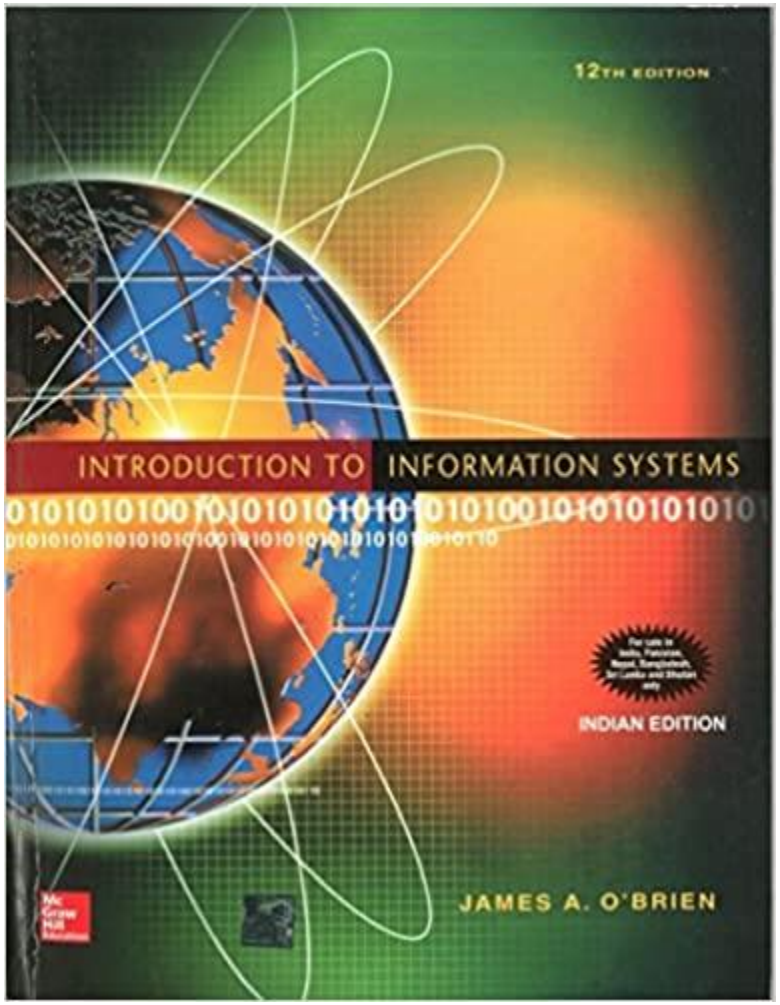
CHAPTER 1

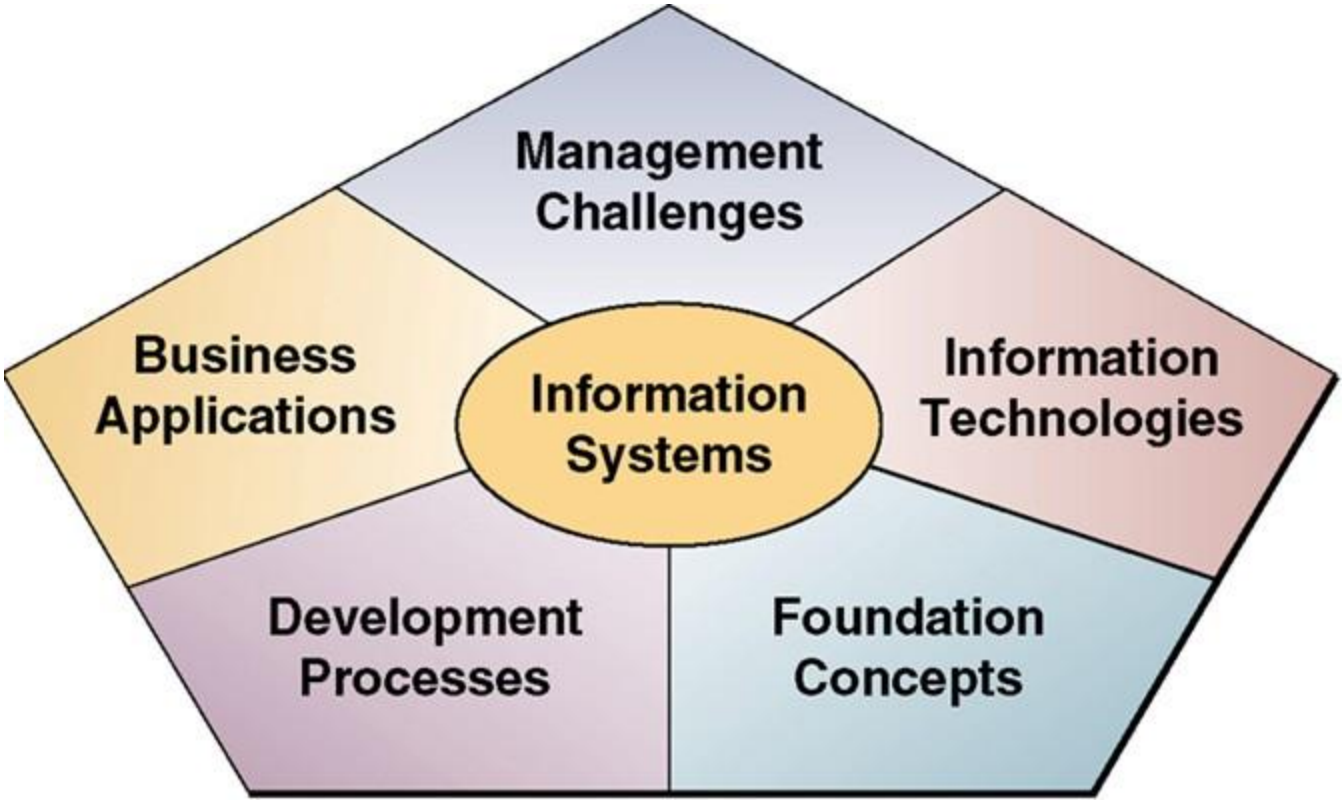
INTRODUCTION TO INFORMATION SYSTEM

Reference

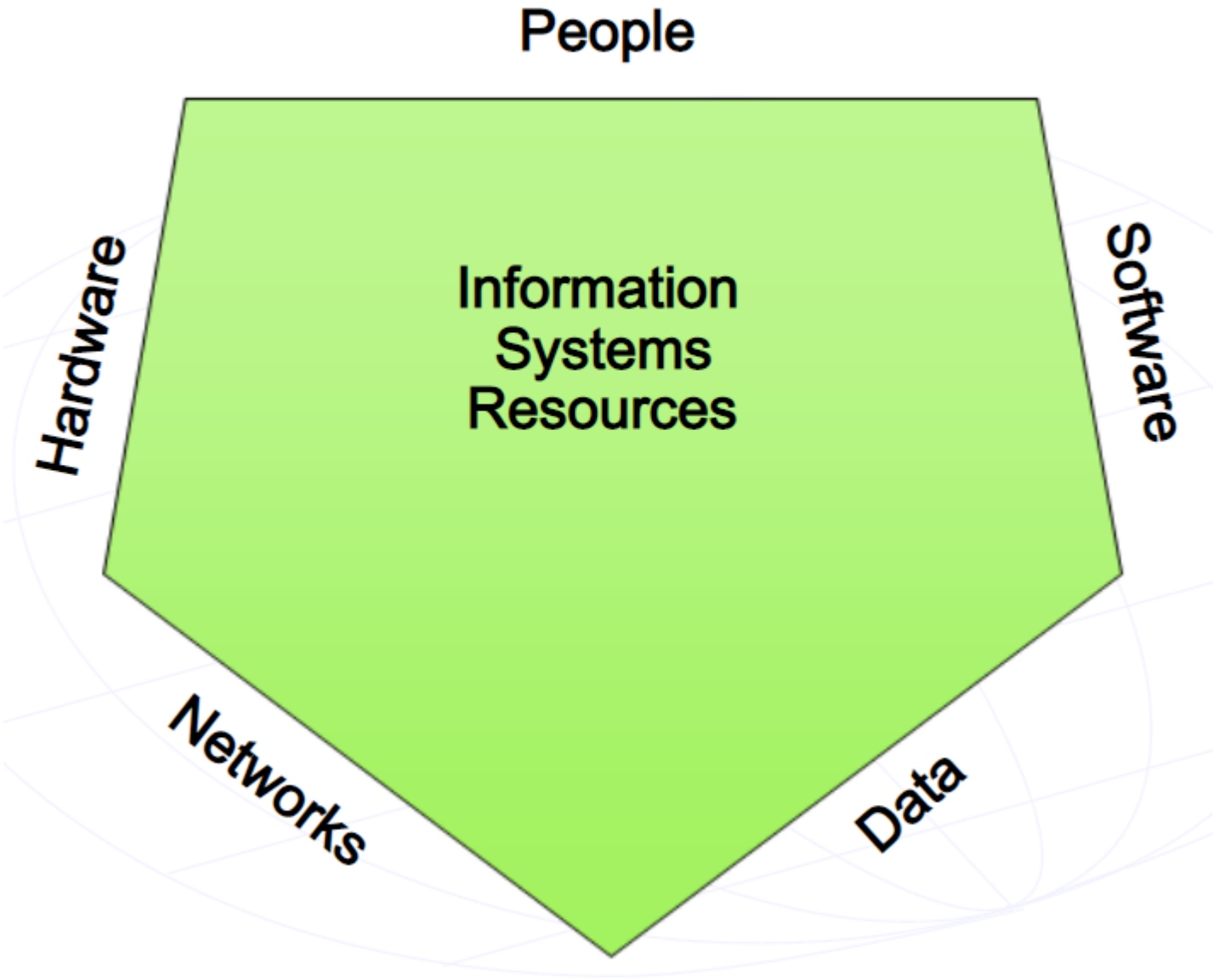
- This chapter refers from the book: Introduction to information systems, 12th edition. James A. O'Brien. McGrow Hill, 2005.



