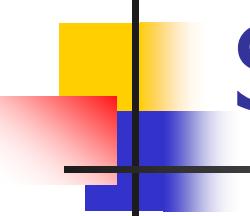


# *COMP 354: Introduction to Software Engineering*

## Software Development and Software Engineering

Based on Chapter 1 of the textbook



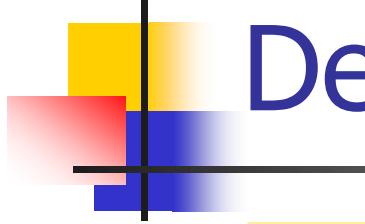
# Programming vs. Software Development

## **Programming:**

- Requirements are given/fixed.
- Select data structures and algorithms.
- Program to implement
- Test the program.

## **Software Development:**

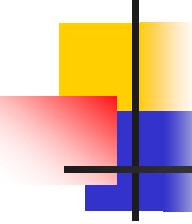
- Find/fix the requirements.
- Design/discuss the architecture and code.
- Implement the design.
- Test the software.
- Maintain the software.



# Nature of Software: Defining Software

Software is:

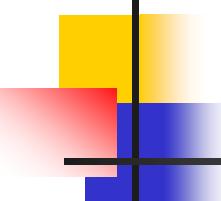
- 1) **Instructions (computer programs)** that when executed provide desired features, function, and performance;
- 2) **Data structures** that enable the programs to adequately manipulate information.
- 3) **Documentation** that describes the operation and use of the programs.



# Legacy Software

## Why must software change?

- Software must be adapted to meet the needs of new computing environments or technology.
- Software must be enhanced to implement new business requirements.
- Software must be extended to make it interoperable with other more modern systems or databases.
- Software must be re-architected to make it viable within a network environment.



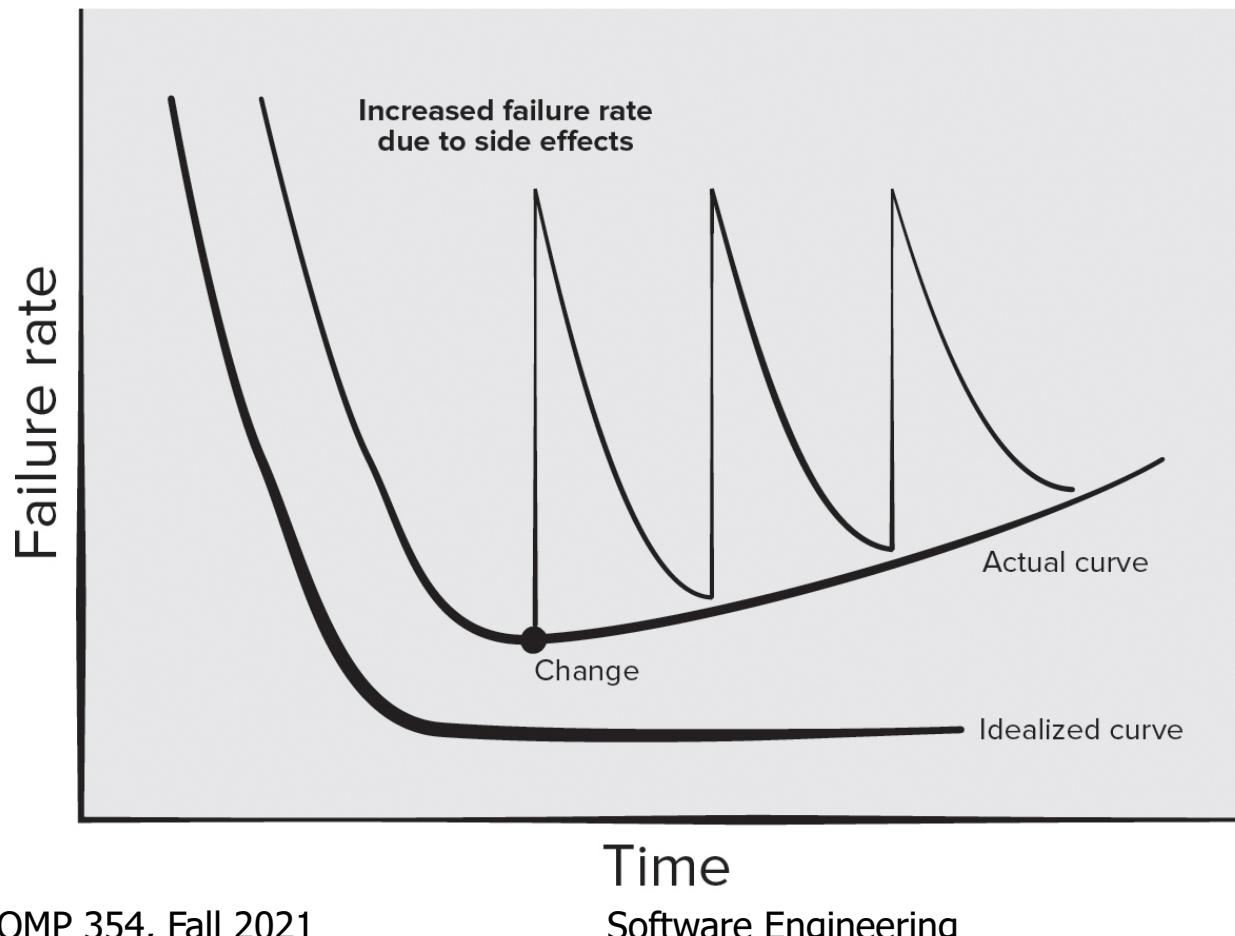
# How it all Starts

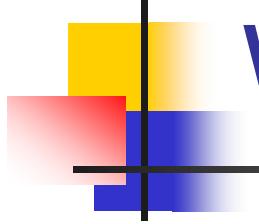
Every software project is precipitated by some business need

- The need to correct a defect in an existing application
- The need to adapt a 'legacy system' to a changing business environment
- The need to extend the functions and features of an existing application, or
- The need to create a new product, service, or system.

# Wear versus Deterioration

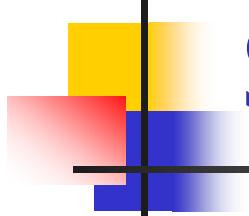
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





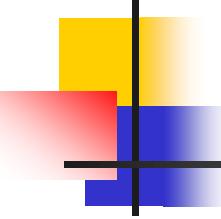
# What is Software?

- Software is developed or engineered it is not manufactured in the classical sense.
- Software doesn't "wear out" but is does deteriorate.
- Although the industry is moving toward component-based construction, most software continues to be custom-built.



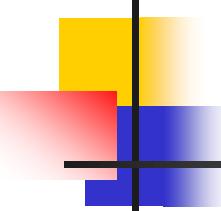
# Essence of Software Engineering Practice

1. Understand the problem (communication and analysis).
2. Plan a solution (modeling and software design).
3. Carry out the plan (code generation).
4. Examine result for accuracy (testing & quality assurance).



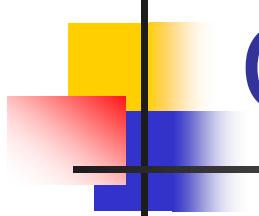
# Understand the Problem

- 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?



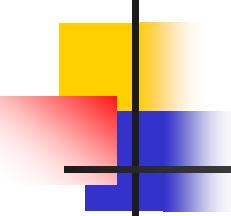
# Plan a Solution

- 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?



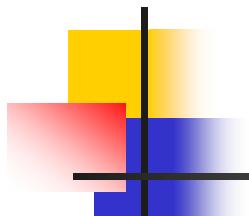
# Carryout the Plan

- Does the solution conform to the plan?
  - Is source code traceable to the design model?
- Is each component part of the solution provably correct?
  - 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?

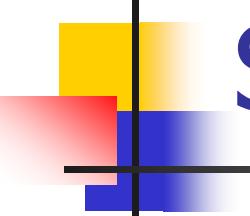


# Defining the Discipline

The IEEE definition:

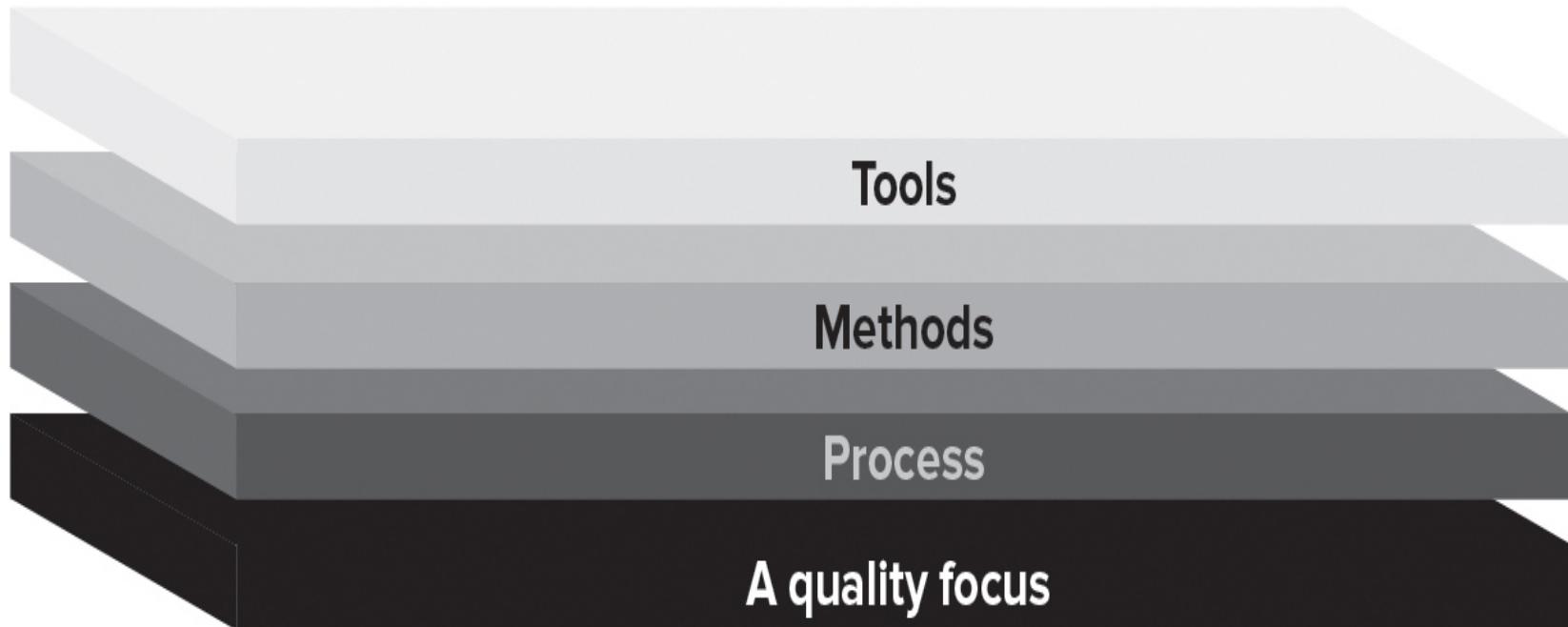
Software Engineering:

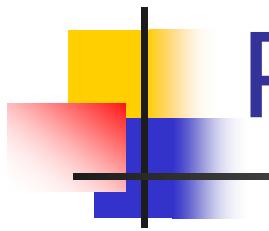
1. The application of a **systematic, disciplined, quantifiable approach** to the development, operation, and maintenance of software; that is, the application of engineering to software.
2. The study of approaches as in (1).



# Software Engineering Layers

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Process Framework Activities

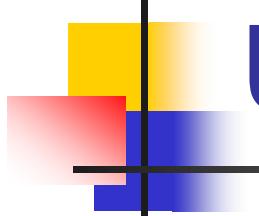
- Communication

- Planning
- Modeling
- Analysis of requirements
- Design

- Construction

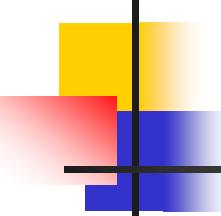
- Code generation
- Testing

- Deployment



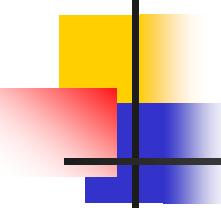
# Umbrella Activities

- Software project tracking and control.
- Risk management.
- Software quality assurance.
- Technical reviews.
- Measurement.
- Software configuration management.
- Reusability management.
- Work product preparation and production.



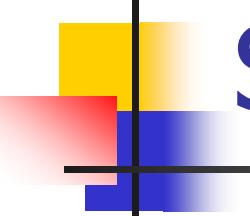
# Process Difference Requiring Adaptation

- Overall flow of activities, actions, and tasks and the interdependencies among them.
- Degree to which actions and tasks are defined within each framework activity.
- Degree to which work products are identified and required.
- Manner in which quality assurance activities are applied.
- Manner in which project tracking and control activities are applied.
- Overall degree of detail and rigor with which the process is described.
- Degree to which the customer and other stakeholders are involved with the project.
- Level of autonomy given to the software team.
- Degree to which team organization and roles are prescribed.



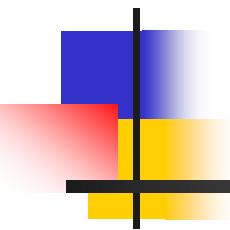
# General Principles

1. **The Reason It All Exists** – provide value to users.
2. **KISS (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.
7. **Think!** – placing thought before action produce results.



# Software Application Domains

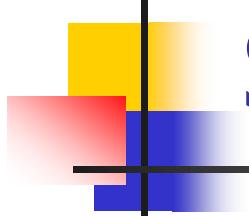
- System software.
- Application software.
- Engineering/Scientific software.
- Embedded software.
- Product-line software.
- Web/Mobile applications.
- AI software (robotics, neural nets, game playing).



# *COMP 354: Introduction to Software Engineering*

## Software Process Models

Based on Chapter 2 of the textbook



# Essence of Software Engineering Practice

1. Understand the problem (communication and analysis).
2. Plan a solution (modeling and software design).
3. Carry out the plan (code generation/construction).
4. Examine result for accuracy (testing & quality assurance/deployment).

# Generic Process Model

## ■ Framework Activities

- Communication
- Planning
- Modeling
- Construction
- Deployment

## ■ Umbrella Activities

- Project Tracking and Control
- Risk Management
- Quality Assurance
- Configuration Management
- Technical Reviews

## Software process

### Process framework

#### Umbrella activities

##### Framework activity #1

software engineering action #1.1

Task sets

work tasks  
work products  
quality assurance points  
project milestones

:

Software engineering action #1.k

Task sets

work tasks  
work products  
quality assurance points  
project milestones

:

##### Framework activity #n

software engineering action #n.1

Task sets

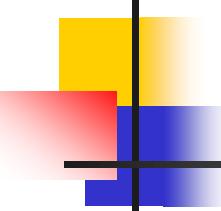
work tasks  
work products  
quality assurance points  
project milestones

:

Software engineering action #n.m

Task sets

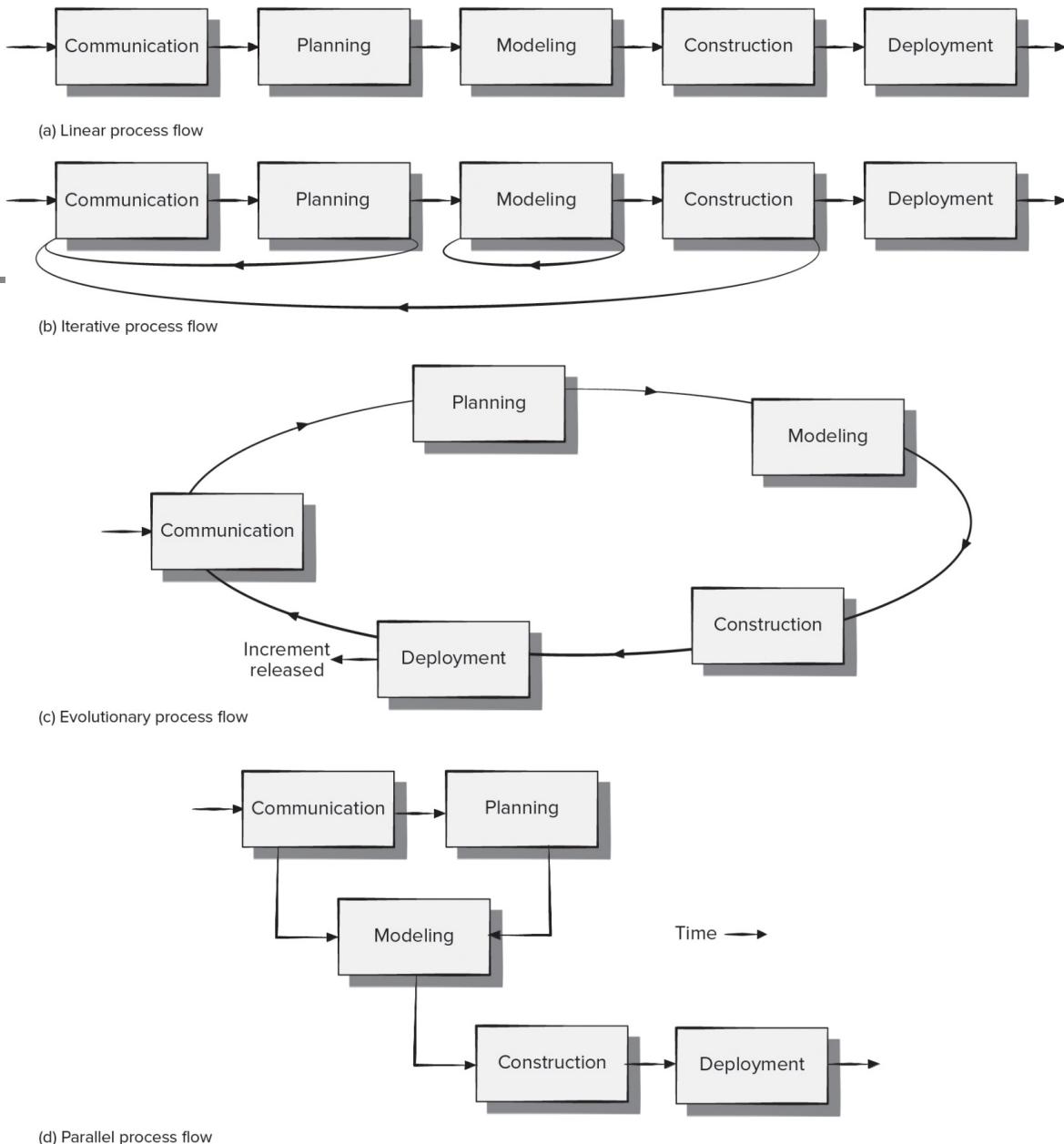
work tasks  
work products  
quality assurance points  
project milestones



# Identifying a Task Set

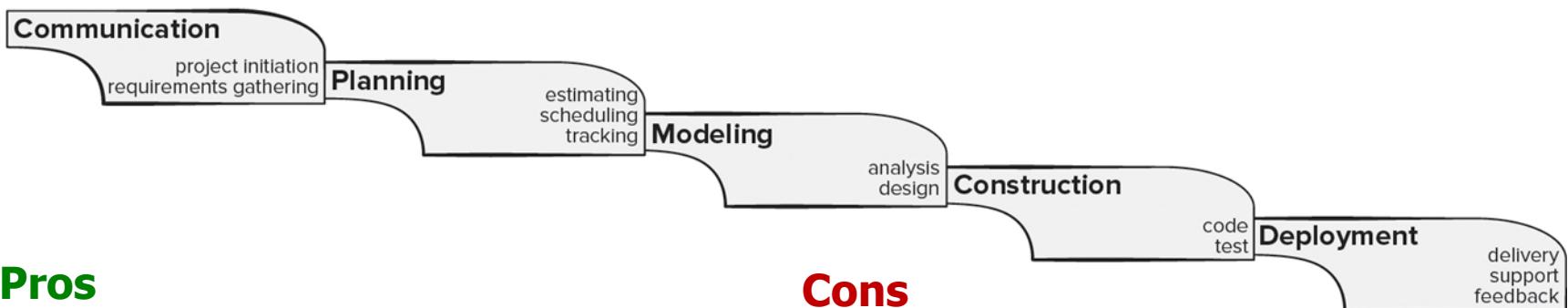
- A task set defines the actual work to be done to accomplish the objectives of a software engineering action.
- A task set is defined by creating several lists:
  - A list of the tasks to be accomplished.
  - A list of the work products to be produced.
  - A list of the quality assurance filters to be applied.

# Process Flow



# Waterfall Process Model

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



## Pros

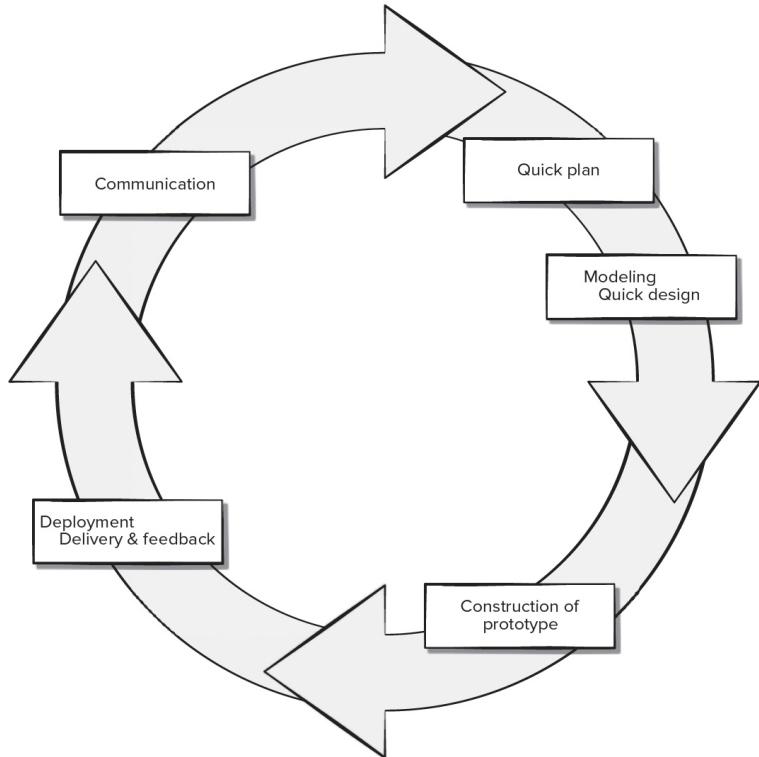
- It is easy to understand and plan.
- It works for well-understood small projects.
- Analysis and testing are straightforward.

## Cons

- It does not accommodate change well.
- Testing occurs late in the process.
- Customer approval is at the end.

# Prototyping Process Model

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



## Pros

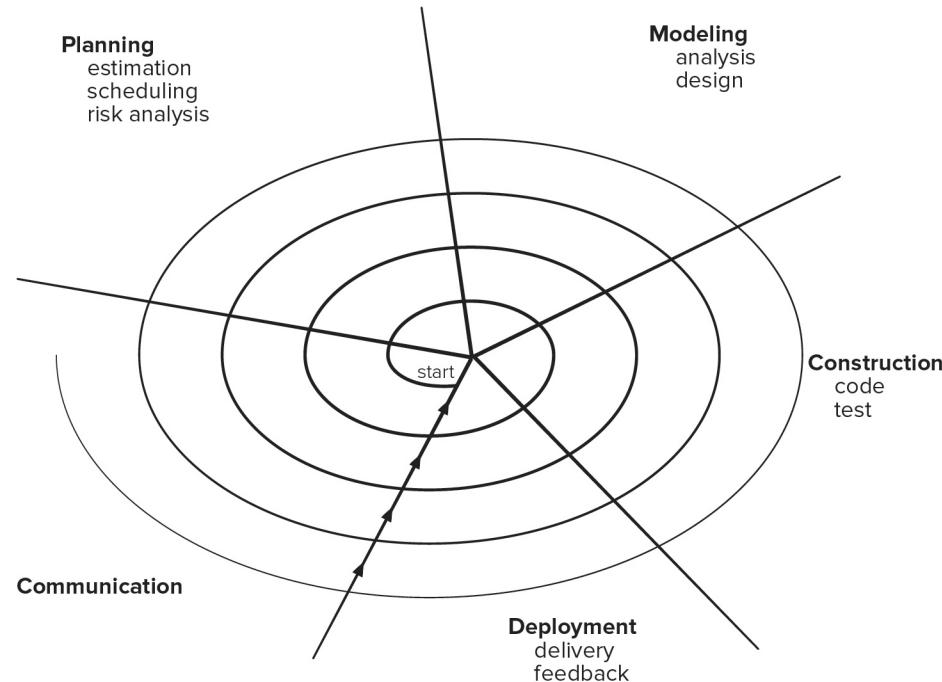
- Reduced impact of requirement changes.
- Customer is involved early and often.
- Works well for small projects.
- Reduced likelihood of product rejection.

## Cons

- Customer involvement may cause delays.
- Temptation to “ship” a prototype.
- Work lost in a throwaway prototype.
- Hard to plan and manage.

# Spiral Process Model

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



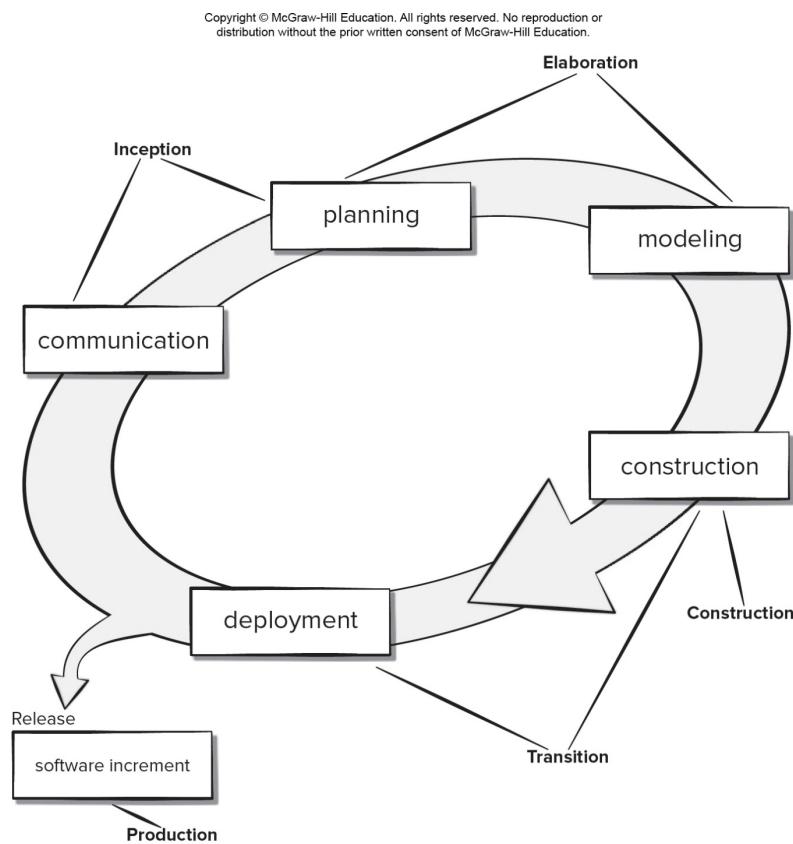
## Pros

- **Continuous customer involvement.**
- Development risks are managed.
- **Suitable for large, complex projects.**
- It works well for extensible products.

## Cons

- Risk analysis failures can doom the project.
- **Project may be hard to manage.**
- Requires an expert development team.

# Unified Process Model

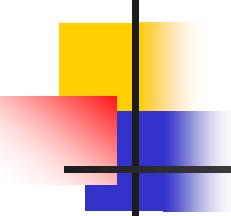


## Pros

- Quality documentation emphasized.
- Continuous customer involvement.
- Accommodates requirements changes.
- Works well for maintenance projects.

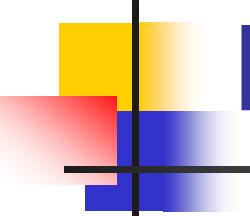
## Cons

- Use cases are not always precise.
- Tricky software increment integration.
- Overlapping phases can cause problems.
- Requires expert development team.



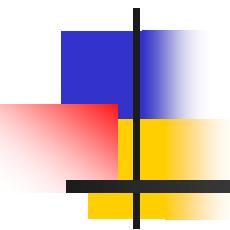
# Process Assessment and Improvement

- The existence of a software process is no guarantee that software will be delivered on time, or meet the customer's needs, or that it will exhibit long-term quality characteristics.
- Any software process can be assessed to ensure that it meets a set of basic process criteria that have been shown to be essential for successful software engineering.
- Software processes and activities should be assessed using numeric measures or software analytics (metrics).



# Prescriptive Process Models

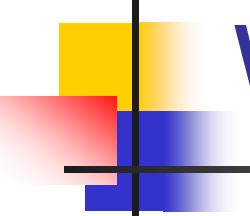
- Prescriptive process models advocate an orderly approach to software engineering.
- *That leads to two questions:*
  - If prescriptive process models strive for structure and order, are they appropriate for a software world that thrives on change?
  - If we reject traditional process models and replace them with something less structured, do we make it impossible to achieve coordination and coherence in software work?



# *COMP 354: Introduction to Software Engineering*

## Agility and Process

Based on Chapter 3 of the textbook

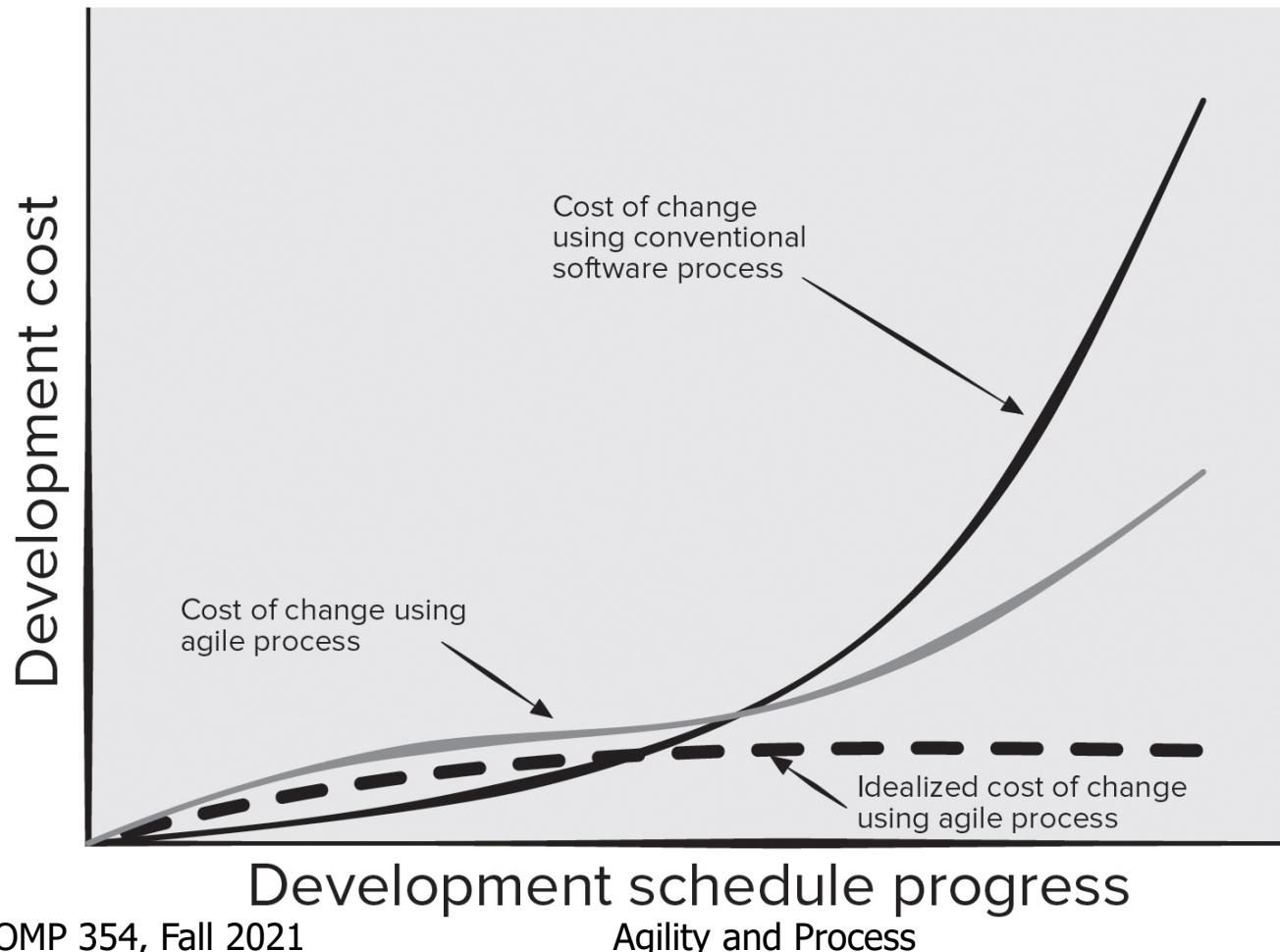


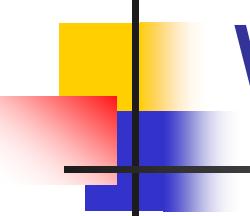
# What is Agility?

- 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.
- Rapid, incremental delivery of software.

# Agility and Cost of Change

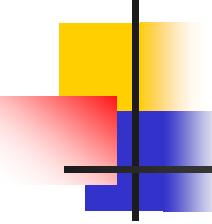
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





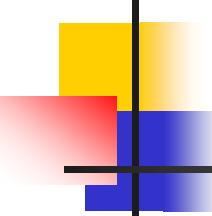
# What is an Agile Process?

- Driven by customer descriptions of what is required (scenarios).
- Customer feedback is frequent and acted on.
- Recognizes that plans are short-lived.
- Develops software iteratively with a heavy emphasis on construction activities.
- Delivers multiple 'software increments' as executable prototypes.
- Adapts as project or technical changes occur.



# Agility Principles

- Customer satisfaction is achieved by providing value through software that is delivered to the customer as rapidly as possible.
- Develop recognizing that requirements will change and welcome changes.
- Deliver software increments frequently (weeks not months) to stakeholders to ensure feedback on their deliveries is meaningful.
- Agile team populated by motivated individuals using face-to-face communication to convey information.
- Team process encourages technical excellence, good design, simplicity, and avoids unnecessary work.

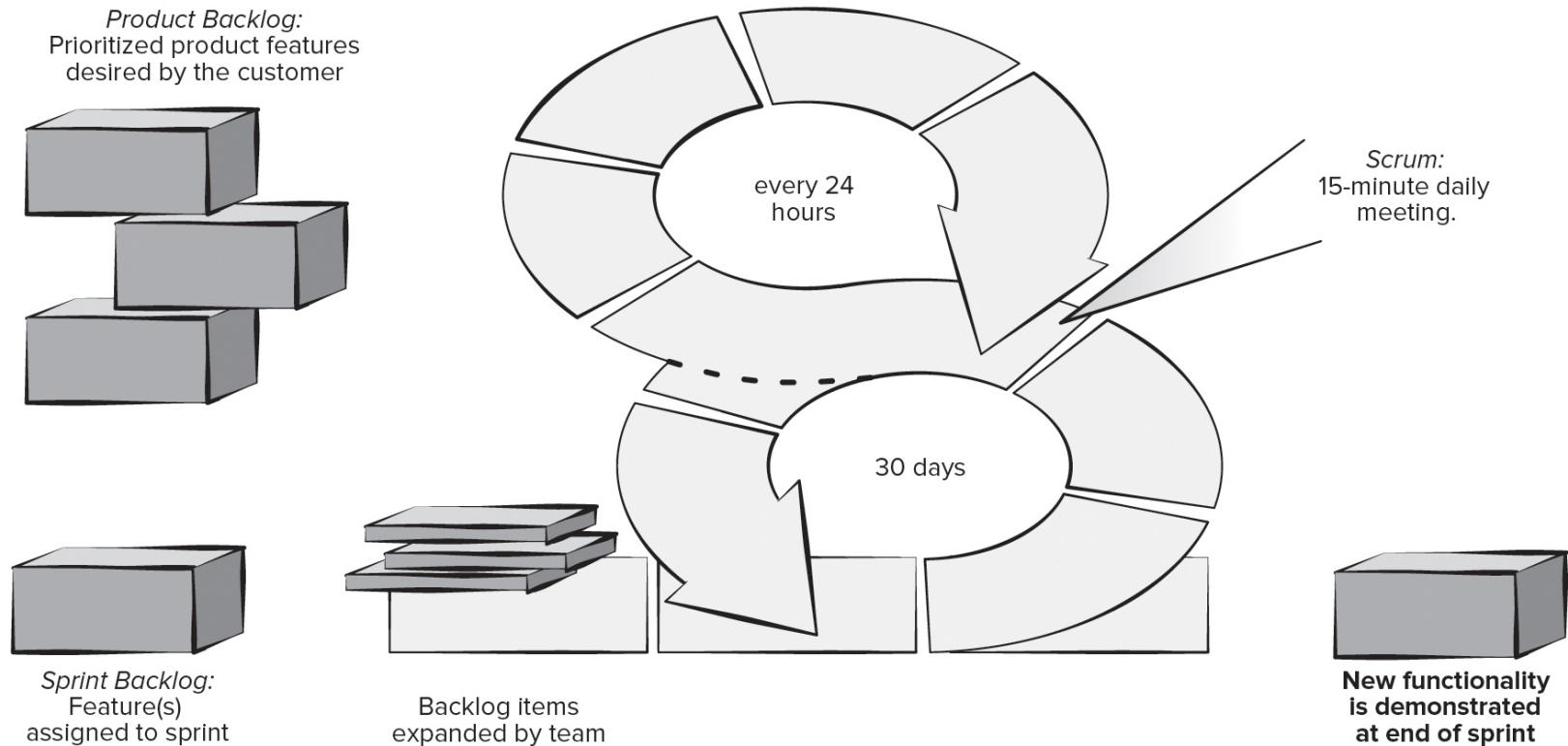


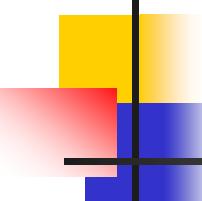
# Agility Principles

- Working software that meets customer needs is the primary goal.
- Pace and direction of the team's work must be "sustainable," enabling them to work effectively for long periods of time.
- An agile team is a "self-organizing team"— that can be trusted to develop well-structured architectures that lead to solid designs and customer satisfaction.
- Part of the team culture is to consider its work introspectively with the intent of improving how to become more effective its primary goal (customer satisfaction).

# Scrum Framework

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Scrum Details

- **Backlog Refinement Meeting**

Developers work with stakeholders to create product backlog.

- **Sprint Planning Meeting** Backlog partitioned into “sprints” derived from backlog and next sprint defined.

- **Daily Scrum Meeting** Team members synchronize their activities and plan work day (15 minutes max).

- **Sprint Review** Prototype “demos” are delivered to the stakeholders for approval or rejection.

- **Sprint Retrospective** After sprint is complete, team considers what went well and what needs improvement.

## Pros

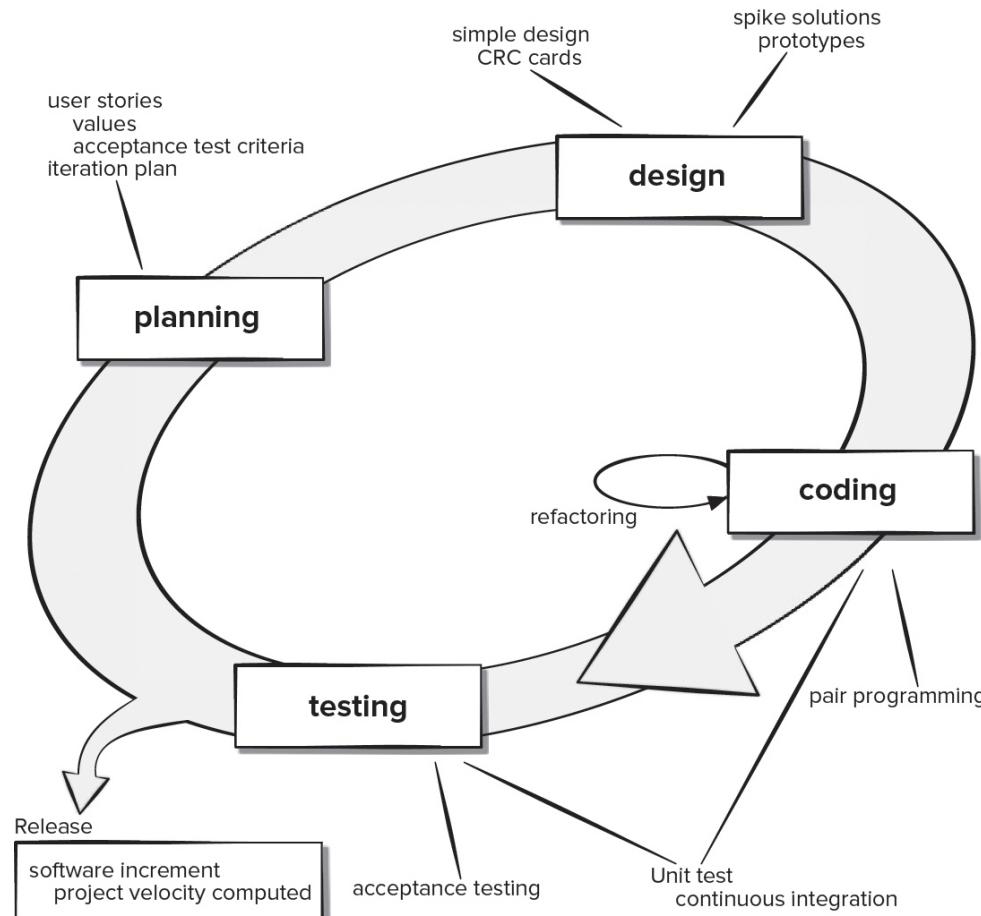
- Product owner sets priorities.
- Team owns decision making.
- Documentation is lightweight.
- Supports frequent updating.

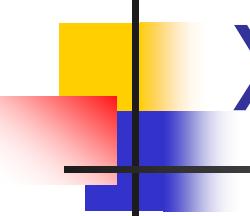
## Cons

- Difficult to control the cost of changes.
- May not be suitable for large teams.
- Requires expert team members.

# Extreme Programming (XP) Framework

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# XP Details

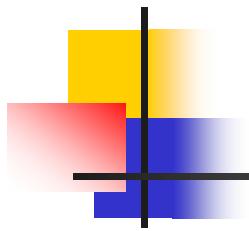
- **XP Planning** – Begins with user stories, team estimates cost, stories grouped into increments, commitment made on delivery date, computer project velocity.
- **XP Design** – Follows KIS principle, encourages use of CRC cards, design prototypes, and refactoring.
- **XP Coding** – construct unit tests before coding, uses pair.
- **XP Testing** – unit tests executed daily, acceptance tests define by customer.

## Pros

- Emphasizes customer involvement.
- Establishes rational plans and schedules.
- High developer commitment to the project.
- Reduced likelihood of product rejection.

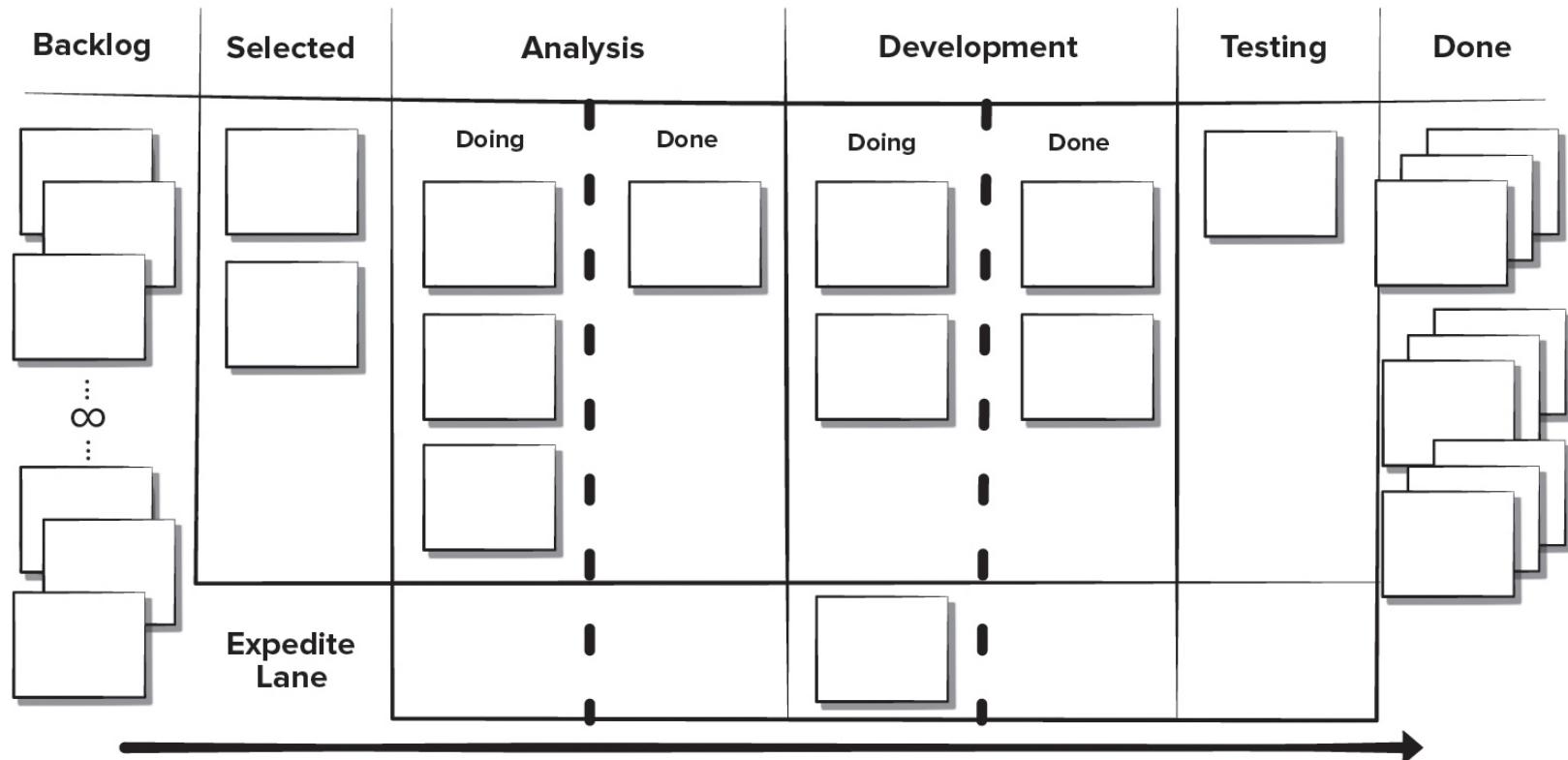
## Cons

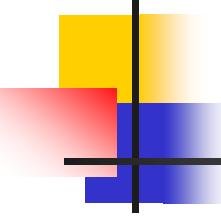
- Temptation to “ship” a prototype.
- Requires frequent meetings about increasing costs.
- Allows for excessive changes.
- Depends on highly skilled team members.



# Kanban Framework

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Kanban Details

- Visualizing workflow using a Kanban board.
- Limiting the amount of *work in progress* at any given time.
- Managing workflow to reduce waste by understanding the current value flow.
- Making process policies explicit and the criteria used to define “done”.
- Focusing on continuous improvement by creating feedback loops where changes are introduced.
- Make process changes collaboratively and involve all stakeholders as needed.

## Pros

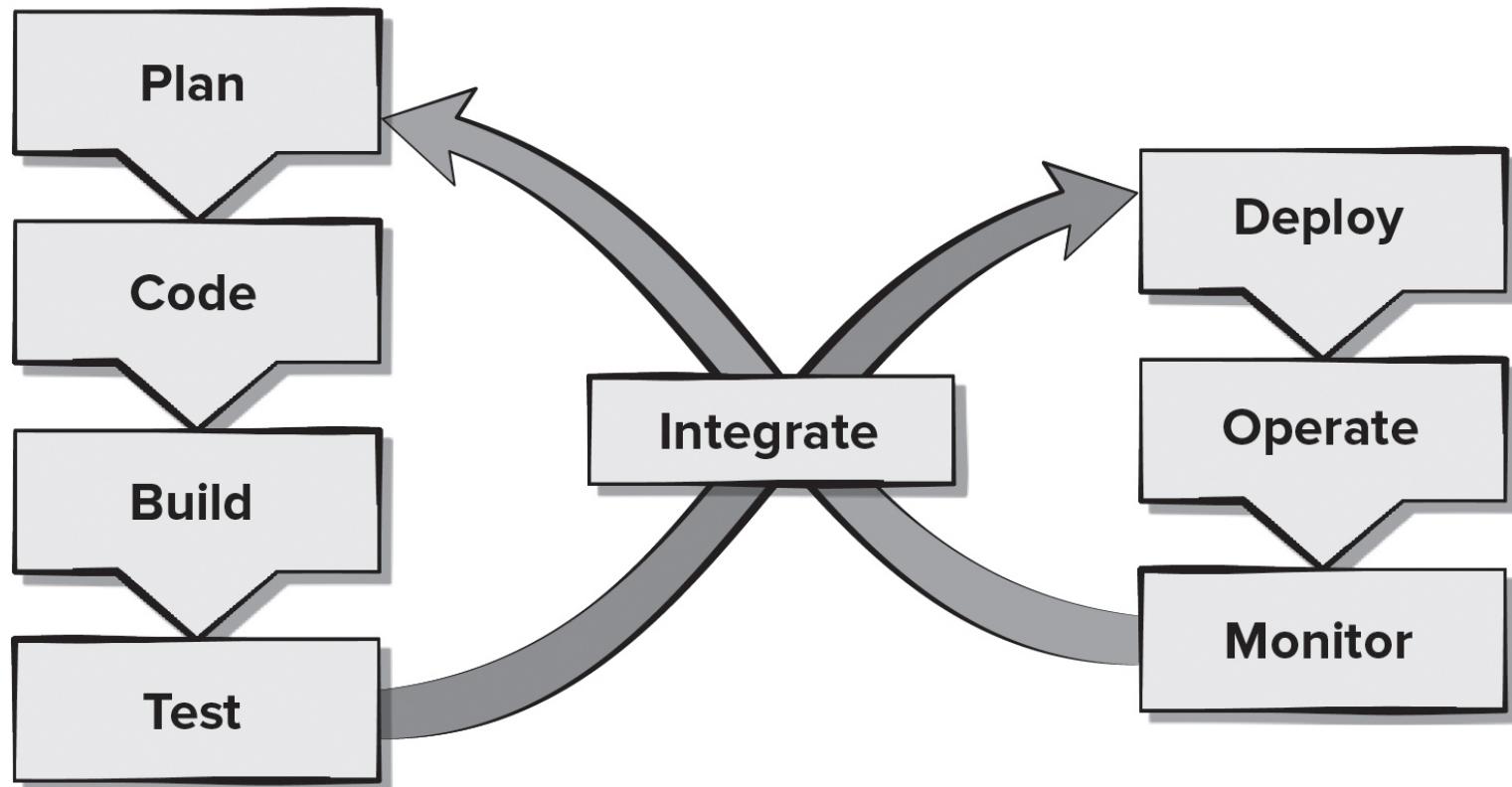
- Lower budget and time requirements.
- Allows early product delivery.
- Process policies written down.
- Continuous process improvement.

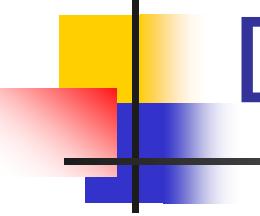
## Cons

- Team collaboration skills determine success.
- Poor business analysis can doom the project.
- Flexibility can cause developers to lose focus.
- Developer reluctance to use measurement.

# DevOps

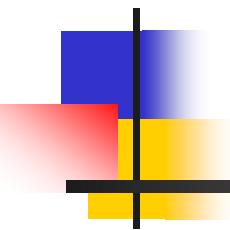
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# DevOps Details

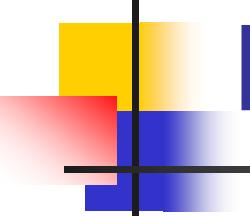
- **Continuous development.** Software delivered in multiple sprints.
  - **Continuous testing.** Automated testing tools used prior to integration.
  - **Continuous integration.** Code pieces with new functionality added to existing code running code.
  - **Continuous deployment.** Integrated code is deployed to the production environment.
  - **Continuous monitoring.** Team operations staff members proactively monitor software performance in the production environment.
- Reduced time to code deployment.
  - Team has developers and operations staff.
  - Team has end-to-end project ownership.
  - Proactive monitoring of deployed product.
- Cons**
- Pressure to work on both old and new code.
  - Heavy reliance on automated tools to be effective.
  - Deployment may affect the production environment.
  - Requires an expert development team.



# *COMP 354: Introduction to Software Engineering*

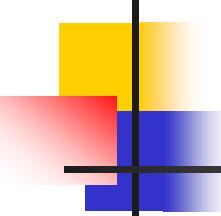
## Understanding Requirements

Based on Chapter 7 of the textbook



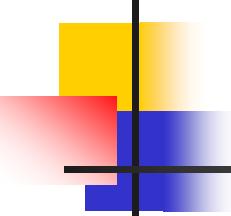
# Requirements Engineering

- **Inception** - establish a basic understanding of the problem, the people who want a solution, and the nature of the solution that is desired, important to establish effective customer and developer communication.
- **Elicitation** - elicit requirements and business goals form from all stakeholders.
- **Elaboration** - focuses on developing a refined requirements model that identifies aspects of software function, behavior, and information.



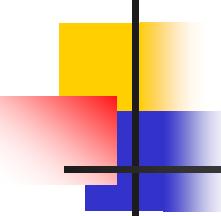
# Requirements Engineering

- **Negotiation**—agree on the scope of a deliverable system that is realistic for developers and customers.
- **Specification**—can be any or all of the following: written documents, graphical models, mathematical models, usage scenarios, prototypes.
- **Validation**—Requirements engineering work products produced during requirements engineering are assessed for quality and consistency.
- **Requirements management** – set of traceability activities to help the project team identify, control, and track requirements and their changes to requirements as the project proceeds.



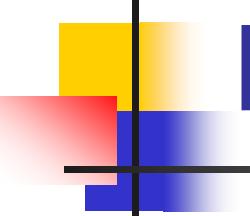
# Non-functional Requirements

- Non-Functional Requirement (NFR) – quality attribute, performance attribute, security attribute, or general system constraint.
- A two-phase process is used to determine which NFR's are compatible:
  - The first phase is to create a matrix using each NFR as a column heading and the system SE guidelines a row labels.
  - The second phase is for the team to prioritize each NFR using a set of decision rules to decide which to implement by classifying each NFR and guideline pair as complementary, overlapping, conflicting, or independent.



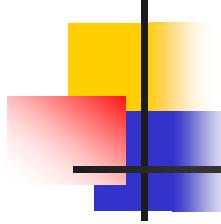
# Establishing the Groundwork

- Identify stakeholders.
  - “who else do you think I should talk to?”
- Recognize multiple points of view.
- Work toward collaboration.
- The first questions.
  - Who is behind the request for this work?
  - Who will use the solution?
  - What will be the economic benefit of a successful solution?
  - Is there another source for the solution that you need?



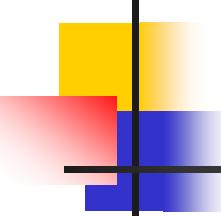
# Collaborative Requirements Gathering

- Meetings (real or virtual) are conducted and attended by both software engineers and other stakeholders.
- Rules for preparation and participation are established.
- Agenda is suggested that is formal enough to cover all important points but informal enough to encourage the free flow of ideas.
- A “facilitator” (customer, developer, or outsider) controls the meeting.
- A “definition mechanism” (worksheets, flip charts, wall stickers or virtual forum) is used.
- Goal is to identify the problem, propose solution elements, and negotiate different approaches.



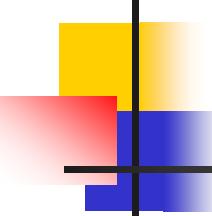
# Elicitation Work Products

- Statement of need and feasibility.
- Bounded statement of scope for the system or product.
- List of customers, users, and other stakeholders who participated in requirements elicitation,
- Description of the system's technical environment.
- List of requirements (preferably organized by function) and the domain constraints that apply to each.
- Set of usage scenarios (written in stakeholders' own words) that provide insight into the use of the system or product under different operating conditions.



# 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?

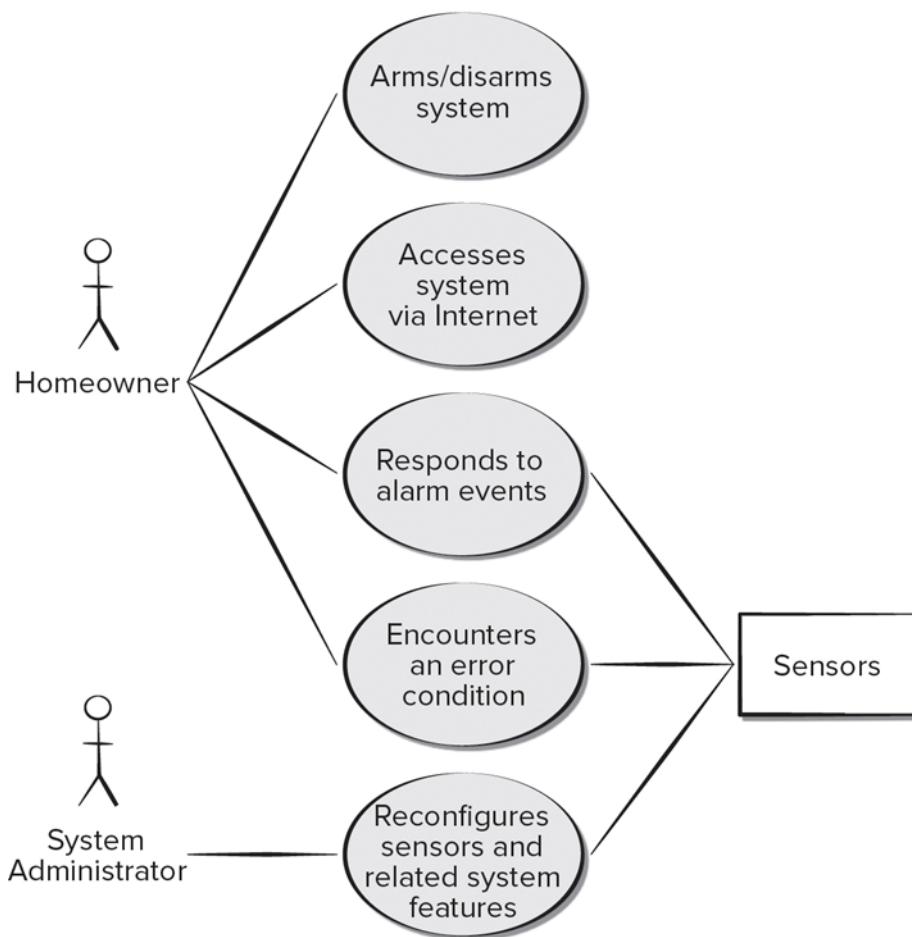


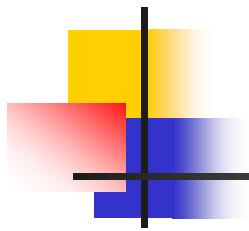
# Analysis Model Elements

- **Analysis model** provides a description of the required informational, functional, and behavioral domains for a computer-based system.
- **Scenario-based elements** – functional descriptions are express in the customers own words and user stories and as interactions of actors with the system expressed using UML use case diagrams.
- **Class-based elements** – collections of attributes and behaviors implied by the user stories and expressed using UML class diagrams (information domain).
- **Behavioral elements** – may be expressed using UML state diagrams as inputs causing state changes.

# UML Use Case Diagram

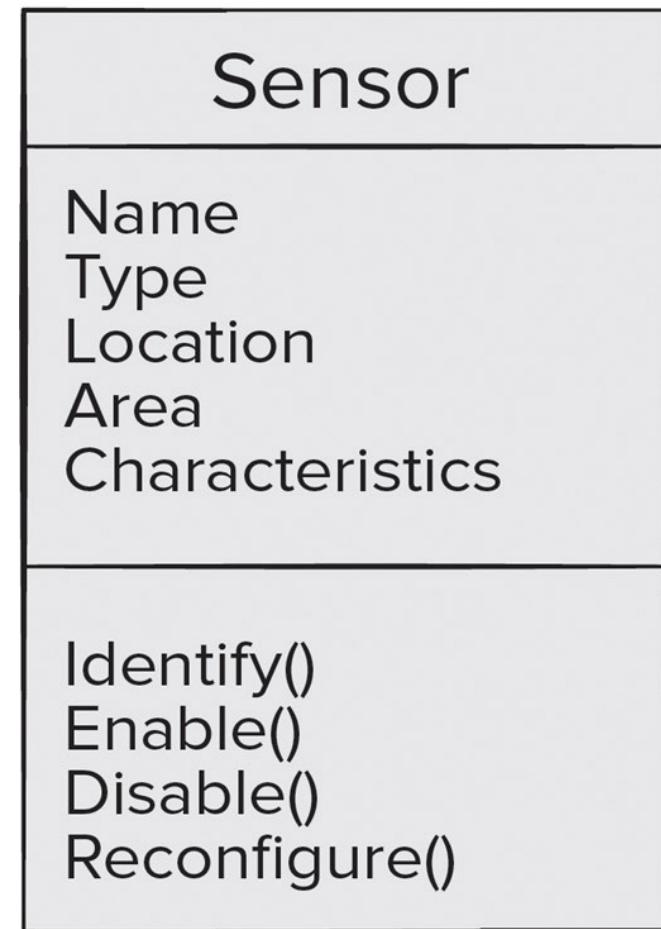
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

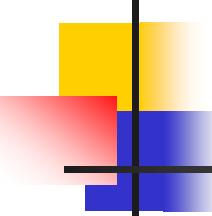




# UML Class Diagram

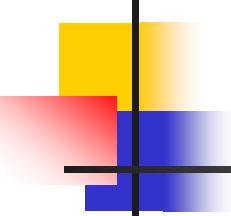
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





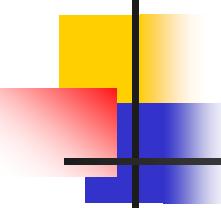
# Analysis Patterns

- Pattern name: A descriptor that captures the essence of the pattern.
- Intent: Describes what the pattern accomplishes or represents.
- Motivation: A scenario that illustrates how the pattern can be used to address the problem.
- Forces and context: A description of external issues (forces) that can affect how the pattern is used and the external issues that will be resolved when the pattern is applied.
- Solution: A description of how the pattern is applied to solve the problem with an emphasis on structural and behavioral issues.



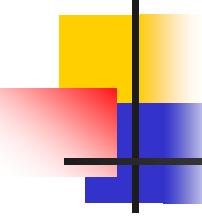
# Analysis Patterns

- Consequences: Addresses what happens when the pattern is applied and what trade-offs exist during its application.
- Design: Discusses how the analysis pattern can be achieved through the use of known design patterns.
- Known uses: Examples of uses within actual systems.
- Related patterns: One or more analysis patterns that are related to the named pattern because
  - (1) it is commonly used with the named pattern;
  - (2) it is structurally similar to the named pattern;
  - (3) it is a variation of the named pattern.



# Negotiating Requirements

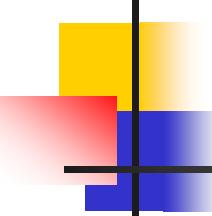
- Negotiations strive for a “win-win” result, stakeholders win by getting a product satisfying most of their needs and developers win by getting achievable deadlines.
- Handshaking is one-way to achieve “win-win”.
  - Developers propose solutions to requirements, describe their impact, and communicate their intentions to the customers.
  - Customer review the proposed solutions, focusing on missing features and seeking clarification of novel requirements.
  - Requirements are determined to be good enough if the customers accept the proposed solutions.
- Handshaking tends to improve identification, analysis, and selection of variants.



# Requirements Monitoring

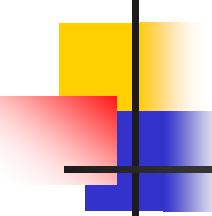
Useful for incremental development includes:

1. **Distributed debugging** - uncovers errors and determines their cause.
2. **Run-time verification** - determines whether software matches its specification.
3. **Run-time validation** - assesses whether the evolving software meets user goals.
4. **Business activity monitoring** - evaluates whether a system satisfies business goals.
5. **Evolution and codesign** - provides information to stakeholders as the system evolves.



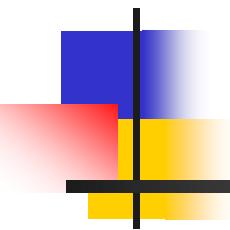
# Validating Requirements

- Is each requirement consistent with the overall objective for the system/product?
- Have all requirements been specified at the proper level of abstraction? That is, do some requirements provide a level of technical detail that is inappropriate at this stage?
- Is the requirement really necessary or does it represent an add-on feature that may not be essential to the objective of the system?
- Is each requirement bounded and unambiguous?
- Does each requirement have attribution? That is, is a source (generally, a specific individual) noted for each requirement?
- **Do any requirements conflict with other requirements?**



# Validating Requirements

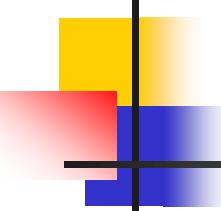
- Is each requirement achievable in the technical environment that will house the system or product?
- Is each requirement testable, once implemented?
- Does the requirements model properly reflect the information, function and behavior of system to be built?
- Has the requirements model been “partitioned” in a way that exposes progressively more detailed information about the system?
- Have requirements patterns been used to simplify the requirements model. Have all patterns been properly validated? Are all patterns consistent with customer requirements?



# *COMP 354: Introduction to Software Engineering*

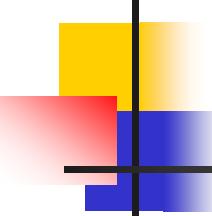
## Requirements Modeling

Based on Chapter 8 of the textbook



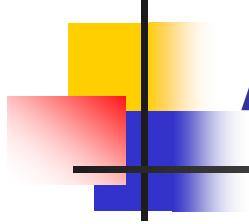
# Requirements Analysis

- Requirements analysis
  - specifies software's operational characteristics.
  - indicates software's interface with other system elements.
  - establishes constraints that software must meet.
- Requirements analysis allows the software engineer to:
  - elaborate on basic requirements established during earlier requirement engineering tasks.
  - build models that depict the user's needs from several different perspectives.



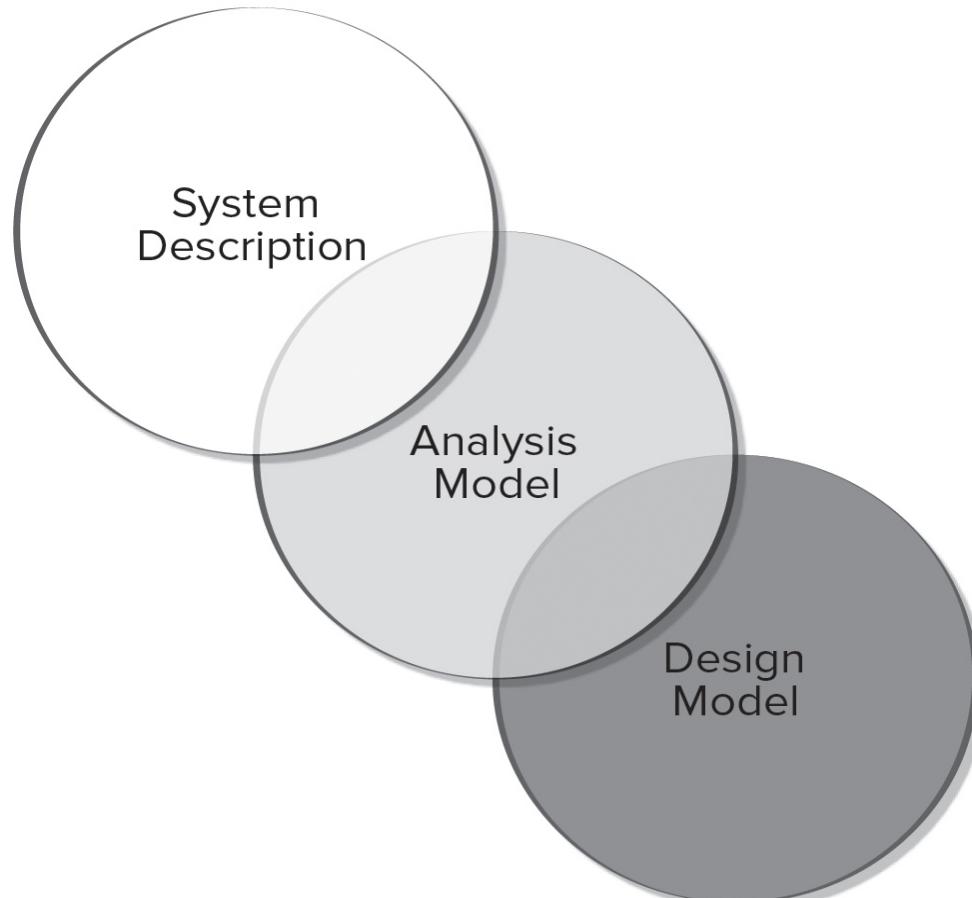
# Requirements Models

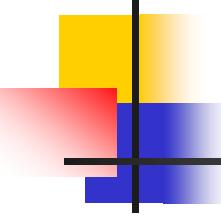
- **Scenario-based models** depict requirements from the point of view of various system “actors.”
- **Class-oriented models** represent object-oriented classes (attributes and operations) and how classes collaborate to achieve system requirements.
- **Behavioral models** depict how the software reacts to internal or external “events.”
- **Data models** depict the information domain for the problem.
- **Flow-oriented models** represent functional elements of the system and how they transform data in the system.



