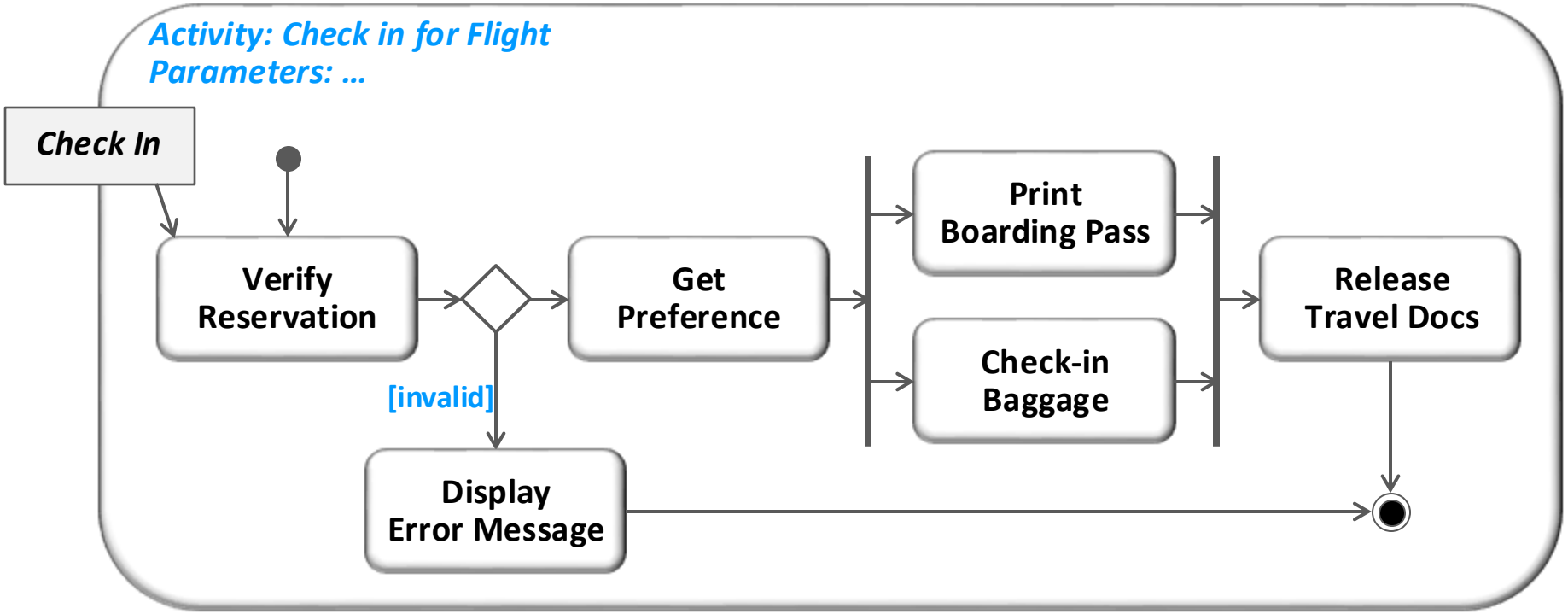
Activity (1)

- To represent a (parametrized) sequence of behavior.
- Granularity of Activity
  - Varies, from an operation to an entire system



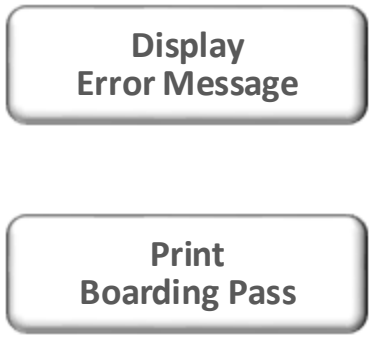
Activity

- Example: Check in for Flight



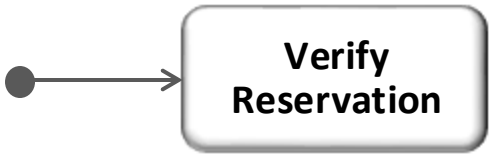
Action

- To represent a single step within an activity
- Atomic Operation
  - Either Completed or Aborted

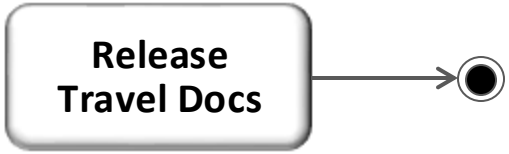


Initial Node, Final Node

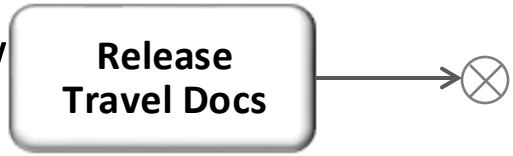
- Initial Node
  - To represent where the flow starts



- Final Node
  - Activity Final Node
    - To represent an end of all control flows in an activity

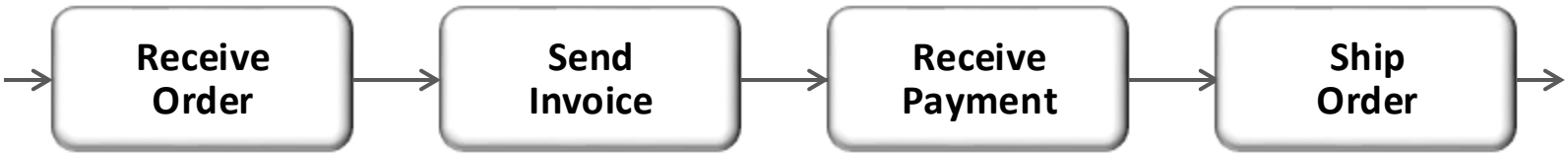


  - Flow Final Node
    - To represent the end of a single control flow



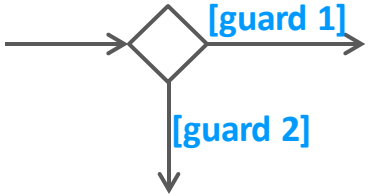
Control Flow

- The flow of control from *one action* to the next.
- Objects and data cannot pass along a control flow.

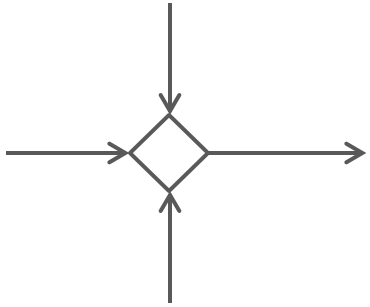


Decision and Merge

- Decision Node
  - To choose one among outgoing flows
  - Each outgoing flow has a guard condition.
  - Guard conditions must be mutually exclusive.

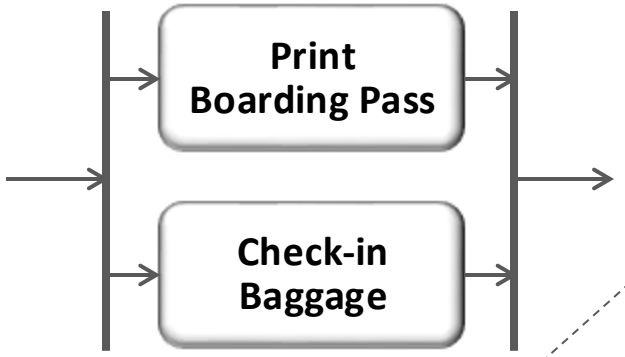


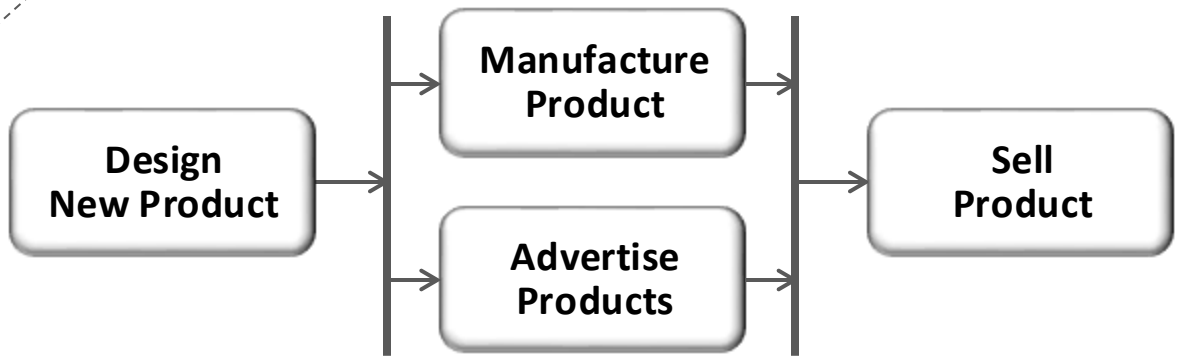
- Merge Node
  - Multiple incoming flows are merged into an outgoing flow.



