2022-2023
CPT203 Revision

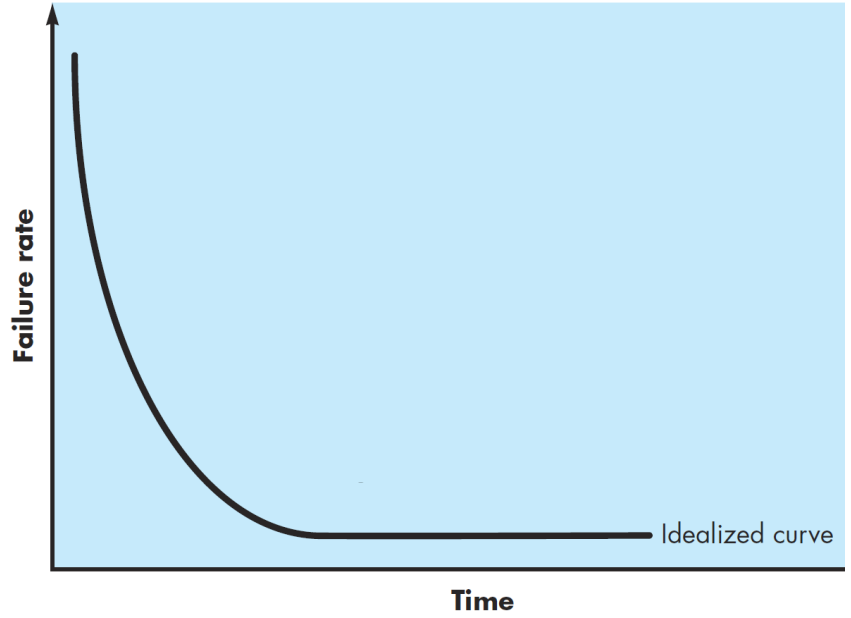
Week 1 To Week 7

Exam Format

- Online Exam
- Answer sheet format
- Visual Paradigm
- Visual Code

Software Deterioration

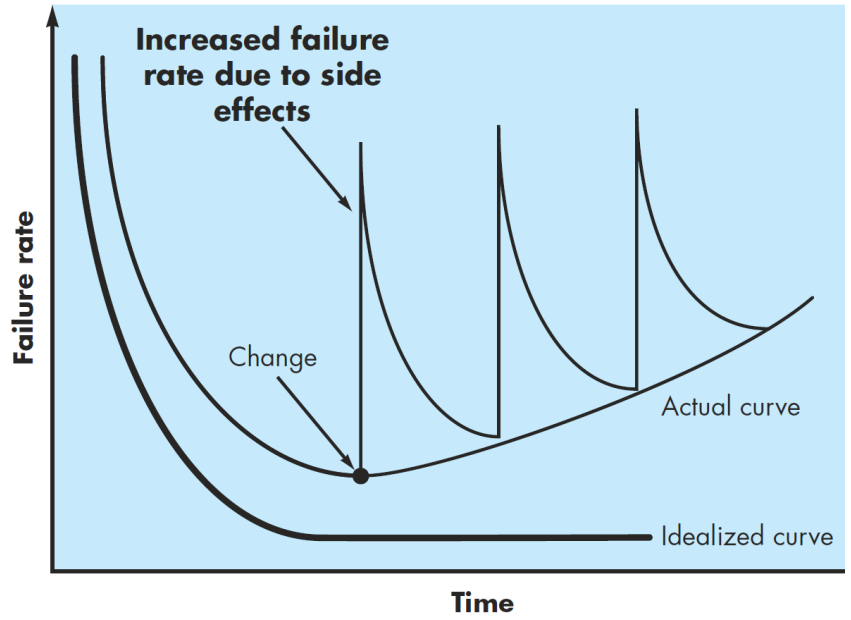
- Software is a logical rather than a physical system element. Therefore, software has one fundamental characteristic that makes it considerably different from hardware: *Software doesn't* ***"wear out."***
- In theory, therefore, the failure rate curve for software should take the form of the "idealized curve" shown in Figure



Software Deterioration

- Software doesn't wear out. But it does ***deteriorate***
- During its life, software will undergo change. As changes are made, it is likely that errors will be introduced, causing the failure rate curve to spike as shown in the "actual curve" (Figure 1.2).
- Before the curve can return to the original steady-state failure rate, another change is requested, causing the curve to spike again.

Software engineering methods strive to reduce the magnitude of the spikes and the slope of the actual curve



Software Engineering Approaches

- The systematic approach that is used in software engineering is sometimes called a software process.
- A software process is a sequence of activities that leads to the production of a software product.
- There are four fundamental activities that are common to all software processes: -
  - Software specification
  - Software development
  - Software validation
  - Software evolution

Software Process Models

- A software process model is a simplified representation of a software process.
- Discuss very general process models and present these from an architectural perspective, framework of the process rather than the details of specific activities.
- These generic models are not definitive descriptions of software processes, they are abstractions of the process that can be used to explain different approaches to software development
- You can think of them as process frameworks that may be extended and adapted to create more specific software engineering processes.

