Final Exam Study Guide

Please note:

The Exam contains a variety of questions: MCQ, T/F, Fill in the Blank, Short and Long Answer Questions

MCQs, T/F & Fill in Blank questions will come from all chapters

Short & Long Answer Q. will be from chapters **4&5, 11, 12, 13, 22, 25, 36, and 37.**

All figures and diagrams (including any descriptions next to them) are very important. You will not be asked to draw a diagram but you're responsible for understanding and knowing its contents.

Chapter 1: The Nature of Software

⊙ Cloud Computing

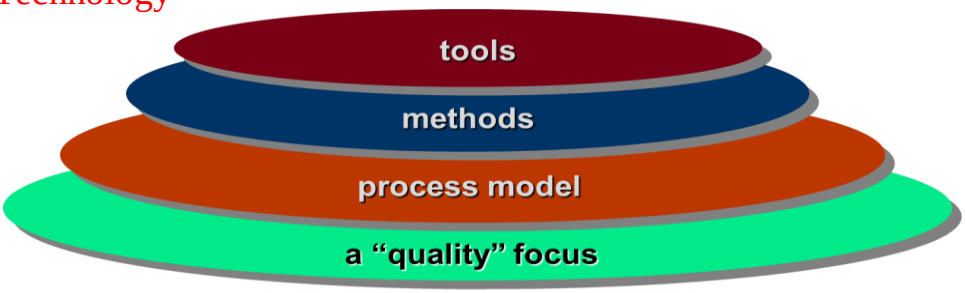
- ✧ *Cloud computing* provides distributed data storage and processing resources to networked computing devices
- ✧ Computing resources reside outside the cloud and have access to a variety of resources inside the cloud
- ✧ Cloud computing requires developing an architecture containing both frontend and backend services
- ✧ Frontend services include the client devices and application software to allow access
- ✧ Backend services include servers, data storage, and server-resident applications
- ✧ Cloud architectures can be segmented to restrict access to private data

⊙ Product Line Software

- ✧ *Product line software* is a set of software-intensive systems that share a common set of features and satisfy the needs of a particular market
- ✧ These software products are developed using the same application and data architectures using a common core of reusable software components
- ✧ A software product line shares a set of assets that include *requirements, architecture, design patterns, reusable components, test cases,* and other work products
- ✧ A software product line allow in the development of many products that are engineered by capitalizing on the commonality among all products with in the product line

Chapter 2: Software Engineering

⊙ A Layered Technology



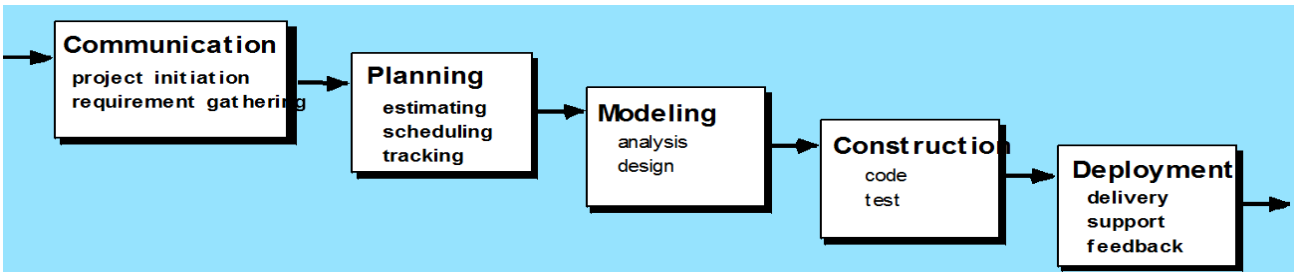
Software Engineering

⊙ Hooker's General Principles

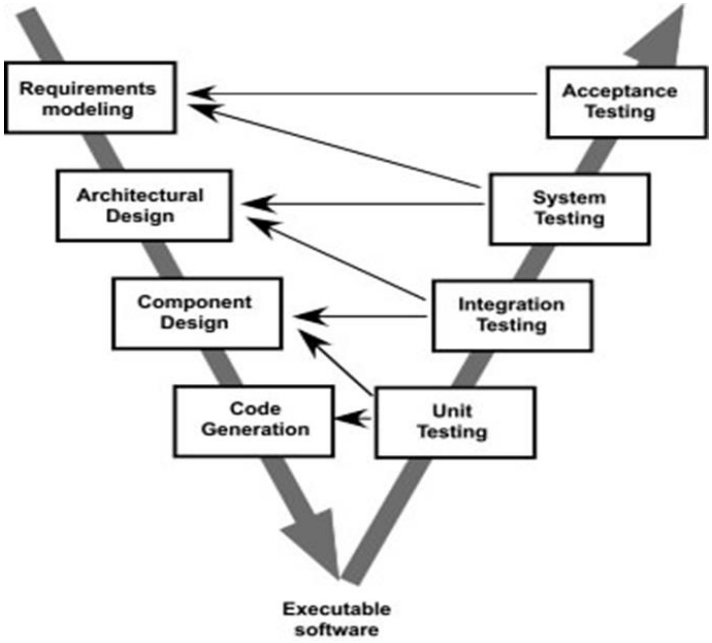
1) The Reason It All Exists
2) KISS (Keep It Simple, Stupid!)
3) Maintain the Vision
4) What You Produce, Others Will Consume
5) Be Open to the Future
6) Plan Ahead for Reuse
7) Think!

