**Chapter1**

**Software is**

- instruction-provide feature function and performance

- data structures-enable program to manipulate information

- documentation-describe use of the program

**Software Products(specification)**

-Generic-general software in market(change decide by developer)

-Customized-specific to meet user needs(change decide by customer)

**Importance of software**

-The economies of all developed nations are dependent on software(42%)

-Expenditure on software represents a significant fraction of GNP in all developed countries.

**Software costs**

-often greater than computer cost

-for longlife software, maintain cost may be several times development cost.

-Software engineering is concerned with cost effective software development

**Importance of SE**

- able to produce reliable and trustworthy systems economically and quickly

- cheaper in the long run

**Software Process Activities**

Development-evolution-specification-validation

**General issues that affect most software**

- Heterogeneity-able to run across network and universal platform.

- Business and social change-world change quickly, be able to change or develop new software

- Security and trust- software intertwined all aspects of our lives, it is essential that we can trust that software

**Software engineering fundamentals**

- understood development process

- understanding what software should do

- should reuse developed software not to write new software if possible

**Chapter2**

**Understanding Requirements**

-this where SE meet the “real world”

-it like listen to someone in foreign language

**Two Key questions**

-what do we want to build?

-how do we write this down?

**Requirements**

-Deliverables-goal(requirements,documents,processes,people)

-Constraints-limitaions(Time,financial,technical,language,system,political)

(Requirements are documented in RS\*)

**Requirement Specification(RS)**

- result of requirements engineering

-use-a contract for the customer,a starting point for a design,criteria for evaluating when the project is done

**Requirement engineering activities**

1. Inception-ask about basic understanding of the problem(who request, who use, who benefit)

2. Elicitation-elicit requirement from all stakeholders(scope, requirements)

- 4 have prio-must,should,could,won’t.

- Obstacles(insufficient access to stakeholder, missing documentation, insufficient time)

3. Elaboration-Create analysis model (document,mathematical, use-case, prototype).

- Good Document(readable, non-ambiguous, modifiable, usable, understandable)

-Three Approaches to documenting requirements(1.Natural language-easy to understand,2.formal language-allow proof correctness, 3.Semi-formal-become popular in software industry)

-**Unified Modelling Language(UML)**

-(static)Structural UML diagrams(class, package, object, component, composite structure, deployment)

-(dynamic)Behavioral UML diagrams(activity, sequence, use-case, state, communication, interaction overview, timing)

4. Negotiation-agree the system that realistic for developers and customers(win-win)

5. Validation-review mechanism(errors, missing information, inconsistencies, unrealistic requirement)

**The sins of the analyst**

-Noise-too much trivial information,redundancy

-Silence-important information are left out

-Contradictions-same aspect said differently

-Ambiguity-phrases has more than 1 meaning

-Forward Reference-things are mentioned before defined later in the text

-Wishful thinking-Describe things that a realistic solution is hard to find

**Chapter3**

**Requirement Modeling**

1.Scenario-based (information actor acquire produce change)-use case,user stories

2.Class (how objects relate to one another)-class diagrams

3.Behavioral (how software will response to external event)-state diagrams,sequence diagrams

4.Flow -DFDS

**Chapter4**

**Why is Architecture Important?**

-software architecture are enabler for communication between all stakeholder

-provide ultimate success of system

**Architectural Genres(software-based)**

-styles-ai, communications, devices, financial, games, OS,etc.