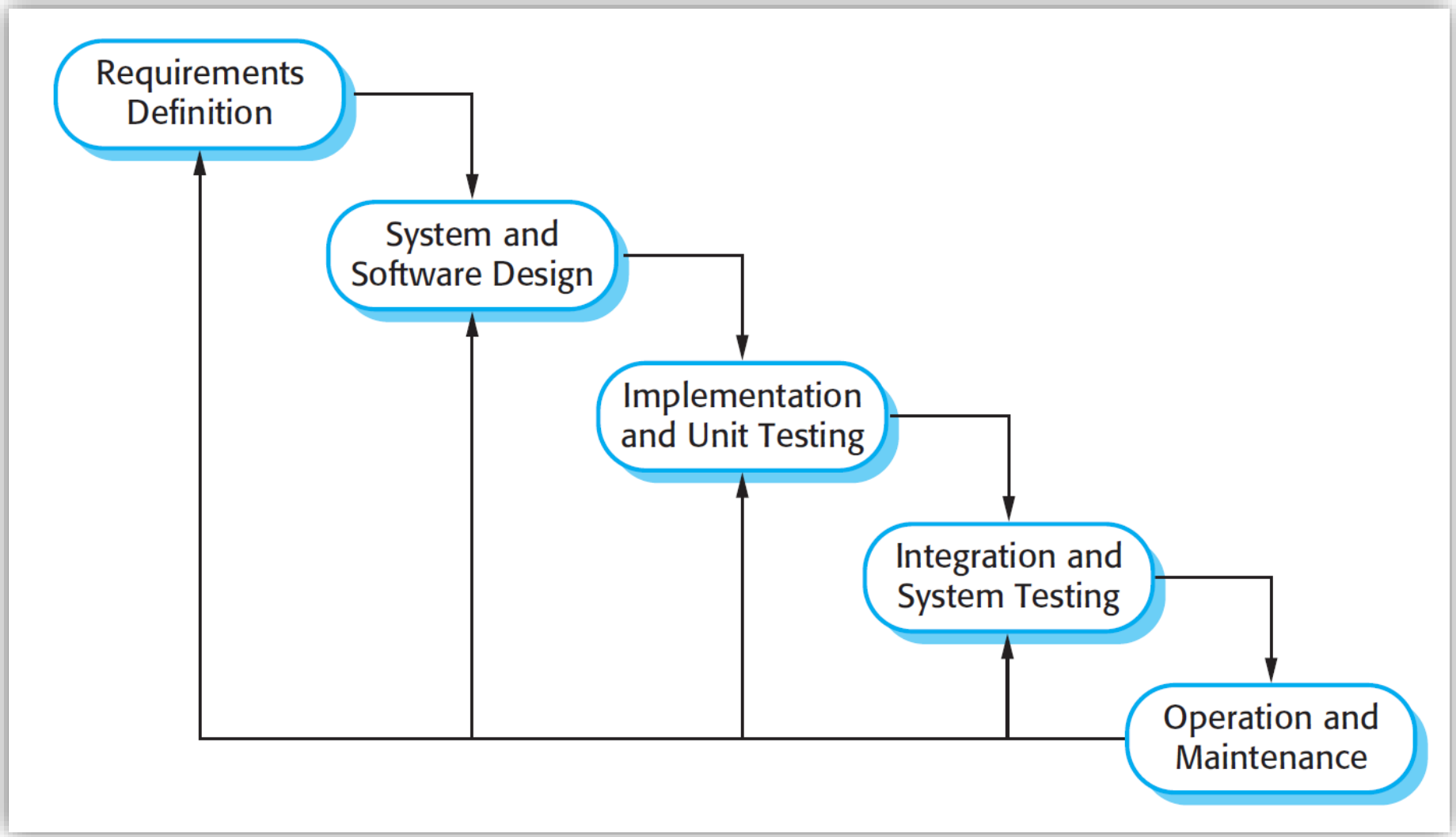
Software Process Models

- The models in our discussion: -
  - The waterfall model - This takes the fundamental process activities of specification, development, validation, and evolution and represents them as separate process phases such as requirements specification, software design, implementation, testing, and so on.
  - Incremental development - This approach interleaves the activities of specification, development, and validation. The system is developed as a series of versions (increments), with each version adding functionality to the previous version.
  - Reuse-oriented software engineering - This approach is based on the existence of a significant number of reusable components. The system development process focuses on integrating these components into a system rather than developing them from scratch.

Software Process Models

- These models are not mutually exclusive and are often used together, especially for large systems development.

The Waterfall Model



The Waterfall Model

- Because of the cascade from one phase to another, this model is known as the ‘waterfall model’ or software life cycle. The waterfall model is an example of a plan-driven process
- The principal stages of the waterfall model directly reflect the fundamental development activities:
  - Requirements analysis and definition
  - System and software design
  - Implementation and unit testing
  - Integration and system testing
  - Operation and maintenance

The Waterfall Model

- In principle, the result of each phase is one or more documents that are approved (‘signed off’). The following phase should not start until the previous phase has finished.
- In practice, these stages overlap and feed information to each other
- The software process is not a simple linear model but involves feedback from one phase to another.
- Documents produced in each phase may then have to be modified to reflect the changes made

The Waterfall Model

- Because of the costs of producing and approving documents, iterations can be costly and involve significant rework.
- Therefore, after a small number of iterations, it is normal to freeze parts of the development, such as the specification, and to continue with the later development stages.
- Problems are left for later resolution, ignored, or programmed around.
- This premature freezing of development tasks may mean that the system won’t deliver what the users are expecting.

Incremental Development

- Incremental development is based on the idea of developing an initial implementation, exposing this to user comment and evolving it through several versions until an adequate system has been developed
- Specification, development, and validation activities are interleaved rather than separate, with rapid feedback across activities.
- Incremental software development, which is a fundamental part of agile approaches, is better than a waterfall approach for most business, e-commerce, and personal systems

Incremental Development

- By developing the software incrementally, it is cheaper and easier to make changes in the software as it is being developed.
- Each increment or version of the system incorporates some of the functionality that is needed by the customer.
- Generally, the early increments of the system include the most important or most urgently required functionality. This means that the customer can evaluate the system at a relatively early stage in the development to see if it delivers what is required.
- If not, then only the current increment has to be changed and, possibly, new functionality defined for later increments

Incremental Development

- Incremental development has three important benefits, compared to the waterfall model:
  1. The cost of accommodating changing customer requirements is reduced.
  2. It is easier to get customer feedback on the development work that has been done.
  3. More rapid delivery and deployment of useful software to the customer is possible, even if all of the functionality has not been included.