Fork and Join

- Fork Node
  - To represent the start of concurrent threads of control
- Join Node
  - To represent the end of concurrent threads of control



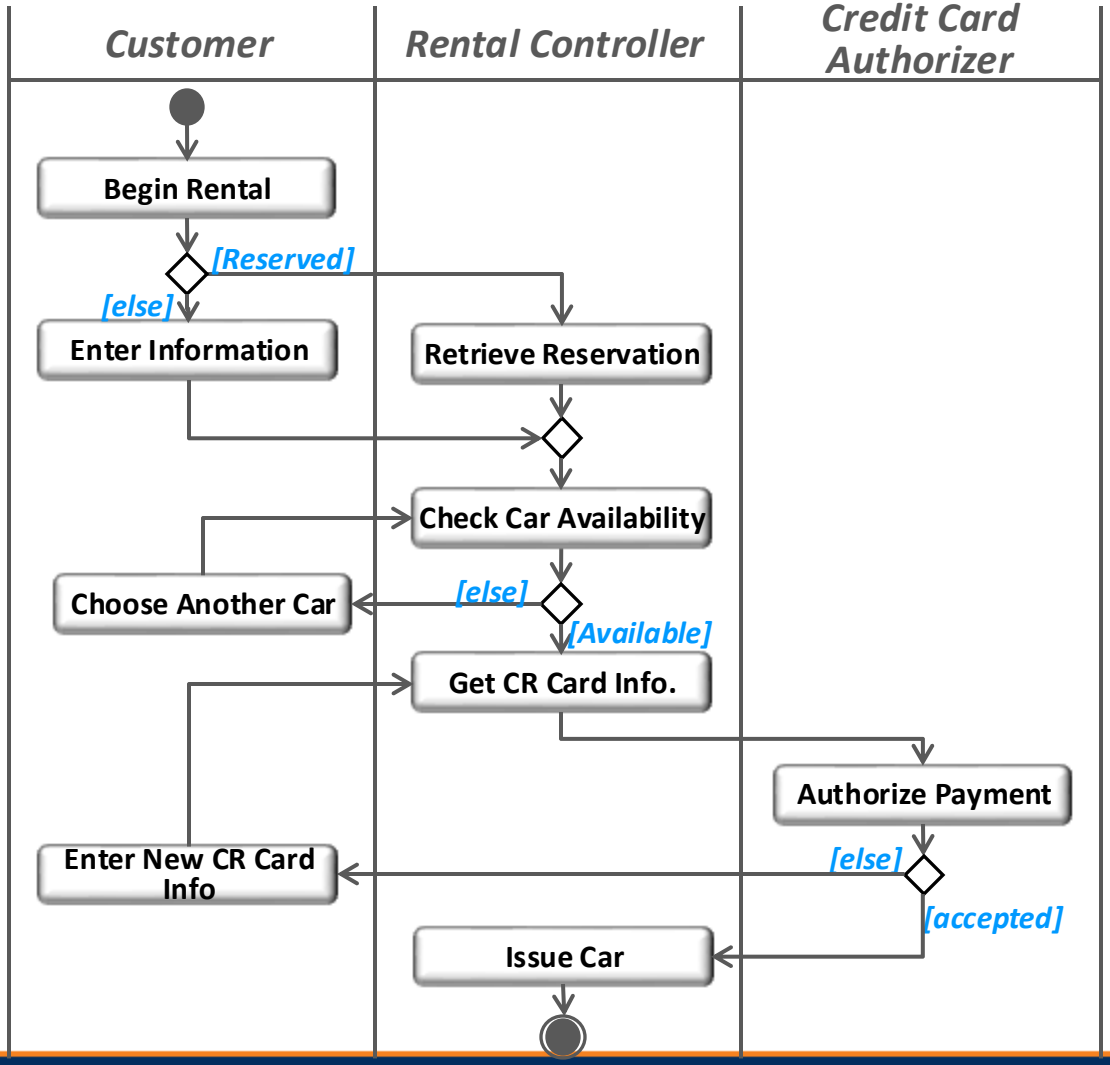


Partition (1)

- To separate actions within an activity into those performed by different participants
- Participants
  - Class, Component, Sub-system, Business Unit, Role, Machine
- Can be shown as Horizontal or Vertical Partititions
- Also called, ‘Swimlane’

Partition (2)

- Horizontal Partitions or Vertical Partitions
- Example)
  - Check Out Car



CPSC 362
Software Engineering

State Machine Diagram

What is a State Machine Diagram?

- **Definition**: A State Machine Diagram is a behavioral model used to represent the different states of an object in a system and the transitions between those states.

- **Used In**:
  - Software design
  - Modeling system behavior
  - Event-driven systems

- **Key Concepts**:
  - **State**: Represents the condition of an object at a given time.
  - **Transition**: Movement from one state to another, triggered by an event.

Why Use State Machine Diagrams?

- **Clarify Object Behavior**: Helps to model how an object changes over time in response to external events.
- **Event-Driven**: Useful in systems where changes are triggered by user actions or system events.
- **Enhanced Understanding**: Provides clear visual documentation of complex systems.

Components of a State Machine Diagram

- **States**: Represent the current status of the system or an object.
  Example: "Logged In", "Idle", "Processing"
- **Transitions**: The movement from one state to another, triggered by events. Example: From "Idle" to "Processing" after a "Start Task" event.
- **Events**: Triggers that cause transitions between states. Example: User clicking "Submit" or receiving a network response.
- **Initial State**: The state where the system starts. Example: "Idle" at system startup.
- **Final State**: Represents the end of the lifecycle of the state machine. Example: "Completed" or "Terminated"

Steps to Create a State Machine Diagram (Step 1)

- **Step 1: Identify the Object/Entity**
  - Choose the object whose behavior you want to model.
    Example: A user account, a process in software, or a vending machine.
- **Questions to Ask**:
  - What does this object represent?

Steps to Create a State Machine Diagram (Step 2)

- **Step 2: Define Possible States**
  - Identify all the possible states the object can be in. Example:
    - For a User Account: "Logged Out", "Logged In", "Suspended"
    - For a Vending Machine: "Idle", "Selecting", "Processing", "Dispensing"

Steps to Create a State Machine Diagram (Step 3)

- **Step 3: Identify Transitions**
  - Define the events that trigger a change between states.
    Example:
    - "Login" event triggers the transition from "Logged Out" to "Logged In".
    - "Cancel" event transitions from "Selecting" to "Idle".

Steps to Create a State Machine Diagram (Step 4)

- **Step 4: Define Events**
  - Outline the specific events that cause transitions. Examples:
    - A user action, such as clicking a button.
    - A system event, such as a timeout or an error.

Steps to Create a State Machine Diagram (Step 5)

- **Step 5: Add Initial and Final States**
  - Set the starting point and potential end points for your system.
    - **Initial State**: Where the system starts.
    - **Final State**: Where the system ends or terminates.

Example: State Machine for a Vending Machine

- **States**:
  - Idle
  - Selecting
  - Processing
  - Dispensing
- **Transitions**:
  - From "Idle" to "Selecting" after item selection.
  - From "Selecting" to "Processing" after confirming payment.
  - From "Processing" to "Dispensing" when the item is ready.