# A Bridge

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



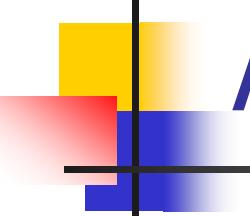


# Rules of Thumb

- The level of abstraction should be relatively high - focus on requirements visible in problem or business domains.
- Analysis model should provide insight into information domain, function, and behavior of the software.
- Delay consideration of infrastructure and other non-functional models until later in the modeling activity.
- The analysis model provides value to all stakeholders keep the model as simple as it can be.

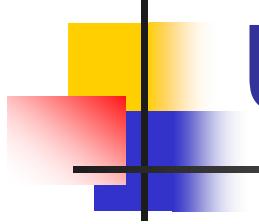
# Requirements Modeling Principles

- **Principle 1.** The information domain of a problem must be represented and understood.
- **Principle 2.** The functions that the software performs must be defined.
- **Principle 3.** The behavior of the software (as a consequence of external events) must be represented.
- **Principle 4.** The models that depict information, function, and behavior must be partitioned in a manner that uncovers detail in a layered (or hierarchical) fashion.
- **Principle 5.** The analysis task should move from essential information toward implementation detail.



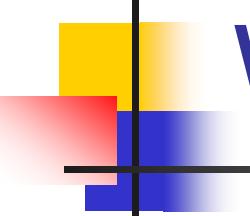
# Scenario-Based Modeling: Actors and Profiles

- A UML **actor** models an entity that interacts with a system object.
  - Actors may represent roles played by human stakeholders or external hardware as they interact with system objects by exchanging information.
- A UML **profile** provides a way of extending an existing model to other domains or platforms.
  - This might allow you to revise the model of a Web-based system and model the system for various mobile platforms.
  - Profiles might also be used to model the system from the viewpoints of different users.



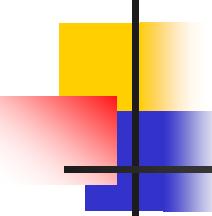
# Use Cases

- Use case as a “contract for behavior” and more formal than a user story.
- Use-cases are simply an aid to defining what exists outside the system (actors) and what should be performed by the system (use-cases).
  1. What should we write about?
  2. How much should we write about it?
  3. How detailed should we make our description?
  4. How should we organize the description?



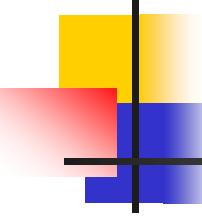
# What to Write About?

- The first two requirements engineering tasks—**inception** and **elicitation**—provide you with the information you'll need to begin writing use cases.
- To begin developing a set of use cases, list the functions or activities performed by a specific actor.
- You can obtain these from a list of required system functions, through conversations with stakeholders, or by an evaluation of activity diagrams developed as part of requirements modeling.
- Use cases of this type are sometimes referred to as **primary scenarios**.



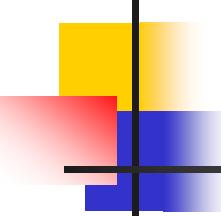
# Alternative Interactions

- Description of alternative interactions is essential to completely understand a function described a use case.
  - Can the actor take some other action at this point?
  - Is it possible that the actor will encounter some error condition at this point?
  - Is it possible that the actor will encounter some other behavior at this point (for example: behavior that is invoked by some event outside the actor's control)?
- Answers to these questions result in the creation of a set of **secondary scenarios** represent alternative use cased behavior.



# Defining Exceptions

- An **exception** describes a situation (either a failure condition or an alternative chosen by the actor) that causes the system to exhibit somewhat different behavior.
- Questions to ask:
  - Are there cases in which some “validation function” occurs during this use case?
  - Are there cases in which a supporting function (or actor) will fail to respond appropriately?
  - Can poor system performance result in unexpected or improper user actions?

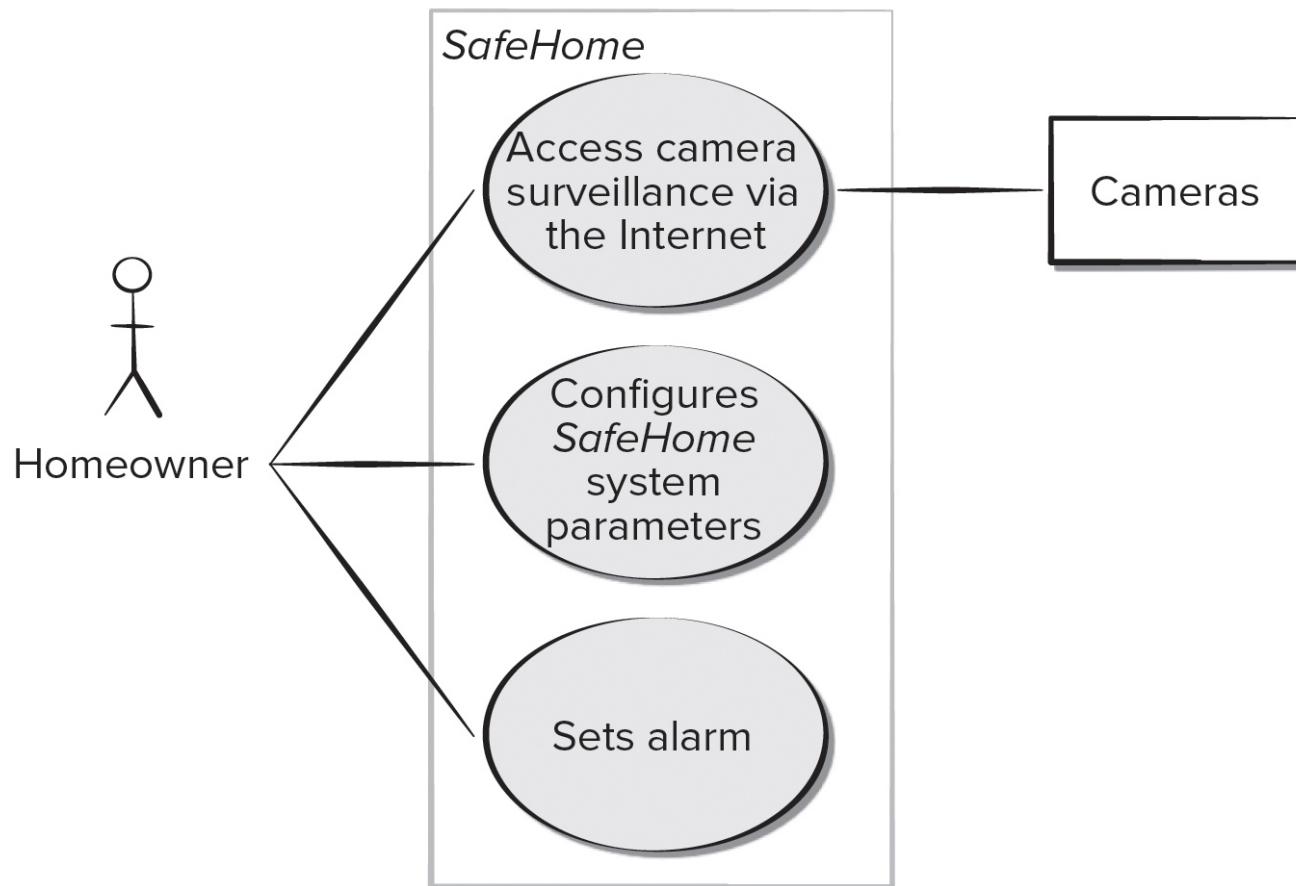


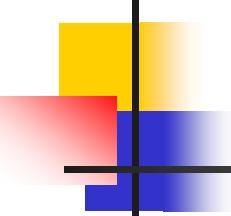
# Documenting Use Cases

- What are the main tasks or functions that are performed by the actor?
- 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?
- What are the preconditions, triggers, exceptions, and open issues?

# Use Case Diagram

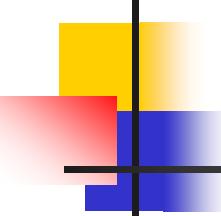
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





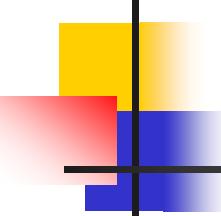
# Class-Based Modeling

- Class-based modeling represents:
  - objects that the system will manipulate.
  - operations (also called methods or services) that will be applied to the objects to effect the manipulation.
  - relationships (some hierarchical) between the objects.
  - collaborations that occur between the classes that are defined.
- The elements of a class-based model include classes and objects, attributes, operations, CRC models, UML class diagrams.



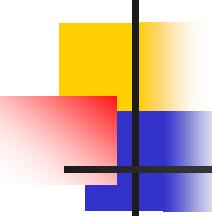
# Identifying Analysis Classes

- Examining the usage scenarios developed as part of the requirements model and perform a "grammatical parse".
  - Classes are determined by underlining each noun or noun phrase and entering it into a simple table.
  - Synonyms should be noted.
  - If the class (noun) is required to implement a solution, then it is part of the solution space; otherwise, if a class is necessary only to describe a solution, it is part of the problem space.
- But what should we look for once all of the nouns have been isolated?



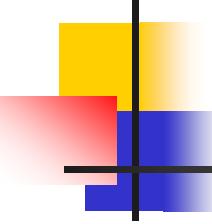
# Potential Analysis Classes

- **External entities** (for example: other systems, devices, people) that produce or consume information.
- **Things** (for example: reports, displays, letters, signals) that are part of the information domain for the problem.
- **Occurrences or events** that occur within the context of system operation.
- **Roles** played by people who interact with the system.
- **Organizational units** that are relevant to an application.
- **Places** that establish the context of the problem and overall function.
- **Structures** (for example: sensors, four-wheeled vehicles, or computers) that define a class of objects or related classes of objects.



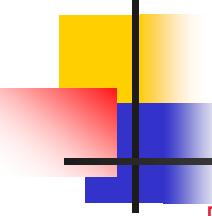
# Analysis Class Selection

- **Retained information.** Potential class will be useful during analysis only if information about it must be remembered.
- **Needed services.** Potential class must have a set of identifiable operations that can change the value of its attributes in some way.
- **Multiple attributes.** Focus should be on "major" information; a class with a single attribute may be better represented as an attribute of another class.
- **Common attributes.** A set of attributes can be defined for the potential class and the attributes apply to all instances of the class.
- **Common operations.** A set of operations can be defined for the potential class and the operations apply to all instances of the class.
- **Essential requirements.** External entities that appear in the problem space and produce or consume information essential to the solution will usually be defined as analysis classes in the model.



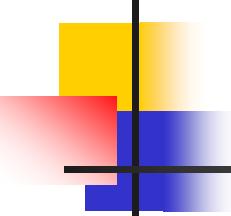
# Defining Attributes

- **Attributes** describe a class that has been selected for inclusion in the analysis model.
- It is the attributes that define the class—that clarify what is meant by the class in the context of the problem space.
- To develop a meaningful set of attributes for an analysis class, you should study each use case and select those “things” that reasonably “belong” to the class.
- What data items(composite and/or elementary) fully define this class in the context of the problem at hand?



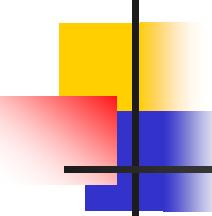
# Defining Operations

- Operations define the behavior of an object.
  1. Operations they can generally be divided into four broad categories:
  2. Operations that manipulate data in some way.
  3. Operations that perform a computation.
  4. Operations that inquire about the state.
  5. Operations that monitor an object for the occurrence of a controlling event.
- These functions are accomplished by operating on attributes and/or associations.
- Therefore, an operation must have “knowledge” of the class attributes and associations.



# CRC Modeling

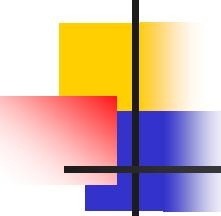
- **Class-Responsibility-Collaborator (CRC) modeling** provides a simple means for identifying and organizing the classes that are relevant to system or product requirements.
- A CRC model is really a collection of standard index cards that represent classes.
- The cards are divided into three sections:
  1. Along the top of the card you write the name of the class.
  2. list the class responsibilities on the left.
  3. list the collaborators on the right.



# CRC Cards

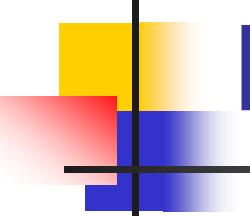
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

C	Class: FloorPlan	
D	Description	
R	<b>Responsibility:</b>	
R	Defines floor plan name/type	
	Manages floor plan positioning	
	Scales floor plan for display	
	Incorporates walls, doors, and windows	
	Shows position of video cameras	
	<b>Collaborator:</b>	
	Wall	
	Camera	



# CRC Model Review Process

- All stakeholders in the review (of the CRC model) are given a subset of the CRC model index cards. No reviewer should have two cards that collaborate.
- The review leader reads the use case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card.
- When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one of the responsibilities satisfies the use case requirement.
- If an error is found, modifications are made to the cards. This may include the definition of new classes (CRC index cards) or revising lists of responsibilities or collaborations on existing cards.

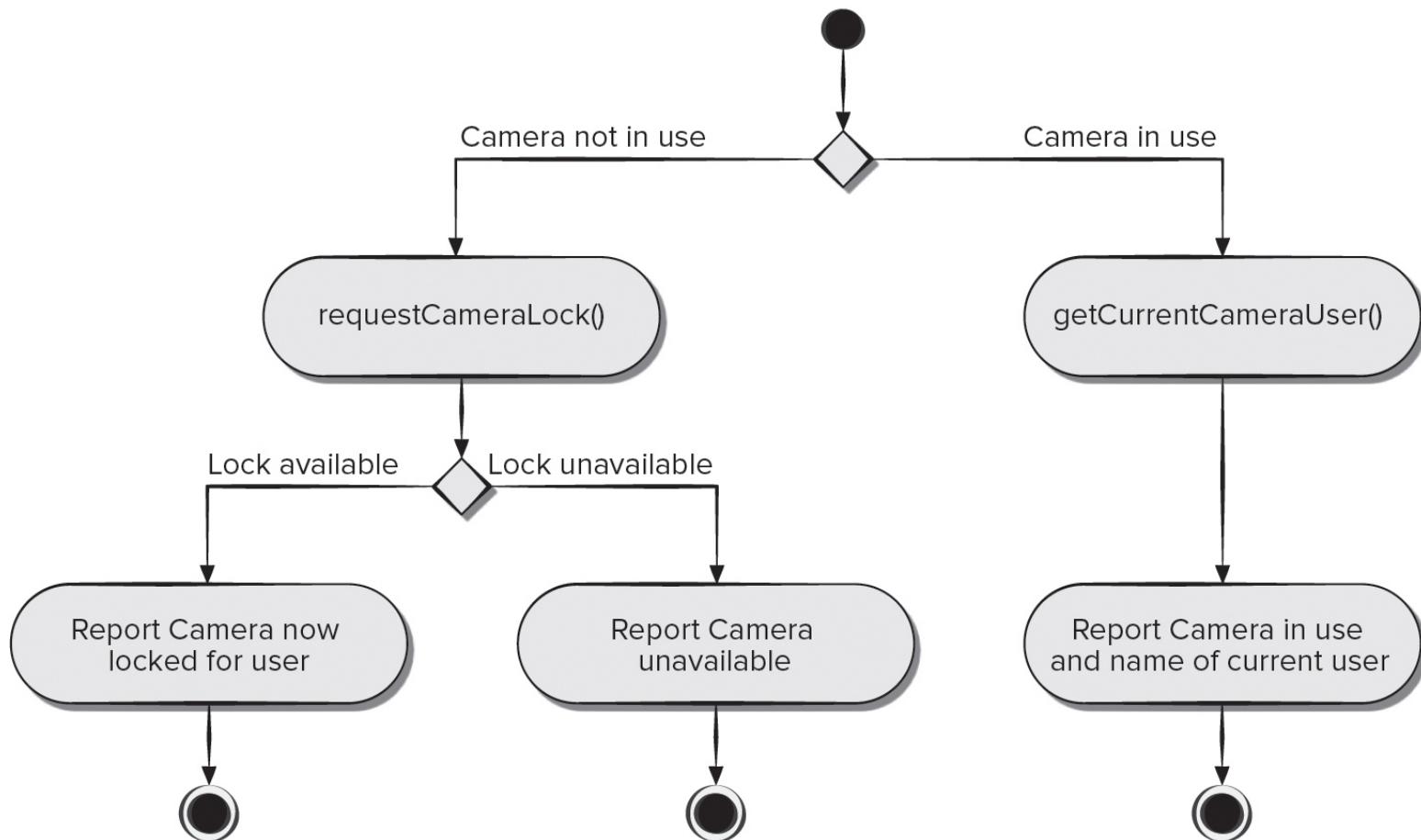


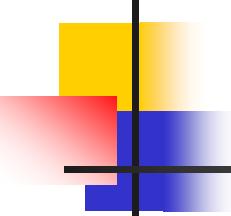
# Functional Modeling

- The **functional model** addresses two application processing elements:
  - user-observable functionality that is delivered by the app to end users.
  - operations contained within analysis classes that implement behaviors associated with the class.
- UML activity diagrams can be used to represent processing details.
- UML activity diagram supplements a use case by providing a graphical representation of the flow of interaction within a specific scenario.

# Activity Diagram

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



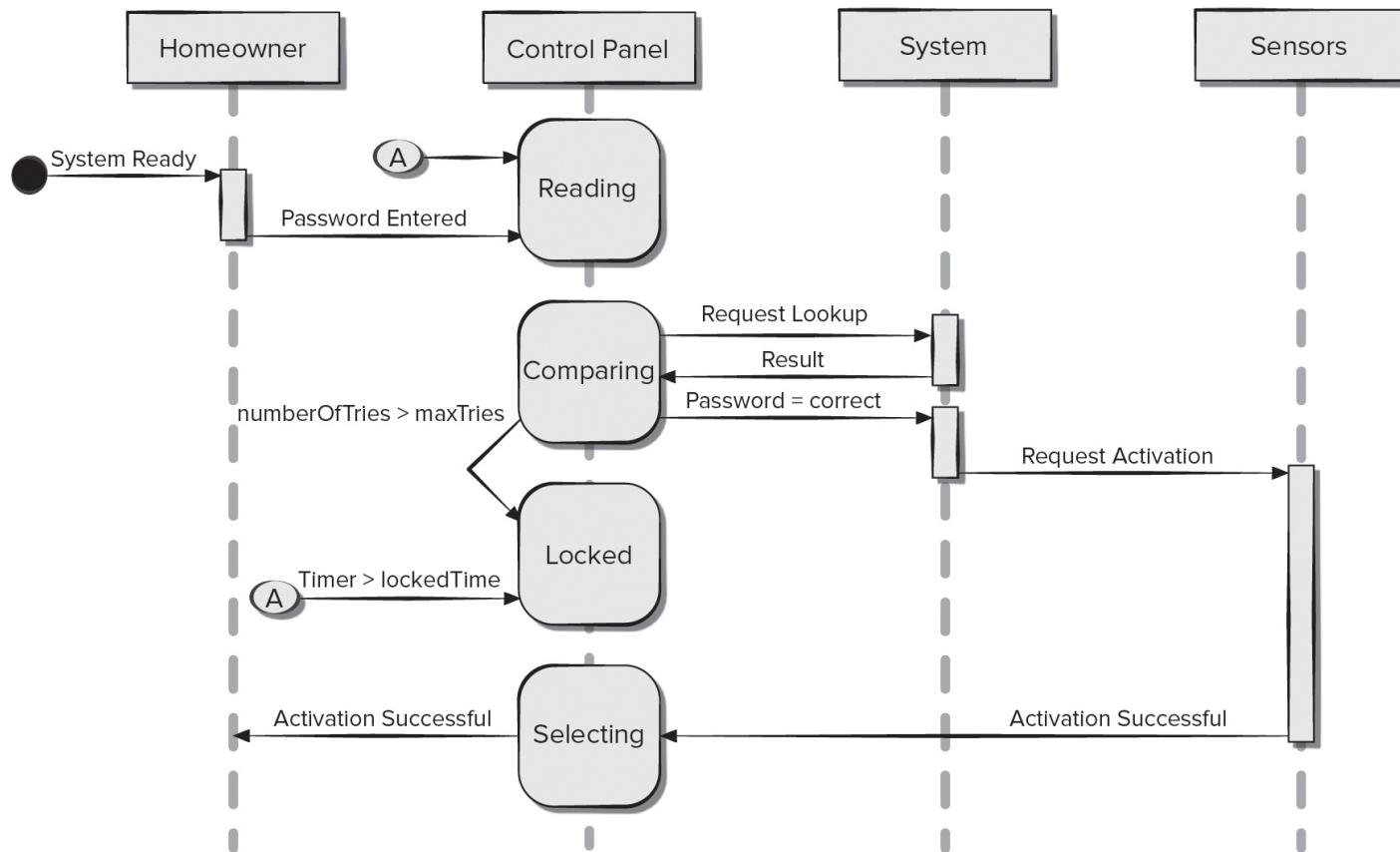


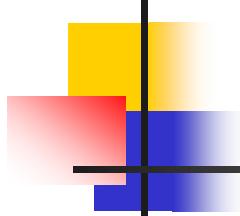
# Sequence Diagram

- The UML **sequence diagram** can be used for behavioral modeling.
- Sequence diagrams can also be used to show how events cause transitions from object to object.
- Once events have been identified by examining a use case, the modeler creates a sequence diagram—a representation of how events cause flow from one object to another as a function of time.
- Sequence diagram is a shorthand version of a use case.

# Sequence Diagram

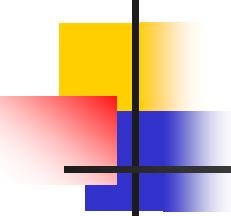
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





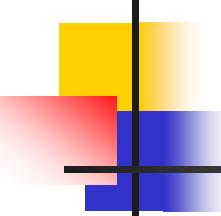
# Behavioral Modeling

- A **behavioral model** indicates how software will respond to internal or external events or stimuli.
- This information is useful in the creation of an effective design for the system to be built.
- UML activity diagrams can be used to model how system elements respond to internal events.
- UML state diagrams can be used to model how system elements respond to external events.



# Creating Behavioral Models

1. Evaluate all use cases to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create a sequence diagram for each use case.
4. Build a state diagram for the system.
5. Review the behavioral model for accuracy and consistency.

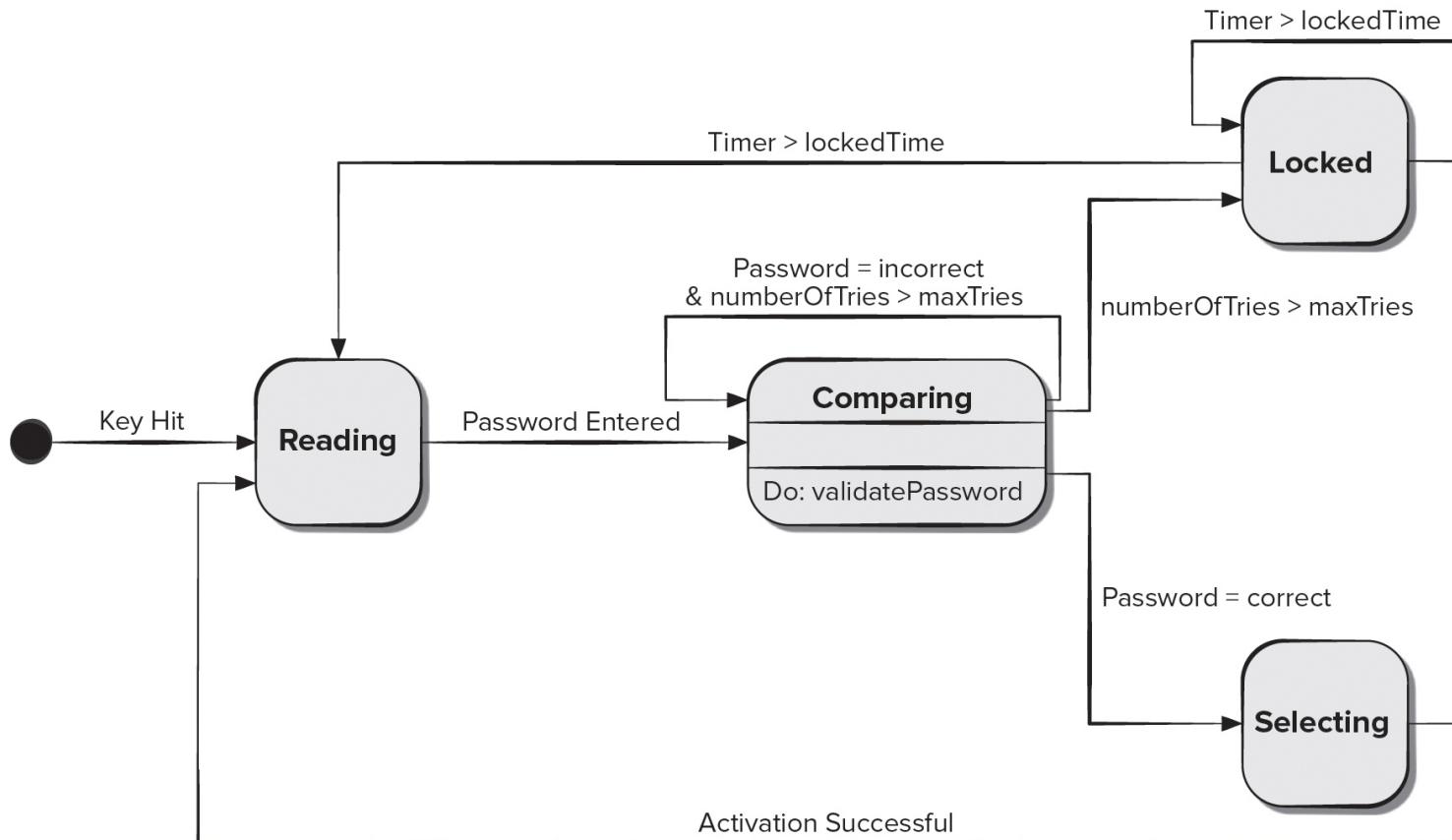


# Identifying Events

- A use case represents a sequence of activities that involves actors and the system.
- An event occurs whenever the system and an actor exchange information.
- An event is not the information that has been exchanged, but rather the fact that information has been exchanged.
- A use case needs to be examined for points of information exchange.
- Events are used to trigger state transitions.

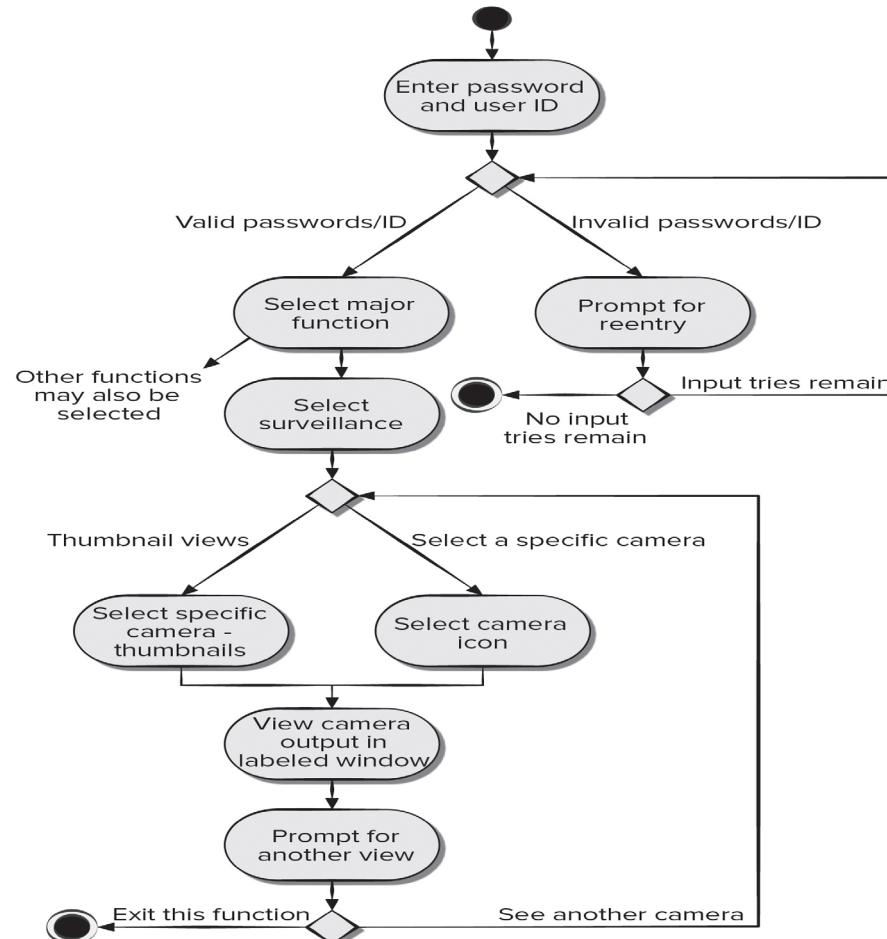
# State Diagram

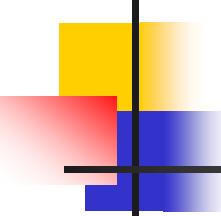
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# Activity Diagram

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



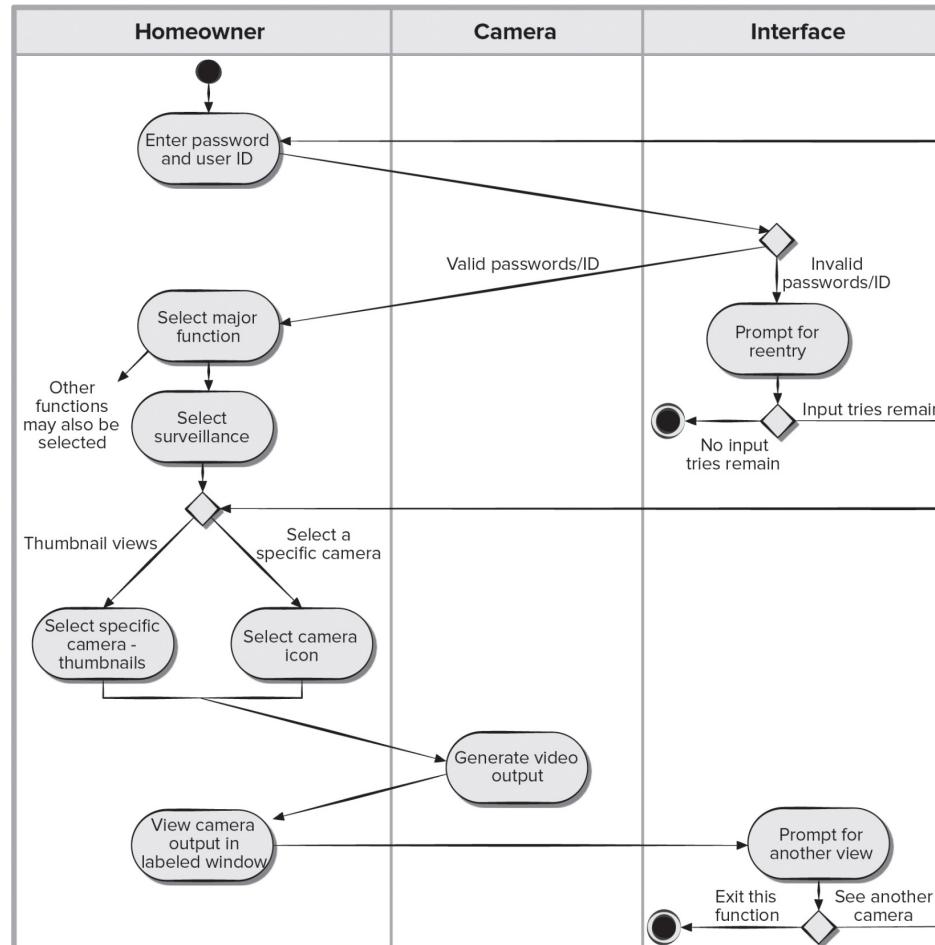


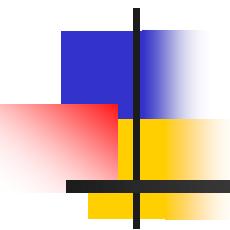
# Swimlane Diagrams

- The UML swimlane diagram is a useful variation of the activity diagram that allows you to represent the flow of activities described by the use case.
- Swimlane diagrams indicate which actor (if there are multiple actors involved in a specific use case) or analysis class has responsibility for the action described by an activity rectangle.
- Responsibilities are represented as parallel segments that divide the diagram vertically, like the lanes in a swimming pool.

# Swimlane Diagram

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

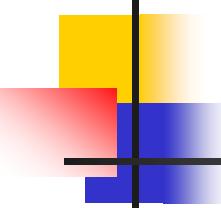




# *COMP 354: Introduction to Software Engineering*

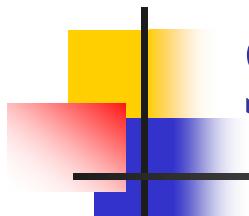
## Design Concepts

Based on Chapter 9 of the textbook



# Software Design

- Encompasses the set of principles, concepts, and practices that lead to the development of a high quality system or product.
- Design principles establish and overriding philosophy that guides the designer as the work is performed.
- Design concepts must be understood before the mechanics of design practice are applied.
- Software design practices change continuously as new methods, better analysis, and broader understanding evolve.

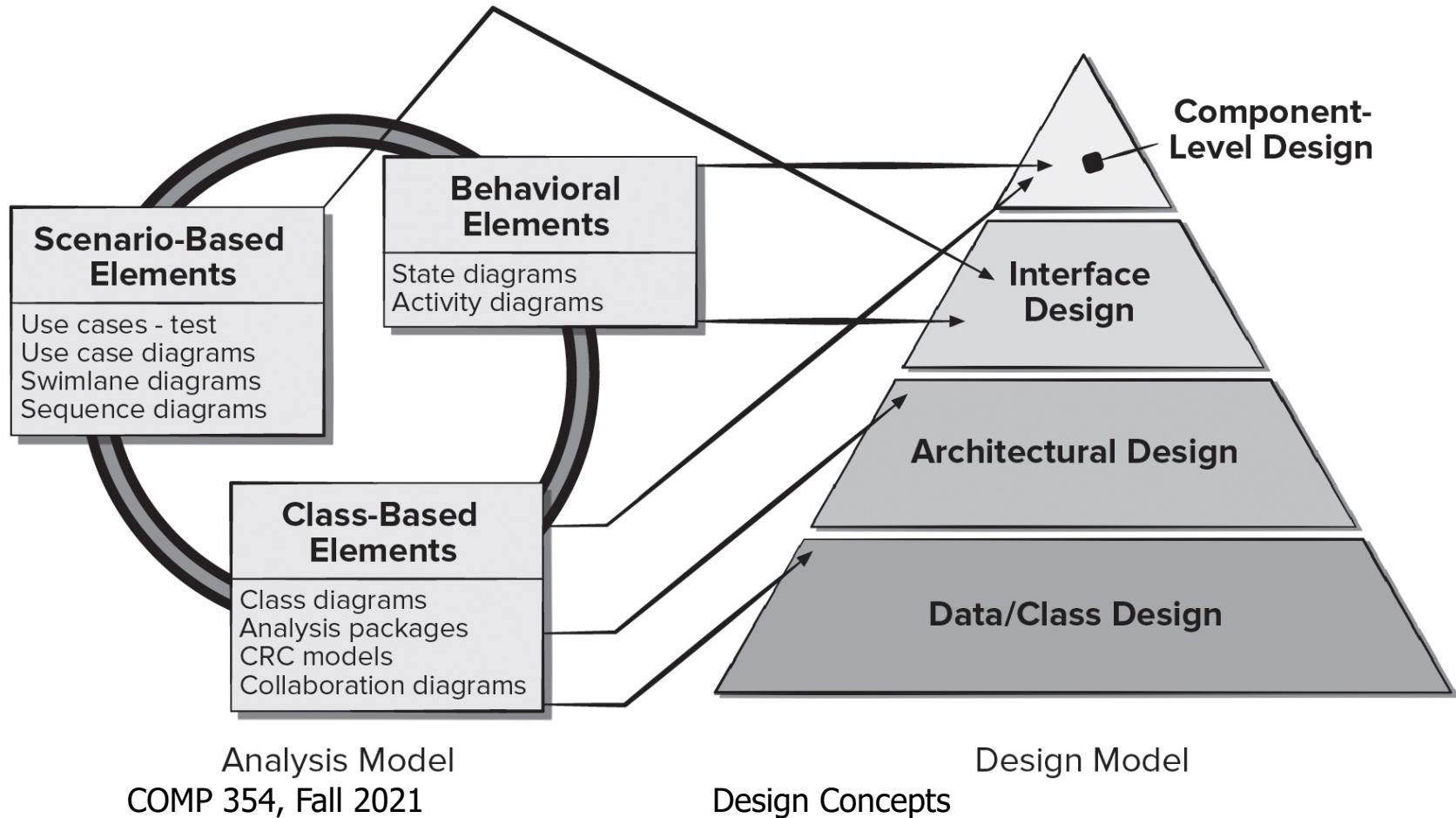


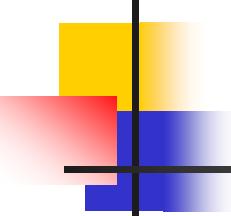
# Software Engineering Design

- **Data/Class design** – transforms analysis classes into implementation classes and data structures.
- **Architectural design** – defines relationships among the major software structural elements.
- **Interface design** – defines how software elements, hardware elements, and end-users communicate.
- **Component-level design** – transforms structural elements into procedural descriptions of software components.

# Mapping Requirements Model to Design Model

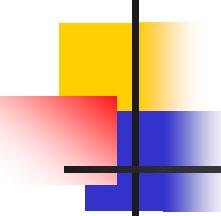
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





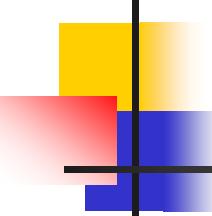
# Design and Quality

- The design must implement all of the explicit requirements contained in the analysis model, and it must accommodate all of the implicit requirements desired by the customer.
- The design should be a readable, understandable guide for those who generate code and for those who test and subsequently support the software.
- The design should provide a complete picture of the software, addressing the data, functional, and behavioral domains from an implementation perspective.



# Quality Guidelines

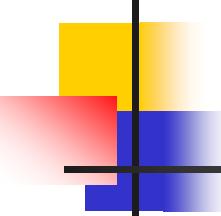
1. A design should exhibit an architecture (a) created using recognizable architectural styles or patterns, (b) composed of well designed components (c) implemented in an evolutionary fashion.
2. A design should be modular.
3. A design should contain distinct representations of data, architecture, interfaces, and components.
4. A design should lead to data structures that are drawn from recognizable data patterns.
5. A design should contain functionally independent components.
6. A design should lead to interfaces that reduce the complexity of connections between components and the external environment.
7. A design should be derived using a repeatable method that is driven by software requirements analysis.
8. A design should be represented using meaningful notation.



# Common Design Characteristics

Each new software design methodology introduces unique heuristics and notions – yet they each contain:

1. A mechanism for translating the requirements model into a design representation.
2. A notation for representing functional components and their interfaces.
3. Heuristics for refinement and partitioning.
4. Guidelines for quality assessment.

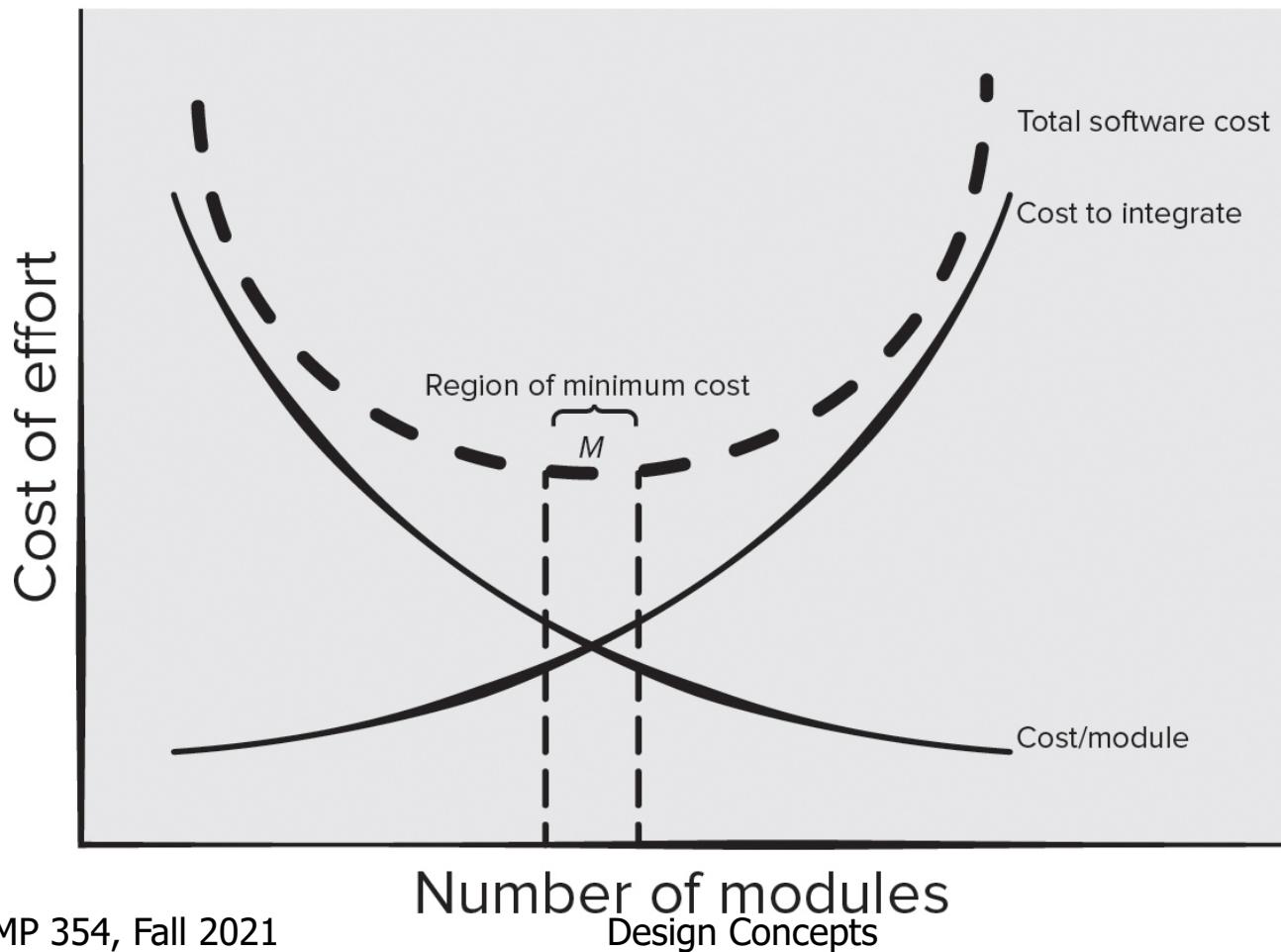


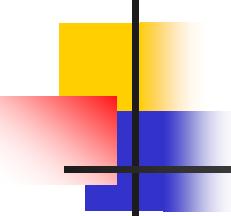
# Design Concepts

- **Abstraction** – data (named collection of data describing data object), procedural (named sequence of instructions with specific and limited function).
- **Architecture** - overall structure or organization of software components, ways components interact, and structure of data used by components.
- **Design Patterns** - describe a design structure that solves a well-defined design problem within a specific context.
- **Separation of concerns** - any complex problem can be more easily handled if it is subdivided into pieces.
- **Modularity**—compartmentalization of data and function.

# Modularity and Software Cost

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



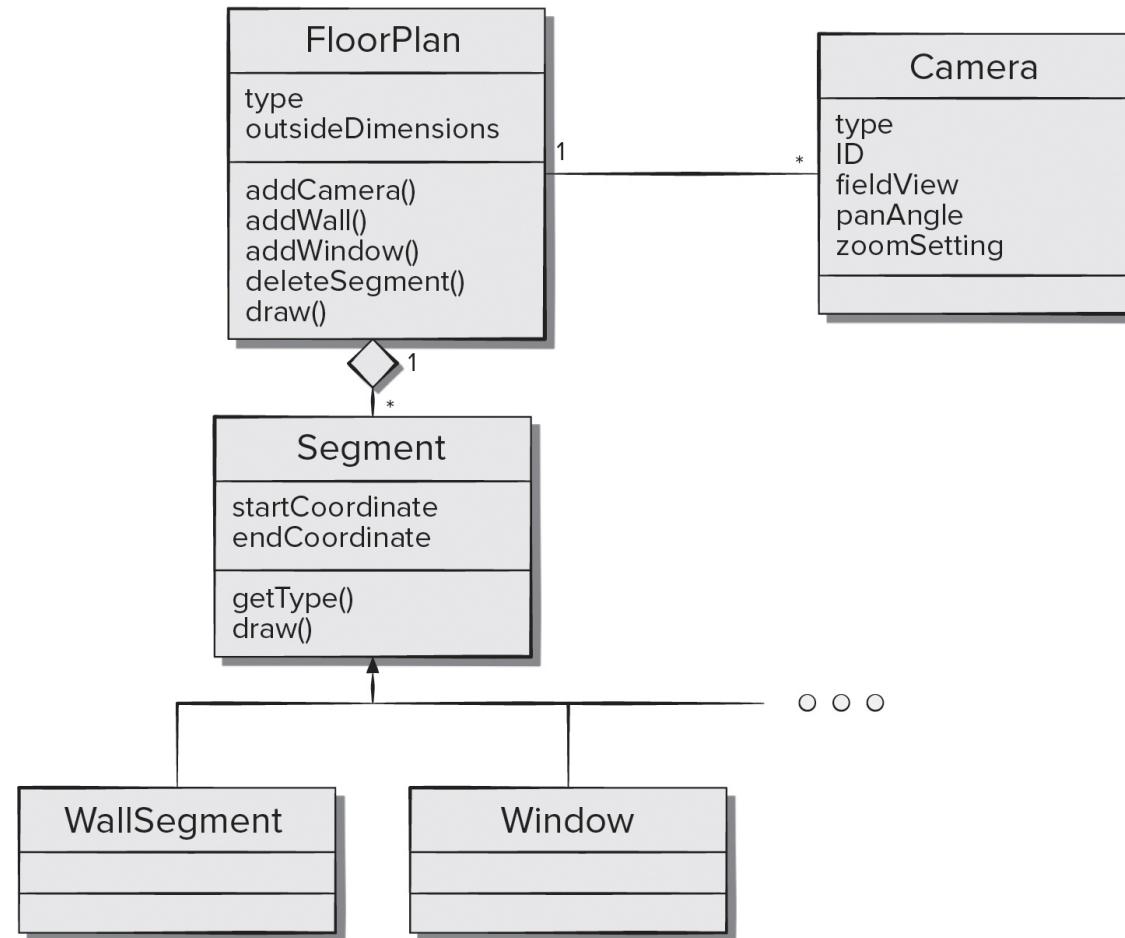


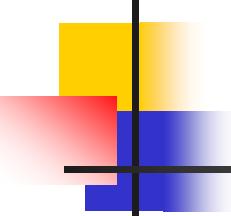
# Design Concepts

- **Information Hiding** - controlled interfaces which define and enforces access to component procedural detail and any local data structure used by the component.
- **Functional independence** - single-minded (high cohesion) components with aversion to excessive interaction with other components (low coupling).
- **Stepwise Refinement** – incremental elaboration of detail for all abstractions.
- **Refactoring**—a reorganization technique that simplifies the design without changing functionality.
- **Design Classes**—provide design detail that will enable analysis classes to be implemented.

# Design Class Example

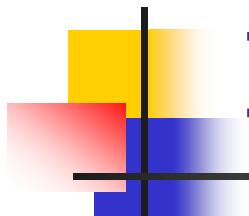
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





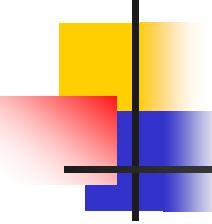
# Design Class Characteristics

- **Complete** - includes all necessary attributes and methods) and sufficient (contains only those methods needed to achieve class intent).
- **Primitiveness** – each class method focuses on providing one service.
- **High cohesion** – small, focused, single-minded classes.
- **Low coupling** – class collaboration kept to minimum.



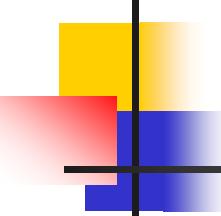
# Information Hiding

- Reduces the likelihood of “side effects.”
- Limits the global impact of local design decisions.
- Emphasizes communication through controlled interfaces.
- Discourages the use of global data.
- Leads to encapsulation—an attribute of high quality design.
- Results in higher quality software.



# Architecture Properties

- **Structural properties.** This aspect of the architectural design representation defines the components of a system (for example, modules, objects, filters) and the manner components are packaged and interact with one another.
- **Extra-functional properties.** The architectural design description should address how the design architecture achieves requirements for performance, capacity, reliability, security, adaptability, and other characteristics.
- **Families of related systems.** The architectural design should draw upon repeatable patterns (building blocks) often encountered in the design of similar systems.

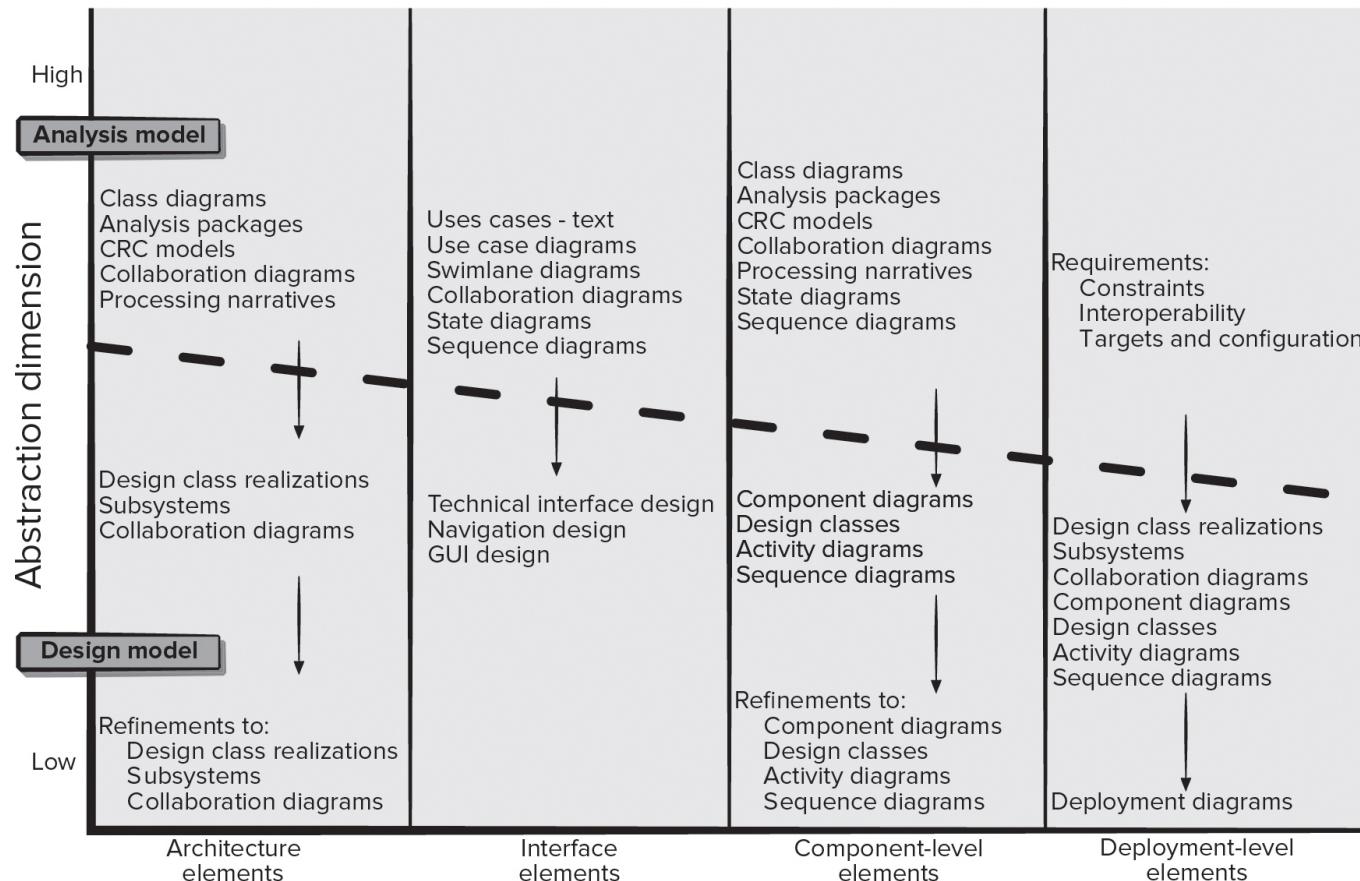


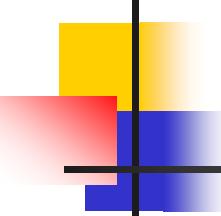
# Design Pattern Template

- **Pattern name** - describes the essence of the pattern in a short but expressive name
- **Intent** - describes the pattern and what it does
- **Also-known-as** - lists any synonyms for the pattern
- **Motivation** - provides an example of the problem
- **Applicability** - notes specific design situations in which the pattern is applicable
- **Structure** - describes the classes that are required to implement the pattern
- **Participants** - describes the responsibilities of the classes that are required to implement the pattern
- **Collaborations** - describes how the participants collaborate to carry out their responsibilities
- **Consequences** - describes the “design forces” that affect the pattern and the potential trade-offs that must be considered when the pattern is implemented
- **Related patterns** - cross-references related design patterns

# Design Model

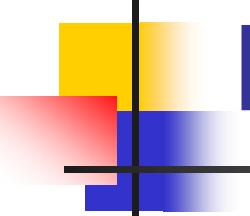
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





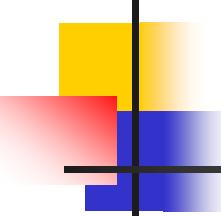
# Design Modeling Principles

- **Principle 1.** Design should be traceable to the requirements model.
- **Principle 2.** Always consider the architecture of the system to be built.
- **Principle 3.** Design of data is as important as design of processing functions.
- **Principle 4.** Interfaces (both internal and external) must be designed with care.
- **Principle 5.** User interface design should be tuned to the needs of the end-user and stress ease of use.



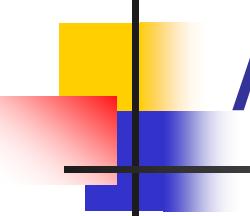
# Design Modeling Principles

- **Principle 6.** Component-level design should be functionally independent.
- **Principle 7.** Components should be loosely coupled to each other than the environment.
- **Principle 8.** Design representations (models) should be easily understandable.
- **Principle 9.** The design should be developed iteratively.
- **Principle 10.** Creation of a design model does not preclude using an agile approach.



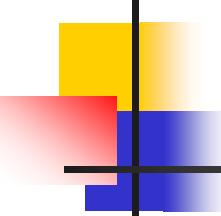
# Data Design Elements

- Data model – data objects and database architectures.
  - Examines data objects independently of processing.
  - Focuses attention on the data domain.
  - Creates a model at the customer's level of abstraction.
  - Indicates how data objects relate to one another.
- Data object can be an external entity, a thing, an event, a place, a role, an organizational unit, or a structure.
- Data objects contain a set of attributes that act as an quality, characteristic, or descriptor of the object.
- Data objects may be connected to one another in many different ways.



# Architectural Design Elements

- Architectural design for software - equivalent to the floor plan for a house.
- The architectural model is derived from three sources:
  - Information about the application domain for the software to be built.
  - Specific requirements model elements such as data flow analysis classes and their relationships (collaborations) for the problem at hand, and
  - Availability of architectural patterns and styles.

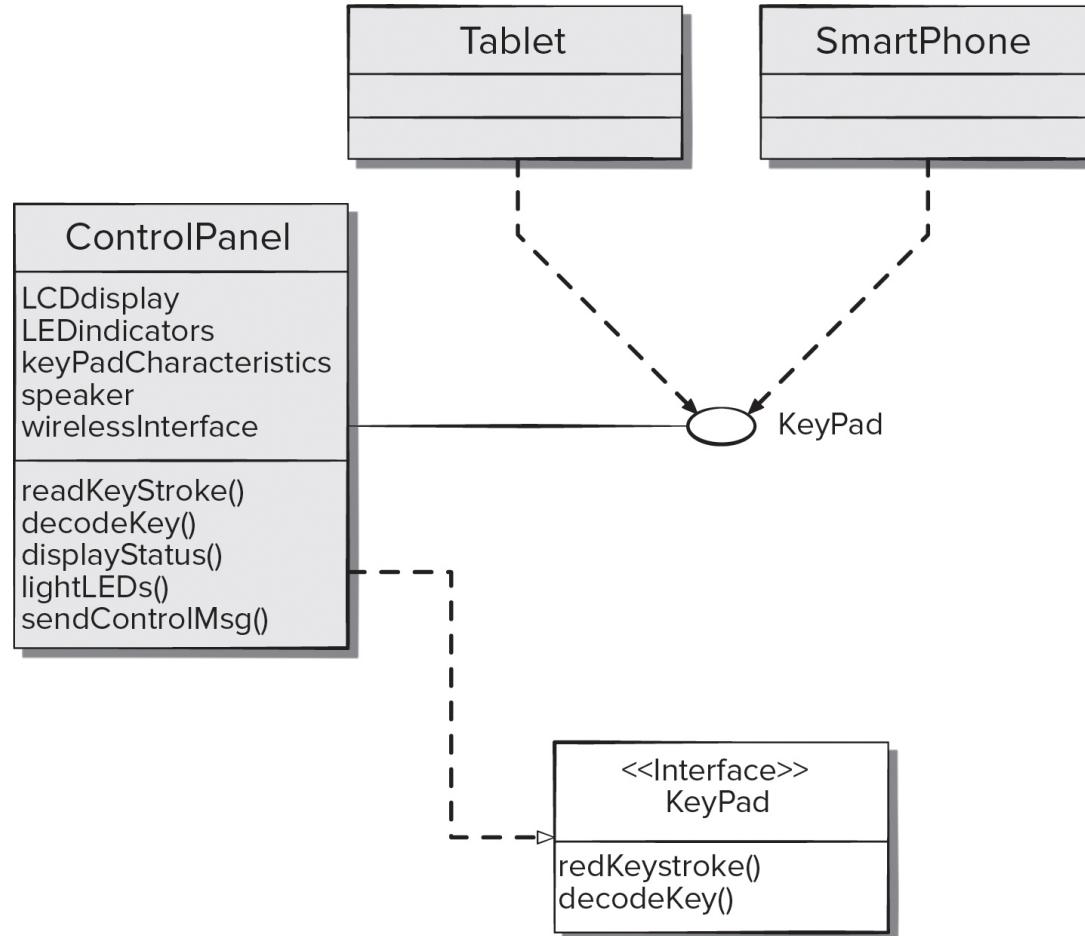


# Interface Design Elements

- Interface is a set of operations that describes the externally observable behavior of a class and provides access to its public operations.
- Important elements:
  - User interface (UI).
  - External interfaces to other systems.
  - Internal interfaces between various design components.
- UI or User Experience (UX) is a major engineering action to ensure the creation on usable software products.
- Internal and external interfaces should incorporate both error checking and appropriate security features.

# Interface Model for Control Panel

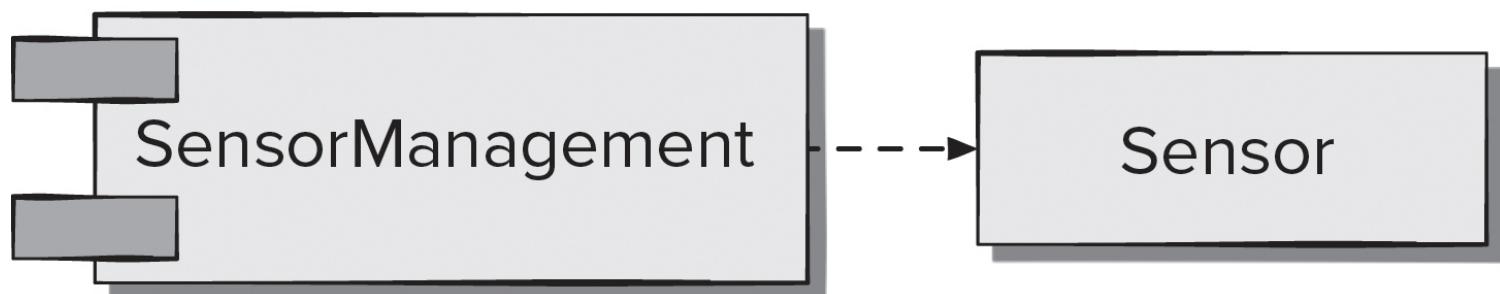
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

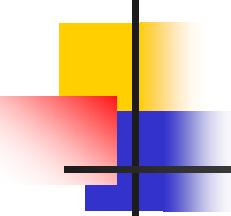


# Component-Level Design Elements

- Describes the internal detail of each software component.
- Defines:
  - Data structures for all local data objects.
  - Algorithmic detail for all component processing functions.
  - Interface that allows access to all component operations.
- Modeled using UML component diagrams.

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





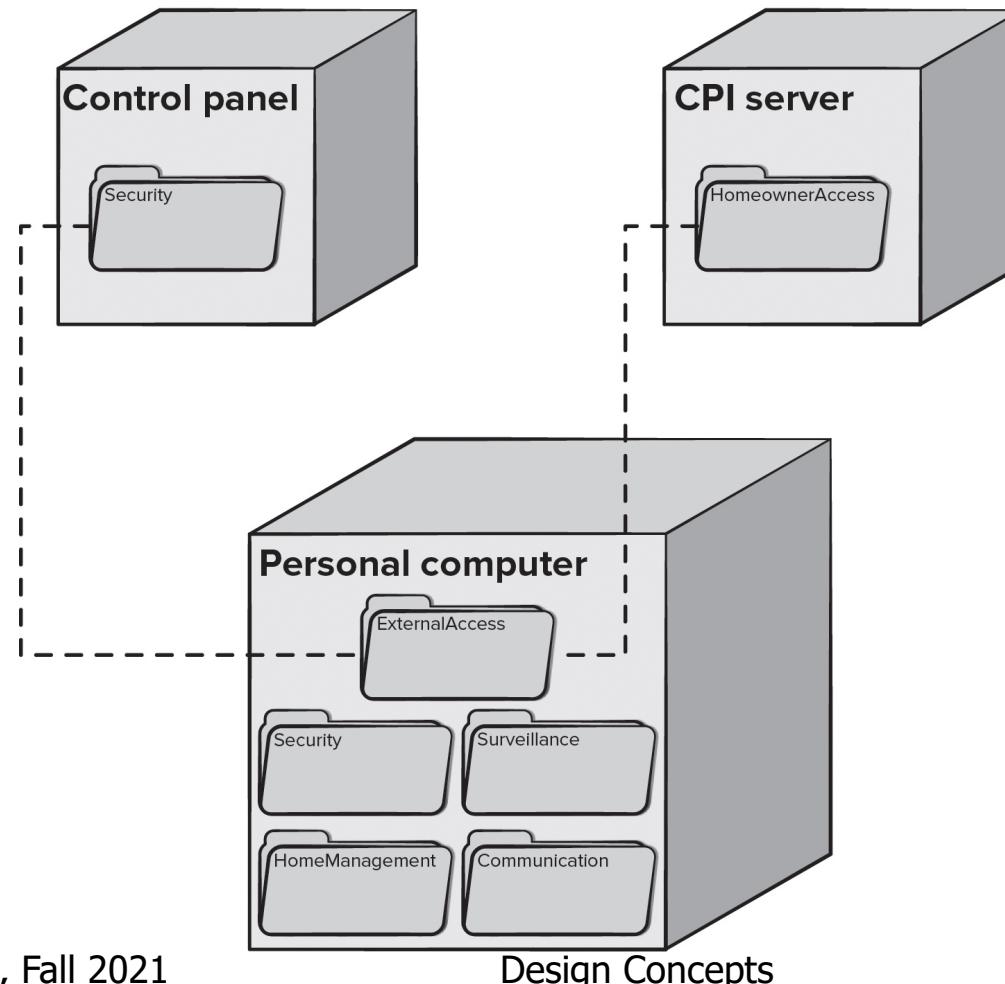
# Deployment Design Elements

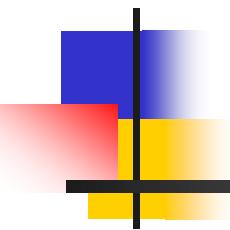
- Indicates how software functionality and subsystems will be allocated within the physical computing environment.
- Modeled using UML deployment diagrams.
  - Descriptor form deployment diagrams - show the computing environment but does not indicate configuration details.
  - Instance form deployment diagrams - identify specific hardware configurations and are developed in the latter stages of design.

# UML

# Deployment Instance Diagram

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

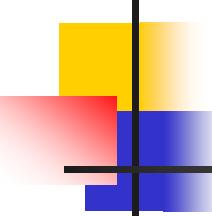




# *COMP 354: Introduction to Software Engineering*

## Architectural Design

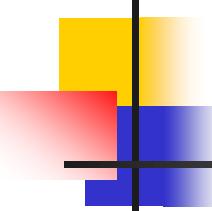
Based on Chapter 10 of the textbook



# What is Software Architecture?

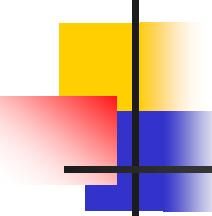
The architecture is not the operational software, it is a representation that enables a software engineer to:

1. Analyze the effectiveness of the design in meeting its stated requirements,
2. Consider architectural alternatives at a stage when making design changes is still relatively easy, and
3. Reduce the risks associated with the construction of the software.



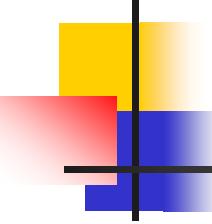
# Why is Software Architecture Important?

- Software architecture provides a representation that facilitates communication among all stakeholders interested in the development of a computer-based system.
- Architecture highlights early design decisions that will have a profound impact on all software engineering work that follows.
- Architecture constitutes a relatively small, intellectually graspable mode of how the system is structured and how its components work together.



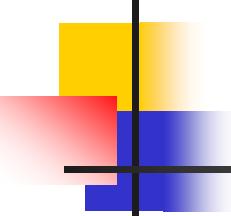
# Architectural Descriptions

- The IEEE Computer Society has proposed IEEE-Std-1528-2011, Systems and Software Engineering – Architecture Description.
  - Describes the use of architecture viewpoints, architecture frameworks, and architecture description languages as a means of codifying the conventions and common practices for architectural description.
- The IEEE Standard defines an architectural description (AD) as a “a collection of products to document an architecture.”
  - An architecture description shall identify the system stakeholders having concerns considered fundamental to the architecture of the system-of-interest.
  - These concerns shall be considered when applicable and identified in the architecture description: system purpose, suitability of the architecture, feasibility of constructing and deploying the system, risks and impacts of the system, and the maintainability and evolvability of the system.