Incremental Development

- From a management perspective, the incremental approach has two problems:
  - The process is not visible.
  - System structure tends to degrade as new increments are added.

- Other problems with incremental development includes: -
  - large organizations have bureaucratic procedures that have evolved over time and there may be a mismatch between these procedures and a more informal iterative or agile process
  - Formal procedures are required by external regulations (e.g., accounting regulations)

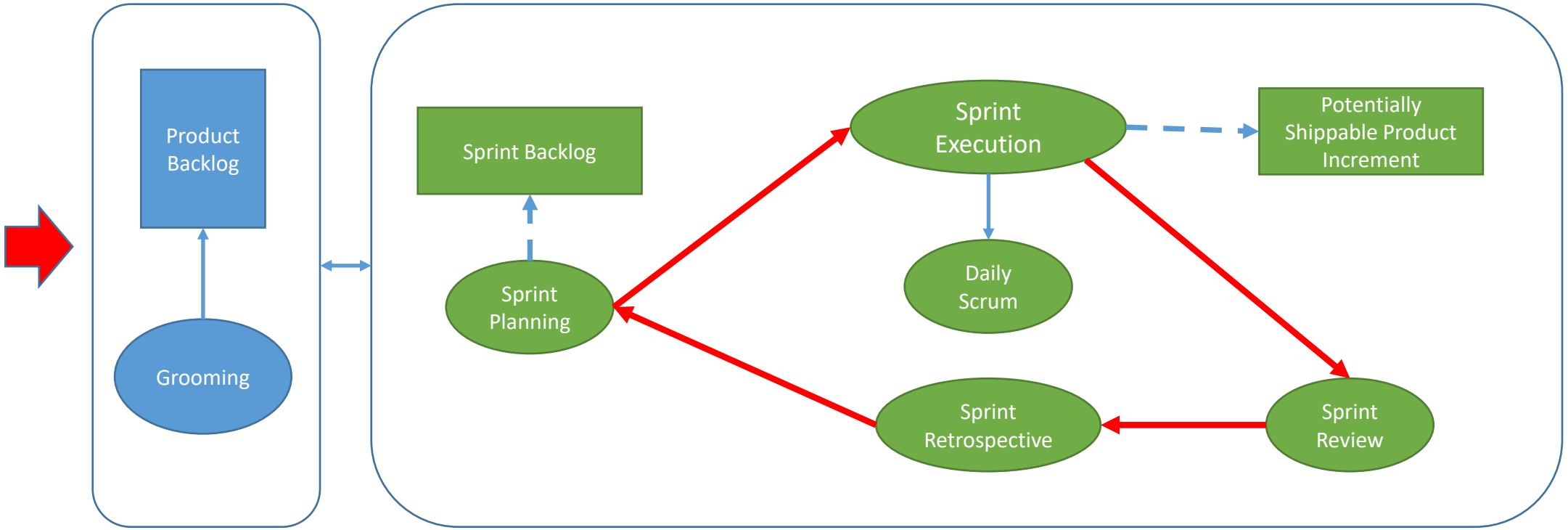
Incremental Development

- The problems of incremental development become particularly acute for large, complex, long-lifetime systems, where different teams develop different parts of the system.
- Large systems need a stable framework or architecture and the responsibilities of the different teams working on parts of the system need to be clearly defined with respect to that architecture.
- This has to be planned in advance rather than developed incrementally.

Agile Methods

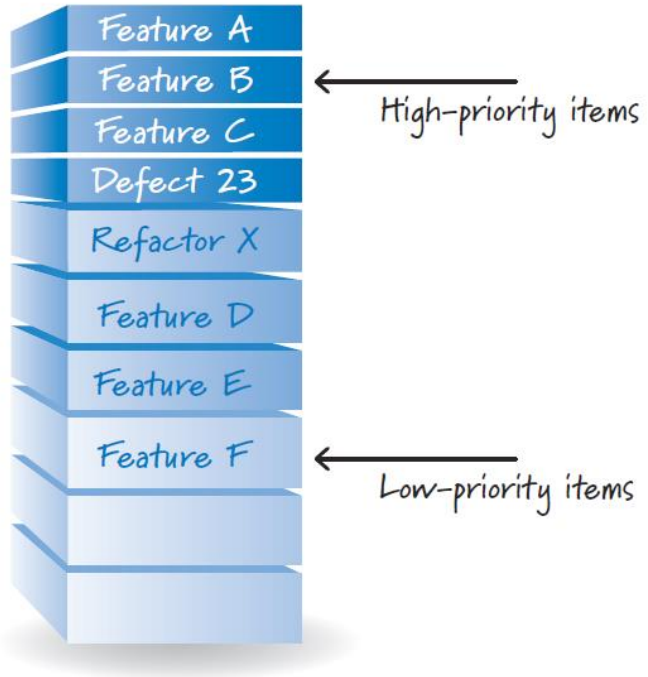
- The philosophy behind agile methods is reflected in the agile manifesto that was agreed on by many of the leading developers of these methods.
  - *Individuals and interactions over processes and tools*
  - *Working software over comprehensive documentation*
  - *Customer collaboration over contract negotiation*
  - *Responding to change over following a plan*