Key Takeaways

- State Machine Diagrams model the lifecycle of objects.
- They are useful for visualizing event-driven systems.
- Focus on states, transitions, events, and the start/end points.

Data Flow Diagram

Data Flow Diagram (DFD)

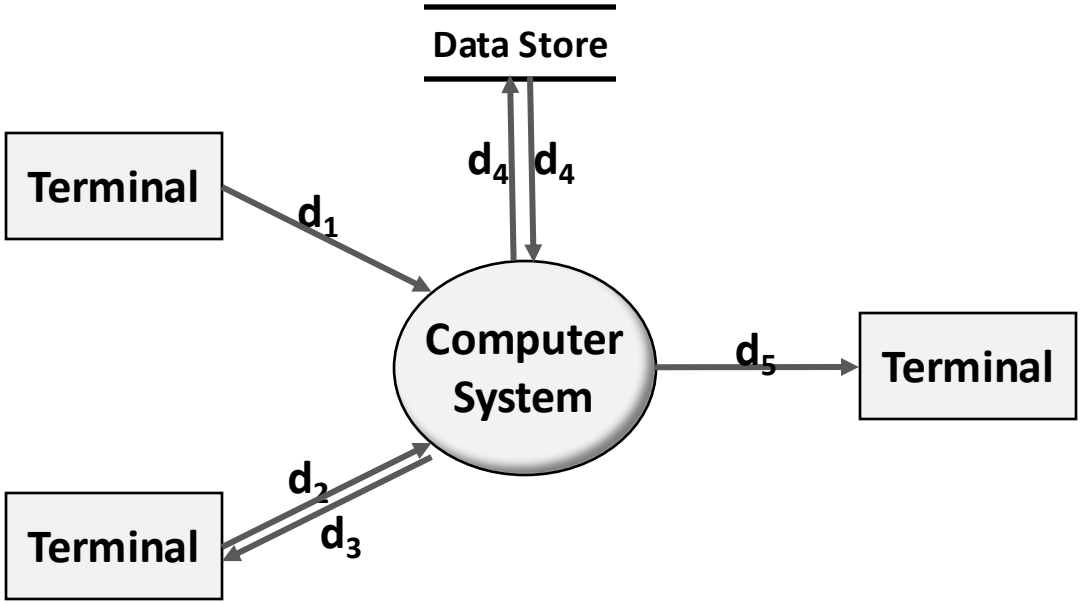
- DFD or Bubble Chart
  - Depicts information flow and the transformation that are applied as data move from input to output.
- 4 Major Components
  - Terminal (External Entity)
    - A producer or consumer of information

  - Process
    - A transformer of information (a function)

  - Data Flow (Data Objects)
    - Arrowhead indicates the direction of Input and Output Data

  - Data Store
    - A repository of data that is to be stored for use by processes.
    - May be as simple as a buffer or as sophisticated as a database.

Data Flow Diagram (DFD)



Terminal (External Entity)

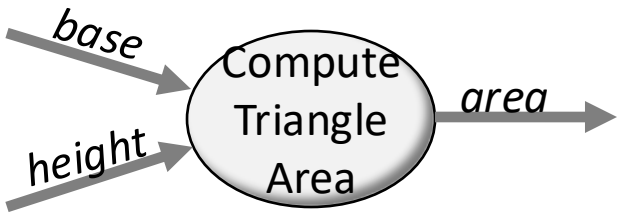
- Producer or Consumer of Data
- Terminal can be;
  - User
    - Consumer, Staff
  - External System
    - Credit Card Authorizer
  - Hardware Device
    - Sensor, Actuator

Process

- Functionality to Manipulate Data
  - Receive input, manipulate, and returns an output.
- Examples
  - Compute Tax
  - Determine Face Emotion
  - Generate Report
  - Deposit Money

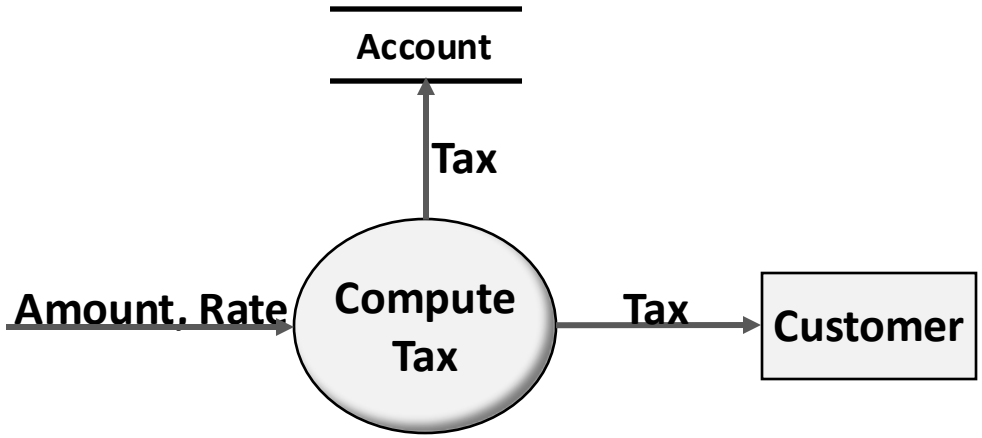
Data Flow

- Flow of Data
- Represented with Arrow and Data Name



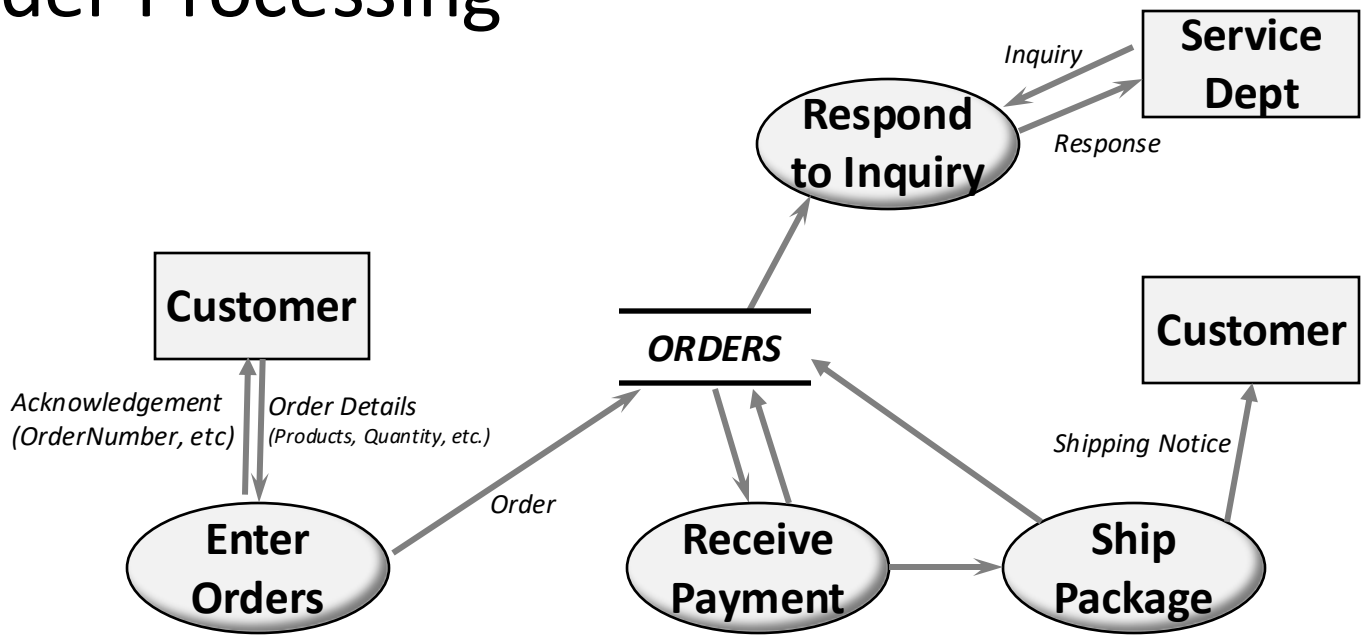
Data Stores

- Represents a long-lasting data.
  - Buffer in Main Memory
  - Database Table



DFD Example

- Order Processing



DFD Leveling

DFD Levels

- A DFD may be partitioned (extended) into levels that represent increasing information flow and functional detail.

