Software Maintenance & Evolution

Software Engineering Brief Review

Dr. Shihong Huang

Department of Computer Science & Engineering
Florida Atlantic University

Objectives

- Learn why software engineering is an important discipline
- See why it is so hard to do properly
- Learn what software engineering – and what it is not
- Gain an appreciation for ethical and culpable software engineering

Outline

- The challenges of software
- Software engineering
- The software process
- Professionalism
- Summary

Challenges of Software –1

Software is everywhere

- It makes up over 95% of the cost of a PC
- It's in nearly all embedded devices, including "information appliances"
- Most companies are now in the software business, whether they realize it or not
  - Aircraft
  - Cars
  - Finance
  - …

Challenges of Software –2

- Large Programs
  - Windows, SAP
  - FaceBook source code
- Medium Programs
  - Academic software, small custom projects
- Small Programs
  - Class projects, small applications

Challenges of Software –3

Software is Problematic

- It's often late
- It's very expensive to construct
- It usually doesn't meet requirements
- It's hard to understand
- It quickly becomes unmaintainable
- It's unreliable
- It's still inefficient – "bloatware"
- It's become far too complex

Challenges of Software –4

How Are We Doing? (1995)

- 16% of software projects were finished on time & within budget
- Over 50% of projects cost nearly 200% of their original estimate
- Projects usually had less than 50% of originally proposed features/functions

Challenges of Software –5

How Are We Doing? (2002)

- 75% of all software projects late or canceled
- 78% of IT organizations involved in disputes leading to litigation
- For the organizations that entered into litigation:
  - In 67% of the disputes, the functionality of the information system as delivered did not meet up to the claims of the developers
  - In 56% of the disputes, the promised delivery date slipped several times
  - In 45% of the disputes, the defects were so severe that the information system was unusable
- 50% of software is so defective / unusable
- Cost to U.S. economy: $5.85B

[Standish Group, Cutter Edge, NIST]

Challenges of Software –6

Software Failure is Expensive

- American Airlines: $50M – schedule bug
- American Express: $167K per minute
- Charles Schwab: $1M per minute
- E*Trade, eBay, amazon.com, …

- Y2K: estimated at $600B (final bill= ?)

Challenges of Software –7

Ariane 5 Launcher Failure

- A European rocket designed to launch commercial payloads into Earth orbit
- Successor to the successful Ariane 4 launchers
- 1996 June 4th, total failure Ariane 5's maiden flight
- Failure occurred at 37 seconds, altitude less than 4 kilometers
- Cost of failure: $500M

Software Engineering –1

- Term “software engineering” first appeared at a NATO conference in Germany in 1968
- The IEEE computer society defines software engineering as “*the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; That is, the application of engineering to software*.” [IEEE standards guide, 1990]

Software Engineering –2

- Based on computer science theory & practice
- Computer science is concerned with theory and fundamentals; software engineering is concerned with the practicalities of developing and delivering useful software.
- Computer science theories are still insufficient to act as a complete underpinning for software engineering (unlike e.g. physics and electrical engineering).
- Theory:
  - Knowing which data structures to use
  - Knowing how to develop new algorithms
  - Understanding computability
- Practice:
  - Skilled at programming in various languages
  - Knowledgeable about computer architectures
  - Adept at engineering complex systems

Software Engineering –3

- Currently, software engineering is neither a science nor an engineering discipline; it's more like a guild
- It is not (currently) a profession in the same sense as, say, electrical engineering
- It is not a profession like law or medicine

- The "in-the-large" companion to software construction
- It's not just coding…

Software Engineering –4

Products

- Different products have different goals:
  - Shrink-wrapped PC office automation software
  - Enterprise applications (CRM)
  - Space shuttle control software
- Software engineering is not just about producing products, but producing products in a cost-effective and predictable manner
- The goal is to produce high-quality software with a finite amount of resources on schedule

Software Engineering –5

CASE (Computer-Aided Software Engineering)

- Software systems that are intended to provide automated support for software process activities.
- CASE systems are often used for method support.
- Upper-CASE
  - Tools to support the early process activities of requirements and design;
- Lower-CASE
  - Tools to support later activities such as programming, debugging and testing.