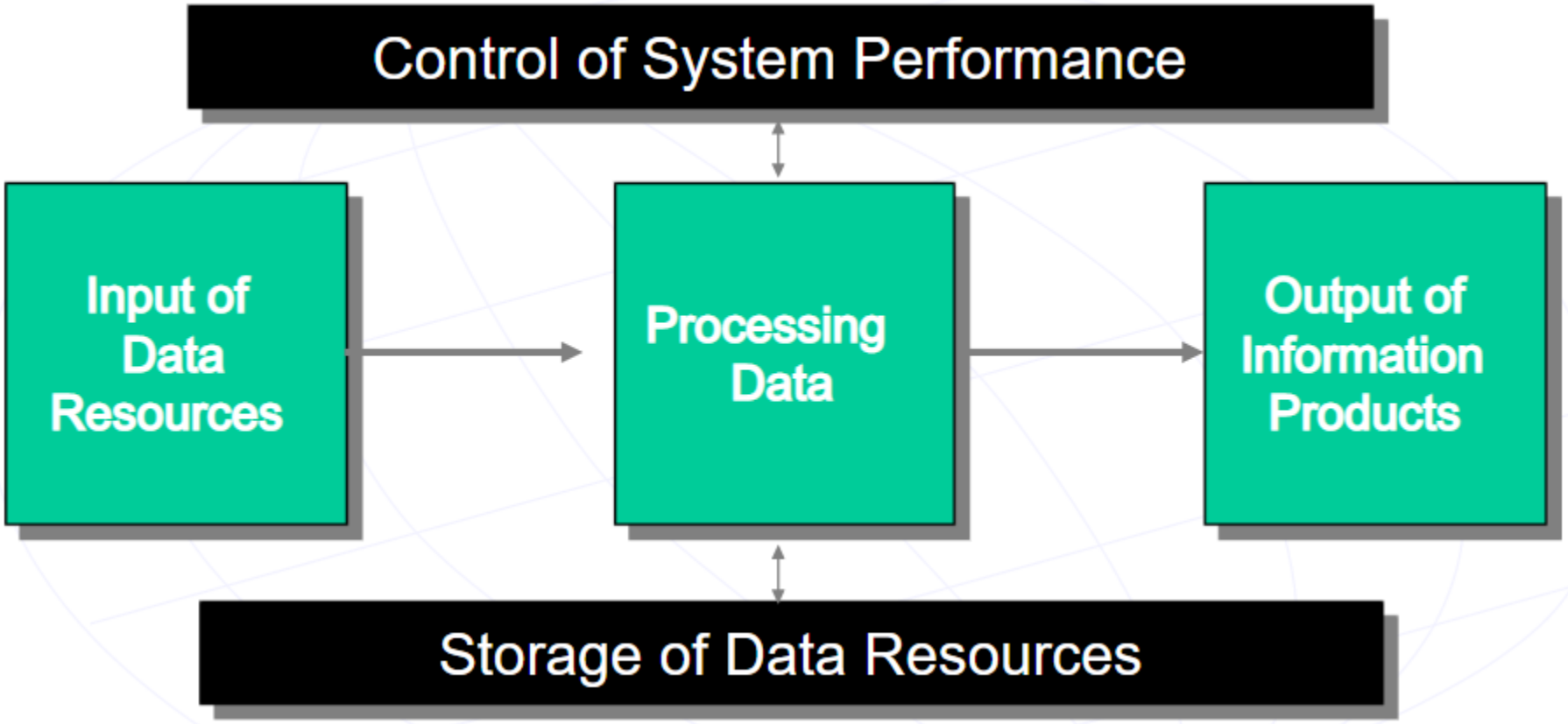
IS framework



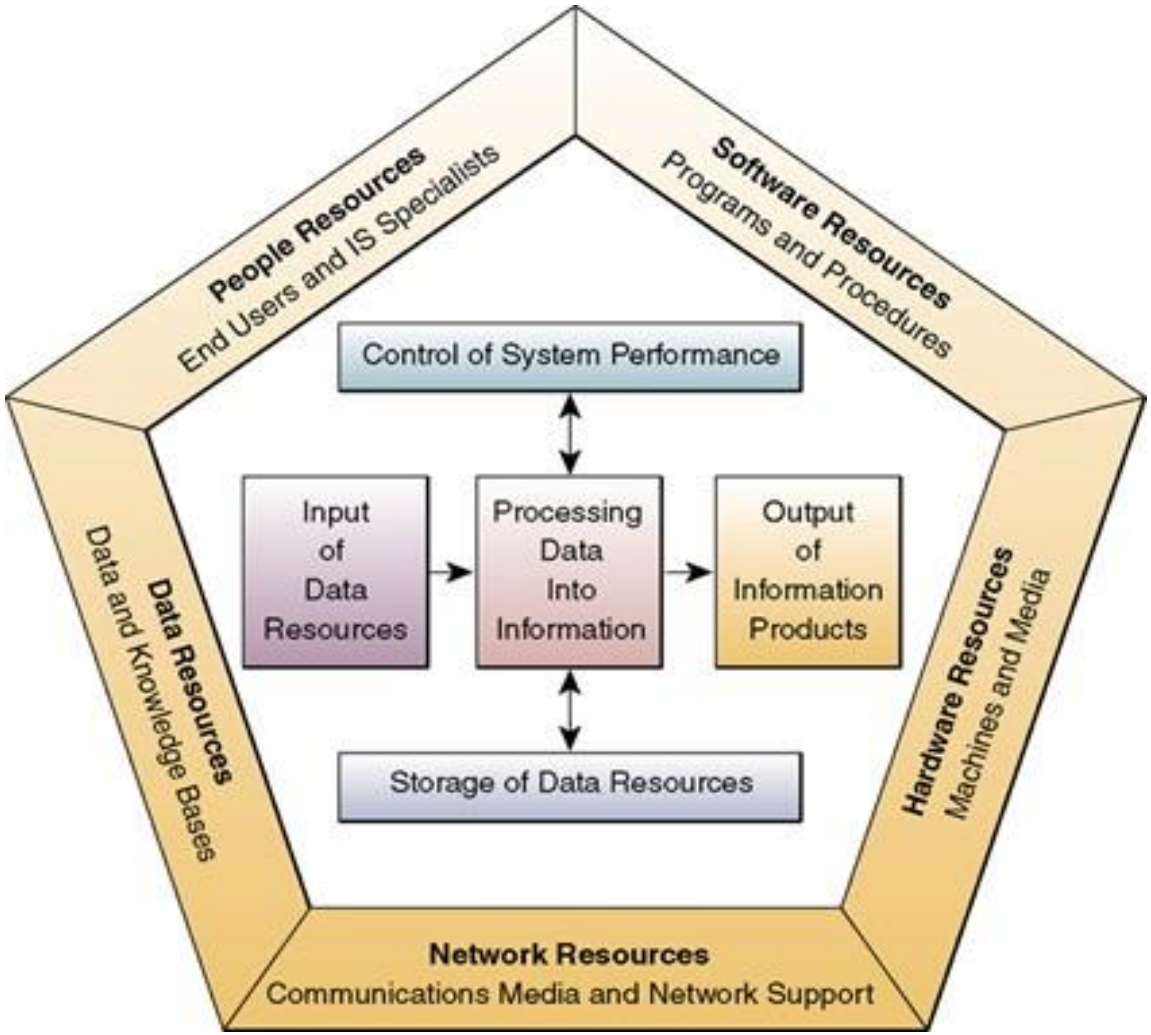
Components of an IS



An IS

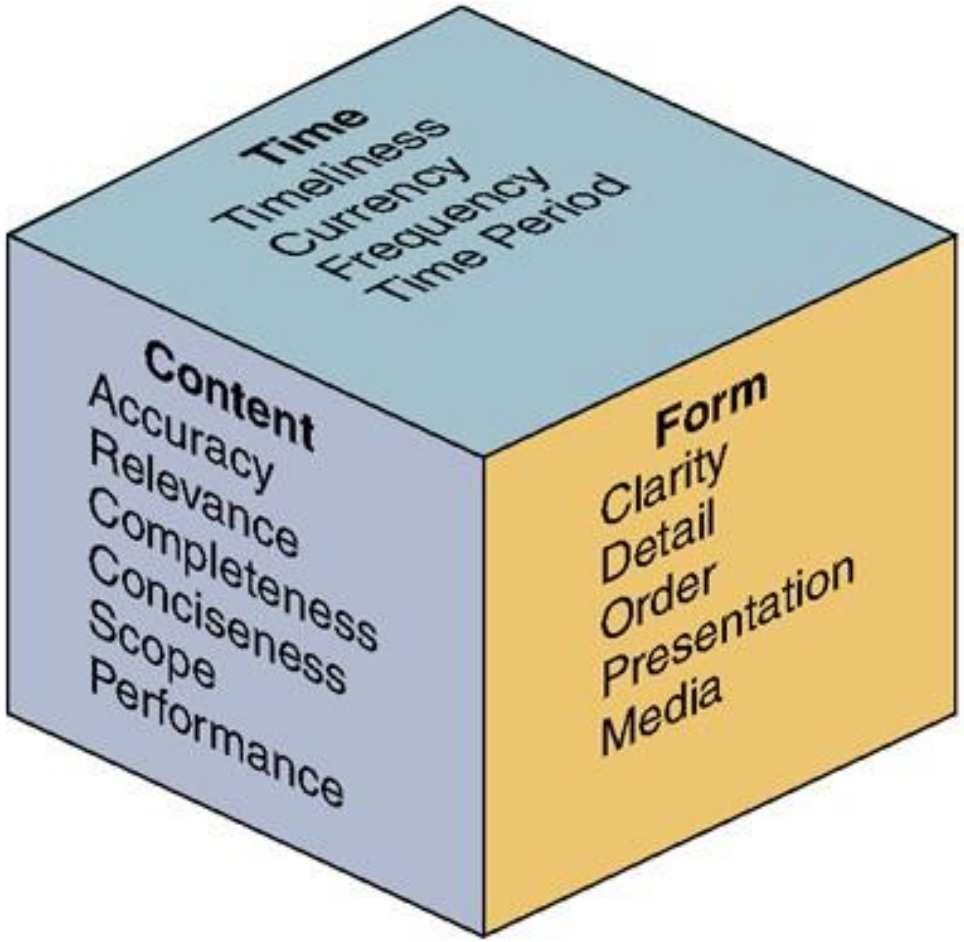

Control of System Performance
Input of Data Resources
Processing Data
Output of Information Products
Storage of Data Resources