Level 0 DFD

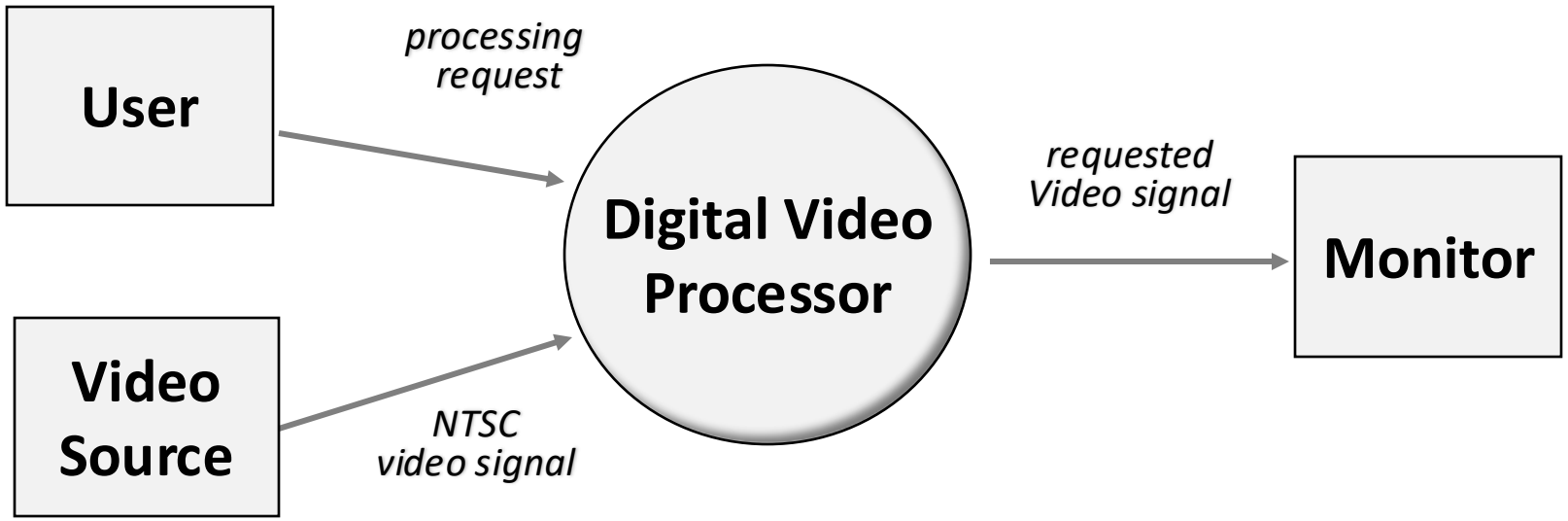
- Fundamental System Model, Context Model/Diagram
- Represents the retire software as a single bubble with input and output data indicated by incoming and outgoing arrows.

Information Flow Continuity (Balancing)

- Input and output to each refinement must remain the same.
- Produce consistent models.