**Architectural Styles**

-Eachstyle describes a system category that encompasses

1. Set of components(perform function required by a system).

2. Set of connectors(enable communication among components)

3. Constraints(define how component integrated to form the system)

4. Semantic models(enable a designer to understand overall properties of a system)

**Architectures**

**Data-centered architecture(repository)**

**ี**-use when large amounts of data are to be shared.

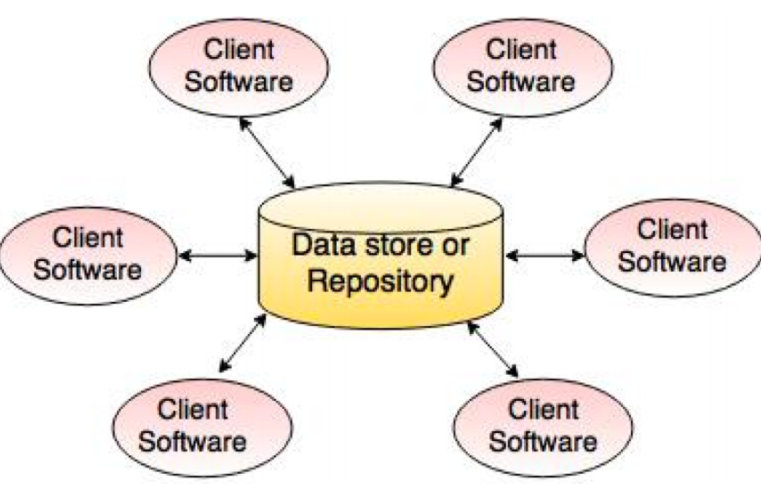
-have center sharing data

- component can be independent

-2ways

1.data held in repository and accessed by all sub-systems

2.each sub-system maintain it own database and pass data explicitly to other sub-systems.



+client don’t need to know existence of others, data can be managed consistently

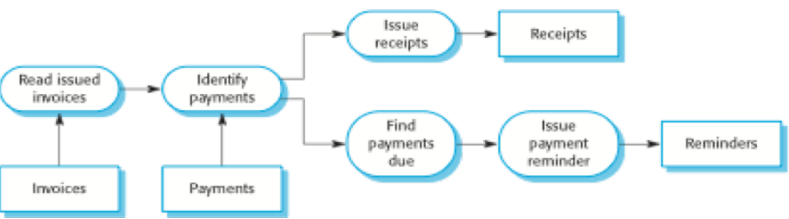
-single point of failure

**Pipe and filter architecture**

-Not really suitable with interactive systems.

-process their inputs to produce outputs

-commonly used in data processing applications

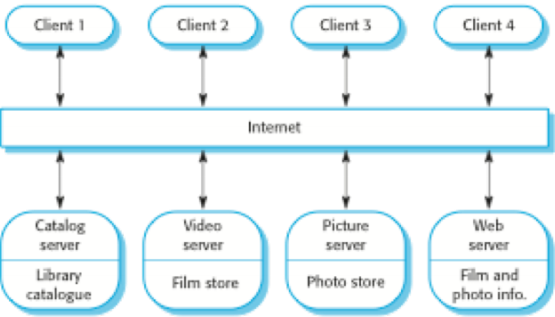


+workflow style match structure of many business, easy to understand

-format for data transfer has to be agreed upon between communicating transformation

**Client-server architecture**

-set of stand-alone servers which provide specific services



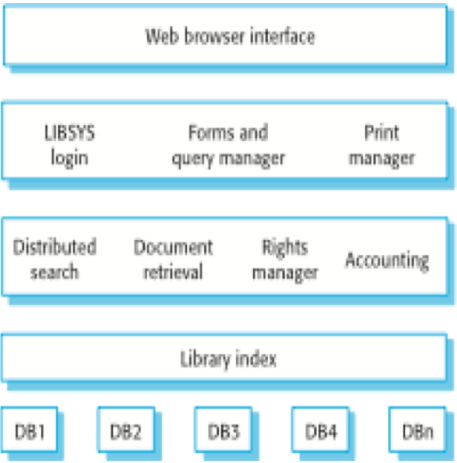
+servers can be distributed across a network, general function can be available to all clients

-each service has a single point of failure

**A generic layered architecture**

-usually use for information management system

-4base layer(UI, UI management(authentication), System utilities, System support)



+allow replacement of entire layers

-performance can be a problem because multiple level of interpretation of a service request as it process at each layer.