Scrum Activities and Artifacts



Product Backlog

- The product owner, with input from the rest of the Scrum team and stakeholders, is ultimately responsible for determining and managing the sequence of works (product backlog items) and communicating it in the form of a prioritized (or ordered) list known as the **product backlog**



Product Backlog

- On new-product development the product backlog items initially are features required to meet the product owner’s vision.
- For ongoing product development, the product backlog might also contain new features, changes to existing features, defects needing repair, technical improvements, and so on.
- Product owner collaborates with internal and external stakeholders to gather and define the product backlog items

Product Backlog

- High-value items appear at the top of the product backlog and the lower-value items appear toward the bottom.
- The product backlog is a constantly evolving artifact. Items can be added, deleted, and revised by the product owner as business conditions change, or as the Scrum team's understanding of the product grows
- In practice, many teams use a **relative size measure** such as **story points** or **ideal days** to express the item size

PBI Example

- User Story: Online user registration
- Description: As a user, I want to be able to register online, so that I can perform online shopping
- Acceptance Criteria:
  - User can register only if the user fills in all required fields
  - The email used in the registration must not be a free email
  - User will receive a notification email after successful registration

PBI Example

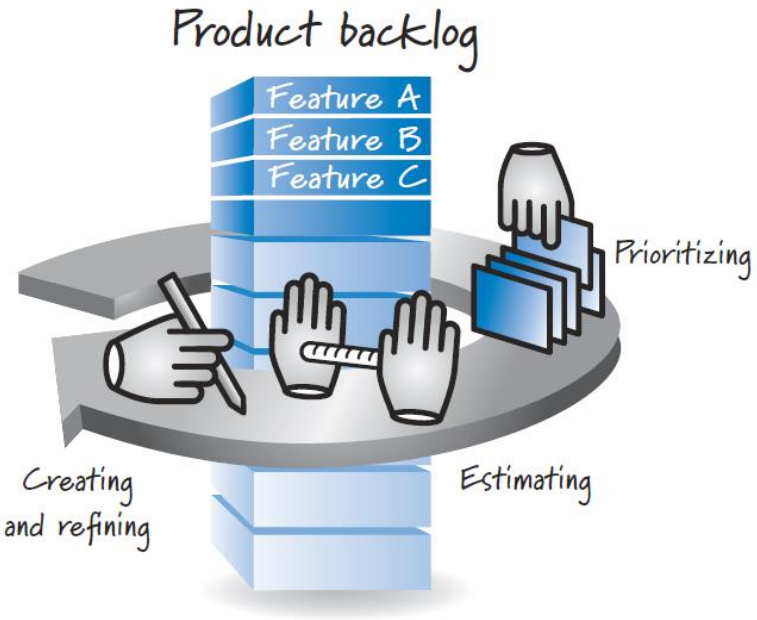
- User Story: Search for customer
- Description: As a marketing manager, I would like to search for customers, so that I can call them
- Acceptance Criteria:
  - Positive test: When I enter “Eddie” in the search box and click the search button, I will see all entries containing Eddie in a grid
  - Negative test: When I enter “ABC” in the search box and click the search button, I will see no entries in the grid
  - Gold plating: If no result, display a message
  - Gold plating: If a large set of results being returned, display in pagination
  - Gold plating: The user can click on the column heading of the grid to sort the information

PBI – User Story Format

- Description
  - As a user, I want to be able to register online, so that I can perform online shopping
  - As a marketing manager, I would like to search for customers, so that I can call them
- Acceptance Criteria
  - Given that I am at the Shopping Cart page, When I tick the checkbox of an item, Then the total of the item I ticked will add to the total amount of the order.
  - Given that the user selected a record, When (s)he click the Print button, Then the detail info of the record will be printed on the default printer.

Product Backlog Grooming

- The activity of creating and refining product backlog items, estimating them, and prioritizing them is known as grooming



User requirements and system requirements

- Different level of details serve different purpose. User requirements and system requirements
  - User requirements - Statements in a natural language plus diagrams to describe the services and constraint of a system
  - System requirements - more detailed descriptions of the software system’s functions, services, and operational constraints

Functional & Non-functional requirements

- Software system requirements are often classified as functional requirements or nonfunctional requirements:
  - Functional requirements These are statements of services the system should provide. How the services should react and behave in certain condition. In some cases, the functional requirements may also explicitly state what the system should not do.
  - Non-functional requirements These are constraints on the services or functions offered by the system. Non-functional requirements often apply to the system as a whole, rather than individual system features or services.

The distinction between different types of requirement is not as clear-cut as these simple definitions suggest

Functional Requirements

- Sample requirement for MHC-PMS system:
  1. A user shall be able to search the appointments lists for all clinics.
  2. The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
  3. Each staff member using the system shall be uniquely identified by his or her eight-digit employee number.

Imprecision in the requirements specification

Functional Requirements

- The functional requirements specification of a system should be both complete and consistent.
  - Completeness means that all services required by the user should be defined.
  - Consistency means that requirements should not have contradictory definitions.
- In practice, for large, complex systems, it is practically impossible to achieve requirements consistency and completeness.
  - it is easy to make mistakes and omissions when writing specifications for complex systems
  - there are many stakeholders in a large system. Stakeholders have different and often inconsistent needs.