Components of an IS

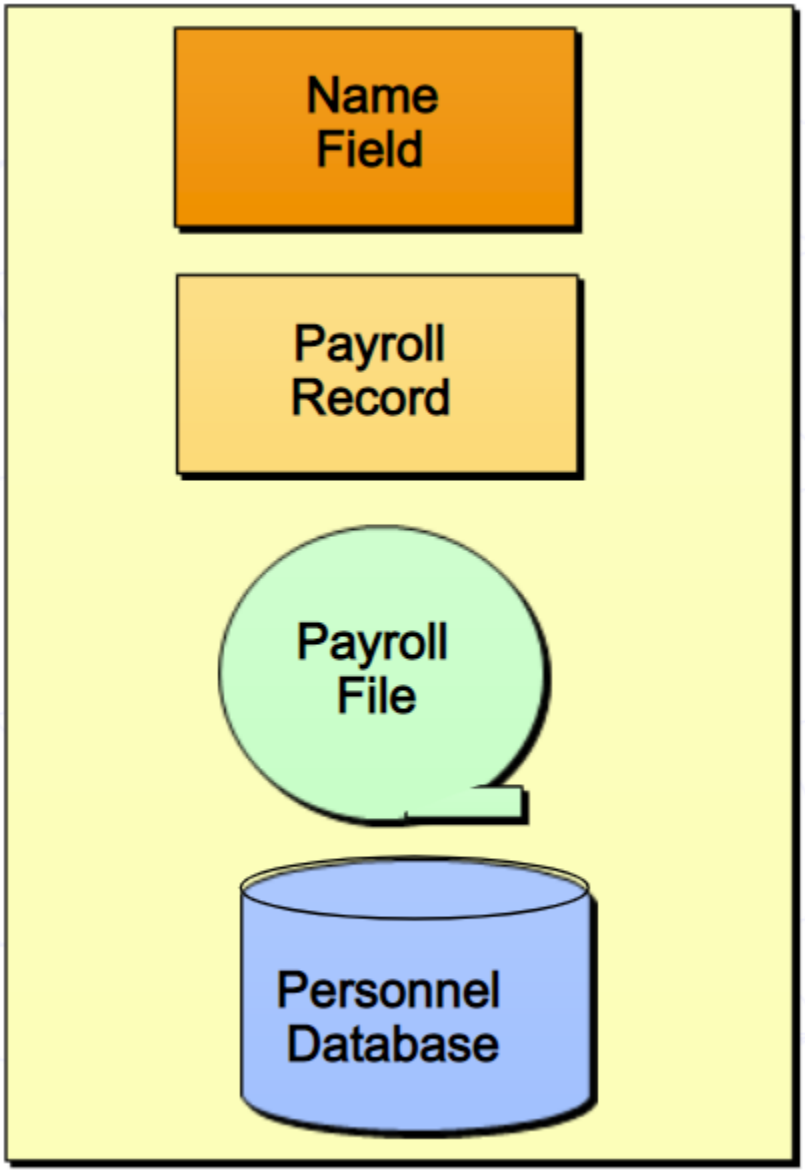


Data vs. Information

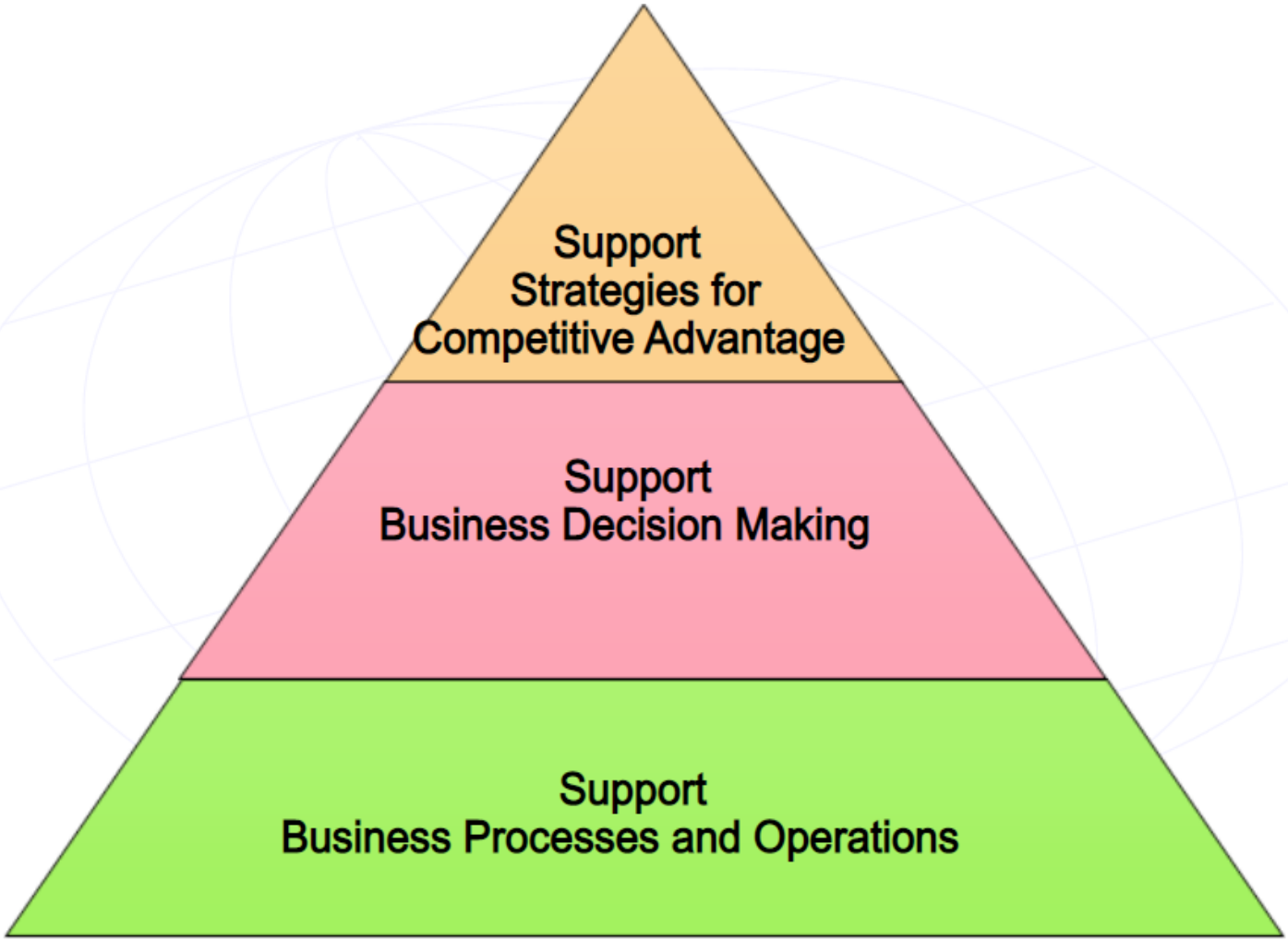
Information quality



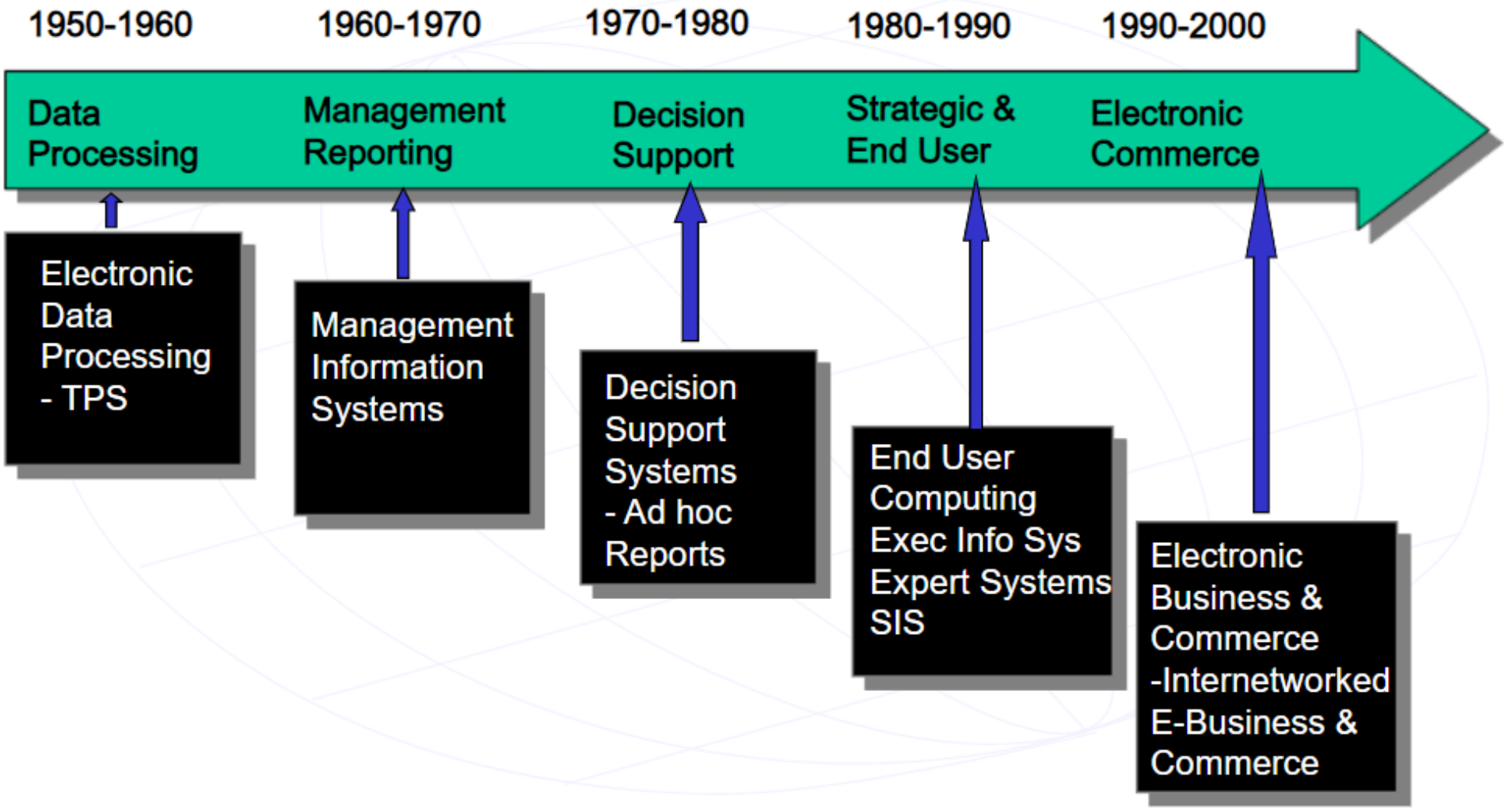
Logical data elements



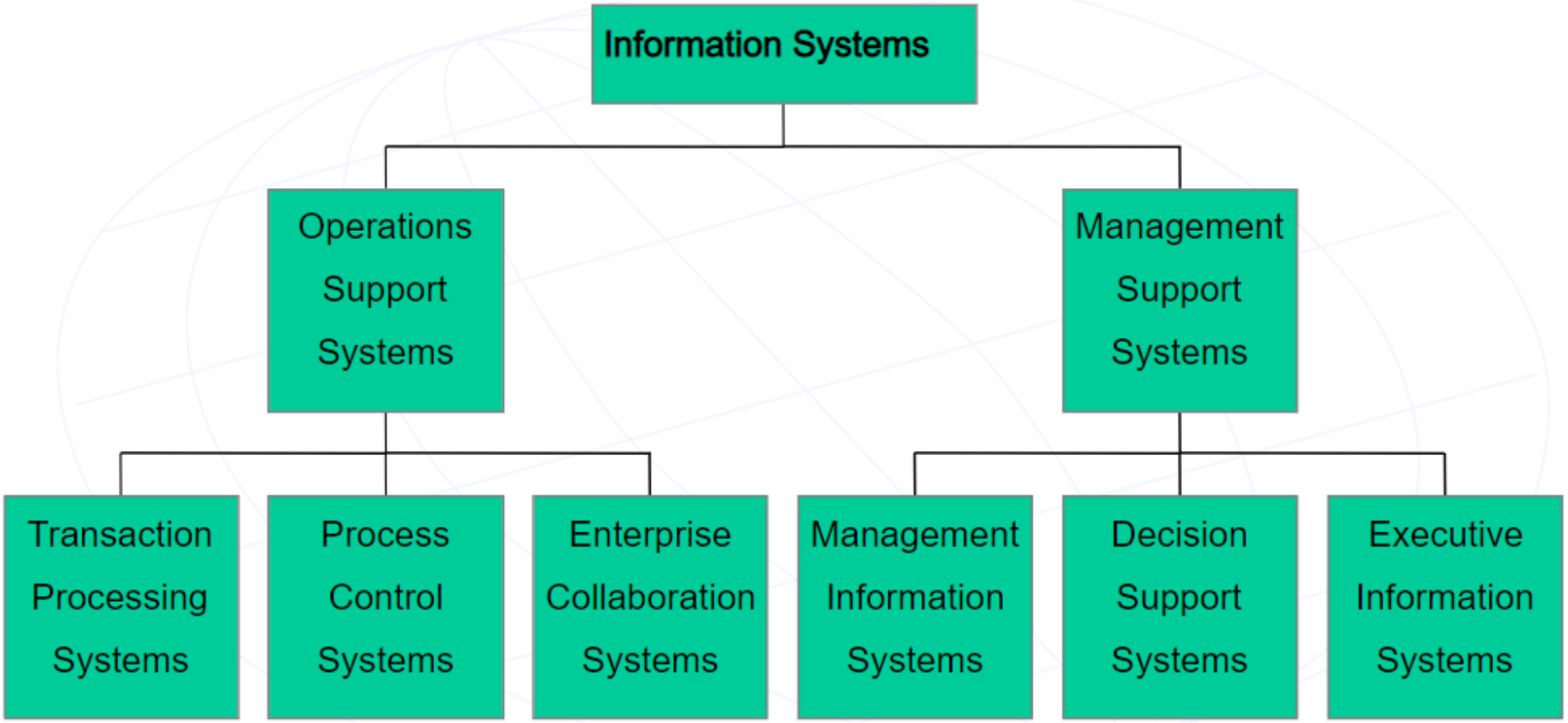
Roles of IS


Support
Strategies for
Competitive Advantage
Support
Business Decision Making
Support
Business Processes and Operations

Roles of IS: history



Types of IS


Information Systems
Operations Support Systems
Management Support Systems
Transaction Processing Systems
Process Control Systems
Enterprise Collaboration Systems
Management Information Systems
Decision Support Systems
Executive Information Systems

IS development process

Introduction to UML

IS development process

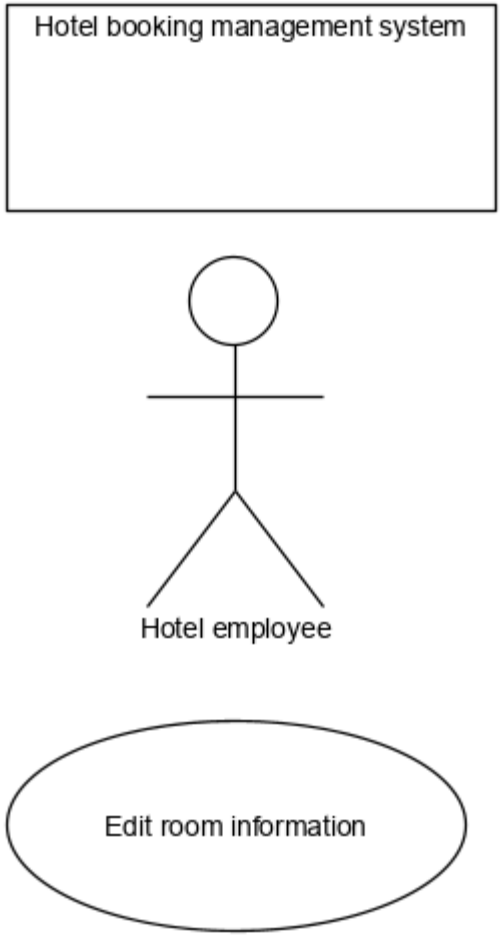
- Requirement
- Analysis
- Design
- Implementation
- Testing

Requirement

- Use case diagram
  - Elements
  - Relationships among elements

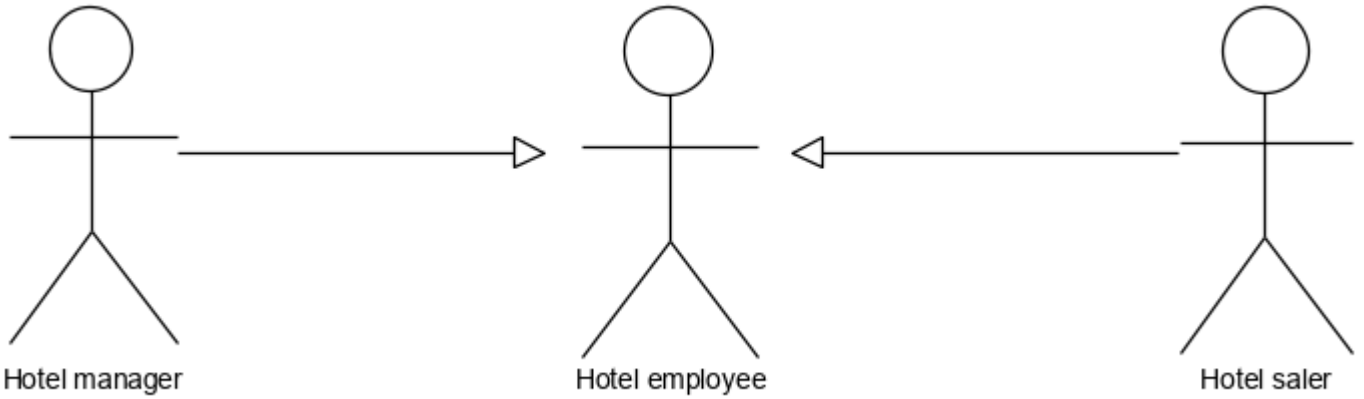
Use case diagram: elements

- System
- Actor
- Use case



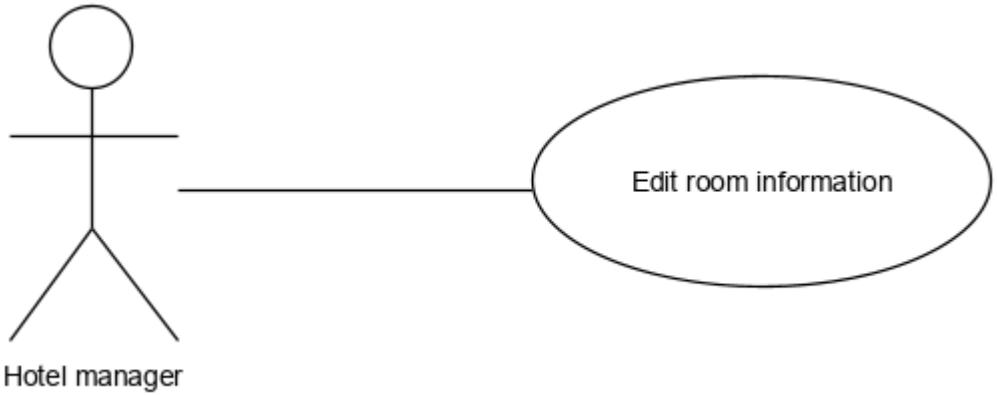
Use case diagram: Actor (1)

- Generalization relationship



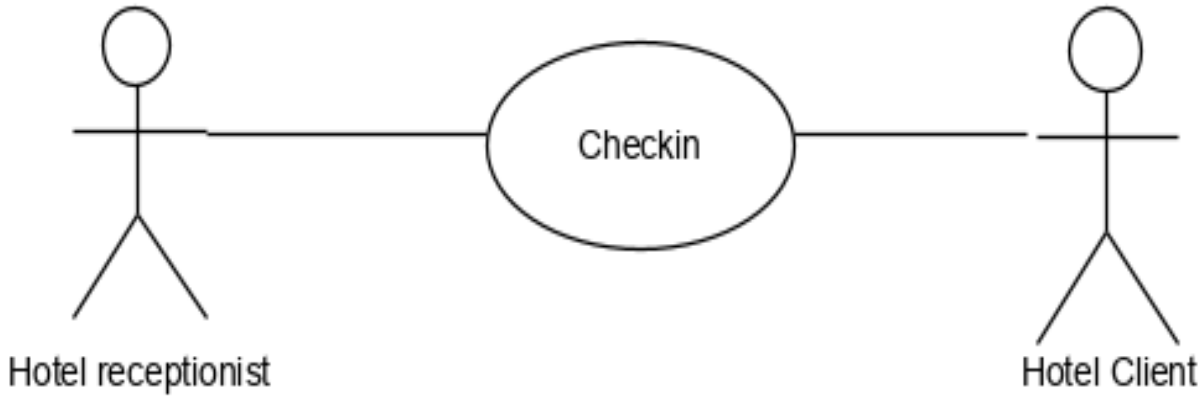
Use case diagram: Actor (2)

- Use case has one actor



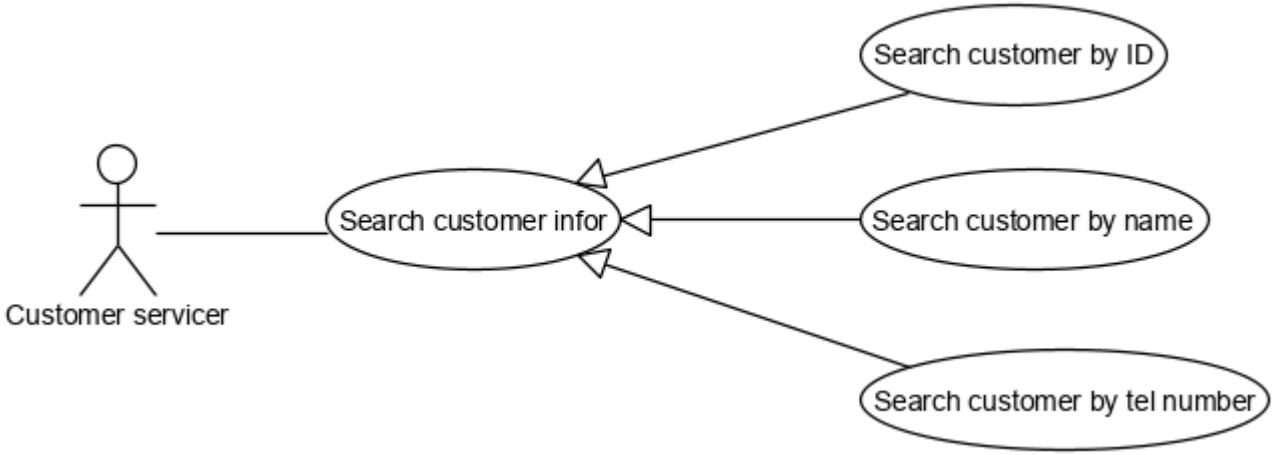
Use case diagram: Actor (3)

- Use case has two actors



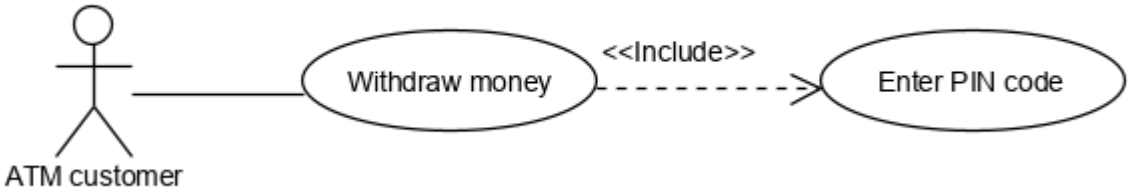
Use case diagram: use case (1)

- Generalization relationship



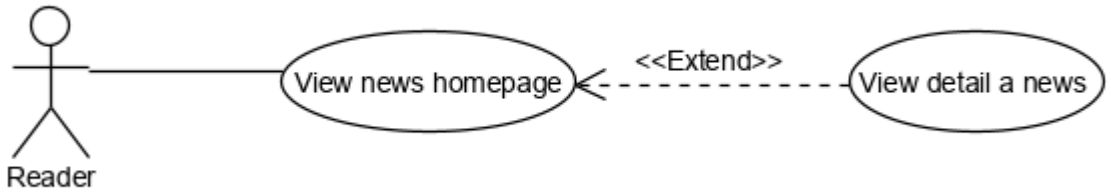
Use case diagram: use case (2)

- Include relationship



Use case diagram: use case (3)