Software Engineering –6

Quality Attributes

- Maintainability: Software must evolve to meet changing needs
- Dependability: Software must be trustworthy
- Efficiency: Software should not make wasteful use of system resources
- Acceptability: Software must accepted by the users for which it was designed. This means it must be understandable, usable and compatible with other systems
- Usability:
- Security

- Assessing quality attributes is difficult

The Meaning of Process

- Process – a set of ordered tasks
- A series of steps involving activities, constraints, and resources that produce an intended output of some kind
- When the process involves the building of some product, we refer to the process as *Life Cycle*
- Software development process is called *Software Life Cycle*
- It describes the life of a software product from its conception to its implementation, delivery, use, and maintenance

Process characteristics

- Process describes all of the major activities
- Process uses resources, subject to a set of constraints (e.g. scheduling), and produce intermediate and final products
- May be composed of subprocesses that are linked in someway
- Each process activity has en try and exit criteria, so that we know when the activity begins and ends
- The activities are organized in a sequence
- Every process has a set of guiding principles that explain the goals of each activity
- Constraints or controls may apply to an activity, resources, or product

The Software Process –1

- "Software process is a set of activities and associated results which lead to the production of a software product" –[Sommerville 00]
- Fundamental activities which are common to all software processes
  - *Software specification:* define functionality of the software and constraints on its operation
  - *Design and implementation*: meet the specification
  - *Software Validation*: it does what customer wants
  - *Software Evolution*: evolve to meet changing customer needs
- A process model is a software product

The Software Process –2

Non-Technical Activities

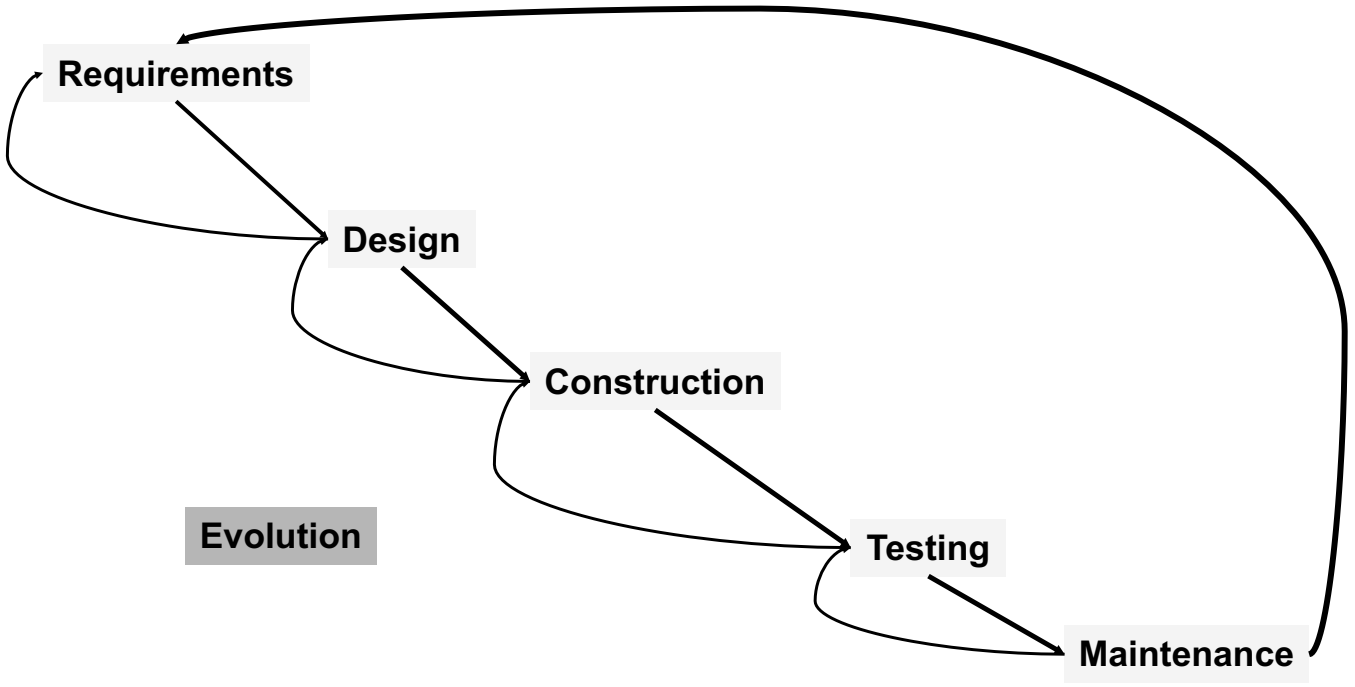
- Management of people, project, process
- Technical writing
- Cost models, budgeting, economics
- Marketing
- …