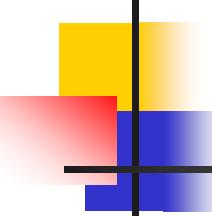
# Architecture Decision Documentation

1. Determine information items needed for each decision.
2. Define links between each decision and appropriate requirements.
3. Provide mechanisms to change status when alternative decisions need to be evaluated.
4. Define prerequisite relationships among decisions to support traceability.
5. Link significant decisions to architectural views resulting from decisions.
6. Document and communicate all decisions as they are made.



# Agility and Architecture

- To avoid rework, user stories are used to create and evolve an architectural model (walking skeleton) before beginning any coding.
- Use models which allow software architects to add user stories to the evolving storyboard and works with the product owner to prioritize the architectural stories as “sprints” (work units) are planned.
- Well run agile projects include delivery of architectural documentation during each sprint.
- After the sprint is completed, the architect reviews the working prototype for quality before the team presents it to the stakeholders in a formal sprint review.



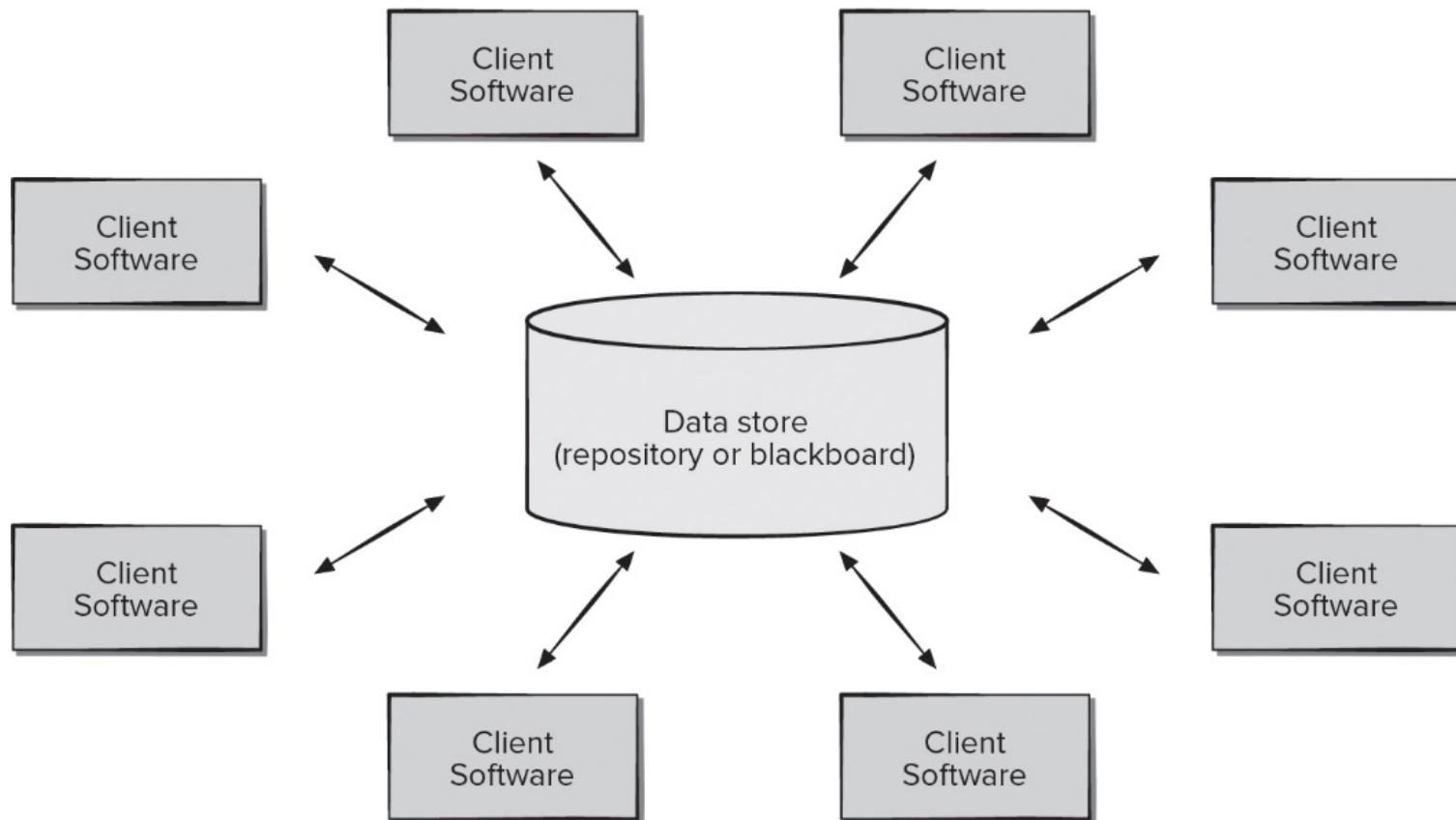
# Architectural Styles

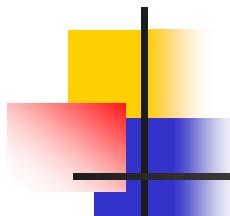
Each style describes a system category that encompasses:

1. **set of components** (for example: a database, computational modules) that perform a function required by a system.
2. **set of connectors** that enable “communication, coordination and cooperation” among components.
3. **constraints** that define how components can be integrated to form the system.
4. **semantic models** that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

# Data Centered Architecture

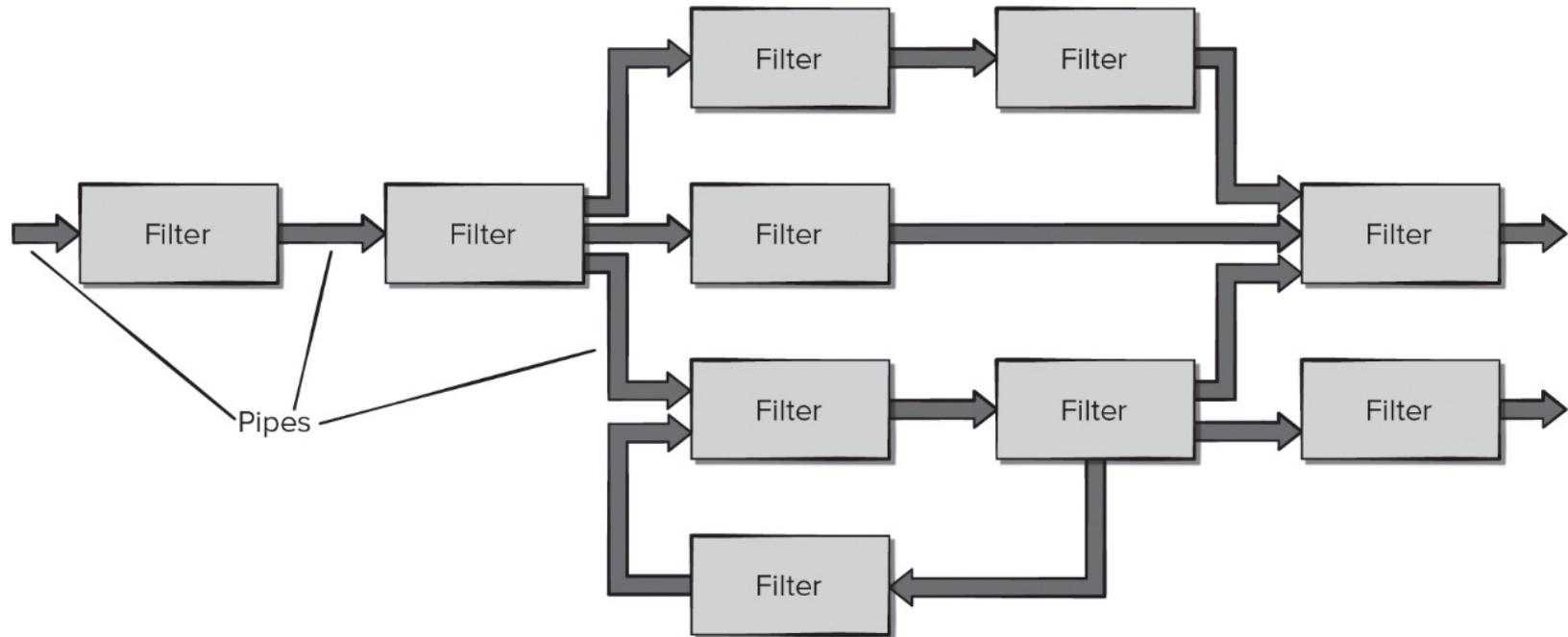
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

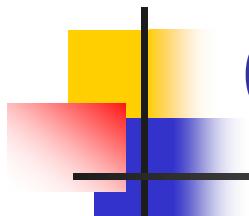




# Data Flow Architecture

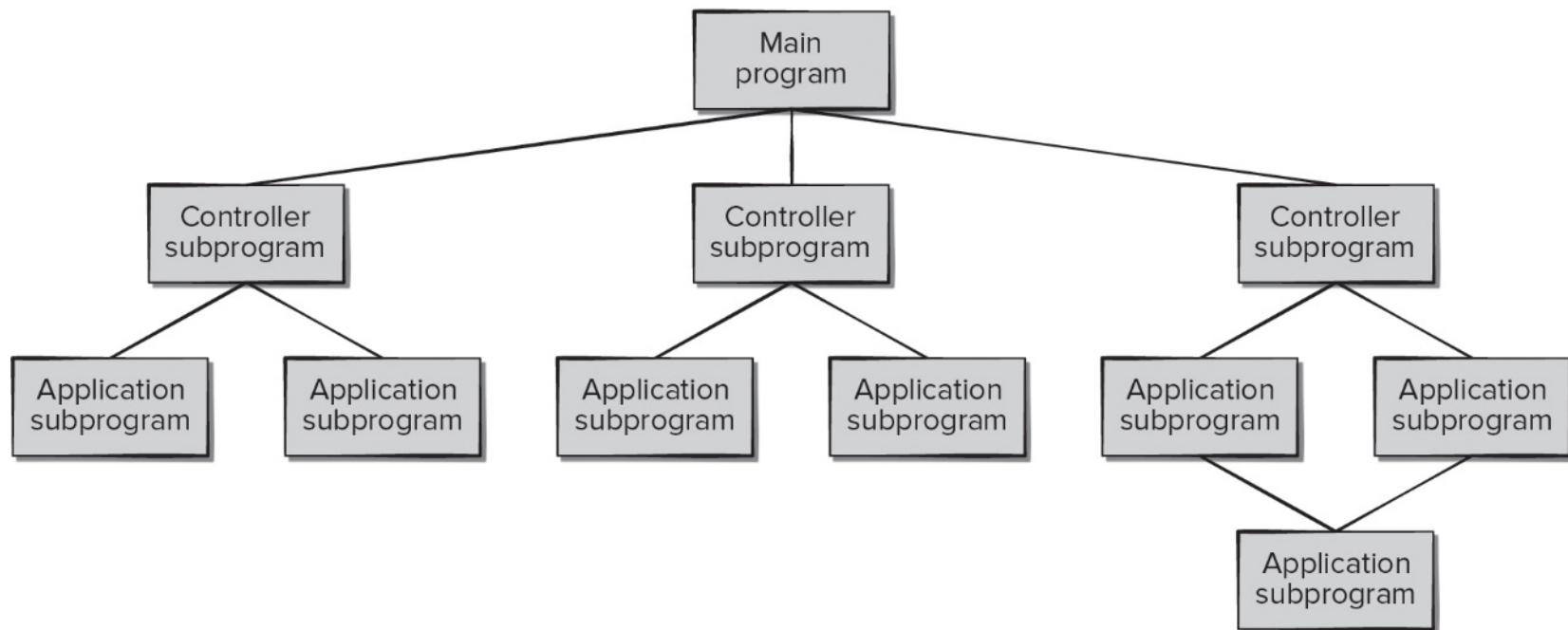
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





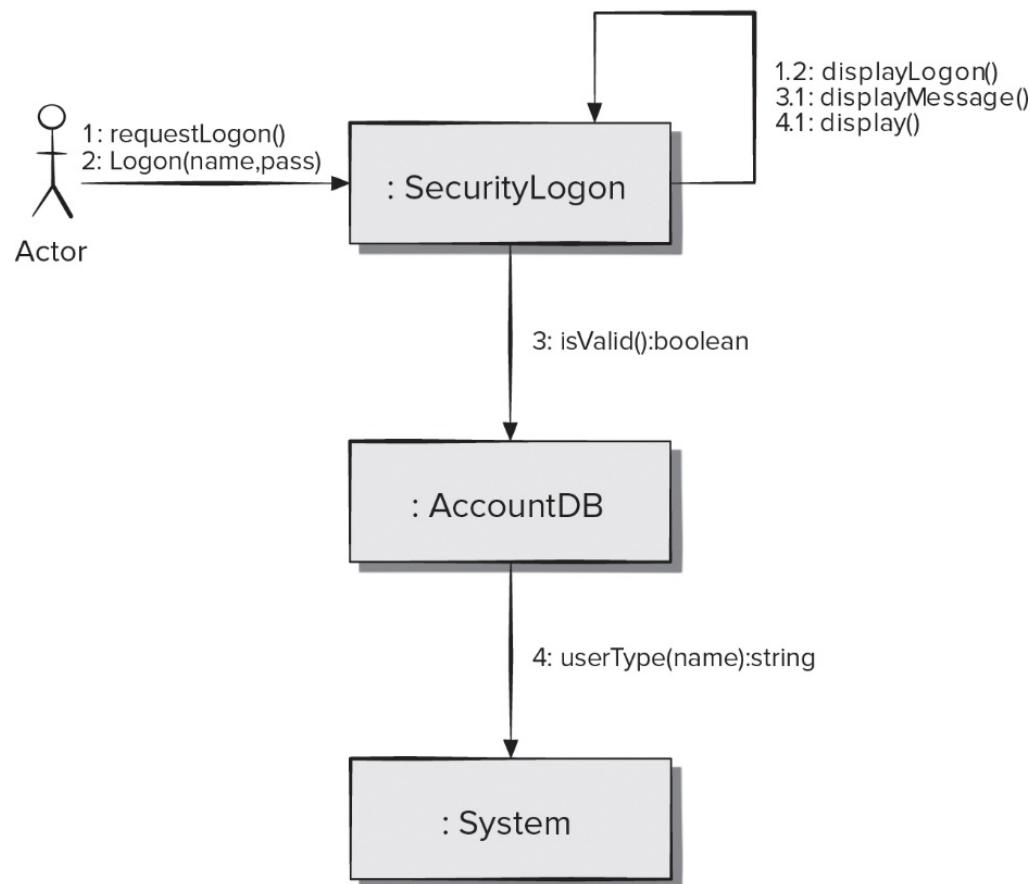
# Call Return Architecture

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



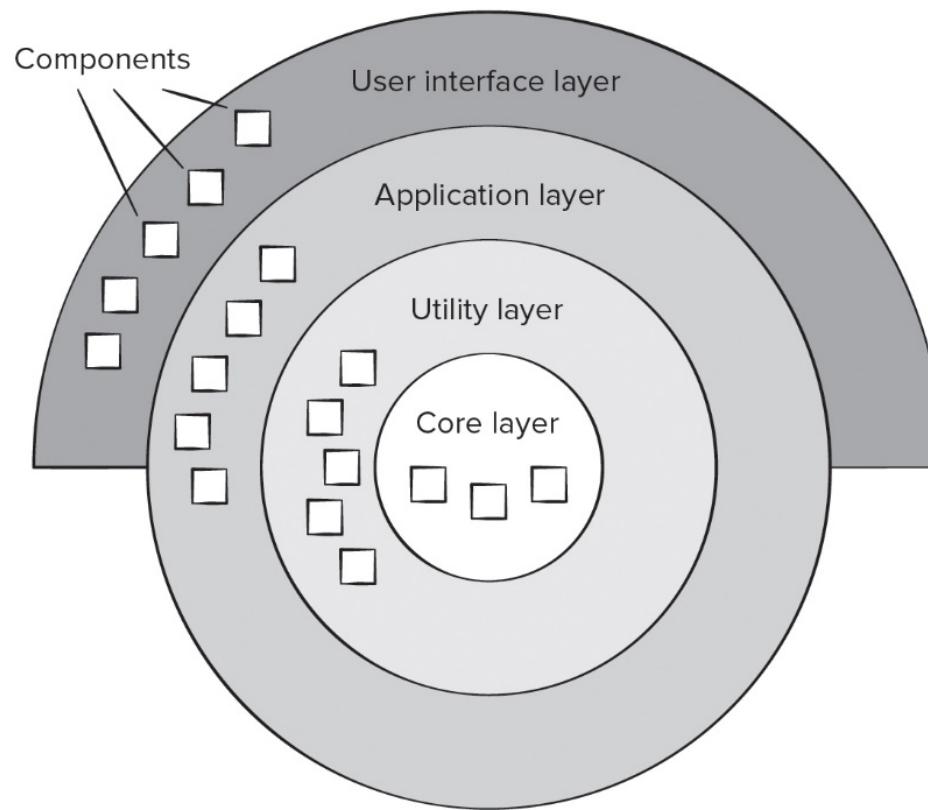
# Object-Oriented Architecture

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



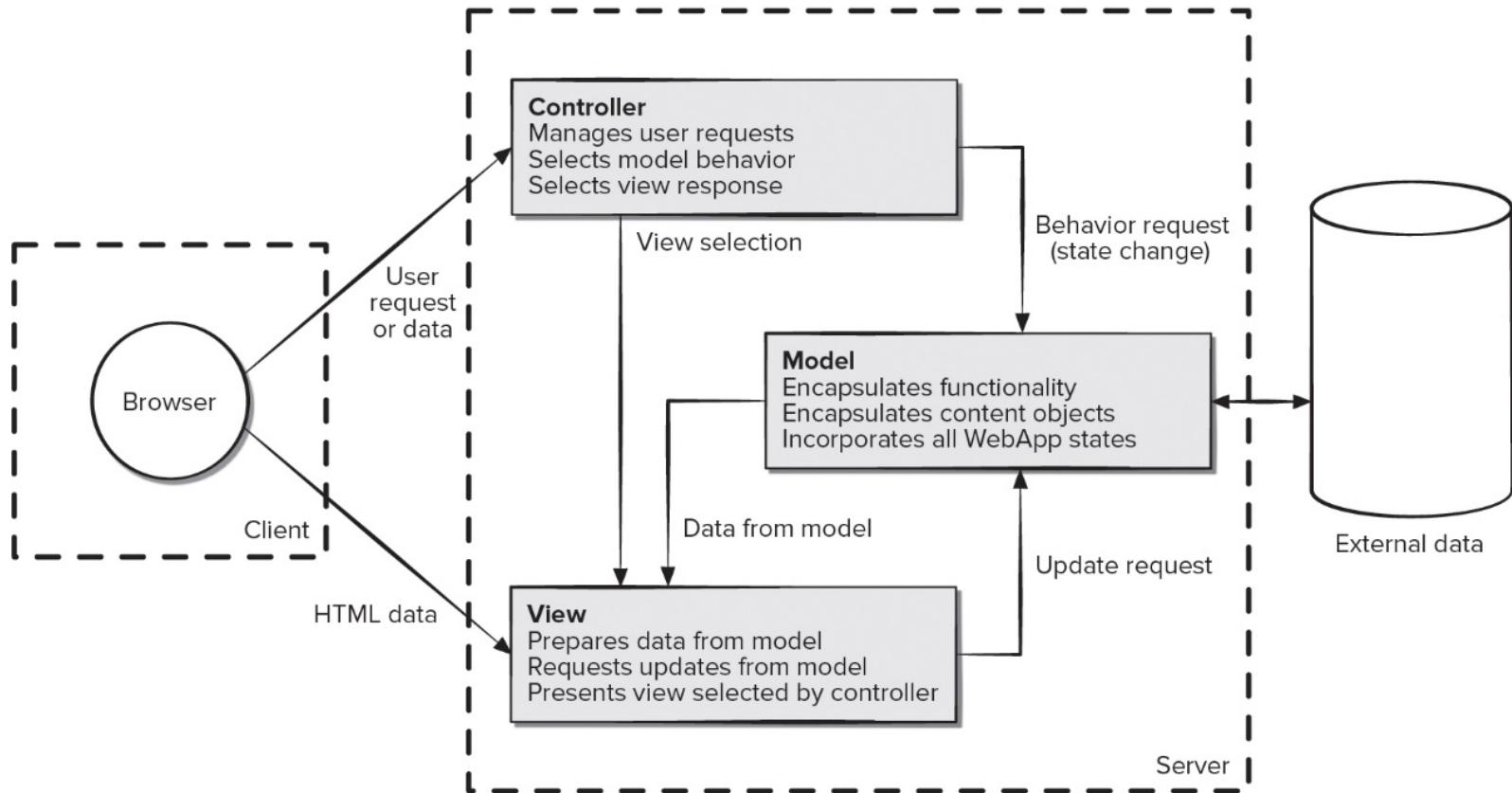
# Layered Architecture

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



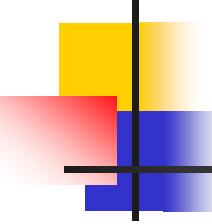
# Model View Controller Architecture

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



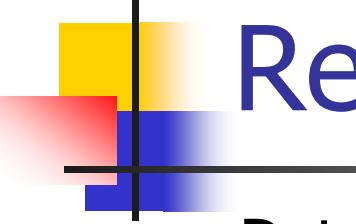
Source: Adapted from Jacyntho, Mark Douglas, Schwabe, Daniel and Rossi, Gustavo, "An Architecture for Structuring Complex Web Applications," 2002, available at <http://www-di.inf.puc-rio.br/schwabe/papers/OOHDJava2%20Report.pdf>

# Architectural Organization and Refinement



Control.

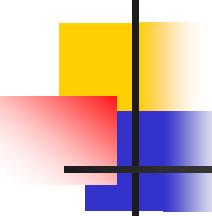
- How is control managed within the architecture?
- Does a distinct control hierarchy exist, and if so, what is the role of components within this control hierarchy?
- How do components transfer control within the system?
- How is control shared among components?
- What is the control topology (that is, the geometric form that the control takes)?
- Is control synchronized, or do components operate asynchronously?



# Architectural Organization and Refinement

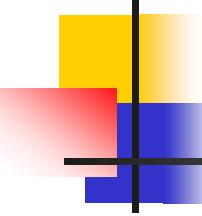
Data.

- How are data communicated between components?
- Is the flow of data continuous, or are data objects passed to the system sporadically?
- What is the mode of data transfer?
- Do data components exist, and if so, what is their role?
- How do functional components interact with data components?
- Are data components passive or active?
- How do data and control interact within the system?



# Architectural Considerations

- **Economy** – software is uncluttered and relies on abstraction to reduce unnecessary detail.
- **Visibility** – Architectural decisions and their justifications should be obvious to software engineers who review.
- **Spacing** – Separation of concerns in a design without introducing hidden dependencies.
- **Symmetry** – Architectural symmetry implies that a system is consistent and balanced in its attributes.
- **Emergence** – Emergent, self-organized behavior and control are key to creating scalable, efficient, and economic software architectures.

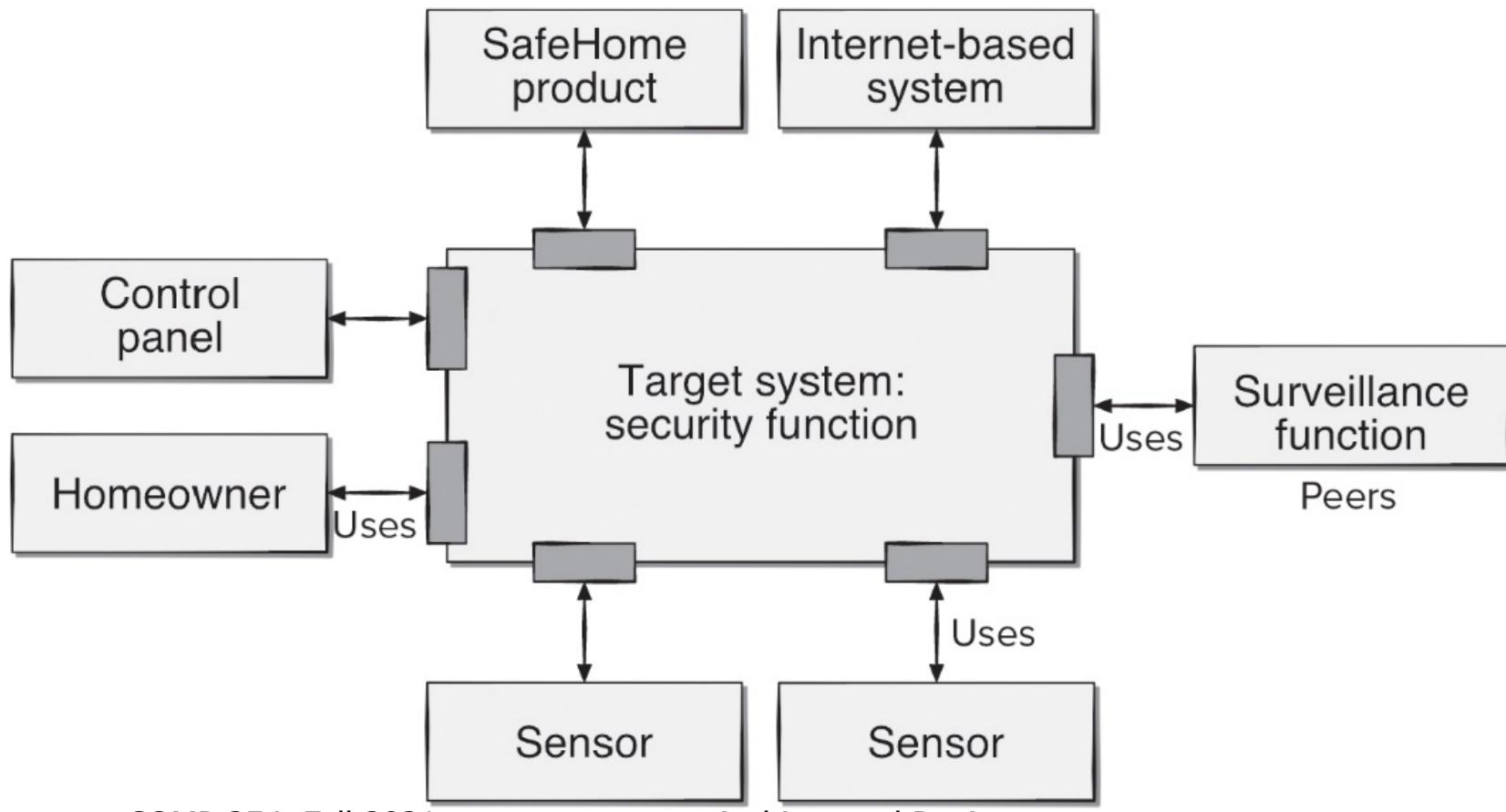


# Architectural Design

- The software must be placed into context.
  - The design should define the external entities (other systems, devices, people) that the software interacts with and the nature of the interaction.
- A set of architectural archetypes should be identified.
  - An **archetype** is an abstraction (similar to a class) that represents one element of system behavior.
- The designer specifies the structure of the system by defining and refining software components that implement each archetype.

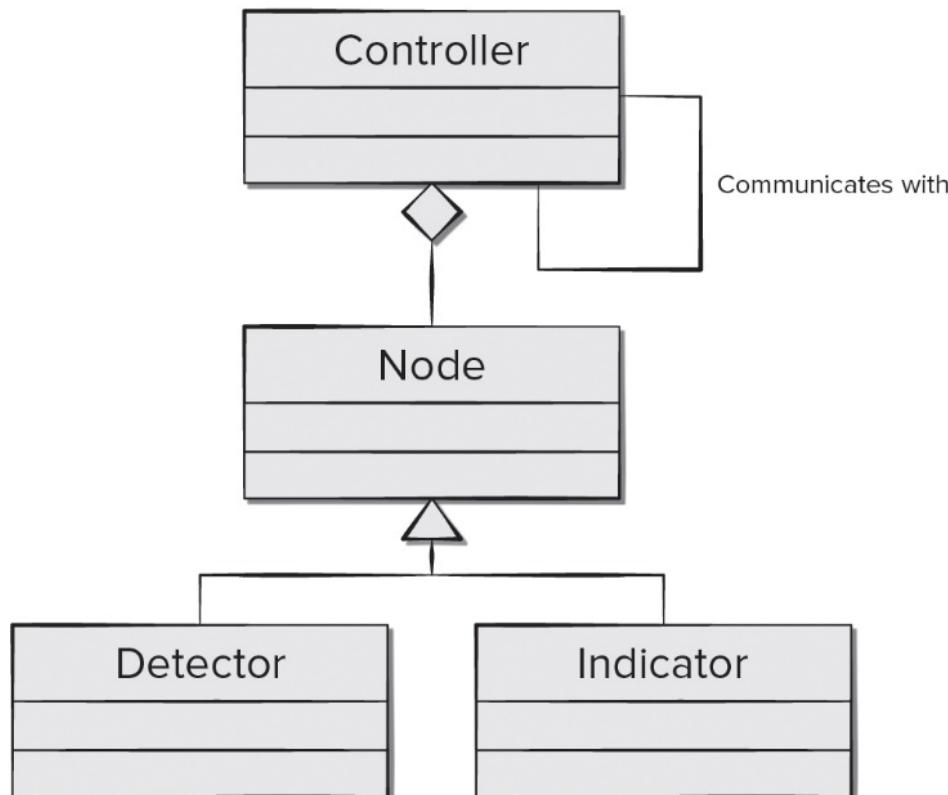
# Architecture Context Diagram

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# SafeHome Security Function Archetype

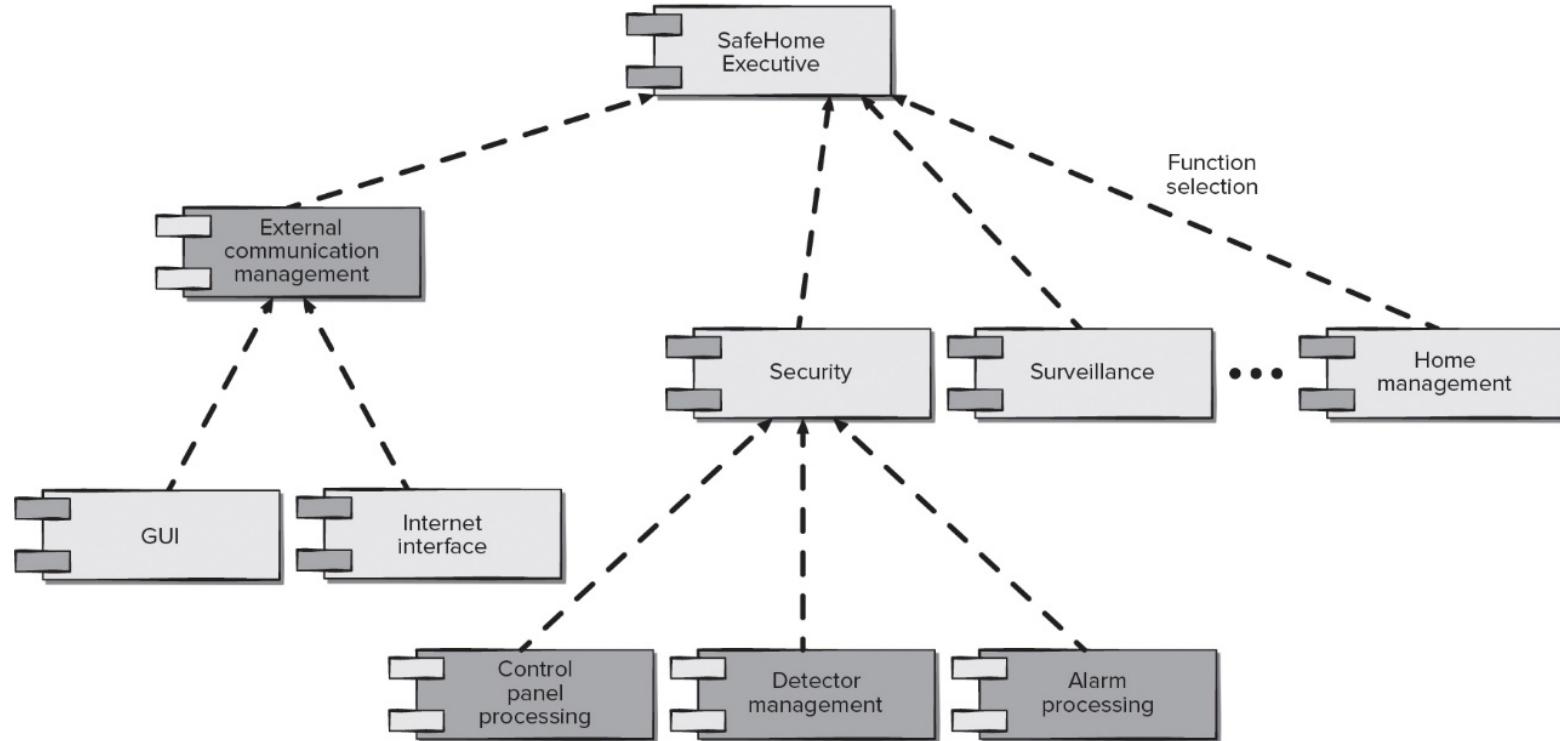
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



Source: Adapted from Bosch, Jan, Design & Use of Software Architectures. Pearson Education, 2000.

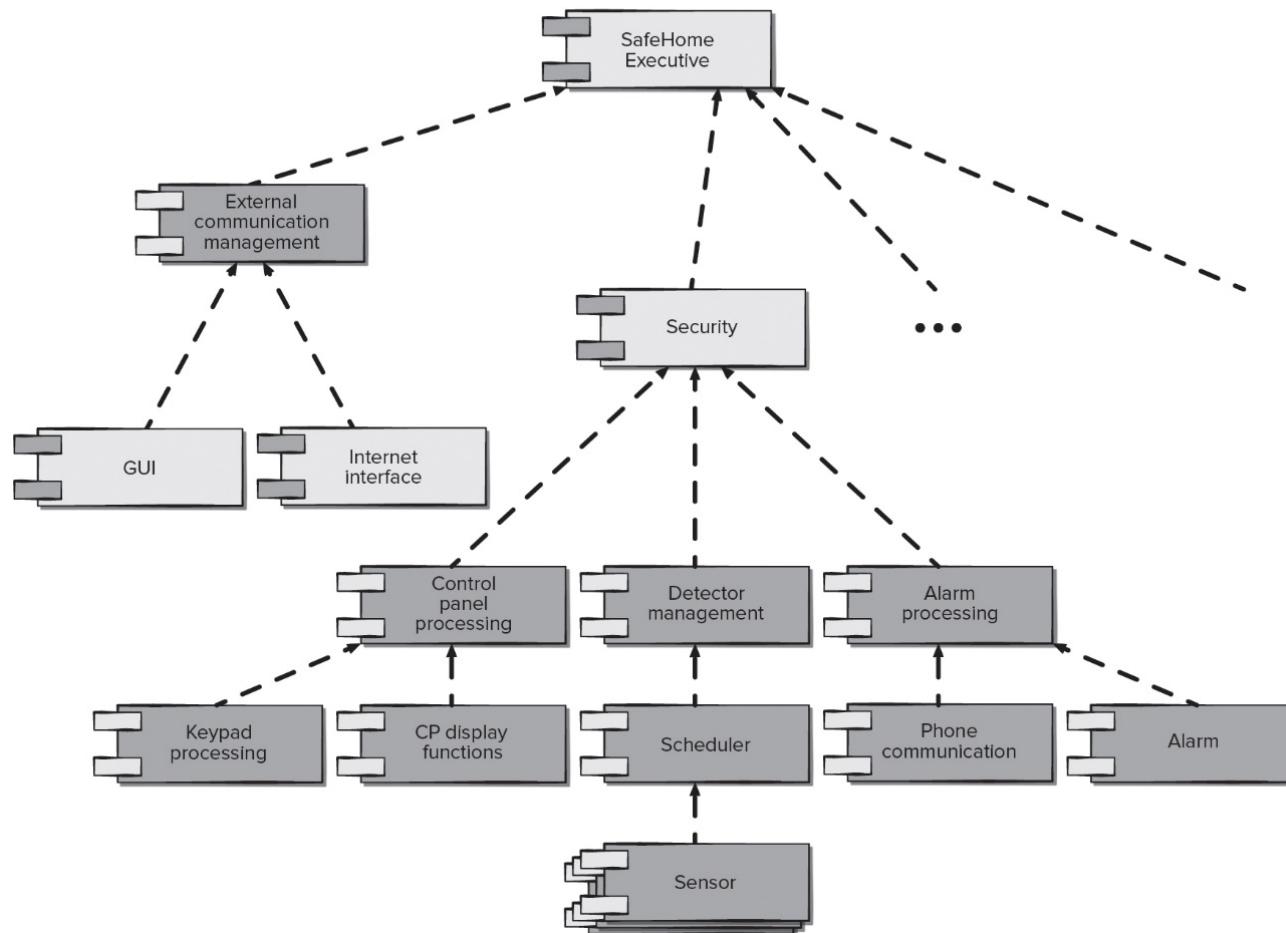
# SafeHome Top-Level Component Architecture

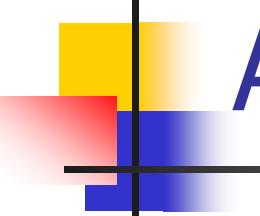
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# SafeHome Refined Component Architecture

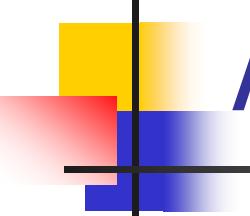
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





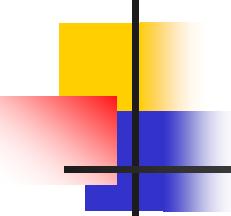
# Architectural Tradeoff Analysis

1. Collect scenarios.
2. Elicit requirements, constraints, environment description.
3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements using one of these views: module, process, data flow.
4. Evaluate quality attributes by considering each one in isolation.
5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
6. Critique candidate architectures (developed in step 3) using the sensitivity analysis (conducted in step 5).



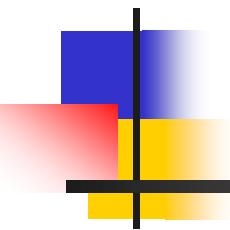
# Architectural Reviews

- Assess the ability of the software architecture to meet the systems quality requirements and identify potential risks.
- Have the potential to reduce project costs by detecting design problems early.
- Often make use of experience-based reviews, prototype evaluation, and scenario reviews, and checklists.



# Pattern-based Architectural Reviews

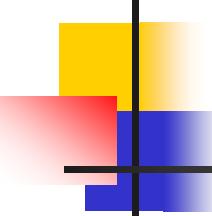
1. Identify and discuss the quality attributes by walking through the use cases.
2. Discuss a diagram of system's architecture in relation to its requirements.
3. Identify the architecture patterns used and match the system's structure to the patterns' structure.
4. Use existing documentation and use cases to determine each pattern's effect on quality attributes.
5. Identify all quality issues raised by architecture patterns used in the design.
6. Develop a short summary of issues uncovered during the meeting and make revisions to the walking skeleton.



# *COMP 354: Introduction to Software Engineering*

## Component-Level Design

Based on Chapter 11 of the textbook

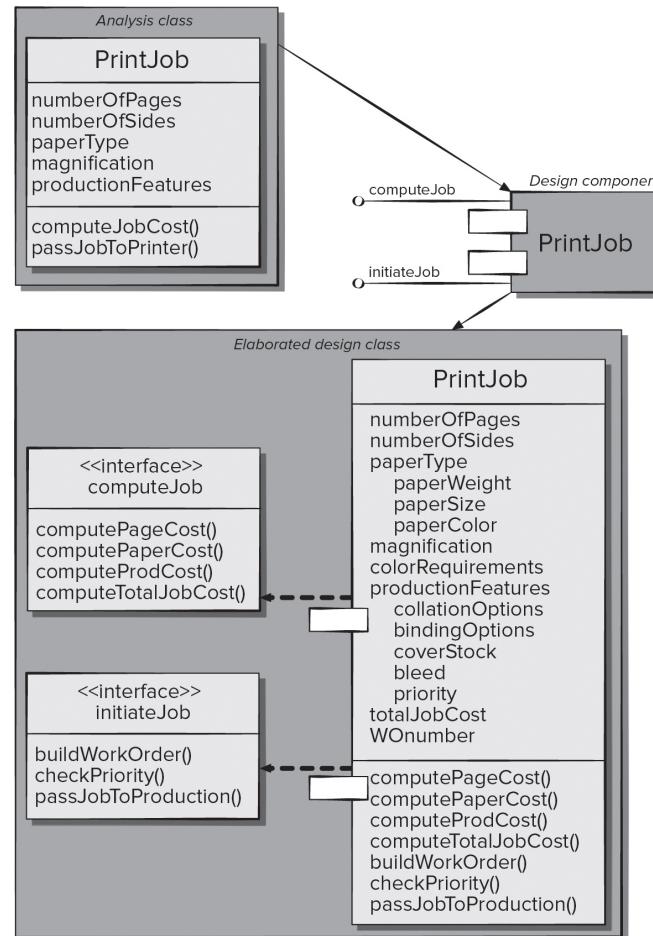


# What is a component?

- *OMG Unified Modeling Language Specification* defines a component as "... a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces."
- **Object-Oriented view:** a component contains a set of collaborating classes.
- **Traditional view:** a component contains processing logic, internal data structures that are required to implement the processing logic, and an interface that enables the component to be invoked and data to be passed to it.
- **Process-related view:** building systems out of reusable software components or design patterns selected from a catalog (component-based software engineering).

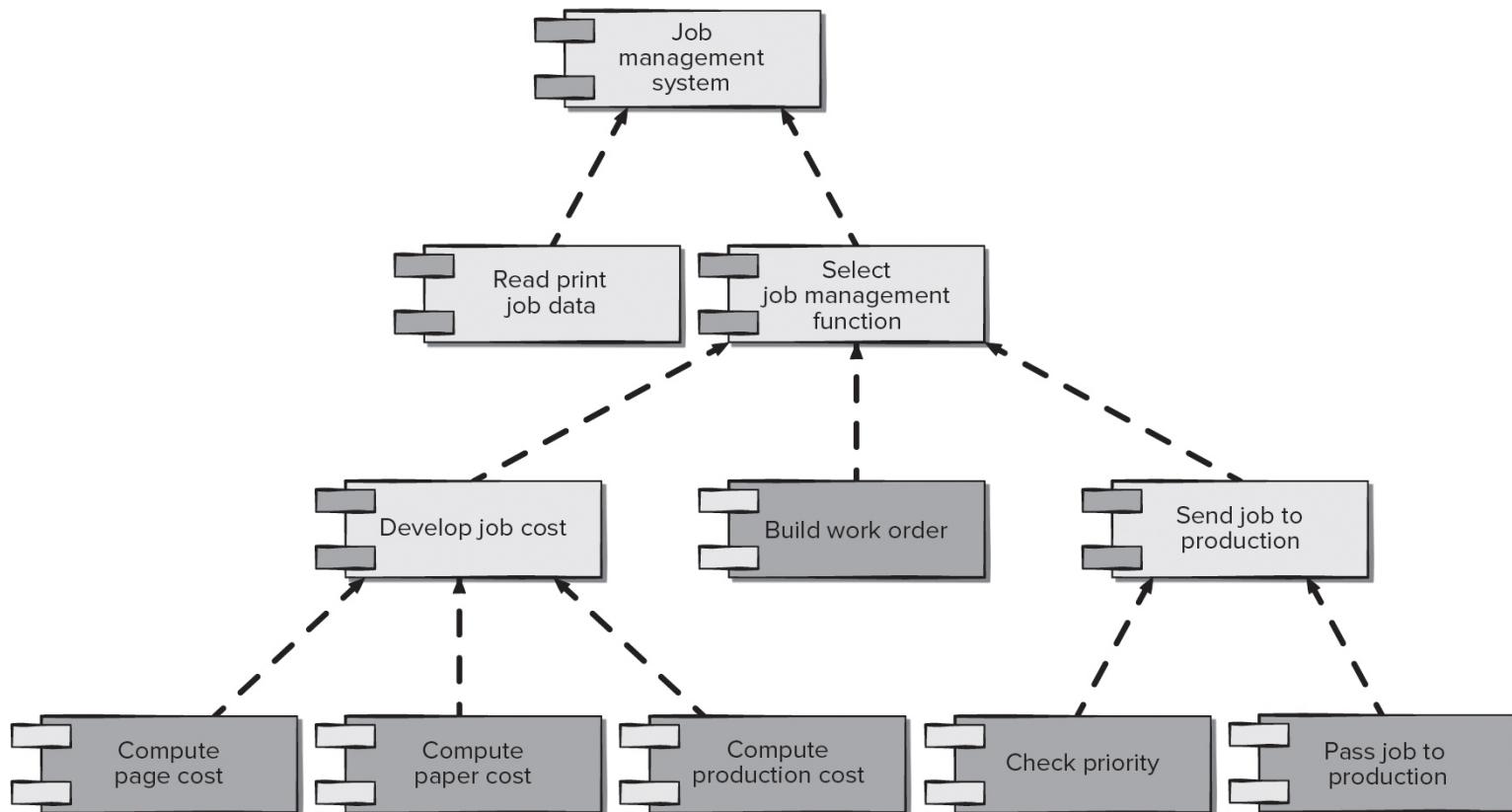
# Class-based Component-Level Design

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



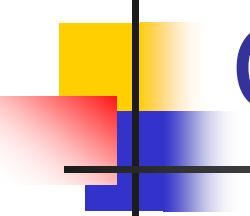
# Traditional Component-Level Design

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



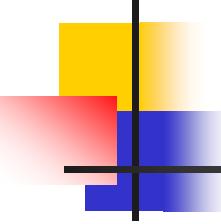
# Basic Component Design Principles

- **Open-Closed Principle (OCP).** "A module [component] should be open for extension but closed for modification."
- **Liskov Substitution Principle (LSP).** "Subclasses should be substitutable for their base classes."
- **Dependency Inversion Principle (DIP).** "Depend on abstractions. Do not depend on concretions."
- **Interface Segregation Principle (ISP).** "Many client-specific interfaces are better than one general purpose interface."
- **Release Reuse Equivalency Principle (REP).** "The granule of reuse is the granule of release."
- **Common Closure Principle (CCP).** "Classes that change together belong together."
- **Common Reuse Principle (CRP).** "Classes that aren't reused together should not be grouped together."



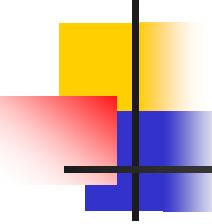
# Component-Level Design Guidelines

- **Components** - Naming conventions should be established for components that are specified as part of the architectural model and then refined and elaborated as part of the component-level model.
- **Interfaces** - provide important information about communication and collaboration (as well as helping us to achieve the OCP).
- **Dependencies and Inheritance** – For readability, it is a good idea to model dependencies from left to right and inheritance from bottom (derived classes) to top (base classes).



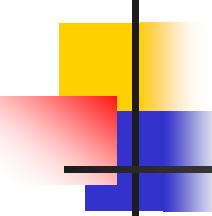
# Cohesion

- **Traditional view** - the “single-mindedness” of a module.
- **Object-Oriented view** - cohesion implies that a component encapsulates only attributes and operations that are closely related to one another and the component itself.
- **Levels of cohesion:**
  - Functional - module performs one and only one computation.
  - Layer - occurs when a higher layer accesses the services of a lower layer, but lower layers do not access higher layers.
  - Communicational - All operations that access the same data are defined within one class.



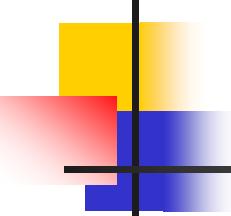
# Coupling

- **Traditional view** - degree to which a component is connected to other components and to the external world.
- **Object-Oriented view** - qualitative measure of the degree to which classes are connected to one another.
- **Levels of coupling.**
  - Content - occurs when one component “surreptitiously modifies data that is internal to another component”.
  - Control – occurs when control flags a passed to components to requests alternate behaviors when invoked.
  - External - occurs when a component communicates or collaborates with infrastructure components.



# Component-Level Design

- Step 1. Identify all design classes that correspond to the problem domain.
- Step 2. Identify all design classes that correspond to the infrastructure domain.
- Step 3. Elaborate all design classes that are not acquired as reusable components.
  - Step 3a. Specify message details when classes or component collaborate.
  - Step 3b. Identify appropriate interfaces for each component.
  - Step 3c. Elaborate attributes and define data types and data structures required to implement them.
  - Step 3d. Describe processing flow within each operation in detail.

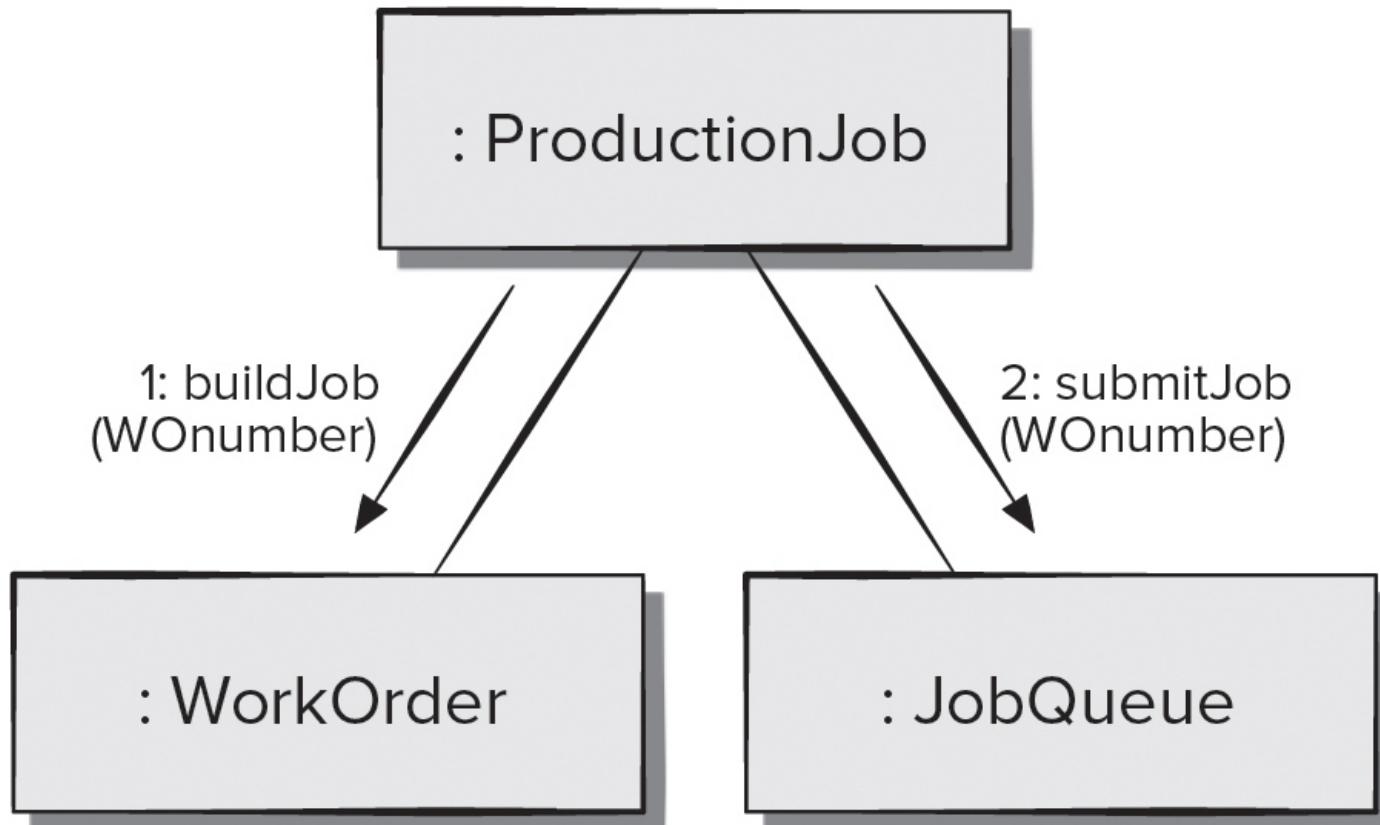


# Component-Level Design

- Step 4. Describe persistent data sources (databases and files) and identify the classes required to manage them.
- Step 5. Develop and elaborate behavioral representations for a class or component.
- Step 6. Elaborate deployment diagrams to provide additional implementation detail.
- Step 7. Factor every component-level design representation and always consider alternatives.

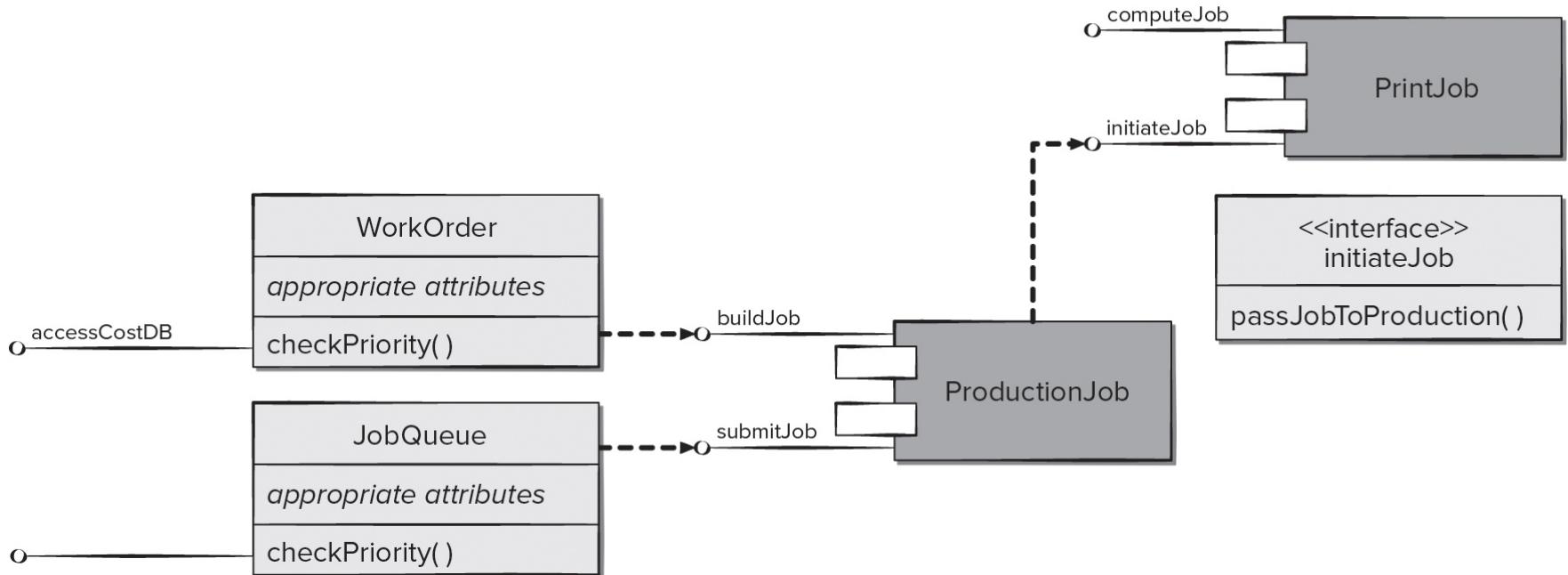
# Collaboration Diagram with Message Detail

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



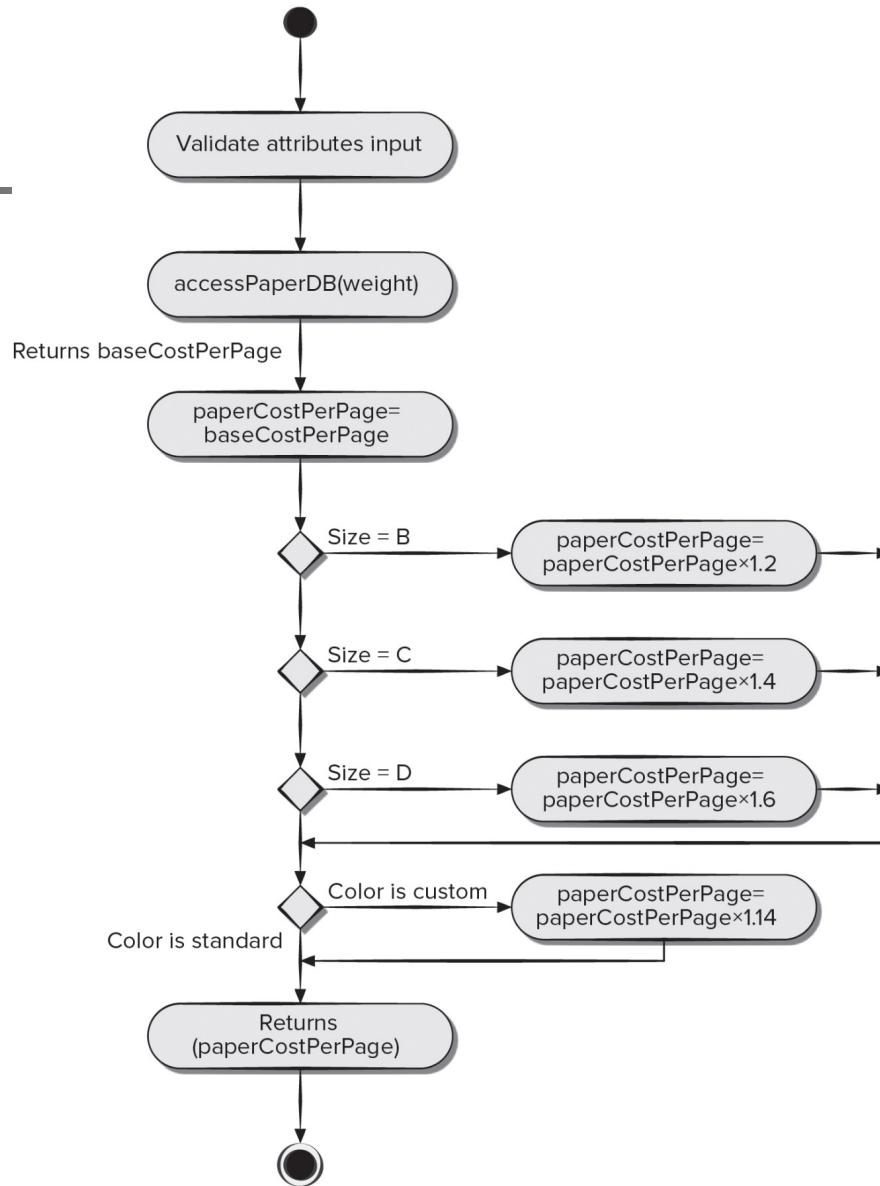
# Define Interfaces

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



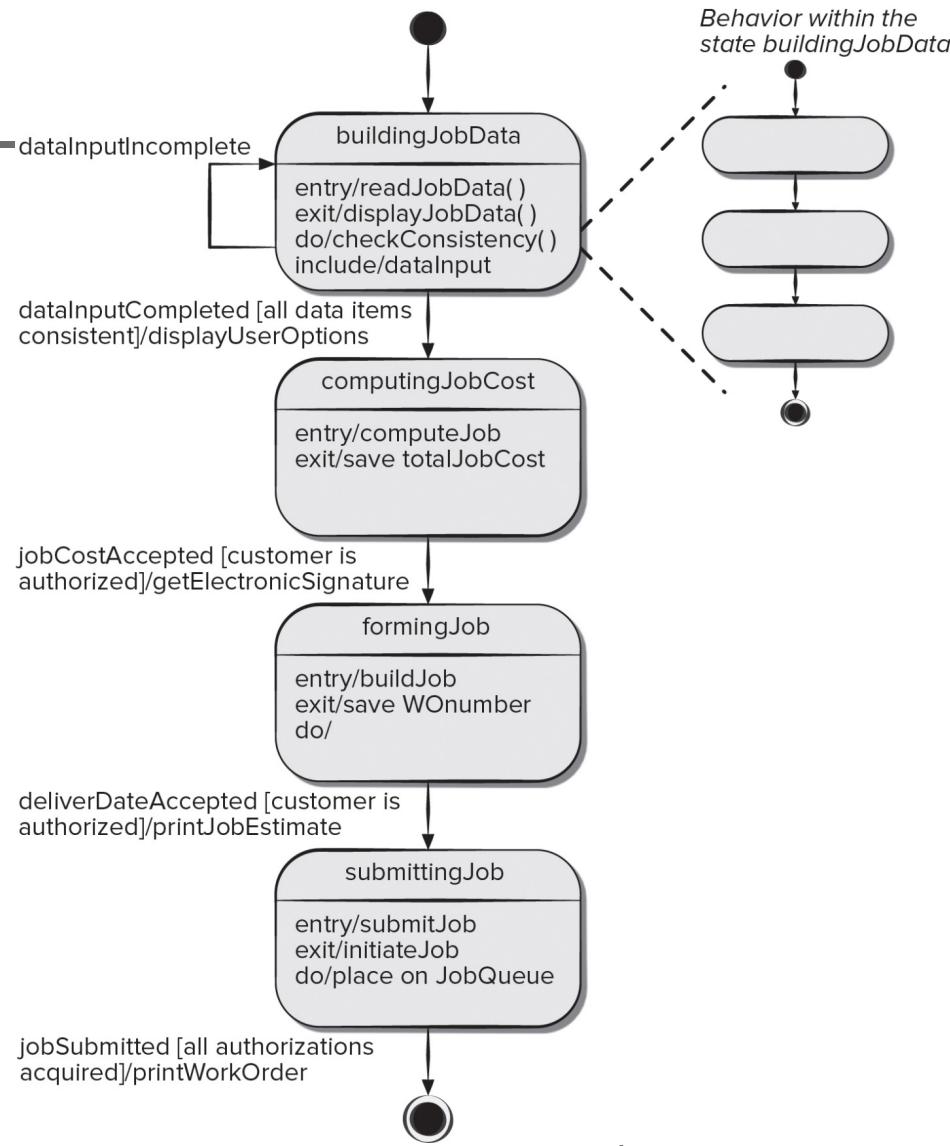
# Describe Processing Flow

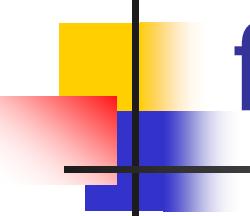
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# Elaborate Behavioral Representations

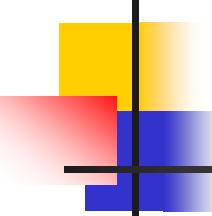
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





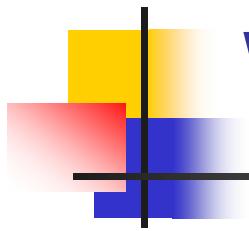
# Component-Level Design for WebApps

- WebApp component is:
  - a well-defined cohesive function that manipulates content or provides computational or data processing for an end-user, or.
  - a cohesive package of content and functionality that provides end-user with some required capability.
- Component-level design for WebApps often incorporates elements of content design and functional design.



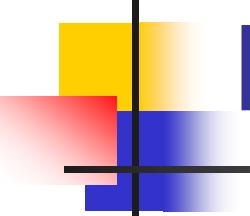
# WebApp Content Design

- Focuses on content objects and the manner in which they may be packaged for presentation to a WebApp end-user
- Consider a Web-based video surveillance capability within SafeHomeAssured.com potential content components can be defined for the video surveillance capability:
  1. the content objects that represent the space layout (the floor plan) with additional icons representing the location of sensors and video cameras;
  2. the collection of thumbnail video captures (each an separate data object), and
  3. the streaming video window for a specific camera.
- Each of these components can be separately named and manipulated as a package.



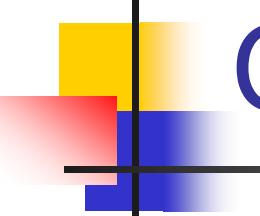
# WebApp Functional Design

- Modern Web applications deliver increasingly sophisticated processing functions that:
  1. perform localized processing to generate content and navigation capability in a dynamic fashion;
  2. provide computation or data processing capability that is appropriate for the WebApp's business domain;
  3. provide sophisticated database query and access, or.
  4. establish data interfaces with external corporate systems.
- To achieve these (and many other) capabilities, you will design and construct WebApp functional components that are identical in form to software components for conventional software.



# Component-Level Design for Mobile Apps

- Thin web-based client.
  - Interface layer only on device.
  - Business and data layers implemented using web or cloud services.
- Rich client.
  - All three layers (interface, business, data) implemented on device.
  - Subject to mobile device limitations.



# Traditional Component-Level Design

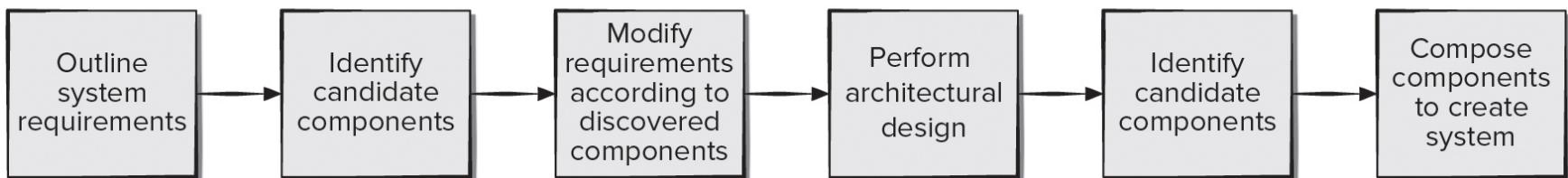
- Design of processing logic is governed by the basic principles of algorithm design and structured programming.
- Design of data structures is defined by the data model developed for the system.
- Design of interfaces is governed by the collaborations that a component must effect.

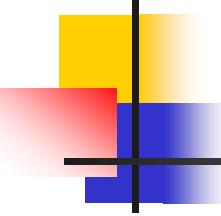
# Component-Based Software Engineering (CBSE)

The software team asks:

- Are commercial off-the-shelf (COTS) components available to implement the requirement?
- Are internally-developed reusable components available to implement the requirement?
- Are the interfaces for available components compatible within the architecture of the system to be built?

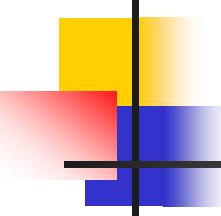
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





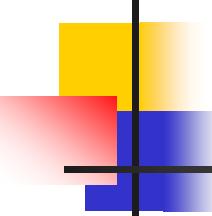
# CBSE Benefits

- **Reduced lead time.** It is faster to build complete applications from a pool of existing components.
- **Greater return on investment (ROI).** Sometimes savings can be realized by purchasing components rather than redeveloping the same functionality in-house.
- **Leveraged costs of developing components.** Reusing components in multiple applications allows the costs to be spread over multiple projects.
- **Enhanced quality.** Components are reused and tested in many different applications.
- **Maintenance of component-based applications.** With careful engineering, it can be relatively easy to replace obsolete components with new or enhanced components.



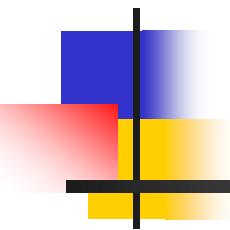
# CBSE Risks

- **Component selection risks.** It is difficult to predict component behavior for black-box components, or there may be poor mapping of user requirements to the component architectural design.
- **Component integration risks.** There is a lack of interoperability standards between components; this often requires the creation of “wrapper code” to interface components.
- **Quality risks.** Unknown design assumptions made for the components makes testing more difficult, and this can affect system safety, performance, and reliability.
- **Security risks.** A system can be used in unintended ways, and system vulnerabilities can be caused by integrating components in untested combinations.
- **System evolution risks.** Updated components may be incompatible with user requirements or contain additional undocumented features.



# Component Refactoring

- Most developers would agree that refactoring components to improve quality is a good practice.
- It is hard to convince management to expend resources fixing components that are working correctly rather than adding new functionality to them.
- Changing software and failing to document the changes can lead to increasing technical debt.
- Reducing this technical debt often involves architectural refactoring, which is generally perceived by developers as both costly and risky.
- Developers can make use of tools to examine change histories to identify the most cost effective refactoring opportunities.