**Architectural Considerations**

- Economy - reduce unnecessary details

- Visibility - easy to understand

- Spacing - separate groups of components

- Symmetry - system is balanced in its attributes

- Emergence - flexible, can accommodates the emergent behavior

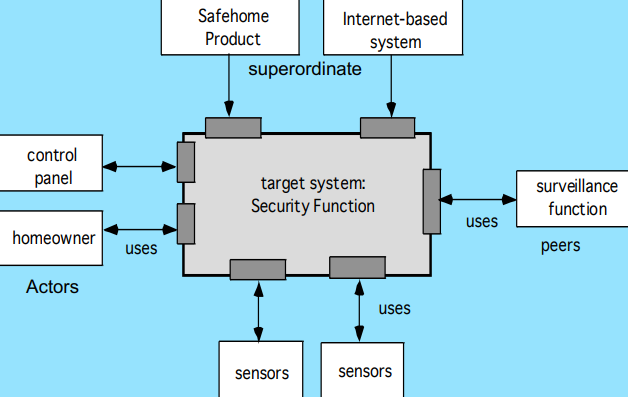
**Architectural context**

- Superordinate systems - superordinate use target

- Subordinate systems - Target use subordinate

- Peer-level system - each uses each other

- Actors - people/devices that interact with the target system



**Chapter5**

**Component**

- Oo view-a component contains a set of collaborating classes

- Conventional view-a component contains processing logic,

**Basic Design Principles**

- The Open-Closed Principle (OCP). “A module [component] should be open for extension but closed for modification.

- The Liskov Substitution Principle (LSP). “Subclasses should be substitutable for their base classes.

- Dependency Inversion Principle (DIP). “Depend on abstractions. Do not depend on concretions.”

- The Interface Segregation Principle (ISP). “Many client-specific interfaces are better than one general purpose interface.

- The Release Reuse Equivalency Principle (REP). “The granule of reuse is the granule of release.”

- The Common Closure Principle (CCP). “Classes that change together belong together.”

- The Common Reuse Principle (CRP). “Classes that aren’t reused together should not be grouped together.”

**Design Guidelines**

- Components-Naming conventions should be established for components

- Interfaces-Interfaces provide important information about communication and collaboration

- Dependencies and Inheritance ◦ it is a good idea to model dependencies from left to right and inheritance from bottom to top

**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. e.g., Collaboration Diagram

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. e.g., Activity diagram

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.

**Domain Engineering**

- is the entire process of reusing domain knowledge in the production of new software systems. (1. Define the domain to be investigated. 2. Categorize the items extracted from the domain. 3. Collect a representative sample of applications in the domain. 4. Analyze each application in the sample. 5. Develop an analysis model for the objects.)

**Qualification**

- Before a component can be used, you must consider(API, integration tools, run-time requirements, OS requirement, embedded design)

- hard to check-the internal workings of commercial components

**Adaptation**

-implication of ‘easy integration’ (1.consistent methods of resource management have been implemented for all components in the library, 2.common activities such as data management exist for all components, 3.interfaces within the architecture and with the external environment have been implemented in a consistent manner)

Architectural ingredients for composition

-include(Data exchange model, Automation, Structured storage, Underlying object model)