Chapter 4 & Chapter 5 Important Concepts : **Process Models**

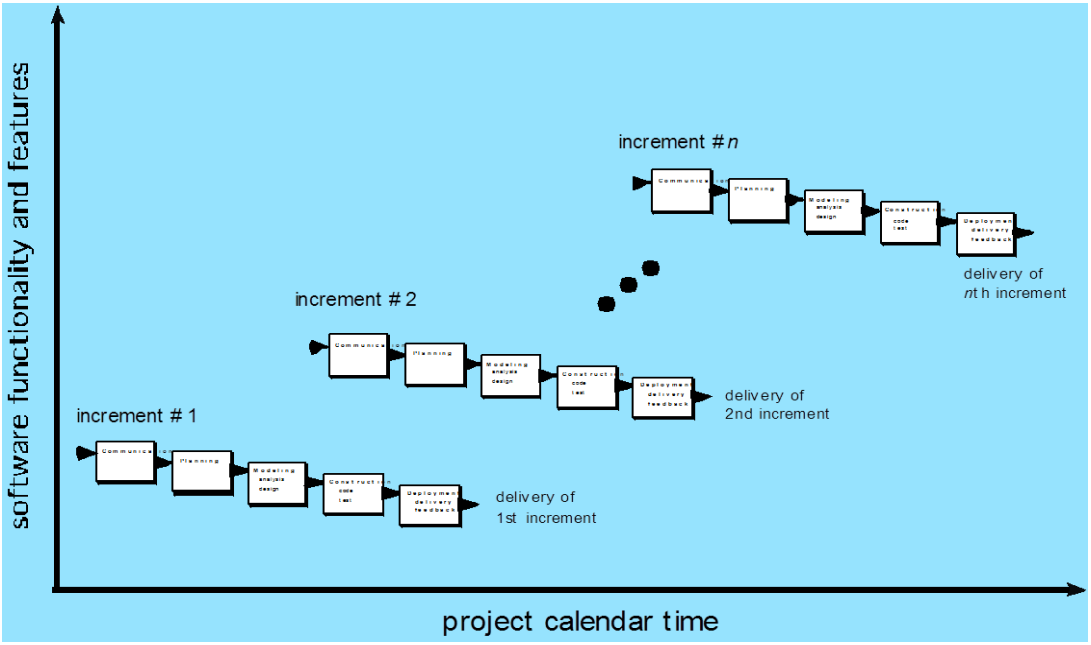
⊙ The Waterfall Model



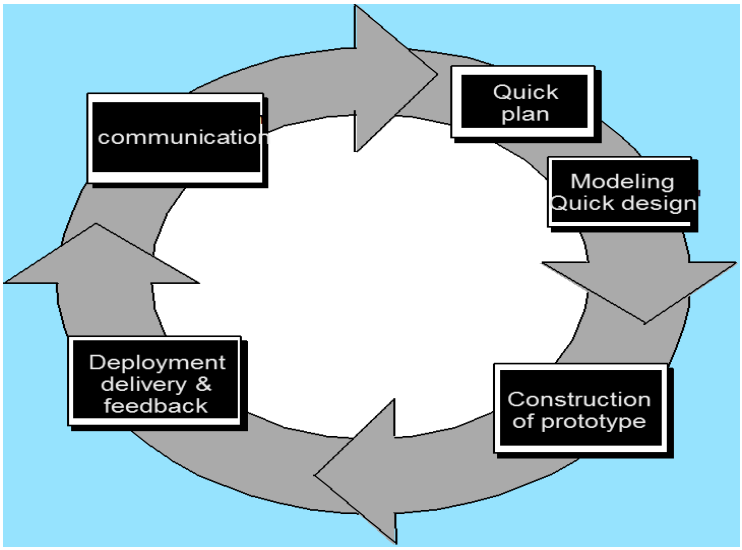
⊙ The V-Model



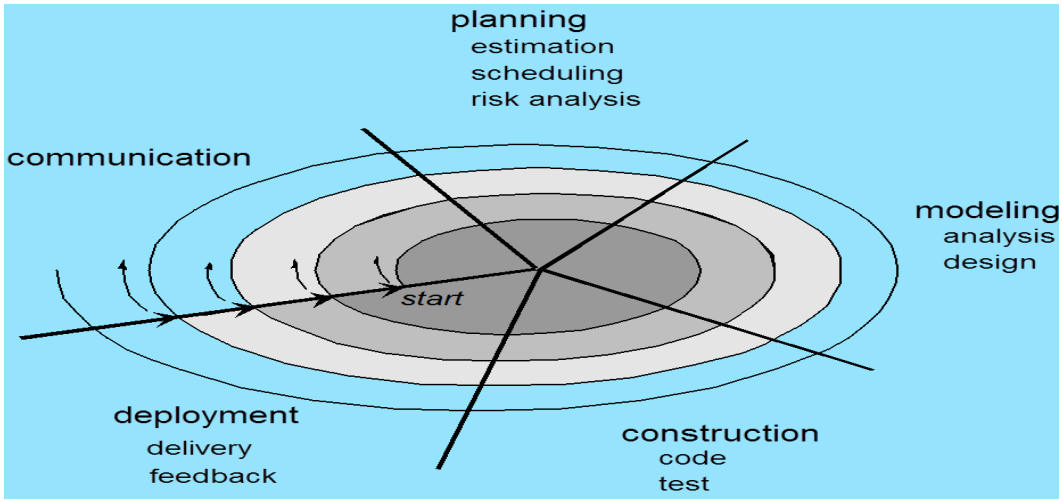
⊙ The Incremental Model



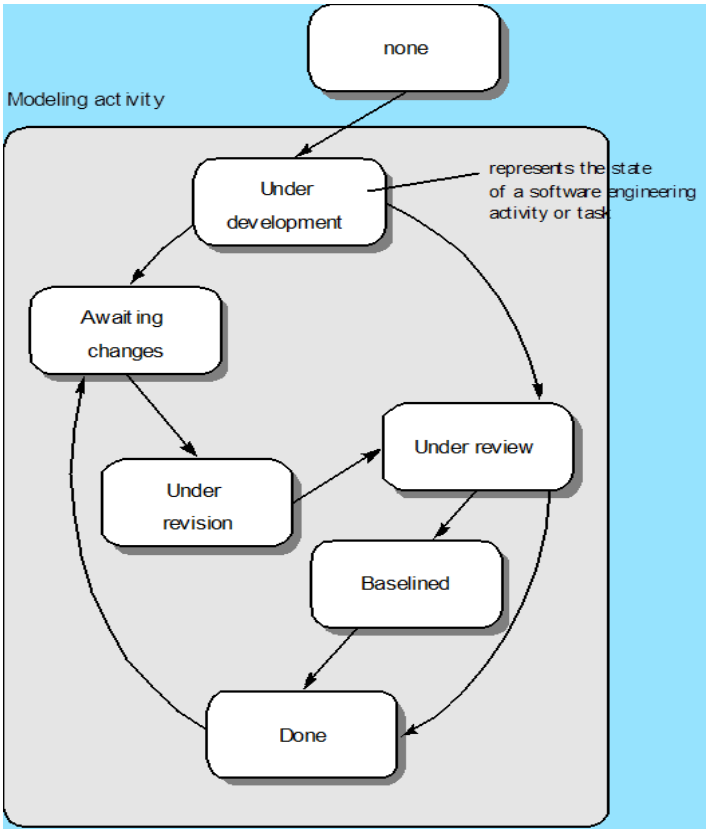
Evolutionary Models: Prototyping



Evolutionary Models: The Spiral



Evolutionary Models: Concurrent



- Extreme Programming (XP)
  - The most widely used agile process, originally proposed by Kent Beck
    - XP Planning
      - Begins with the creation of "user stories"
      - Agile team assesses each story and assigns a cost
      - Stories are grouped to for a deliverable increment
      - A commitment is made on delivery date
      - After the first increment "project velocity" is used to help define subsequent delivery dates for other increments
    - XP Design
      - Follows the KIS principle
      - Encourage the use of CRC cards (see Chapter 8)
      - For difficult design problems, suggests the creation of "spike solutions"—a design prototype
      - Encourages "refactoring"—an iterative refinement of the internal program design
    - XP Coding
      - Recommends the construction of a unit test for a store *before* coding commences
      - Encourages "pair programming"
    - XP Testing
      - All unit tests are executed daily
      - "Acceptance tests" are defined by the customer and excuted to assess customer visible functionality

Chapter 6 : **Human Aspects of Software Engineering**

- Avoid Team "Toxicity"
  - A frenzied work atmosphere in which team members waste energy and lose focus on the objectives of the work to be performed.
  - High frustration caused by personal, business, or technological factors that cause friction among team members.
  - "Fragmented or poorly coordinated procedures" or a poorly defined or improperly chosen process model that becomes a roadblock to accomplishment.
  - Unclear definition of roles resulting in a lack of accountability and resultant finger-pointing.
  - "Continuous and repeated exposure to failure" that leads to a loss of confidence and a lowering of morale.
- Organizational Paradigms
  - closed paradigm—structures a team along a traditional hierarchy of authority
  - random paradigm—structures a team loosely and depends on individual initiative of the team members
  - open paradigm—attempts to structure a team in a manner that achieves some of the controls associated with the closed paradigm but also much of the innovation that occurs when using the random paradigm