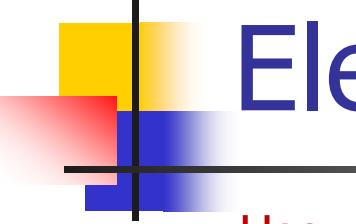


# *COMP 354: Introduction to Software Engineering*

## User Experience Design

Based on Chapter 12 of the textbook

# User Experience Design Elements

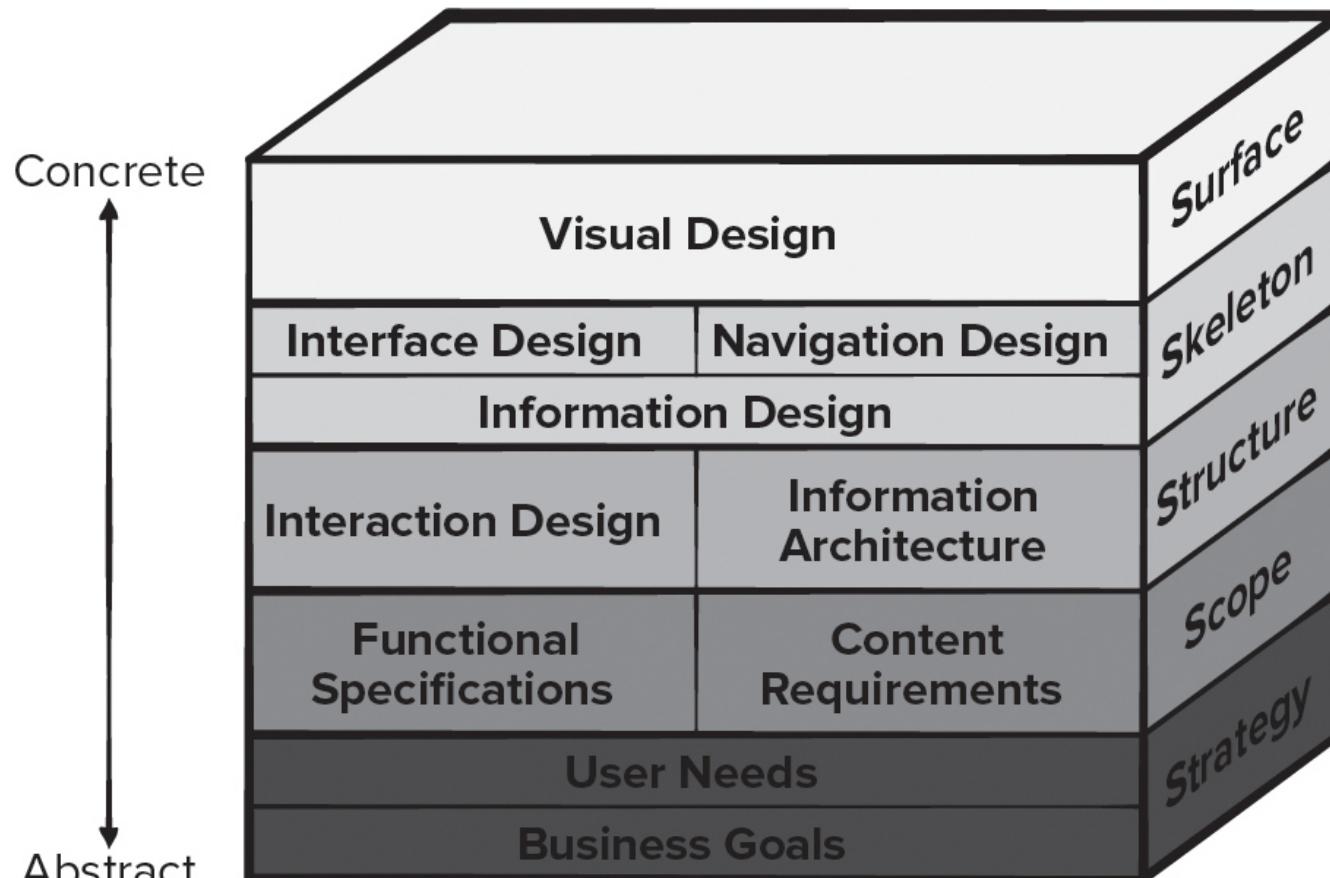


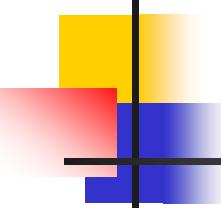
User experience design tries to ensure that no aspect of your software appears in the final product without the explicit decision of stakeholders to include it.

- **Strategy.** Identifies user needs and customer business goals that form the basis for all UX design work.
- **Scope.** Includes both the functional and content requirements needed to realize a feature set consistent with the project strategy.
- **Structure.** Consists of the interaction design [For example, how the system reacts in response to user action] and information architecture.
- **Skeleton.** Comprised of three components: information design, interface design, navigation design.
- **Surface.** Presents visual design or the appearance of the finished project to its users.

# User Experience Design Elements

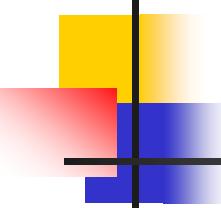
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





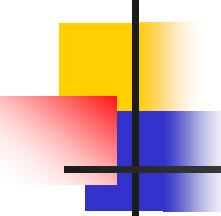
# Information Architecture

- **Information architecture** structures lead organization, labeling, navigation, and searching of content objects.
- **Content architecture** focuses on the manner content objects are structured for presentation and navigation.
- **Software architecture** addresses the manner the application is structured to manage user interaction, effect navigation, and present content.
- Architecture design is conducted in parallel with interface design, aesthetic design, and content design.
- Decisions made during architecture design action will influence work conducted during navigation design.



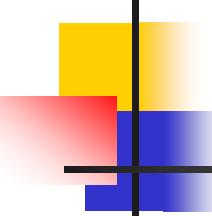
# User Interaction Design

- **Interaction design** focuses on interface between product and user.
- Modes of user input and output include voice input, computer speech generation, touch input, 3D printed output, immersive augmented reality experiences, and sensor tracking of users.
- User interaction should be defined by the stakeholders in the user stories created to describe how users can accomplish their goals using the software product.
- User interaction design should also include a plan for how information should be presented within such a system and how to enable the user to understand that information.
- It is important to recall that the purpose of the user interface is to present just enough information to help the users decide what their next action should be to accomplish their goal and how to perform it.



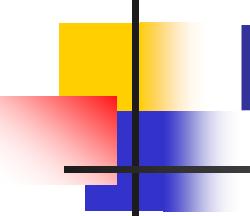
# Usability Engineering

- **Usability engineering** is part of UX design work that defines the specification, design, and testing of the human-computer interaction portion of a software product.
- This software engineering action focuses on devising human-computer interfaces that have high usability.
- If developers focus on making a product easy to learn, easy to use, and easy to remember over time, usability can be measured quantitatively and tested for improvements in usability.
- **Accessibility** is the degree to which people with special needs are provided with a means to perceive, understand, navigate, and interact with computer products.
- Accessibility is another aspect of usability engineering that needs to be considered during design.



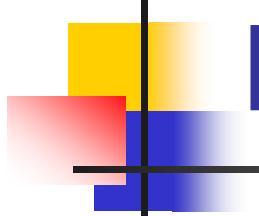
# Visual Design

- **Visual design** (aesthetic design or graphic design) is an artistic endeavor that complements the technical aspects of the user experience design.
- Without it, a software product may be functional, but unappealing.
- With it, a product draws its users into a world that embraces them on an emotional as well as an intellectual level.
- Graphic design considers every aspect of the look and feel of a web or mobile app.
- Not every software engineer has artistic talent. If you fall into this category, hire an experienced graphic designer to help.



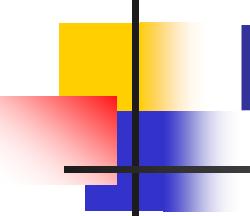
# Golden Rule 1: Place User in Control

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions.
- Provide for flexible interaction.
- Allow user interaction to be interruptible and undoable.
- Streamline interaction as skill levels advance and allow the interaction to be customized.
- Hide technical internals from the casual user.
- Design for direct interaction with screen objects.



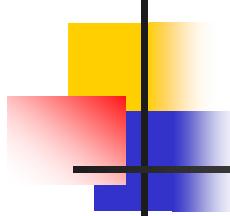
# Golden Rule 2: Reduce User's Memory Load

- Reduce demand on short-term memory.
- Establish meaningful defaults.
- Define shortcuts that are intuitive.
- The visual layout of the interface should be based on a real-world metaphor.
- Disclose information in a progressive fashion.



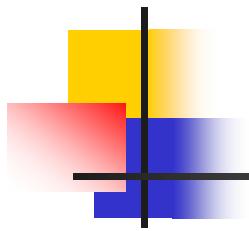
# Golden Rule 3: Make Interface Consistent

- Allow the user to put the current task into a meaningful context.
- Maintain consistency across a family of applications.
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so.



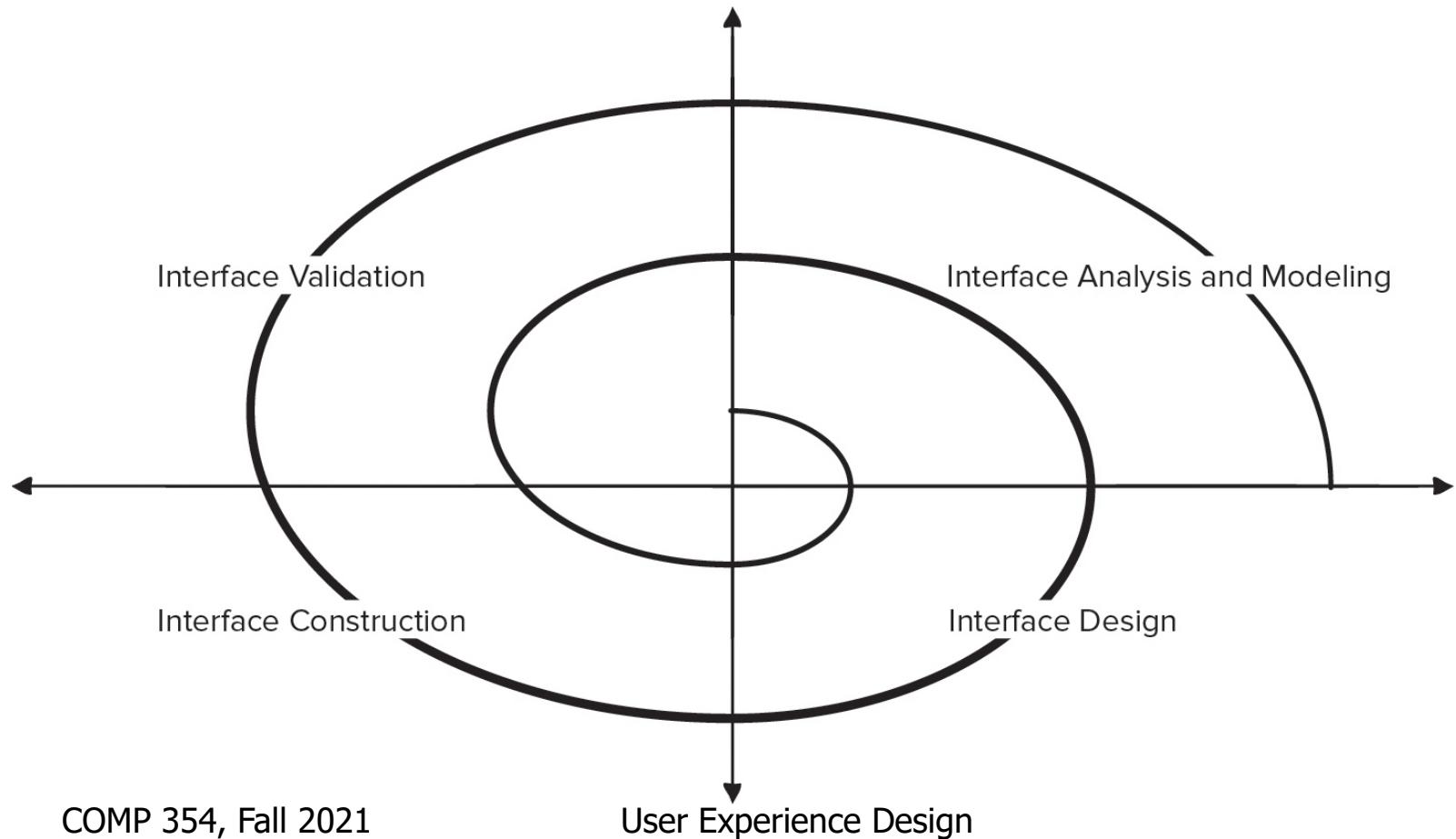
# User Interface Design Models

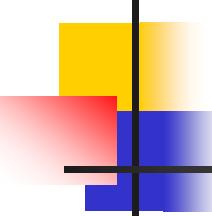
- **User model** — a profile of all end users of the system.
- **Design model** — a design realization of the user model.
- **Mental model** (system perception) — the user's mental image of what the interface is.
- **Implementation model** — the interface "look and feel" coupled with supporting information that describe interface syntax and semantics.
- An interface designer needs to reconcile these models and derive a consistent representation of the interface.



# User Interface Design Process

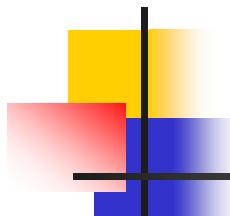
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# User Interface Analysis and Design

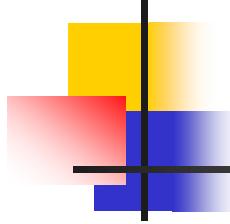
- **Interface analysis** focuses on the profile of the users who will interact with the system.
- **Interface design** defines a set of interface objects and actions that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.
- **Interface construction** normally begins with the creation of a prototype that enables usage scenarios to be evaluated.
- **Interface validation** focuses on:
  1. The ability of the interface to implement every user task correctly.
  2. The degree to which interface is easy to use and easy to learn.
  3. The user's acceptance of the interface as a tool in her work.



# User Experience Analysis

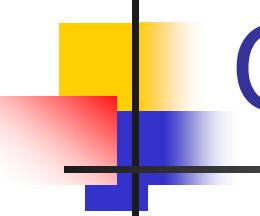
In the case of user experience design, understanding the problem means understanding:

1. the people (end users) who will interact with the system through the interface.
2. the tasks that end users must perform to do their work.
3. the content that is presented as part of the interface.
4. the environment in which these tasks will be conducted.



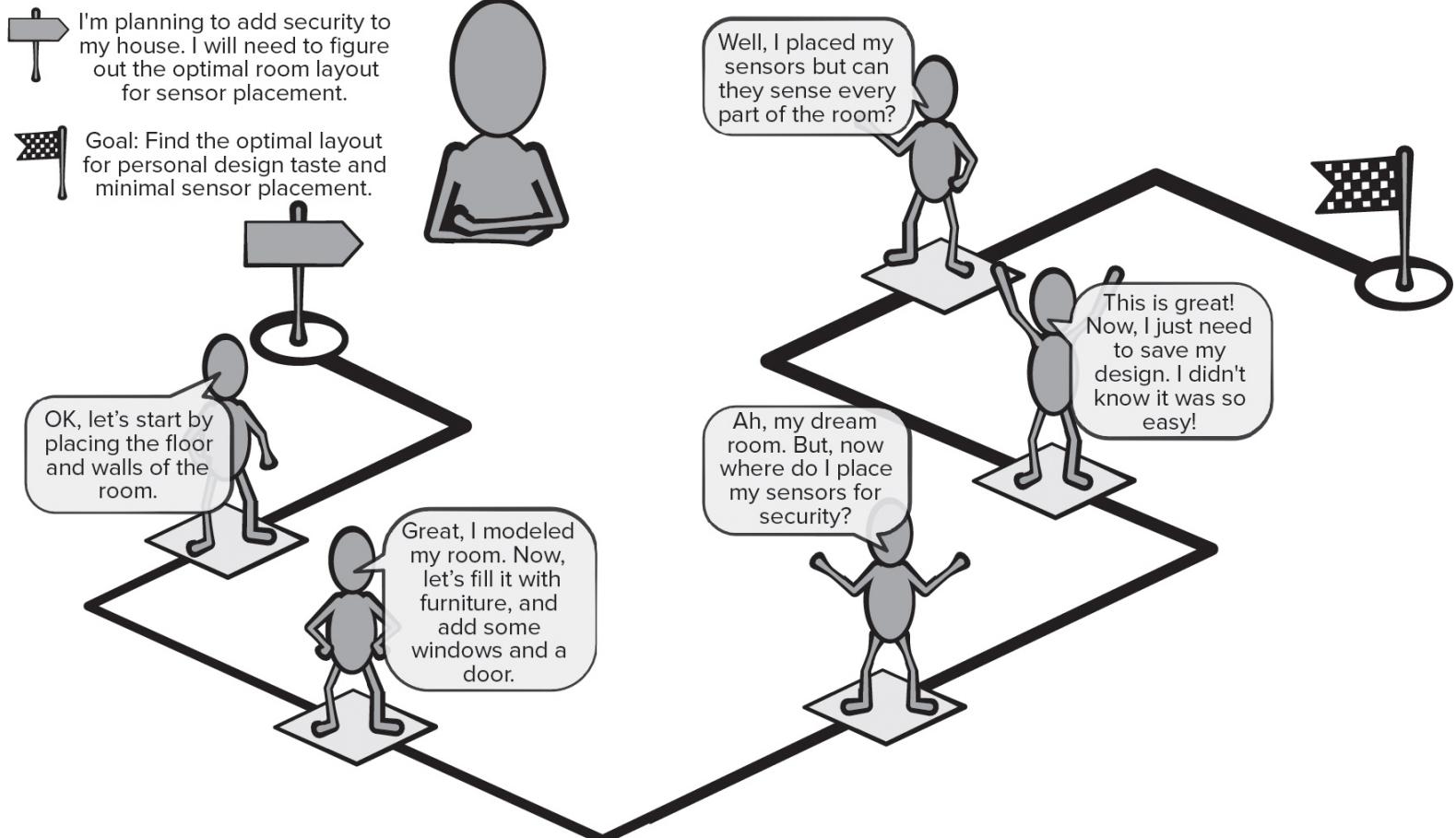
# Using Customer Journey Map

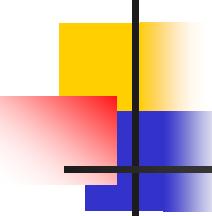
1. Gather stakeholders.
2. Conduct research. Collect all information you can about all the things users may experience as they use the software product and define your customer phases (touchpoints).
3. Build the model. Create a visualization of the touch points.
4. Refine the design. Recruit a designer to make the deliverable visually appealing and ensure touchpoints are identified clearly.
5. Identify gaps. Note any gaps in the customer experience or points of friction or pain (poor transition between phases).
6. Implement your findings. Assign responsible parties to bridge the gaps and resolve pain points found.



# Customer Journey Map

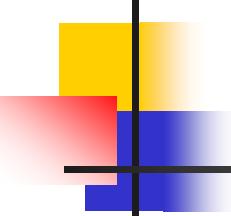
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Creating and Using Personas in UX Design

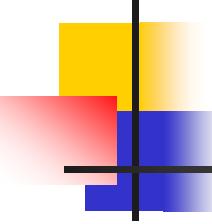
- **Data collection and analysis.** Stakeholders collect information about proposed product users and determine the user group needs.
- **Describe personas.** The developers need to decide how many personas to create and decide which persona will be their focus.
- **Develop scenarios.** Scenarios are user stories about how personas will use the product being developed. They may focus on the touchpoints and obstacles described in the customer journey.
- **Acceptance by stakeholders.** Often this is done by validating the scenarios using a review technique or demonstration called cognitive walkthrough (stakeholders assume the role defined by a persona and work through a scenario using a system prototype).



# Persona Example

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

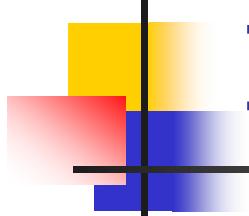
	<p>Works as an elementary teacher in a small midwestern city.</p> <p>Is 38 years old and holds a masters in elementary education.</p> <p>Prefers open-design concepts and shabby chic interior design.</p>	<p>Used to working with computers, but has little experience with virtual reality and tends to get motion sickness.</p> <p>Wants to renovate her house with her design preferences and added security features, but needs help visualizing layouts and lines of sight.</p>
------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------



# Task Analysis

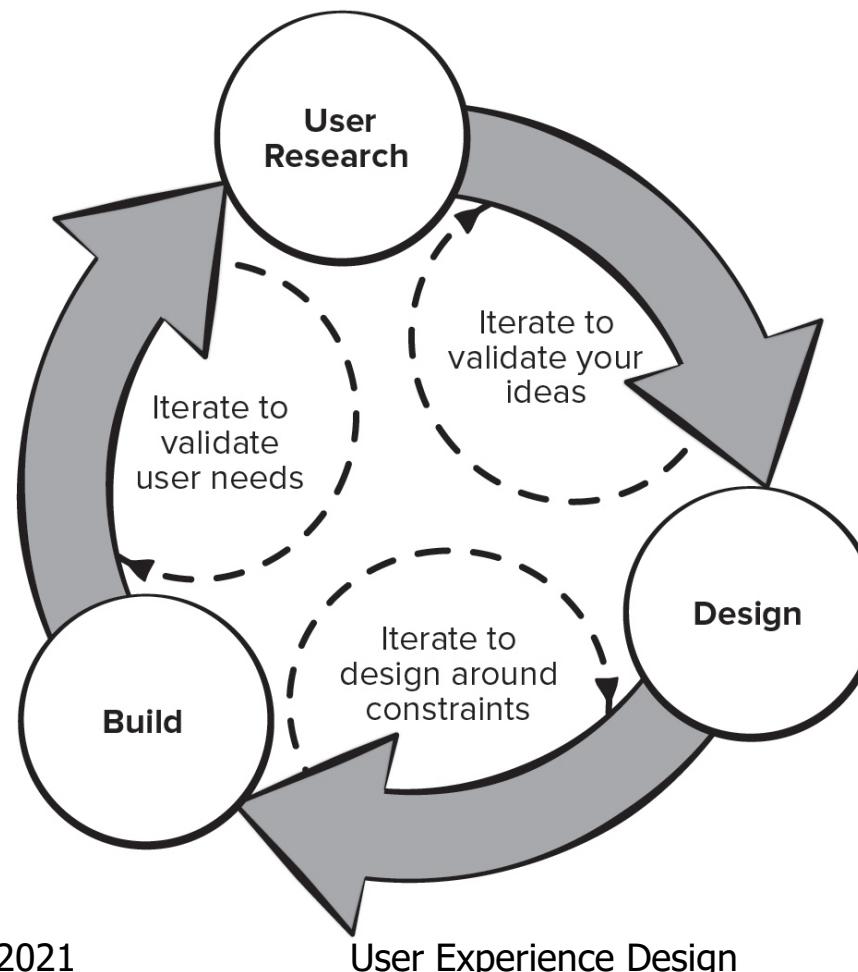
Goal of task (scenario) analysis is to answer the following questions:

- What work will the user perform in specific circumstances?
- What tasks and subtasks will be performed as the user does the work?
- What specific problem domain objects will the user manipulate as work is performed?
- What is the sequence of work tasks—the workflow?
- What is the hierarchy of tasks?



# Iterative UX Design Process

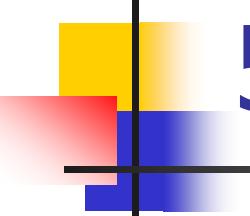
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# 5-Day UX Design Sprint

- **Understand.** User research activities (user needs and business goals) for the software product. This information is posted on whiteboards (For example, customer journey maps, personas, user task workflow) for easy reference throughout the sprint.
- **Sketch.** Individual stakeholders are given the time and space needed to brainstorm solutions to the problems discovered in the understand phase. Paper drawings and notes are easy to generate, easy to modify, and quite inexpensive.
- **Decide.** Each stakeholder presents his solution sketch and the team votes to determine the solutions that should be tackled in the prototyping activities that will follow. If there is not a clear consensus following the voting, the development team may decide to consider assumptions that involve project constraints and resources.

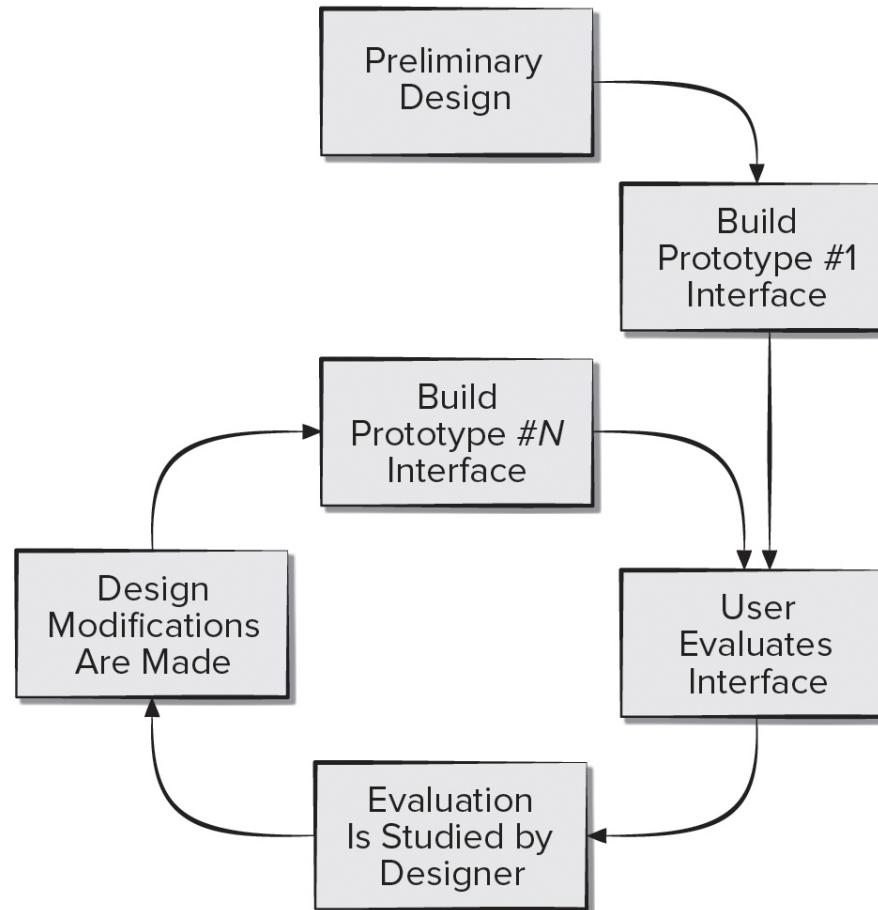


# Google 5-Day UX Design Sprint

- **Prototype.** May be a minimally viable product based on the solution selected from the sketch phase, or it may be based on the portions of the customer journey map or storyboard you want to evaluate with potential users in the validate phase. This means the team should be developing test cases based on the user stories as the prototype is being built.
- **Validate.** Every developer watching users try out the prototype this is the best way to discover major issues with its UX design, which in turn lets you start iterating immediately. This is critical to capturing potential learning opportunities by exposing product decision makers to user feedback in real time.

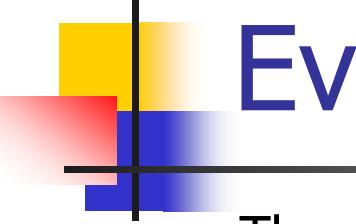
# Interface Design Evaluation Cycle

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



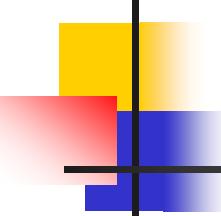
# User Interface Design

## Evaluation Criteria



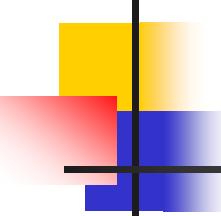
The design model (user stories, storyboard, personas, etc.) of the interface can be evaluated during early design reviews:

1. Length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.
2. Number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.
3. Number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.
4. Interface style, help facilities, and error-handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.



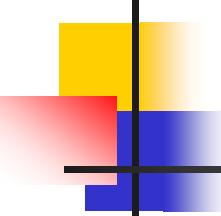
# Usability Guidelines

- **Anticipation.** An application should be designed so that it anticipates the user's next move.
- **Communication.** The interface should communicate the status of any activity initiated by the user.
- **Consistency.** The use of navigation controls, menus, icons, and aesthetics (For example, color, shape, layout) should be consistent throughout.
- **Controlled Autonomy.** The interface should facilitate user movement throughout the application, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- **Efficiency.** The design of the application and its interface should optimize the user's work efficiency.



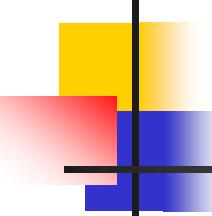
# Usability Guidelines

- **Flexibility.** The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the application in a somewhat random fashion.
- **Focus.** The interface (and the content it presents) should stay focused on the user task(s) at hand.
- **Human Interface Objects.** A vast library of reusable human interface objects has been developed for both Web and mobile apps. Use them.
- **Latency Reduction.** Rather than making the user wait for some internal operation to complete (for example, downloading a complex graphical image), the application should use multitasking in a way that lets the user proceed with work as if the operation has been completed.



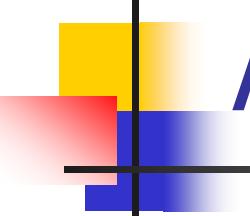
# Usability Guidelines

- **Learnability.** An application interface should be designed to minimize learning time and, once learned, to minimize relearning required when the app is revisited.
- **Metaphors.** An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user.
- **Readability.** All information presented through the interface should be readable by young and old.
- **Track State.** When appropriate, the state of the user interaction should be tracked and stored so that a user can log off and return later to pick up where he left off.
- **Visible Navigation.** A well-designed interface provides the illusion that users are in the same place, with the work brought to them.



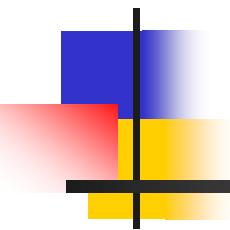
# Accessibility Guidelines

- **Application Accessibility.** Software engineers must ensure that interface design encompasses mechanisms that enable easy for people with special needs.
- **Response Time.** System response time has two important characteristics: length and variability. Aim for consistency to avoid user frustration.
- **Help Facilities.** Modern software should provide online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.
- **Error Handling.** Every error message or warning produced by an interactive system should: use user understandable jargon, provide constructive error recovery advice, identify negative consequences of errors, contain an audible or visual cue, and never blame user for causing the error.



# Accessibility Guidelines

- **Menu and Command Labeling.** The use of window-oriented, point-and-pick interfaces has reduced reliance on typed commands. However, it is important to: ensure every menu option has a command version, make commands easy for users to type, make commands easy to remember, allow for command abbreviation, make sure menu labels are self-explanatory, make sure submenus match style of master menu items, and ensure command conventions work across the family of applications.
- **Internationalization.** Software engineers and their managers invariably underestimate the effort and skills required to create user interfaces that accommodate the needs of different locales and languages.



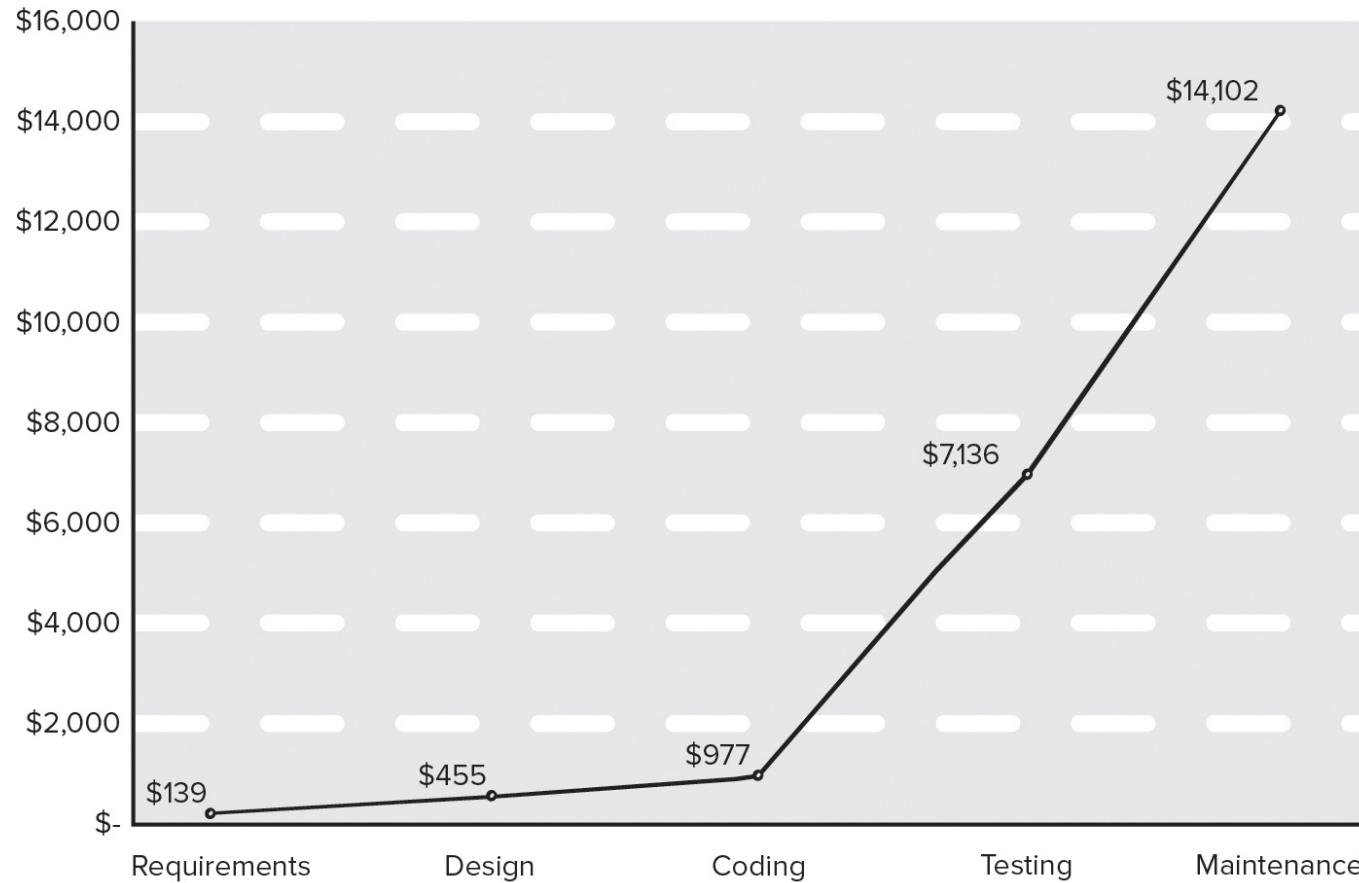
# *COMP 354: Introduction to Software Engineering*

## Software Quality Concepts

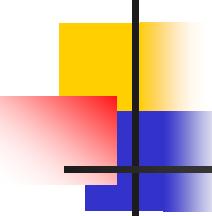
Based on Chapter 15 of the textbook

# Relative Costs to Find and Repair a Defect

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

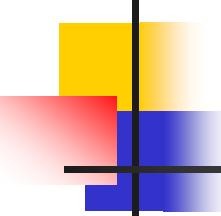


Source: Boehm, Barry and Basili, Victor R., "Software Defect Reduction Top 10 List," IEEE Computer, vol. 34, no. 1, January 2001.



# What is Quality?

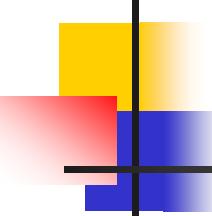
- The *American Heritage Dictionary* defines **quality** as “a characteristic or attribute of something.”
- For software, three kinds of quality may be encountered:
  - **Quality of design** encompasses requirements, specifications, and the design of the system.
  - **Quality of conformance** is an issue focused primarily on implementation.
  - **User satisfaction** = compliant product + good quality + delivery within budget and schedule.



# Quality – Philosophical View

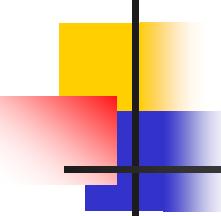
Robert Persig commented on the thing we call quality:

- Quality . . . you know what it is, yet you don't know what it is. But that's self-contradictory.
- But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about.
- But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one knows what it is, then for all practical purposes it doesn't exist at all.
- But for all practical purposes it really does exist. What else are the grades based on? Why else would people pay fortunes for some things and throw others in the trash pile.
- What the hell is Quality? What is it?



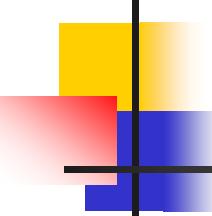
# Quality – Pragmatic Views

- The **transcendental view** argues that quality is something that you immediately recognize but cannot explicitly define.
- The **user view** sees product quality in terms of meeting the end-user's specific goals.
- The **manufacturer's view** defines quality in terms of making sure a product its original specification.
- The **product view** suggests that quality can be tied to inherent characteristics (for example: functions and features) of a product.
- The **value-based view** measures quality based on how much a customer is willing to pay for a product.
- Quality encompasses all of these views and more.



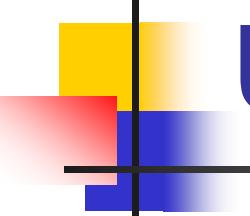
# Software Quality

- Software quality can be defined as:
  - An effective software process applied in a manner that creates a useful product that provides measurable value for those who produce it and those who use it.
- Advantages of providing useful products:
  - Greater software product revenue.
  - Better profitability when an application supports a business process.
  - Improved availability of information that is crucial for the business.



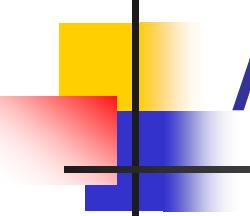
# Software Quality – Effective Process

- An **effective software process** establishes infrastructure that supports building a high-quality software product.
- The management aspects of process create the checks and balances that help avoid project chaos—a key contributor to poor quality.
- Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.
- Umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.



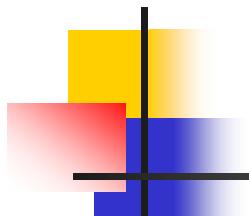
# Software Quality – Useful Product

- A **useful product** delivers the content, functions, and features that the end-user desires.
- But as important, it delivers these assets in a reliable, error free way.
- A useful product always satisfies those requirements that have been explicitly stated by stakeholders.
- A useful product satisfies a set of implicit requirement that are expected of all high-quality software.



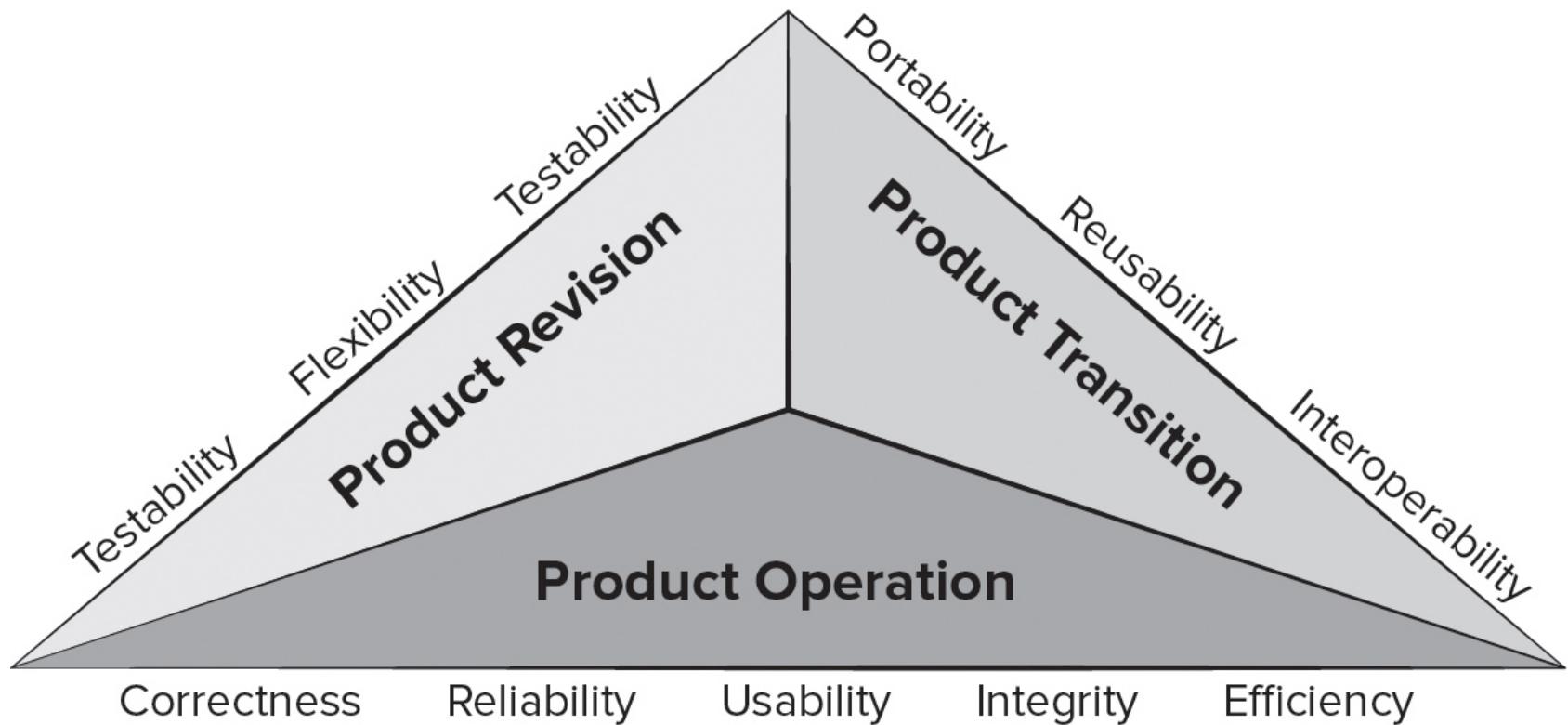
# Software Quality – Adding Value

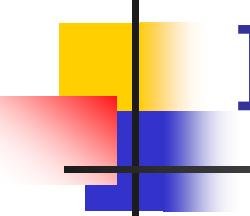
- By **adding value for both the producer and user** of a software product, high quality software provides benefits for the software organization and the end-user community.
- The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.
- The user community gains added value because the application provides a useful capability in a way that expedites some business process.



# McCall's Quality Factors

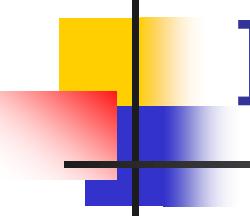
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





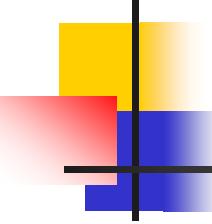
# Quality in Use – ISO25010:2017

- **Effectiveness.** Accuracy and completeness with which users achieve goals.
- **Efficiency.** Resources expended to achieve user goals completely with desired accuracy.
- **Satisfaction.** Usefulness, trust, pleasure, comfort
- **Freedom from risk.** Mitigation of economic, health, safety, and environmental risks.
- **Context coverage.** Completeness, flexibility.



# Product Quality – ISO25010:2017

- **Functional suitability.** Complete, correct, appropriate.
- **Performance efficiency.** Timing, resource use, capacity.
- **Compatibility.** Coexistence, interoperability.
- **Usability.** Appropriateness, learnability, operability, error protection, aesthetics, accessibility.
- **Reliability.** Maturity, availability, fault tolerance.
- **Security.** Confidentiality, integrity, authenticity.
- **Maintainability.** Reusability, modifiability, testability.
- **Portability.** Adaptability, installability, replaceability.

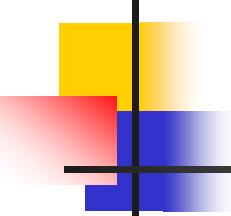


# Qualitative Quality Assessment

- These product quality dimensions and factors presented focus on the complete software product and can be used as a generic indication of the quality of an application.
- You and your team might decide to create a user questionnaire and a set of structured tasks for users to perform for each quality factor you want to assess.
- You might observe the users while they perform these tasks and have them complete the questionnaire when they finish.
- For some quality factors it may be important to test the software in the wild (or in the production environment).

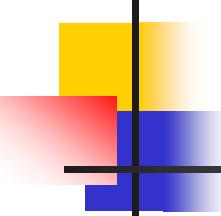
# Quantitative Quality Assessment

- The software engineering community strives to develop precise measures for software quality.
- Internal code attributes can sometimes be described quantitatively using software metrics.
- Any time software metric values computed for a code fragment fall outside the range of acceptable values, it may signal the existence of a quality problem.
- Metrics represent indirect measures; we never really measure quality but rather some manifestation of quality.
- The complicating factor is the accuracy of the relationship between the variable that is measured and the quality of software.



# Software Quality Dilemma

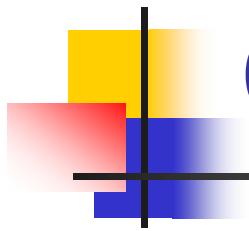
- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If you spend infinite time, extremely large effort, and huge sums of money to build a perfect piece of software, then it's going to take so long to complete and will be so expensive to produce that you'll be out of business.
- You will either miss the market window, or you exhausted all your resources.
- People in industry try to find that magical middle ground where the product is good enough not to be rejected right away, but also not the object of so much perfectionism that it would take too long or cost too much to complete.



# Good Enough Software

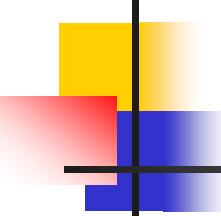
Arguments against “good enough” (buggy software):

- It is true that “good enough” may work in some application domains and for a few major software companies.
- If you work for a small company and you deliver a “good enough” (buggy) product, you risk permanent damage to your company’s reputation and may lose customers.
- If you work in certain application domains (for example: real time embedded software - application software that is integrated with hardware) delivering “good enough” may be considered negligent and open your company to expensive litigation.



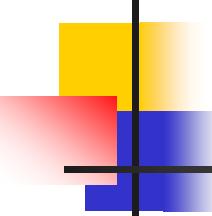
# Cost of Quality

- **Prevention costs** - quality planning, formal technical reviews, test equipment, training.
- **Appraisal costs** - conducting technical reviews, data collection and metrics evaluation, testing and debugging.
- **Internal failure costs** – rework, repair, failure mode analysis.
- **External failure costs** - complaint resolution, product return and replacement, help line support, warranty work



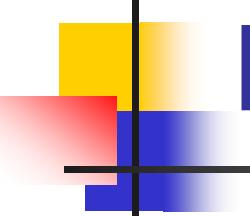
# Negligence and Liability

- A governmental or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based “system”.
- The system might support a major corporate function (for example: pension management) or some governmental function (for example: healthcare administration or homeland security).
- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.
- The system is late, fails to deliver desired features and functions, error-prone, and does not get customer acceptance.
- Litigation ensues.



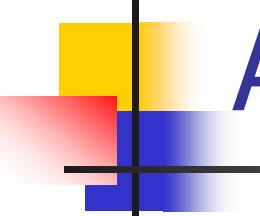
# Quality, Risk, and Security

- Low quality software increases risks for both developers and end-users.
- When systems are delivered late, fail to deliver functionality, and does not meet customer expectations litigation ensues.
- Low quality software is easier to hack and can increase the security risks for the application once deployed.
- A secure system cannot be built without focusing on quality (security, reliability, dependability) during design.
- Low quality software is liable to contain architectural flaws as well as implementation problems (bugs).



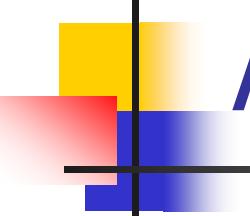
# Impact of Management Decisions

- **Estimation decisions** – irrational delivery date estimates cause teams to take short-cuts that can lead to reduced product quality.
- **Scheduling decisions** – failing to pay attention to task dependencies when creating the project schedule.
- **Risk-oriented decisions** – reacting to each crisis as it arises rather than building in mechanisms to monitor risks may result in products having reduced quality.



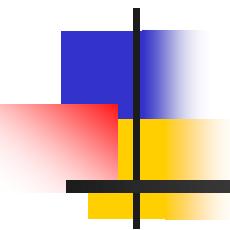
# Achieving Software Quality

- Software quality is the result of good project management and solid engineering practice.
- To build high quality software you must understand the problem to be solved and be capable of creating a quality design that conforms to the problem requirements.
- Project management – project plan includes explicit techniques for quality and change management.
  - Use estimation to verify that delivery dates are achievable.
  - Schedule is understood and team avoids taking shortcuts.
  - Risk planning is conducted so problems do not breed chaos, software quality will be affected in a positive way.



# Achieving Software Quality

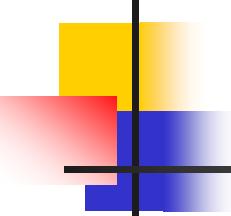
- Project plan should include explicit techniques for quality and change management.
- **Quality control** - series of inspections, reviews, and tests used to ensure conformance of a work product to its specifications.
- **Quality assurance** - consists of the auditing and reporting procedures used to provide management with data needed to make proactive decisions.
- Defect prediction is an important part of identifying software components that may have quality concerns.
- Machine learning and statistical models may help identify relationships between metrics and defect components.



# *COMP 354: Introduction to Software Engineering*

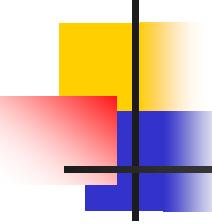
## Software Reviews

Based on Chapter 16 of the textbook



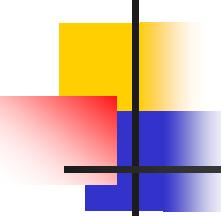
# Reviews

- What are they?
  - A meeting conducted by technical people.
  - A technical assessment of a work product created during the software engineering process.
  - A software quality assurance mechanism.
  - A training ground.
- What they are not!
  - A project summary or progress assessment.
  - A meeting intended solely to impart information.
  - A mechanism for political or personal reprisal!



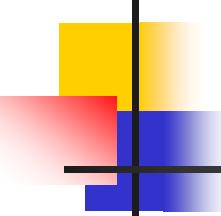
# Cost Impact of Software Defects

- **Error**—a quality problem found before the software is released to end users.
- **Defect**—a quality problem found only after the software has been released to end-users.
- We make this distinction because errors and defects have very different economic, business, psychological, and human impact.
- Design activities introduce 50 to 65% of all software defects.
- Review activities have been shown to be 75% effective in uncovering design flaws.
- The sooner you find a defect the cheaper it is to fix it.



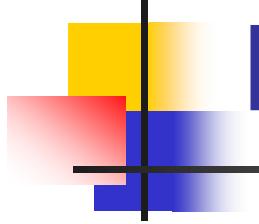
# Defect Amplification and Removal

- Defect amplification is a term used to describe how a defect introduced early in the software engineering workflow (for example: during requirement modeling) and undetected, can and often will be amplified into multiple errors during design and more errors in construction.
- Defect propagation is a term used to describe the impact an undiscovered defect has on future development activities or product behavior.
- Technical debt is the term used to describe the costs incurred by failing to find and fix defects early or failing to update documentation following software changes.



# Review Metrics

- **Preparation effort,  $E_p$**  — the effort (in person-hours) required to review a work product prior to the actual review meeting.
- **Assessment effort,  $E_a$**  — the effort (in person-hours) that is expending during the actual review.
- **Rework effort,  $E_r$**  — the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review.
- **Work product size,  $WPS$**  — a measure of the size of the work product that has been reviewed (for example: the number of UML models, or the number of document pages, or the number of lines of code).
- **Minor errors found,  $Err_{minor}$**  — the number of errors found that can be categorized as minor (requiring less than some pre-specified effort to correct).
- **Major errors found,  $Err_{major}$**  — the number of errors found that can be categorized as major (requiring more than some pre-specified effort to correct).



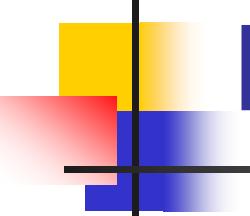
# Review Metrics

- Total errors found,  $Err_{tot}$ , represents the sum of the errors found:

$$Err_{tot} = Err_{minor} + Err_{major}$$

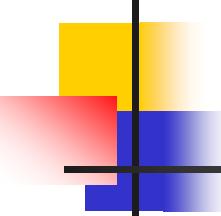
- Error density represents the errors found per unit of work product reviewed:

$$Error\ density = Err_{tot} \div WPS$$



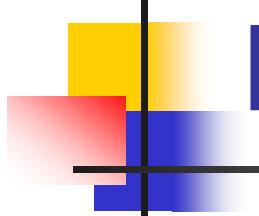
# Metrics Example

- The average defect density for a requirements model is 0.68 errors per page, and a new requirement model is 40 pages long.
- A rough estimate suggests that your software team will find about 27 errors during the review of the document.
- If you find only 9 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough.



# Metrics Example

- The effort required to correct a minor model error (immediately after the review) was found to require 4 person-hours.
- The effort required for a major requirement error was found to be 18 person-hours.
- Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors.
- Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 6 person-hours.



# Metrics Example

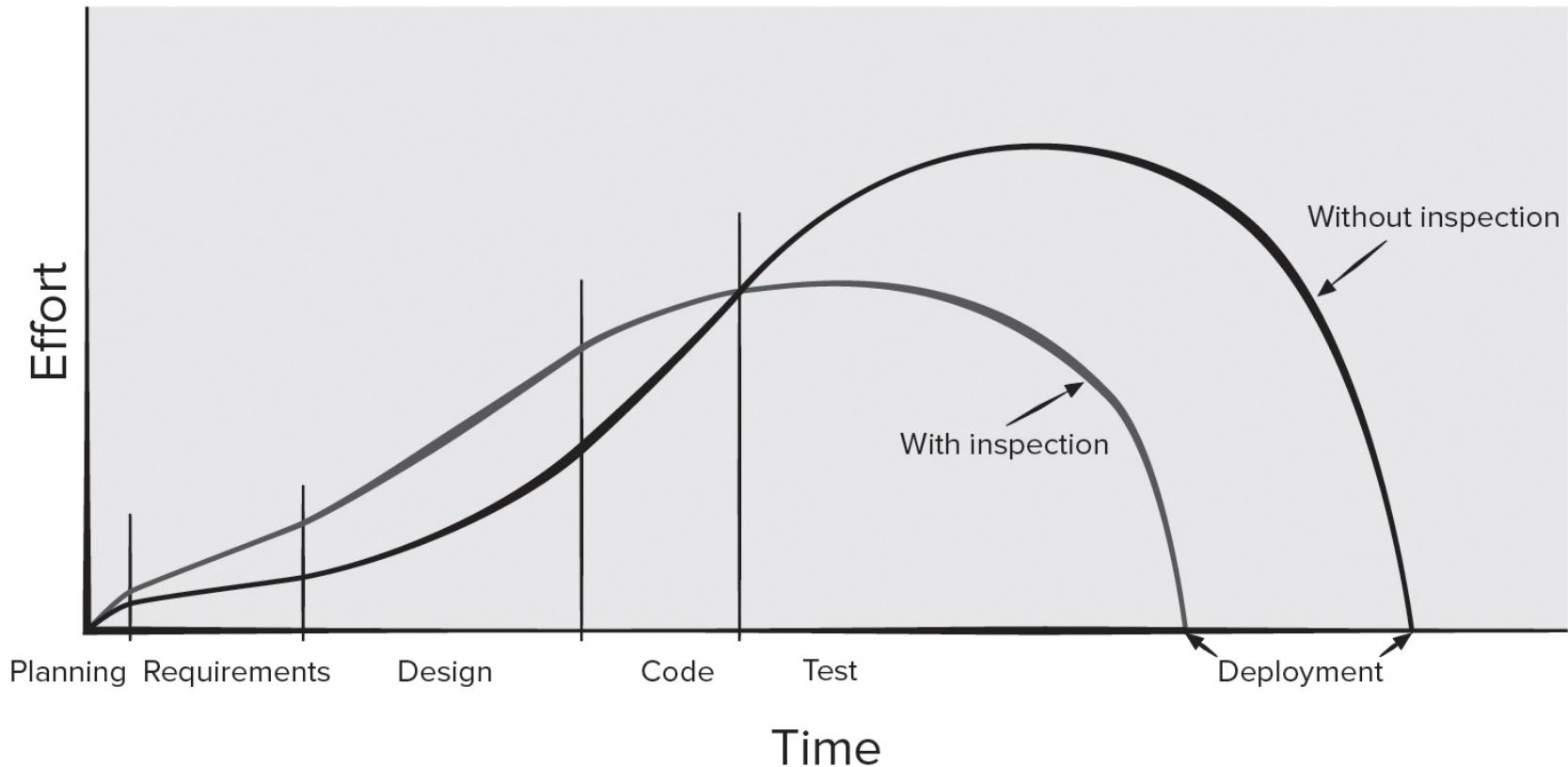
- Requirements related errors uncovered during testing require an average of 45 person-hours to find and correct. Using the averages noted, we get:

$$\begin{aligned}\text{Effort saved per error} &= E_{testing} - E_{reviews} \\ &= 45 - 6 = 39 \text{ person-hours/error}\end{aligned}$$

- Since 22 errors were found during the review of the requirements model, a saving of about 858 person-hours of testing effort would be achieved. And that's just for requirements-related errors.

# Effort Expended With and Without Reviews

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

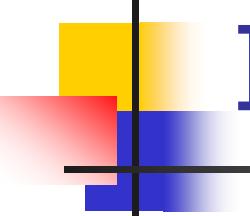


Source: Fagan, Michael E., "Advances in Software Inspections," IEEE Transactions on Software Engineering, vol. SE-12, no. 7, July 1986, 744–751.

# Reference Model for Technical Reviews

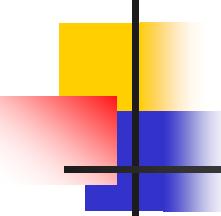
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Informal Reviews

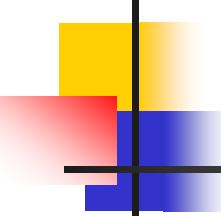
- The benefit is immediate discovery of errors and better work product quality.
- Informal reviews include:
  - A simple desk check of a software engineering work product with a colleague.
  - A casual meeting (involving more than 2 people) for the purpose of reviewing a work product, or
  - The review-oriented aspects of pair programming which encourages continuous review as work is created.



# Formal Technical Reviews

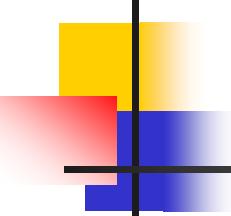
The objectives of an FTR (walkthrough or inspection) are:

- To uncover errors in function, logic, or implementation for any representation of the software.
- To verify that the software under review meets its requirements.
- To ensure that the software has been represented according to predefined standards.
- To achieve software that is developed in a uniform manner.
- To make projects more manageable.



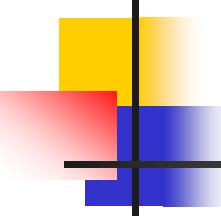
# Review Meeting

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- Focus is on a work product (for example: a portion of a requirements model, a detailed component design, source code for a component).



# Review Players

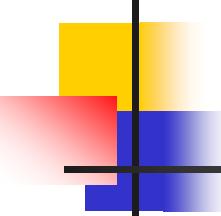
- **Producer**—the individual who has developed the work product.
- **Review leader**—evaluates the product for readiness, generates copies of product materials, and distributes them to two or three reviewers for advance preparation and facilitates the meeting discussion.
- **Reviewer(s)**—expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- **Recorder**—reviewer who records (in writing) all important issues raised during the review.



# Review Outcome

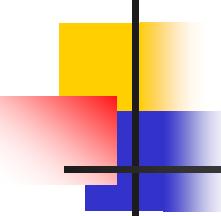
At the end of the review, all attendees of the FTR must decide whether to:

- Accept the product without further modification.
- Reject the product due to severe errors (once corrected, another review must be performed).
- Accept the product provisionally (minor errors have been encountered and must be corrected, but no additional review will be required).



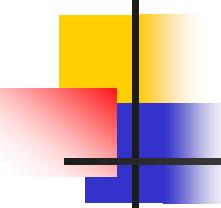
# Review Reporting and Record Keeping

- During the FTR, the recorder records all issues raised and summarizes these in a review issues list to serve as an action list for the producer.
- A formal technical review summary report is created that answers three questions:
  - What was reviewed?
  - Who reviewed it?
  - What were the findings and conclusions?
- You should establish a follow-up procedure to ensure that items on the issues list have been properly corrected.



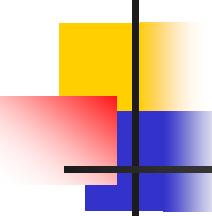
# Review Guidelines

- Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and rebuttal.
- Enunciate problem areas, but don't try to solve every problem noted.
- Take written notes.
- Limit the number of participants and insist upon advance preparation.
- Develop a checklist for each product that is likely to be reviewed.
- Allocate resources and schedule time for FTRs.
- Conduct meaningful training for all reviewers.
- Review your early reviews.



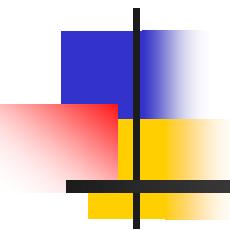
# Postmortem Evaluations

- A **postmortem evaluation (PME)** is a mechanism to determine what went right and what went wrong with the software engineering process and practices applied to a specific project.
- APME is attended by members of the software team and stakeholders who examine the entire software project, focusing on excellences (achievements and positive experiences) and challenges (problems and negative experiences).
- The intent is to extract lessons learned from the challenges and excellences and to suggest improvements to process and practice moving forward.



# Agile Reviews

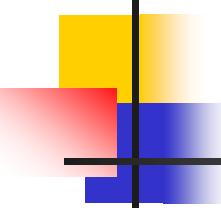
- During the sprint planning meeting, user stories are reviewed and ordered according to priority.
- The daily Scrum meeting is an informal way to ensure that team members are all working on the same priorities and try to catch any defects that may cause the sprint to fail.
- The sprint review meeting is often conducted using guidelines like the formal technical review discussed in this chapter.
- The sprint retrospective meeting is really a postmortem meeting in that the development team is trying to capture its lessons learned.



# *COMP 354: Introduction to Software Engineering*

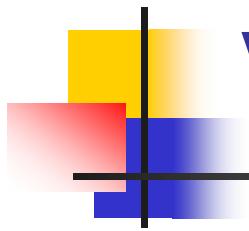
## Software Component Testing

Based on Chapter 19 of the textbook



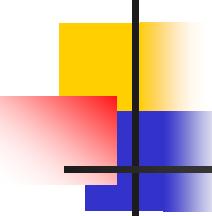
# Strategic Approach to Testing

- You should conduct effective technical reviews as this can eliminate many errors before testing begins.
- Testing begins at the component level and works "outward" toward the integration of the entire system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.



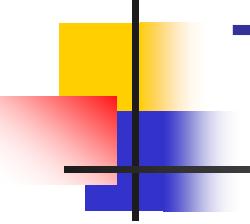
# Verification and Validation

- **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.
  - Verification: Are we building the product right?
- **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
  - Validation: "Are we building the right product?"



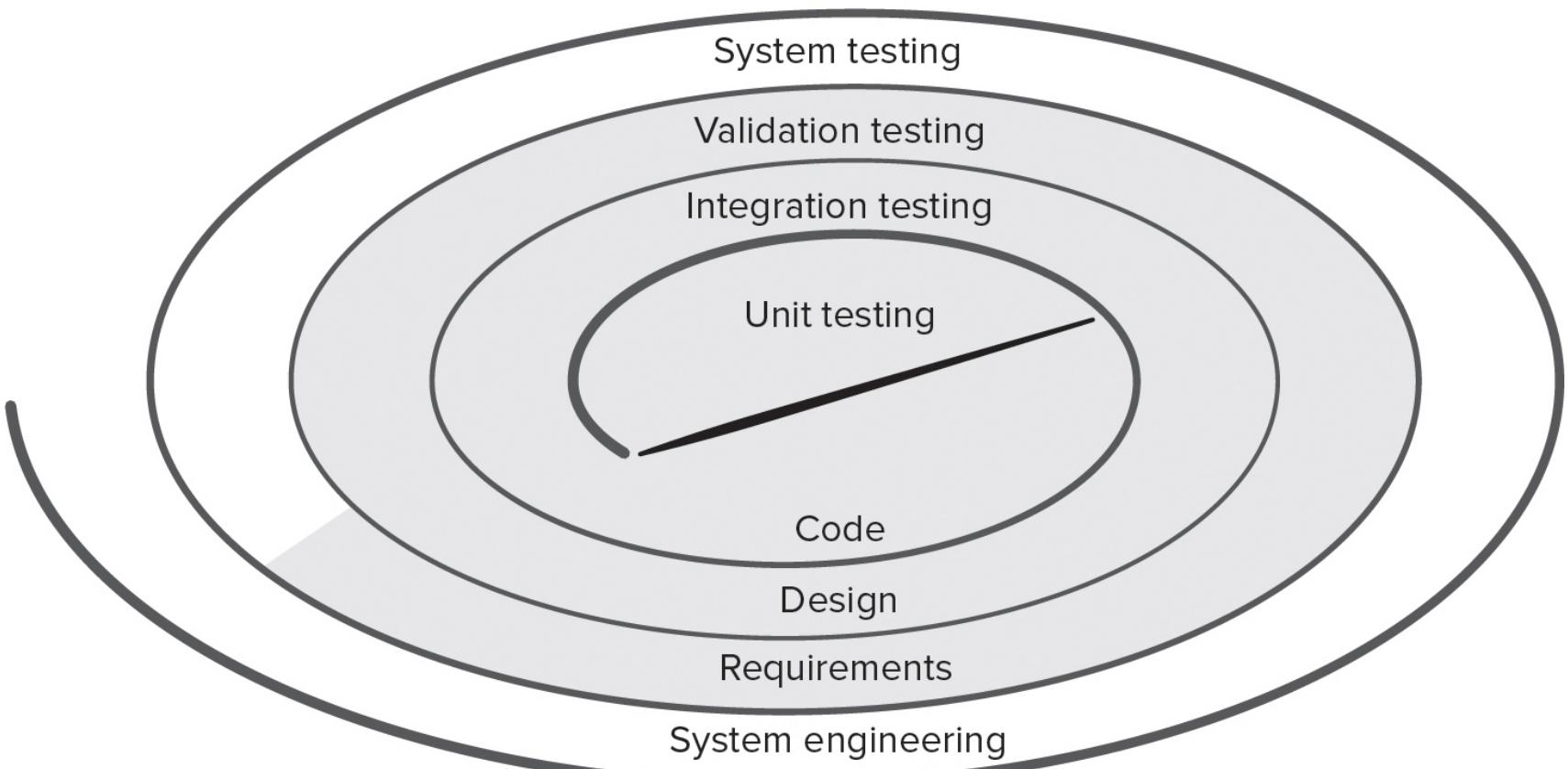
# Organizing for Testing

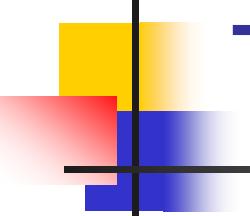
- Software developers are always responsible for testing individual program components and ensuring that each performs its designed function or behavior.
- Only after the software architecture is complete does an independent test group become involved.
- The role of an **independent test group (ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built.
- ITG personnel are paid to find errors.
- Developers and ITG work closely throughout a software project to ensure that thorough tests will be conducted.