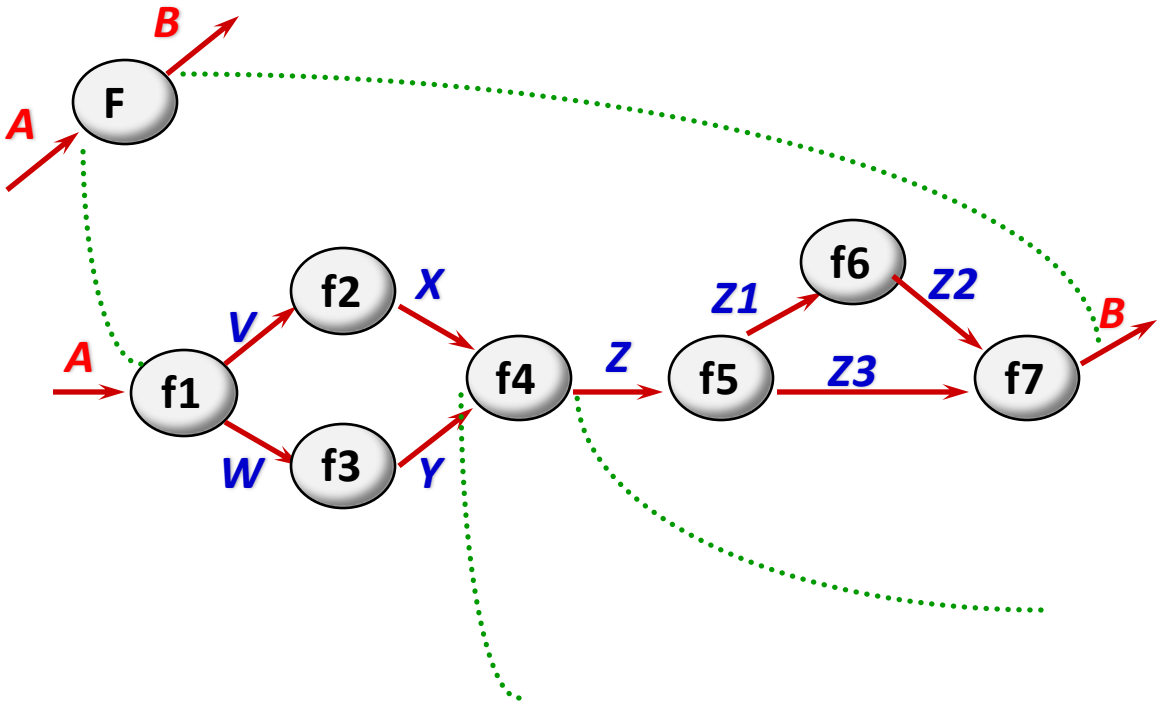
DFD Leveling

- Level 0, Context Diagram



DFD Leveling

- Balancing

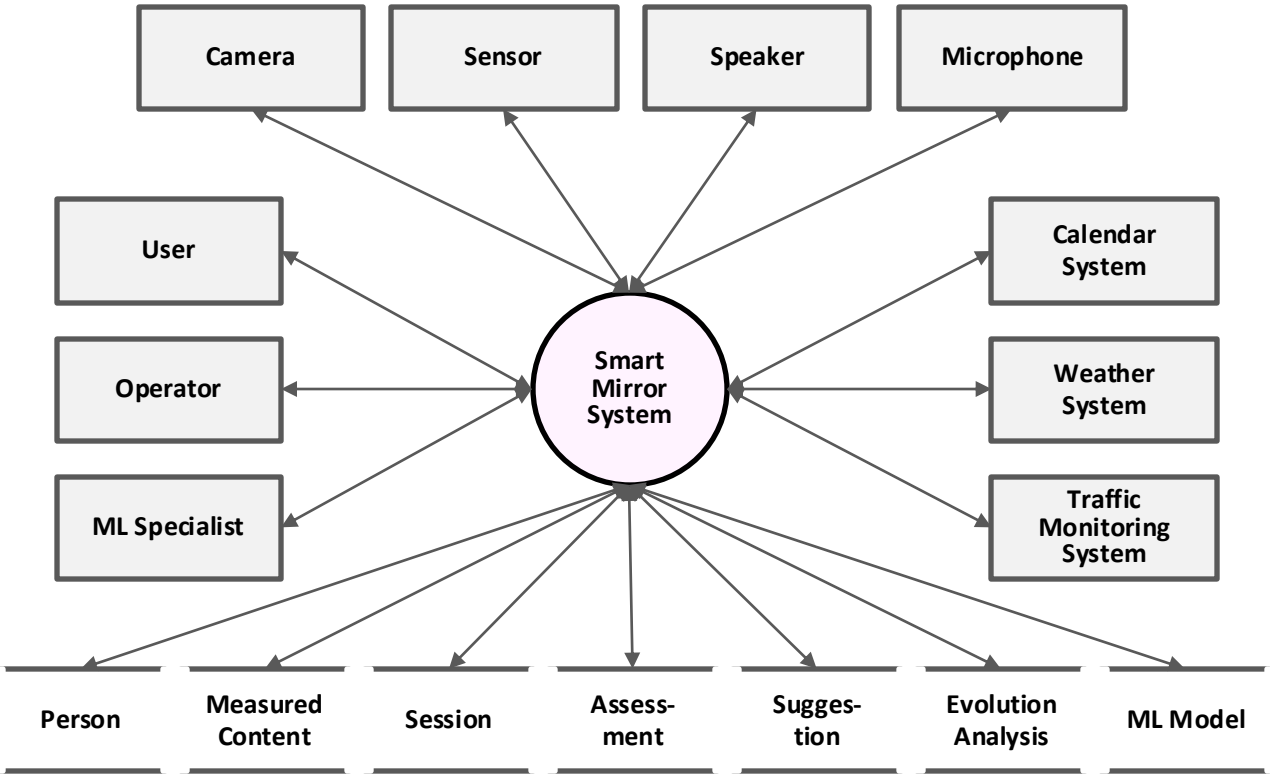


Case Study: Smart Mirror System

- Mirror Functionality
- Touch Screen
- Face Recognition with Camera
- Public Information
  - Weather
  - Traffic
- Private Information
  - Calendar
- Age Determination
- Emotion Detection

Case Study: Smart Mirror System

- DFD



CPSC 362
Software Engineering

Transforming Design into Implementation

Design Artifacts of OOAD

- Functional View Design
  - Use Case Diagram for Functionality
  - Component Diagram for Functional Component
- Information View Design
  - Class Diagram for Persistent Datasets
  - Component Diagram for Data Components
- Behavior View Design
  - Activity Diagram for Overall Workflow/Control Flow
  - Sequence Diagram for Message Flows among Objects
  - State Machine Diagram for State-dependent Behavior

Programs in C++

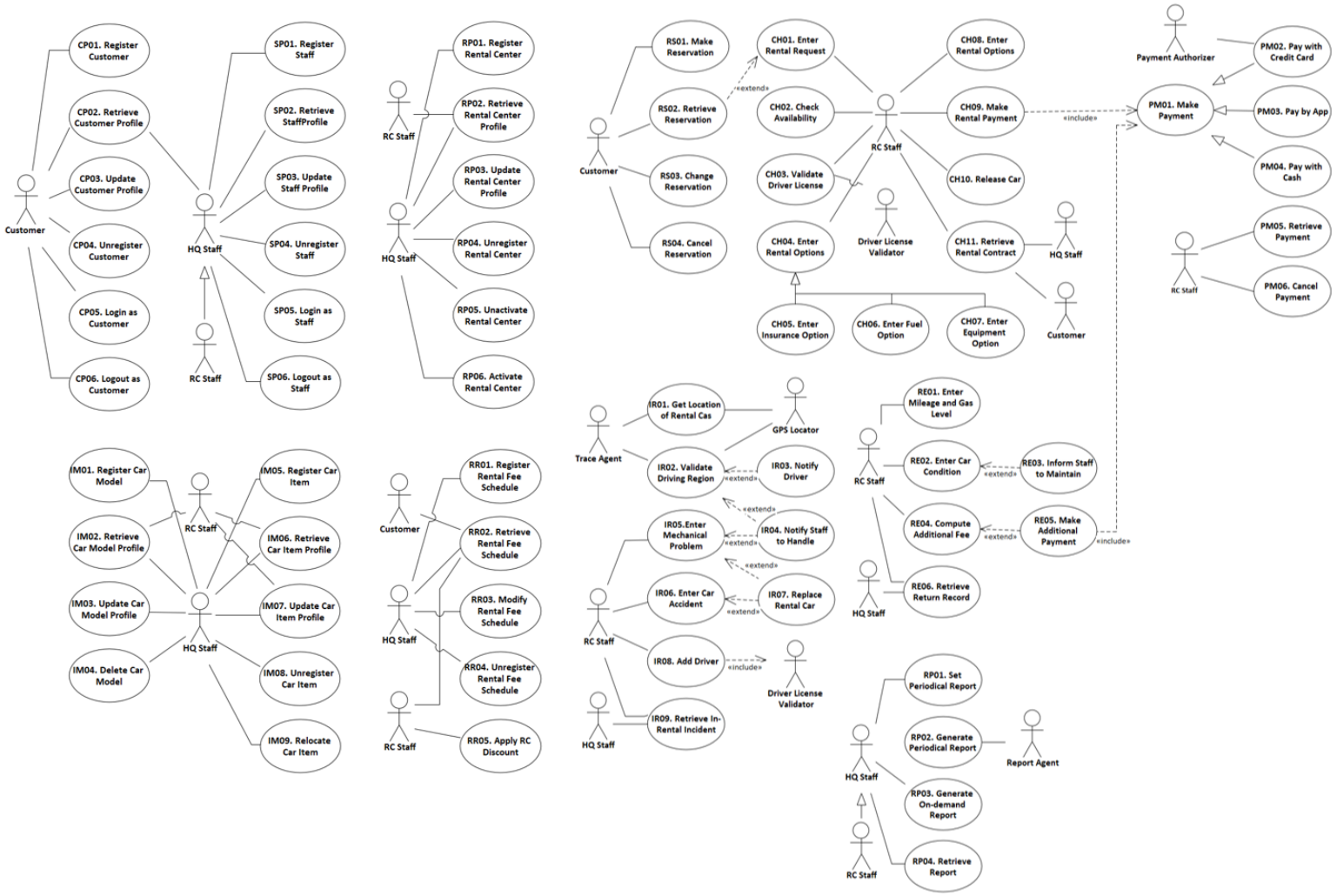
- C++ is a hybrid language.
- Procedural Programming
  - C++ inherits its procedural roots from C, which is fundamentally a procedural language.
  - Programming paradigm focused on executing a sequence of instructions or statements to achieve a specific task, organized into functions.
- Object-Oriented Programming (OOP)
  - C++ introduced object-oriented features beyond C language constructs.
  - Encapsulation with Class
  - Inheritance
  - Polymorphism

Programs in Java

- Java is a pure object-oriented programming language.
- Java Virtual Machine (JVM)
  - Write Once, Run Anywhere!
- Supports all OOP principles
- Platform Independence
- Simplicity
  - Easy to learn and use, effectively managing the complexity of other powerful languages.
  - It eliminates complex features of C/C++.
- Multithreading built-in