- Extend relationship

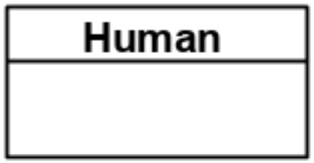


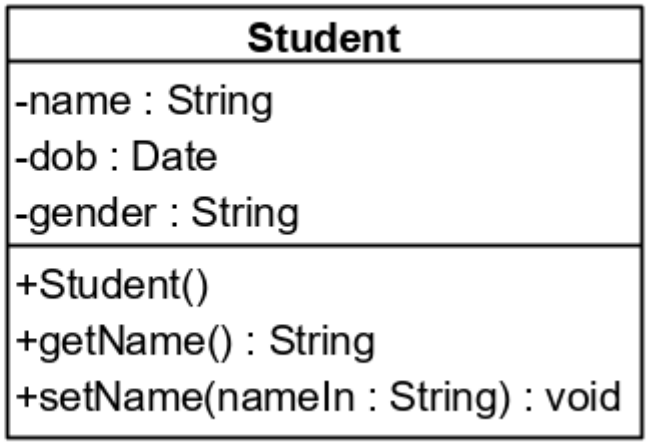
Analysis

- Class diagram
- State diagram
- Sequence diagram
- Collaboration/communication diagram

Class diagram: elements

- Class



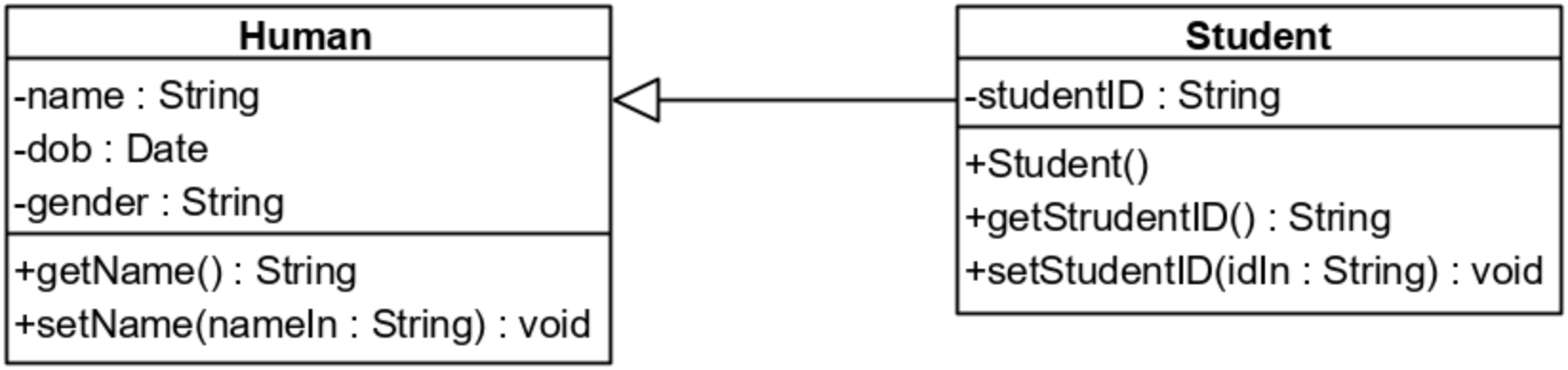


Class diagram: relationship (1)

- Interaction

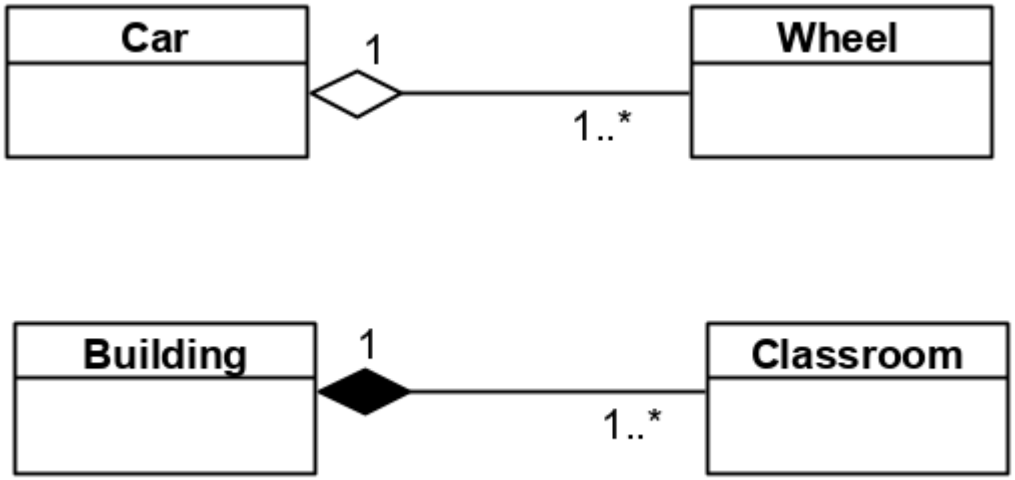
Class diagram: relationship (2)

- Generalization



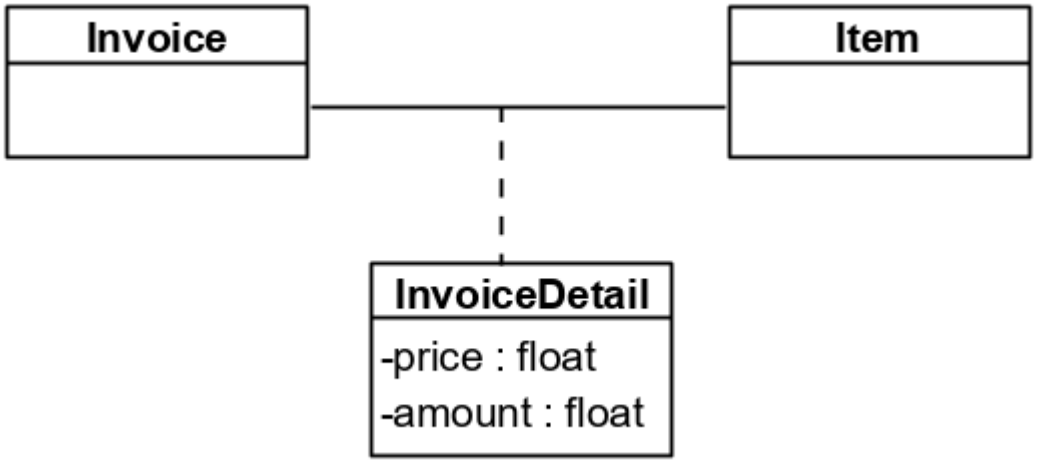
Class diagram: relationship (3)

- Aggregation vs. composition



Class diagram: relationship (4)

- Association

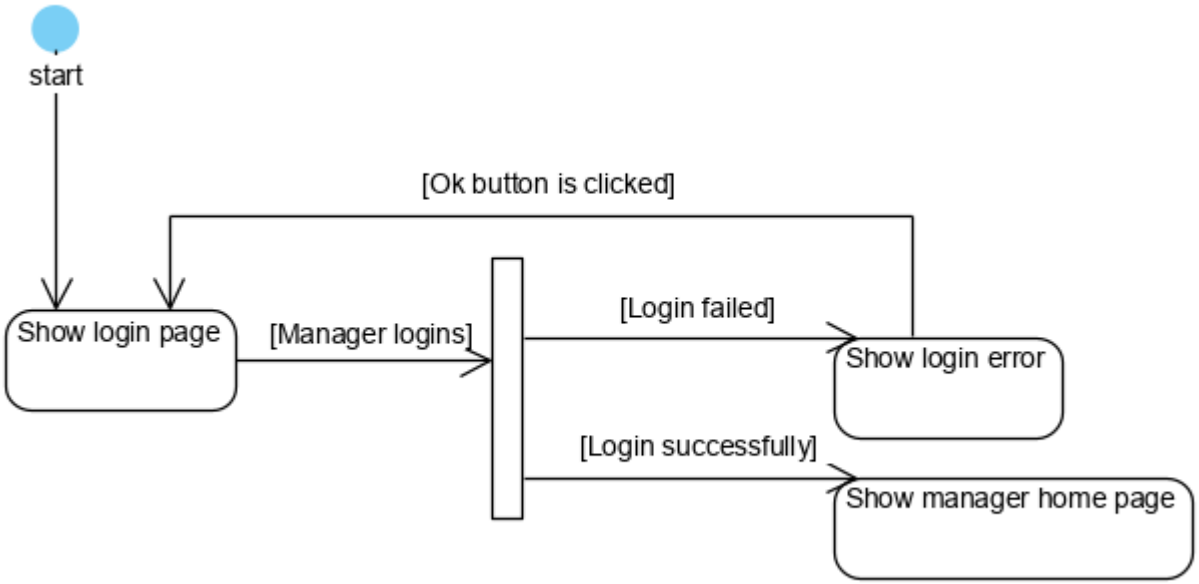


State diagram: elements

- Start
- State
- End

State diagram: relationship

- Change state

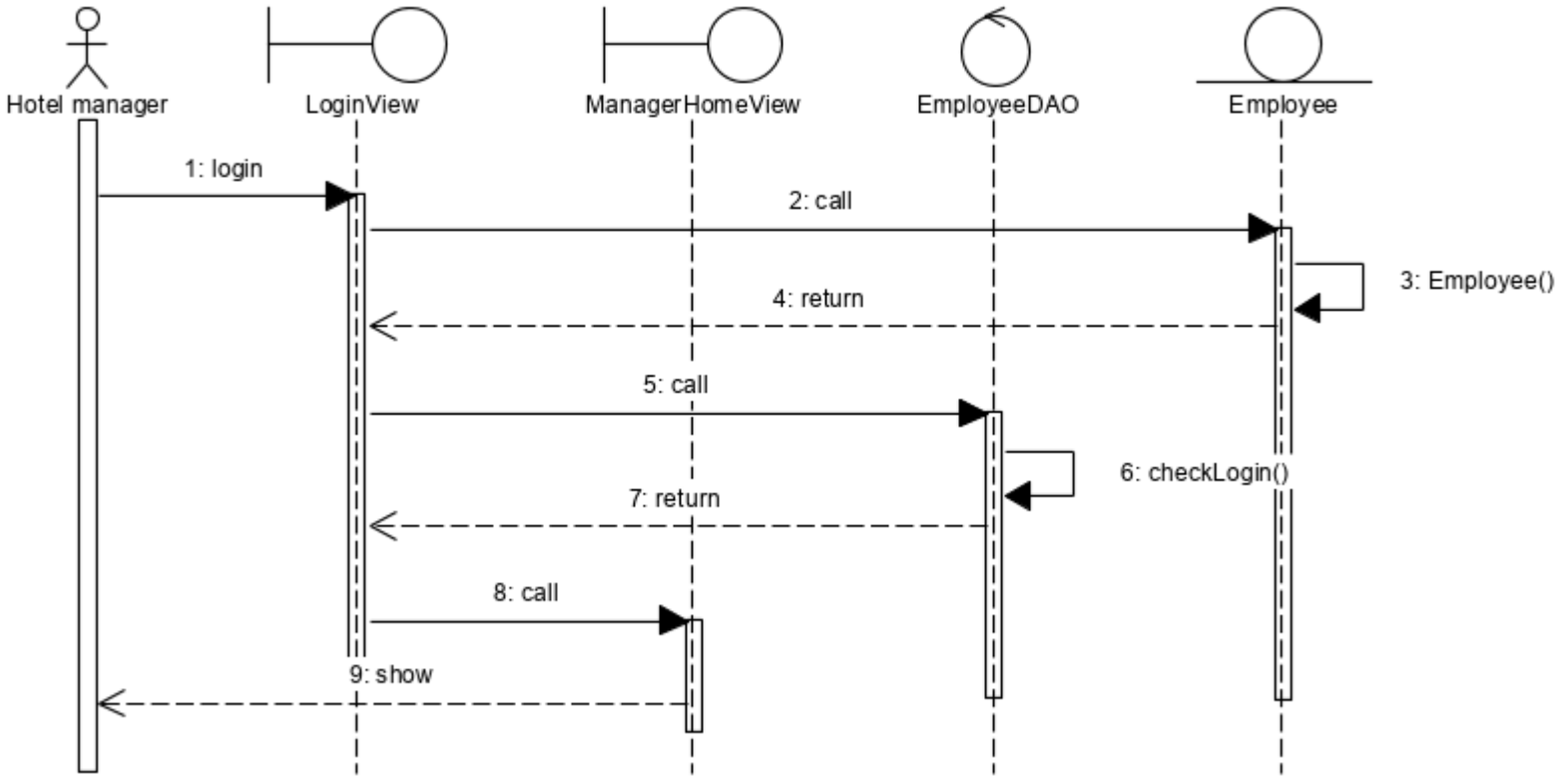


Sequence diagram: elements

- Actor
- View/interface/boundary class
- Control/business class
- Model/entity class

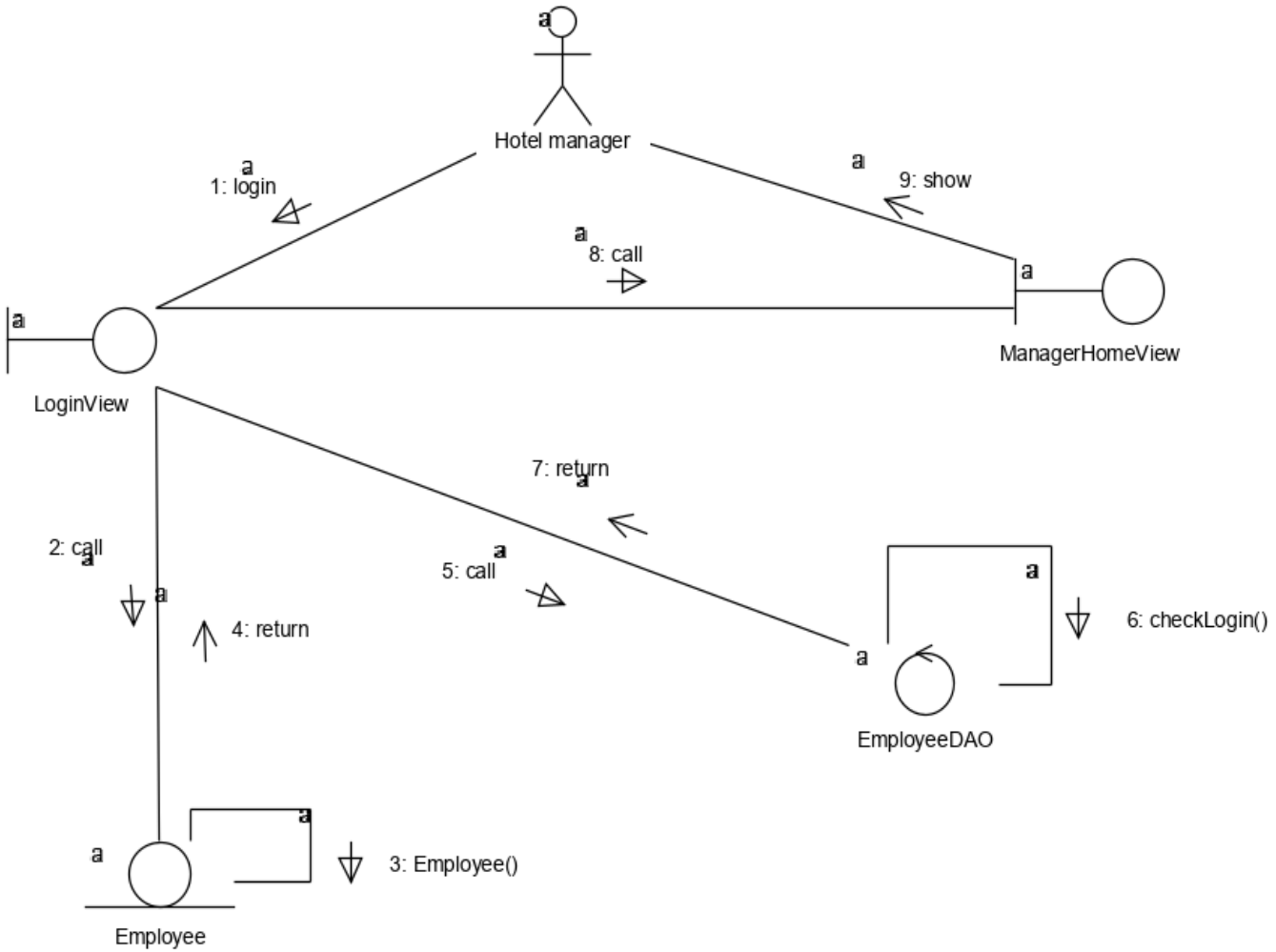
Sequence diagram: event steps

- Ex: login



Communication diagram: event steps

- Ex: login

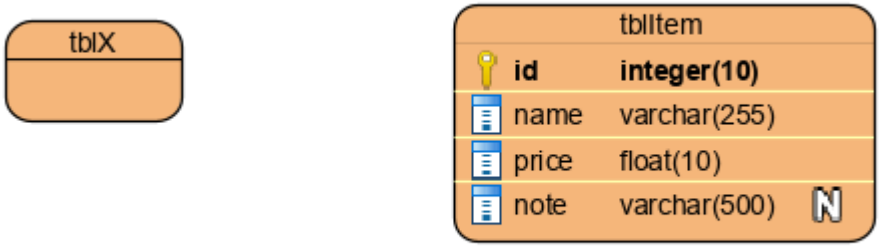


Design

- Class diagram (entity, full detail – presented)
- Database diagram
- Activity diagram
- Sequence/communication diagram (presented)
- Package diagram
- Deployment diagram