# Testing Strategy

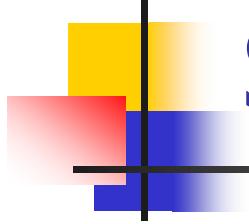
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





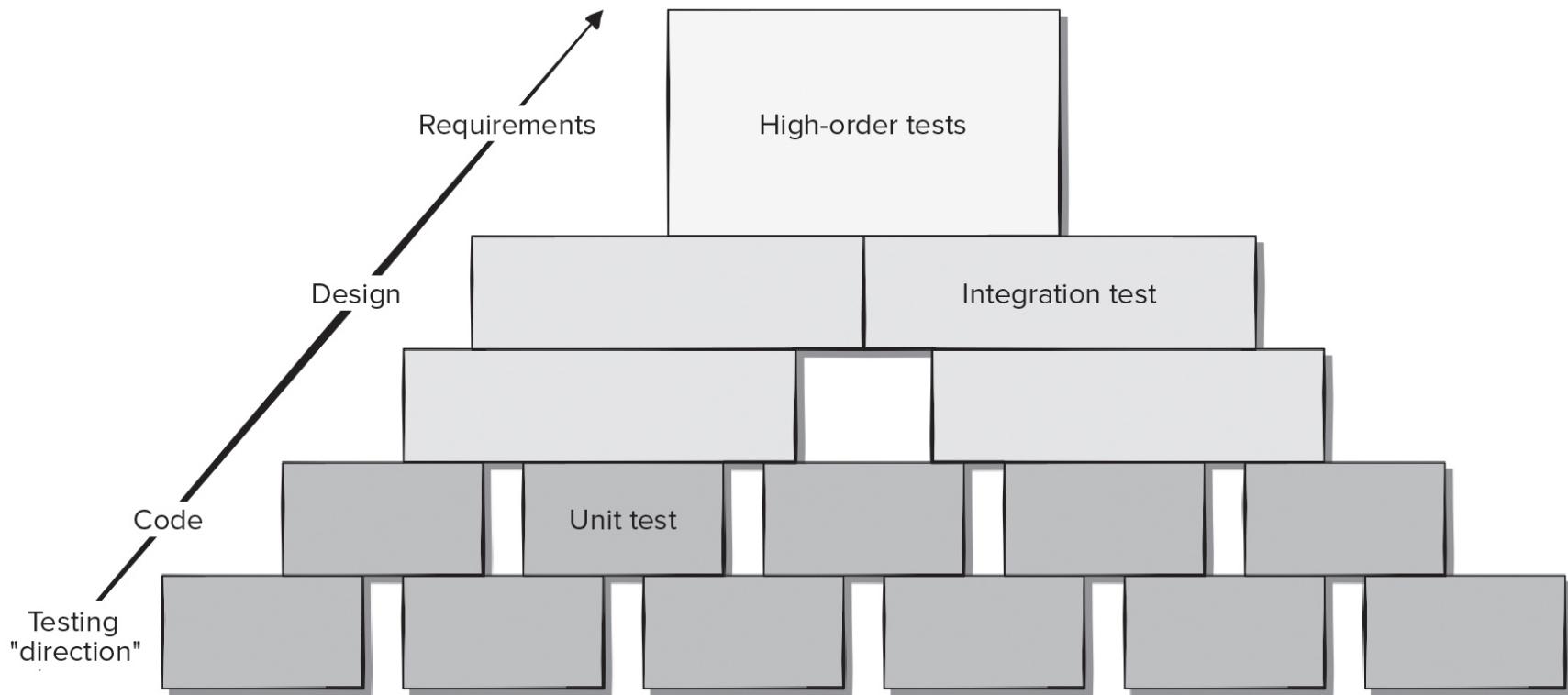
# Testing the Big Picture

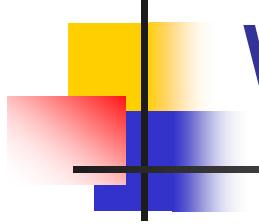
- Unit testing begins at the center of the spiral and concentrates on each unit (for example, component, class, or content object) as they are implemented in source code.
- Testing progresses to integration testing, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral.
- Validation testing, is where requirements established as part of requirements modeling are validated against the software that has been constructed.
- In system testing, the software and other system elements are tested as a whole.



# Software Testing Steps

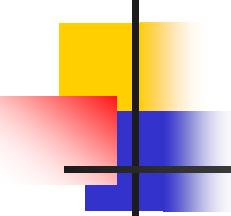
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





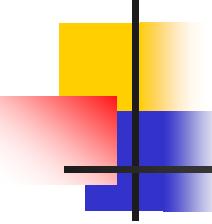
# When is Testing Done?





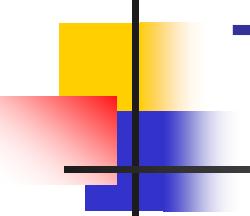
# Criteria for Done

- You're never done testing; the burden simply shifts from the software engineer to the end user. (Wrong).
- You're done testing when you run out of time or you run out of money. (Wrong).
- The **statistical quality assurance** approach suggests executing tests derived from a statistical sample of all possible program executions by all targeted users.
- By collecting metrics during software testing and making use of existing statistical models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"



# Test Planning

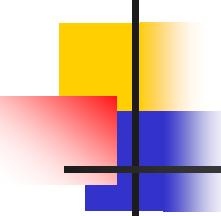
1. Specify product requirements in a quantifiable manner long before testing commences.
2. State testing objectives explicitly.
3. Understand the users of the software and develop a profile for each user category.
4. Develop a testing plan that emphasizes “rapid cycle testing.”
5. Build “robust” software that is designed to test itself.
6. Use effective technical reviews as a filter prior to testing.
7. Conduct technical reviews to assess the test strategy and test cases themselves.
8. Develop a continuous improvement approach for the testing process.



# Test Recordkeeping

Test cases can be recorded in Google Docs spreadsheet:

- Briefly describes the test case.
- Contains a pointer to the requirement being tested.
- Contains expected output from the test case data or the criteria for success.
- Indicate whether the test was passed or failed.
- Dates the test case was run.
- Should have room for comments about why a test may have failed (aids in debugging).

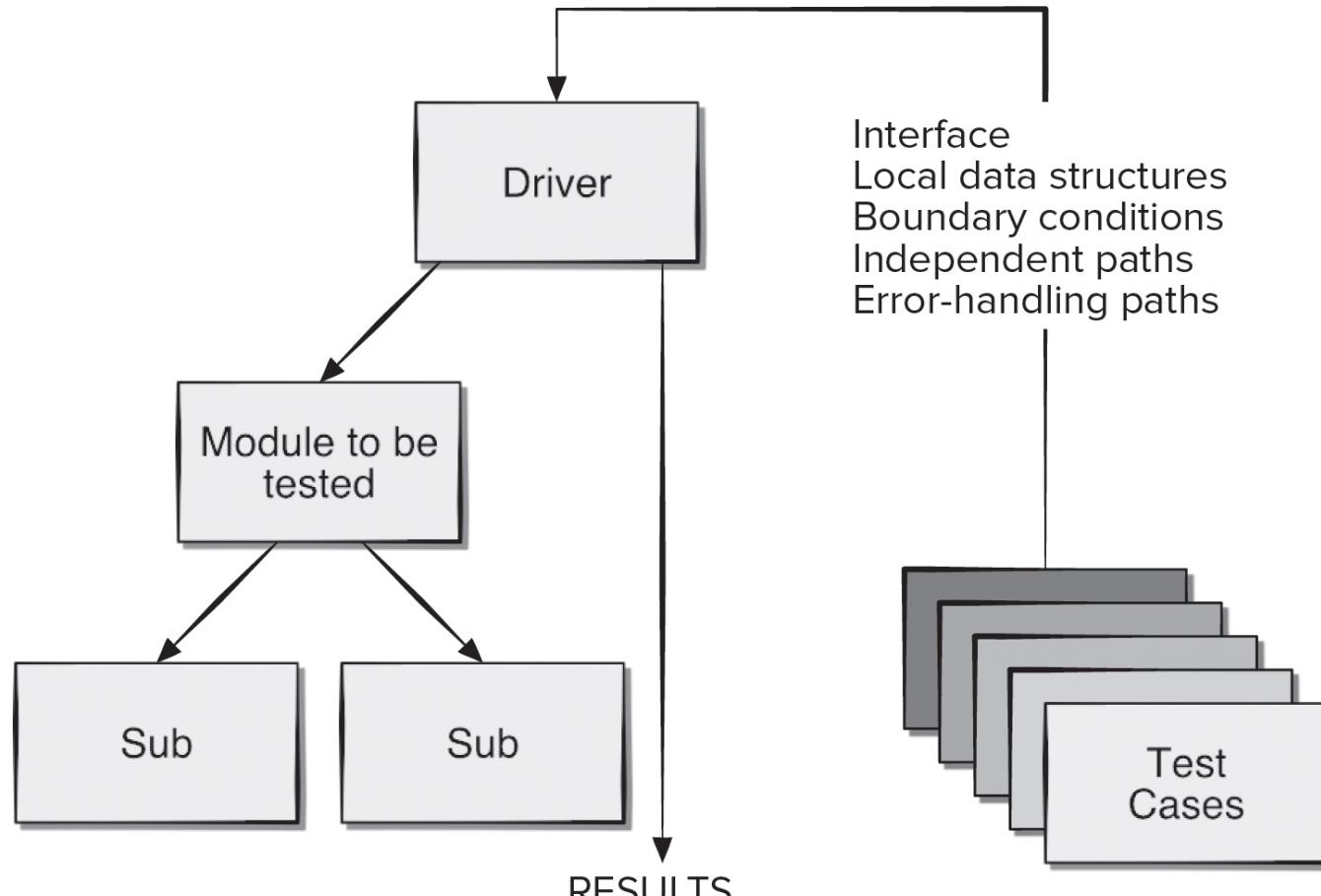


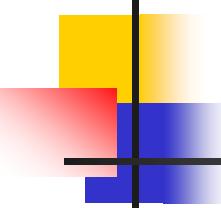
# Role of Scaffolding

- Components are not stand-alone program some type of **scaffolding** is required to create a testing framework.
- As part of this framework, driver and/or stub software must often be developed for each unit test.
- A **driver** is nothing more than a “main program” that accepts test-case data, passes such data to the component (to be tested), and prints relevant results.
- **Stubs** (dummy subprogram) serve to replace modules invoked by the component to be tested.
- A stub uses the module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

# Unit Test Environment

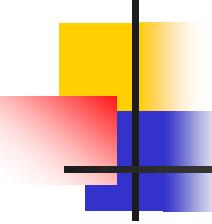
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





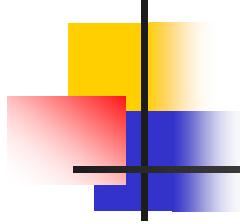
# Cost Effective Testing

- Exhaustive testing requires every possible combination and ordering of input values be processed by the test component.
- The return on exhaustive testing is often not worth the effort, since testing alone cannot be used to prove a component is correctly implemented.
- Testers should work smarter and allocate their testing resources on modules crucial to the success of the project or those that are suspected to be error-prone as the focus of their unit testing.



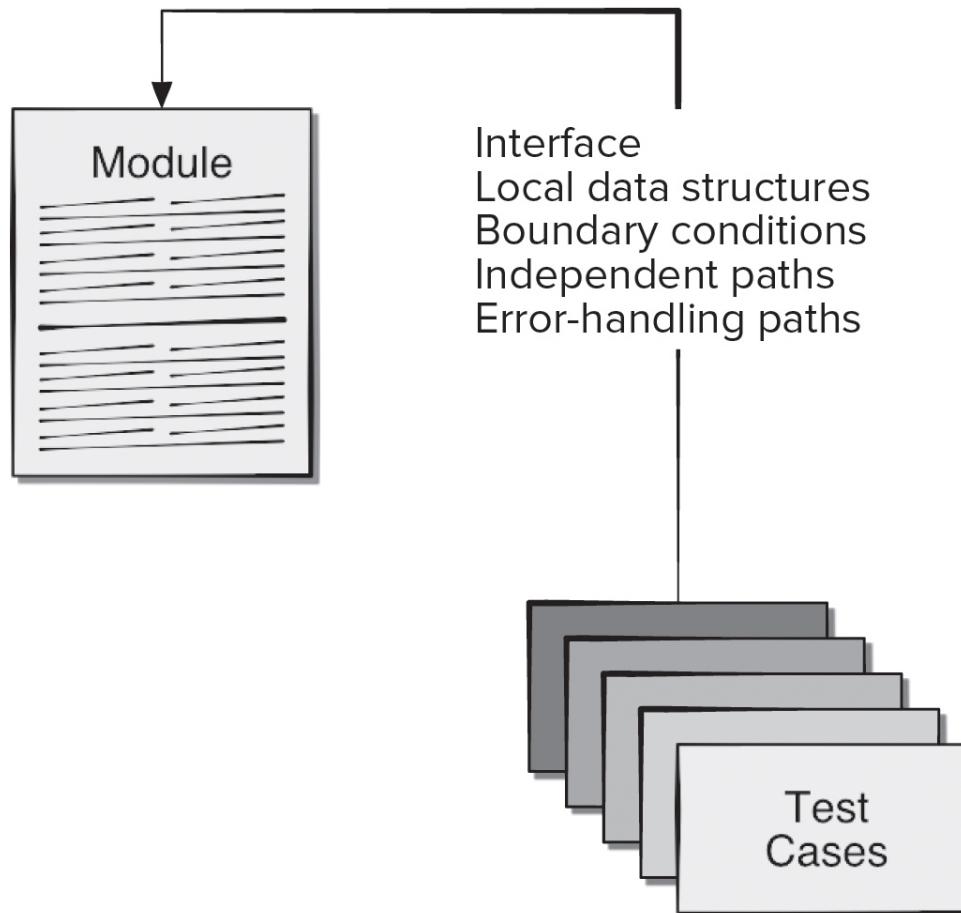
# Test Case Design

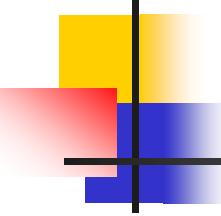
- Design unit test cases before you develop code for a component to ensure that code that will pass the tests.
- Test cases are designed to cover the following areas:
  - The module interface is tested to ensure that information properly flows into and out of the program unit.
  - Local data structures are examined to ensure that stored data maintains its integrity during execution.
  - Independent paths through control structures are exercised to ensure all statements are executed at least once.
  - Boundary conditions are tested to ensure module operates properly at boundaries established to limit or restrict processing.
  - All error-handling paths are tested.



# Module Tests

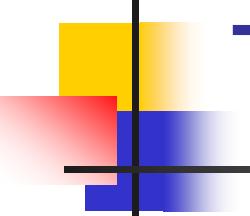
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





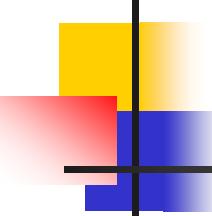
# Error Handling

- A good design anticipates error conditions and establishes error-handling paths which must be tested.
- Among the potential errors that should be tested when error handling is evaluated are:
  - Error description is unintelligible.
  - Error noted does not correspond to error encountered.
  - Error condition causes system intervention prior to error handling,
  - Exception-condition processing is incorrect.
  - Error description does not provide enough information to assist in the location of the cause of the error.



# Traceability

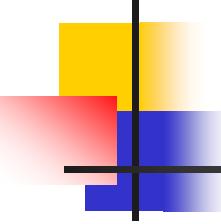
- To ensure that the testing process is auditable, each test case needs to be traceable back to specific functional or nonfunctional requirements or anti-requirements.
- Often nonfunctional requirements need to be traceable to specific business or architectural requirements.
- Many test process failures can be traced to missing traceability paths, inconsistent test data, or incomplete test coverage.
- Regression testing requires retesting selected components that may be affected by changes made to other collaborating software components.



# White Box Testing

Using white-box testing methods, you can derive test cases that:

- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Exercise internal data structures to ensure their validity.



# Basis Path Testing

Determine the number of independent paths in the program by computing Cyclomatic Complexity:

- The number of regions of the flow graph corresponds to the cyclomatic complexity.
- Cyclomatic complexity  $K(G)$  for a flow graph  $G$  is defined as

$$K(G) = E - N + 2$$

$E$  is the number of flow graph edges

$N$  is the number of nodes.

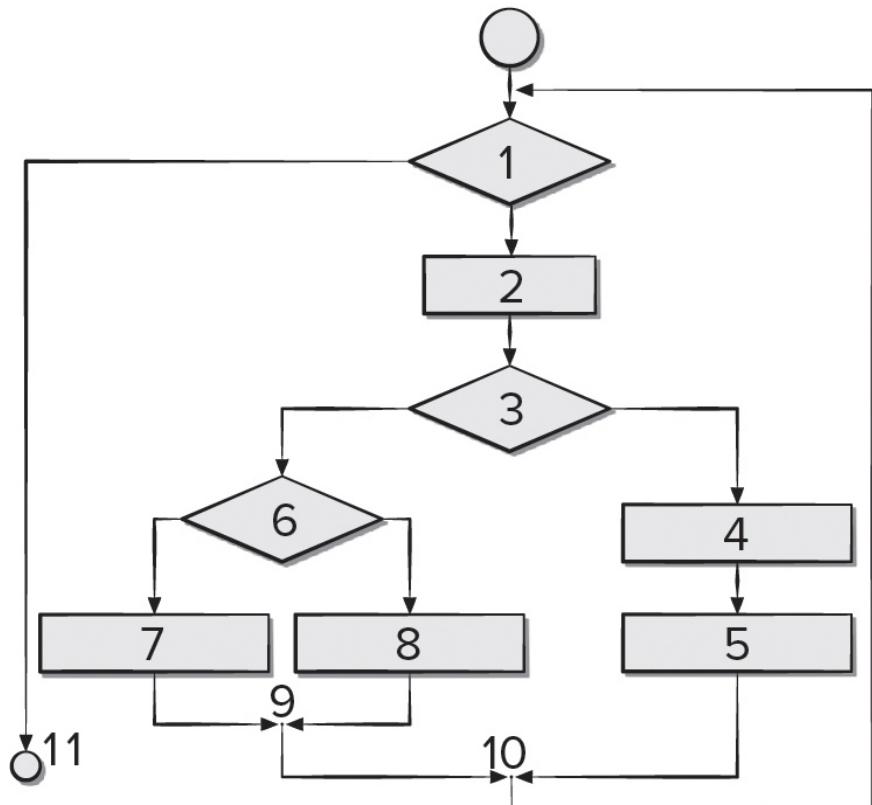
- Cyclomatic complexity  $K(G)$  for a flow graph  $G$  is also defined as

$$K(G) = P + 1$$

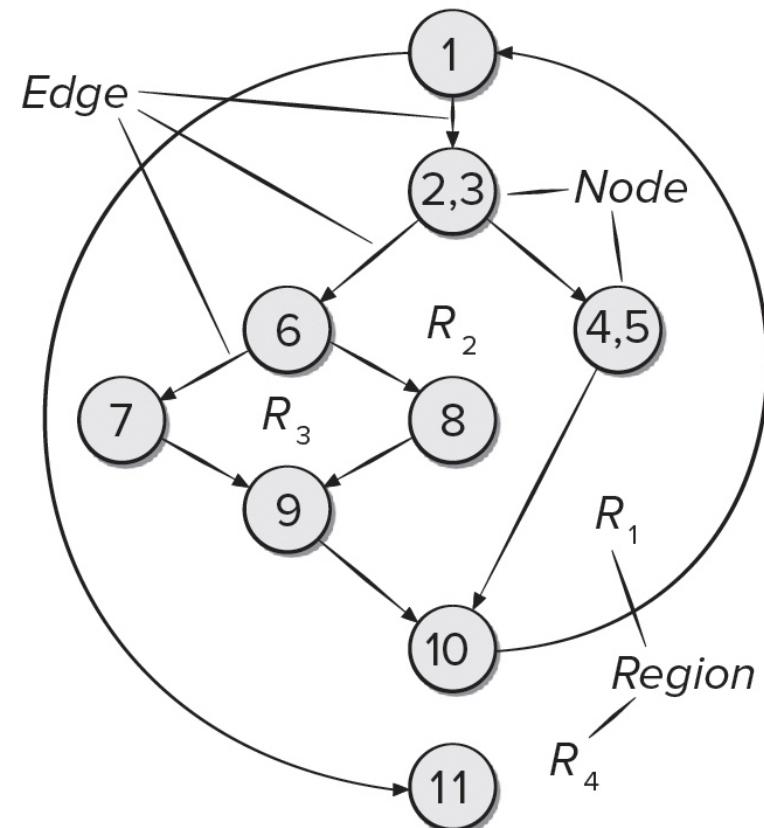
$P$  is number of predicate nodes contained in the flow graph  $G$ .

# Flowchart (a) and Flow Graph (b)

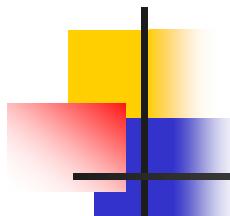
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



(a)

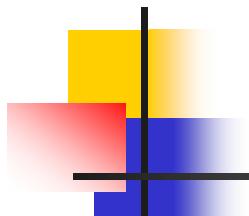


(b)



# Basis Path Testing

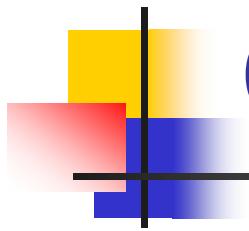
- Cyclomatic Complexity of the flow graph is 4
  - The flow graph has four regions.
  - $V(G) = 11 \text{ edges} - 9 \text{ nodes} + 2 = 4.$
  - $V(G) = 3 \text{ predicate nodes} + 1 = 4.$
- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition (we need 4 independent paths to test)
  - Path 1: 1-11
  - Path 2: 1-2-3-4-5-10-1-11
  - Path 3: 1-2-3-6-8-9-10-1-11
  - Path 4: 1-2-3-6-7-9-10-1-11



# Basis Path Testing

## Designing Test Cases

- Using the design or code as a foundation, draw a corresponding flow graph.
- Determine the cyclomatic complexity of the resultant flow graph.
- Determine a basis set of linearly independent paths.
- Prepare test cases that will force execution of each path in the basis set.

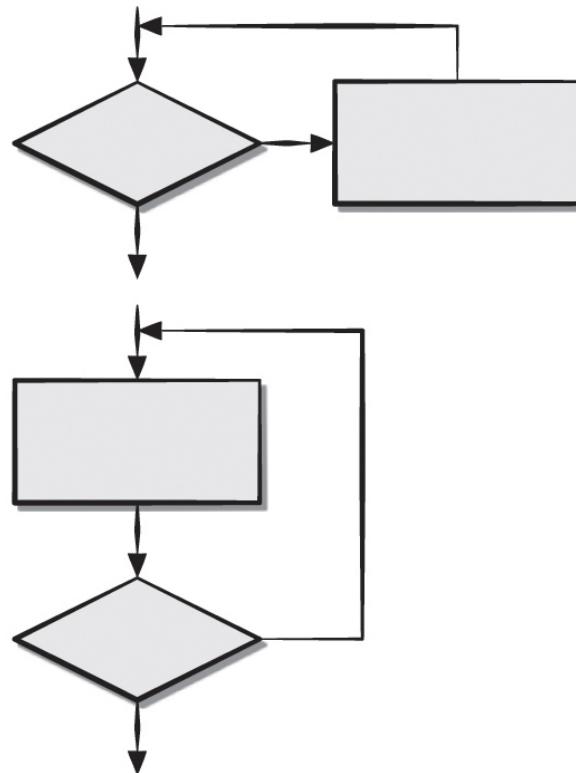


# Control Structure Testing

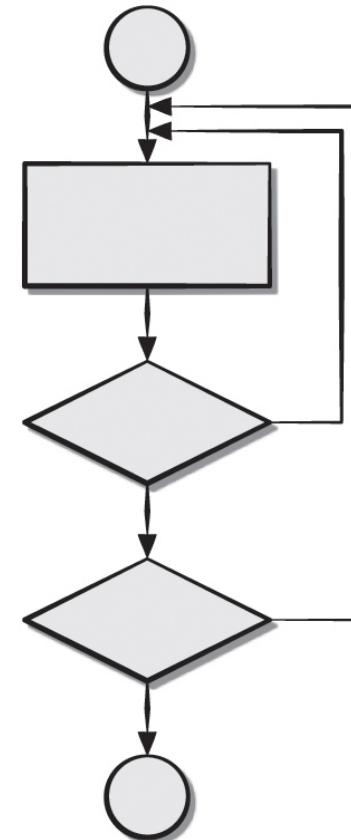
- **Condition testing** is a test-case design method that exercises the logical conditions contained in a program module.
- **Data flow testing** selects test paths of a program according to the locations of definitions and uses of variables in the program.
- **Loop testing** is a white-box testing technique that focuses exclusively on the validity of loop constructs.

# Classes of Loops

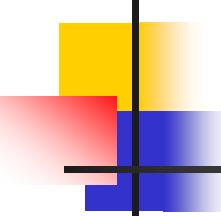
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



Simple loops



Nested loops



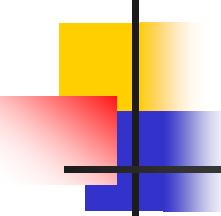
# Loop Testing

Test cases for simple loops:

- Skip the loop entirely.
- Only one pass through the loop.
- Two passes through the loop.
- $m$  passes through the loop where  $m < n$ .
- $n - 1, n, n + 1$  passes through the loop.

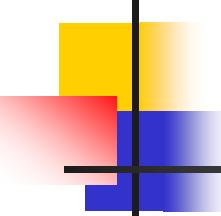
Test cases for nested loops:

- Start at the innermost loop. Set all other loops to minimum values.
- Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (for example, loop counter) values.
- Add other tests for out-of-range or excluded values.
- Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.
- Continue until all loops have been tested.



# Black Box Testing

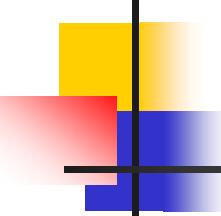
- Black-box (functional) testing attempts to find errors in the following categories:
  - Incorrect or missing functions.
  - Interface errors.
  - Errors in data structures or external database access.
  - Behavior or performance errors.
  - Initialization and termination errors.
- Unlike white-box testing, which is performed early in the testing process, black-box testing tends to be applied during later stages of testing.



# Black Box Testing

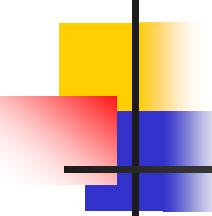
Black-box test cases are created to answer questions like:

- How is functional validity tested?
- How are system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?



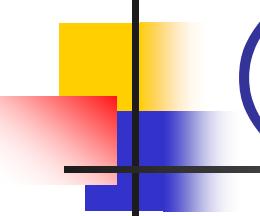
# Black Box – Interface Testing

- **Interface testing** is used to check that a program component accepts information passed to it in the proper order and data types and returns information in proper order and data format.
- Components are not stand-alone programs testing interfaces requires the use of stubs and drivers.
- Stubs and drivers sometimes incorporate test cases to be passed to the component or accessed by the component.
- Debugging code may need to be inserted inside the component to check that data passed was received correctly.



# Black Box – Boundary Value Analysis (BVA)

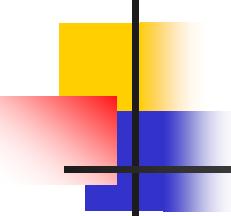
- **Boundary value analysis** leads to a selection of test cases that exercise bounding values.
- Guidelines for BVA:
  - If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
  - If an input condition specifies a number of values, test cases should be developed that exercise the min and max numbers as well as values just above and below min and max.
  - Apply guidelines 1 and 2 to output conditions.
  - If internal program data structures have prescribed boundaries (for example, array with max index of 100) be certain to design a test case to exercise the data structure at its boundary.



# Object-Oriented Testing (OOT)

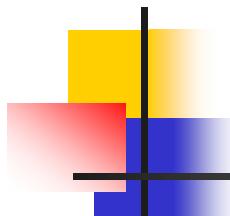
To adequately test OO systems, three things must be done:

- The definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models.
- The strategy for unit and integration testing must change significantly.
- The design of test cases must account for the unique characteristics of OO software.



# OOT – Class Testing

- Class testing for object-oriented (OO) software is the equivalent of unit testing for conventional software.
- Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface.
- Class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.
- Valid sequences of operations and their permutations are used to test that class behaviors - equivalence partitioning can reduce number sequences needed.

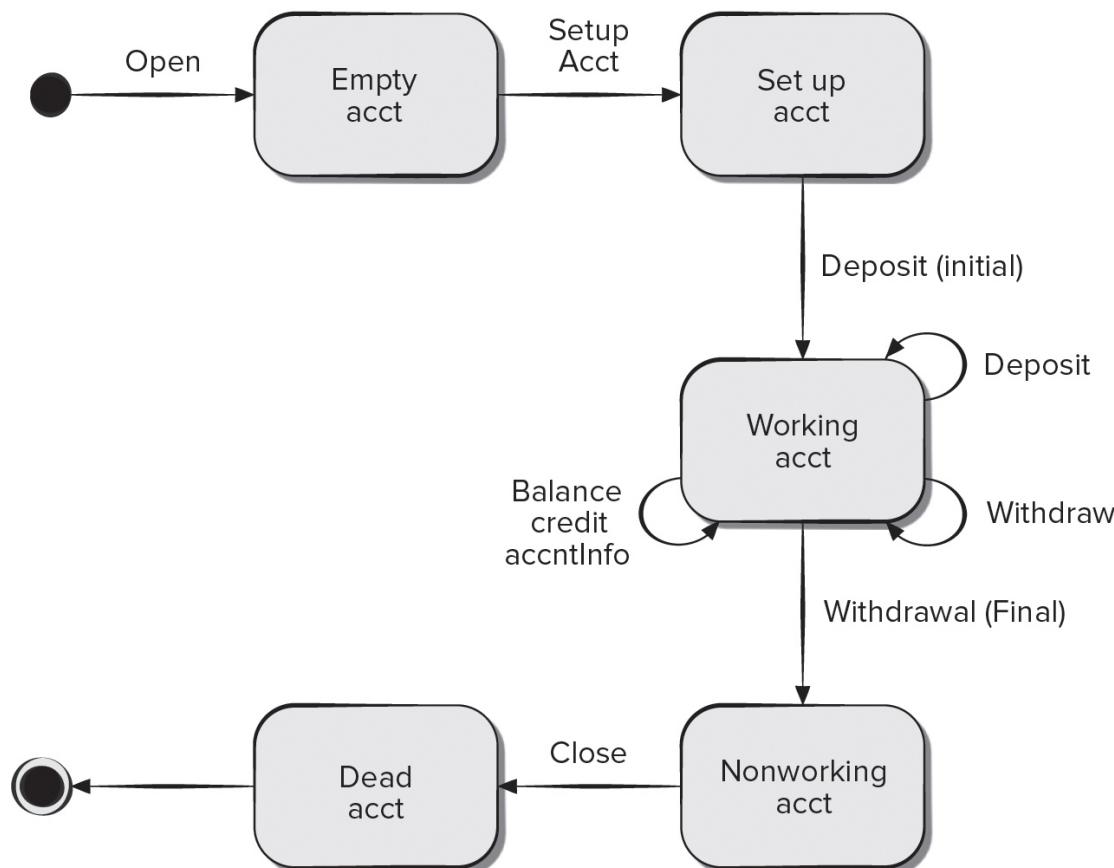


# OOT– Behavior Testing

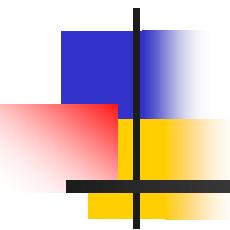
- A state diagram can be used to help derive a sequence of tests that will exercise dynamic behavior of the class.
- Tests to be designed should achieve full coverage by using operation sequences cause transitions through all allowable states.
- When class behavior results in a collaboration with several classes, multiple state diagrams can be used to track system behavioral flow.
- A state model can be traversed in a breadth-first manner by having test case exercise a single transition and when a new transition is to be tested only previously tested transitions are used.

# State Diagram for Account Class

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



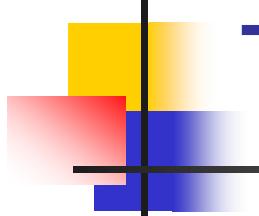
Source: Kirani, Shekhar and Tsai, W. T., "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, University of Minnesota, December 1994, 79.



# *COMP 354: Introduction to Software Engineering*

## Software Integration Testing

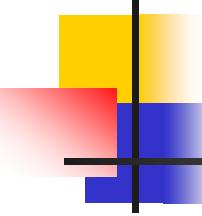
Based on Chapter 20 of the textbook



# Testing Fundamentals

Attributes of a good test:

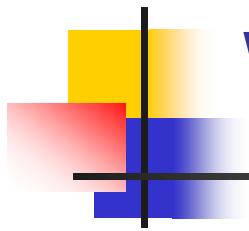
- A good test has a high probability of finding an error.
- A good test is not redundant.
- A good test should be “best of breed.”
- A good test should be neither too simple nor too complex.



# Approaches to Testing

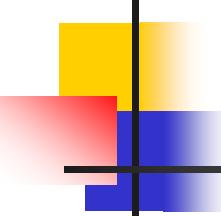
Any engineered product can be tested in one of two ways:

1. Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.
2. Knowing the internal workings of a product, tests can be conducted to ensure that “all gears mesh,” that is, internal operations are performed according to specifications and all internal components have been adequately exercised.



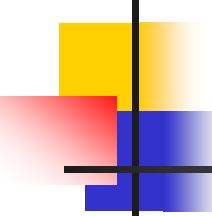
# White Box Integration Testing

- **White-box testing**, is an integration testing philosophy that uses implementation knowledge of the control structures described as part of component-level design to derive test cases.
- White-box tests can be only be designed after source code exists and program logic details are known.
- Logical paths through the software and collaborations between components are the focus of white-box integration testing.
- Important data structures should also be tested for validity after component integration.



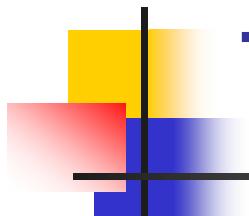
# Integration Testing

- **Integration testing** is a systematic technique for constructing the software architecture while conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that matches the design.
- In the **big bang** approach, all components are combined at once and the entire program is tested as a whole. Chaos usually results!
- In **incremental integration** a program is constructed and tested in small increments, making errors easier to isolate and correct. Far more cost-effective!



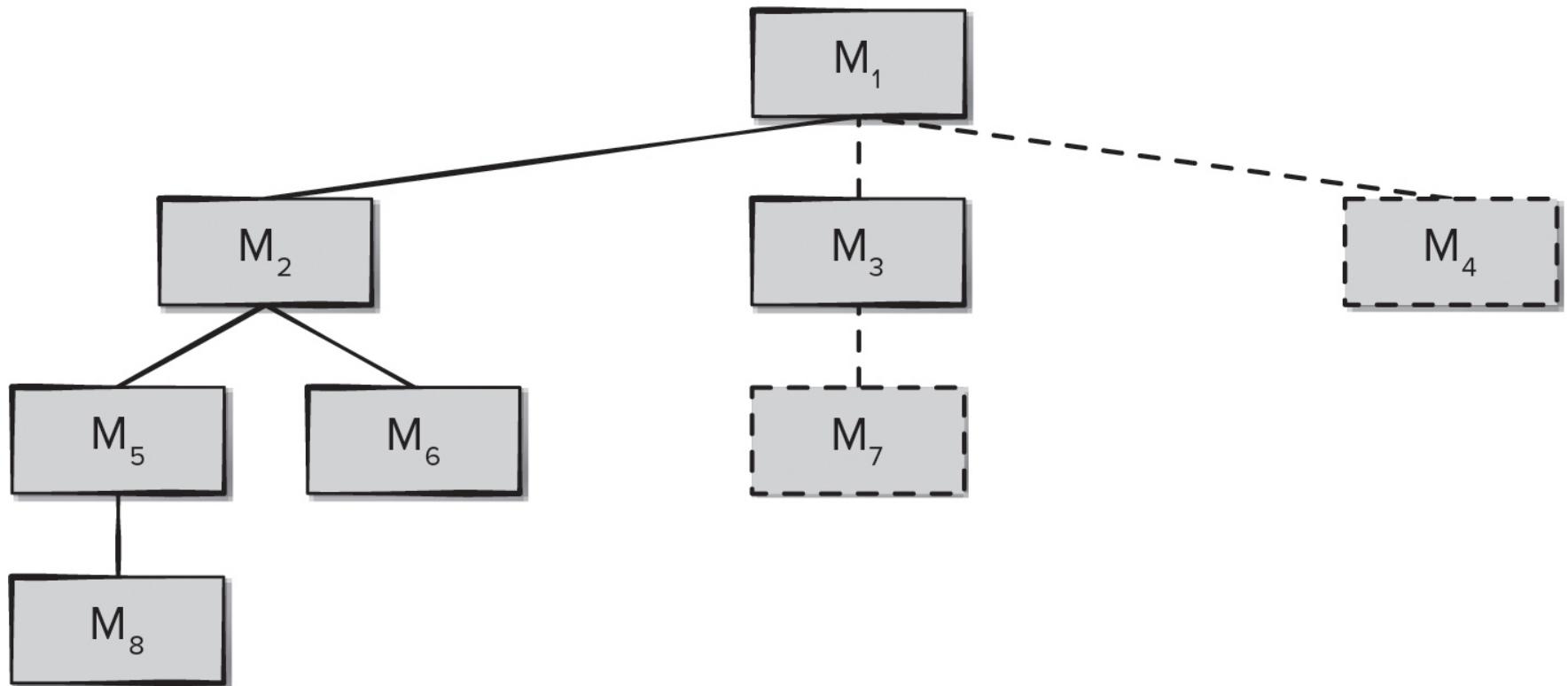
# Top-Down Integration

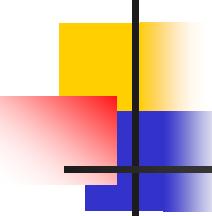
- **Top-down integration testing** is an incremental approach to construction of the software architecture.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate to the main control module are incorporated into the structure followed by their subordinates.
- **Depth-first integration** integrates all components on a major control path of the program structure before starting another major control path.
- **Breadth-first integration** incorporates all components directly subordinate at each level, moving across the structure horizontally before moving down to the next level of subordinates.



# Top-Down Integration

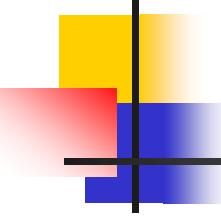
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Top-Down Integration Testing

- The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.
- Depending on the integration approach selected (for example, depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.
- On completion of each set of tests, another stub is replaced with the real component.
- Regression testing may be conducted to ensure that new errors have not been introduced.



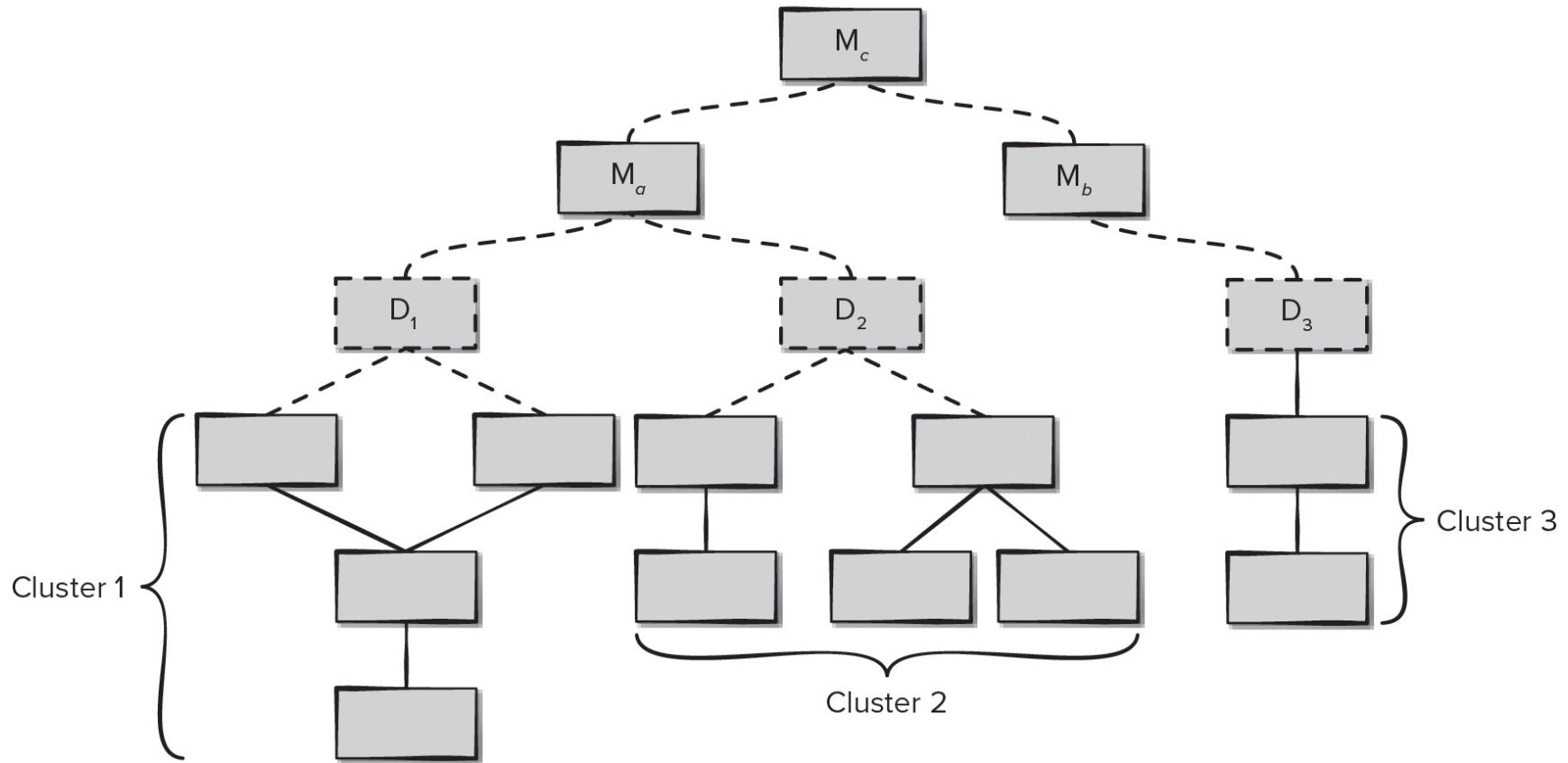
# Bottom-Up Integration Testing

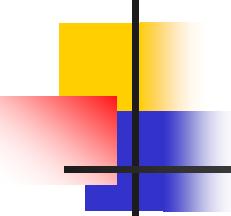
**Bottom-up integration testing**, begins construction and testing with atomic modules components at the lowest levels in the program structure.

- Low-level components are combined into clusters (**builds**) that perform a specific software subfunction.
- A **driver** (a control program for testing) is written to coordinate test-case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined, moving upward in the program structure.

# Bottom-Up Integration

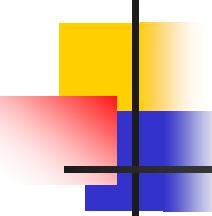
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





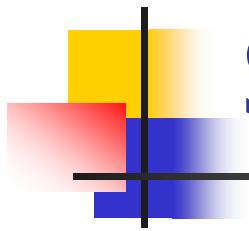
# Continuous Integration

- **Continuous integration** is the practice of merging components into the evolving software increment at least once a day.
- This is a common practice for teams following agile development practices such as XP or DevOps. Integration testing must take place quickly and efficiently if a team is attempting to always have a working program in place as part of continuous delivery.
- **Smoke testing** is an integration testing approach that can be used when software is developed by an agile team using short increment build times.



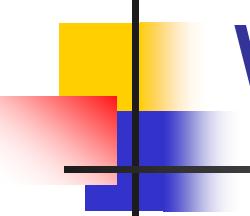
# Smoke Testing Integration

- Software components that have been translated into code are integrated into a build. – that includes all data files, libraries, reusable modules, and components required to implement one or more product functions.
- A series of tests is designed to expose “show-stopper” errors that will keep the build from properly performing its function cause the project to fall behind schedule.
- The build is integrated (either top-down or bottom-up) with other builds, and the entire product (in its current form) is smoke tested daily.



# Smoke Testing Advantages

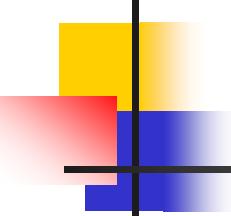
- **Integration risk is minimized**, since smoke tests are run daily.
- **Quality of the end product is improved**, functional and architectural problems are uncovered early.
- **Error diagnosis and correction are simplified**, errors are most likely in (or caused by) the new build.
- **Progress is easier to assess**, each day more of the final product is complete.
- Smoke testing resembles regression testing by ensuring newly added components do not interfere with the behaviors of existing components.



# Integration Testing

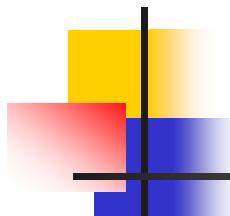
## Work Products

- An overall plan for integration of the software and a description of specific tests is documented in a **test specification**.
- Test specification incorporates a test plan and a test procedure and becomes part of the software configuration.
- Testing is divided into phases and incremental builds that address specific functional and behavioral characteristics of the software.
- Time and resources must be allocated to each increment build along with the test cases needed.
- A history of actual test results, problems, or peculiarities is recorded in a test report and may be appended to the test specification.
- It is often best to implement the test report as a shared Web document to allow all stakeholders access to the latest test results and the current state of the software increment.



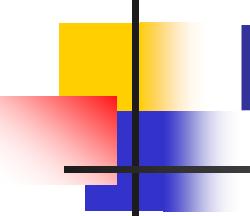
# Regression Testing

- **Regression testing** is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.
- AI tools may be able to help select the best subset of test cases to use in regression automatically based on previous experiences of the developers with the evolving software product.



# OO Integration Testing

- **Thread-based testing**, integrates the set of classes required to respond to one input or event for the system.
  - Each thread is integrated and tested individually.
  - Regression testing is applied to ensure no side effects occur.
- **Use-based testing**, begins the construction of the system by testing those classes (called **independent classes**) that use very few server classes.
  - The next layer classes, (called **dependent classes**) use the independent classes are tested next.
  - This sequence of testing layers of dependent classes continues until the entire system is constructed.

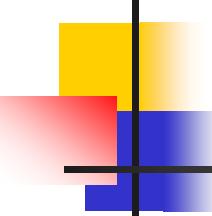


# OO Testing – Fault-Based Test Case Design

- The object of **fault-based testing** is to design tests that have a high likelihood of uncovering plausible faults.
- Because the product or system must conform to customer requirements, fault-based testing begins with the analysis model.
- The strategy for fault-based testing is to hypothesize a set of plausible faults and then derive tests to prove each hypothesis.
- To determine whether these faults exist, test cases are designed to exercise the design or code.

# Fault-Based OO Integration Testing

- Fault-based integration testing looks for plausible faults in operation calls or message connections:
  - unexpected result
  - wrong operation/message used
  - incorrect invocation
- Integration testing applies to attributes and operations – class behaviors are defined by the attributes.
- Focus of integration testing is to determine whether errors exist in the calling (client) code, not the called (server) code.

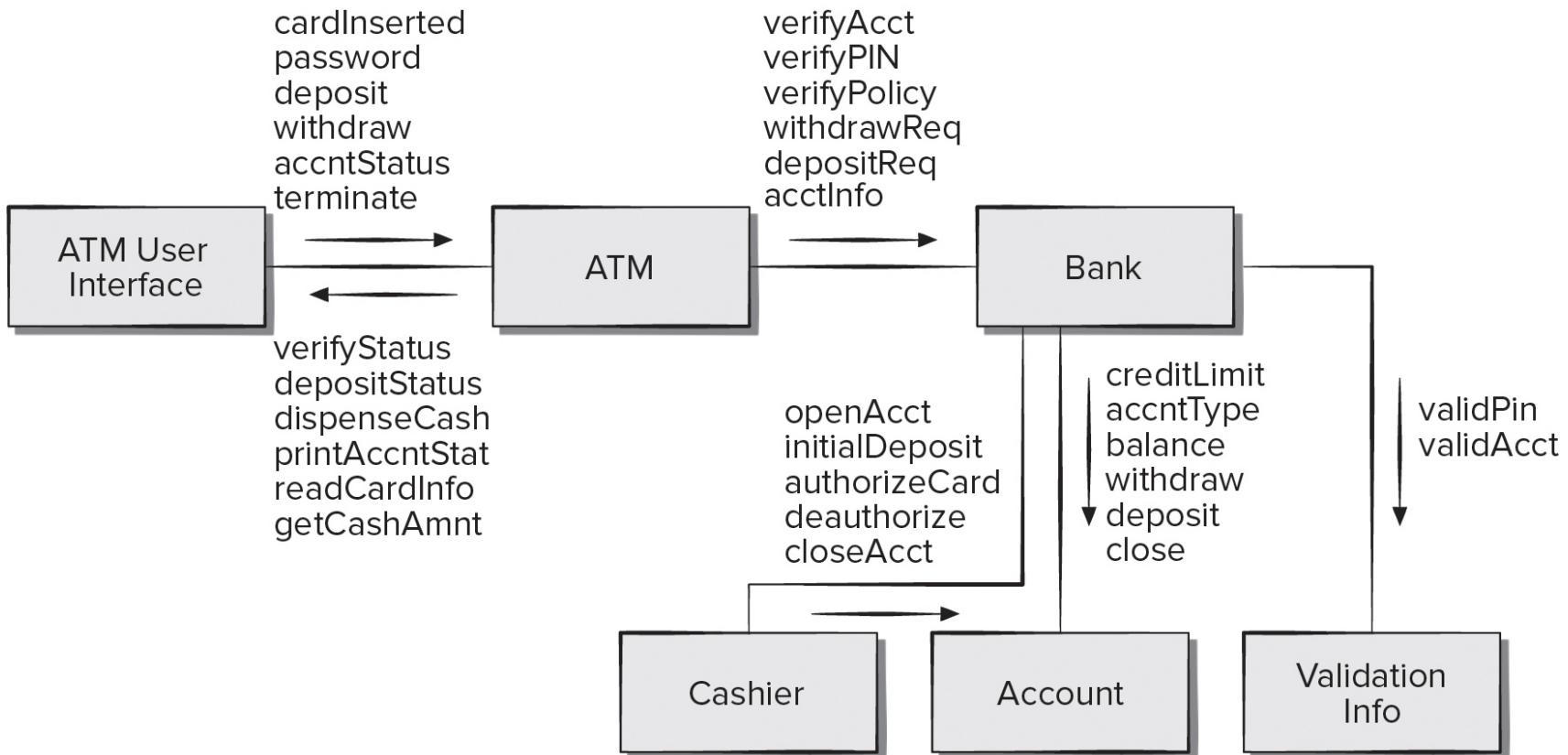


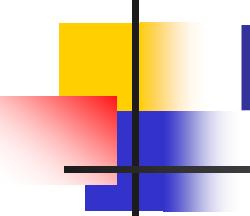
# OO Testing – Fault-Based Test Case Design

- Scenario-based testing uncovers errors that occur when any actor interacts with the software.
- Scenario-based testing concentrates on what the user does, not what the product does.
- This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests.
- Scenario testing uncovers interaction errors.
- Scenario-based testing tends to exercise multiple subsystems in a single test.
- Test-case design becomes more complicated as integration of the object-oriented system occurs since this is when testing of collaborations between classes must begin.

# Collaboration Diagram for Banking Application

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

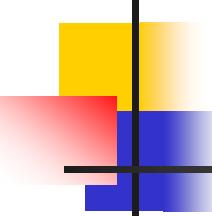




# OO Testing – Random Test Case Design

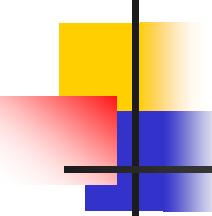
- For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.
- For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
- For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
- For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.
- A random test case for the Bank class might be:

*Test case r3 = verifyAcct·verifyPIN·depositReq*



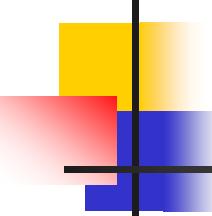
# OO Testing – Scenario-Based Test Case Design

- Scenario-based testing uncovers errors that occur when any actor interacts with the software.
- Scenario-based testing concentrates on what the user does, not what the product does.
- This means capturing the tasks (via use cases) that the user has to perform and then applying them and their variants as tests.
- Scenario testing uncovers interaction errors.
- Scenario-based testing tends to exercise multiple subsystems in a single test.
- Test-case design becomes more complicated as integration of the object-oriented system occurs since this is when testing of collaborations between classes must begin.



# Validation Testing

- **Validation testing** tries to uncover errors, but the focus is at the requirements level - on user visible actions and user-recognizable output from the system.
- Validation testing begins at the culmination of integration testing, the software is completely assembled as a package and errors have been corrected.
- Each user story has user-visible attributes, and the customer's acceptance criteria which forms the basis for the test cases used in validation-testing.
- A **deficiency list** is created when a deviation from a specification is uncovered and their resolution is negotiated with all stakeholders.
- An important element of the validation process is a **configuration review** (audit) that ensures the complete system was built properly.

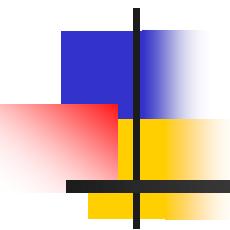


# Software Testing Patterns

- Testing patterns are described in much the same way as design patterns.
- Example:

*Pattern name:* **ScenarioTesting**

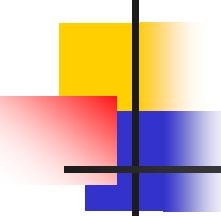
*Abstract:* Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The ScenarioTesting pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement.



# *COMP 354: Introduction to Software Engineering*

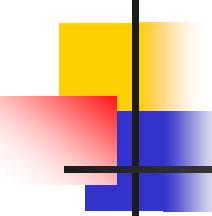
## Software Metrics and Analytics

Based on Chapter 23 of the textbook



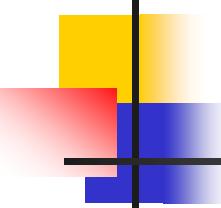
# Measures, Metrics, and Indicators

- A **measure** provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process.
- A **metric** is a quantitative measure of the degree to which a system, component, or process possesses a given attribute.
- An **indicator** is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself.



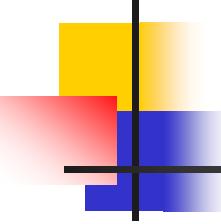
# Attributes of Effective Metrics

- **Simple and computable.** It should be relatively easy to learn how to derive the metric.
- **Empirically and intuitively persuasive.** Satisfies the engineer's intuitive notions about the product attribute.
- **Consistent and objective.** The metric should yield results that are unambiguous.
- **Consistent in its use of units and dimensions.** Computation of the metric should not lead to bizarre combinations of units.
- **Programming language independent.** Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- **Effective mechanism for quality feedback.** Should provide a software engineer with information that can lead to a higher quality end-product.



# Software Analytics

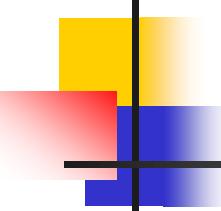
- **Key performance indicators** (KPIs) are metrics that are used to track performance and trigger remedial actions when their values fall in a predetermined range.
- How do you know that metrics are meaningful in the first place?
- **Software analytics** is the systematic computational analysis of software engineering data or statistics to provide managers and software engineers with meaningful insights and empower their teams to make better decisions.



# Software Analytics

Software analytics can help developers make decisions regarding:

- Targeted testing.
- Targeted refactoring.
- Release planning.
- Understanding customers.
- Judging stability.
- Targeting inspection.



# Requirements Model Metrics

- Requirement specificity (lack of ambiguity):

$$Q_1 = n_{ui} / n_r$$

where  $n_{ui}$  is the number of requirements for which all reviewers had identical interpretations.

- $Q_1$  close to 1 is good.
- Assume there are  $n_r$  requirements in a specification:

$$n_r = n_f + n_{nf}$$

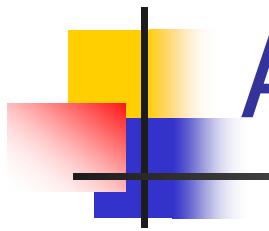
$n_f$  is the number of functional requirements

$n_{nf}$  is the number of nonfunctional requirements

# Mobile Software Requirements Model Metrics

- Number of static screen displays. ( $N_{sp}$ )
- Number of dynamic screen displays. ( $N_{dp}$ )
- Number of persistent data objects.
- Number of external systems interfaced.
- Number of static content objects.
- Number of dynamic content objects.
- Number of executable functions.
- Customization index  $C = N_{dp} / (N_{dp} + N_{sp})$

*C* ranges from 0 to 1, larger *C* is better



# Architectural Design Metrics

Architectural design metrics

- Structural complexity =  $g(\text{fan-out})$ .
- Data complexity =  $f(\text{input \& output variables, fan-out})$ .
- System complexity =  $h(\text{structural \& data complexity})$ .

Morphology metrics: a function of the number of modules and the number of interfaces between modules

Size =  $n + a$

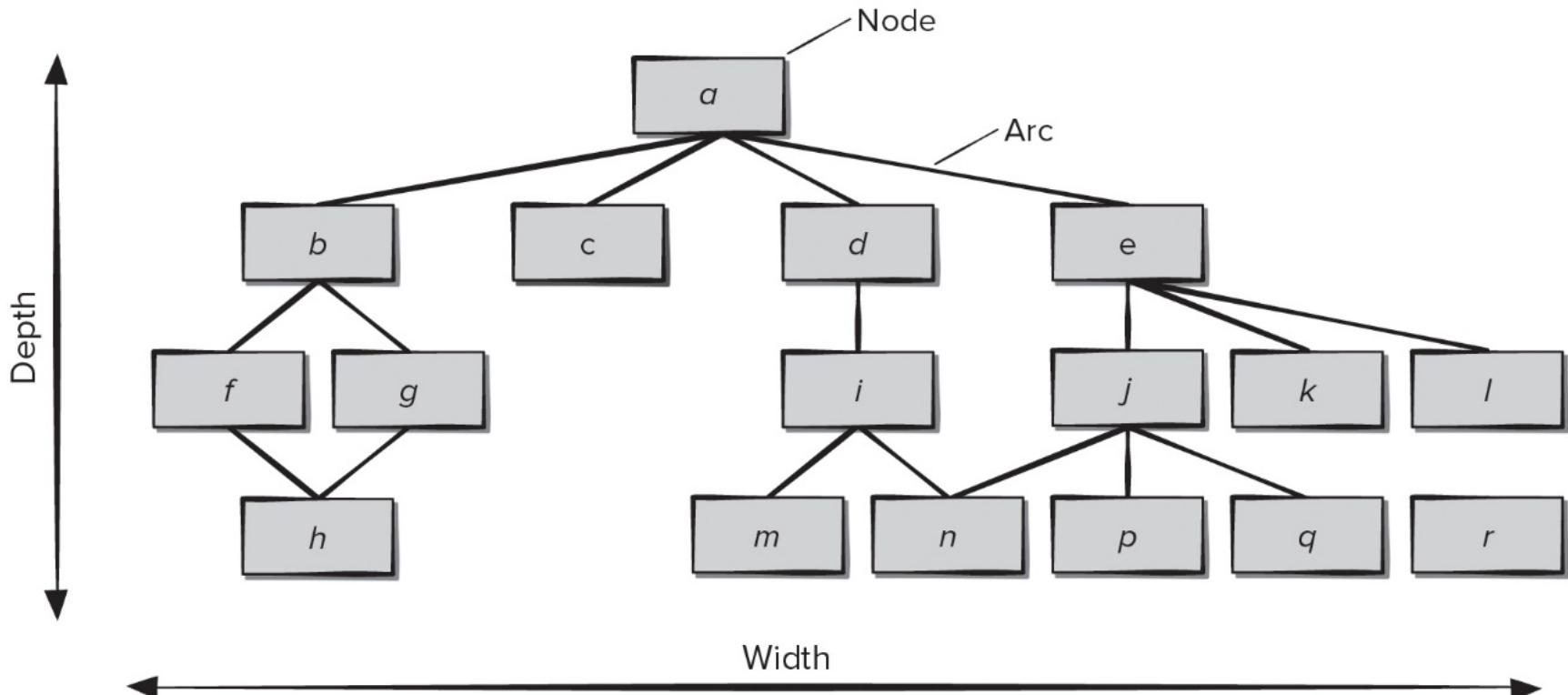
$n$  = number of nodes,  $a$  = number of arcs

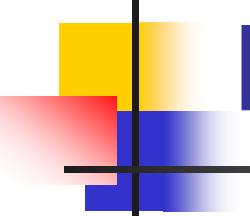
Depth = longest path root to leaf node

Width = maximum number of nodes at each level

# Morphology Metrics

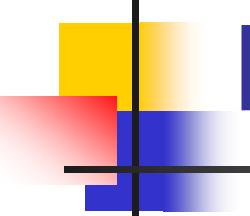
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Object-Oriented Design Metrics

- **Weighted methods per class (WMC).** The number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class.
- **Depth of the inheritance tree (DIT).** A deep class hierarchy (DIT is large) leads to greater design complexity.
- **Number of children (NOC).** As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

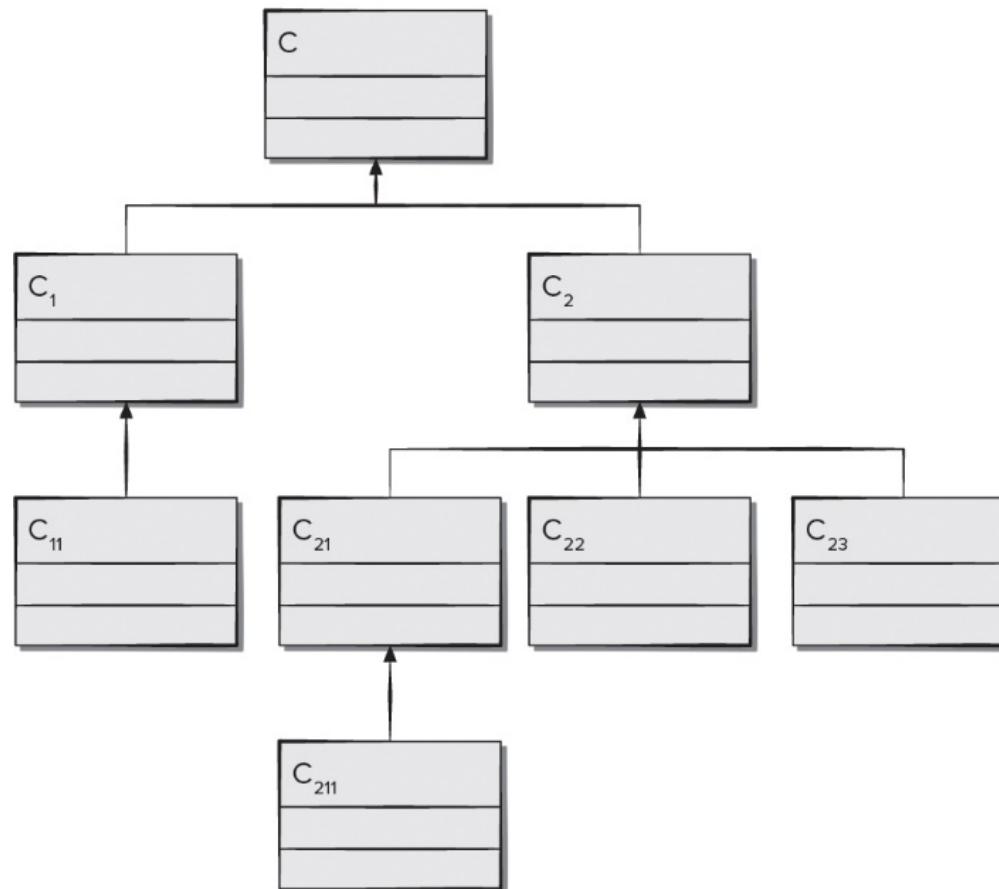


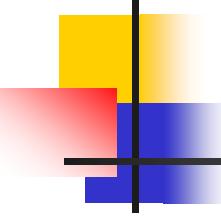
# Object-Oriented Design Metrics

- **Coupling between object classes (CBO).** High values of CBO indicate poor reusability and make testing of modifications more complicated.
- **Response for a class (RFC).** The number of methods that can potentially be executed in response to a message received by an object of the class.
- **Lack of cohesion in methods (LCOM).** LCOM is the number of methods that access one or more of the same attributes

# Class Hierarchy

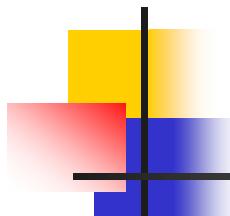
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# User Interface Design Metrics

- **Interface metrics.** Ergonomics measures (for example, memory load, typing effort, recognition time, layout complexity)
- **Aesthetic (graphic design) metrics.** Aesthetic design relies on qualitative judgment but some measures are possible (for example, word count, graphic percentage, page size)
- **Content metrics.** Focus on content complexity and on clusters of content objects that are organized into pages
- **Navigation metrics.** Address the complexity of the navigational flow and they are applicable only for static Web applications.



# Source Code Metrics

**Halstead's Software Science:** a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program.

$n_1$  = number of distinct operators that appear in a program

$n_2$  = number of distinct operands that appear in a program

$N_1$  = total number of operator occurrences

$N_2$  = total number of operand occurrences

Program length

$$M = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

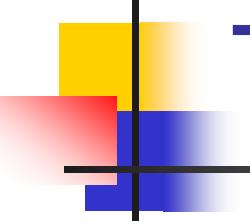
Program volume

$$V = M \log_2 (n_1 + n_2)$$

Volume Ratio

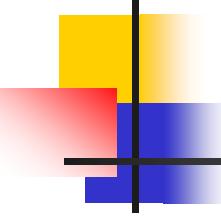
$$L = (2 / n_1) \times (n_2 / N_2)$$

$L$  is ratio of most compact form of the program to actual program size



# Testing Metrics

- Testing effort estimated using metrics derived from Halstead measures
  - Program level  $PL = 1 / L$
  - Effort  $e = V / PL$
- Some OO design metrics have influence on “testability”
  - Lack of cohesion in methods (LCOM).
  - Percent public and protected (PAP).
  - Public access to data members (PAD).
  - Number of root classes (NOR).
  - Fan-in (FIN).
  - Number of children (NOC) and depth of the inheritance tree (DIT).



# Maintenance Metrics

- IEEE Std. 982.1-2005 software maturity index (SMI) that provides an indication of the software product stability (based on changes made).

$MT$  = number of modules in current release

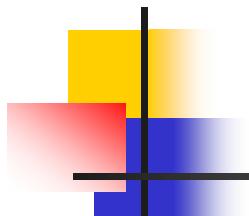
$F_c$  = number of modules in current release that have been changed

$F_a$  = number of modules in current release that have been added

$F_d$  = number of preceding release modules deleted

$$SMI = [MT - (F_a + F_c + F_d)] / MT$$

- As SMI approaches 1.0, the product begins to stabilize.