  - synchronous paradigm—relies on the natural compartmentalization of a problem and organizes team members to work on pieces of the problem with little active communication among themselves

Chapter 7 : **Principles that Guide Practice**

⊙ Planning Principles

- ◊ Principle #1. Understand the scope of the project. It's impossible to use a roadmap if you don't know where you're going. Scope provides the software team with a destination.
- ◊ Principle #2. Involve the customer in the planning activity. The customer defines priorities and establishes project constraints.
- ◊ Principle #3. Recognize that planning is iterative. A project plan is never engraved in stone. As work begins, it very likely that things will change.
- ◊ Principle #4. Estimate based on what you know. The intent of estimation is to provide an indication of effort, cost, and task duration, based on the team's current understanding of the work to be done.

⊙ Coding Principles

▪ ***As you begin writing code, be sure you:***

- Constrain your algorithms by following structured programming [Boh00] practice.
- Consider the use of pair programming
- Select data structures that will meet the needs of the design.
- Understand the software architecture and create interfaces that are consistent with it.
- Keep conditional logic as simple as possible.
- Create nested loops in a way that makes them easily testable.
- Select meaningful variable names and follow other local coding standards.
- Write code that is self-documenting.
- Create a visual layout (e.g., indentation and blank lines) that aids understanding.

Chapter 8 : **Understanding Requirements**

⊙ Requirements Engineering-I

♣ Inception—ask a set of questions that establish …

- o basic understanding of the problem
- o the people who want a solution
- o the nature of the solution that is desired, and
- o the effectiveness of preliminary communication and collaboration between the customer and the developer

♣ Elicitation—elicit requirements from all stakeholders

♣ Elaboration—create an analysis model that identifies data, function and behavioral requirements

♣ Negotiation—agree on a deliverable system that is realistic for developers and customers

⊙ Negotiating Requirements

♣ Identify the key stakeholders

- ▪ These are the people who will be involved in the negotiation

♣ Determine each of the stakeholders "win conditions"

- ▪ Win conditions are not always obvious

♣ Negotiate

- ▪ Work toward a set of requirements that lead to "win-win"

Chapter 9 : **Requirements Modeling: Scenario-Based Methods**

⊙ Use-Cases

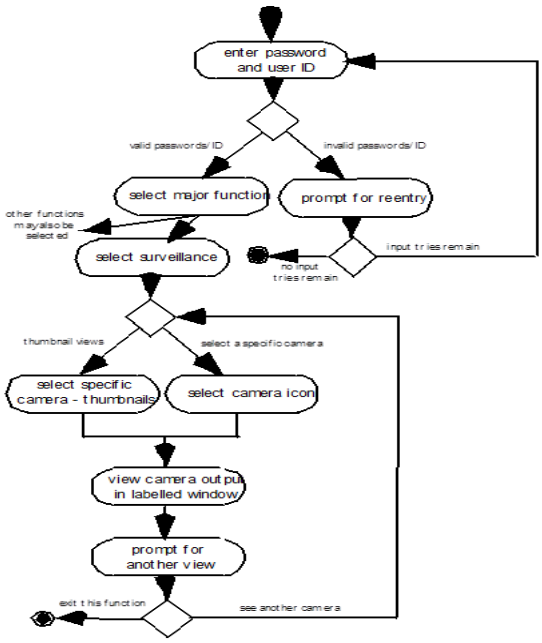
▪ a scenario that describes a "thread of usage" for a system