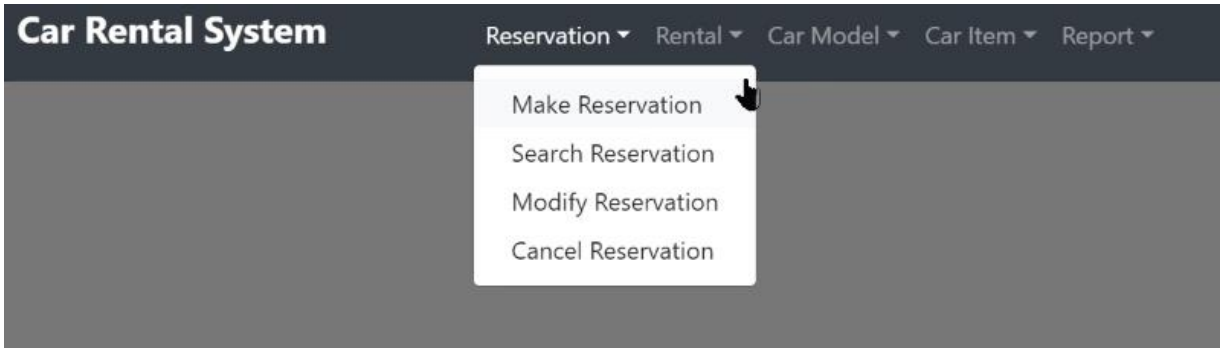
Implementing Functional View Design

Implementing a Use Case Diagram

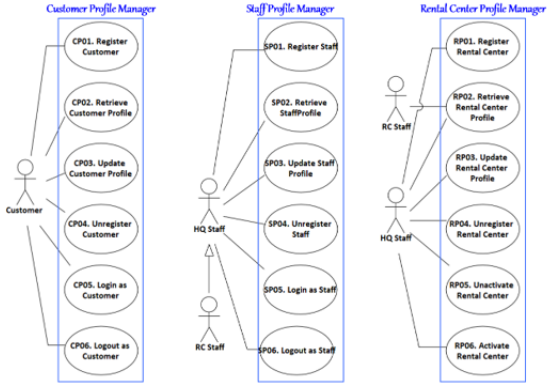


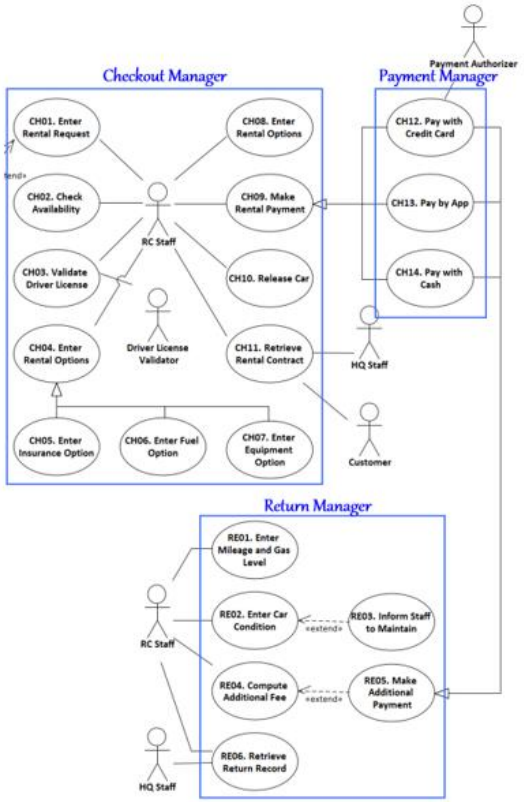
Basis for Developing User Interface

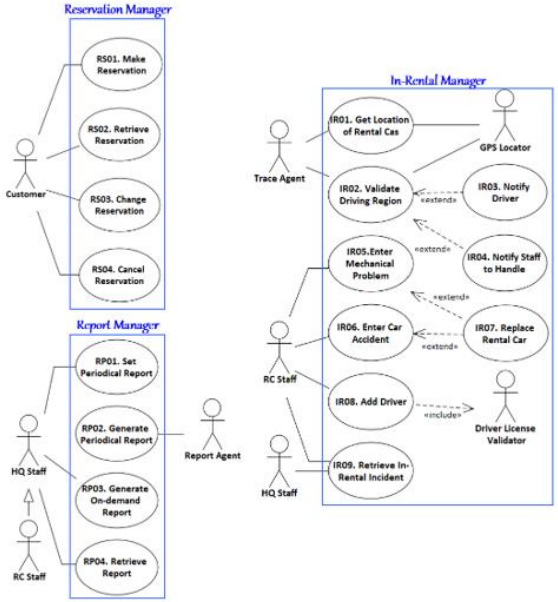
- The functional groups of the system become the main menu options.
  - Use cases in each functional group become the sub-menu options.
- A Use Case Model corresponds to a system menu.