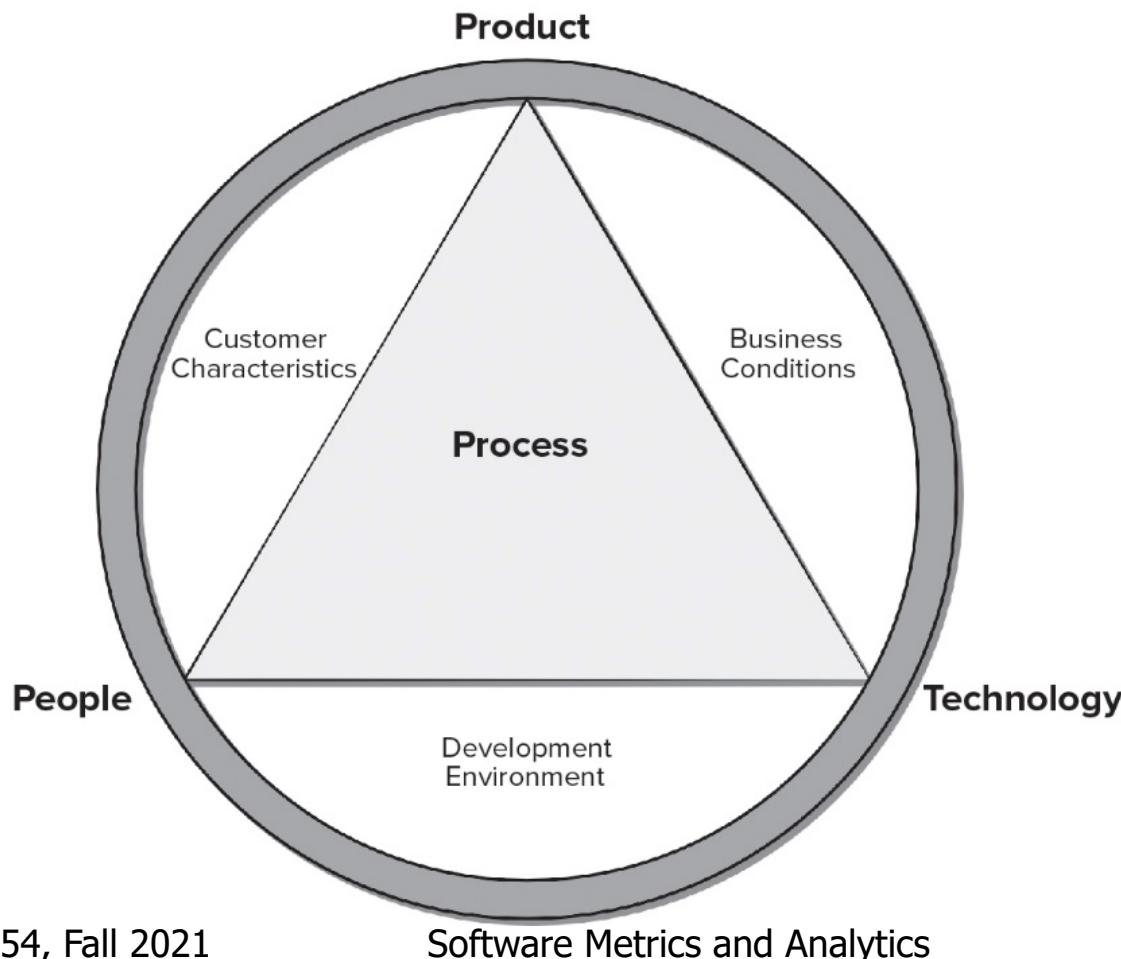


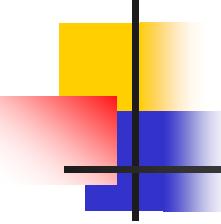
# Process and Project Metrics

- **Process metrics** collected across all projects, over long periods of time. Their intent is to provide a set of indicators that lead to long-term software process improvement
- **Project metrics** enable a software project manager to:
  - assess the status of an ongoing project.
  - track potential risks.
  - uncover problem areas before they go “critical.”
  - adjust work flow or tasks.
  - evaluate project team’s ability to control quality of software work products.

# Determinants of Software Quality and Organizational Effectiveness

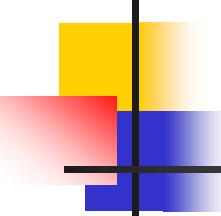
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





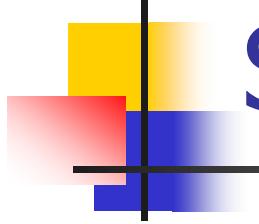
# Process Measurement

- We measure the efficacy of a software process indirectly – by deriving metrics based on the outcomes that can be derived from the process:
  - measures of errors uncovered before release of the software.
  - defects delivered to and reported by end-users.
  - work products delivered (productivity).
  - human effort expended.
  - calendar time expended.
  - schedule conformance.
  - other measures.
- We also derive process metrics by measuring the characteristics of specific software engineering tasks



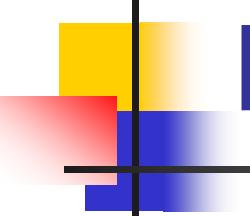
# Process Metrics Guidelines

- Use common sense and organizational sensitivity when interpreting metrics data.
- Provide regular feedback to the individuals and teams who collect measures and metrics.
- Don't use metrics to appraise individuals.
- Work with practitioners and teams to set clear goals and metrics that will be used to achieve them.
- Never use metrics to threaten individuals or teams.
- Metrics data that indicate a problem area should not be considered "negative." These data are merely an indicator for process improvement.
- Don't obsess on a single metric to the exclusion of other important metrics.



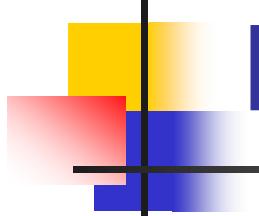
# Software Measurement

- **Direct measures** of the software process include cost and effort applied.
- Direct measures of the product include lines of code (LOC) produced, execution speed, memory size, and defects reported over some set period of time.
- **Indirect measures** of the product include functionality, quality, complexity, efficiency, reliability, maintainability, and many others.
- Direct measures are relatively easy to collect, the quality and functionality of software are more difficult to assess and can be measured only indirectly.



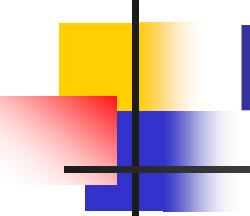
# Normalized Size-Oriented Metrics

- errors per KLOC (thousand lines of code)
- defects per KLOC
- \$ per LOC
- pages of documentation per KLOC
- errors per person-month
- errors per review hour
- LOC per person-month
- \$ per page of documentation



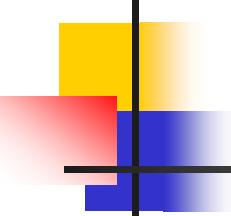
# Normalized Function-Oriented Metrics

- errors per FP (thousand lines of code)
- defects per FP
- \$ per FP
- pages of documentation per FP
- FP per person-month



# Why Opt For Function-Oriented Metrics

- Programming language independent.
- Used readily countable characteristics that are determined early in the software process.
- Does not “penalize” inventive (short) implementations that use fewer L O C than other more clumsy versions.
- Makes it easier to measure the impact of reusable components.



# Software Quality Metrics

- **Correctness.** degree to which the software performs its required function (for example, defects per K L O C).
- **Maintainability.** degree to which a program is amenable to change (for example, M T T C - mean time to change).
- **Integrity.** degree to which a program is impervious to outside attack.

$$\text{Integrity} = \sum [1 - (\text{threat} \times (1 - \text{security}))]$$

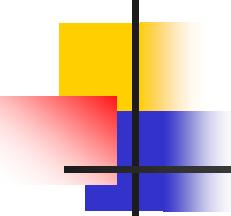
threat = probability specific attack occurs

security = probability specific attack is repelled

- **Usability.** quantifies ease of use (for example, error rate).

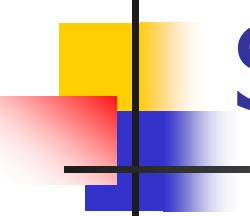
# Defect Removal Efficiency (DRE)

- DRE is a measure of the filtering ability of quality assurance and control actions as they are applied throughout all process framework activities.
$$\text{DRE} = E / (E + D)$$
$$E = \text{number of errors found before delivery}$$
$$D = \text{number of errors found after delivery}$$
- The ideal value for DRE is 1. No defects ( $D = 0$ ) are found by the consumers of a work product after delivery.
- The value of DRE begins to approach 1 as  $E$  increases many the team is catching its own errors.



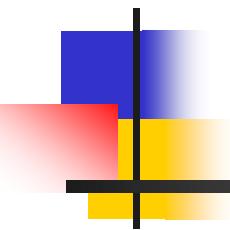
# Goal Driven Metrics Program

1. Identify your business goals.
2. Identify what you want to know or learn.
3. Identify your subgoals.
4. Identify the entities and attributes related to your subgoals.
5. Formalize your measurement goals.
6. Identify quantifiable questions and the related indicators that you will use to help you achieve your measurement goals.
7. Identify the data elements that you will collect to construct the indicators.
8. Identify the measures to be used, and make these definitions operational.
9. Identify the actions that you will take to implement the measures.
10. Prepare a plan for implementing the measures.



# Metrics for Small Organizations

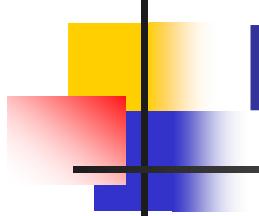
- time (hours or days) elapsed from the time a request is made until evaluation is complete,  $t_{queue}$  .
- effort (person-hours) to perform the evaluation,  $W_{eval}$  .
- time (hours or days) elapsed from completion of evaluation to assignment of change order to personnel,  $t_{eval}$  .
- effort (person-hours) required to make the change,  $W_{change}$  .
- time required (hours or days) to make the change,  $t_{change}$  .
- errors uncovered during work to make change,  $E_{change}$  .
- defects uncovered after change is released to the customer base,  $D_{change}$  .



# *COMP 354: Introduction to Software Engineering*

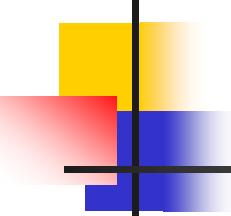
## Project Management Concepts

Based on Chapter 24 of the textbook



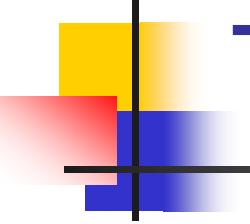
# Management Spectrum – Four P's

- **People** — the most important element of a successful project.
- **Product** — the software to be built.
- **Process** — the set of framework activities and software engineering tasks to get the job done.
- **Project** — all work required to make the product a reality.



# Stakeholders

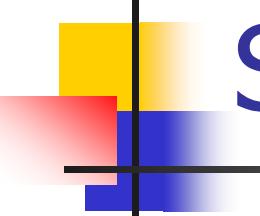
- **Senior managers** - define the business issues that often have significant influence on the project.
- **Project (technical) managers** - who must plan, motivate, organize, and control the practitioner.
- **Practitioners** - who deliver the technical skills that are necessary to engineer a product or application.
- **Customers** - specify requirements for the software to be engineered and other interested product stakeholders.
- **End-users** - interact with the software once it is released for production use.



# Team Leaders

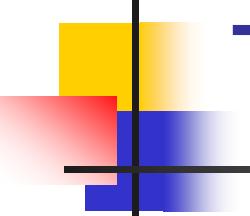
Kouzes exemplary practices for technology leaders:

- **Model the way.** Leaders must practice what they preach. They demonstrate commitment to team and project by shared sacrifice.
- **Inspire and shared vision.** Motivate team members to tie their personal aspirations to team goals. Involve stakeholders early.
- **Challenge the process.** Encourage team members to experiment and take risks by helping them generate frequent small successes while learning from their failures.
- **Enable others to act.** Increase the team's sense of competence through sharing decision making and goal setting.
- **Encourage the heart.** Build community (team) spirit by celebrating shared goals and victories (individual and team).



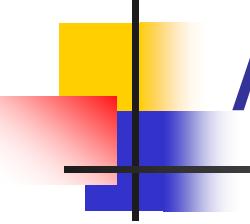
# Factors Affecting Software Team Structure

- **Difficulty** of the problem to be solved.
- **Size** of the resultant program(s) in lines of code or function points.
- **Team lifetimes** - time that the team will stay together.
- Degree to which the problem can be **modularized**.
- Required **quality** and **reliability** of the system to be built.
- Rigidity of the **delivery date**.
- **Communication** (degree of sociability) required.



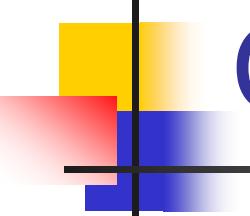
# Team Toxicity Factors

- **Frenzied work atmosphere** team members waste energy and lose focus on work objectives.
- **High frustration** caused by personal, business, or technological factors causing team member friction.
- **Fragmented or poorly coordinated procedures** poorly defined or improperly chosen process model.
- **Unclear definition of roles** resulting in lack of accountability and resultant finger-pointing.
- **Continuous and repeated exposure to failure** leads to a loss of confidence and a lowering of morale.



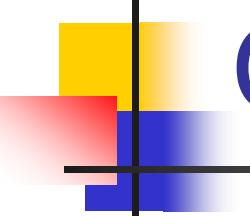
# Agile Teams

- Team members must have trust in one another.
- The distribution of skills must be appropriate to the problem.
- Mavericks may have to be excluded from the team, if team cohesiveness is to be maintained.
- Team is “self-organizing.”
  - An adaptive team structure.
  - Planning is kept to a minimum.
  - Team select its own approach constrained by business requirements and organizational standards.



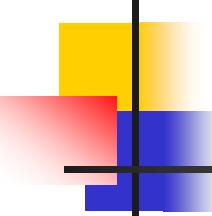
# Team Coordination and Communication Issues

- **Scale** of many development efforts is large, leading to complexity, confusion, and significant difficulties in coordinating team members.
- **Uncertainty** is common, resulting in a continuing stream of changes that ratchets the project team.
- **Interoperability** - new software must communicate with existing software and conform to constraints imposed by existing systems or products.



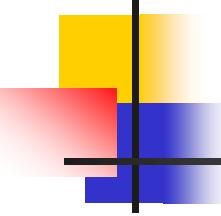
# Team Coordination and Communication

- To accomplish this, mechanisms for formal and informal communication among team members and between multiple teams must be established.
- **Formal communication** is accomplished through writing, structured meetings, and other relatively non-interactive and impersonal communication channels.
- **Informal communication** is more personal and allow team members to interact with one another on a daily basis - share ideas on an ad hoc basis and ask for help as problems arise.



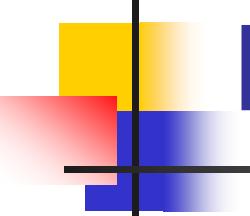
# Software Scope

- Software project scope must be unambiguous and understandable at management and technical levels.
- **Context.** How does the software to be built fit into a larger system, product, or business context and what constraints are imposed as a result of the context?
- **Information objectives.** What customer-visible data objects are produced as output from the software? What data objects are required for input?
- **Function and performance.** What function does the software perform to transform input data into output? Are any special performance characteristics to be addressed?



# Problem Decomposition

- Sometimes called partitioning or problem elaboration.
- Once scope is defined ...
  - It is decomposed into constituent functions.
  - It can be decomposed into user-visible data objects. *or*
  - It can be decomposed into a set of problem classes.
- Decomposition process continues until all functions or problem classes have been defined.

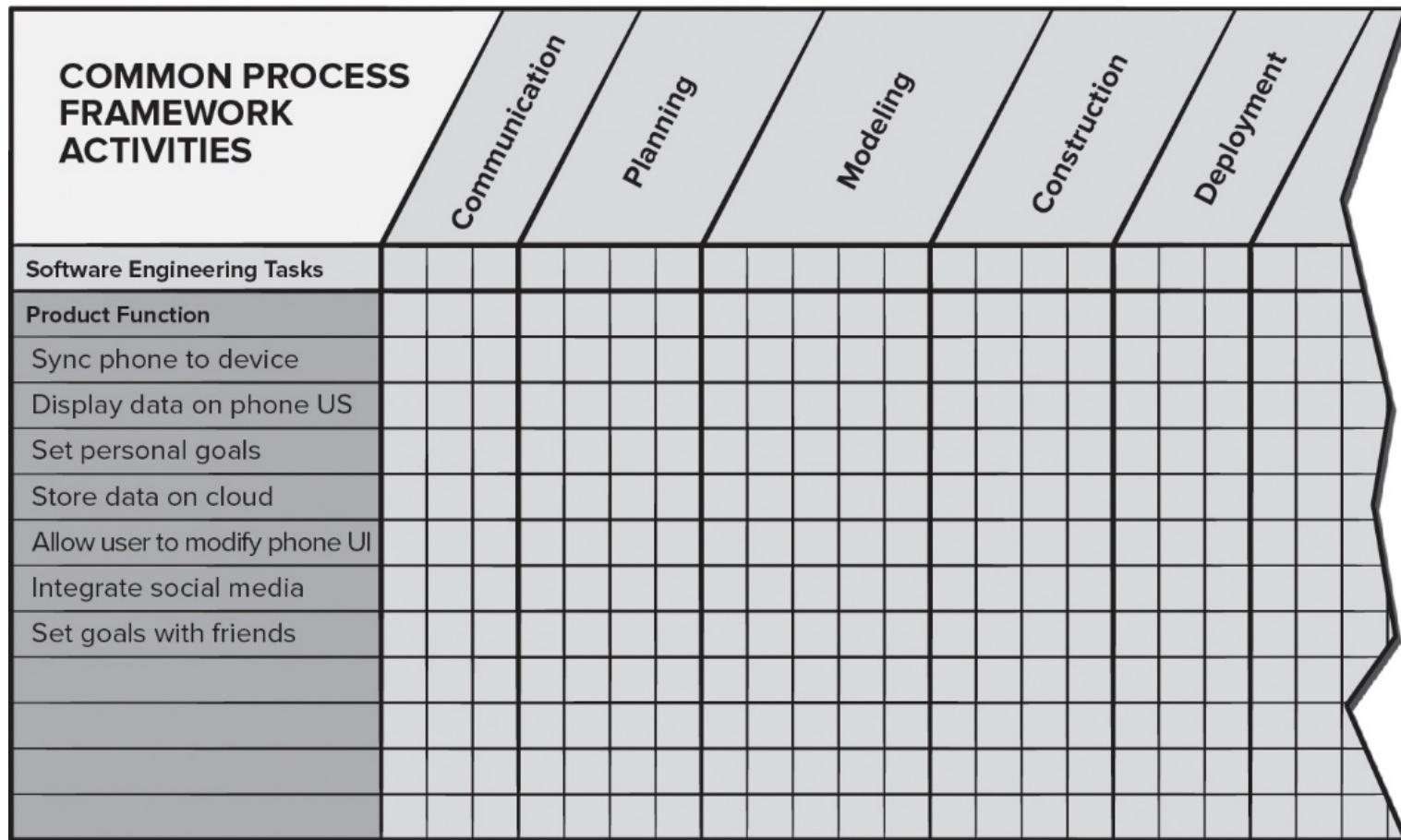


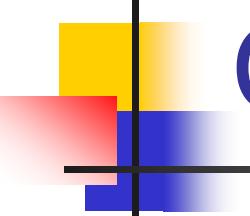
# Process

- Your team must decide which process model is most appropriate for.
  - the customers who have requested the product.
  - the people who will do the work.
  - the characteristics of the product itself.
  - the project environment in which the software team works.
- Team selects process model and defines a preliminary project plan (based set of process framework activities).
- Once the preliminary plan is established, process decomposition begins.

# Melding Product and Process

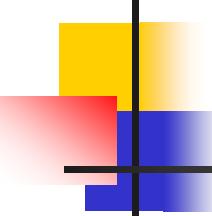
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





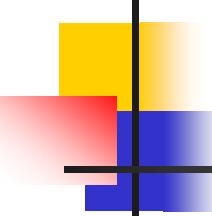
# Successful Project Characteristics

- Clear and well-understood requirements accepted by all stakeholders.
- Active and continuous participation of users throughout the development process.
- A project manager with required leadership skills who is able to share project vision with the team.
- A project plan and schedule developed with stakeholder participation to achieve user goals.
- Skilled and engaged team members.



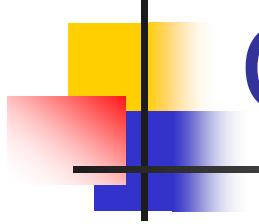
# Successful Project Characteristics

- Development team members with compatible personalities who enjoy working in a collaborative environment.
- Realistic schedule and budget estimates which are monitored and maintained.
- Customer needs that are understood and satisfied.
- Team members who experience a high degree of job satisfaction.
- A working product that reflects desired scope and quality.



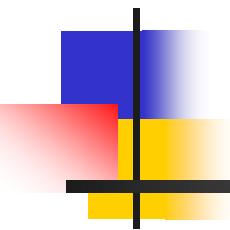
# W<sup>5</sup>HH Principle

- Why is the system being developed?
- What will be done?
- When will it be accomplished?
- Who is responsible?
- Where are they organizationally located?
- How will the job be done technically and managerially?
- How much of each resource (for example, people, software, tools, database) will be needed?



# Critical Practices

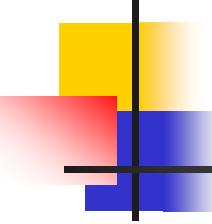
- Formal risk management.
- Empirical cost and schedule estimation.
- Metrics-based project management.
- Earned value tracking.
- Defect tracking against quality targets.
- People aware project management.



# *COMP 354: Introduction to Software Engineering*

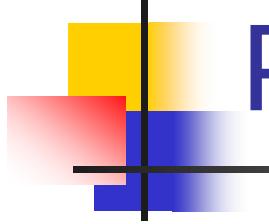
## Viable Software Plan

Based on Chapter 25 of the textbook



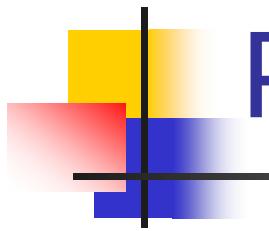
# Estimation Issues

- Estimation of resources, cost, and schedule for a software engineering effort requires:
  - Experience.
  - Access to good historical information (metrics).
  - The courage to commit to quantitative predictions when qualitative information is all that exists.
- Estimation carries inherent risk and this risk leads to uncertainty:
  - Project complexity.
  - Project size (makes decomposition tougher).
  - Degree of structural uncertainty (requirements stability).



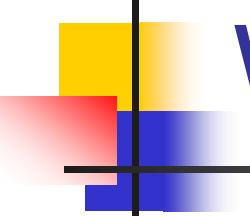
# Project Planning Task Set

1. Establish project scope.
2. Determine feasibility.
3. Analyze risks (Chapter 26).
4. Define required resources.
  - a. Determine required human resources.
  - b. Define reusable software resources.
  - c. Identify environmental resources.
5. Estimate cost and effort.
  - a. Decompose the problem.
  - b. Develop two or more estimates.
  - c. Reconcile the estimates.



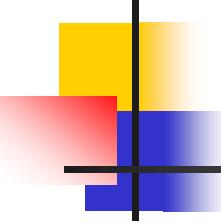
# Project Planning Task Set

6. Develop an initial project schedule.
  - a. Establish a meaningful task set.
  - b. Define a task network.
  - c. Use scheduling tools to develop a time-line chart.
  - d. Define schedule tracking mechanisms.
7. Repeat steps 1 to 6 to create a detailed schedule for each prototype as the scope of each prototype is defined.



# What is Scope?

- Software scope describes
  - Functions and features to be delivered to end-users.
  - Data input and output.
  - Content presented to users of using the software.
  - Performance, constraints, interfaces, and reliability that bound the system.
- Scope is defined using one of two techniques:
  - A narrative description of software scope is developed after communication with all stakeholders.
  - A set of use-cases is developed by end-users.



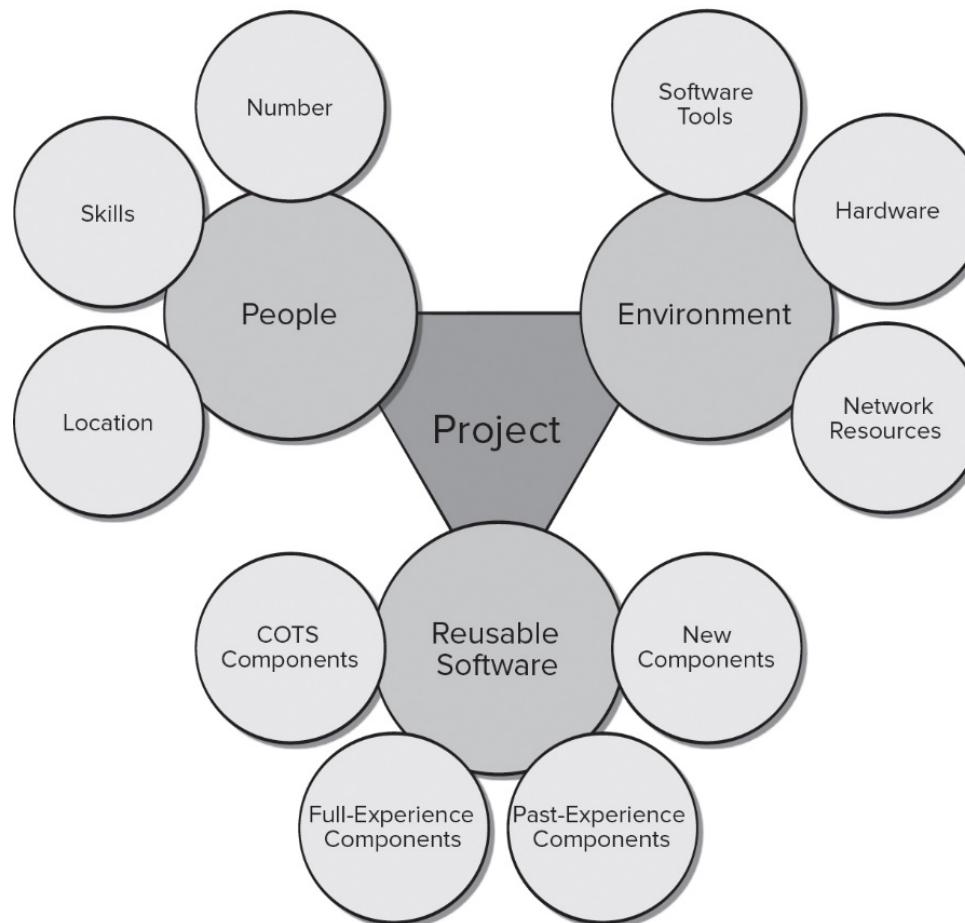
# Project Feasibility

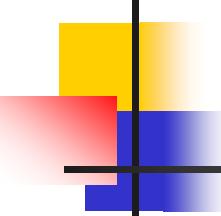
Once scope has been identified (with the concurrence of the customer), it is reasonable to ask:

- Can we build software to meet this scope?
- Is the project feasible?
- You must try to determine if the system can be created using available technology, dollars, time, and other resources.
- Consideration of business need is important too - it does no good to build a high-tech system or product that no one wants.

# Project Resources

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

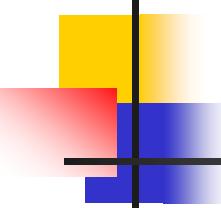




# Data Analytics and Estimation Accuracy

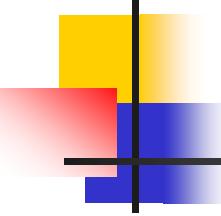
To achieve reliable cost and effort estimates several options arise:

- Delay estimation until late in the project (we can achieve 100 percent accurate estimates after the project is complete!).
- Base estimates on similar projects that have already been completed (works great if you have completed similar projects).
- Use relatively simple decomposition techniques to generate project cost and effort estimates (similar to divide and conquer).
- Use one or more empirical models for software cost and effort estimation (often derived using statistical regression models).



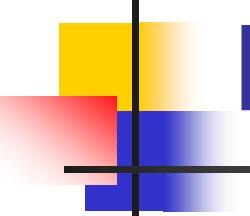
# Reconciling Estimates

- Any estimation technique must be checked by computing at least one other estimate using a different approach.
- If you have created multiple estimates they need to be compared and reconciled.
- If both estimates show agreement, there is good reason to believe that the estimates are reliable.
- Widely divergent estimates can often be traced to one of two causes:
  1. The scope of the project is not adequately understood or has been misinterpreted by the planner.
  2. Productivity data used for problem-based estimation techniques is inappropriate for the application or has been misapplied.



# Problem-Based Estimation

- LOC and FP data are used in two ways during software project estimation:
  1. as estimation variables to “size” each element of the software
  2. as baseline metrics collected from past projects and used with other variable to develop cost and effort projections.
- When collecting productivity metrics for projects, be sure to establish a taxonomy of project types.
- Be sure that your estimates include the effort required to develop “infrastructure” software.



# Problem-Based Estimation

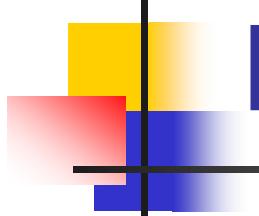
- Begin with a bounded statement of software scope.
- Decompose the statement of scope into problem functions that can each be estimated individually.
- LOC or FP is then estimated for each function.
- Baseline productivity metrics (For example, LOC/pm or FP/pm) are then applied to the appropriate estimation variable.
- Cost/effort for the function is derived using historic data.
- Function estimates are combined to produce an overall estimate for the entire project.



# LOC-Based Estimation Table

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

Function	Estimated LOC
User-interface and control facilities (UICF)	2,300
Two-Dimensional geometric analysis (2DGA)	5,300
Three-Dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (GCDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
<i>Estimated lines of code</i>	33,200



# LOC-Based Estimation

- Average productivity for these systems is 620 LOC/pm.
- Burdened labor rate is \$8000 per month.
- Cost per line of code is approximately \$13.
- Based on LOC estimates and historical data:
  - estimated project cost is \$431,000
  - estimated effort is 54 person-months

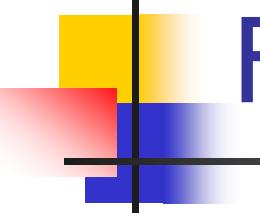


# F P-Based Estimation Table

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

Information domain value	Opt.	Likely	Pess.	Est count.	Weight	F P count
Number of external inputs	20	24	30	24	4	96 ( $24 \times 4 = 96$ )
Number of external outputs	12	14	22	14	5	70 ( $14 \times 5 = 70$ )
Number of external enquiries	16	20	28	20	5	100 ( $20 \times 5 = 100$ )
Number of internal logical files	4	4	5	4	10	40 ( $4 \times 10 = 40$ )
Number of external interface files	2	2	3	2	7	14 ( $2 \times 7 = 14$ )
<b>Count Total</b>						320

**Table 25.1 Estimating information domain values**



# FP-Based Estimation

- To compute the FP equation:

$$FP_{\text{estimated}} = \text{count total} \times [0.65 + 0.01 \times \sum(F_i)]$$

- For the purposes of this estimate, the complexity weighting factor is assumed to be average and the FP count total from the table is 320.
- Assume the sum of the 14 complexity factors  $\sum(F_i)$  is 52.

$$[0.65 + 0.01 \times \sum(F_i)] = 1.17$$

- The estimated number of FP can be computed:

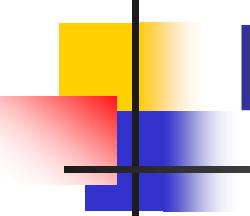
$$FP_{\text{estimated}} = \text{count total} \times [0.65 + 0.01 \times \sum F_i] = 375$$

- If the historic cost per FP is approximately \$1,230 then total estimated project cost is \$461,000 estimated effort is 58 person-months.

# Process-Based Estimation Table

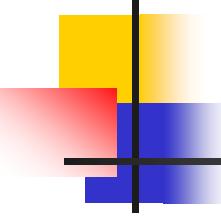
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
<i>Totals</i>	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		



# Process-Based Estimation

- Process-based estimation begins with a delineation of software functions obtained from the project scope.
- A series of framework activities are performed for each function.
- Functions and related framework activities may be represented as part of a table with tasks as columns and rows as functions.
- The effort estimates (for example, person-months) are entered as the matrix cell values.
- Average labor rates (that is, cost/unit effort) are then applied to the effort estimated for each process activity.
- Based on an average burdened labor rate of \$8,000 per month, the total estimated project cost is \$368,000 and the estimated effort is 46 person-months based on the matrix entries.



# Use Case Point Estimation

Computation of use case points takes the following into account:

- The number and complexity of the use cases in the system.
- The number and complexity of the actors on the system.
- Various nonfunctional requirements not written as use cases.
- The environment in which the project will be developed.

$$UCP = (UUCW + UAW) \times TCF \times ECF$$

UUCW – unadjusted sum of use case weights

UAW – unadjusted sum of actor weight

TCF – technical complexity 13 factors

ECF – environment complexity 8 factors

# Use Case Point Estimation Example

The engineering subsystem group is described by 14 average use cases and 8 simple use cases. And the infrastructure subsystem is described with 10 simple use cases.

$$\begin{aligned} \text{UUCW} = & (16 \text{ use cases} \times 15) + [(14 \text{ use cases} \times 10) \\ & + (8 \text{ use cases} \times 5)] + (10 \text{ use cases} \times 5) = 470 \end{aligned}$$

There are 8 simple actors, 12 average actors, and 4 complex actors.

$$\begin{aligned} \text{UAW} = & (8 \text{ actors} \times 1) + (12 \text{ actors} \times 2) + (4 \text{ actors} \times 3) \\ = & 44 \end{aligned}$$

After evaluation of the technology and the environment,

$$\text{TCF} = 1.04$$

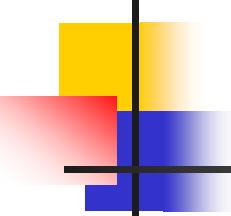
$$\text{ECF} = 0.96$$

$$\text{UCP} = (470 + 44) \times 1.04 \times 0.96 = 513$$

# Use Case Point Estimation

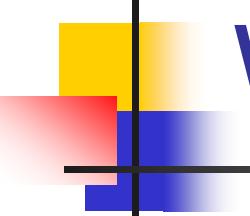
## Example

- Using past project data as a guide, the development group produces 85 LOC per UCP.
- An estimate of the overall size of the CAD project is 43,600 LOC.
- Using 620 LOC/pm as the average productivity for systems of this type and a burdened labor rate of \$8,000 per month, and the cost per line of code is approximately \$13.
- Based on the use case estimate and the historical productivity data:
  - Total estimated project cost is \$552,000.
  - Estimated effort is about 70 person-months.



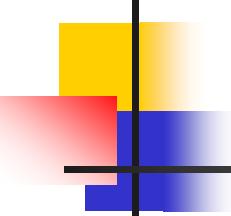
# Agile Project Estimation

1. Each user story is considered separately for estimation purposes.
2. Each user story is decomposed into the set of software engineering tasks that will be required to develop it.
- 3a. Each task is estimated separately (historic data, empirical model, experience, or planning poker).
- 3b. Alternatively, the “volume” of the user story can be estimated in LOC, FP, or use case count.
- 4a. Estimates for each task are summed to estimate the user story.
- 4b. Alternatively, the volume translated into effort using historical data.
5. Effort estimates for all user stories are summed to create effort estimate for the increment.



# Why Are Projects Late?

- Unrealistic deadline established by someone outside the software team and forced on managers and practitioners on the group.
- Changing requirements not reflected in schedule changes.
- An honest underestimate of the amount of effort and/or the number of resources that will be required to do the job.
- Predictable and/or unpredictable risks that were not considered when the project commenced.
- Technical difficulties that could not have been foreseen in advance.
- Human difficulties that could not have been foreseen in advance.
- Miscommunication among project staff that results in delays.
- Failure by project management to recognize that the project is falling behind schedule and lack of action to correct the problem.

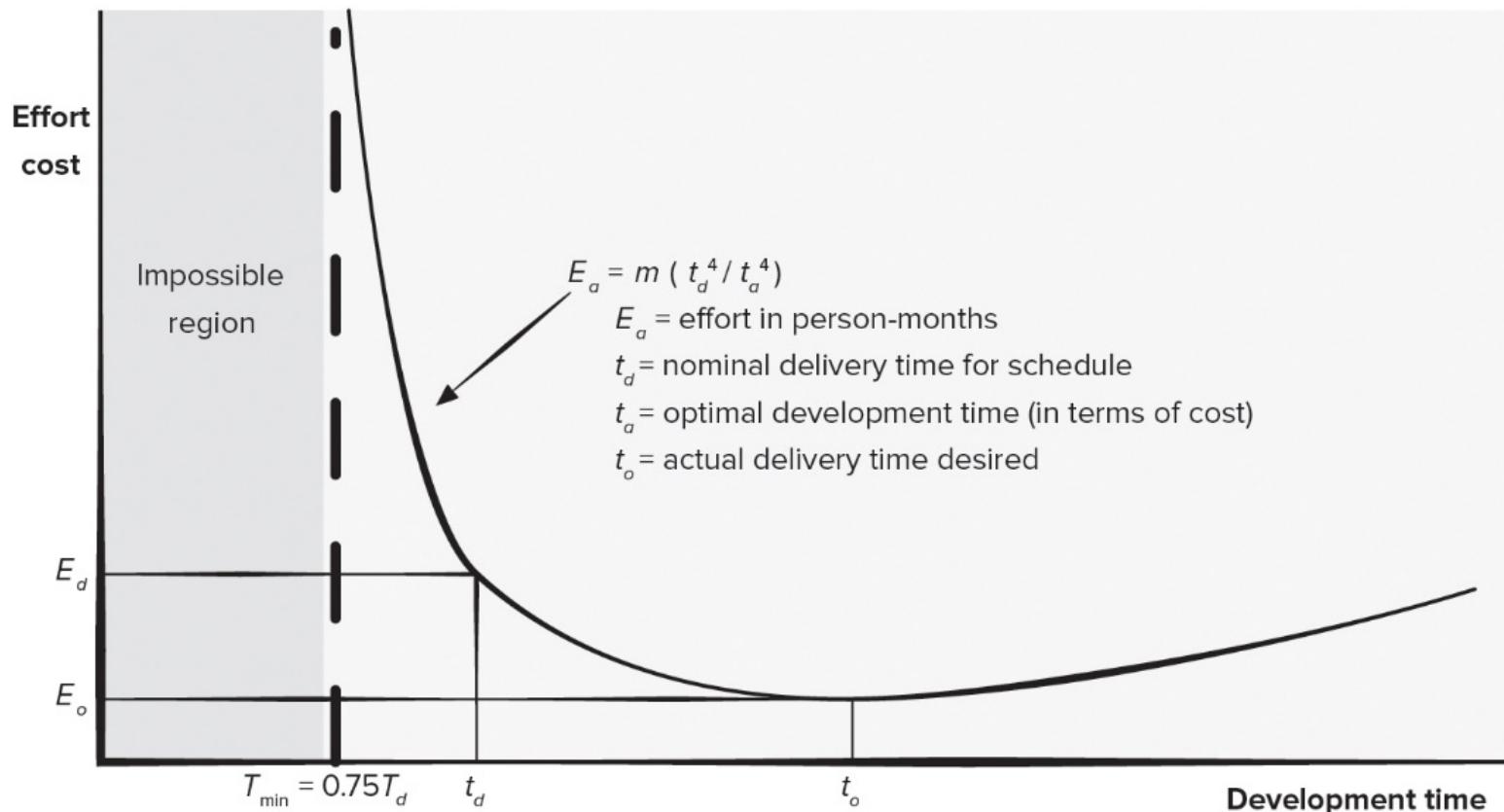


# Scheduling Principles

- **Compartmentalization.** The project must be compartmentalized by decomposing the product and the process.
- **Interdependency.** The interdependency of each compartmentalized activity or task must be determined.
- **Time allocation.** Each task must be allocated some number of work units and assigned a start date and a completion date.
- **Effort validation.** Ensure that no more than the allocated number of people has been scheduled at any given time.
- **Defined responsibilities.** Every task that is scheduled should be assigned to a specific team member.
- **Defined outcomes.** Every task should have a defined outcome.
- **Defined milestones.** Every task should be associated with a project milestone.

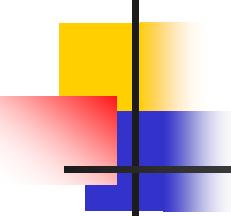
# Relationship Between Effort and Delivery Date

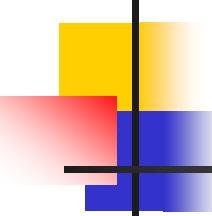
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# Concept Development

## Task Set

- 
- 1.1 **Concept scoping** determines the overall scope of the project.
  - 1.2 **Preliminary concept planning** establishes the organization's ability to undertake the work implied by the project scope.
  - 1.3 **Technology risk assessment** evaluates the risk associated with the technology to be implemented as part of the project scope.
  - 1.4 **Proof of concept** demonstrates the viability of a new technology in the software context.
  - 1.5 **Concept implementation** implements the concept representation in a manner that can be reviewed by a customer and is used for "marketing" purposes when a concept must be sold to other customers or management.
  - 1.6 **Customer reaction** to the concept solicits feedback on a new technology concept and targets specific customer applications.



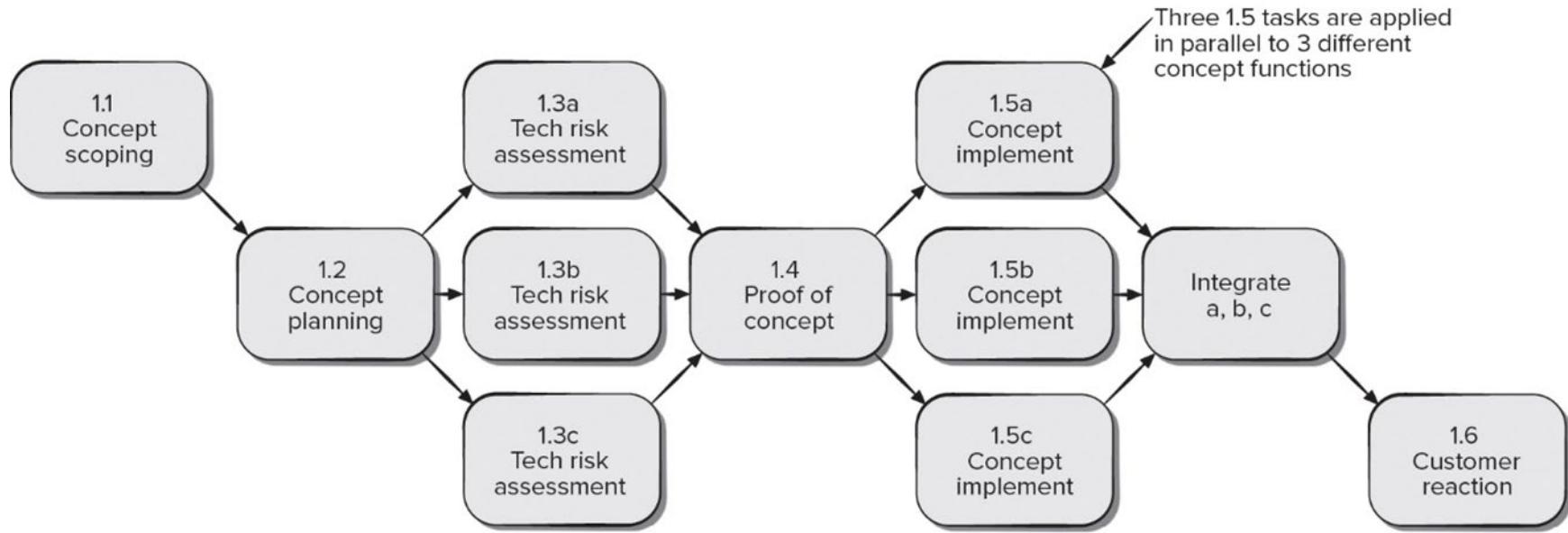
# Task 1.1 Refinement

*Task definition: Task 1.1 Concept Scoping*

- 1.1.1     *Identify need, benefits and potential customers;*
- 1.1.2     *Define desired output/control and input events that drive the application;*  
*Begin Task 1.1.2*
  - 1.1.2.1     *FTR: Review written description of need indicates that a FTR is to be conducted.*
  - 1.1.2.2     *Derive a list of customer visible outputs/inputs*
  - 1.1.2.3     *FTR: Review outputs/inputs with customer and revise as required;*  
*endtask Task 1.1.2*
- 1.1.3     *Define the functionality/behavior for each major function;*  
*Begin Task 1.1.3*
  - 1.1.3.1     *FTR: Review output and input data objects derived in task 1.1.2;*
  - 1.1.3.2     *Derive a model of functions/behaviors;*
  - 1.1.3.3     *FTR: Review functions/behaviors with customer and revise as required;*  
*endtask Task 1.1.3*
- 1.1.4     *Isolate those elements of the technology to be implemented in software;*
- 1.1.5     *Research availability of existing software;*
- 1.1.6     *Define technical feasibility;*
- 1.1.7     *Make quick estimate of size;*
- 1.1.8     *Create a Scope Definition;*  
*endTask definition: Task 1.1*

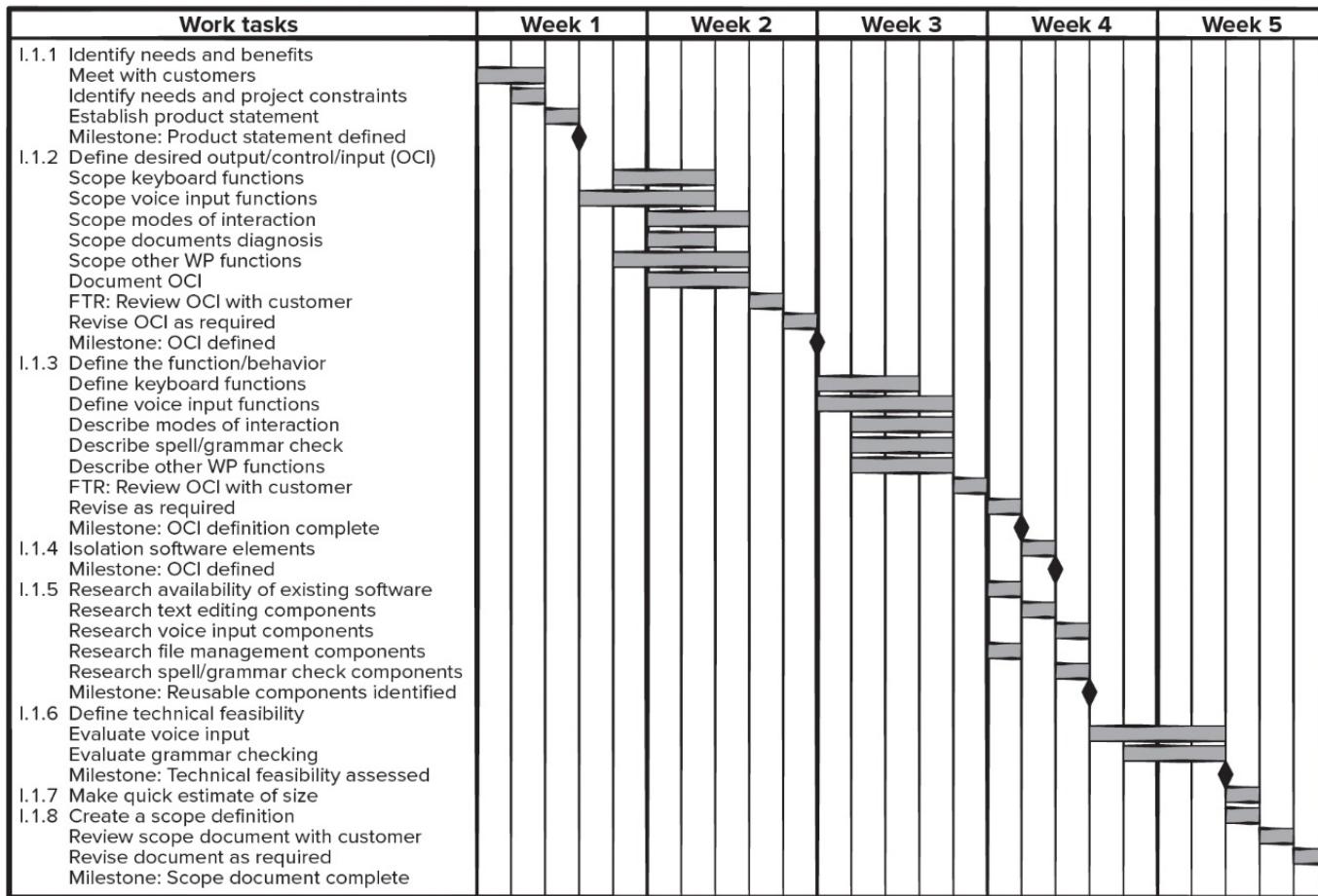
# Task Network (Activity Network)

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# Timeline Chart (Gantt Chart)

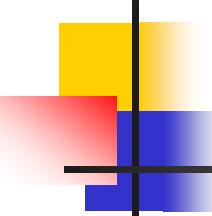
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



# Project Table for Project Tracking

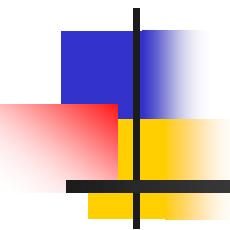
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

Work tasks	Planned start	Actual start	Planned complete	Actual complete	Assigned person	Effort allocated	Notes
I.1.1 Identify needs and benefits Meet with customers Identify needs and project constraints Establish product statement Milestone: Product statement defined	wk1, d1 wk1, d2 wk1, d3 wk1, d3	wk1, d1 wk1, d2 wk1, d3 wk1, d3	wk1, d2 wk1, d2 wk1, d3 wk1, d3	wk1, d2 wk1, d2 wk1, d3 wk1, d3	BLS JPP BLS/	2 p-d 1 p-d 1 p-d	Scoping will require more effort/time
I.1.2 Define desired output/control/input (OCI) Scope keyboard functions Scope voice input functions Scope modes of interaction Scope documents diagnosis Scope other WP functions Document OCI FTR: Review OCI with customer Revise OCI as required Milestone: OCI defined	wk1, d4 wk1, d3 wk2, d1 wk2, d1 wk1, d4 wk1, d4 wk2, d1 wk2, d3 wk2, d4 wk2, d5	wk1, d4 wk1, d3 wk2, d1 wk2, d1 wk1, d4 wk1, d4 wk2, d1 wk2, d3 wk2, d4 wk2, d5	wk2, d2 wk2, d2 wk2, d3 wk2, d2 wk2, d3 wk2, d3 wk2, d3 wk2, d3 wk2, d4 wk2, d5		BLS JPP MLL BLS JPP MLL all all	1.5 p-d 2 p-d 1 p-d 1.5 pd 2 p-d 3 p-d 3 p-d 3 p-d	
I.1.3 Define the Function/behavior							



# Schedule Tracking

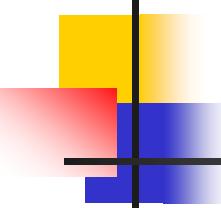
- Conducting periodic project status meetings in which each team member reports progress and problems.
- Evaluating the results of all reviews conducted throughout the software engineering process.
- Determining whether formal project milestones have been accomplished by the scheduled date.
- Comparing the actual start date to the planned start date for each project task listed in the project resource table.
- Meeting informally with practitioners to obtain their subjective assessment of progress to date and problems on the horizon.
- Tracking the project velocity, which is a way of seeing how quickly the development team is clearing the user story backlog.



# *COMP 354: Introduction to Software Engineering*

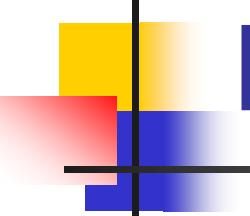
## Risk Management

Based on Chapter 26 of the textbook



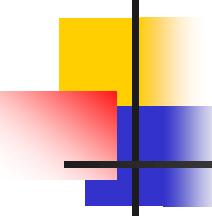
# Reactive Risk Management

- Project team reacts to risks when they occur.
- Mitigation—plan for additional resources in anticipation of fire fighting.
- Fix on failure—resource are found and applied when the risk strikes.
- Crisis management—failure does not respond to applied resources and project is in jeopardy.



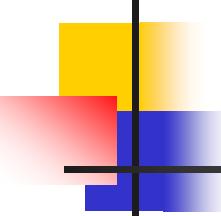
# Proactive Risk Management

- Potential risks are identified, their probability and impact are assessed, and they are ranked by importance.
- Software team establishes a plan for managing risk.
- Primary objective is to avoid risk, but because not all risks can be avoided.
- Team works to develop a contingency plans that will enable it to respond in a controlled and effective manner.
- Proactive risk management is a software engineering tools that can be used to reduce technical debt.



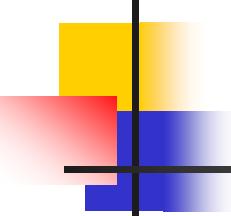
# Software Risks

- **Project risks** threaten the project plan.
- **Technical risks** threaten the quality and timeliness of the software to be produced.
- **Business risks** threaten the viability of the software to be built and often jeopardize the project or the product.
- **Known risks** are those that can be uncovered after careful evaluation of the project plan.
- **Predictable risks** are extrapolated from past project experience.
- **Unpredictable risks** can and do occur, but they are extremely difficult to identify in advance.



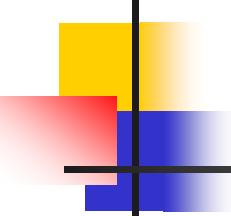
# Risk Management Principles

- **Maintain a global perspective** - view software risks within the context of system and the business problem.
- **Take a forward-looking view** - think about the risks that may arise in the future; establish contingency plans.
- **Encourage open communication** - if someone states a potential risk, don't discount it.
- **Integrate** - a consideration of risk must be integrated into the software process.



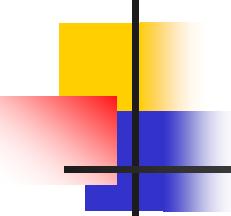
# Risk Management Principles

- **Emphasize a continuous process** - team must be vigilant throughout the software process, modifying identified risks as more information is known and adding new ones as better insight is achieved.
- **Develop a shared product vision** - if all stakeholders share the same vision of the software, it is likely that better risk identification and assessment will occur.
- **Encourage teamwork** - the talents, skills and knowledge of all stakeholder should be pooled.



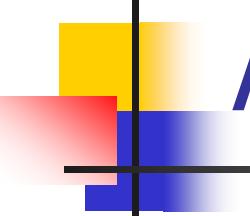
# Risk Identification

- **Product size** - risks associated with the overall size of the software to be built or modified.
- **Business impact** - risks associated with constraints imposed by management or the marketplace.
- **Customer characteristics** - risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.
- **Process definition** - risks associated with the degree to which the software process has been defined and is followed by the development organization.



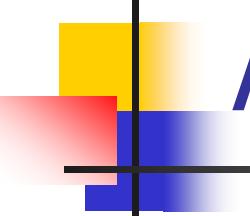
# Risk Identification

- **Development environment** - risks associated with the availability and quality of the tools to be used to build the product.
- **Technology to be built** - risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.
- **Staff size and experience** - risks associated with the overall technical and project experience of the software engineers who will do the work.



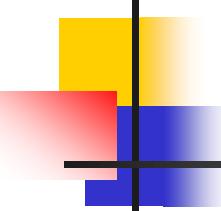
# Assessing Project Risk

- Have top software and customer managers formally committed to support the project?
- Are end-users enthusiastically committed to the project and the system/product to be built?
- Are requirements fully understood by the software engineering team and their customers?
- Have customers been involved fully in the definition of requirements?
- Do end-users have realistic expectations?
- Is project scope stable?



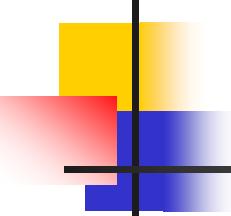
# Assessing Project Risk

- Does the software engineering team have the right mix of skills?
- Are project requirements stable?
- Does the project team have experience with the technology to be implemented?
- Is the number of people on the project team adequate to do the job?
- Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?



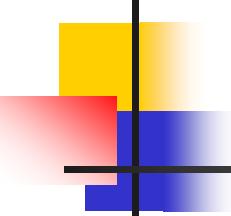
# Risk Components

- **Performance risk** - the degree of uncertainty that the product will meet its requirements and be fit for its intended use.
- **Cost risk** - the degree of uncertainty that the project budget will be maintained.
- **Support risk** - the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.
- **Schedule risk** - the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.



# Risk Projection (Risk Estimation)

- Risk projection attempts to rate each risk in two ways:
  - Likelihood or probability that the risk is real.
  - Consequences of the problems associated with the risk,
- There are four risk projection steps:
  1. Establish a scale that reflects the perceived likelihood of a risk.
  2. Delineate the consequences of the risk.
  3. Estimate the impact of the risk on the project and the product,
  4. Note the overall accuracy of the risk projection so that there will be no misunderstandings.



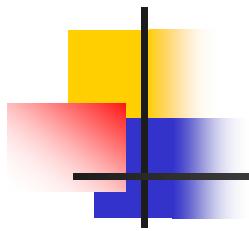
# Risk Table

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

Risk	Category	Probability	Impact	RMMM
Size estimate may be significantly low	PS	60%	2	
Larger number of users than planned	PS	30%	3	
Less reuse than planned	PS	70%	2	
End users resist system	BU	40%	3	
Delivery deadline will be tightened	BU	50%	2	
Funding will be lost	CU	40%	1	
Customer will change requirements	PS	80%	2	
Technology will not meet exceptions	TR	30%	1	
Lack of training on tools	DE	80%	3	
Staff inexperienced	ST	30%	2	
Staff turnover will be high	ST	60%	2	

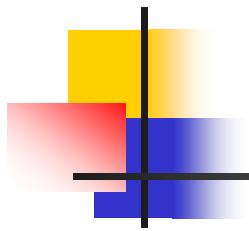
Impact values:

- 1 – catastrophic
- 2 – critical
- 3 – marginal
- 4 – negligible



# Building Risk Table

- Estimate the probability of occurrence.
- Estimate the impact on the project on a scale of 1 to 5, where,
  - 1 = low impact on project success
  - 5 = catastrophic impact on project success
- Sort the table by probability and impact.



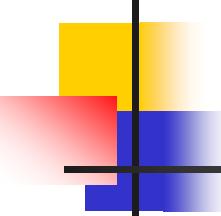
# Risk Impact (Exposure)

The overall risk exposure, RE, is determined using the following relationship [Hal98]:

$$RE = P \times C$$

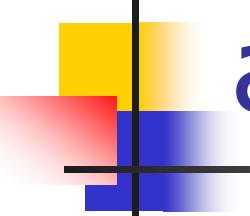
where

$P$  is the probability of occurrence for a risk, and  
 $C$  is the cost to the project should the risk occur.



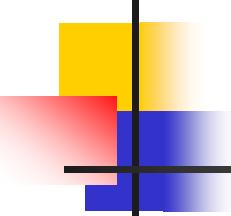
# Risk Exposure Example

- **Risk identification.** Only 70 percent of the software components scheduled for reuse will be used, the rest will have to be custom developed.
- **Risk probability.** 80% (likely).
- **Risk impact.** 60 reusable software components were planned. If only 70 percent can be used, 18 components would have to be developed from scratch. The average component is 100 L O C and the software engineering cost for each L O C is \$14.00, the overall cost (impact) to develop the components is  $18 \times 100 \times 14 = \$25,200$ .
- **Risk exposure.**  $RE = 0.80 \times 25,200 \square \$20,200$ .



# Risk Mitigation, Monitoring, and Management

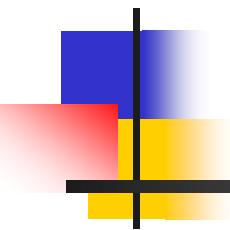
- **Mitigation** - how can we avoid the risk?
- **Monitoring** - what factors can we track that will enable us to determine if the risk is becoming more or less likely?
- **Management** - what contingency plans do we have if the risk becomes a reality?



# Risk Information Sheet

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

Risk information sheet			
Risk ID: P02-4-32	Date: 5 / 9 / 19	Prob: 80%	Impact: high
<b>Description:</b> Only 70 percent of the software components scheduled for reuse will, in fact, be integrated into the application. The remaining functionality will have to be custom developed.			
<b>Refinement/context:</b> Subcondition 1: Certain reusable components were developed by a third party with no knowledge of internal design standards. Subcondition 2: The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components. Subcondition 3: Certain reusable components have been implemented in a language that is not supported on the target environment.			
<b>Mitigation/monitoring:</b> <ol style="list-style-type: none"><li>1. Contact third party to determine conformance with design standards.</li><li>2. Press for interface standards completion; consider component structure when deciding on interface protocol.</li><li>3. Check to determine number of components in subcondition 3 category; check to determine if language support can be acquired.</li></ol>			
<b>Management/contingency plan/trigger:</b> RE computed to be \$20,200. Allocate this amount within project contingency cost. Develop revised schedule assuming that 18 additional components will have to be custom built; allocate staff accordingly. Trigger: Mitigation steps unproductive as of 7 / 1 / 19.			
<b>Current status:</b> 5 / 12 / 19: Mitigation steps initiated.			
Originator: D. Gagne		Assigned: B. Laster	



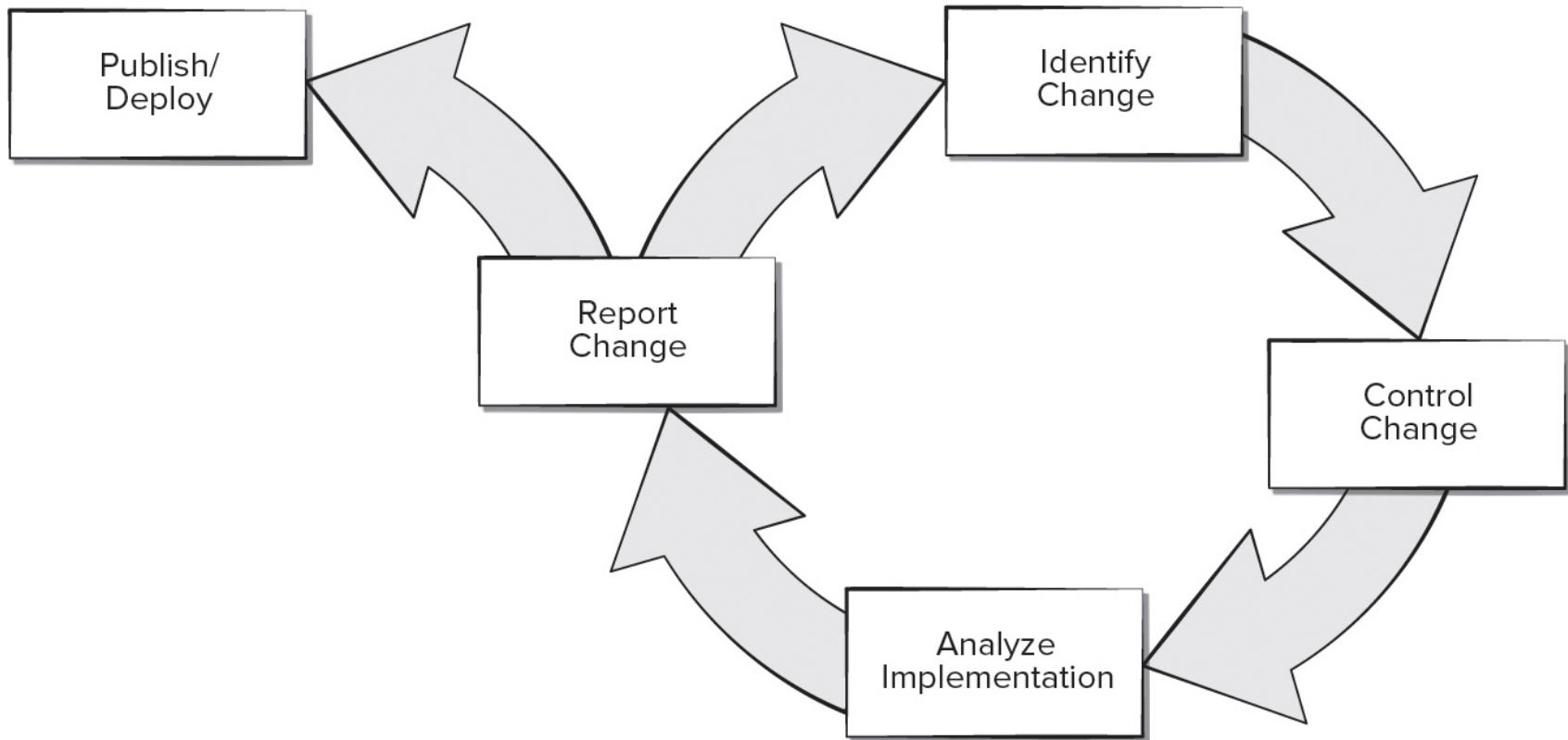
# *COMP 354: Introduction to Software Engineering*

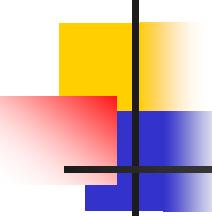
## Configuration Management

Based on Chapter 22 of the textbook

# Software Configuration Management Work Flow

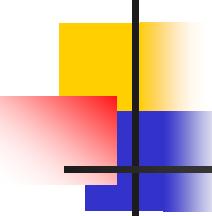
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Configuration Management System Elements

- **Component elements.** A set of tools coupled within a file management system (for example, a database) that enables access to and management of each software configuration item.
- **Process elements.** A collection of procedures and tasks that define an effective approach to change management (and related activities) for all constituencies involved in the management, engineering, and use of computer software.
- **Construction elements.** A set of tools that automate the construction of software by ensuring that the proper set of validated components (that is, the correct version) have been assembled.
- **Human elements.** A set of tools and process features (encompassing other CM elements) used by the software team to implement effective SCM.

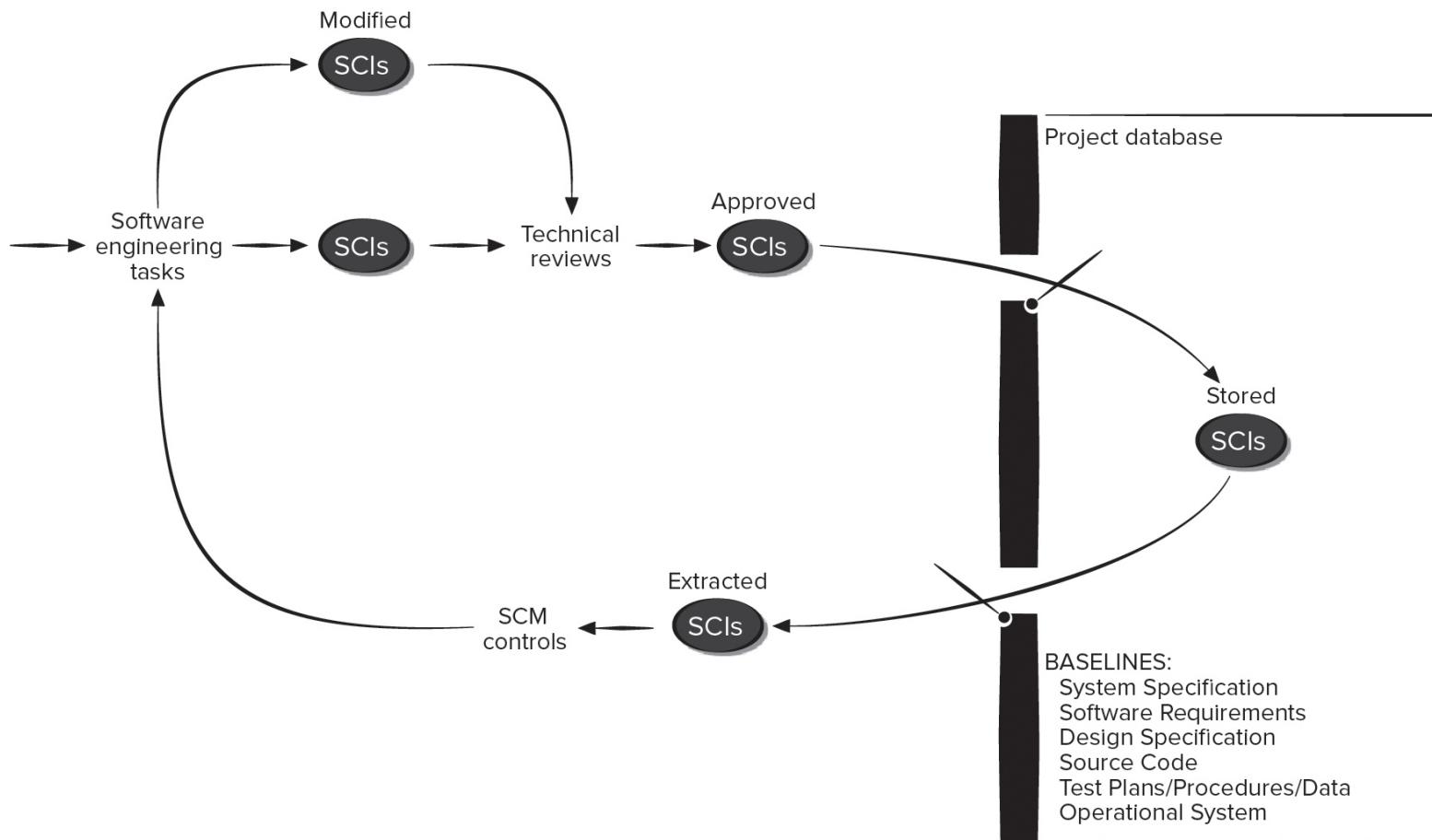


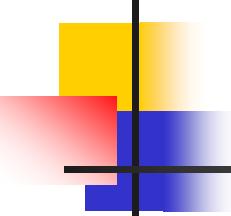
# Baselines

- The IEEE defines a baseline as:  
A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
- A baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCI that is obtained through a formal technical review.