Non-functional Requirements

- Non-functional requirements, as the name suggests, are requirements that are not directly concerned with the specific services delivered by the system to its users.
- System properties such as reliability, response time, and store occupancy.
- Constraints on the system implementation such as performance, security, or availability.
- Usually specify constrain characteristics of the system as a whole
- Often more critical than individual functional requirements

Non-functional Requirements

- Often more critical than individual functional requirements
- failing to meet a non-functional requirement can mean that the whole system is unusable
- The implementation of non-functional requirements may be intricately dispersed throughout the system.
  - They may affect the overall architecture of a system rather than the individual components.
  - A single non-functional requirement may generate a number of related functional requirements.

Non-functional Requirements - Testability

- A common problem with non-functional requirements is that users or customers often propose these requirements as general goals, such as …

The system should be easy to use by medical staff and should be organized in such a way that user errors are minimized.

vs

Medical staff shall be able to use all the system functions after four hours of training. After this training, the average number of errors made by experienced users shall not exceed two per hour of system use.

Non-functional Requirements - Testability

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Tutorial

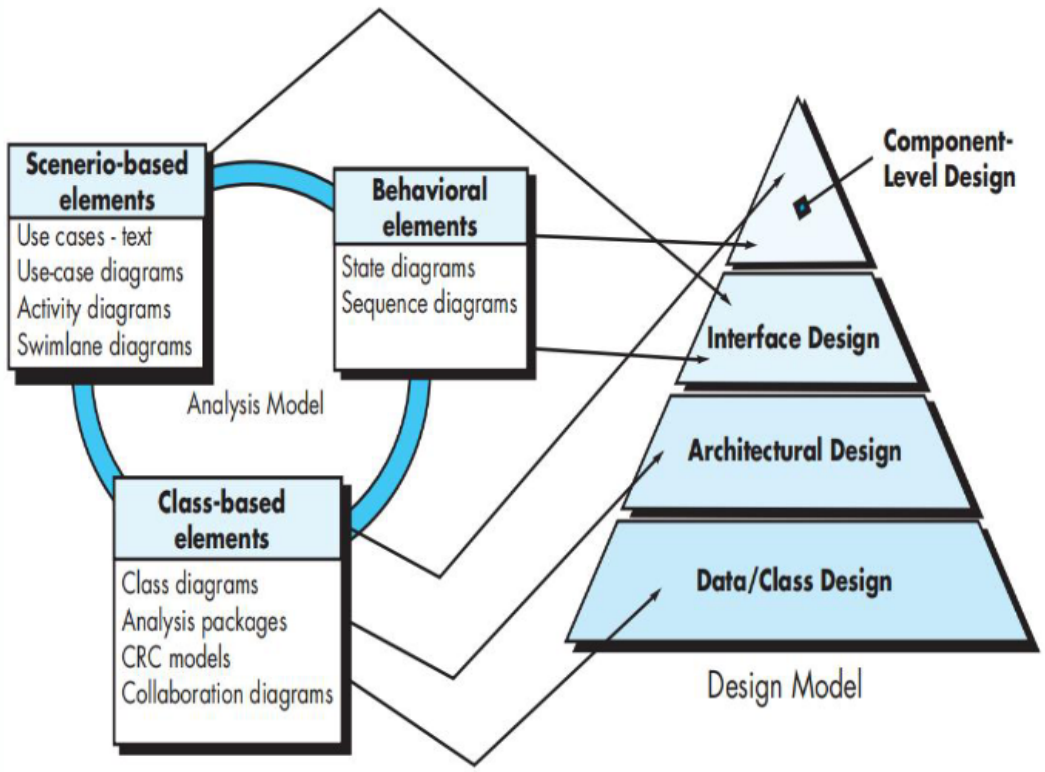
- T2Q1
- T2Q5
- T3Q2
- T3Q4
- T4Q2
- T4Q3

Week 9-13

- Software Design (week 9-10)
- Software Testing (week 11-12)
- Project Management (week 13)

Software Design

Design in the SE context



Component-level design: transforms structural elements of the software architecture into a procedural description of software components.
Interface design： describes how the software communicates with systems that interoperate with it, and with humans who use it
Architectural design： defines the relationship between major structural elements of the software, the architectural styles and patterns that can be used to achieve the requirements defined for the system, and the constraints that affect the way in which architecture can be implemented.
Data design/class design: transforms class models into design class realizations and the requisite data structure required to implement it.

Software Design

Design concepts

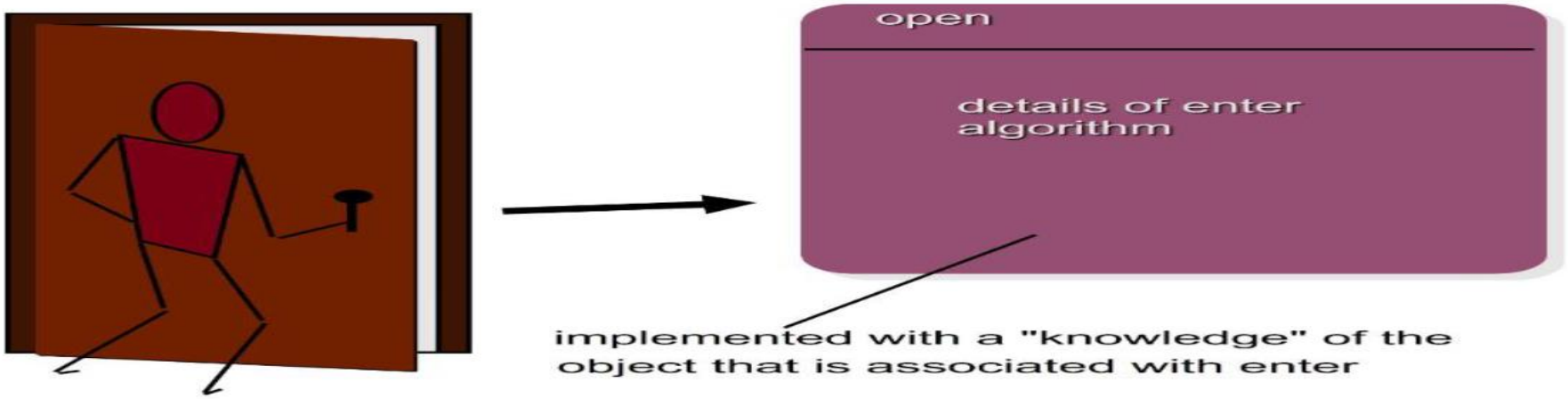
- Some general design concepts:
  - Abstraction
  - Modularity
  - Functional Independence
  - Coupling
  - Cohesion
  - Object-oriented design
  - Design classes