▪ actors represent roles people or devices play as the system functions

▪ users can play a number of different roles for a given scenario

⊙ Activity Diagram

Supplements the use case by providing a graphical representation of the flow of interaction within a specific scenario



Chapter 10 : **Requirements Modeling: Class-Based Methods**

⊙ Potential Classes

- ✧ Retained information. The potential class will be useful during analysis only if information about it must be remembered so that the system can function.
- ✧ Needed services. The potential class must have a set of identifiable operations that can change the value of its attributes in some way.
- ✧ Multiple attributes. During requirement analysis, the focus should be on "major" information; a class with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another class during the analysis activity.
- ✧ Common attributes. A set of attributes can be defined for the potential class and these attributes apply to all instances of the class.
- ✧ Common operations. A set of operations can be defined for the potential class and these 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 operation of any solution for the system will almost always be defined as classes in the requirements model.

⊙ Dependencies

Chapter 11 : **Requirements Modeling: Behavior, Patterns, and Web/Mobile Apps**

⊙ State Representations

♣ In the context of behavioral modeling, two different characterizations of states must be considered:

- the state of each class as the system performs its function and
- the state of the system as observed from the outside as the system performs its function

♣ The state of a class takes on both passive and active characteristics [CHA93].

- A *passive state* is simply the current status of all of an object's attributes.
- The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing.

⊙ The Content Model

- ♦ Content objects are extracted from use-cases
  - examine the scenario description for direct and indirect references to content
- ♦ Attributes of each content object are identified
- ♦ The relationships among content objects and/or the hierarchy of content maintained by a WebApp
  - Relationships—entity-relationship diagram or UML
  - Hierarchy—data tree or UML

⊙ The Interaction Model

- ♦ Composed of four elements:
  - ✧ use-cases
  - ✧ sequence diagrams
  - ✧ state diagrams
  - ✧ a user interface prototype
- ♦ Each of these is an important UML notation and is described in Appendix I

Chapter 12 : **Design Concepts**

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

Fundamental Concepts

- ✧ Abstraction—data, procedure, control
- ✧ Architecture—the overall structure of the software
- ✧ Patterns—"conveys the essence" of a proven design solution
- ✧ Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- ✧ Modularity—compartmentalization of data and function
- ✧ Hiding—controlled interfaces
- ✧ Functional independence—single-minded function and low coupling
- ✧ Refinement—elaboration of detail for all abstractions
- ✧ Aspects—a mechanism for understanding how global requirements affect design
- ✧ Refactoring—a reorganization technique that simplifies the design
- ✧ OO design concepts—Appendix II
- ✧ Design Classes—provide design detail that will enable analysis classes to be implemented

Separation of Concerns

- ✧ Any complex problem can be more easily handled if it is subdivided into pieces that can each be solved and/or optimized independently
- ✧ A concern is a feature or behavior that is specified as part of the requirements model for the software
- ✧ By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

Modularity

- "modularity is the single attribute of software that allows a program to be intellectually manageable"
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.
  - The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

Functional Independence

- Functional independence is achieved by developing modules with "single-minded" function and an "aversion" to excessive interaction with other modules.
- *Cohesion* is an indication of the relative functional strength of a module.
  - A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- *Coupling* is an indication of the relative interdependence among modules.
  - Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.

Aspects

- Consider two requirements, *A* and *B*. Requirement *A crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account. [Ros04]
- An aspect is a representation of a cross-cutting concern.

OO Design Concepts

- Design classes
  - Entity classes
  - Boundary classes
  - Controller classes
- Inheritance —all responsibilities of a superclass is immediately inherited by all subclasses
- Messages —stimulate some behavior to occur in the receiving object
- Polymorphism —a characteristic that greatly reduces the effort required to extend the design

Chapter 13 : **Architectural Design**

Why Architecture?

The architecture is not the operational software. Rather, 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.

Architectural Genres

- *Genre* implies a specific category within the overall software domain.
- Within each category, you encounter a number of subcategories.
  - For example, within the genre of *buildings*, you would encounter the following general styles: houses, condos, apartment buildings, office buildings, industrial building, warehouses, and so on.

- Within each general style, more specific styles might apply. Each style would have a structure that can be described using a set of predictable patterns.

Architectural Styles

✧ Each style describes a system category that encompasses: (1) a **set of components** (e.g., a database, computational modules) that perform a function required by a system, (2) a **set of connectors** that enable "communication, coordination and cooperation" among components, (3) **constraints** that define how components can be integrated to form the system, and (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 architectures
- Data flow architectures
- Call and return architectures
- Object-oriented architectures
- Layered architectures