# Baselined Software Configuration Items

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



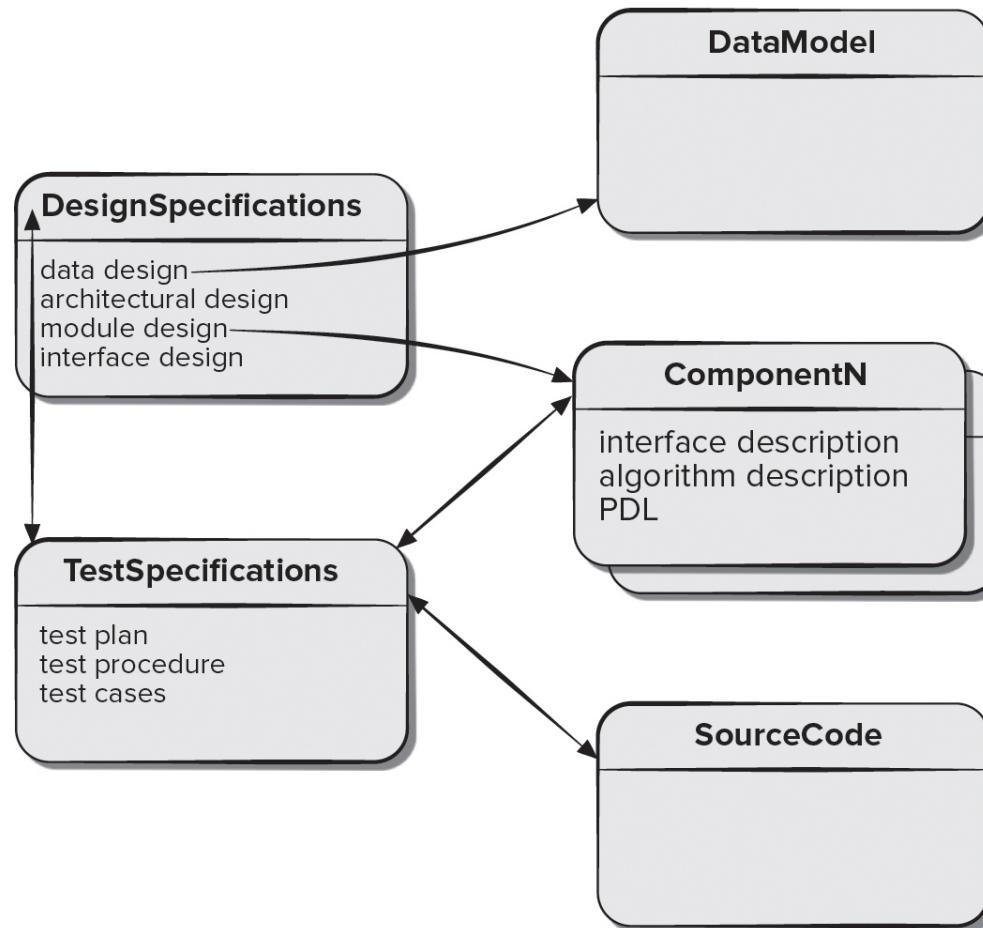


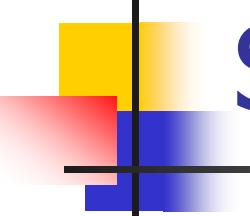
# Management of Dependencies and Changes

- It is important for developers to maintain software work products to ensure the dependencies among the SCI are documented.
- Developers must establish discipline when checking items in and out of the SCM repository.
- Impact management involves two complementary aspects:
  1. Ensuring that developers employ strategies to minimize the impact of their colleagues' actions on their own work.
  2. Encouraging software developers to use practices that minimize the impact of their own work on that of their colleagues.

# Software Configuration Items

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





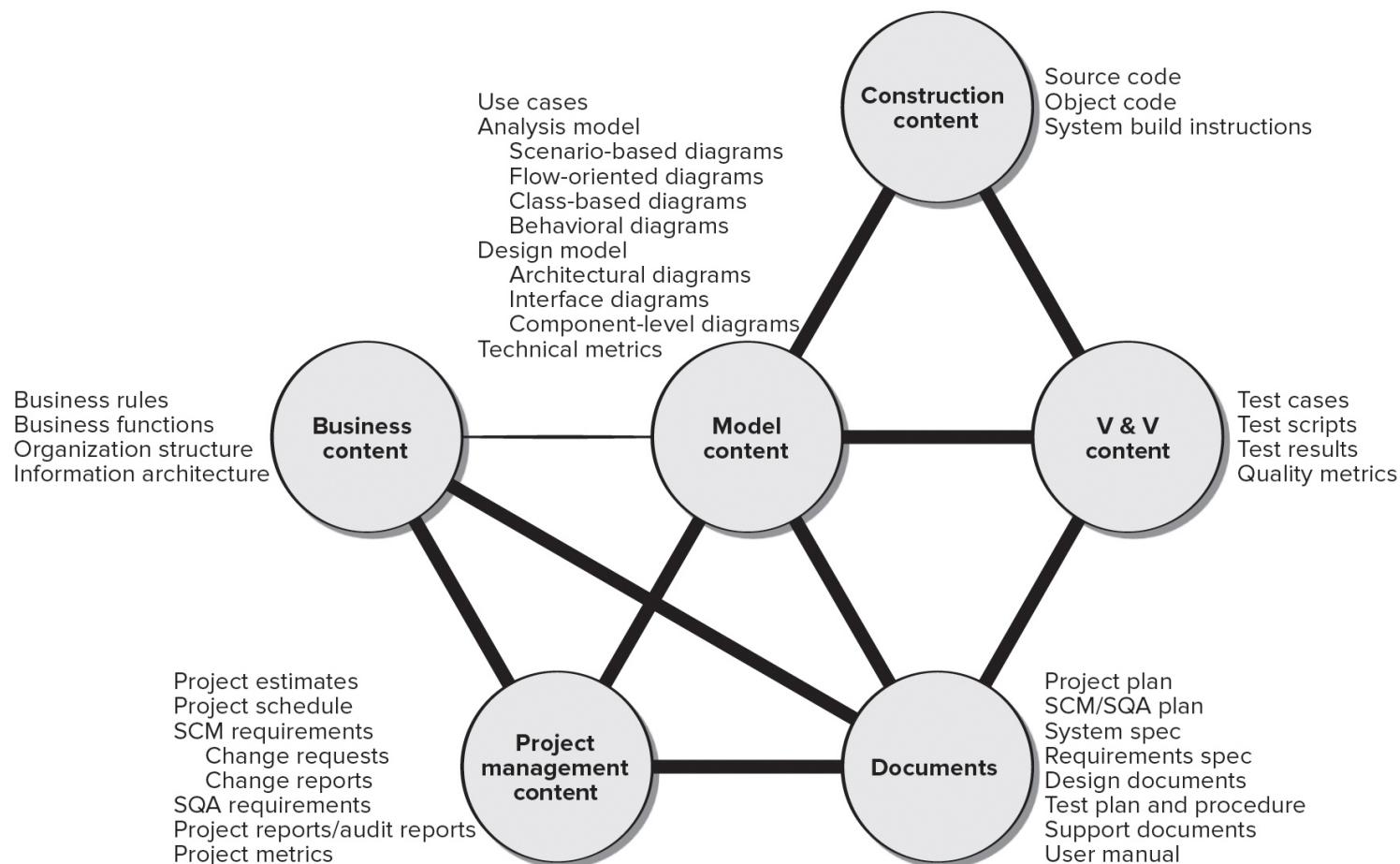
# SCM Repository

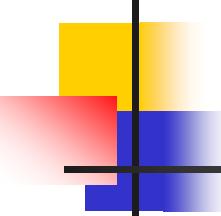
The SCM repository is the set of mechanisms and data structures that provides the following functions that allow a software team to manage change:

- Data integrity.
- Information sharing.
- Tool integration.
- Data integration.
- Methodology enforcement.
- Document standardization.

# SCM Repository

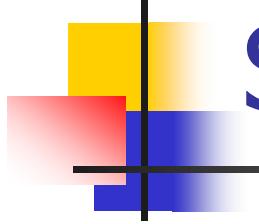
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# SCM Repository Features

- **Versioning.** - saves versions to manage product releases and allow developers to go back to previous versions.
- **Dependency tracking and change management.** - manages a wide variety of relationships among the data elements stored in it.
- **Requirements tracing.** - provides the ability to track all design and construction components and deliverables resulting from a specific requirement specification.
- **Configuration management.** - tracks series of configurations representing specific project milestones or production releases and provides version management.
- **Audit trails.** - establishes additional information about when, why, and by whom changes are made.



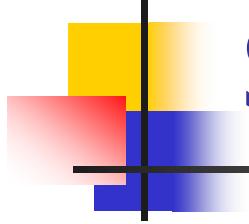
# SCM Best Practices

- Keeping the number of code variants small.
- Test early and often.
- Integrate early and often.
- Use tools to automate testing, building, and code integration.

# Continuous Integration

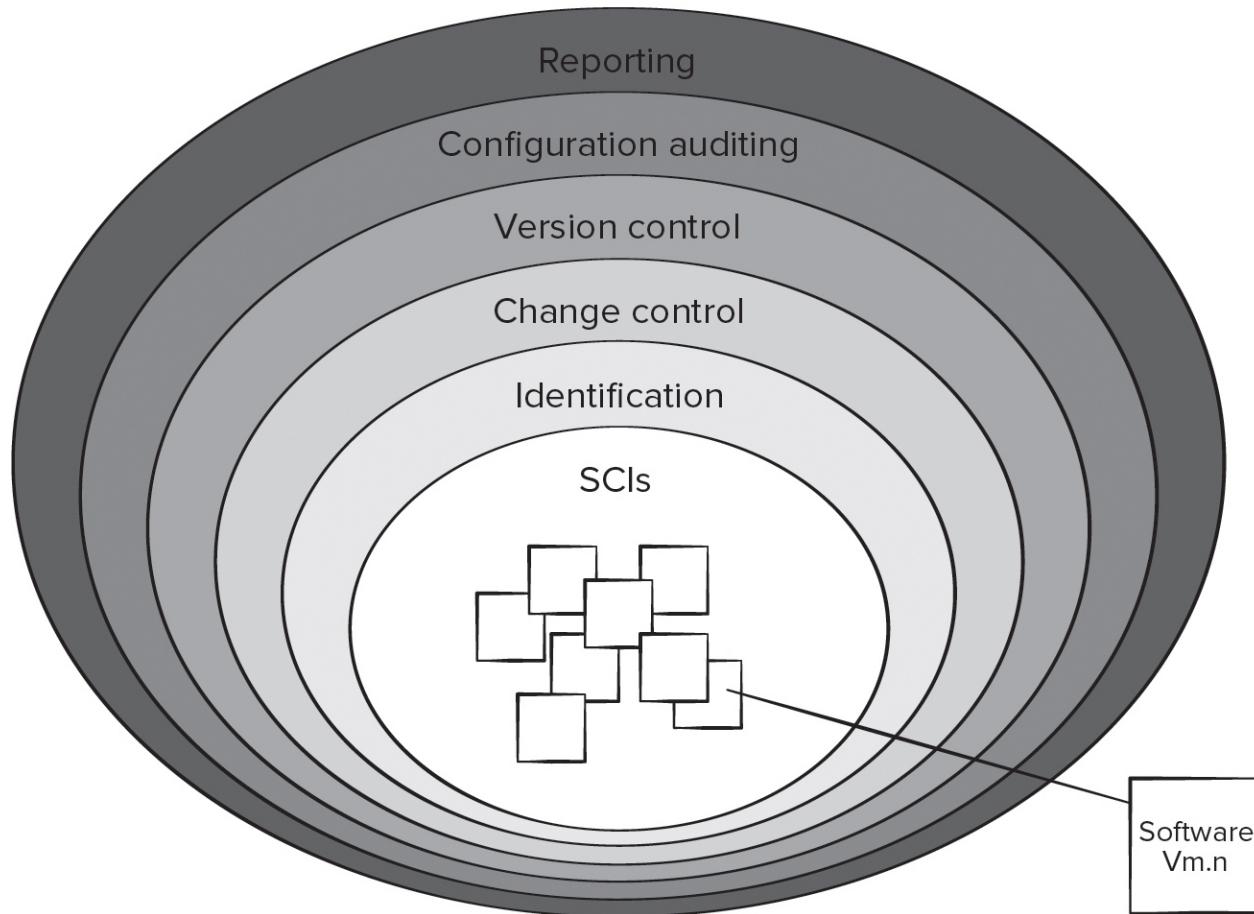
## Advantages

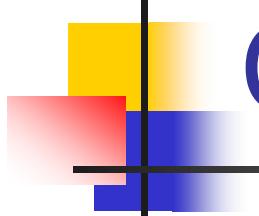
- **Accelerated feedback.** Notifying developers immediately when integration fails, allows fixes when the number of changes is small.
- **Increased quality.** Building and integrating software whenever necessary, provides confidence into the quality of the product.
- **Reduced risk.** Integrating components early avoids a long integration phase, design failures are discovered and fixed early.
- **Improved reporting.** Providing additional information (for example, code analysis metrics), allows for accurate configuration status accounting.
- CI is becoming a key technology as software organizations begin their shift to more agile software development processes.
- Early defect capture always reduces the development costs.



# SCM Process Layers

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





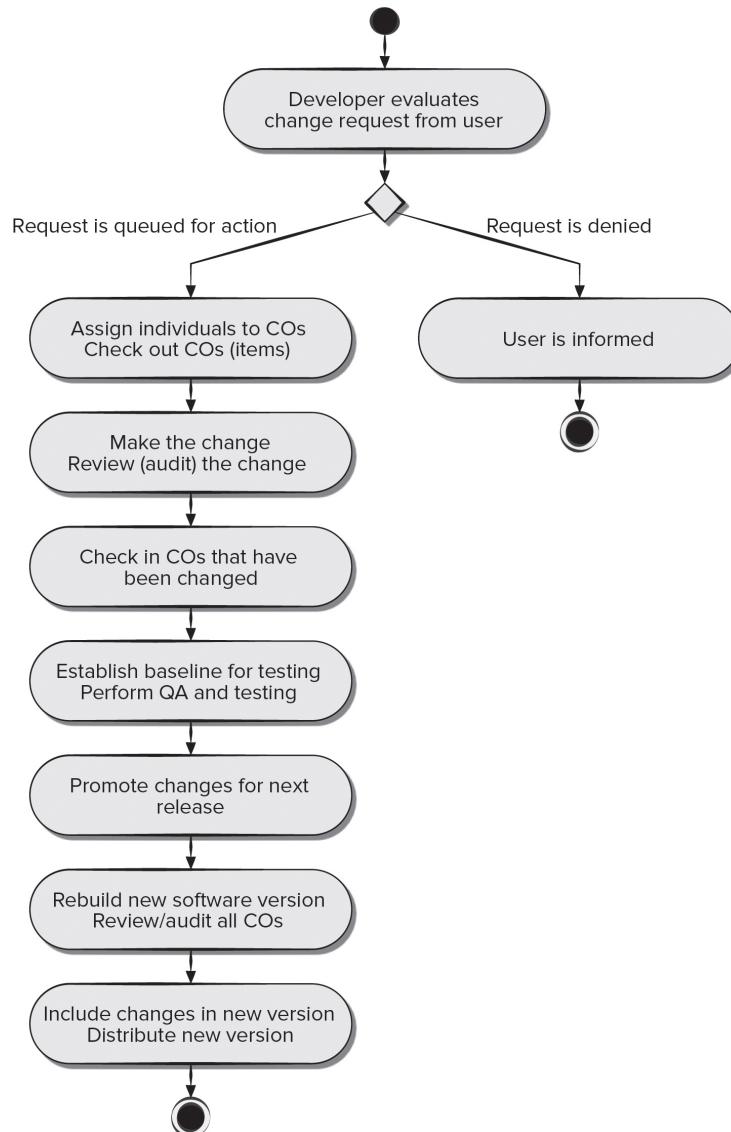
# Change Management Objectives

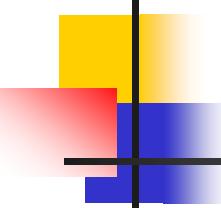
Software change management process defines a series of tasks that have four primary objectives:

1. To identify all items that collectively define the software configuration.
2. To manage changes to one or more of these items.
3. To facilitate the construction of different versions of an application.
4. To ensure that software quality is maintained as the configuration evolves over time.

# Change Control Process

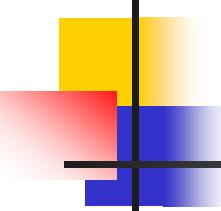
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Impact Management

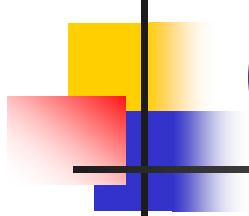
- A web of software work product interdependencies must be considered every time a change is made.
- Impact management is accomplished with three actions:
  1. An impact network identifies the stakeholders who might effect or be affected by changes that are made to the software based on its architectural documents.
  2. Forward impact management assesses the impact of your own changes on the members of the impact network and then informs members of the impact of those changes.
  3. Backward impact management examines changes that are made by other team members and their impact on your work and incorporates mechanisms to mitigate the impact.



# Software Configuration Audit

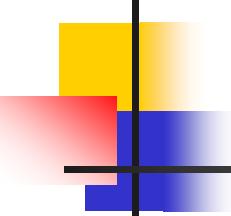
**Software configuration audit** complements a technical review by asking and answering the following questions:

- Has the change specified in the ECO been made? Have any additional modifications been incorporated?
- Has a technical review been conducted to assess technical correctness?
- Has the software process been followed, and have software engineering standards been properly applied?
- Has the change been “highlighted” in the SCI? Do the attributes of the configuration object reflect the change?
- Have SCM procedures for noting the change, recording it, and reporting it been followed?
- Have all related SCI been properly updated?



# Configuration Status Reporting (CSR)

- Configuration status reporting is an SCM task that answers the following questions any time a change or audit occurs:
  - What happened?
  - Who did it?
  - When did it happen?
  - What else will be affected?
- Output from CSR may be placed in an online database or website, so that software developers or support staff can access change information by keyword category.

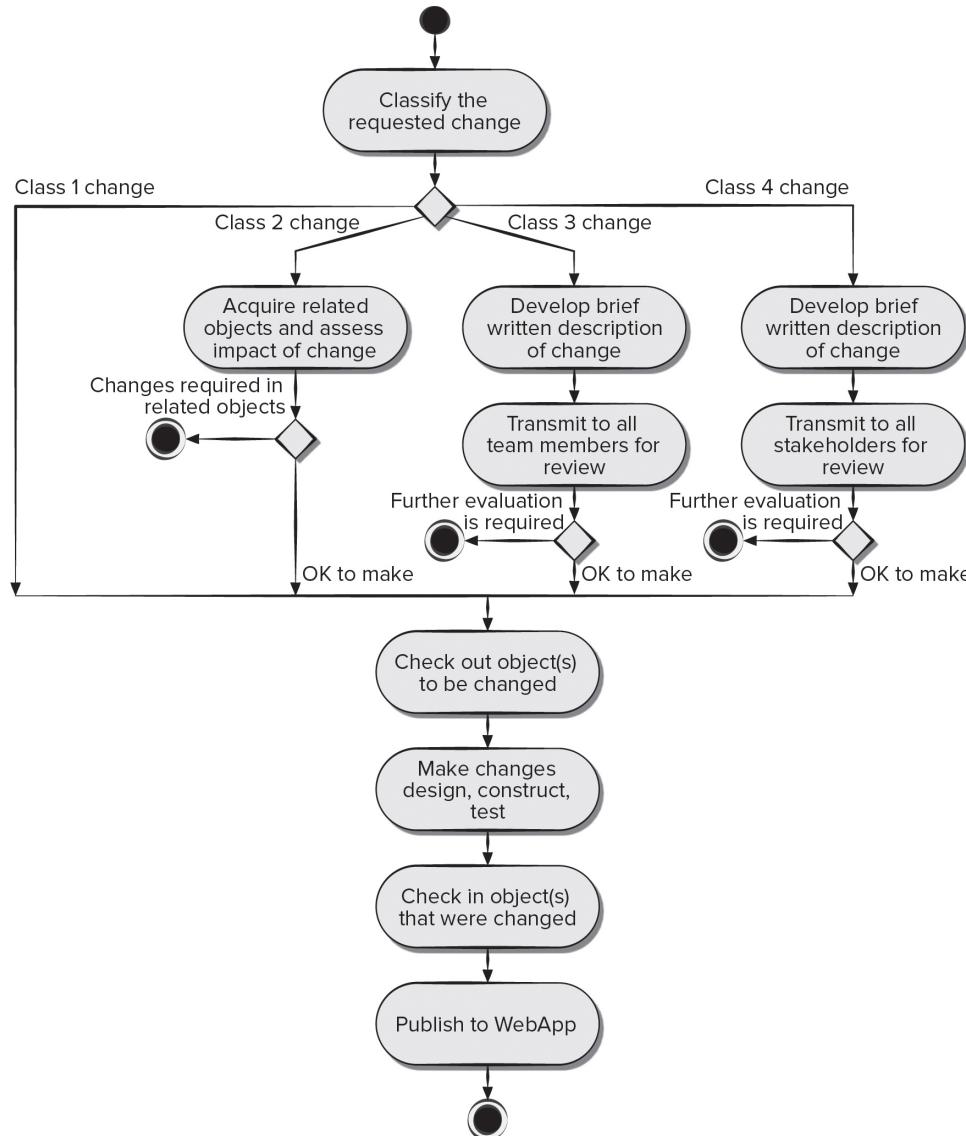


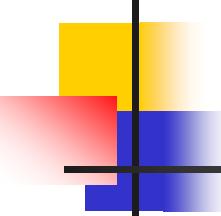
# Mobility and Agile Change

- Mobile developers and game developers use an iterative, incremental process model that applies many agile principles.
- An engineering team often develops an increment in a very short time period using a customer-driven approach.
- Subsequent increments add additional content and functionality, and each is likely to implement changes that lead to enhanced content, better usability, improved aesthetics, better navigation, enhanced performance, and stronger security.
- Members of software teams that build apps or games must embrace change, but do not want excessive bureaucracy.
- Traditional SCM principles, practices, and tools must be molded them to meet the special needs of mobile projects.

# Agile Change Management

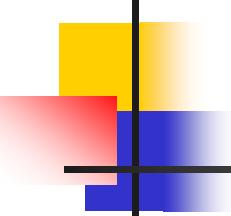
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Content Management

- **Collection subsystem** encompasses all actions required to create or acquire content, and the technical functions that are necessary to:
  - Convert content into a form that can be represented by a mark-up language (for example, HTML, XML).
  - Organize content into packets that can be displayed effectively on the client-side.
- **Management subsystem** implements a repository that encompasses:
  - Content database - information structure to store all content objects.
  - Database capabilities - functions to search for content objects, store and retrieve objects, and manage the content file structure.
  - Configuration management functions - supports content object identification, version control, change management, change auditing, and reporting.

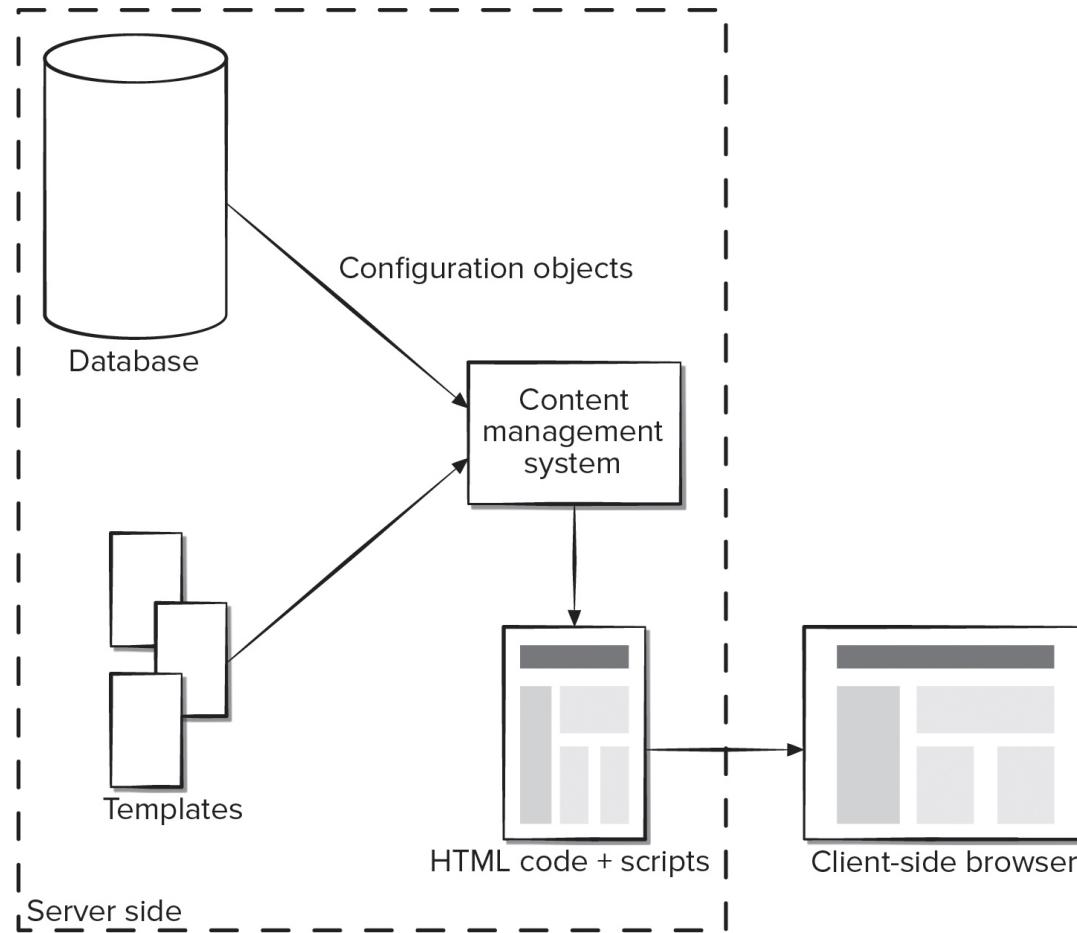


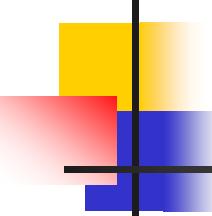
# Content Management

- **Publishing subsystem** - extracts content from the repository, converts it to a publishable form, and formats it so that it can be transmitted to client-side browsers.
- The publishing subsystem uses a series of templates for each type:
  - *Static elements*—text, graphics, media, and scripts that require no further processing are transmitted directly to the client-side.
  - *Publication services*—function calls to specific retrieval and formatting services that personalize content (using predefined rules), perform data conversion, and build appropriate navigation links.
  - *External services*—provide access to external corporate information infrastructure such as enterprise data or “back-room” applications.

# Content Management System

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

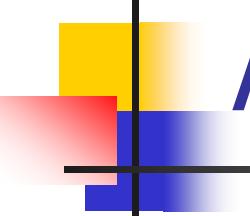




# Version Control

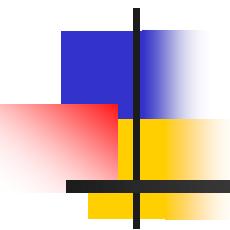
As apps and games evolve through a series of increments different versions may exist at the same time and version control process is required to avoid overwriting changes:

- A central repository for the app or game project should be established.
- Each developer creates his own working folder.
- The clocks on all developer workstations should be synchronized.
- As new configuration objects are developed or existing objects are changed, they are imported into the central repository.
- As objects are imported or exported from the repository, an automatic, time-stamped log message is made.



# Auditing and Reporting

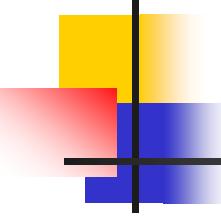
- Interest of agility, auditing and reporting functions are deemphasized during development of games or apps.
- As objects are checked into or out of the repository the action is recorded in a log that can be reviewed.
- Complete log report can be generated so that all members of the team have a chronology of changes.
- An e-mail notification can be generated automatically any time an object is checked in or out of the repository.



# *COMP 354: Introduction to Software Engineering*

## Software Quality Assurance

Based on Chapter 17 of the textbook

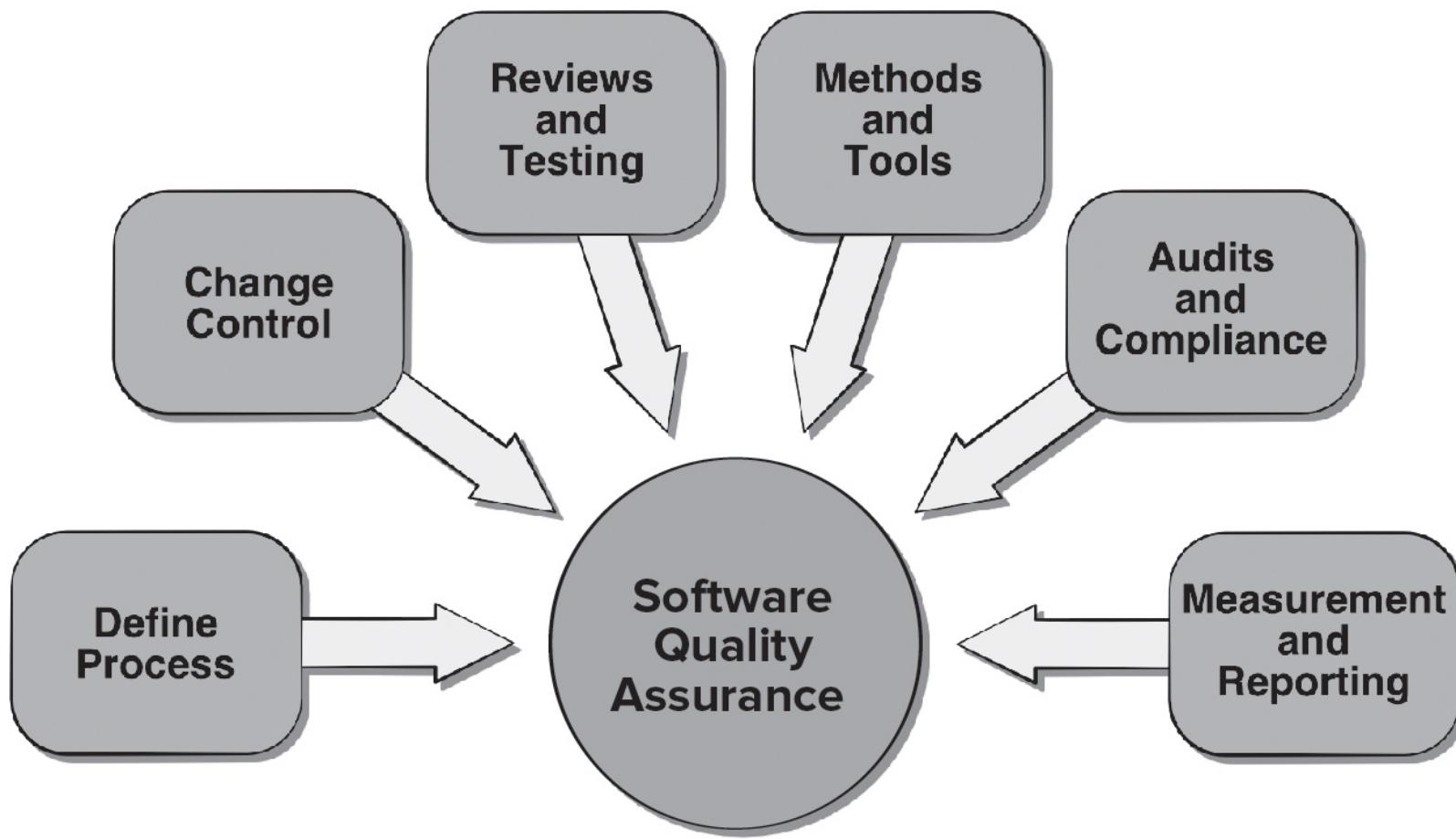


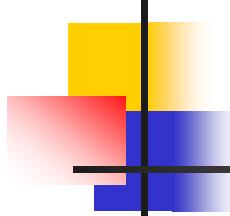
# Quality Management

- The problem of quality management is not what people don't know about it. The problem is what they think they do.
- Everybody is for it. (Under certain conditions, of course.)
- Everyone feels they understand it. (Even though they wouldn't want to explain it.)
- Everyone thinks execution is only a matter of following natural inclinations. (After all, we do get along somehow.)
- Most people feel that problems in these areas are caused by other people. (If only they would take the time to do things right.)

# Software Quality Assurance

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





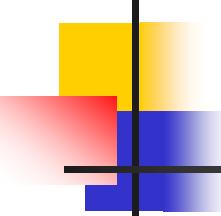
# Elements of SQA

- Standards.
- Reviews and Audits.
- Testing.
- Error/defect collection and analysis.
- Change management.
- Education.
- Vendor management.
- Security management.
- Safety.
- Risk management.

# Data Driven SQA

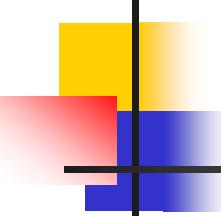
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





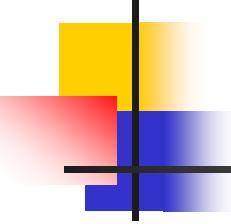
# Role of SQA Group

- Prepares an SQA plan for a project which identifies:
  - Evaluations to be performed.
  - Audits and reviews to be performed.
  - Standards that are applicable to the project.
  - Procedures for error reporting and tracking.
  - Documents to be produced by the SQA group.
  - Amount of feedback provided to the software project team.
- Participates in the development of the project's software process description.
  - Reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (for example, ISO-9001), and other parts of the software project plan.



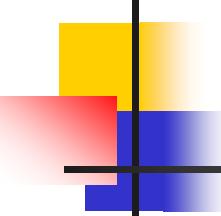
# Role of SQA Group

- Reviews software engineering activities to verify compliance with the defined software process.
  - Identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- Audits designated software work products to verify compliance with those defined as part of the software process.
  - Reviews selected work products; identifies, documents, and tracks deviations; verifies that corrections have been made.
  - Periodically reports the results of its work to the project manager.
- Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
- Records any noncompliance and reports to senior management.
  - Noncompliance items are tracked until they are resolved.



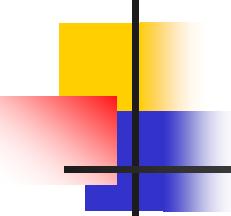
# SQA Goals

- **Requirements quality.** The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products.
- **Design quality.** Every element of the design model should be assessed to ensure that it exhibits high quality and that the design itself conforms to requirements.
- **Code quality.** Source code and related work products must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
- **Quality control effectiveness.** A software team should apply limited resources in a way that has the highest likelihood of achieving a high quality result.



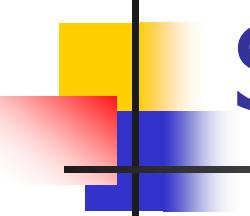
# Formal SQA

- Assumes that a rigorous syntax and semantics can be defined for every programming language.
- Allows the use of a rigorous approach to the specification of software requirements.
- Applies mathematical proof of correctness techniques to demonstrate that a program conforms to its specification.
- Although formal methods are interesting to some software engineering researchers, most commercial developers rarely use of formal methods.



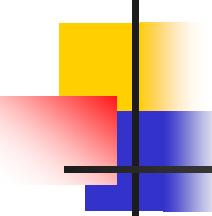
# Statistical SQA

- Information about software errors and defects is collected and categorized.
- An attempt is made to trace each error and defect to its underlying cause (for example, design error, violation of standards, non-conformance to specifications, poor communication with the customer).
- Using the Pareto principle (80 percent of the defects can be traced to 20 percent of all possible causes), isolate the 20 percent (the vital few).
- Once the vital few causes have been identified, move to correct the problems that caused the errors and defects.



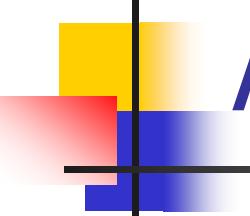
# Six Sigma for Software Engineering

- The term “six sigma” is derived from six standard deviations from the mean - 3.4 instances (defects) per million occurrences - implying an extremely high quality standard.
- The three cores steps:
  - **Define** customer requirements and deliverables and project goals via well-defined methods of customer communication.
  - **Measure** the existing process and its output to determine current quality performance (collect defect metrics).
  - **Analyze** defect metrics and determine the vital few causes.



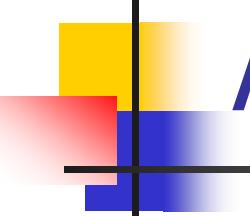
# Six Sigma for Software Engineering

- For an existing process that needs improvement:
  - **Improve** the process by eliminating the root causes of defects.
  - **Control** the process to ensure that future work does not reintroduce the causes of defects.
- For a new process being developed:
  - **Design** the process to: (1) avoid the root causes of defects and (2) to meet customer requirements.
  - **Verify** that the process model will, in fact, avoid defects and meet customer requirements.



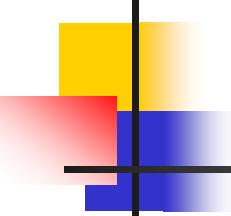
# Software Reliability and Availability

- A simple measure of reliability is mean-time-between-failure (MTBF):  
$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$
  
    MTTF is mean-time-to-failure  
    MTTR is mean-time-to-repair, respectively.
- Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as  
$$\text{Availability} = [\text{MTTF}/(\text{MTTF} + \text{MTTR})] \times 100\%$$



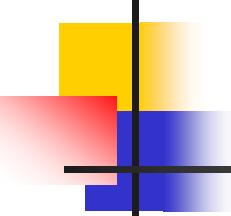
# AI and Reliability Models

- **Software reliability** is the probability of failure-free software operation for a specified time period in a specified environment.
- **Bayesian inference** is a method of statistical inference in which Bayes' theorem is used to update the probability for a hypothesis (such as system reliability) being correct as more evidence or information becomes available.
- Bayesian inference can be used to estimate probabilistic quantities using historic data even when some of the information is missing.
- Making use of predictive data analytics tools such as a regression model involving M T B F can been used to estimate where and what types of defects might occur in future prototypes.
- **Genetic algorithms** can be used to grow reliability models by discovering relationships using historic system data to predict future software component failures.



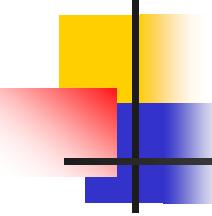
# Software Safety

- **Software safety** is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.



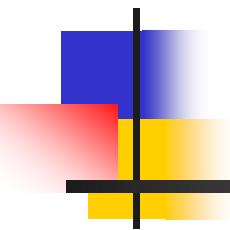
# ISO 9001:2015 Standard

- ISO 9001:2015 is the quality assurance standard that applies to software engineering.
- The requirements delineated by ISO 9001:2008 address topics such as:
- Management responsibility, quality system, contract review, design control, document and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.
- For an organization to become registered to ISO 9001:2015, it must establish procedures to address each of the requirements listed and able to demonstrate these policies and being followed.



# SQA Plan Contents

- Purpose and scope of the plan.
- Description of all software engineering work products that fall within the purview of SQA.
- Applicable standards and practices that are applied during the software process.
- SQA actions and tasks (including reviews and audits) and their placement throughout the software process.
- Tools and methods supporting SQA actions and tasks.
- Software configuration management procedures.
- Methods for safeguarding and maintaining SQA records.
- Organizational roles and responsibilities.



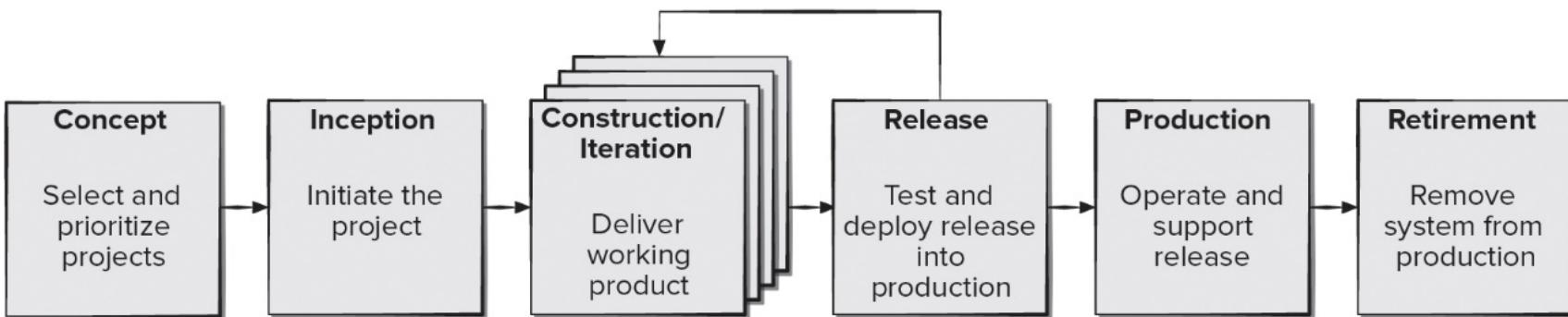
# *COMP 354: Introduction to Software Engineering*

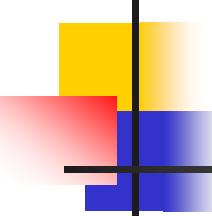
## Software Support

Based on Chapter 26 of the textbook

# Prototype Evolution Process Model

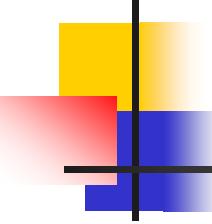
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Lehman's Laws of Software Evolution

- **Law of continuing change (1974).** Software implemented in a real-world will evolve over time and must be continually adapted.
- **Law of increasing complexity (1974).** As a system evolves its complexity increases unless work is done to maintain or reduce it.
- **Law of conservation of familiarity (1980).** As a system evolves all associated with it (all stakeholders) must maintain knowledge of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that knowledge.
- **Law of continuing growth (1980).** The functional content of systems must be continually increased to maintain user satisfaction.
- **Law of declining quality (1996).** The quality of systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.



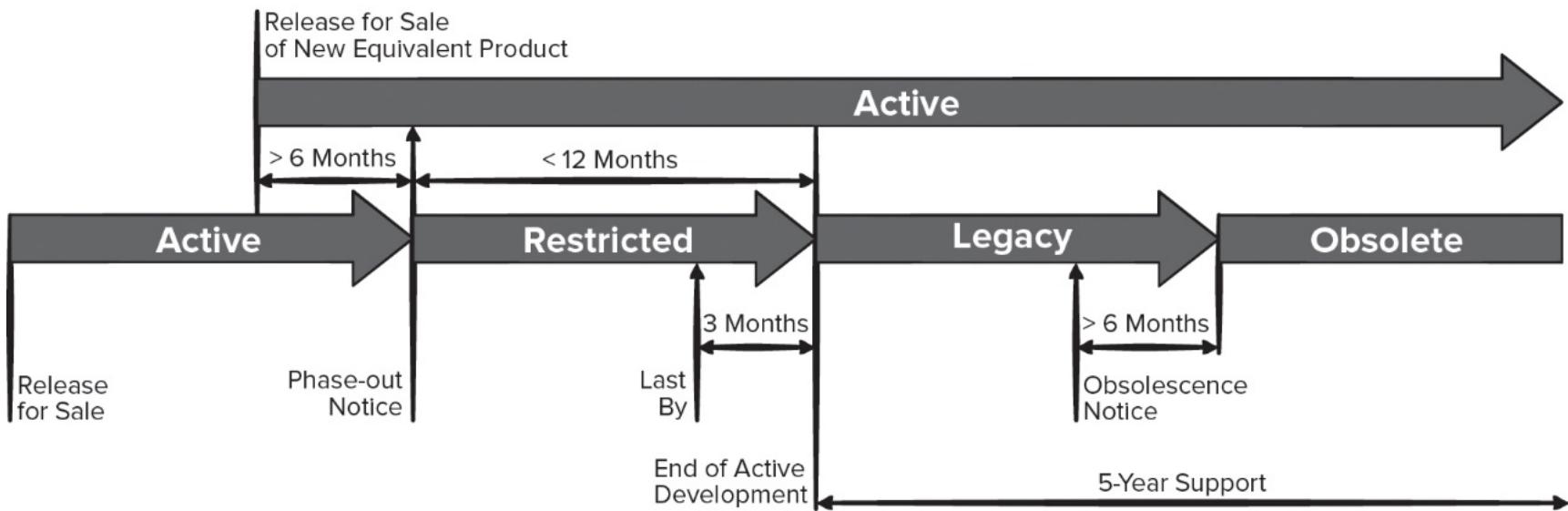
# Software Support

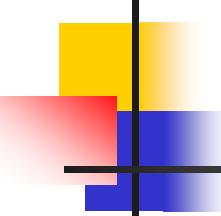
Software support can be considered an umbrella activity that includes:

- Change management.
- Proactive risk management.
- Process management.
- Configuration management.
- Quality assurance.
- Release management.

# Software Release and Retirement

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





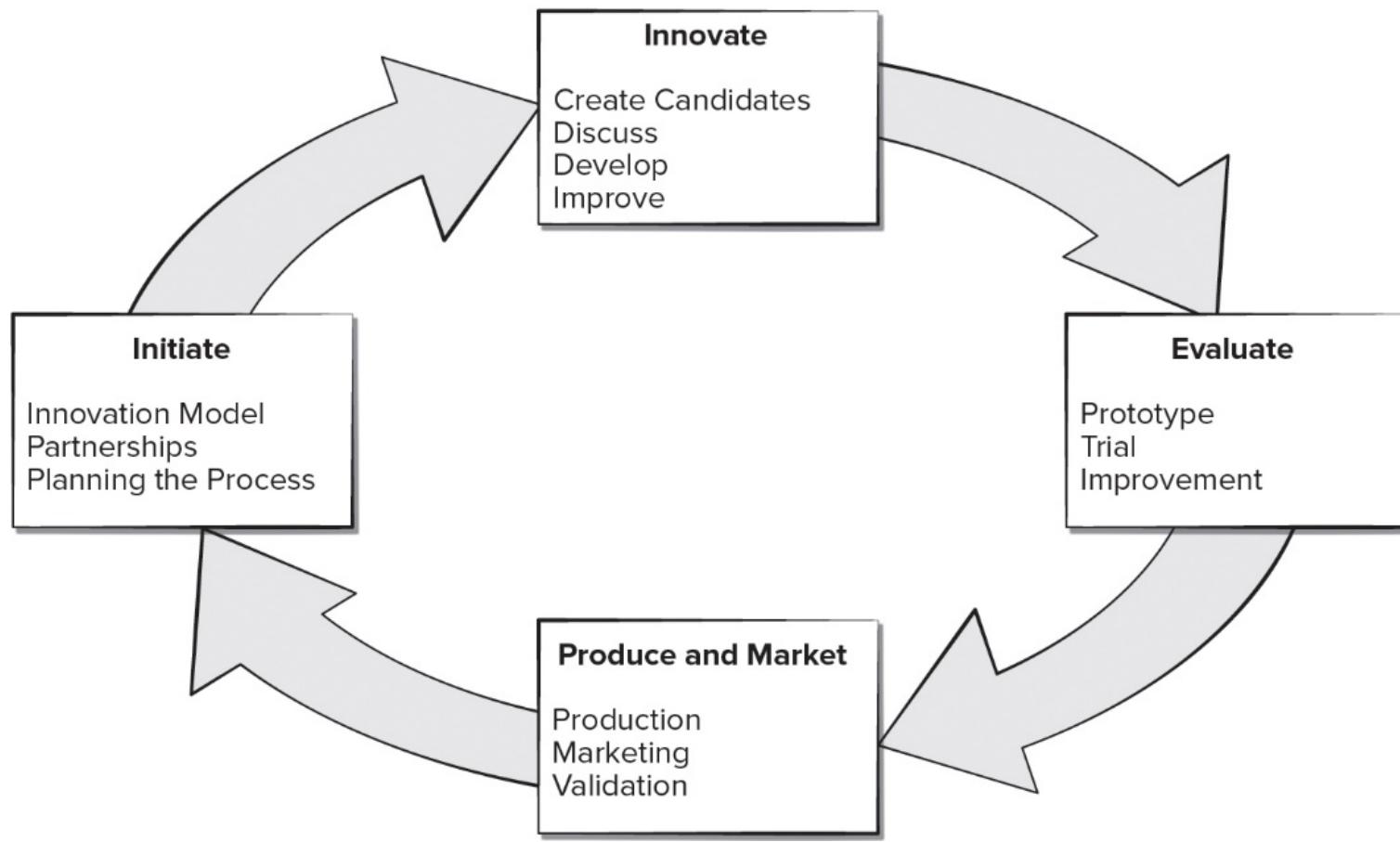
# Release Management

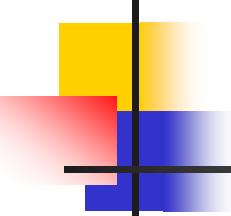
Release management - process that brings high-quality code from developer's workspace to the end user includes:

- Code change integration.
- Continuous integration.
- Build system specifications.
- Infrastructure-as-code.
- Deployment and release.
- Retirement.

# Iterative Software Support Model

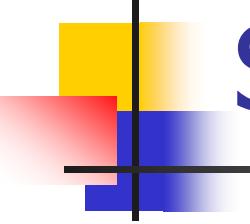
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





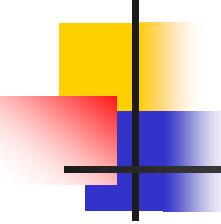
# Software Supportability

- Capability of supporting software over its whole lifetime.
- Implies satisfying all necessary requirements and also the provision of resources, support infrastructure, additional software, facilities, and manpower needed to ensure software is capable of performing its functions.
- Software should contain facilities to assist support personnel when a defect is encountered in the operational environment.
- Support personnel should have access to a database containing records of all defects that have already been encountered—their characteristics, cause, and cure.



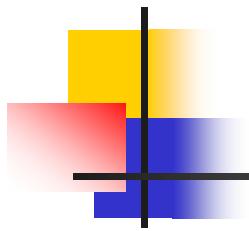
# Software Maintenance

- Software is released to end-users, and:
  - Within days, bug reports filter back to the software engineering organization.
  - Within weeks, one class of users indicates that the software must be changed so that it can accommodate the special needs of their environment.
  - Within months, another corporate group who wanted nothing to do with the software when it was released, now recognizes that it may provide them with benefits and want few enhancements to make it work better for their needs.
- All of this work is **software maintenance**



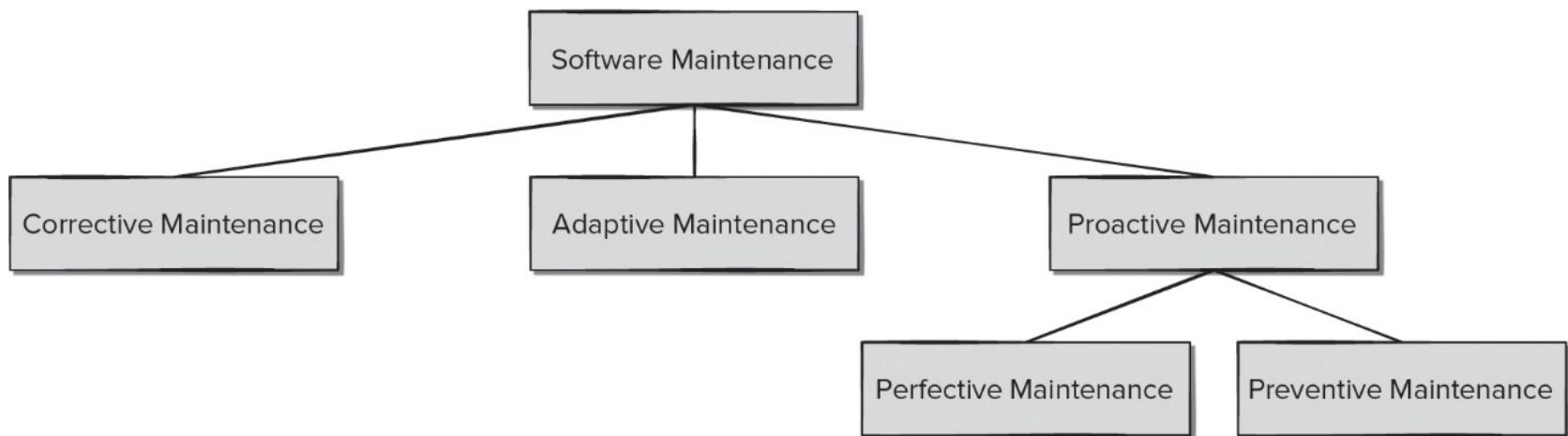
# Maintainable Software

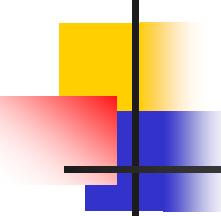
- Maintainable software exhibits effective modularity.
- It makes use of design patterns that allow ease of understanding.
- It has been constructed using well-defined coding standards, leading to understandable source code.
- It has undergone quality assurance techniques that uncovered maintenance problems before release.
- It was created by software engineers who recognize that they may not be around when changes must be made.
- The design and implementation of the software must “assist” the person who is making the change.



# Maintenance Types

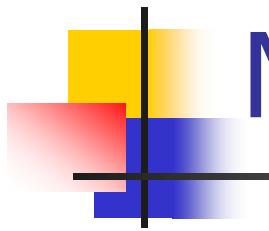
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





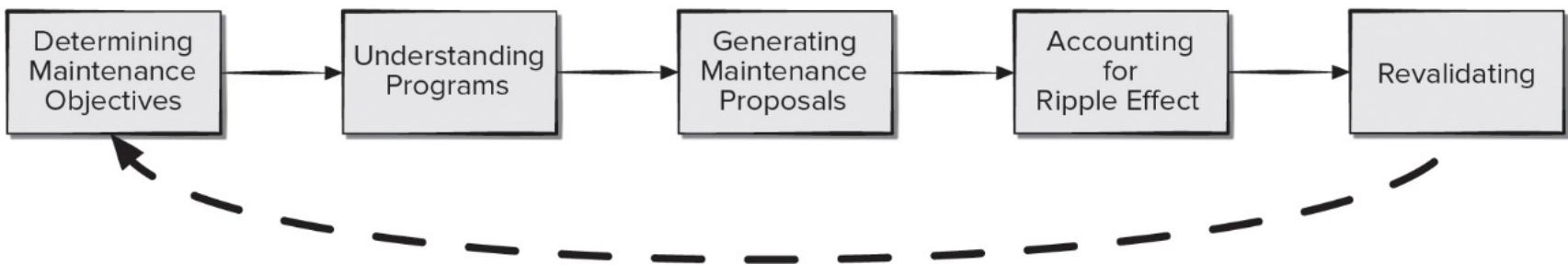
# Maintenance and Support

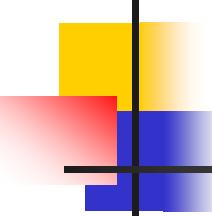
- **Reverse engineering** - process of analyzing a software system to create representations of the system at a higher level of abstraction. Often used to rediscover and redocument system design elements prior to modifying the system source code.
- **Refactoring** - process of changing a software system to improves its internal structure without altering its external behavior. Often used to improve the quality of a software product and make it easier to understand and maintain.
- **Reengineering (evolution)** - process of taking an existing software system and generating a new system that has the same quality as software created using modern software engineering practices.



# Maintenance Tasks

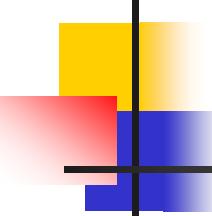
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





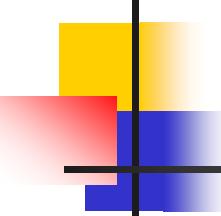
# Agile Maintenance

- Use sprints to organize the maintenance work.  
Balance the goal of keeping customers happy with technical needs of the developers.
- Allow urgent customer requests to interrupt scheduled maintenance sprints, make time for them during sprint planning.
- Facilitate team learning by ensuring that more experienced developers are able to mentor less experienced team members.
- Allow multiple team members to accept customer requests as they coordinate their processing with maintenance team members.



# Agile Maintenance

- Balance the use of written documentation with face-to-face communication to ensure planning meeting time is used wisely.
- Write informal use cases to supplement documentation being used for communications with stakeholders.
- Have developers test each other's work (both defect repairs and new feature implementations).
- Make sure developers are empowered to share knowledge with one another. Motivates people to improve the skills and knowledge.
- Keep planning meetings short, frequent, and focused.

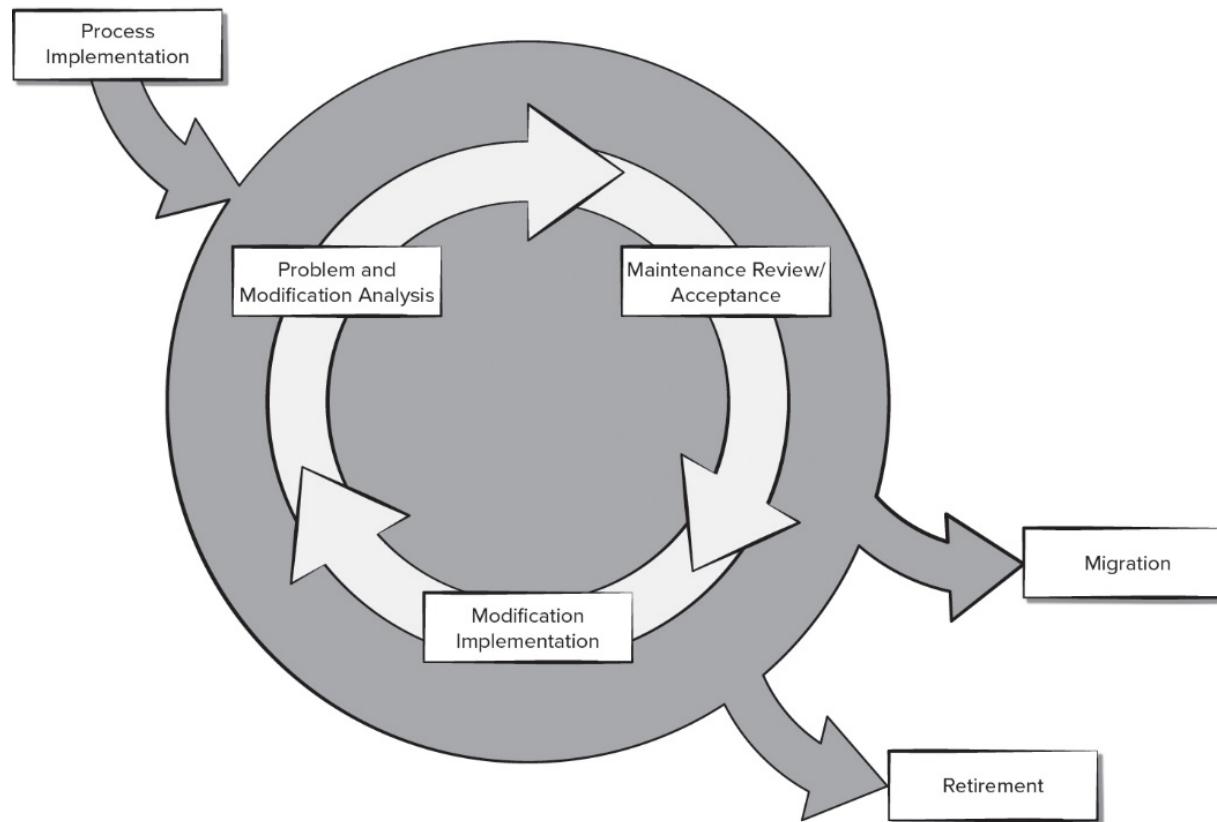


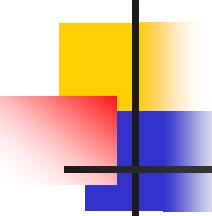
# Reverse Engineering

- **Reverse Engineering to Understand Data** - first reengineering task often begins by constructing UML class diagram.
- **Reverse Engineering of Internal Data Structures** - focuses on the definition of object classes.
- **Reverse Engineering of Database Structure** - reengineering one database schema into another requires an understanding of existing objects and their relationships.
- **Reverse engineering to understand processing** - attempts to understand procedural abstractions in source code.
- **Reverse engineering to understand user interfaces** - may need to be done as part of the maintenance task (for example, adding a GUI).

# Proactive Software Support Model

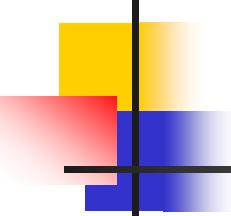
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





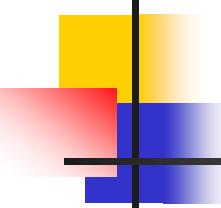
# Software Analytics and Proactive Maintenance

- Be sure you are using analytics to identify meaningful development problems, or you will get no buy-in from the software engineers.
- The analytics must make use of application domain knowledge to be useful to developers (this implies the use of experts to validate the analytics).
- Developing analytics requires iterative and timely feedback from the intended users.
- Make sure the analytics are scalable to larger problems and customizable to incorporate new discoveries made over time.
- Evaluation criteria used needs to be correlated to real software engineering practices.



# Role of Social Media

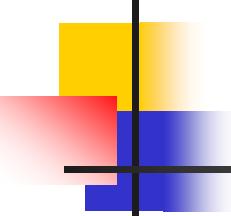
- Many online stores allow users to provide feedback on the apps by posting ratings or comments.
- The feedback found in these reviews may contain usage scenarios, bug reports, or feature requests.
- Mining these reports can help developers identify potential maintenance and software evolution tasks.
- Many companies maintain Facebook pages or Twitter feeds to support their user communities.
- Some companies encourage product users to send program crash information for analysis by the support team members.
- Some companies use the questionable practice of tracking how and where products are used by customers without their knowledge.



# Cost of Support

Nine parameters are defined:

- P1 = current annual maintenance cost for an application.
- P2 = current annual operation cost for an application.
- P3 = current annual business value of an application.
- P4 = predicted annual maintenance cost after reengineering.
- P5 = predicted annual operations cost after reengineering.
- P6 = predicted annual business value after reengineering.
- P7 = estimated reengineering costs.
- P8 = estimated reengineering calendar time.
- P9 = reengineering risk factor ( $P9 = 1.0$  is nominal).
- L = expected life of the system.



# Cost of Support

- The cost associated with continuing maintenance of a candidate application (that is, reengineering is not performed) can be defined as:

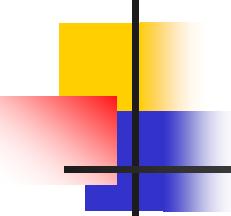
$$C_{\text{maint}} = [P_3 - (P_1 + P_2)] \times L$$

- The costs associated with reengineering are defined using the following relationship:

$$C_{\text{reeng}} = [P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9)]$$

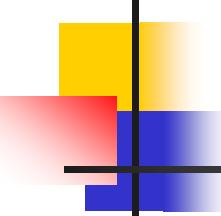
- Using the costs presented in equations above, the overall benefit of reengineering can be computed as:

$$\text{Cost benefit} = C_{\text{reeng}} - C_{\text{maint}}$$



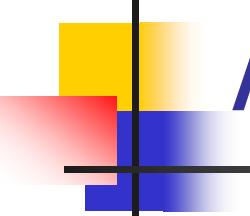
# Data Refactoring

- Data refactoring should be preceded by **source code analysis**.
- Data analysis requires the evaluation of programming language statements containing data definitions, file descriptions, I/O, and interface descriptions are evaluated.
- **Data redesign** involves a **data record standardization** which clarifies data definitions to achieve consistency among data item names or physical record formats, **data name rationalization** ensures that data naming conventions conform to local standards.
- When refactoring moves beyond standardization and rationalization, physical modifications to existing data structures are made to make the data design more effective.
- This may mean a translation from one file format to another, or in some cases, translation from one type of database to another.



# Code Refactoring

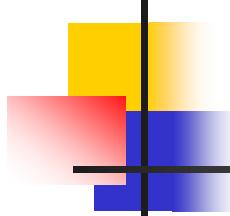
- **Code refactoring** is performed to yield a design that produces the same function but with higher quality than the original program.
- The objective is to take “spaghetti-bowl” code and derive a design that conforms to the quality factors defined for the product.
- Another approach relies on the use of anti-patterns to identify bad code design practices and suggest possible solutions.
- Code refactoring can alleviate immediate problems associated with debugging or small software changes; it is not reengineering.
- Real benefit of code refactoring is achieved only when data and architecture are refactored as well.



# Architectural Refactoring

Architectural refactoring is one of the design trade-off options for dealing with a messy program:

- You can struggle through modification after modification, fighting the ad hoc design and tangled source code to implement the necessary changes.
- You can attempt to understand the broader inner workings of the program to make modifications more effectively.
- You can revise (redesign, recode, and test) those portions of the software that require modification, applying a meaningful software engineering approach to all revised segments.
- You can completely redo (redesign, recode, and test) the program, using reengineering tools to understand the current design.

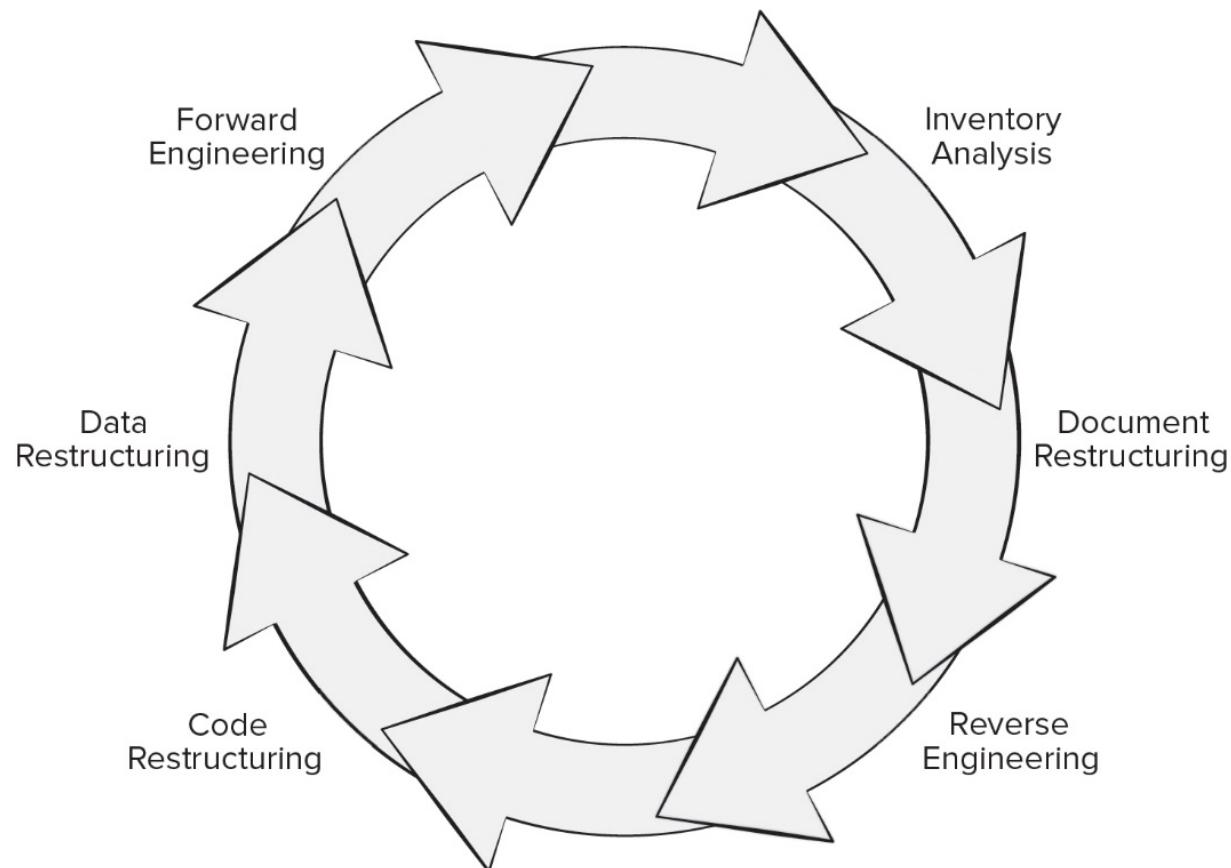


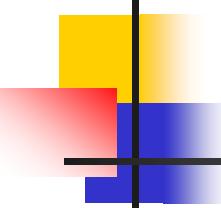
# Software Evolution

- The cost to maintain one line of source code may be 20 to 40 times the cost of initial development of that line.
- Redesign of the software architecture (program and/or data structure), using modern design concepts, can greatly facilitate future maintenance.
- Because a prototype of the software already exists, development productivity should be much higher than average.
- Tools for reengineering will automate some parts of the job.
- Users now have experience with the software so new requirements and change direction can be ascertained with greater ease.
- A complete software configuration (documents, programs and data) when the evolutionary preventive maintenance is done.