Software Process –3

- Software Process
- Waterfall Model
- Rapid-Prototyping Model
- Evolutionary Model
- Synchronize-and-stabilize Model
- Spiral Development Model
- The product line model
- Rational Unified Process
- Incremental and Iterative Model
- XP and Agile Processes

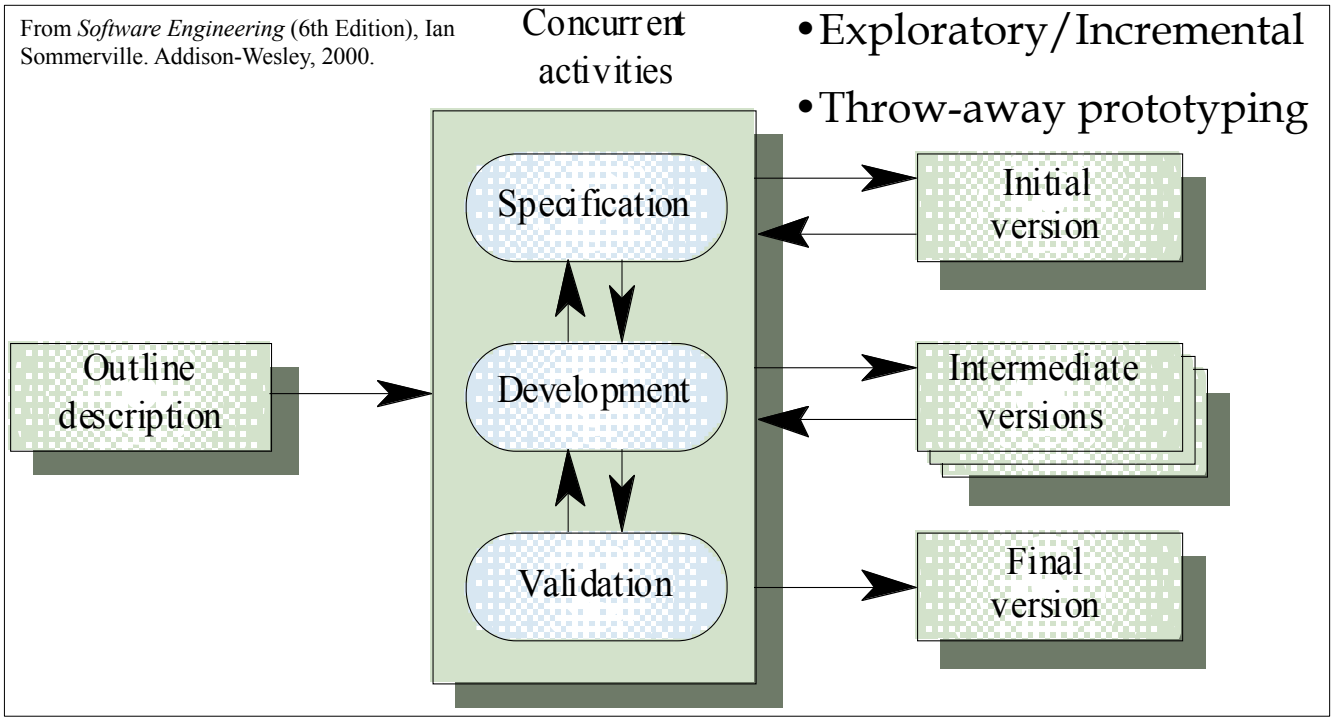
The Software Process –4

The Waterfall Model



Software Process -5

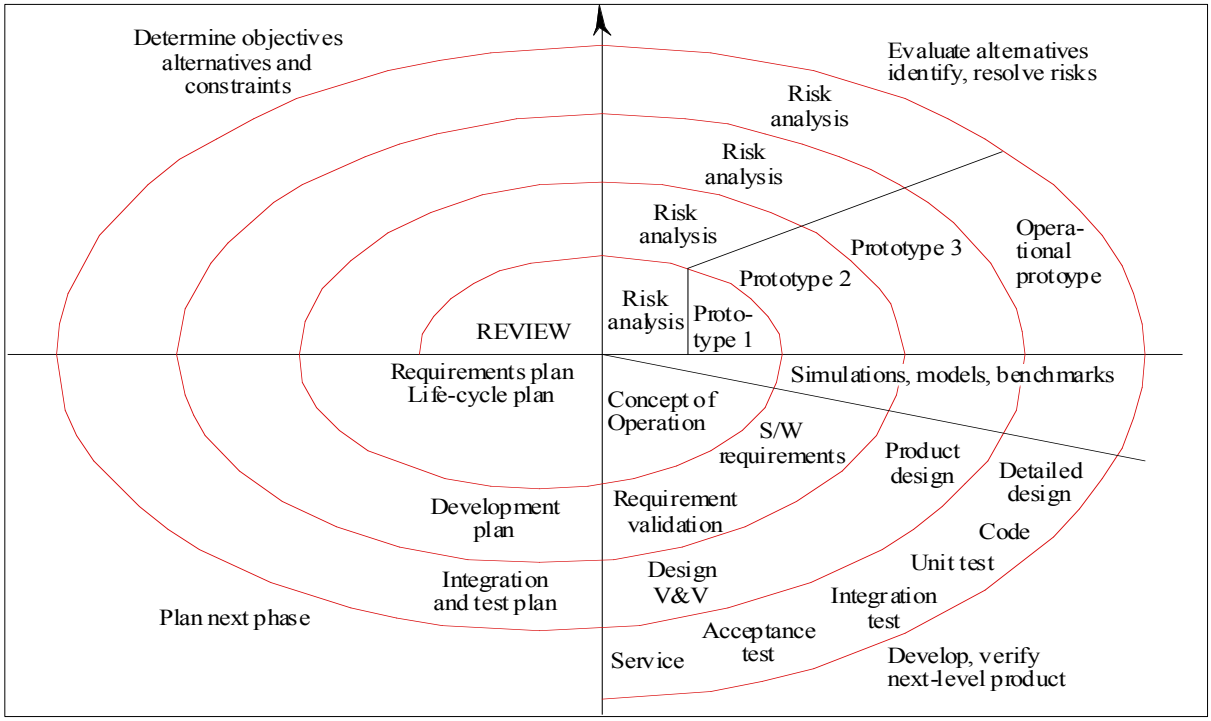
- The Evolutionary Model



Spiral Model –1

- The spiral model was developed by Barry Boehm of USC around 1988
- The process is represented a a spiral (rather than a sequence of activities w/ backtracking)
- Each loop represents a phase of software process (feasibility, requirements, design, etc)
- Each loop has 4 sectors:
  - Objective setting
  - Risk assessment and reduction
  - Development & validation
  - Planning
- Explicitly recognizes risk
- Subsumes all generic process models