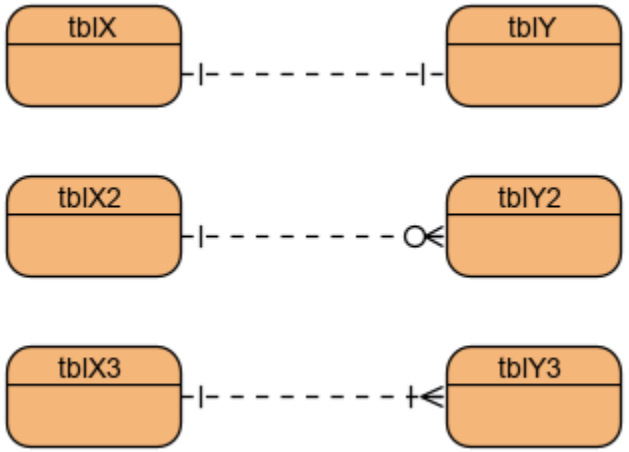
Database diagram: elements

- table



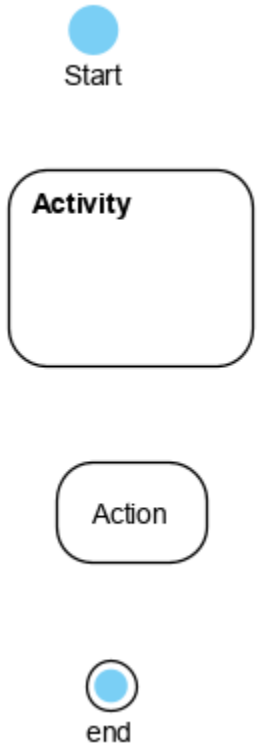
Database diagram: relationships

- 1-1
- 1-n
- n-n (convert to many 1-n relationships



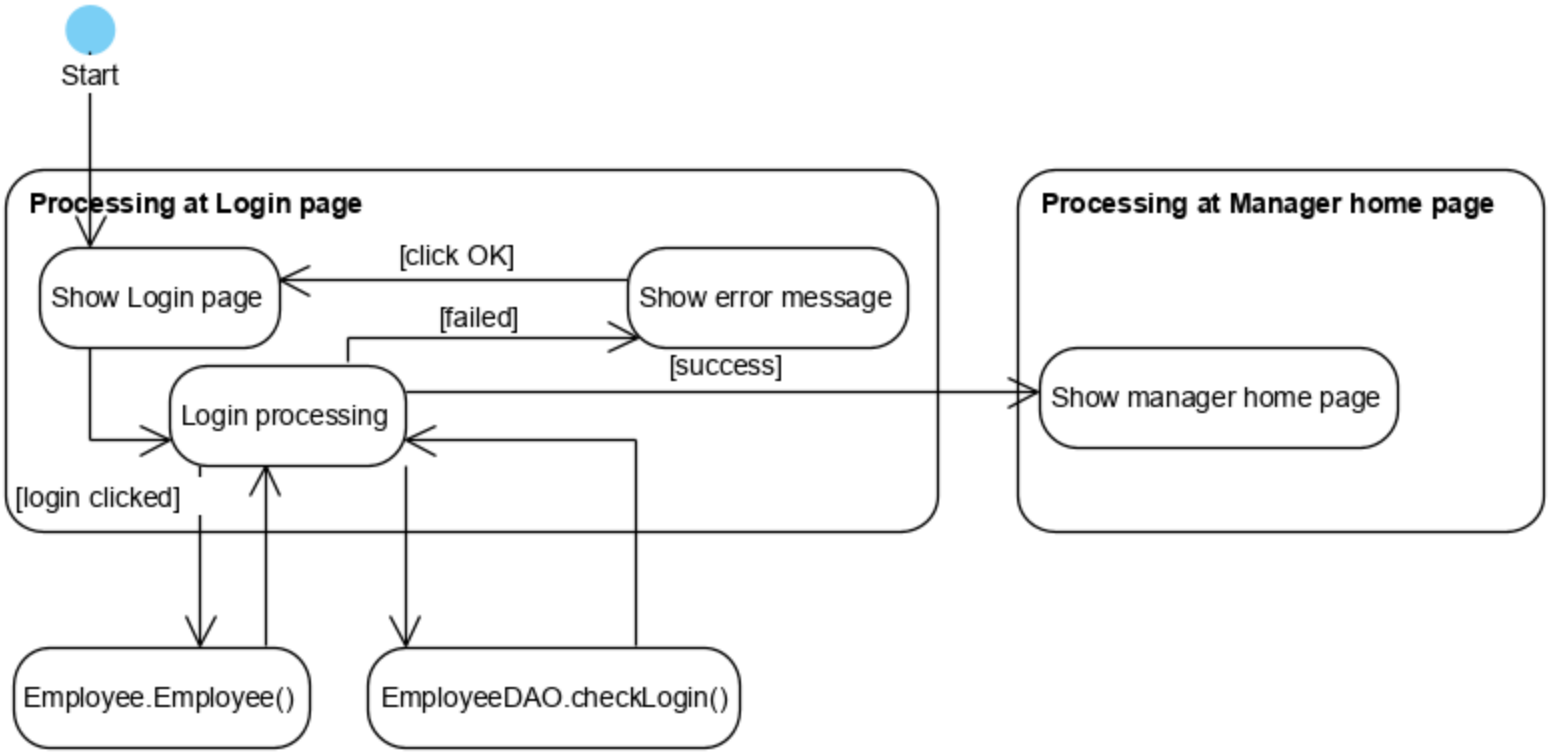
Activity diagram: elements

- Start
- Activity
- Action
- End



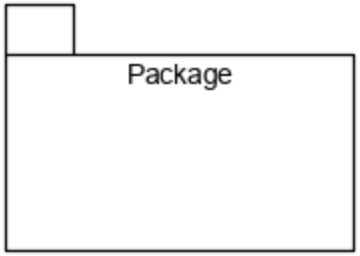
Activity diagram: relationships

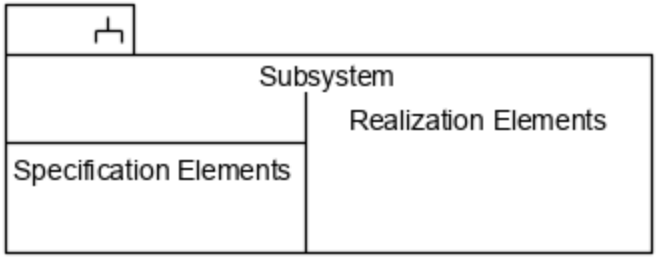
- Change action



Package diagram

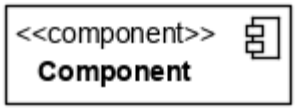
- Package
- sub-system

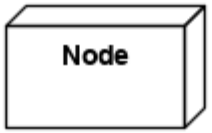




Deployment diagram

- artifact
- node
- component





Chapter 2:
Introduction of software quality assurance

Reference

- This chapter refers from the book:
  - Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers
  - Introduction to Software Testing

Software Quality

Software Quality

Software quality is:

(1) The degree to which a system, component, or process meets specified requirements.

by Philip Crosby

(2) The degree to which a system, component, or process meets customer or user needs or expectations.

by Joseph M. Juran

Now, more closely…

SQA - Expanded Definition

Software quality assurance is:

A systematic, planned set of actions necessary to provide adequate confidence that the software development process or the maintenance process of a software system product conforms to established functional technical requirements as well as with the managerial requirements of keeping the schedule and operating within the budgetary confines.

The objectives of SQA activities in Software Development

(1) Assuring an acceptable level of confidence that the software will conform to functional technical requirements.

(2) Assuring an acceptable level of confidence that the software will conform to managerial scheduling and budgetary requirements.

(3) Initiation and management of activities for the improvement and greater efficiency of software development and SQA activities.

The objectives of SQA activities in Software Maintenance (product-oriented)

(1) Assuring an acceptable level of confidence that the software maintenance activities will conform to the functional technical requirements.

(2) Assuring an acceptable level of confidence that the software maintenance activities will conform to managerial scheduling and budgetary requirements.

(3) Initiate and manage activities to improve and increase the efficiency of software maintenance and SQA activities.

Software Quality factors

The Requirements Document

- Requirement Documentation (Specification) is one of the **most important elements** for achieving software quality
- Need to explore what constitutes a **good** software requirements document.
- Some SQA Models suggest 11-15 factors categorized; some fewer; some more
- Want to become **familiar** with these quality factors, and
- Who is really interested in them.
- **The need for comprehensive software quality requirements is pervasive in numerous case studies (see a few in this chapter).**
- (Where do the quality factors go??)

Need for Comprehensive Software Quality Requirements

- Need for improving poor requirements documents is widespread
- Frequently lack **quality factors** such as: usability, reusability, maintainability, …
- Software industry groups the long list of related attributes into what we call ***quality factors. (Sometimes non-functional requirements)***
- Natural to assume an unequal emphasis on all quality factors.
- Emphasis varies from project to project
  - Scalability; maintainability; reliability; portability; etc.
- Let's look at some of the categories…

Extra Thoughts

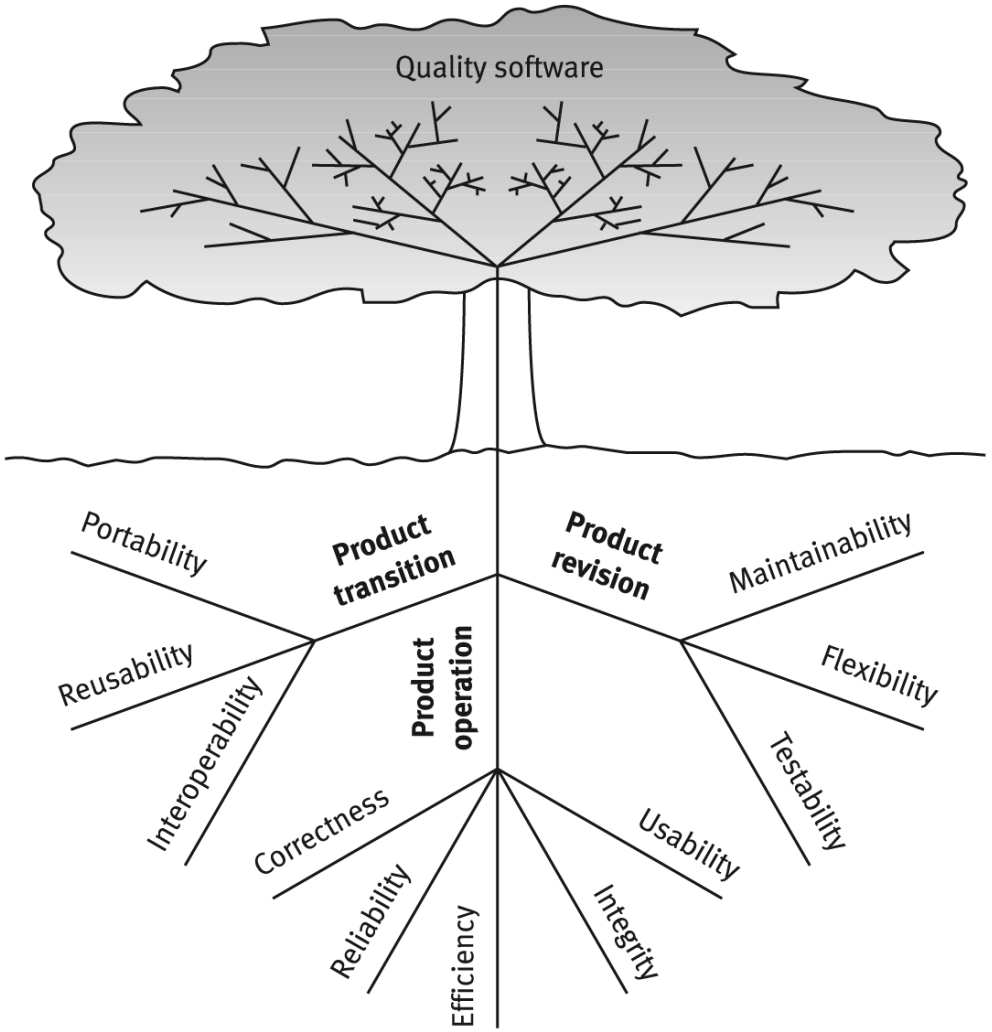
- Seems like in Software Engineering we concentrate on capturing, designing, implementing, and deploying with emphasis on **functional requirements.**
- Little (not none!) emphasis on the **non-functional requirements** (quality factors).
- More and more emphasis now placed on quality factors
- Can be a critical factor in satisfying overall requirements.

- In the RUP, non-functional requirements are captured in the Software Requirements Specification (SRS); functional requirement usually captured in Use Case stories.

McCall's Quality Factors

- McCall has 11 factors; Groups them into categories.
  - 1977; others have added, but this still prevail.
- Three categories:
  - **Product Operation Factors**
    - How well it runs….
    - Correctness, reliability, efficiency, integrity, and usability
  - **Product Revision Factors**
    - How well it can be changed, tested, and redeployed.
    - Maintainability; flexibility; testability
  - **Product Transition Factors**
    - How well it can be moved to different platforms and interface with other systems
    - Portability; Reusability; Interoperability
- Since these underpin the notion of quality factors and others who have added, reword or add one or two, we will spend time on these factors.