# Reengineering Process Model

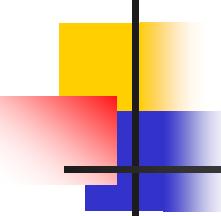
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





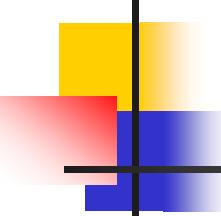
# Inventory Analysis

- Every software organization should have an inventory of all applications.
- The inventory can be a spreadsheet model containing information that provides a detailed description (For example size, age, business criticality) of every active application.
- Sorting this information according to business criticality, longevity, current maintainability, and other criteria, helps to identify candidates for reengineering.
- Resources can then be allocated to candidate applications for reengineering work.
- The inventory should be revisited on a regular basis.



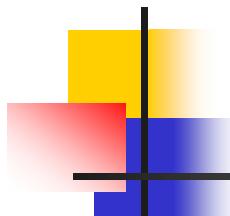
# Document Restructuring

- Weak documentation is the trademark of many legacy systems.
- In some cases, creating documentation when none exists is simply too costly.
- In other cases, some documentation must be created, but only when changes are made.
- If a modification occurs, document it - try to reign in technical debt. There are situations in which a critical system must be fully documented, but documents to an essential minimum.



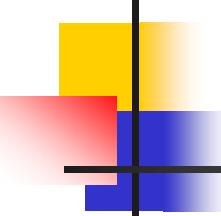
# Code Restructuring

- Source code is analyzed using a refactoring tool.
- Poorly design code segments are redesigned.
- Violations of structured programming are noted and code is refactored (this can be done automatically).
- The resultant refactored code is reviewed and tested to ensure that no anomalies have been introduced.
- Internal code documentation is updated.



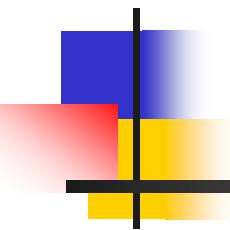
# Data Restructuring

- Data refactoring is a full-scale reengineering activity.
- The current data architecture is analyzed and necessary data models are defined.
- Data objects and attributes are identified, and existing data structures are reviewed for quality.
- When data structure is weak (for example, flat files are currently implemented, when a relational approach would greatly simplify processing), the data are reengineered.
- Data architecture has a strong influence on program architecture and the algorithms that populate it, changes to the data result in architectural or code-level changes.



# Forward Engineering

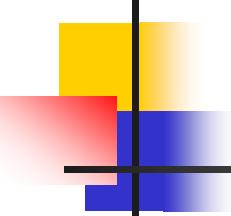
- In an ideal world, applications would be rebuilt using an automated reengineering engine.
- **Forward engineering** recovers design information from existing software and uses this information to alter or reconstitute the existing system to improve its overall quality.
- Reengineered software re-creates the function of the existing system and adds new functions and/or improves overall performance.
- Forward engineering does not simply create a modern equivalent of an older program - the redeveloped program extends the capabilities of the older application.



# *COMP 354: Introduction to Software Engineering*

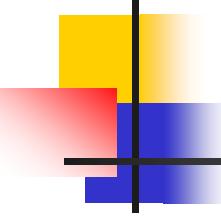
## Pattern-Based Design

Based on Chapter 14 of the textbook



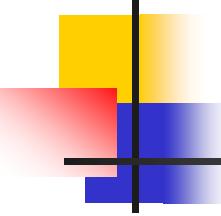
# Design Patterns

- **Design pattern** can be thought of as a three-part rule which expresses a relation between a certain context, a problem, and a solution.
- **Context** allows the reader to understand the environment in which the problem resides and what solution might be appropriate within that environment.
- Requirements, including limitations and constraints, acts as a system of forces that influences how:
  - The problem can be interpreted within its context.
  - The solution can be effectively applied.



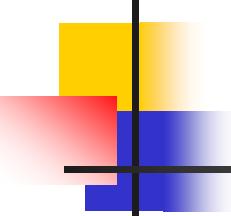
# Effective Design Pattern

- **Solves a problem:** Patterns capture solutions, not just abstract principles or strategies.
- **Proven concept:** Patterns capture solutions with proven track records, not theories or speculation.
- **Solution isn't obvious:** The best patterns generate a solution to a problem indirectly--a necessary approach for the most difficult problems of design.
- **Describes a relationship:** Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- **Elegant in its approach and utility:** describes simple solutions to specific problems; the best patterns explicitly appeal to aesthetics and usefulness.



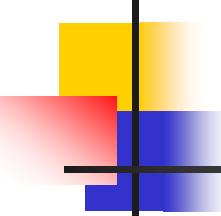
# Kinds of Patterns

- **Architectural patterns** describe broad-based design problems that are solved using a structural approach.
- **Data patterns** describe recurring data-oriented problems and the data modeling solutions that can be used to solve them.
- **Component patterns** (design patterns) address problems associated with the development of subsystems and components, the manner in which they communicate with one another, and their placement within a larger architecture.



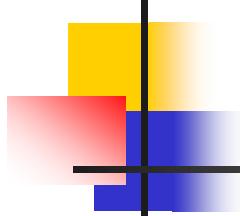
# Kinds of Patterns

- **Interface design patterns** describe common user interface problems and their solution with a system of forces that includes the specific characteristics of end-users.
- **WebApp patterns** address a problem set that is encountered when building WebApps and often incorporate many of other patterns categories.
- **Mobile patterns** describe solutions to problems commonly encountered when developing solutions for mobile platforms.



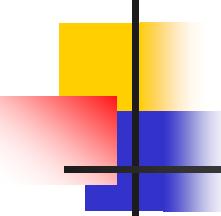
# Kinds of Patterns

- **Creational patterns** focus on “creation, composition, and representation of objects”
  - **Abstract factory pattern**: centralize decision of what factory to instantiate.
  - **Builder pattern**: separates the construction of a complex object from its representation.
- **Structural patterns** focus on problems and solutions associated with how classes and objects are organized and integrated to build a larger structure:
  - **Adapter pattern**: 'adapts' one interface for a class into one that a client expects.
  - **Container pattern**: create objects for the sole purpose of holding other objects and managing them.



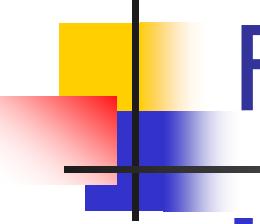
# Kinds of Patterns

- **Behavioral patterns** address problems associated with the assignment of responsibility between objects and the manner in which communication is effected between objects:
  - **Chain of responsibility pattern:** Command objects are handled or passed on to other objects by logic-containing processing objects.
  - **Command pattern:** Command objects encapsulate an action and its parameters.



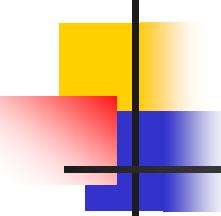
# Frameworks

- Patterns themselves may not be sufficient to develop a complete design.
  - In some cases it may be necessary to provide an implementation-specific skeletal infrastructure, called a framework, for design work.
  - You can select a reusable architecture that provides the generic structure and behavior for a family of software abstractions along with their context which specifies use within a given domain.
- A framework is not an architectural pattern, but rather a skeleton with a collection of “plug points” (hooks or slots) that enable it to be adapted to a specific problem domain.
  - Plug points enable you to integrate problem specific classes or functionality within the skeleton.
  - Collection of cooperating classes.



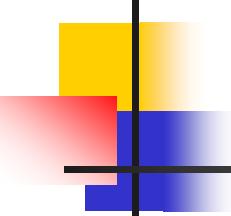
# Patterns, Components and Frameworks

- Design patterns are abstract solutions that have been observed to work for concrete problems.
  - They define the conditions for the problem, the solution used, and any consequences that might result.
  - They lead to efficiency and reuse.
- Components are concrete implementations of design patterns.
  - They provide code you can copy and paste directly into a project.
  - Ideally they'll be flexible and offer a mechanism for modification.
  - There can be multiple components for a single design pattern.
- Frameworks combine components and package them together.
  - They provide APIs and tools for using the framework and tend to be specialized in nature.



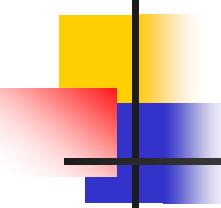
# Pattern Description

- **Pattern name**—describes the essence of the pattern in a short but expressive name
- **Problem**—describes the problem that the pattern addresses
- **Motivation**—provides an example of the problem
- **Context**—describes the environment in which the problem resides including application domain
- **Forces**—lists the system of forces that affect the manner in which the problem must be solved; includes a discussion of limitation and constraints that must be considered
- **Solution**—provides a detailed description of the solution proposed for the problem



# Pattern Description

- **Intent**—describes the pattern and what it does
- **Collaborations**—describes how other patterns contribute to the solution
- **Consequences**—describes the potential trade-offs that must be considered when the pattern is implemented and the consequences of using the pattern
- **Implementation**—identifies special issues that should be considered when implementing the pattern
- **Known uses**—provides examples of actual uses of the design pattern in real applications
- **Related patterns**—cross-references related design patterns

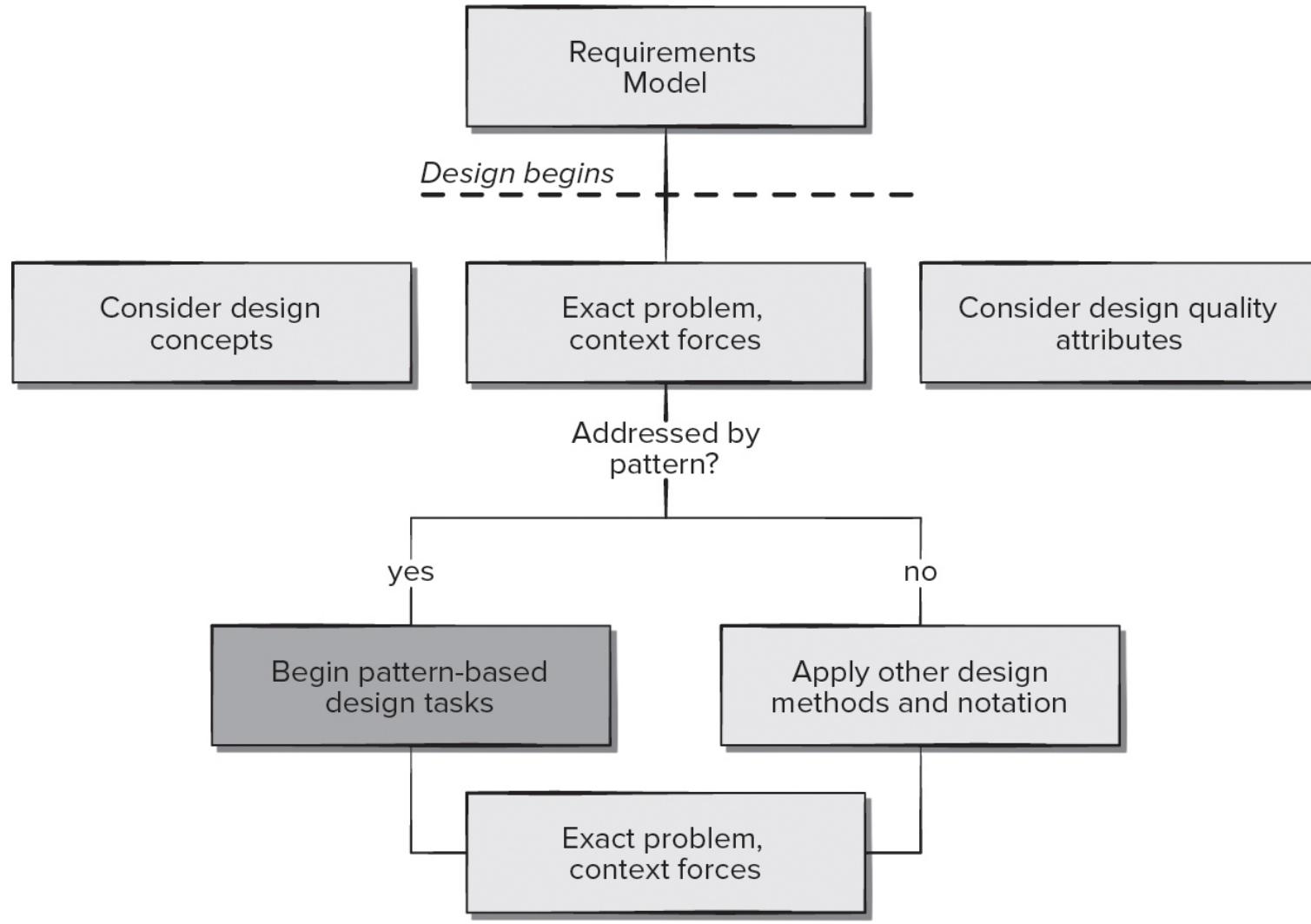


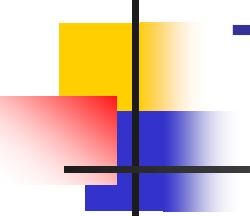
# Pattern-Based Design

- A software designer begins with a requirements model (either explicit or implied) that presents an abstract representation of the system.
- The requirements model describes the problem set, establishes the context, and identifies the system of forces that hold sway.
- If you discover you are faced with a problem, context, and system of forces that have solved before then use that solution.
- If it is a new problem then use the methods and modeling tools available for architectural, component-level, and interface design to create a new solution (pattern).

# Pattern-Based Design in Context

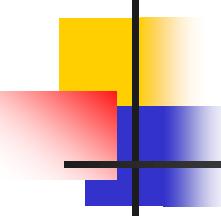
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





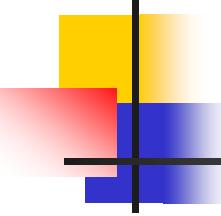
# Thinking in Patterns

1. Be sure you understand big picture (requirements model) - the context in which the software to be built resides.
2. Examining the big picture, extract the patterns that are present at that level of abstraction.
3. Begin your design with 'big picture' patterns that establish a context or skeleton for further design work.
4. Work inward from the context, look for patterns at lower levels of abstraction that contribute to design solution.
5. Repeat steps 1 to 4 until complete design is fleshed out.
6. Refine the design by adapting each pattern to the specifics of the software you're trying to build.



# Design Tasks

1. Examine the requirements model and develop a problem hierarchy.
2. Determine if a reliable pattern language has been developed for the problem domain.
3. Beginning with a broad problem, determine whether one or more architectural patterns are available for it.
4. Using the collaborations provided for the architectural pattern, examine subsystem or component level problems, and search for patterns to address them.



# Design Tasks

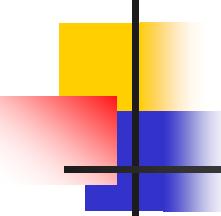
5. Repeat steps 2 through 5 until all broad problems have been addressed.
6. If user interface design problems have been isolated, search the user interface design pattern repositories for appropriate patterns.
7. Regardless of its level of abstraction, if a pattern language and/or patterns repository or individual pattern shows promise, compare the problem to be solved against the existing pattern(s) presented.
8. Be certain to refine the design as it is derived from patterns using design quality criteria as a guide.

# Pattern Organizing Table

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.

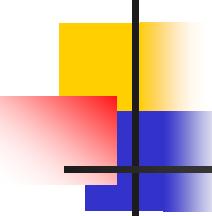
	Database	Application	Implementation	Infrastructure
<b>Data/Content</b>				
Problem statement ...	PatternName(s)		PatternName(s)	
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...	PatternName(s)			PatternName(s)
<b>Architecture</b>				
Problem statement ...		PatternName(s)		
Problem statement ...		PatternName(s)		PatternName(s)
Problem statement ...				
<b>Component-level</b>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...				PatternName(s)
Problem statement ...		PatternName(s)	PatternName(s)	
<b>User interface</b>				
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	
Problem statement ...		PatternName(s)	PatternName(s)	

Source: Adapted from Microsoft, "Prescriptive Architecture: Integration and Patterns," MSDN, May 2004.



# Common Mistakes

- Not enough time has been spent to understand the underlying problem, its context and forces, and as a consequence, you select a pattern that looks right, but is inappropriate for the solution required.
- Once the wrong pattern is selected, you refuse to see your error and force fit the pattern.
- In other cases, the problem has forces that are not considered by the pattern you've chosen, resulting in a poor or erroneous fit.
- Sometimes a pattern is applied too literally and the required adaptations for your problem space are not implemented.



# Architectural Patterns

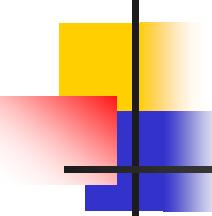
- Example: every house (and every architectural style for houses) employs a Kitchen pattern.
- Kitchen pattern and patterns it collaborates with address problems associated with storage and preparation of food, tools required to accomplish these tasks, and rules for placement of these tools relative to workflow in the room.
- The pattern might address problems associated with counter tops, lighting, wall switches, a central island, flooring, and so on.
- Obviously, there is more than a single design for a kitchen, but every design can be conceived within the context and system forces of the 'solution' suggested by the Kitchen pattern.

# Component-Level Design Patterns

- Component-level design patterns provide a proven solution that addresses one or more sub-problems extracted from the requirement model.
- In many cases, design patterns of this type focus on some functional element of a system.
- For example: the SafeHomeAssured.com application must address the following design sub-problem: How can we get product specifications and related information for any SafeHome device?

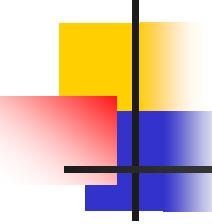
# Component-Level Design Patterns

- Having enunciated the sub-problem that must be solved, consider context and the system of forces that affect the solution.
- Examining the appropriate requirements model use case, the specification for a SafeHome device (for example: a security sensor or camera) is used for informational purposes by the consumer.
- However, other information that is related to the specification (for example: pricing) may be used when e-commerce functionality is selected.
- The solution to the sub-problem involves a search. Since searching is a very common problem, it should come as no surprise that there are many search-related patterns.



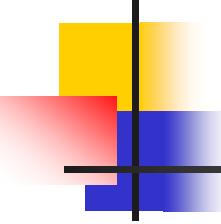
# Search Patterns

- **HelpWizard.** Users need help on a certain topic related to the website or when they need to find a specific page within the site.
- **SearchArea.** Users must find a page.
- **SearchTips.** Users need to know how to control the search engine.
- **SearchResults.** Users must process a list of search results.
- **SearchBox.** Users must find an item or specific information.



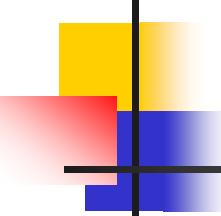
# Anti-Patterns

- Anti-patterns describe commonly used solutions to design problems that usually have negative effects on software quality.
- Anti-patterns can provide tools to help developers recognize when these problems exist and may provide detailed plans for reversing the underlying problem causes and implementing better solutions to these problems.
- Anti-patterns can provide guidance to developers looking for ways to refactor software to improve its quality.
- Anti-patterns can be used by technical reviewers to uncover areas of concern.



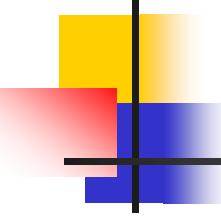
# Selected Anti-Patterns

- **Blob.** Single class with large number of attributes, operators, or both.
- **Stovepipe system.** A barely maintainable assemblage of ill-related components.
- **Boat anchor.** Retaining a part of system that no longer has any use.
- **Spaghetti code.** Program whose structure is barely comprehensible, especially because of misuse of code structures.
- **Copy and paste programming.** Copying existing code several times rather than creating generic solutions.
- **Silver bullet.** Assume that a favorite technical solution will always solve a larger process or problem.
- **Programming by permutation.** Trying to approach a solution by successively modifying the code to see if it works.



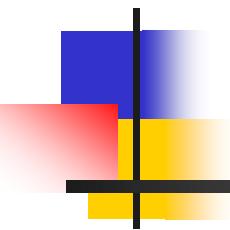
# User Interface (UI) Patterns

- **Whole UI.** Provide design guidance for top-level structure and navigation throughout the entire interface.
- **Top-level Navigation.** Provides a top-level menu, often coupled with a logo or identifying graphic, that enables direct navigation to any of the system's major functions.
- **Page layout.** Address the general organization of pages (for Websites) or distinct screen displays (for interactive applications).
- **E-commerce.** Specific to Web sites, these patterns implement recurring elements of e-commerce applications.
- **Shopping cart.** Provides a list of items selected for purchase.



# Mobility Design Patterns

- **Check-in screens.** How do I check in from a specific location, make a comment, and share comments with friends and followers on a social network?
- **Maps.** How do I display a map within the context of an app that addresses some other subject?
- **Popovers.** How do I represent a message or information that arises in real time or as the consequence of a user action?
- **Sign-up flows.** How do I provide a simple way to sign in or register for information or functionality?
- **Custom tab navigation.** How do I represent a variety of different content objects in a manner that enables user to select one?
- **Invitations.** How do I inform the user that he must participate in some action or dialog?



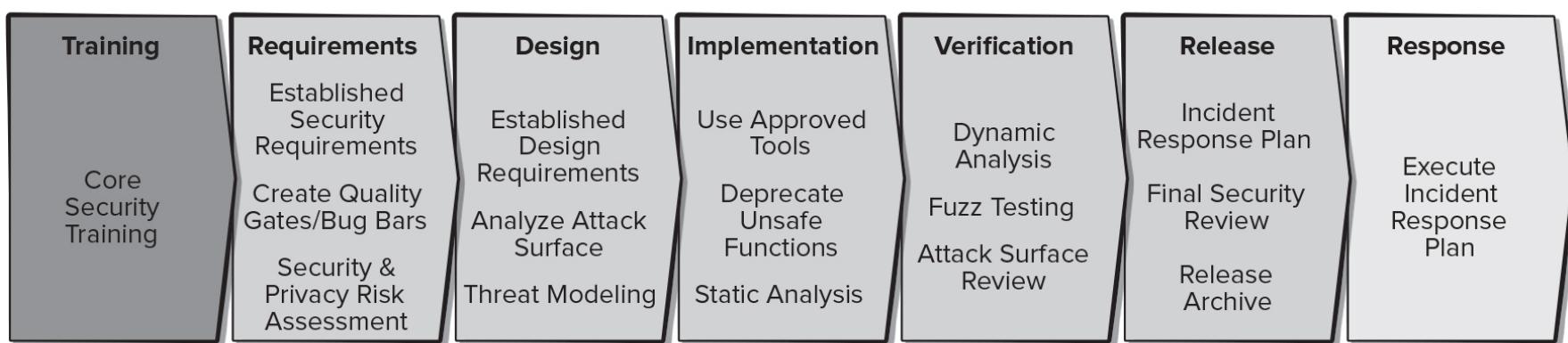
# *COMP 354: Introduction to Software Engineering*

## Software Security Engineering

Based on Chapter 18 of the textbook

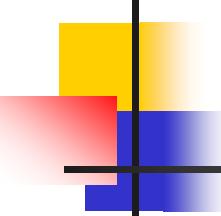
# Secure Software Development Process Model

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



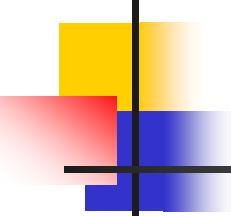
## Secure Software Development Process Model at Microsoft

Adapted from Shunn, A., et al. Strengths in Security Solutions, Software Engineering Institute, Carnegie Mellon University, 2013. Available at <http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=77878>.



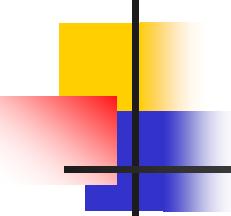
# Microsoft Secure by Design

- **Secure architecture, design, and structure.** Developers consider security issues part of the software architectural design process.
- **Threat modeling and mitigation.** Threat models created and mitigations are present in all design and functional specifications.
- **Elimination of vulnerabilities.** This review includes the use of analysis and testing tools to eliminate classes of vulnerabilities presents in the code.
- **Improvements in security.** Less secure legacy protocols and code are deprecated, users are provided with secure alternatives consistent.



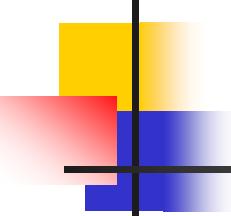
# Microsoft Secure by Default

- **Least privilege.** All components run with the fewest possible permissions.
- **Defense in depth.** Components do not rely on a single threat mitigation solution that exposes users if it fails.
- **Conservative default settings.** Development team minimizes attack surface in default configuration.
- **Avoidance of risky default changes.** Applications do not make any changes that reduce computer security.
- **Less commonly used services off by default.** If fewer than 80 percent of a program's users use a feature, that feature should not be activated by default.



# Microsoft Secure in Deployment

- **Deployment guides.** Prescriptive deployment guides outline how to deploy each feature of a program securely, including providing users with information that enables them to assess the security risk of activating non-default options.
- **Analysis and management tools.** Security analysis and management tools enable administrators to configure the optimal security level for a release.
- **Patch deployment tools.** Deployment tools aid in patch deployment.

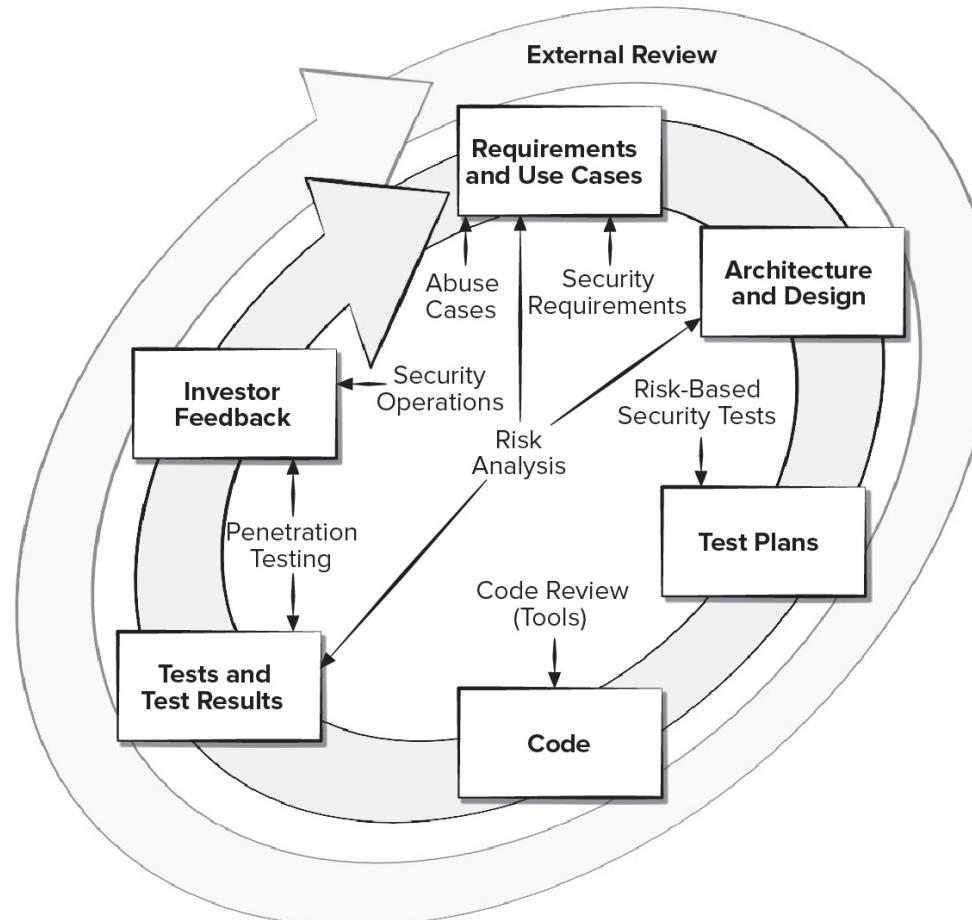


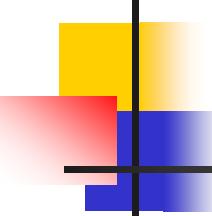
# Communications

- **Security response.** Development teams respond promptly to reports of security vulnerabilities and communicate information about security updates.
- **Community engagement.** Development teams proactively engage with users to answer questions about security vulnerabilities, security updates, or changes in the security landscape.

# Software Security Touchpoints (Activities)

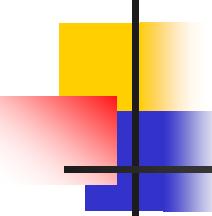
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





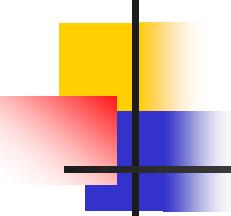
# SQUARE Security Requirements Engineering

- Step 1. **Agree on definitions.** Needed as a prerequisite to security requirements engineering so there is no semantic confusion.
- Step 2. **Identify assets and security goals.** Step occurs at project organizational level and needed to support software development.
- Step 3. **Develop artifacts.** Often, organizations do not have key documents needed to support requirements definition, or they may not be up to date.
- Step 4. **Perform risk assessment.** Requires an expert in risk assessment methods, support of stakeholders, and support of a security requirements engineer.



# SQUARE Security Requirements Engineering

- Step 5. **Select elicitation technique.** This step becomes important when there are diverse stakeholders.
- Step 6. **Elicit security requirements.** This builds on the artifacts that were developed in earlier steps.
- Step 7. **Categorize requirements.** Allows security requirements engineer to identify essential requirements.
- Step 8. **Prioritize requirements.** Performs a cost-benefit analysis to determine security requirements with a high payoff relative to their cost.
- Step 9. **Requirements inspection.**

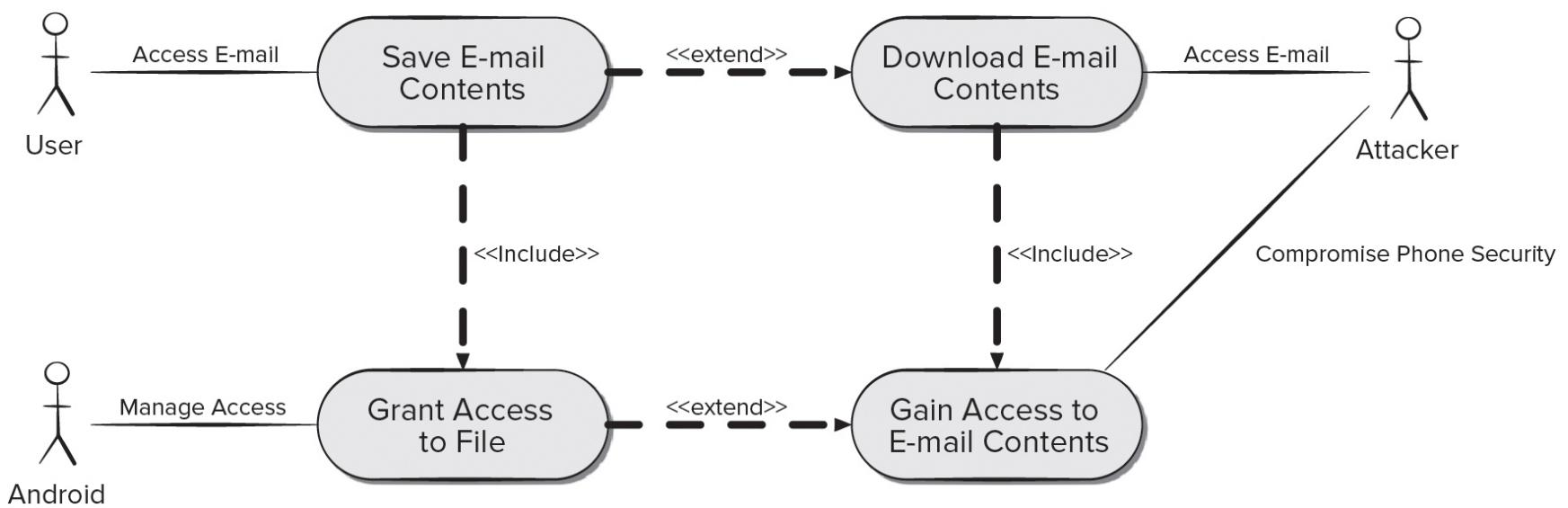


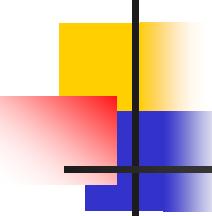
# Misuse (Abuse) Cases

- A **misuse case** can be thought of as a use case that the attacker initiates.
- Misuse cases need to be prioritized as generated.
- Trying to answer such questions like these help developers to analyze their assumptions and allows them to fix problems up front:
  - How can the system distinguish between valid and invalid input data?
  - Can it tell whether a request is coming from a legitimate application or a rogue application?
  - Can an insider cause a system to malfunction?

# Misuse Case Example

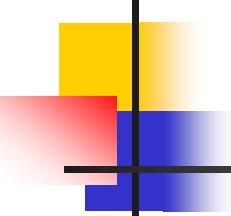
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





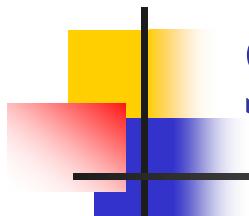
# Attack Patterns

- Attack patterns can provide some help by providing a blueprint for creating an attack.
- For example, buffer overflow is one type of security exploitation.
- Attackers trying to capitalize on a buffer overflow make use of similar steps.
- Attack patterns can document these steps (for example, timing, resources, techniques) as well as practices software developers can use to prevent or mitigate their success.
- When you're trying to develop misuse cases, attack patterns can help.



# Risk Management Framework (RMF) Steps

- **Categorize** the information system and the information processed, stored, and transmitted by that system based on an impact analysis.
- **Select** an initial set of baseline security controls for the information system based on the security categorization.
- **Implement** the security controls and describe how the controls are employed within the information system and its environment.
- **Assess** security controls to determine the extent to which they are operating to meeting system security requirements.
- **Authorize** information system operation based on a determination that the risk to organization, assets, and individuals is acceptable.
- **Monitor** the security controls in the information system on an ongoing basis including assessing control effectiveness, documenting changes to the system.



# STRIDE Threat Categories

## Threat

Spoofing

Tampering

Repudiation

Information disclosure

Denial of service

Elevation of privilege

## Security Property

Authentication

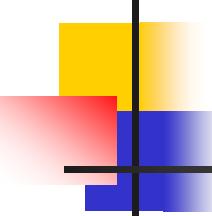
Integrity

Nonrepudiation

Confidentiality

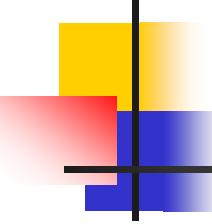
Availability

Authorization



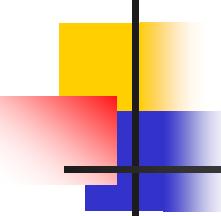
# STRIDE Threat Modeling Steps

- Typical STRIDE implementation includes modeling a system with data flow diagrams (DFDs):
  - Mapping the DFD elements to the six threat categories,
  - Determining the specific threats via checklists or threat trees.
  - Documenting the threats and steps for their prevention.
- In the next stage, the STRIDE user works through a checklist of specific threats that are associated with each match between a DFD element and threat category.
- Once the threats have been identified, mitigation strategies can be developed and prioritized.
- Typically, prioritization is based on cost and value considerations of implementing or not implementing a mitigation strategy.



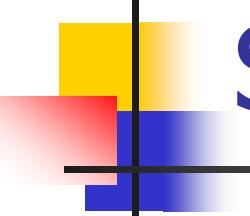
# Attack Surface

- The **attack surface** of an application is:
  - The sum of all paths for data/commands into and out of the application.
  - The code that protects these paths.
  - All valuable data used in the application.
  - The code that protects these data.
- **Attack Surface Analysis** involves mapping the parts of a system need to be reviewed and tested for security vulnerabilities with the intention of minimizing risks to the attack surface.



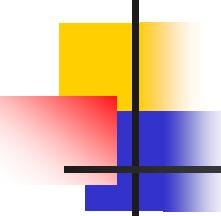
# Secure Coding Practices

- **Validate input.** Validate input from all untrusted data sources.
- **Heed compiler warnings.** Compile code using the highest warning level available for your compiler and eliminate warnings by modifying the code.
- **Architect and design for security policies.** Create a software architecture and design your software to implement and enforce security policies.
- **Keep it simple.** Keep the design as simple and as small as possible.
- **Default deny.** Base access decisions on permission rather than exclusion.



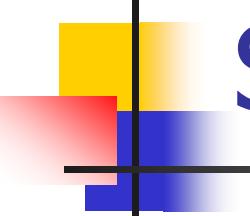
# Secure Coding Practices

- Adhere to the principle of least privilege. Every process should execute with the least set of privileges necessary to complete the job.
- Sanitize data sent to other systems. Sanitize all data passed to complex subsystems such as command shells, relational databases, and commercial off-the-shelf (COTS) components.
- Practice defense in depth. Manage risk with multiple defensive strategies.
- Use effective quality assurance techniques.
- Adopt a secure coding standard.



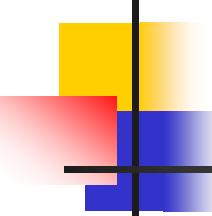
# Measurement

- Measures of software quality can go a long way toward measuring software security.
- Defect and vulnerability count are useful measures.
- Not all software defects are security problems, vulnerabilities in software generally result from a defect of some kind in the requirements, architecture, or code.
- To assess software vulnerabilities and associated security issues, data must be collected so that patterns can be analyzed over time.
- Without collecting data about software security issues, it is impossible to measure its improvement.



# Security Measure Examples

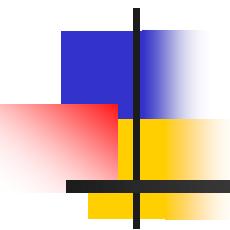
- Percentage of security requirements covered by attack patterns, misuse and abuse cases, and other specified means of threat modeling and analysis (Requirements Engineering).
- Percentage of architectural and design components subject to attack surface analysis and measurement (Architecture and Design).
- Financial and/or human safety estimate of impact for each threat category (Risk).
- Number of (vetted) trusted suppliers in the supply chain by level (Trusted Dependencies).



# Software Assurance Maturity Model (SAMM)

SAMM is an open framework with the following objectives:

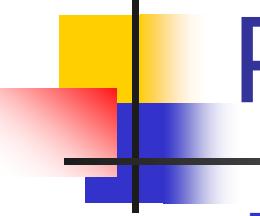
- Evaluate an organization's existing software security practices.
- Build a balanced software security assurance program in well-defined iterations.
- Demonstrate concrete improvements to a security assurance program.
- Define and measure security-related activities throughout an organization.



# *COMP 354: Introduction to Software Engineering*

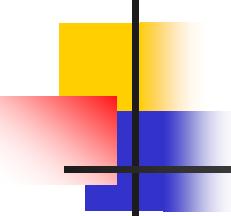
## Software Engineering Principles

Based on Chapter 6 of the textbook



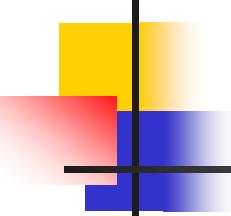
# Principles that Guide Process

- **Principle #1.** **Be agile.** Regards of your process model, let the basic tenets of agile development govern your approach.
- **Principle #2.** **Focus on quality at every step.** The exit condition for every process activity, action, and task should focus on the quality of the work product produced.
- **Principle #3.** **Be ready to adapt.** Dogma has no place in software development. Adapt your approach to constraints imposed by the problem, the people, and the project itself.
- **Principle #4.** **Build an effective team.** Software engineering process and practice are important, but the bottom line is people. Build a self-organizing team.



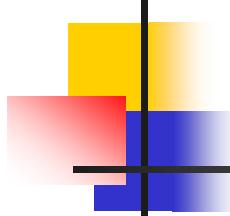
# Principles that Guide Process

- **Principle #5.** Establish mechanisms for communication and coordination. Projects fail because information falls into the cracks and/or stakeholders fail to coordinate their efforts.
- **Principle #6.** Manage change. Approach may formal or informal. You need mechanisms to manage how changes are requested, assessed, approved and implemented.
- **Principle #7.** Assess risk. Lots of things can go wrong as software is being developed, establish contingency plans.
- **Principle #8.** Create work products that provide value for others. Create only those work products that provide value for other process activities, actions or tasks.



# Principles that Guide Practice

- **Principle #1.** **Divide and conquer.** Analysis and design should always emphasize separation of concerns (SoC).
- **Principle #2.** **Understand the use of abstraction.** Abstraction is a simplification of a complex system element used to communicate meaning simply.
- **Principle #3.** **Strive for consistency.** A familiar context makes software easier to use.
- **Principle #4.** **Focus on the transfer of information.** Pay special attention to the analysis, design, construction, and testing of interfaces.

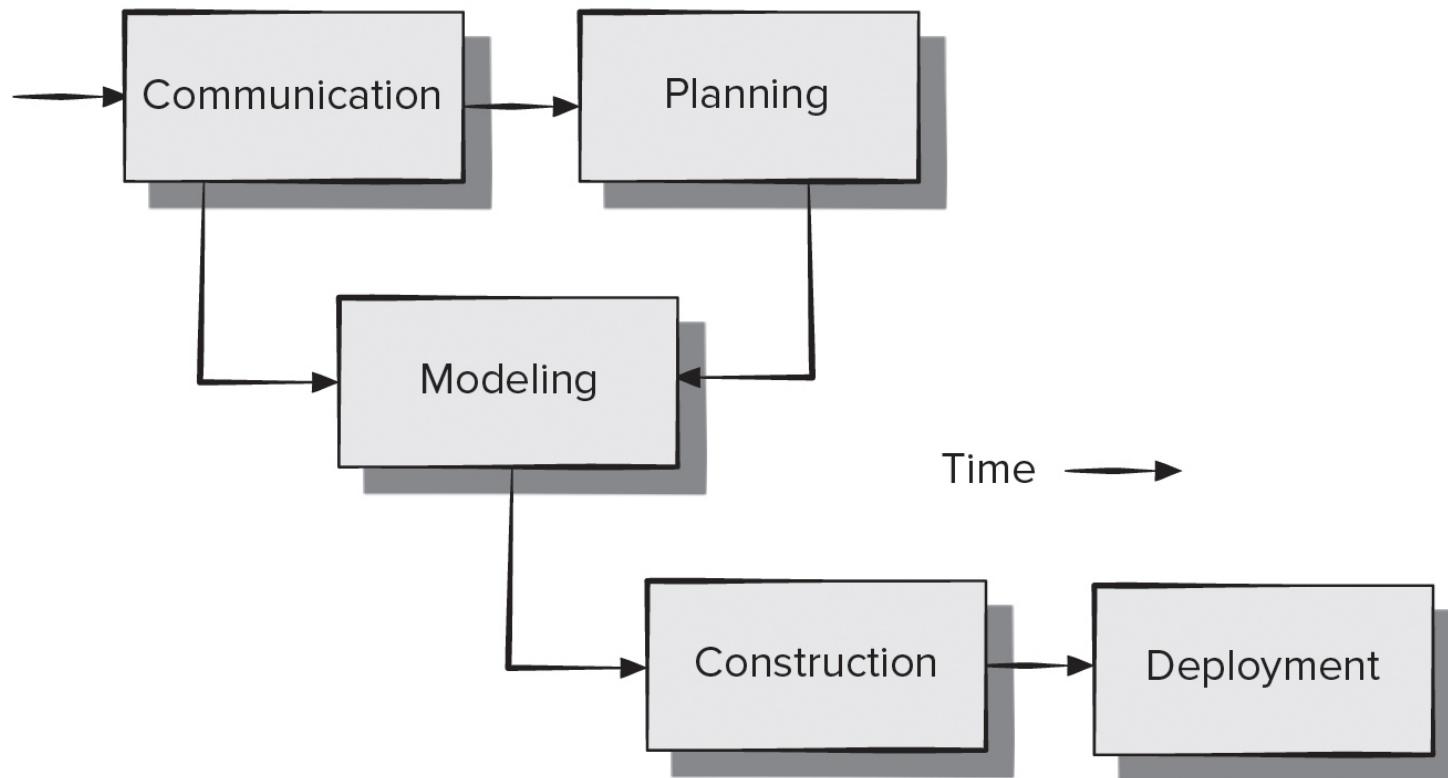


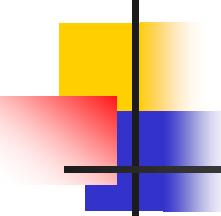
# Principles that Guide Practice

- **Principle #5.** Build software that exhibits effective modularity. Provides a mechanism for realizing the philosophy of Separation of concerns .
- **Principle #6.** Look for patterns. The goal of patterns is to create a body of literature to help developers resolve recurring problems encountered in software development.
- **Principle #7.** Use multiple viewpoints. Represent the problem and solution from different perspectives.
- **Principle #8.** Some consumes your work products. Remember that someone will maintain the software.

# Simplified Process Framework

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



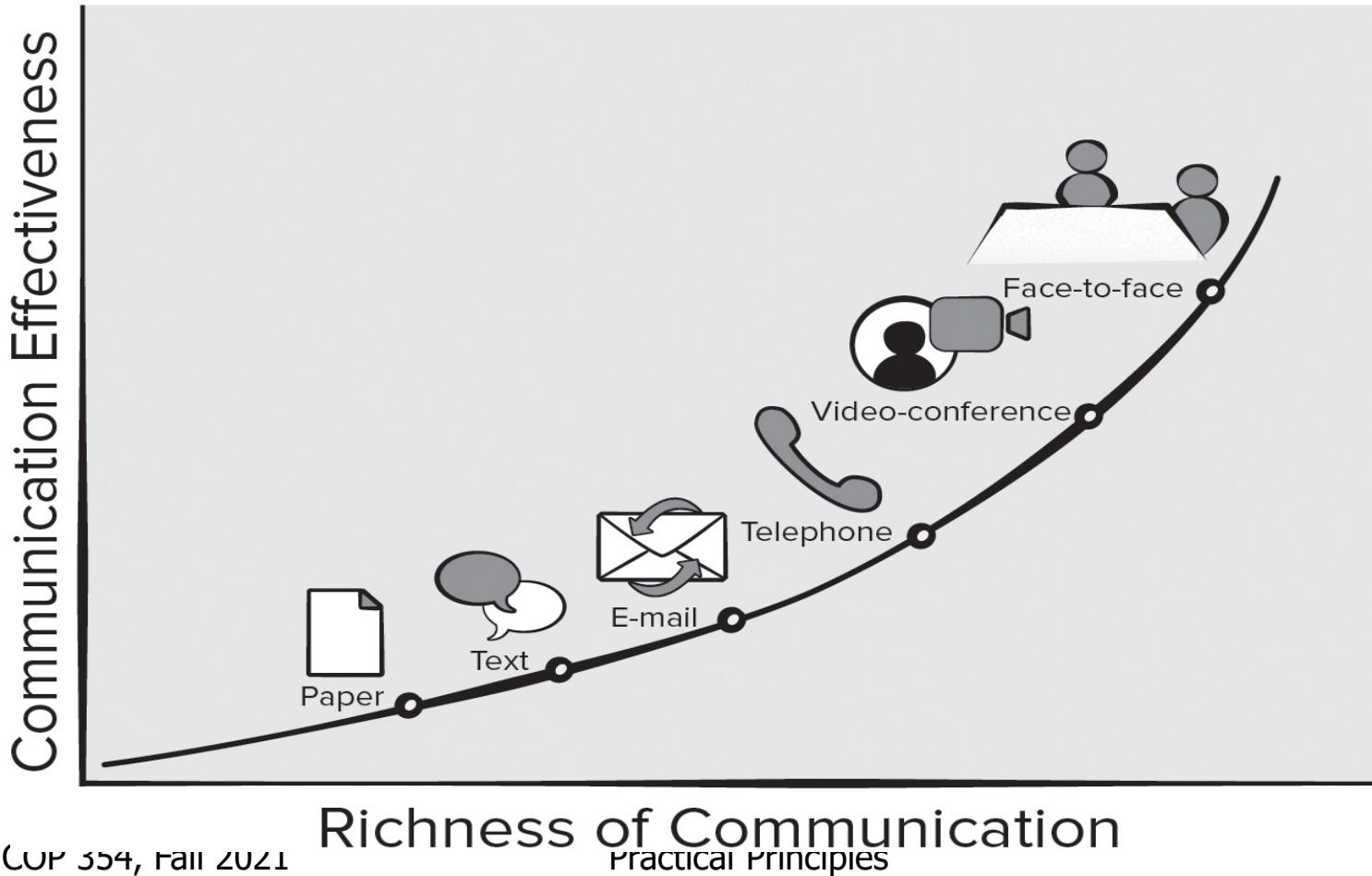


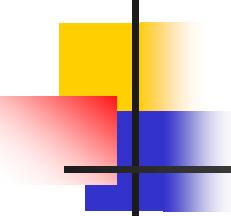
# Communications Principles

- **Principle #1.** Listen. Try to focus on the speaker's words, not formulating your response to those words.
- **Principle # 2.** Prepare before you communicate. Understand a problem before meeting with others.
- **Principle # 3.** Someone should facilitate the activity. Every communication meeting should have a leader to keep the conversation moving in a productive direction.
- **Principle #4.** Face-to-face communication is best. Visual representations of information can be helpful.
- **Principle # 5.** Take notes and document decisions. Someone should serve as a "recorder" and write down all important points and decisions.

# Communications Mode Effectiveness

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



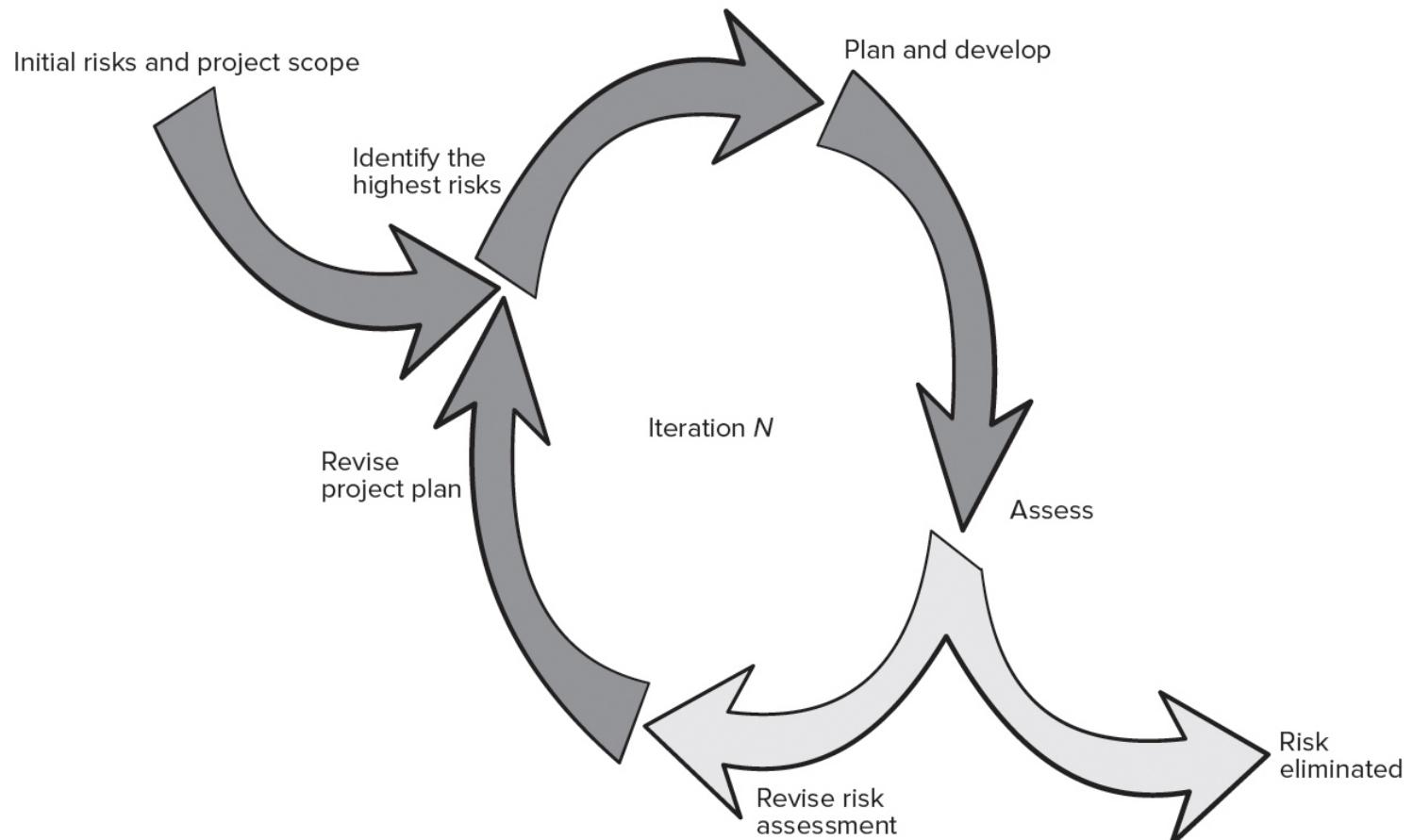


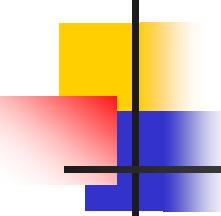
# Communications Principles

- **Principle # 6.** *Strive for collaboration.* Consensus occurs when collective team knowledge is combined.
- **Principle # 7.** *Stay focused, modularize your discussion.* The more people involved in communication the more likely discussion will bounce between topics.
- **Principle # 8.** *If something is unclear, draw a picture.*
- **Principle # 9.** (a) Once you agree to something, move on; (b) If you can't agree to something, move on; (c) If a feature or function is unclear and cannot be clarified at the moment, move on.
- **Principle # 10.** *Negotiation is not a contest or a game. It works best when both parties win.*

# Iterative Planning Process

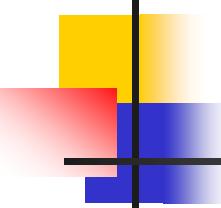
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Planning Principles

- **Principle #1.** Understand the scope of the project.  
Scope provides the software team with a destination as the roadmap is created.
- **Principle #2.** Involve the customer in the planning activity. They define priorities and project constraints.
- **Principle #3.** Recognize that planning is iterative. A project plan is likely to change as work begins.
- **Principle #4.** Estimate based on what you know.  
Estimation provides an indication of effort, cost, and task duration, based on team's current understanding of work.
- **Principle #5.** Consider risk as you define the plan.  
Contingency planning is needed for identified high impact and high probability risks.

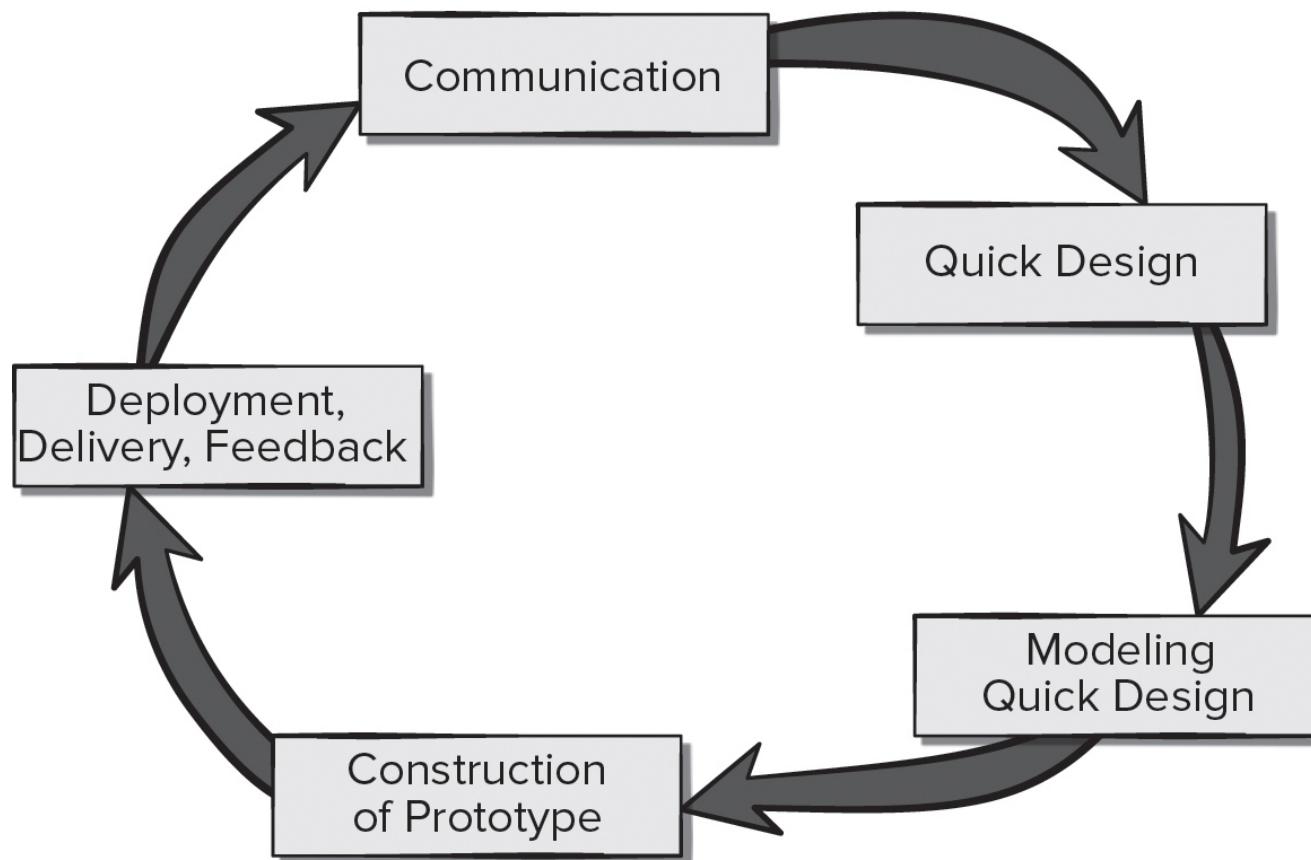


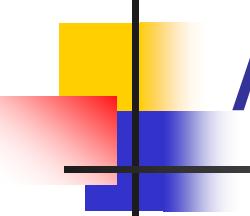
# Planning Principles

- **Principle #7.** Adjust granularity as you define the plan. Granularity refers to the level of detail that is introduced as a project plan is developed.
- **Principle #8.** Define how you intend to ensure quality. Your plan should identify how the software team intends to ensure quality.
- **Principle #9.** Describe how you intend to accommodate change. Even the best planning can be obviated by uncontrolled change.
- **Principle #10.** Track the plan frequently and make adjustments as required. Software projects fall behind schedule one day at a time.

# Software Modeling

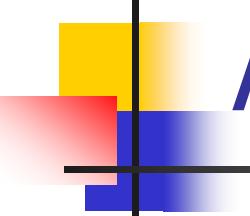
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





# Agile Modeling Principles

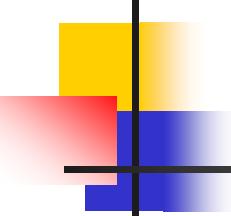
- **Principle #1.** The primary goal of the software team is to build software not create models.
- **Principle #2.** Travel light – don't create more models than you need.
- **Principle #3.** Strive to produce the simplest model that will describe the problem or the software.
- **Principle #4.** Build models in a way that makes them amenable to change.
- **Principle #5.** Be able to state an explicit purpose for each model that is created.



# Agile Modeling Principles

- **Principle #6.** Adapt the models you create to the system at hand.
- **Principle #7.** Try to build useful models, forget about building perfect models.
- **Principle #8.** Don't become dogmatic about model syntax. Successful communication is key.
- **Principle #9.** If your instincts tell you a paper model isn't working you may have a reason to be concerned.
- **Principle #10.** Get feedback as soon as you can.

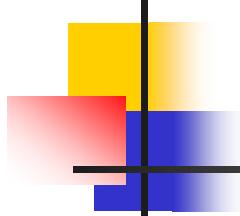
# Construction Principles - Coding



**Preparation Principles:** Before you write one line of code, be sure you:

- **Principle 1.** Understand the problem to be solved.
- **Principle 2.** Understand basic design principles and concepts.
- **Principle 3.** Pick a programming language that meets the needs of the software to be built.
- **Principle 4.** Select a programming environment that provides tools that will make your work easier.
- **Principle 5.** Create a set of unit tests that will be applied once the component you code is completed.

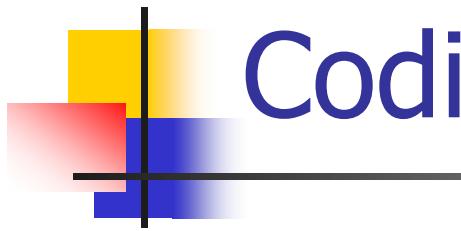
# Construction Principles - Coding



**Coding Principles:** As you begin writing code, be sure you:

- **Principle 6.** Constrain your algorithms by following structured programming practice.
- **Principle 7.** Consider the use of pair programming.
- **Principle 8.** Select data structures that will meet the needs of the design.
- **Principle 9.** Understand the software architecture and create interfaces that are consistent with it.

# Construction Principles - Coding

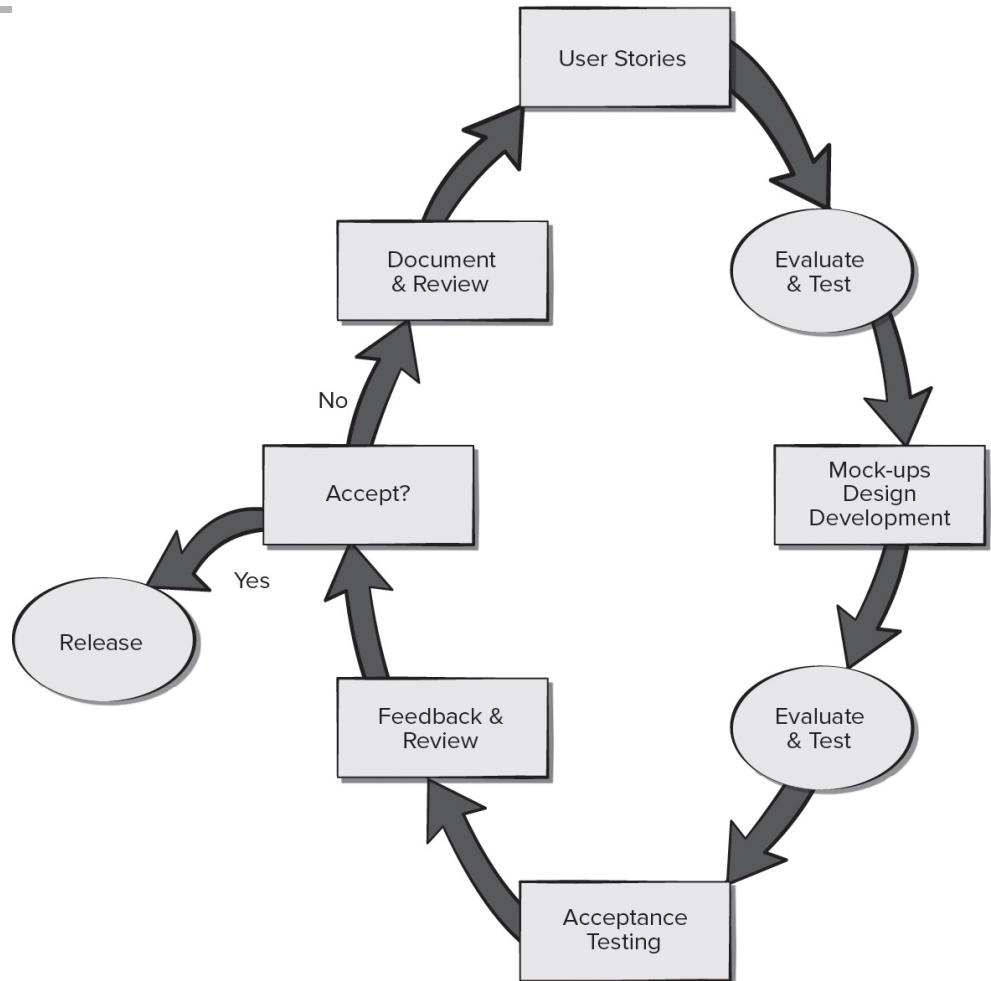


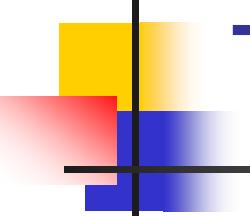
**Validation Principles:** After you've completed your first coding pass, be sure you:

- **Principle 10.** Conduct a code walkthrough when appropriate.
- **Principle 11.** Perform unit tests and correct errors you've uncovered.
- **Principle 12.** Refactor the code to improve its quality.

# Agile Testing

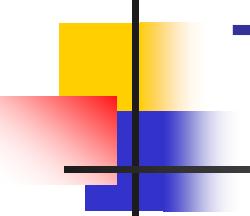
Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.





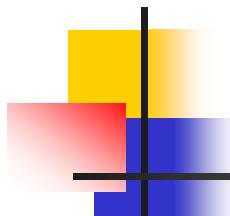
# Testing Principles

- **Principle #1.** All tests should be traceable to customer requirements.
- **Principle #2.** Tests should be planned long before testing begins.
  - Testing is a process of executing a program with intent of finding an error,
  - A good test case is one that has a high probability of finding an as-yet-undiscovered error.
  - A successful test is one that uncovers an as-yet-undiscovered error.
- **Principle #3.** The Pareto principle applies to software testing.



# Testing Principles

- **Principle #4.** Testing should begin “in the small” and progress toward testing “in the large.”
- **Principle #5.** Exhaustive testing is not possible.
- **Principle #6.** Testing effort for each system module commensurate to expected fault density.
- **Principle #7.** Static testing can yield high results.
- **Principle #8.** Track defects and look for patterns in defects uncovered by testing.
- **Principle #9.** Include test cases that demonstrate software is behaving correctly.

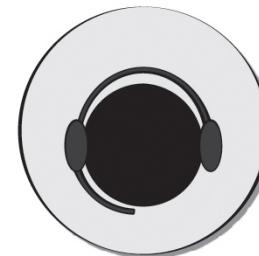


# Software Deployment Actions

Copyright © McGraw-Hill Education. All rights reserved. No reproduction or distribution without the prior written consent of McGraw-Hill Education.



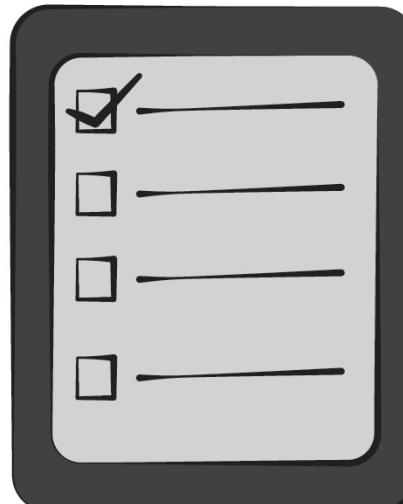
Assemble Deployment Package



Establish Support Regimen



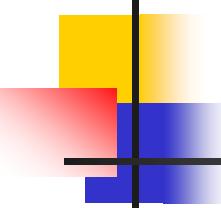
Manage Customer Expectations



Practical Principles

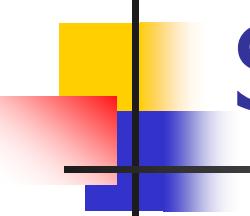


Provide Instructional Materials to End Users



# Deployment Principles

- **Principle #1.** Customer expectations for the software must be managed.
- **Principle #2.** A complete delivery package should be assembled and tested.
- **Principle #3.** A support regime must be established before the software is delivered.
- **Principle #4.** Appropriate instructional materials must be provided to end-users.
- **Principle #5.** Buggy software should be fixed first, delivered later.



# Traits of Successful Software Engineers

- Sense of individual responsibility.
- Acutely aware of the needs of team members and stakeholders.
- Brutally honest about design flaws and offers constructive criticism.
- Resilient under pressure.
- Heightened sense of fairness.
- Attention to detail.
- Pragmatic adapting software engineering practices based on the circumstances at hand.