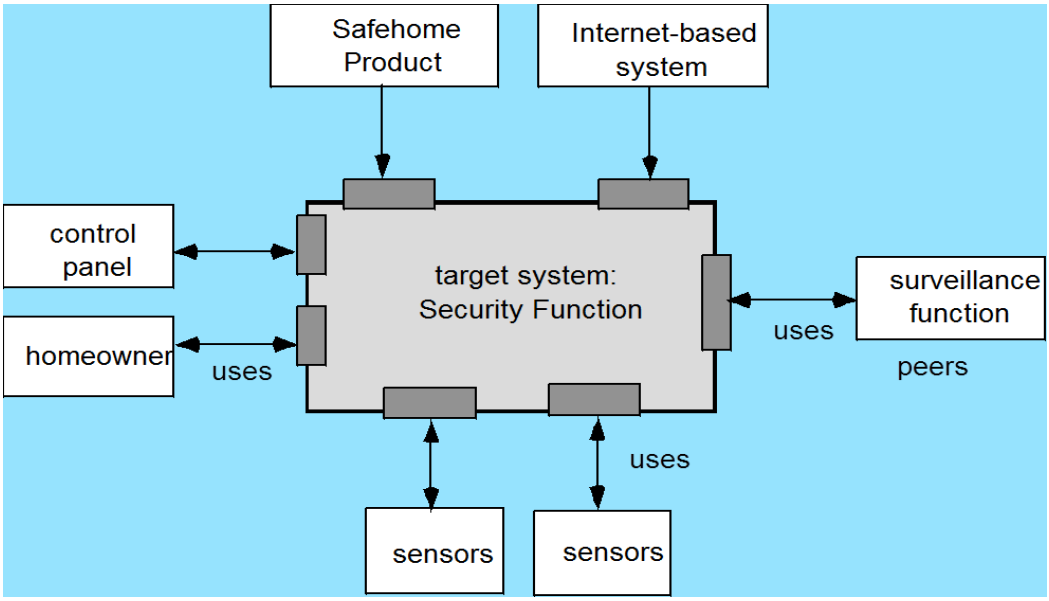
Architectural Design

✧ 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

Architectural Context



ADL

✧ Architectural description language (ADL) provides a semantics and syntax for describing a software architecture

✧ Provide the designer with the ability to:
- decompose architectural components
- compose individual components into larger architectural blocks and
- represent interfaces (connection mechanisms) between components.

Chapter 17 : **WebApp Design**

Design & WebApps

"There are essentially two basic approaches to design: the artistic ideal of expressing yourself and the engineering ideal of solving a problem for a customer." *Jakob Nielsen*

When should we emphasize WebApp design?

- ✧ when content and function are complex
- ✧ when the size of the WebApp encompasses hundreds of content objects, functions, and analysis classes
- ✧ when the success of the WebApp will have a direct impact on the success of the business

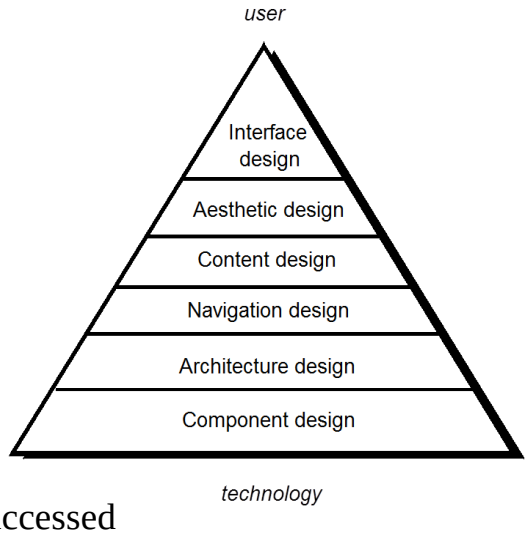
Design & WebApp Quality

- ✧ Security
  - Rebuff external attacks
  - Exclude unauthorized access
  - Ensure the privacy of users/customers
- ✧ Availability
  - the measure of the percentage of time that a WebApp is available for use
- ✧ Scalability
  - **Can** the WebApp and the systems with which it is interfaced handle significant variation in user or transaction volume
- ✧ Time to Market

Quality Dimensions for End-Users

- ✧ **Time**
  - How much has a Web site changed since the last upgrade?
  - How do you highlight the parts that have changed?
- ✧ **Structural**
  - How well do all of the parts of the Web site hold together.
  - Are all links inside and outside the Web site working?
  - Do all of the images work?
  - Are there parts of the Web site that are not connected?
- ✧ **Content**
  - Does the content of critical pages match what is supposed to be there?
  - Do key phrases exist continually in highly-changeable pages?
  - Do critical pages maintain quality content from version to version?
  - What about dynamically generated HTML pages?
  - Quality Dimensions for End-Users
- ✧ **Accuracy and Consistency**
  - Are today's copies of the pages downloaded the same as yesterday's? Close enough?
  - Is the data presented accurate enough? How do you know?
- ✧ **Response Time and Latency**
  - Does the Web site server respond to a browser request within certain parameters?
  - In an E-commerce context, how is the end to end response time after a SUBMIT?
  - Are there parts of a site that are so slow the user declines to continue working on it?
- ✧ **Performance**
  - Is the Browser-Web-Web site-Web-Browser connection quick enough?
  - How does the performance vary by time of day, by load and usage?
  - Is performance adequate for E-commerce applications?

WebE Design Pyramid



WebApp Interface Design

- Where am I? The interface should
  - provide an indication of the WebApp that has been accessed
  - inform the user of her location in the content hierarchy.
- What can I do now? The interface should always help the user understand his current options
  - what functions are available?
  - what links are live?
  - what content is relevant?
- Where have I been, where am I going? The interface must facilitate navigation.
  - Provide a "map" (implemented in a way that is easy to understand) of where the user has been and what paths may be taken to move elsewhere within the WebApp.

Effective WebApp Interfaces

- Bruce Tognazzi [TOG01] suggests…
  - Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.
  - Effective interfaces do not concern the user with the inner workings of the system. Work is carefully and continuously saved, with full option for the user to undo any activity at any time.
  - Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

Architecture Design

- Content architecture focuses on the manner in which content objects (or composite objects such as Web pages) are structured for presentation and navigation.
  - The term information architecture is also used to connote structures that lead to better organization, labeling, navigation, and searching of content objects.
- WebApp architecture addresses the manner in which the application is structured to manage user interaction, handle internal processing tasks, effect navigation, and present content.
- Architecture design is conducted in parallel with interface design, aesthetic design and content design.