McCall's Quality Factors



Product operation factors

- Correctness
- Reliability
- Efficiency
- Integrity
- Usability

How well does it run and ease of use.

McCall's Quality Factors

Category: Product **Operation** Factors

1. Correctness.

- **Please note that we are asserting that 'correctness' issues are arising from the requirements documentation and the specification of the outputs…**
- **Examples include:**
  - **Specifying accuracies for correct outputs** at, say, NLT <1% errors, that could be affected by inaccurate data or faulty calculations;
  - **Specifying** the **completeness of the outputs** provided, which can be impacted by incomplete data (often done)
  - **Specifying the timeliness of the output** (time between event and its consideration by the software system)
  - **Specifying the standards** for coding and documenting the software system
  - we have talked about this: standards and integration; Essential!!

McCall's Quality Factors
Category: Product Operation Factors

2. Reliability Requirements. (remember, this quality factor is specified in the **specs**!)

- Reliability requirements deal with the failure to provide service.
  - –Address failure rates either overall or to required functions.
- Example specs:
  - –A heart monitoring system must have a failure rate of less than one per million cases.
  - –**Downtime** for a system will not be more than ten minutes per month (me)
  - –MTBF and MTTR - old and engineering, but still applicable.

3. Efficiency Requirements. Deals with the **hardware** resources needed to perform the functions of the software.

- –Here we consider MIPS, MHz (cycles per second); data **storage** capabilities measured in MB or TB; communication lines (usually measured in KBPS, MBPS, or GBPS).
- –Example spec: simply very slow communications…

McCall's Quality Factors
Category: Product Operation Factors

4. Integrity – deal with system security that prevent unauthorized persons access.

- Huge nowadays; Cyber Security; Internet security; network security, and more. These are certainly not the same!

5. Usability Requirements – deals with the scope of staff resources needed to train new employees and to operate the software system.

– Deals with learnability, utility, usability, and more. (me)

– Example spec: A staff member should be able to process n transactions / unit time. (me)

Product revision factors

- **Maintainability**
- **Flexibility**
- **Testability**

Can I fix it easily, retest, version it, and deploy it easily?

McCall's Quality Factors

Category: Product **Revision** Software Factors

These deal with requirements that affect the complete range of software maintenance activities:

- corrective maintenance,
- adaptive maintenance, and
- perfective maintenance
- KNOW THE DIFFERENCES!

- **1. Maintainability Requirements**
  - The degree of effort needed to identify reasons (find the problem) for software failure and to correct failures and to verify the success of the corrections.
  - Deals with the modular structure of the software, internal program documentation, programmer manual, architectural and detail design and corresponding documentation
  - Example specs: size of module <= 30 statements.
  - Refactoring...

McCall's Quality Factors
Category: Product Revision Software Factors

2. Flexibility Requirements – deals with resources to change (adopt) software to different types of customers that use the app perhaps a little differently;

- May also involve a little perfective maintenance to perhaps do a little better due to the customer's perhaps slightly more robust environment.
- Different clients exercise software differently. This is big!

3. Testability Requirements –

- Are intermediate results of computations predefined to assist testing?
- Are log files created? Backup?
- Does the software diagnose itself prior to and perhaps during operations?

Product transition factors

- **Portability**
- **Reusability**
- **Interoperability**

Can I move the app to different hardware? Interface easily with different hardware / software systems; can I reuse major portions of the code with little modification to develop new apps?

McCall's Quality Factors
Category: Product **Transition** Software Quality Factors

1. Portability Requirements: If the software must be ported to different environments (different hardware, operating systems, …) and still maintain an existing environment, then portability is a must.

2. Reusability Requirements: Are we able to reuse **parts** of the app for new applications?

- Can save immense development costs due to errors found / tested.
- Certainly higher quality software and development more quickly results.
- Very big deal nowadays.

McCall's Quality Factors

Category: Product Transition Software Quality Factors

3. Interoperability Requirements: Does the application need to **interface** with other existing systems

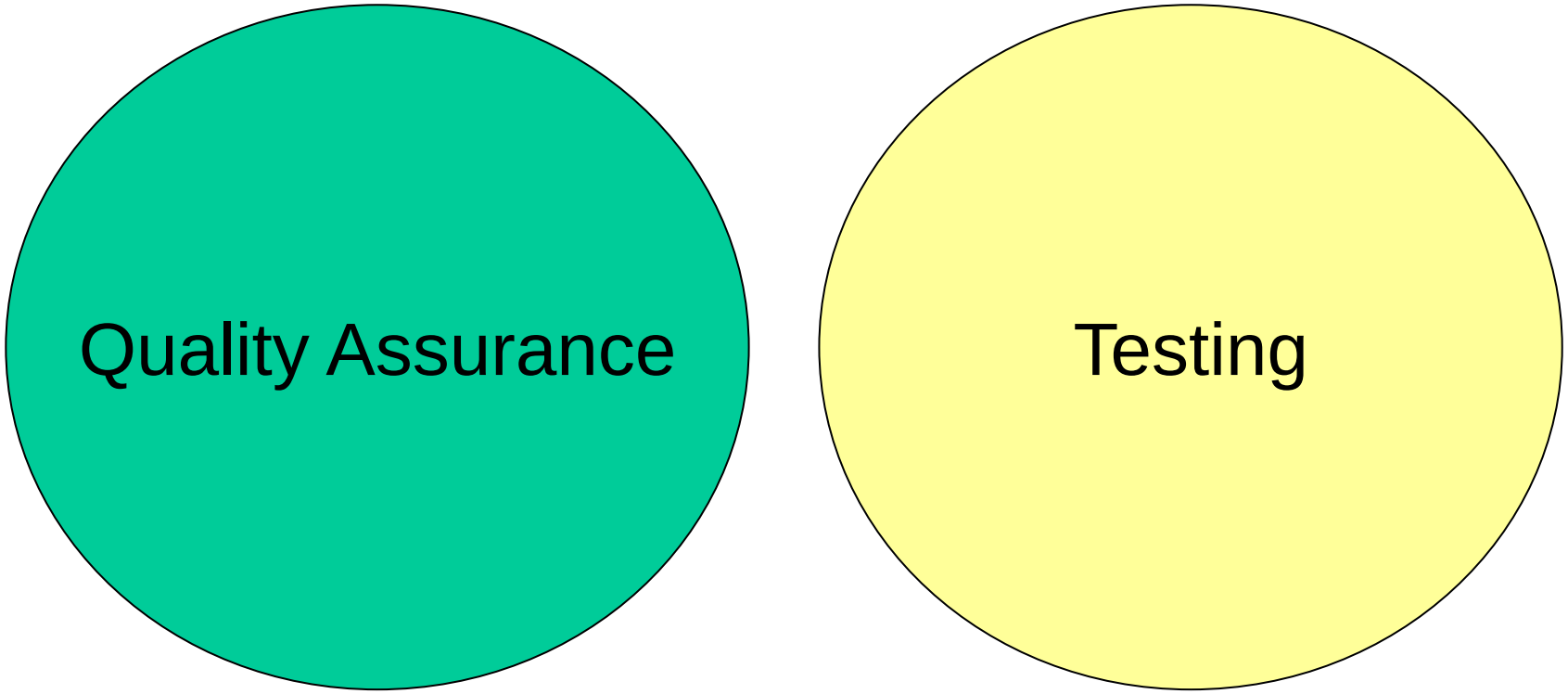
–Frequently these will be known ahead of time and plans can be made to provide for this requirement during design time.

Sometimes these systems can be quite different; different platforms, different databases, and more

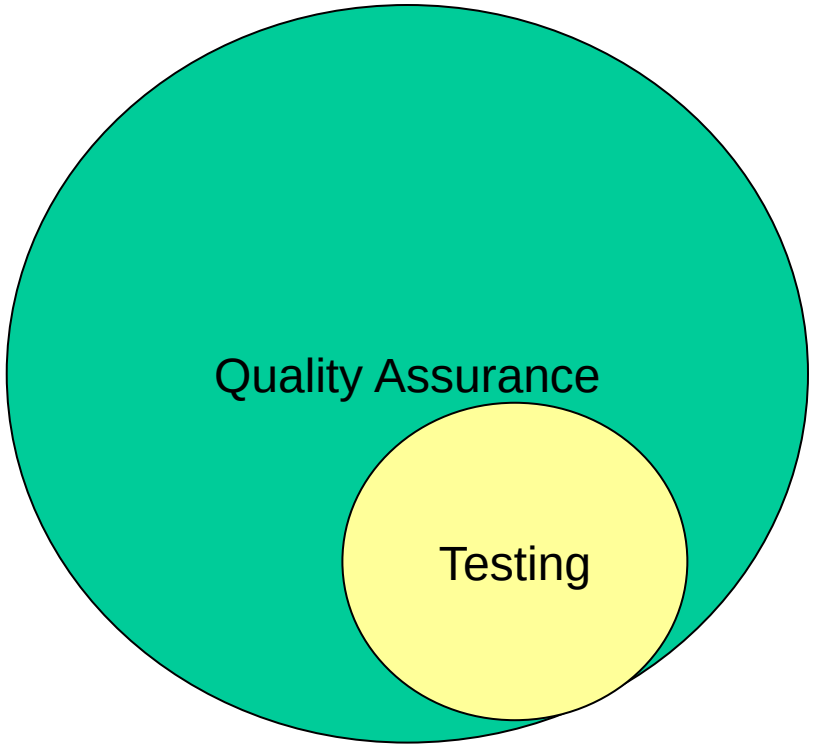
–Also, industry or standard application structures in areas can be specified as requirements.

SQA vs Testing

Quality Assurance vs Testing



Quality Assurance vs Testing



Quality Assurance

- Multiple activities throughout the dev process
  - Development standards
  - Version control
  - Change/Configuration management
  - Release management
  - Testing
  - Quality measurement
  - Defect analysis
  - Training

Testing

Also consists of multiple activities

- Unit testing
- Whitebox Testing
- Blackbox Testing
- Data boundary testing
- Code coverage analysis
- Exploratory testing
- Ad-hoc testing
- …

Testing Axioms

Testing cannot show that bugs do not exist

Exhaustive testing is impossible for non-trivial applications

Software Testing is a Risk-Based Exercise. Testing is done differently in different contexts, i.e. safety-critical software is tested differently from an e-commerce site.

Testing should start as early as possible in the software development life cycle

The More Bugs you find, the More bugs there are.

Common Error Categories

Boundary-Related

Calculation/Algorithmic

Control flow

Errors in handling/interpretting data

User Interface

Exception handling errors

Version control errors

Testing Principles

All tests should be traceable to customer requirements

- The objective of software testing is to uncover errors.
- The most severe defects are those that cause the program to fail to meet its requirements.

Tests should be planned long before testing begins

- Detailed tests can be defined as soon as the system design is complete

Tests should be prioritised by risk since it is impossible to exhaustively test a system.

- Pareto principle holds true in testing as well.

What do we test? When do we test it?

All artefacts, throughout the development life cycle.

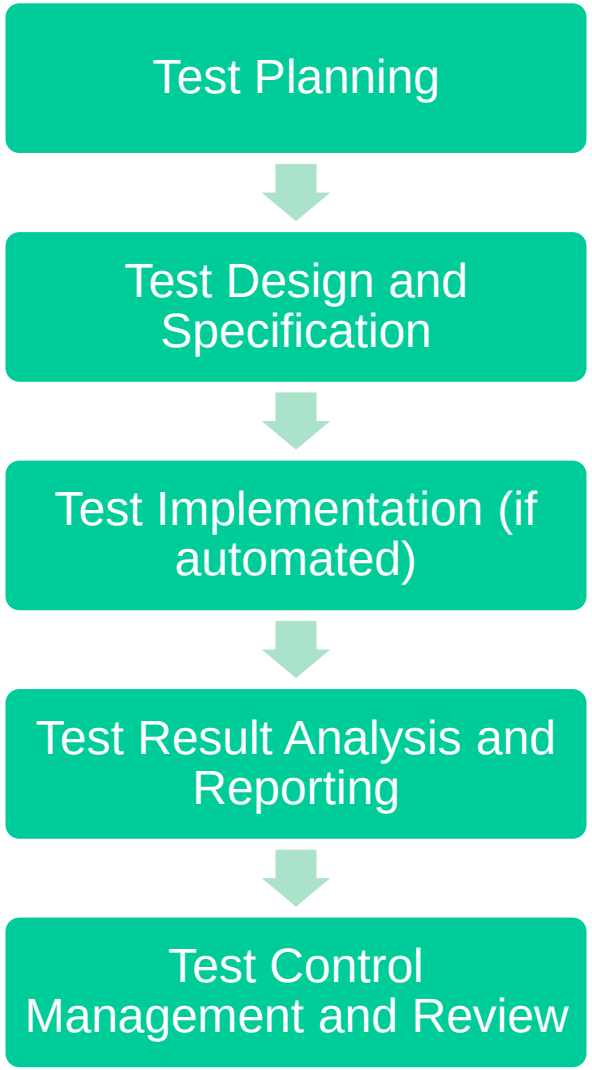
Requirements

- Are the complete?
- Do they conflict?
- Are they reasonable?
- Are they testable?

What do we test? When do we test it?

- Design
  - Does this satisfy the specification?
  - Does it conform to the required criteria?
  - Will this facilitate integration with existing systems?
- Implemented Systems
  - Does the system do what is it supposed to do?
- Documentation
  - Is this documentation accurate?
  - Is it up to date?
  - Does it convey the information that it is meant to convey?