- More design concepts please refer to our textbook: *Software Engineering: A Practitioner's Approach, Chapter 12*

Software Design

Design concepts - Abstraction

- **A procedural abstraction** refers to a sequence of instructions that have a specifi c and limited function. The name of a procedural abstraction implies these functions, but specific details are suppressed.

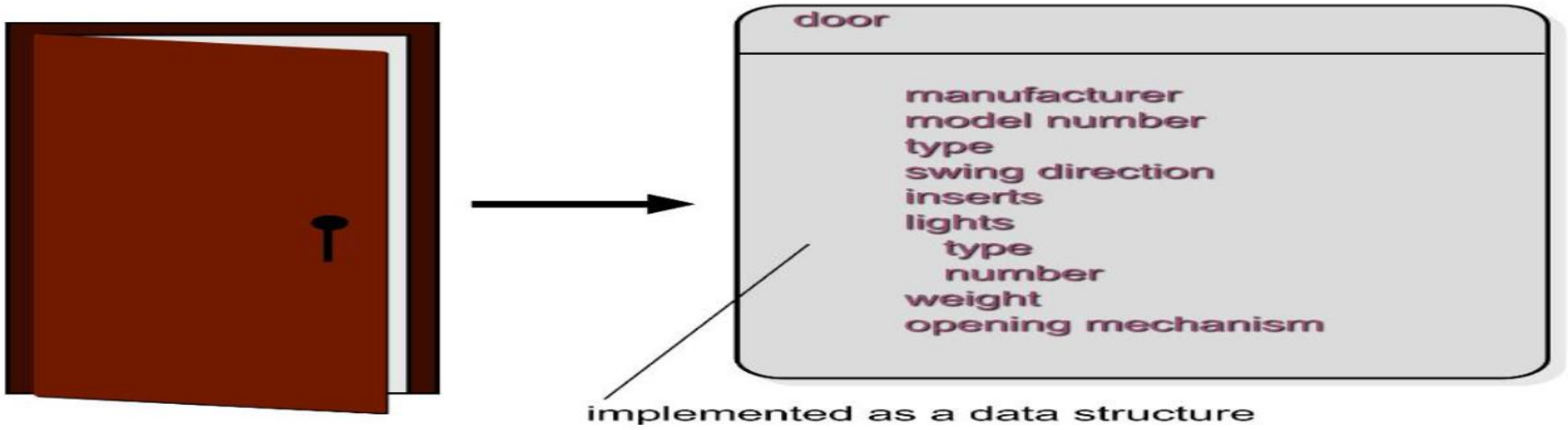


open()–> walk to the door, reach the knob, pull the door, step away

Software Design

Design concepts - Abstraction

- **A data abstraction** is a named collection of data that describes a data object.



Software Design

Design concepts - Coupling

- Coupling refers to the degree of interdependence between software modules
- Two modules are considered independent if one can function completely without the presence of the other.
- If two modules are independent, they are solvable and modifiable separately.
- However, all the modules in a system must interact so that together they produce the desired behavior of the system.
- The more connections between modules, the more dependent they are

Software Design

Design concepts - Coupling

- Tight coupling

```
class Author {
    String name;
    String skypeID;
    public String getSkypeID() {
        return skypeID;
    }
}
```

```
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.skypeID);
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

Tight coupling; nonpublic variable skypeID is referred to outside its class Author.

What happens, if a programmer changes the name of the variable skypeID in class Author to skypeName?

Software Design

- Loose coupling