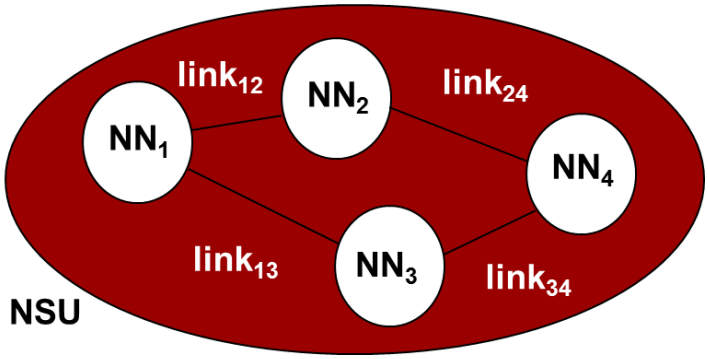
Navigation Design

- Begins with a consideration of the user hierarchy and related use-cases
  - Each actor may use the WebApp somewhat differently and therefore have different navigation requirements
- As each user interacts with the WebApp, she encounters a series of navigation semantic units (NSUs)

- NSU—"a set of information and related navigation structures that collaborate in the fulfillment of a subset of related user requirements"

Navigation Semantic Units

✧ Ways of navigation (WoN)—represents the best navigation way or path for users with certain profiles to achieve their desired goal or sub-goal. Composed of …
  - Navigation nodes (NN) connected by Navigation links



Navigation Syntax

✧ Individual navigation link—text-based links, icons, buttons and switches, and graphical metaphors..

✧ Horizontal navigation bar—lists major content or functional categories in a bar containing appropriate links. In general, between 4 and 7 categories are listed.

✧ Vertical navigation column
  - lists major content or functional categories
  - lists virtually all major content objects within the WebApp.

✧ Tabs—a metaphor that is nothing more than a variation of the navigation bar or column, representing content or functional categories as tab sheets that are selected when a link is required.

✧ Site maps—provide an all-inclusive tab of contents for navigation to all content objects and functionality contained within the WebApp.

Chapter 15 : **User Interface Design**

✧ Reduce the 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.

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

Interface Analysis

✧ Interface analysis means understanding
  - the people (end-users) who will interact with the system through the interface;
  - the tasks that end-users must perform to do their work,
  - the content that is presented as part of the interface
  - the environment in which these tasks will be conducted.

Task Analysis and Modeling

- Answers 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?
- Use-cases define basic interaction
- Task elaboration refines interactive tasks
- Object elaboration identifies interface objects (classes)
- Workflow analysis defines how a work process is completed when several people (and roles) are involved

Chapter 22 : **Software Testing Strategies**

V & V

- Verification refers to the set of tasks that ensure that software correctly implements a specific function.
- Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
  - Verification: "Are we building the product right?"
  - Validation: "Are we building the right product?"

General Testing Criteria

- Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity – test to uncover functional defects in the software
- Information content – test for errors in local or global data structures
- Performance – verify specified performance bounds are tested

OO Testing Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

MobileApp Testing

- User experience testing – ensuring app meets stakeholder usability and accessibility expectations
- Device compatibility testing – testing on multiple devices
- Performance testing – testing non-functional requirements
- Connectivity testing – testing ability of app to connect reliably
- Security testing – ensuring app meets stakeholder security expectations
- Testing-in-the-wild – testing app on user devices in actual user environments
- Certification testing – app meets the distribution standards

High Order Testing

- Validation testing
  - Focus is on software requirements

- System testing
  - Focus is on system integration
- Alpha/Beta testing
  - Focus is on customer usage
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- Performance Testing
  - test the run-time performance of software within the context of an integrated system

Chapter 25 : **Testing Web Applications**

Testing Quality Dimensions

- **Content** is evaluated at both a syntactic and semantic level.
  - syntactic level—spelling, punctuation and grammar are assessed for text-based documents.
  - semantic level—correctness (of information presented), consistency (across the entire content object and related objects) and lack of ambiguity are all assessed.
- **Function i**s tested for correctness, instability, and general conformance to appropriate implementation standards (e.g.,Java or XML language standards).
- **Structure** is assessed to ensure that it
  - properly delivers WebApp content and function
  - is extensible
  - can be supported as new content or functionality is added.
- **Usability** is tested to ensure that each category of user
  - is supported by the interface
  - can learn and apply all required navigation syntax and semantics
- **Navigability** is tested to ensure that
  - all navigation syntax and semantics are exercised to uncover any navigation errors (e.g., dead links, improper links, erroneous links).
- **Performance** is tested under a variety of operating conditions, configurations, and loading to ensure that
  - the system is responsive to user interaction
  - the system handles extreme loading without unacceptable operational degradation
- **Compatibility** is tested by executing the WebApp in a variety of different host configurations on both the client and server sides.
  - The intent is to find errors that are specific to a unique host configuration.
- **Interoperability** is tested to ensure that the WebApp properly interfaces with other applications and/or databases.
- **Security** is tested by assessing potential vulnerabilities and attempting to exploit each.
  - Any successful penetration attempt is deemed a security failure.

WebApp Testing Strategy

- The content model for the WebApp is reviewed to uncover errors.

- ✧ The interface model is reviewed to ensure that all use-cases can be accommodated.
- ✧ The design model for the WebApp is reviewed to uncover navigation errors.
- ✧ The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- ✧ Selected functional components are unit tested.

Testing Interface Mechanisms

- ✧ **CGI scripts**—a common gateway interface (CGI) script implements a standard method that allows a Web server to interact dynamically with users (e.g., a WebApp that contains forms may use a CGI script to process the data contained in the form once it is submitted by the user).
- ✧ **Streaming content**—rather than waiting for a request from the client-side, content objects are downloaded automatically from the server side. This approach is sometimes called "push" technology because the server pushes data to the client.
- ✧ **Cookies**—a block of data sent by the server and stored by a browser as a consequence of a specific user interaction. The content of the data is WebApp-specific (e.g., user identification data or a list of items that have been selected for purchase by the user).
- ✧ **Application specific interface mechanisms**—include one or more "macro" interface mechanisms such as a shopping cart, credit card processing, or a shipping cost calculator.