**OO DESIGN**

**Encapsulation**

- Encapsulation is enclosing data and processes within one single unit

- Encapsulation enables the object to enforce business rules with authority

**Encapsulation result in 2 spaces**

- Private-process inside the object

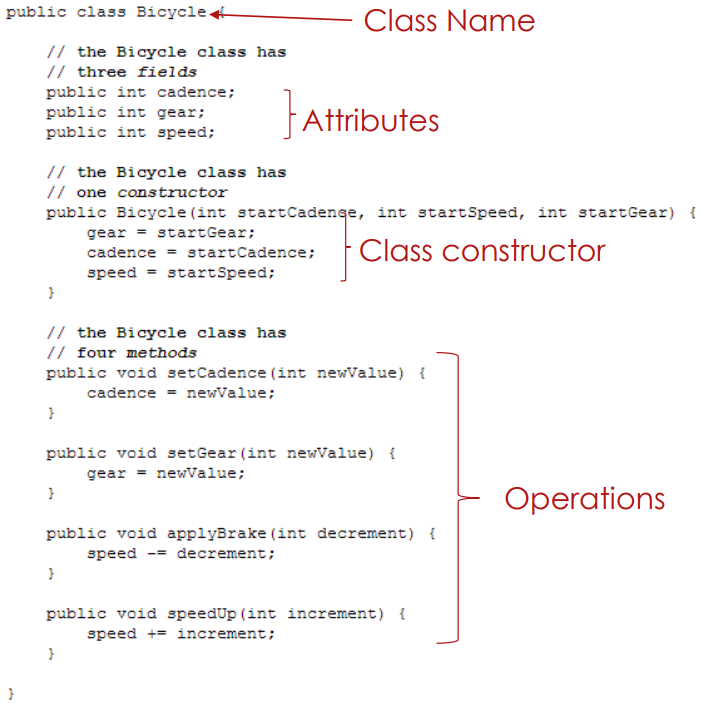
- Public-visible to outside world

**Structuring the Interface**

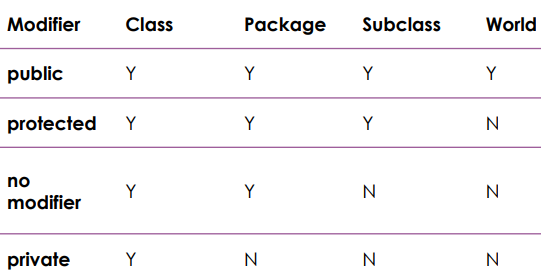
1. Name(always capitalized,noun,singular)

2. Attributes(what an abject know,begin with lower-case letter(firstName))

3. Operations(what an object does,what can be done to object,ex. move(verb), getStarted(verb))



**Visibility**



**Polymorphism**

- means “many forms”

**OO Relationships**





- 2kind of relationship(1.Generalization(parent-child relationship) 2.Association(student enrolls in course)can be classified as aggregation(container) and composition(belong))

**Multiplicity**

**1** one, **0..1** zero or one, **\*** zero to + , **0..\*** zero to +, **1..\*** one to +

**Cohesion VS Coupling**

-Cohesion-class connect to class(within a module)

-Coupling-module connect to module

-Best way-High Cohesion & Low Coupling

**Types of cohesion(Bad to Good)**

- Coincidental cohesion-have no relationship

- Logical cohesion-conceptually related

- Temporal cohesion-perform at the same time

- Procedural cohesion-related by perform step

- Communicational cohesion-related by step & same data

- Functional cohesion-exactly one operation

- Information cohesion-perform number of operation which has same data structure and with independent code for each operation

**Types of Coupling(Bad to Good)**

- Content coupling-one module directly reference the contents of the other

- Common coupling-modules access to the same global data

- Control coupling-one module control the flow of another by passing a control flag to the other

- Stamp coupling-modules share data structure, use some part but whole structure passed to the function

- Data coupling-all data are homogeneous and used by others

**Chapter6**

**UL Design**

- strongly affects perception of software

- An attractive user interface may seem “user friendly” even if it’s not really usable

- Users blame themselves for UI failings

- If the program is slow, or crashes, or gets hacked, user blame SE

**UI is HARD TO DESIGN**

- You are not the user

- The user is always right but not always right because user aren’t designers

**Dimensions of usability**

- Learnability: is it easy to learn?

- Efficiency: once learned, is it fast to use?

- Memorability: is it easy to remember what you learned?

- Errors: are errors few and recoverable?

- Satisfaction: is it enjoyable to use?

**USABILITY DIMENSIONS NOT UNIFORMLY**

- Depends on the user

- Novice users need learnability

- Infrequent users need memorability

- Experts need efficiency

**ITERATIVE DESIGN**

- Design,implement,evaluate

**THE GOLDEN RULES FOR UI DESIGN**

1. Strive for consistency.(menus, color, layout,fonts)

2. Cater to universal usability(age, range, disabilities)

3. Offer informative feedback(feedback for the action)

4. Design dialogs to yield closure.(design move page to page)

5. Prevent errors(Provide recovery for error)

6. Permit easy reversal of actions(action should be reversible)

7. Support internal locus of control(Tell user what to do)

8. Reduce short-term memory load(don’t add too much thing)

**SEVEN RULES FOR CREATING GORGEOUS UI**

1. No teleportation(always use transition)

2. Toggles are better than buttons(on or off)

3. Triggers should be nearby(transition move to his next move)

4. Use natural transition timing(don’t transition too fast)

5. Transition rollbacks should never break the user sense of control(animation should be reverse instantly when user cancel)