The Testing Process



Test Planning

Test planning involves the establishment of a test plan

Common test plan elements:

- Entry criteria
- Testing activities and schedule
- Testing tasks assignments
- Selected test strategy and techniques
- Required tools, environment, resources
- Problem tracking and reporting
- Exit criteria

Test Design and Specification

Review the test basis (requirements, architecture, design, etc)

Evaluate the testability of the requirements of a system

Identifying test conditions and required test data

Design the test cases

- Identifier
- Short description
- Priority of the test case
- Preconditions
- Execution
- Post conditions

Design the test environment setup (Software, Hardware, Network Architecture, Database, etc)

Test Execution

Verify that the environment is properly set up

Execute test cases

Record results of tests (PASS | FAIL | NOT EXECUTED)

Repeat test activities

- Regression testing

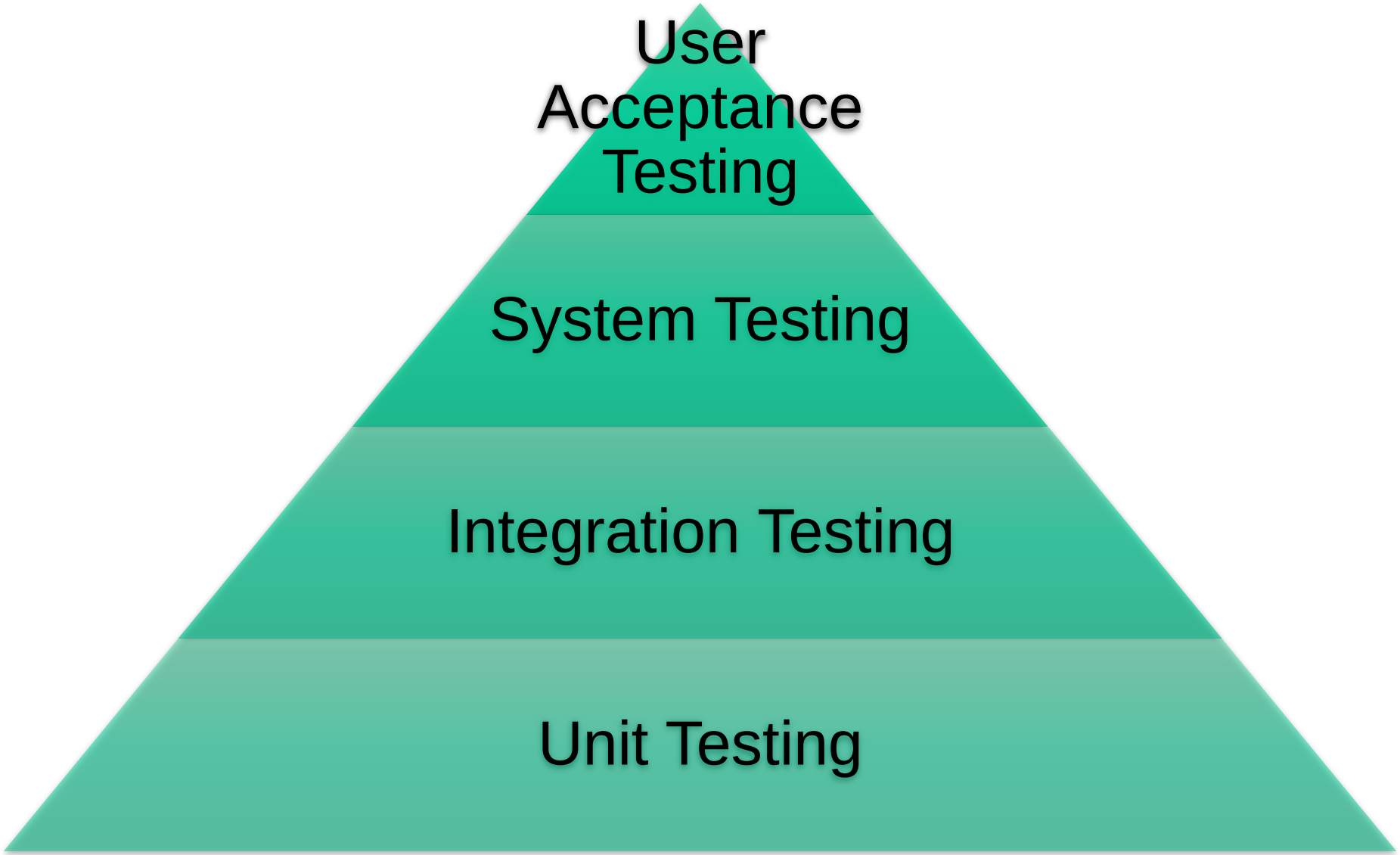
Result Analysis and Reporting

- Reporting problems
  - Short Description
  - Where the problem was found
  - How to reproduce it
  - Severity
  - Priority
- Can this problem lead to new test case ideas?

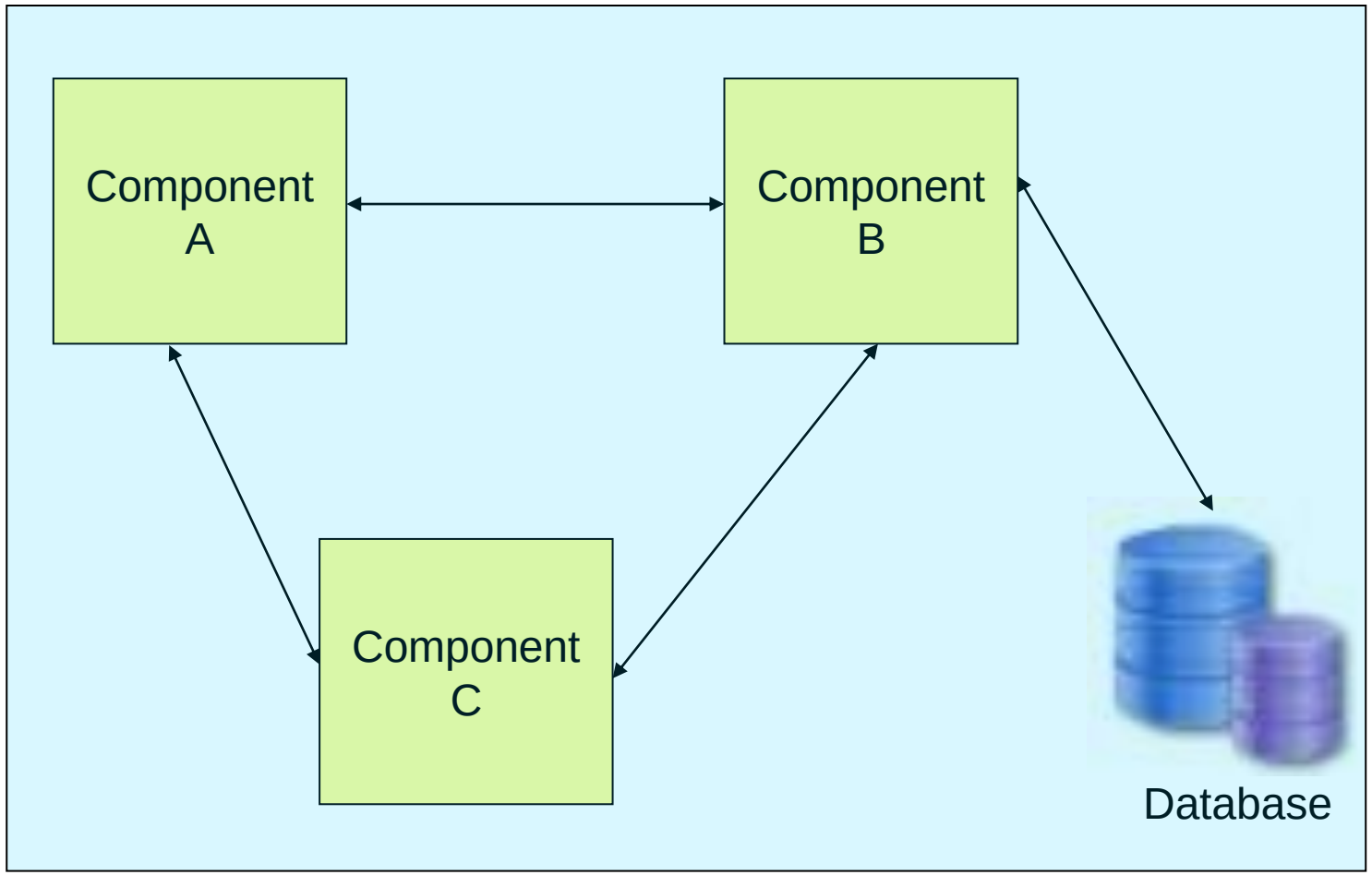
Test Control, Management and Review

- Exit criteria should be used to determine when testing should stop.
  - Criteria may include:
    - Coverage analysis
    - Faults pending
    - Time
    - Cost
- Tasks in this stage include
  - Checking test logs against exit criteria
  - Assessing if more tests are needed
  - Write a test summary report for stakeholders

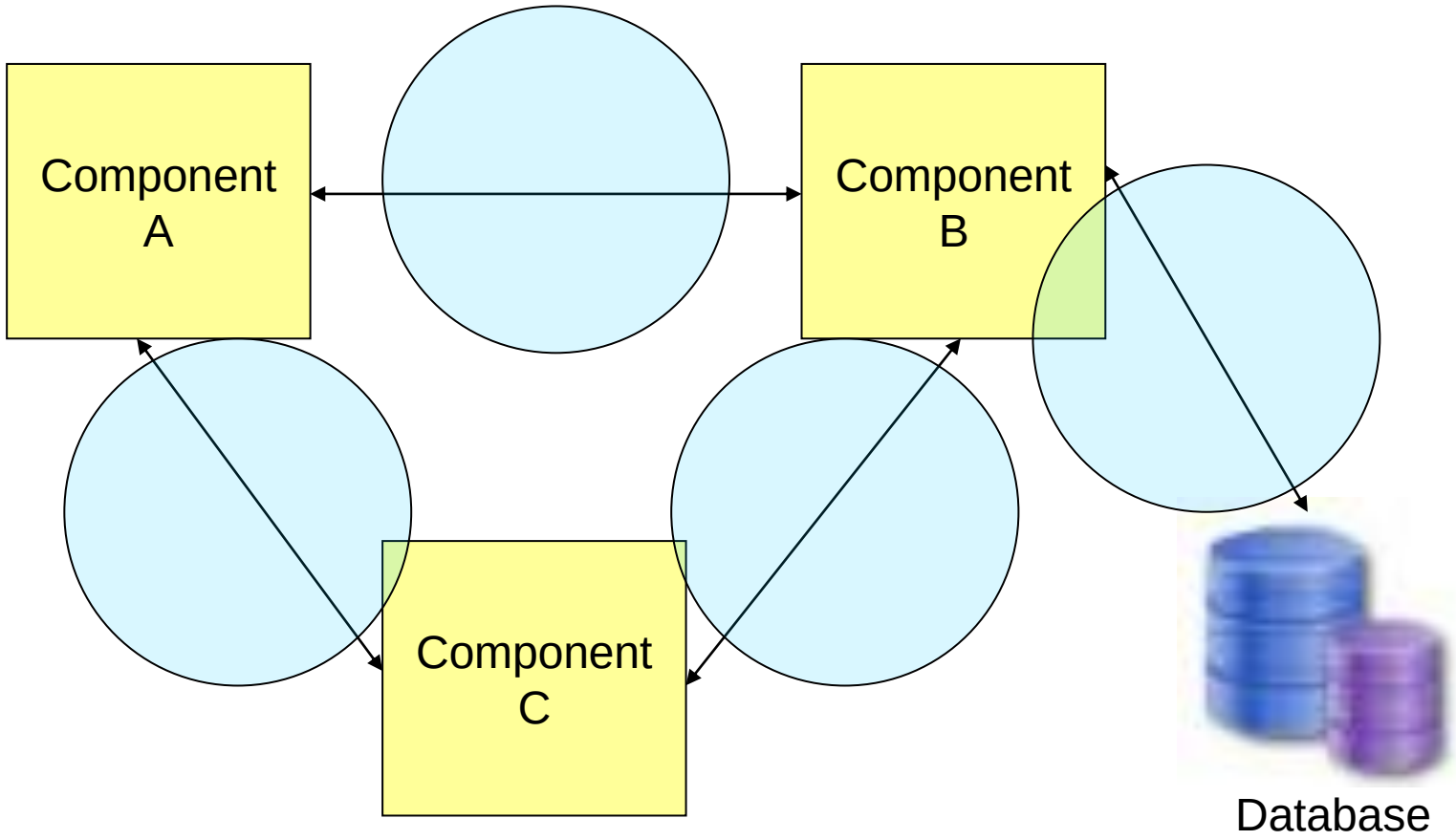
Levels of Testing



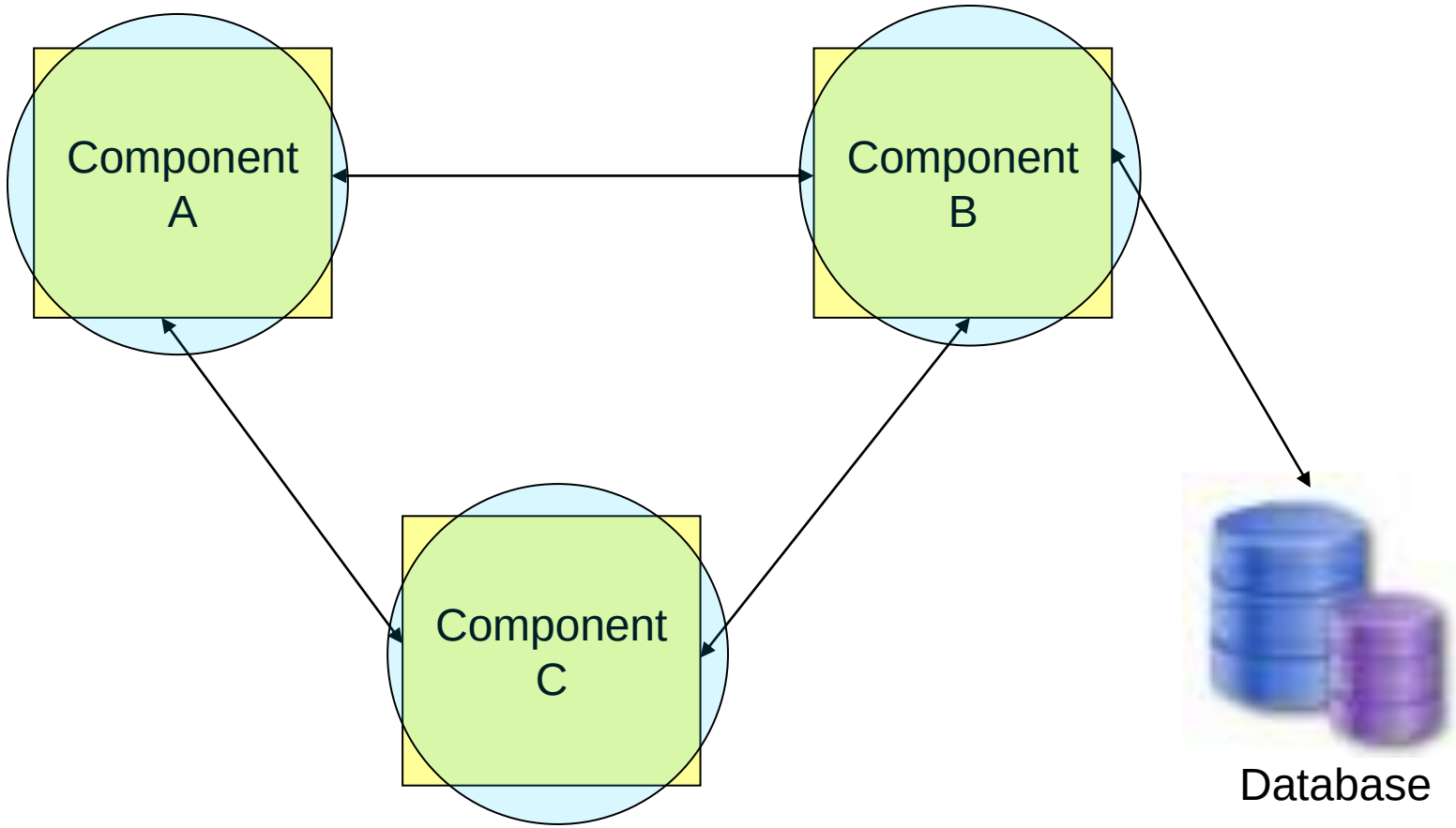
System Testing



Integration Testing



Unit Testing



SQA pllan

SQA Plan

The software quality assurance plan is one of the most important plans that should be prepared before embarking on a software development project.

The following details are recorded in the software quality assurance plan:

1. **Standards**—Include coding guidelines, design guidelines, testing guidelines, etc. selected for use in the project. These standards ensure a minimum level of quality in software development as well as uniformity of output from the project resources.