```
class Author {
    String name;
    String skypeName;
    public String getSkypeID() {
        return skypeName;
    }
}
class Editor{
    public void clearEditingDoubts(Author author) {
        setUpCall(author.getSkypeID());
        converse(author);
    }
    void setUpCall(String skypeID) { /* */}
    void converse(Author author) {/* */}
}
```

Change in instance variable name won't affect classes that access this method

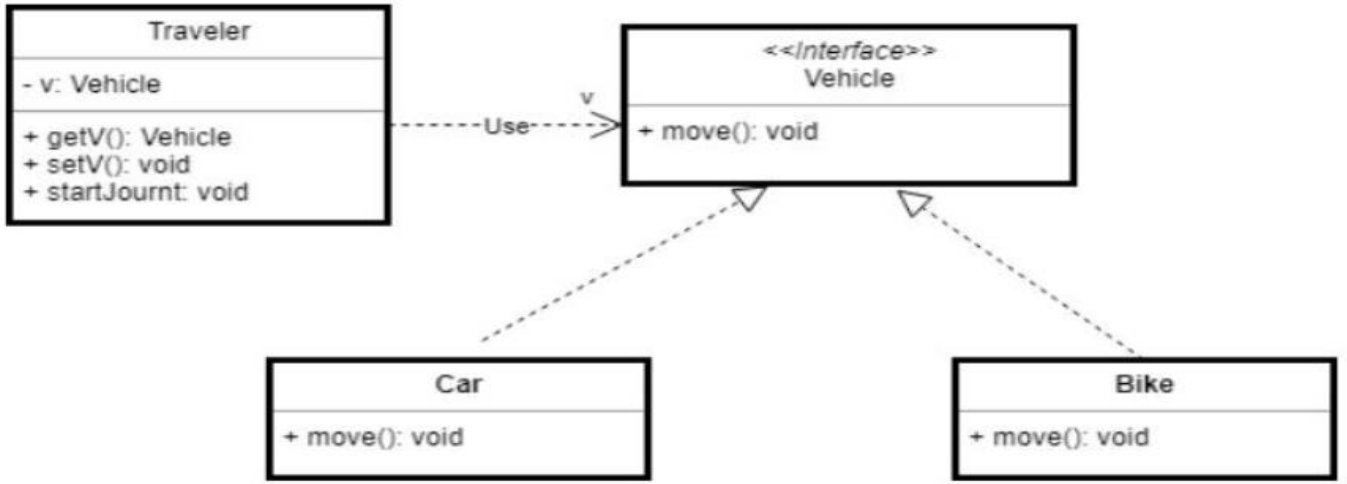
Loose coupling; public method getSkypeID() accesses Author's skypeName.

Use the public method getSkypeID() in class Editor (changes in bold)

Software Design

Design concepts - Coupling

Implementation Example



Software Design

Design concepts - Cohesion

- Another way to lower the coupling between modules is to strengthen the bond between elements of the same module by maximizing the relationship between elements of the same module.
- Cohesion is the degree of how closely the elements of a module are related to each other
- Cohesion refers to how focused a class or a module is.
- Cohesion of a module gives the designer an idea about whether the different elements of a module belong together in the same module.

Software Design

Design concepts - Cohesion

- Method cohesion example