6. Always auto-focus on the next action in a series(magically show the next move for user when they click something)

7. Always tell the user when you are done(show feedback to user when they’re done doing something)

**UI Rule**

- Light come from the sky(Shadows are invaluable cues in telling the human brain what user interface elements we’re looking at)

- Flat Design(There are no simulated protrusions or indentations, just lines and shapes of solid color)

- Black and white(This is a reliable and easy way to keep apps looking “clean” and “simple”.) PS.There are some cases where B&WF isn’t as useful. Designs that have a strong specific attitude “sporty”, “flashy”, “cartoony”, etc. need a designer who can use color extremely well

- Color theory(is a body of practical guidance to color mixing and the visual effects of a specific color combination)

**IMPLEMENT**

- Prototyping: Cheap, throw-away implementations

- Paper simulation

- Clickable prototypes

**EVALUATE**

- Expert evaluation(Heuristics and walkthroughs)

- Predictive evaluation(Testing against an engineering model (simulated user))

- Empirical evaluation(Watching users do it)

**USABILITY TESTING**

- is gold standard by which you can determine if the design of an application meets the need of user

**Chapter7**

**Good Software**

- Functionality, Usability, Robustness, Maintainability, Longevity, Cost Effective

**PRINCIPLE OF SOFTWARE VALIDATION**

- Verification(provides objective evidence that looks for consistency, completeness, and correctness of the software)

- Validation(is a confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses)

**SOFTWARE VALIDATION**

- Business domain: Does the requirements specification match customer needs? This is not your problem.

- Software engineering domain: Does software comply with requirements specification?

- Overlap domain: Does complete software product meet the customer needs?

**VALIDATION APPROACHES**

- Two basic approaches: (Experimental (dynamic)

Analytical (static) - correctness proof with mathematical model)

**SOFTWARE LIFE CYCLE ACTIVITIES**

- Quality Planning, System Requirements Definition, Detailed Software Requirements Specification, Software Design Specification, Construction or Coding, Testing, Installation, Operation and Support, Maintenance, Retirement

**TESTING BY DEVELOPER**

- Test Planning, Structural & Functional Test Case Identification, Traceability Analysis, Unit Tests to Detailed Design and execution, Integration Tests to High Level Design and execution, System Tests to Software Requirements and execution, Acceptance Test Execution, Test Results Evaluation, Error Evaluation/Resolution, Final Test Report

**TESTING TERMINOLOGY**

- Error or defect = problem or mistake in the code

- Failure = incorrect behavior

- Test case = a specific input intended to trigger a failure, in order to identify an error

- Test set or test suite = a group of test cases

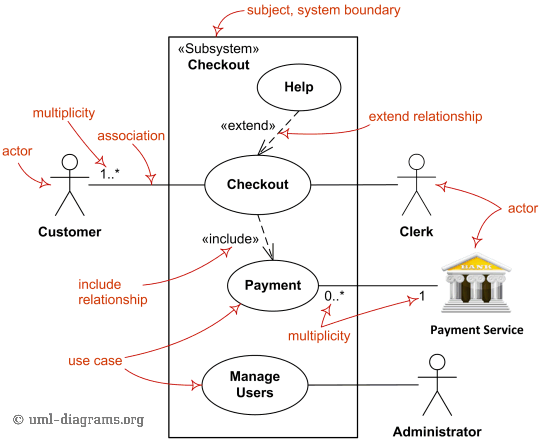
- Test script = a method for organizing test cases, as a set of sequential testing steps, with expected results

- Test plan = a document that specifies how a system will be tested.

- Test coverage = the degree to which a set of test cases exercises all paths or possibilities in a system.