Basis for Deriving Functional Components

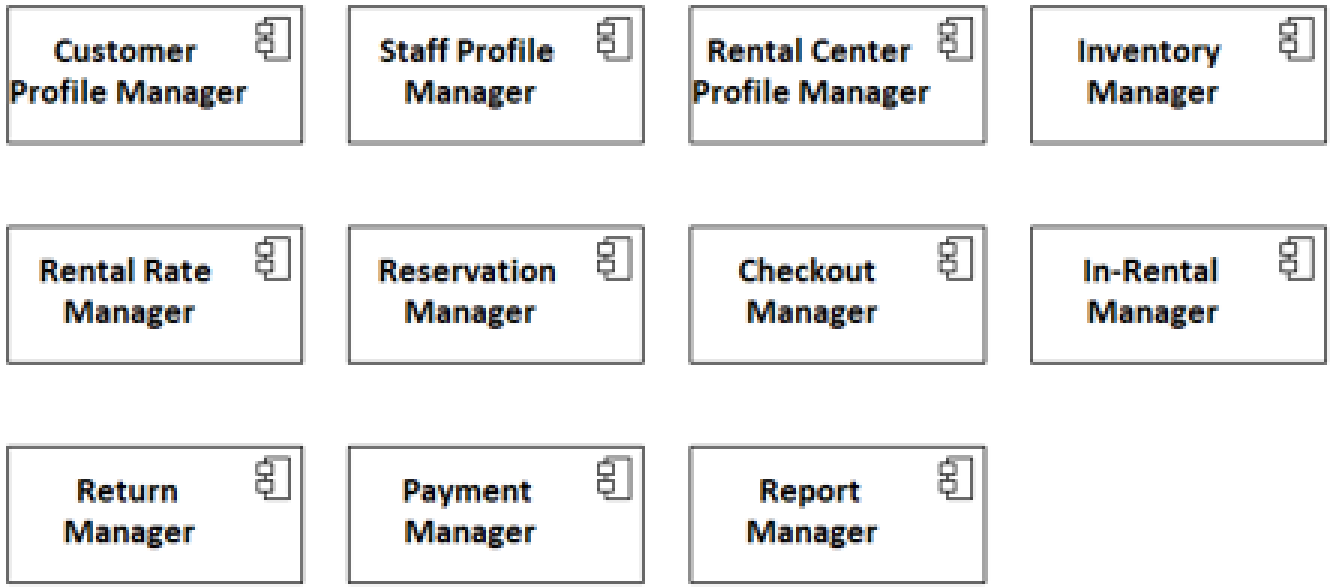




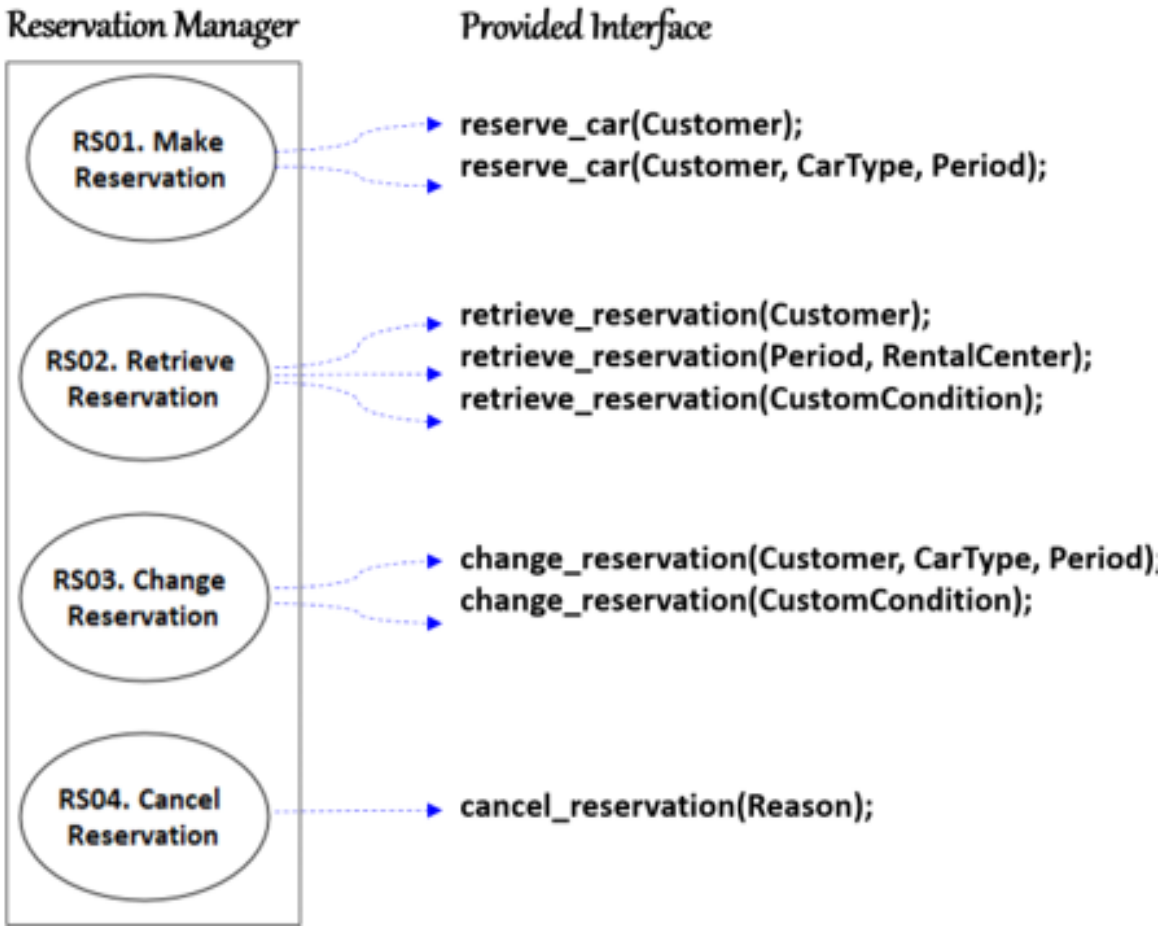


Basis for Deriving Functional Components

- Functional Components in Component Diagram



Design Interfaces of Functional Components



Implement Functional Components

```
public interface iReservationManager {
    void reserve_car(Customer customer);
    void reserve_car(Customer customer, CarType carType, Period period);
    void retrieve_reservation(Customer customer);
    void retrieve_reservation(Period period, RentalCenter rentalCenter);
    void retrieve_reservation(CustomCondition customCondition);
    void change_reservation(Customer customer, CarType carType, Period period);
    void change_reservation(CustomCondition customCondition);
    void cancel_reservation(Reason reason);
}

public class ReservationManager implements iReservationManager {
    public void reserve_car(Customer customer) {
      // Implementation code
   }
    public void reserve_car(Customer customer, CarType carType, Period period) {
      // Implementation code
   }
    public void retrieve_reservation(Customer customer) {
      // Implementation code
   }
    public void retrieve_reservation(Period period, RentalCenter rentalCenter) {
    …
```

Implementing Information View Design

Implementing Class Diagram

- Translate the structure and relationships in a class diagram into Java code.
- **Classes**
  - For each class in the diagram, define a Java class with the class name. If the class is meant to be a base class that should not be instantiated, you may consider making it abstract.

```
public class ClassName {
    …
}
```

- **Add Attributes**
  - For each attribute in a class, define a corresponding variable within the class.

```
public class ClassName {
    private DataType attributeName;
    // …
}
```

Implementing Class Diagram

- Methods
  - Define a corresponding method within the class.
  - Determine the method's return type, name, and parameters.

```
public class ClassName {
    // ...

    public ReturnType
methodName(ParameterType parameter) {
        // method body
    }
}
```

Implementing Class Diagram

- Relationships
  - Associations
    - If there's an association between two classes, one class will typically hold a reference to the other as an attribute.

```
public class Student {
    private Teacher scienceTeacher; }
public class Teacher {
    private List<Student> students;}
```

  - Aggregations and Compositions ('has-a' relationship)
    - Manage the lifecycle of the referenced objects more carefully, especially in the case of composition where the lifecycle of the contained object is controlled by the container object

```
public class Engine { // Engine specific details }
public class Car {
    private Engine engine = new Engine(); }
```

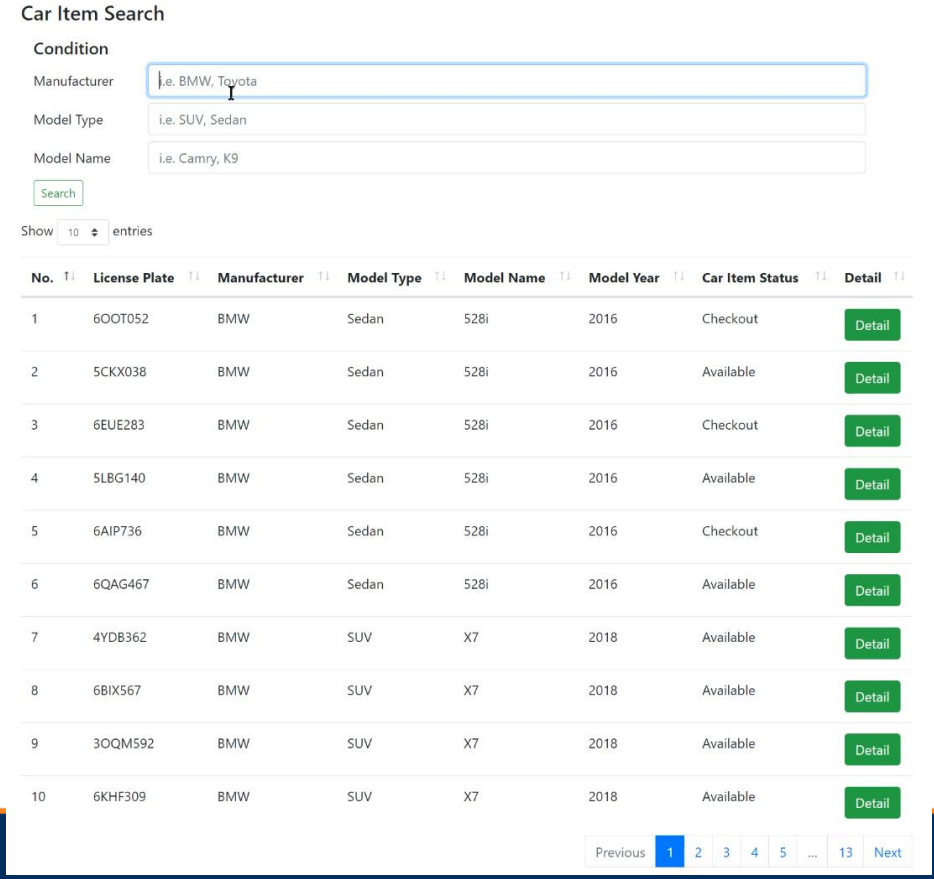
  - Inheritance ('Is-a' relationship)
    - If a class is a subclass of another, use the extends keyword to inherit from a superclass.

```
public class Bicycle extends Vehicle {
   // additional attributes and methods
}
```

Object-to-Relational (OR) Mapping

- Design a relational database for the persistent object classes.
- Choose a R-DBMS such as MySQL.
- Example) Table CarItem