Compatibility Testing

- ✧ Compatibility testing is to define a set of "commonly encountered" client side computing configurations and their variants
- ✧ Create a tree structure identifying
  - each computing platform
  - typical display devices
  - the operating systems supported on the platform
  - the browsers available
  - likely Internet connection speeds
  - similar information.
- ✧ Derive a series of compatibility validation tests
  - derived from existing interface tests, navigation tests, performance tests, and security tests.
  - intent of these tests is to uncover errors or execution problems that can be traced to configuration differences.

Stress Testing

- ✧ Does the system degrade 'gently' or does the server shut down as capacity is exceeded?
- ✧ Does server software generate "server not available" messages? More generally, are users aware that they cannot reach the server?
- ✧ Does the server queue requests for resources and empty the queue once capacity demands diminish?
- ✧ Are transactions lost as capacity is exceeded?
- ✧ Is data integrity affected as capacity is exceeded?
- ✧ What values of *N*, *T*, and *D* force the server environment to fail? How does failure manifest itself? Are automated notifications sent to technical support staff at the server site?
- ✧ If the system does fail, how long will it take to come back on-line?
- ✧ Are certain WebApp functions (e.g., compute intensive functionality, data streaming capabilities) discontinued as capacity reaches the 80 or 90 percent level?

Chapter 19 : **Quality Concepts**

Quality

- The American Heritage Dictionary defines quality as
  - "a characteristic or attribute of something."
- For software, two 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

Measuring Quality

- General quality dimensions and factors are not adequate for assessing the quality of an application in concrete terms
- Project teams need to develop a set of targeted questions to assess the degree to which each application quality factor has been satisfied
- Subjective measures of software quality may be viewed as little more than personal opinion
- Software metrics represent indirect measures of some manifestation of quality and attempt to quantify the assessment of software quality

"Good Enough" Software

- Good enough software delivers high quality functions and features that end-users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs.
- Arguments *against* "good enough."
  - It is true that "good enough" may work in some application domains and for a few major software companies. After all, if a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in locking them in.
  - If you work for a small company be wary of this philosophy. If you deliver a "good enough" (buggy) product, you risk permanent damage to your company's reputation.
  - You may never get a chance to deliver version 2.0 because bad buzz may cause your sales to plummet and your company to fold.
  - If you work in certain application domains (e.g., real time embedded software, application software that is integrated with hardware can be negligent and open your company to expensive litigation.

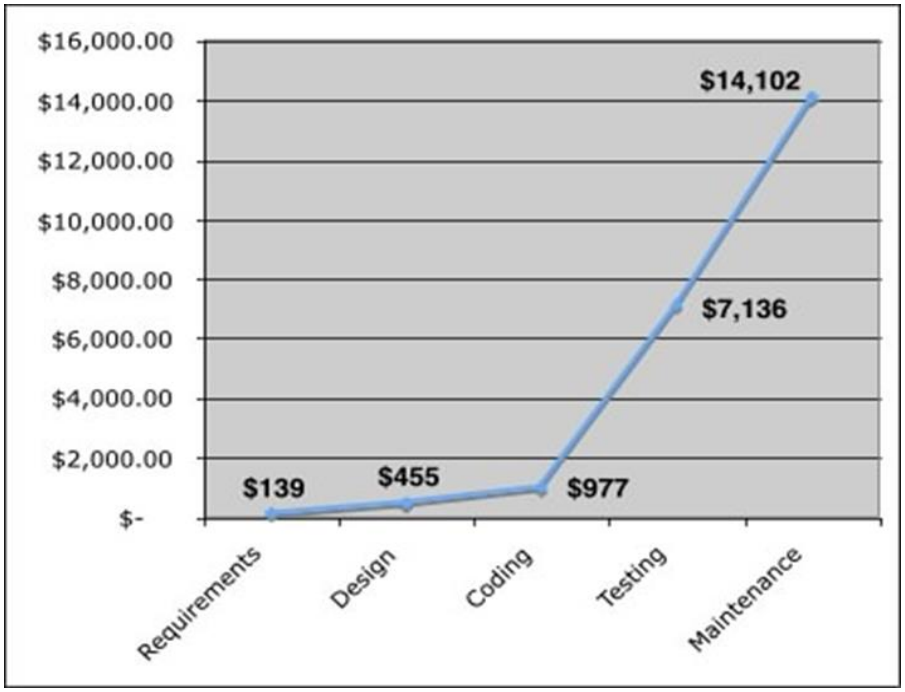
Cost of Quality

- **Prevention costs** include :-
  - quality planning
  - formal technical reviews
  - test equipment
  - Training
- **Internal failure costs** include :-
  - rework
  - repair
  - failure mode analysis
- **External failure costs** are :-
  - complaint resolution
  - product return and replacement

- help line support
- warranty work

Cost

✧ The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.



Low Quality Software

✧ 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 the design phase
✧ Low quality software is liable to contain architectural flaws as well as implementation problems (bugs)

Chapter 36 : **Maintenance and Reengineering**

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 and conventions, leading to source code that is self-documenting and understandable.
✧ It has undergone a variety of quality assurance techniques that have uncovered potential maintenance problems before the software is released.
✧ It has been created by software engineers who recognize that they may not be around when changes must be made.
  - Therefore, the design and implementation of the software must "assist" the person who is making the change

Software Supportability

✧ "the capability of supporting a software system over its whole product life.
  - This implies satisfying any necessary needs or requirements, but also the provision of equipment, support infrastructure, additional software, facilities, manpower, or any

other resource required to maintain the software operational and capable of satisfying its function."

- ✧ The software should contain facilities to assist support personnel when a defect is encountered in the operational environment (and make no mistake, defects *will* be encountered).
- ✧ Support personnel should have access to a database that contains records of all defects that have already been encountered—their characteristics, cause, and cure.