- Test driver = code that is not part of the software system, but which is written in order to exercise and test part of the system.

**Use-Case**



**Use-case narratives(login)**

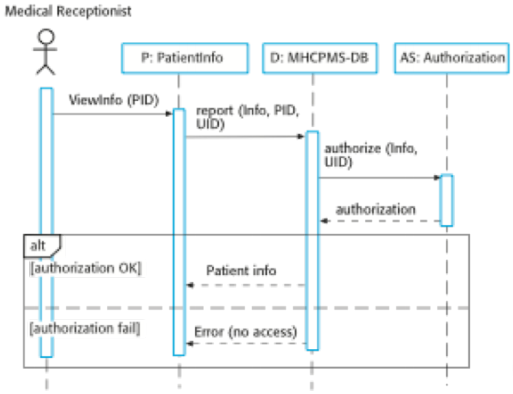
Actors : Mr.A

Goal: login to system

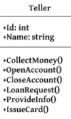
Pre-conditions: Login dialog is visible

Main success scenario(enter username,enter password, system hide password, click login, system validate username and password)

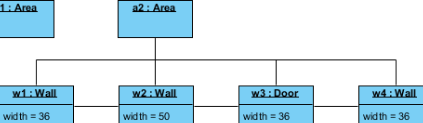
**Sequence Diagram**



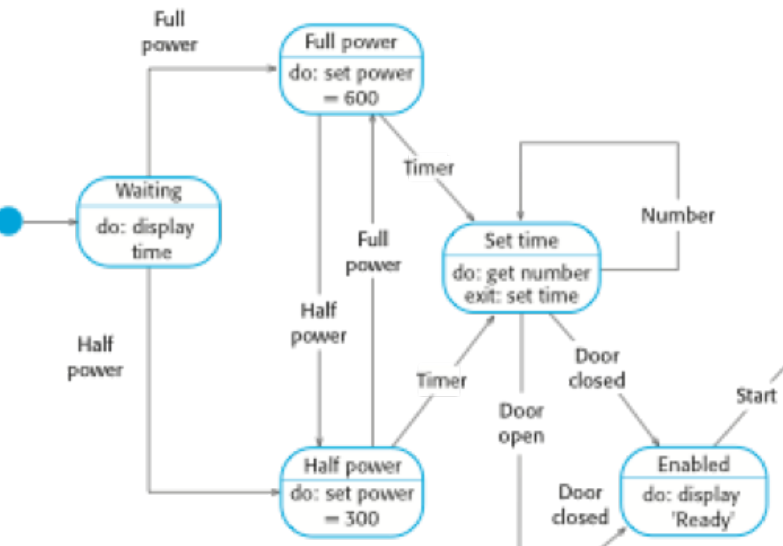
**Class Diagram**



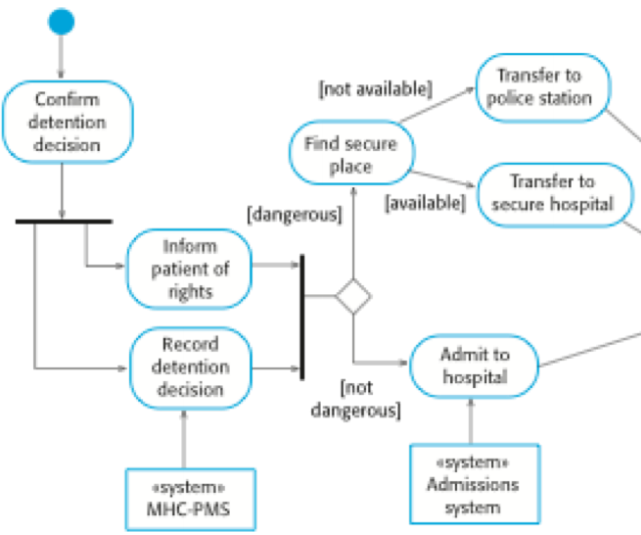
**Object Diagram**



**State Diagram**



**Activity Diagram**



**Object diagram**