2. **Quality control activities**—Proposed activities for the project include code walkthrough, requirements and design review, and tests (unit testing, integration testing, functional testing, negative testing, endto-end testing, system testing, acceptance testing, etc.).

SQA Plan

4. **Procedures and events that trigger causal analysis**—Include failures, defects, and successes.

5. **Audits**—To analyze the exceptions in the project so that necessary corrective and preventive actions are taken to ensure the exceptions do not recur in the project.

6. **Institute of Electrical and Electronics Engineers Standard 730**—Gives details on how to prepare a quality assurance plan, including a suggested template.

SQA Plan

Details of the following standards should be included in the software quality assurance plan to guide project personnel in carrying out their assignments effectively and with the desired levels of productivity and quality:

- Coding standards for the programming languages used in the project
- Database design standards
- Graphical user interface design standards
- Test case design standards
- Testing standards
- Review standards
- Organizational process reference

SQA Plan

The following specifications of quality levels (quality metrics) for the project should be stated in the software quality assurance plan:

- Defect injection rate
- Defect density
- Defect removal efficiency for various quality assurance activities
- Productivity for various artifacts of the project
- Schedule variances

SQA Plan

The following quality control activities proposed to be implemented in the project should be included in the software quality assurance plan:

Code walkthrough

Peer review

Formal review

Various types of software tests that would be carried out during project execution, which at a minimum should include the following:

- Unit testing
- Integration testing
- System testing
- Acceptance testing

SQA Plan

It also should contain the schedules for the following audits proposed for the project:

- Periodic conformance audits
- Phase-end audits
- Investigative audits (and criteria)
- Delivery audits

Review

Review

The **design document** is key.

It is checked **repeatedly** in the development process.

Typically, reviewed many times before getting a stamp of approval to proceed with development.

Unfortunately, we often don't find our own errors and thus we need others for reviews.

Different stakeholders with different viewpoints are used in the review process.

Review

A **review process** is : “a process or meeting during which a work product or set of work products is presented to project personnel, managers, users, customers, or other interested parties for comment or approval.” (IEEE)

Essential to detect / correct errors in these earlier work products because the **cost of errors downstream is very expensive**!!

Review Choices:

- Formal Design Reviews (FDR)
- Peer reviews (inspections and walkthroughs)
- Used especially in design and coding phase

Review objectives

Direct objectives – Deal with the current project

a. To detect analysis and design errors as well as subjects where corrections, changes and completions are required

b. To identify new risks likely to affect the project.

c. To locate deviations from templates, style procedures and conventions.

d. To approve the analysis or design product. Approval allows the team to continue on to the next development phase.

Indirect objectives – are more general in nature.

a. To provide an informal meeting place for exchange of professional knowledge about methods, tools and techniques.

b. To record analysis and design errors that will serve as a basis for future corrective actions. (very important)

Review objectives

Many different kinds of reviews that apply to different objectives.

Reviews are not randomly thrown together.

Well-planned and orchestrated.

Objectives, roles, actions, participation, …. Very involved tasks.

Participants are expected to contribute in their area of expertise.

Idea behind reviews is to **discover problems <u>NOT</u>** to fix them/

Typically fixed after review and ‘offline’ so to speak.

Very common to review design documents.

Thus they are **usually well-prepared** initially prior to review.

Formal Design Reviews

DPR – Development Plan Review
SRSR – Software Requirement Specification Review
PDR – Preliminary Design Review
DDR – Detailed Design Review
DBDR – Data Base Design Review
TPR – Test Plan Review
STPR – Software Test Procedure Review
VDR – Version Description Review
OMR – Operator Manual Review
SMR – Support Manual Review
TRR – Test Readiness Review
PRR – Product Release Review
IPR – Installation Plan Review

Important to note that a design review can take place any time an analysis or design document is produced, regardless whether that document is a requirement specification or an installation document.

Blackbox testing technique

Blackbox testing

- Blackbox testing is a technique for testing without knowing software source code.
- Blackbox testing (also called behavioral or behavior-based techniques) are based on an analysis of the appropriate test basis (e.g., formal requirements documents, …).
  - are applicable to both functional and non- functional testing.
  - concentrate on the inputs and outputs of the test object without reference to its internal structure.

Test Techniques

The purpose of a test technique is to help in identifying test conditions, test cases, and test data.

The choice of which test techniques depends on a number of factors:

- Component or system complexity
- Regulatory standards
- Customer or contractual requirements
- Risk levels and types
- Available documentation
- Tester knowledge and skills
- Available tools
- Time and budget
- Software development lifecycle model
- The types of defects expected in the component or system

Test Techniques (cont.)

Some techniques are more applicable to certain situations and test levels; others are applicable to all test levels.

When creating test cases, testers generally use a combination of test techniques to achieve the best results from the test effort.

The use of test techniques in the test analysis, test design, and test implementation activities can range from very informal (little to no documentation) to very formal.

The appropriate level of formality depends on the context

Blackbox test design technique

a. Equivalence Class Partitioning.
b. Boundary value analysis.
c. Decision Tables.
d. State Transition.
e. Pairwise testing.

Whitebox testing technique

White-box Testing (WBT) introduction

WBT relies on a specific algorithm, on the internal data structure of the module to be tested, to determine if the module is performing correctly.

Therefore, WBT tester must have skills and knowledge to be able to understand in detail about the code to be tested.

WBT usually takes a lot of time and effort

For important modules, which perform the main computation of the system, this approach is necessary.

White box testing methods:

- Control flow testing
- Data flow testing

CHAPTER 3

REQUIREMENTS AND ANALYSIS

REQUIREMENTS

Requirement steps

- Concept exploration
  - Discover term/concepts in the application domain
  - Build the glossary list
- Business model
  - Description by natural language
  - Description by UML

Concept exploration

- Glossary list
  - Discover: brain storming, teamwork
  - Organise into glossary list

No.	Your language	English	Meanings
Category 1			
1	...		
2			
Category 2			
3	...		
4			

BM: natural language (1)

- Objective?
- Scope?
- How do the modules work?
- Information about objects?
- Relationships among objects?

BM: natural language (1)

- Objective
  - Description about the system
- Scope
  - Which type of application? (web, descktop, mobile)
  - Who can directly use the system?
  - Who can indirectly use the system?
  - What are the functions that each user could do?

BM: natural language (2)

- Operating in each function
  - Description about the order of steps to process in the function
  - Description the information displayed in each step
  - Description the action of the user in each step
  - Description all possible cases could happen after an user action at each step

BM: natural language (3)

- Object information in the system
  - Detect all entities/objects need to be managed or used in the system
  - Detect all necessary attributes for each entity/object in the system
  - Detect the data type and the value range of each attribute of entity/object.

BM: natural language (4)

- Relationships among objects in the system
  - Detect all possible quantity relationships among entities/objects
  - 1-1 relationship (zero or one, exactly one)
  - 1-n relationship (zero or more, one or more)
  - n-n relationship (zero or more, one or more)

BM: UML (1)

- General use case diagram (for the whole system)
- Detail use case diagram (for each function)

BM: UML (2)

- General use case diagram (for the whole system)
  - Detect actors of the system
  - Detect use cases for each actor
  - Refine the diagram

BM: General use case (1)

- Detect actors of the system
  - Input: the scope of the system by natural language
  - Each (direct/indirect) user → create an (direct/indirect) actor
  - Proposal some abstract actors if necessary

BM: General use case (1)

- Detect actors of the system
  - Input: the scope of the system by natural language
  - Each (direct/indirect) user → create an (direct/indirect) actor
  - Proposal some abstract actors if necessary

BM: General use case (2)

- Detect use cases of actor
  - In: the scope of the system by natural language
  - Each function of an user → create an use case for the corresponding actor
- Refine use case:
  - Proposal some abstract use cases if necessary

BM: Detail use case (1)

- Extract the main use case from the general UC
- Detect related sub use cases
- Detect the relationship to the main usecase
- Description each use case

BM: Detail use case (2)

- Extract the main use case from the general UC
  - Extract the actor(s) and the main use case from the general use case diagram
  - Extract the relationships among extracted actor(s) and the extracted main UC
- Detect related sub use cases
  - Input: description of the function operating in NL
  - Each interface with user → propose a sub use case
  - Ignore all alerting, confirmation or simple messages

BM: Detail use case (3)

- Detect relationships to the main UC
  - For each new sub use case, detect if it has include/extend relationship to the main UC
  - Some similar use cases may have the same abtract parent use case
- Description of use case
  - Each use case has a brief description: This use case enables who (someone) to do what (something)

Requirement

APPLY TO THE CASE STUDY

ANALYSIS

Analysis steps

- Scenarios
- Entity class extraction
- State diagram
- Module class diagram
- Sequence/communication diagram

Scenarios

- The standard scenario
  - There is no error in the operation of the system
  - There is no error or illogic in the manipulation of the actor
- All exception scenarios
  - In case of error or inexpected results

Entity class diagram

- Extract entity class
  - Extract all nouns from all related scenarios
  - Consider if the noun could represent an entity class or not
  - Detect all neccesary attribute of each entity class
- Detect relationships among entity classes
  - 1-1 relationship: could be merged
  - 1-n relationship: let's it be
  - n-n relationship: propose more intermediate class(es) between them, if neccesary

State diagram

- Propose states
  - An interface to interact to user → a state
  - Ignore simple message
- Relationships among states
  - Trigger to change state: user action

Class diagram of module(1)

- Boundary/view/interface classes
  - Input: all scenarios + state diagram
  - An interface to interact to user or a state → a view class
  - Detect attributes of each view class:
    - Input attribute
    - Output attribute
    - Control/redirect attribute
    - Combined of them

Class diagram of module(2)

- Processing at the lower level
  - Each data processing → create a method
  - Detect input/output data
  - Assign the method to a related entity class:
    - Output related entity
    - Input related entity
- Relationships among classes
  - Extract all related relationships among entity classes of the module
  - Create a relationship if there are interaction between two view classes or between a view class and an entity class,

Sequence/communication diagram

- Scenario v.2
  - The user clicks on the interface of class X
  - Class X calls/requires class Y to do A
  - Class Y does method A...
- Diagram
  - A scenario has a diagram
  - The entity classes are also control classes (have actions/methods)
  - Convert from sequence to communication diagram

Analysis

APPLY TO THE CASE STUDY

CHAPTER 4

DESIGN

Design steps

- Design of entity classes
- Design of database
- Design of interface
- Detail design module class diagram
- Activity diagram of module
- Sequence/communication diagram
- Package diagram
- Deployment diagram

Entity class diagram at the design phase

- Input
  - The entity class diagram at the analysis phase
- Process
  - Add id attribute to all classes that are not inherited from any class
  - Design the datatype to all attributes
  - Convert association relationships to aggregation/composition relationships
  - Add the object attributes corresponding to the aggregation/composition relationships

Design of database (1)

- Input
  - The entity class diagram of the design phase
- Process
  - An entity class → create a table
  - Non-object attribute of the class → attritue of the corresponding table
  - Quantity relationships between two classes → quantuty relationships between the two corresponding tables
    - 1-1: should merged
    - 1-n: let's it be
    - n-n: return to the entity class diagram to correct it

Design of database (2)

- Process
  - Key attributes
    - Primary key: the id of the tables which have it
    - Foreinger key: if tblA – tblB have an 1-n relationship → the tblB must have a FK which refers to the PK of the tblA.
  - Remove redondant attributes
    - Doublicate attributes
    - Secondary attributes

Design of interface

- Process
  - An interface to interact to user
  - Combination of some simple interfaces into one
  - Message/dialogue/confirmation/Alert

Class diagram of module (1)

- View classes
  - Input: interface design
  - An interface → a view class
  - Design explicite attributes view class:
    - Input attribute
    - Output attribute
    - Control/redirect attribute
    - Combined of them
  - Design implicite attributes of view class
    - To receive data from previous class

Class diagram of module (2)

- Processing at the lower level
  - Each data processing → create a method
  - Design input parameters
  - Design output parameters
  - Assign the method to a related DAO class:
    - Output related entity
    - Input related entity
- Relationships among classes

Activity diagram of module

- Process
  - Processing at an interface → an activity
  - Each method → action
  - Consider all possible cases

Sequence/communication diagram

- Scenario v.3
  - The user clicks on the interface of class X
  - Class X calls/requires class Y to do A
  - Class Y does method A...
- Diagram
  - A scenario has a diagram
  - Each method has a sub-life line
  - Convert from sequence to communication diagram

Package and deployment diagram

- Package
  - Present all packages
  - All classes included in each package
- Deployment
  - Site of database
  - Site of server(s)
  - Site of client

APPLY TO THE CASE STUDY

CHAPTER 5

REVIEW & TESTING

Reference

- This chapter refers from the book:
  - Mastering Software Quality Assurance: Best Practices, Tools and Techniques for Software Developers
  - Introduction to Software Testing

REVIEW

Reviewing steps

- The review will ensure that the documents (specification, analysis, design, etc.) are free of errors. Usually, we will review based on the corresponding checklists.
- The process steps are as follows:
  - Step 1: Select and develop a checklist suitable for the type of review.
  - Step 2: Review the questions in the checklist. In case there is an error or problem, it is necessary to clearly describe the error and request correction.
- Following slides demonstrate a checklist

Checklist for reviewing user requirements specification (1)

Question
Are the requirements in compliance with the contract?
Have all the requirements been listed?
Are there any ambiguous requirements?
Is each requirement described completely?
Have the requirements been specified consistently throughout the document?
Can the requirements be verified?
Has any additional functionality been included beyond the scope of the contract?
Are project management requirements included in the requirements?

Checklist for reviewing user requirements specification (2)

Question
Is the rationale for any derived requirements satisfactory?
Are the specified external interfaces compatible?
Are the user interface requirements complete?
Can the requirements be tested? Can the requirements be used directly for validation during acceptance testing?
Are the performance requirements adequate and feasible?
Have the security requirements been determined?
Do any requirements conflict with or duplicate other requirements?

Checklist for reviewing user requirements specification (3)

Question
Is each requirement written in clear, concise, unambiguous language?
Is each requirement free of content and grammatical errors?
Are the time-critical functions identified, and are the timing criteria for them specified?
Have internationalization issues been adequately addressed?
Is the format in conformance with the format in the organizational process?

Checklist for reviewing user requirements specification (4)

Question
Are all internal cross-references to other requirements correct?
Do the requirements provide an adequate basis for software re- quirement specification?
Have algorithms intrinsic to the functional requirements been defined?
Is each requirement in scope for the project?
Are all security and safety considerations properly specified?

REVIEW

APPLY TO THE CASE STUDY

TESTING

Functional testing

- Functional testing is to ensure that the system's functions work according to the requirements of the specification.

Functional testing

The process steps are as follows:

- Step 1: Build a checklist of the contents to be tested for each function (optional).
- Step 2: Write test cases
  - GUI (interface)
  - Function
  - stream, business
  - other
- Step 3: Prepare test data (if necessary)
- Step 4: perform the test and record the pass/false results. In case of false, it is necessary to clearly describe the error and discuss with the Programmer to correct it.

Writing test cases

- The main contents of Test case include description of input (test purpose, execution steps, test data) and desired result.
- One note is that some test cases below don't require test data (e.g. interface test, action test...), but some test cases need test data (e.g. functional test case...).

APPLY TO THE CASE STUDY