```
Public Shared Sub Main()
    ' Create an instance of StreamWriter to write text to a file.
    Dim sw As StreamWriter = New StreamWriter("TestFile.txt")
    ' Add some text to the file.
    sw.Write("This is the ")
    sw.WriteLine("text for the file.")
    sw.Close()
End Sub
```

- The above method performs a sequence of tasks such as opening a file, write to a file, and close a file.
- How about a function sw.Read() ?

Software Design

Design concepts - Cohesion

- Class cohesion
  - A high class cohesion is the class that implements a single concept or abstraction with all elements contributing toward supporting this concept.
  - Highly cohesive classes are usually more easily understood and more stable.
  - Whenever there are multiple concepts encapsulated within a class, the cohesion of the class is not as high as it could be, and a designer should try to change the design to have each class encapsulate a single concept.

Software Design

Design concepts - Cohesion

- Class cohesion example

Fig: Low cohesion

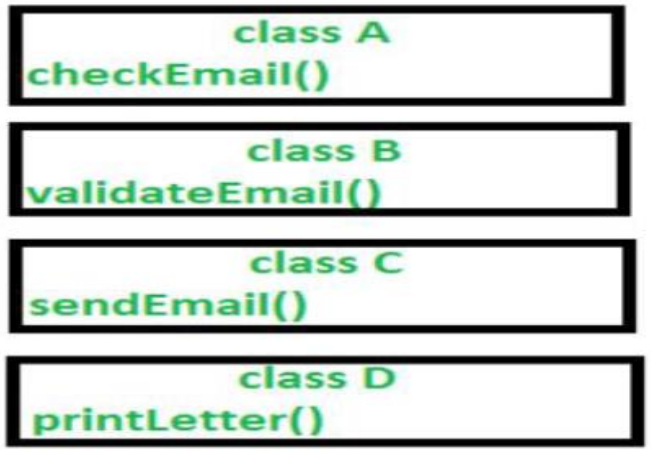


Fig: High cohesion

Software Design

- Architecture Design
  - Concept
  - Patterns
- Component-Level Design
  - Concept
  - Design Process
- Interface Design
  - Golden Rules
  - Design Process

Saoftware Testing

- Development Testing
  - Unit
  - Component
  - System
- Release Testing
- Acceptance Testing

Software Testing

- Static methods defined in org.junit.Assert
- Assert a Boolean condition is true or false

  assertTrue(*condition*)

  assertFalse(*condition*)

- Optionally, include a failure message

  assertTrue(*condition*,*message*)

  assertFalse(*condition*,*message*)

Software Testing

- Assert two object references are identical

  assertSame(*expected, actual*)

  - True if: expected == actual

  assertNotSame(*expected, actual*)

  - True if: expected != actual

- With a failure message

  assertSame(*expected, actual, message*)

  assertNotSame(*expected, actual, message*)

Software Testing

- Assert two objects are equal:

  assertEquals(expected, actual)
  - True if: expected.equals(actual)
  - Relies on the equals() method
  - Up to the class under test to define a suitable equals() method.
- With a failure message

  assertEquals(*expected, actual, message*)

Software Testing

- Assert two arrays are equal:

  assertArrayEquals(expected, actual)

  - arrays must have same length
  - Recursively check for each valid index i,

    assertEquals(expected[i],actual[i])

    or

    assertArrayEquals(expected,actual)

- With a failure message

  assertArrayEquals(*expected*, *actual*, *message*)

Software Testing

- A.k.a., robustness testing
- The expected outcome of a test is an exception.

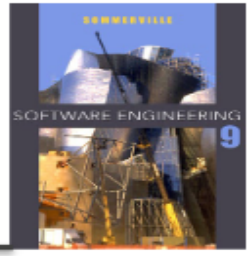
```
public class MathTools {
    public static double convertToDecimal(int numerator, int denominator) {
        if (denominator == 0) {
            throw new IllegalArgumentException("Denominator must not be 0");
        }
        return (double)numerator / (double)denominator;
    }
}
```

Software Testing

Annotation	Description
@BeforeEach	The annotated method will be run before each test method in the test class.
@AfterEach	The annotated method will be run after each test method in the test class.
@BeforeAll	The annotated method will be run before all test methods in the test class. This method must be static.
@AfterAll	The annotated method will be run after all test methods in the test class. This method must be static.
@Test	It is used to mark a method as a junit test.
@DisplayName	Used to provide any custom display name for a test class or test method
@Disable	It is used to disable or ignore a test class or test method from the test suite.
@Nested	Used to create nested test classes
@Tag	Mark test methods or test classes with tags for test discovering and filtering
@TestFactory	Mark a method is a test factory for dynamic tests.

Project Management

Outline



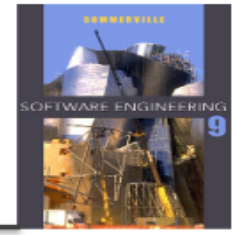
- ✧ Project Management Overview
- ✧ Risk Management
- ✧ Managing People
- ✧ Software Team and Teamwork
- ✧ Managing Product

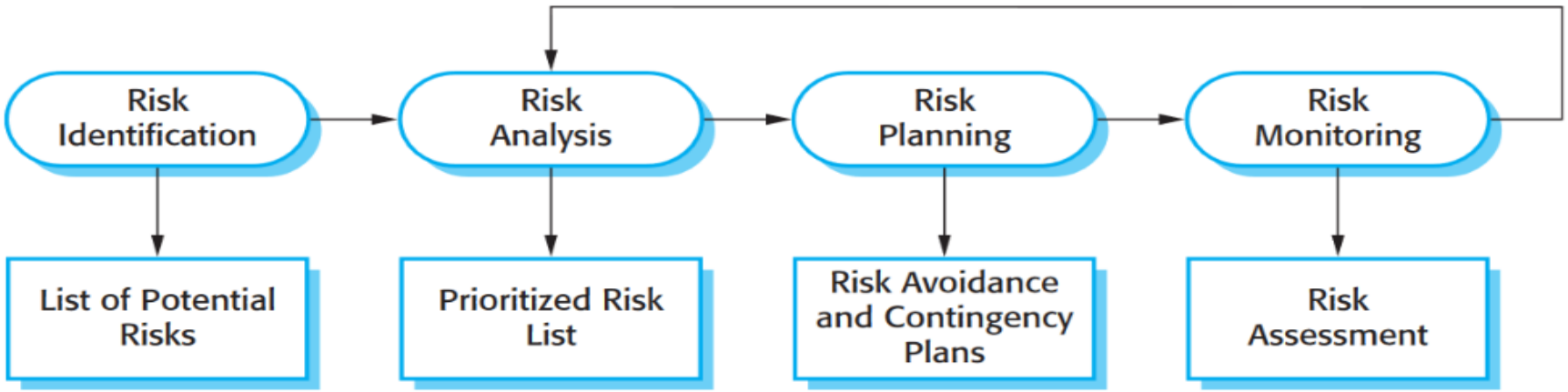
Project Management

Risk Management

- Risk management involves anticipating risks that might affect the project schedule or the quality of the software being developed, and then taking action to avoid these risks.
- Three related categories of risks:
  - **Project risks:** Risks that affect the project schedule or resources (e.g., loss of an experienced designer).
  - **Product risks:** Risks that affect the quality or performance of the software being developed (e.g., failure of a purchased component to perform as expected).
  - **Business risks:** Risks that affect the organization developing or procuring the software (e.g., a new product from competitors).
- Note: These risk types overlap.

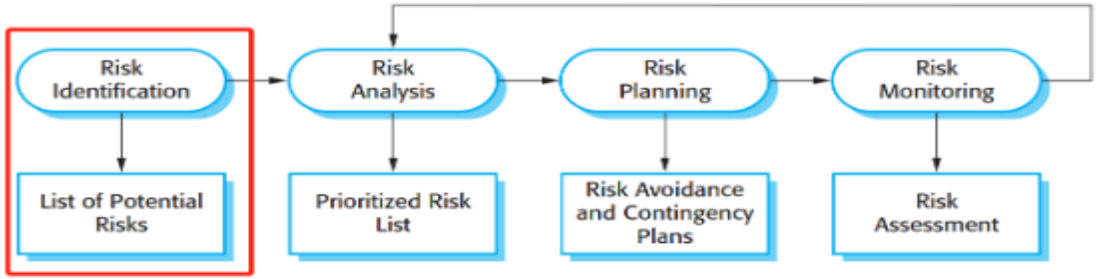
Risk Management Process





- **Risk identification:** identify possible project, product, and business risks
- **Risk analysis:** assess the likelihood and consequences of these risks
- **Risk planning:** plans to address the risk, either by avoiding it or minimizing its effects on the project
- **Risk monitoring:** regularly assess the risk and your plans for risk mitigation and revise these when you learn more about the risk

Risk Identification



- First stage of the risk management process, concerning with identifying major risks.
- Can be a team process
- Can solely rely on managers' experiences
- Risk Checklist:
  - Technology risk
  - People risk
  - Organizational risk
  - Tool risk
  - Requirement risk
  - Estimation risk

Important TTL Questions

- W9 - 1
- W9 - 2
- W9 - 3
- W11 - 1
- W12 - 2
- W13 - 1