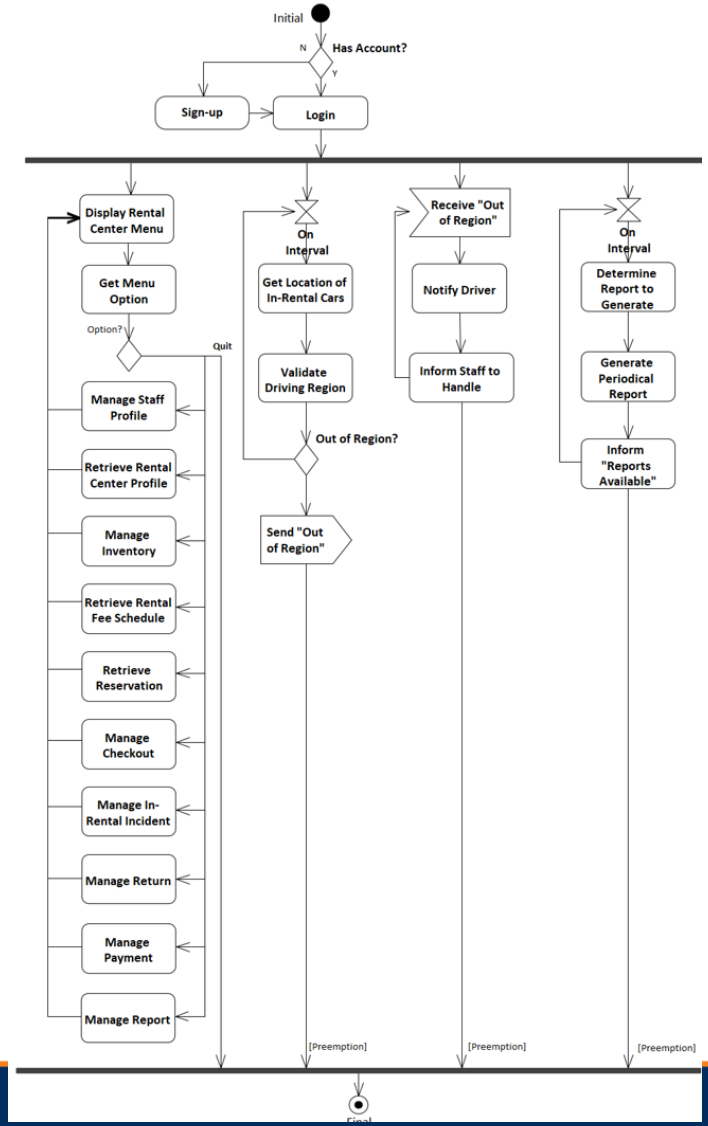
```
public class CarItem {
    private String licensePlate;
    private String manufacturer;
    private String modelType;
    private String modelName;
    private int modelYear;
    ....
}
```



Implementing Behavior View Design

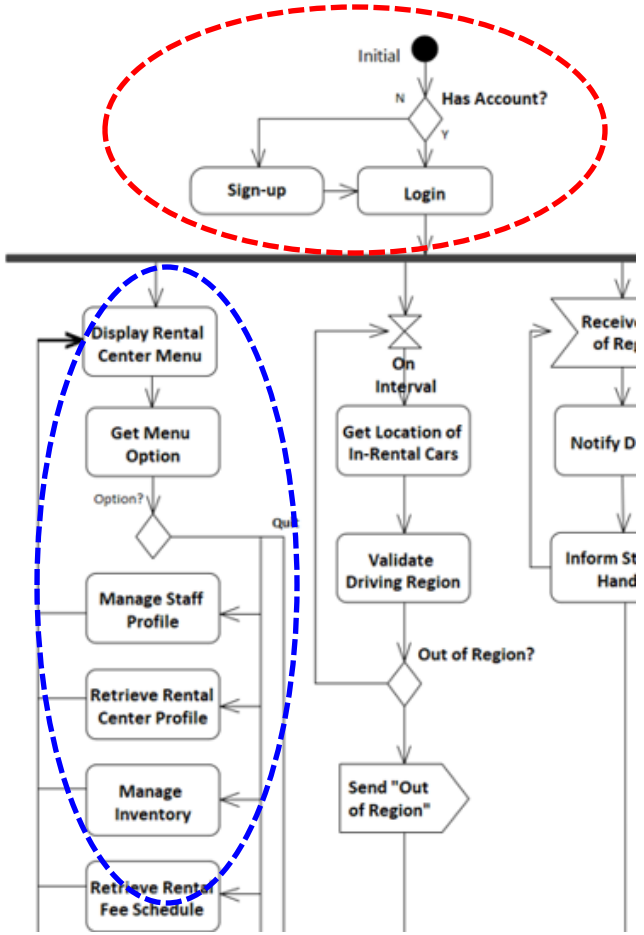
Implementing Activity Diagram

- Activity Diagram for Car Rental Management System



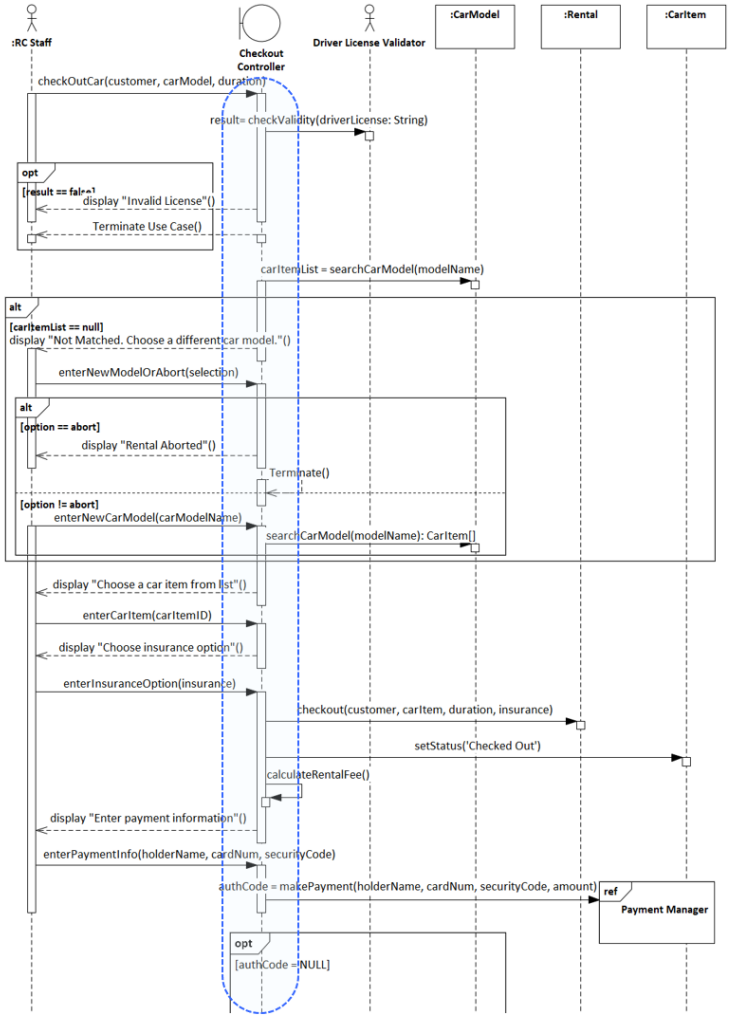
Implementing Activity Diagram

```
public class RentalSystem {
public static void main(String[] args) {
boolean loggedIn = false;
while (!loggedIn) {
    if (userHasAccount()) {
        loggedIn = login();
    } else {
        signUp();
    loggedIn = login();
}
boolean quit = false;
while (!quit) {
    displayRentalCenterMenu();
    int option = getMenuOption();
    switch (option) {
    case 1: manageStaffProfile();
        break;
    case 2:
retrieveRentalCenterProfile();
        break;
    case …
```



Implementing Sequence Diagram

- Sequence Diagram for checkout_Car() method.
- The control flow of the diagram defines the skeleton of the method in a program.



Implementing Sequence Diagram

```
public void checkOutCar(Customer customer, String carModel, int duration) {
        boolean result = checkoutController.checkValidity(customer.getDriverLicense());
        if (!result) {
            System.out.println("Invalid License");
            // Terminate use case, possibly with an exception or by returning early
            return;
        }

        CarItemList carItemList = checkoutController.searchCarModel(carModel);
        if (carItemList == null || carItemList.isEmpty()) {
            System.out.println("Not Matched. Choose a different car model.");
            String selection = customer.enterNewModelOrAbort();
            if ("abort".equals(selection)) {
                System.out.println("Rental Aborted");
                // Terminate use case
            } else {
                // The customer enters a new car model
                carItemList = checkoutController.searchCarModel(selection);
                // Continue the rental process with the new carItemList
            }
        }
        …
}
}
```

Concluding Remarks

- Design
  - Design becomes the concrete basis for implementation.
- Implementation
  - Implementation is to add detailed code-level decisions on top of the design.
- Language
  - OOAD can best be implemented in an object-oriented programming language.
- The Message
  - Design determines the quality of implementation