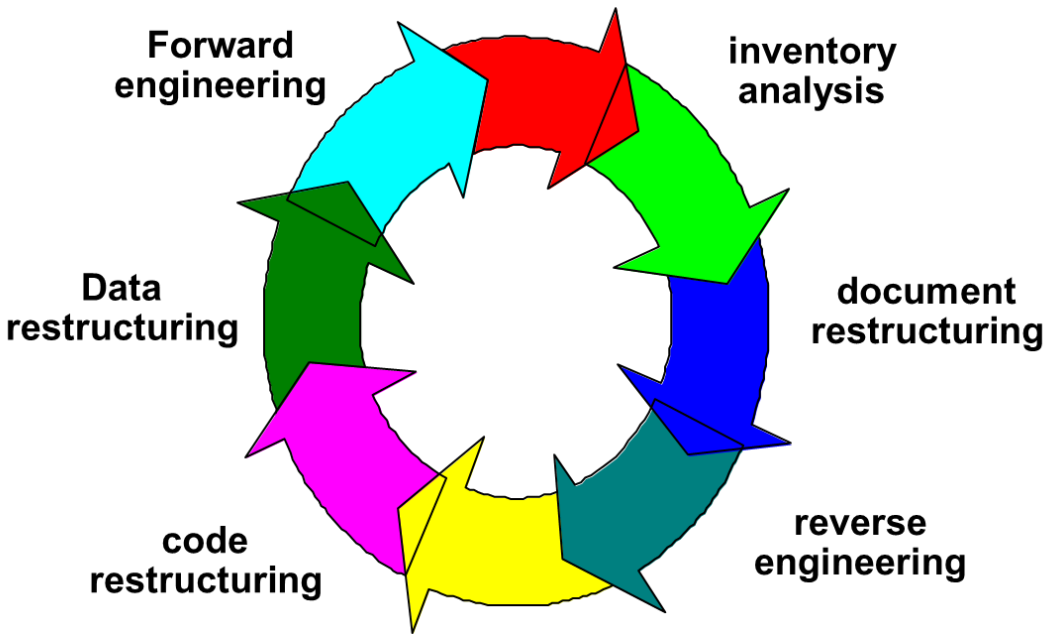
Business Process Reengineering

- ✧ **Business definition.** Business goals are identified within the context of four key drivers: cost reduction, time reduction, quality improvement, and personnel development and empowerment.
- ✧ **Process identification.** Processes that are critical to achieving the goals defined in the business definition are identified.
- ✧ **Process evaluation.** The existing process is thoroughly analyzed and measured.
- ✧ **Process specification and design.** Based on information obtained during the first three BPR activities, use-cases are prepared for each process that is to be redesigned.
- ✧ **Prototyping.** A redesigned business process must be prototyped before it is fully integrated into the business.
- ✧ **Refinement and instantiation.** Based on feedback from the prototype, the business process is refined and then instantiated within a business system.

BPR Principles

- ✧ Organize around outcomes, not tasks.
- ✧ Have those who use the output of the process perform the process.
- ✧ Incorporate information processing work into the real work that produces the raw information.
- ✧ Treat geographically dispersed resources as though they were centralized.
- ✧ Link parallel activities instead of integrated their results. When different
- ✧ Put the decision point where the work is performed, and build control into the process.
- ✧ Capture data once, at its source.

Software Reengineering

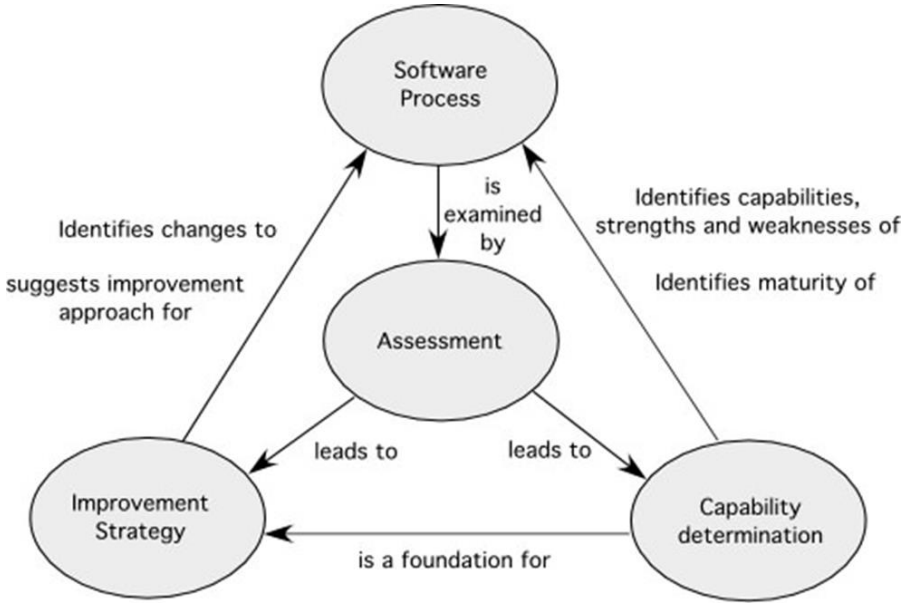


Code Restructuring

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

Chapter 37 : **Software Process Improvement**

Elements of a SPI Framework



Maturity Models

- ✧ A maturity model is applied within the context of an SPI framework.
- ✧ The intent of the maturity model is to provide an overall indication of the "process maturity" exhibited by a software organization.
  - an indication of the quality of the software process, the degree to which practitioner's understand and apply the process,
  - the general state of software engineering practice.

The SPI Process

- ✧ **Installation/Migration**
  - actually *software process redesign* (SPR) activities. Scacchi [Sca00] states that "SPR is concerned with identification, application, and refinement of new ways to dramatically improve and transform software processes."
- ✧ three different process models are considered:
  - the existing ("as-is") process,
  - a transitional ("here-to-there") process, and
  - the target ("to be") process.
- ✧ **Evaluation**
  - assesses the degree to which changes have been instantiated and adopted,
  - the degree to which such changes result in better software quality or other tangible process benefits, and
  - the overall status of the process and the organizational culture as SPI activities proceed
- ♦ From a qualitative point of view, past management and practitioner attitudes about the software process can be compared to attitudes polled after installation of process changes.