Spiral Model –2



Requirements Engineering

- Software Requirements
- Requirements Engineering
- Example of two types of requirements documents (definition vs. specification)
- Specification
  - Text:
    - Informal natural language
    - Structural natural languages
    - Design description languages
  - Graphical notations
  - Formal methods

Requirements Engineering

- Feasibility studies
- Composed of three main activities:
  - Elicitation
  - Specification
  - Validation & Verification (V&V)
    - Validation: Are we building the right product?
    - Verification: Are we building the product right?
- Sometimes “feasibility analysis” stage
- Output are requirements documents

Dilbert on Requirements

© Scott Adams, Inc./Dist. by UFS, Inc.

"The requirements specification takes the narrative customer requirements [definitions] and constructs specification models. An output of the specification phase is an elaborated requirements document."

– *Requirements Analysis and System Design*
Leszek Maciaszek (Addison-Wesley, 2001)

Requirements

- *Requirements* for a system are the descriptions of the services provided by the system and its operational constraints [Sommerville07]
- Three kinds of requirements:
  - those that absolutely must be met
  - those that are highly desirable but not necessary
  - those that are possible but could be eliminated

Functional vs. Non-Functional Requirements

- Functional: describes an interaction between the system and its environment
- Examples:
  - System shall communicate with external system X
  - What conditions must be met for a message to be sent
- Non-functional: describes a restriction or constraint that limits our choices for constructing a solution
- Examples:
  - Paychecks distributed no more than 4 hours after initial data are read.
  - System limits access to senior managers

Requirements Specifications

- *Requirements definition*: complete listing of what the customer expects the system to do
- *Requirements specification*: restates the definition in technical terms so that the designer can start on the design
- Requirements specifications add detail to the requirements definitions; definition and specification should be consistent
- Taken together, requirements definitions and specifications form a bond between the user and the engineer; a contract

Example: Requirements Definition

- An example satellite tracking system: [Pfleeger Section 4.8]

"4.1.3.1 INITIATE TRACK ON IMAGE. Logical processing shall be done to INITIATE TRACK ON IMPAGE. This shall have as input HANDOVER DATA. This shall have as output HOIQ, STATE DATE, and IMAGE ID. This logical processing shall, when appropriate, identify the type of entity instance as being IMAGE ON TRACK. NOTE: A request for pulses is made by entering a formal record into the HOIQ which feeds the pulse-send procedures"

Example: Requirements Specification (Using RSL)

- Use the same example in the previous slide.

```
ALPHA: INITITATE_Track_ON_IMAGE.
  INPUTS: HANDOVER_DATA
  OUTPUTS: HOIQ, STATE_DATA, IMAGE_ID.
  CREATES: IMAGE.
        SETS: IMAGE_ON_TRACK.
  DESCRIPTION: "(4.1.3.1) A REQUEST FOR PULSE IS MADE BY ENTERING A
    FORMAL RECORD REQUEST INTO THE HOIQ WHICH FEEDS THE PULSE
    SENDING PROCEDURES"
```

- It's clear that customer can understand the definition document, but may have more difficulty with RSL specification
  - IEEE and DoD have standards for content and format of the requirements documents

Specifications

- Text Specifications
  - Informal natural language
  - Structured natural language
  - Design description languages
  - Requirements specification languages
- Graphical notations (e.g. UML)
- Formal specification languages
  - Finite State Machine
  - Algebraic approach
  - Model-based: Z, VDM, Petri net

Informal Natural Languages –1

- Informal specifications are written in a natural language
  - Examples: English, Mandarin, Hindi

- Example

  "If the sales for the current month are below the target sales, then a report is to be printed, unless the difference between target sales and actual sales is less than half of the difference between target sales and actual sales in the previous month, or if the difference between target sales and actual sales for the current month is under 5%"

Dilbert on Specifications

- **Informal natural language: needs to be precise to avoid ambiguity**

- **Natural language tends to have contradictions, ambiguities, and omissions.**

Structured Natural Languages

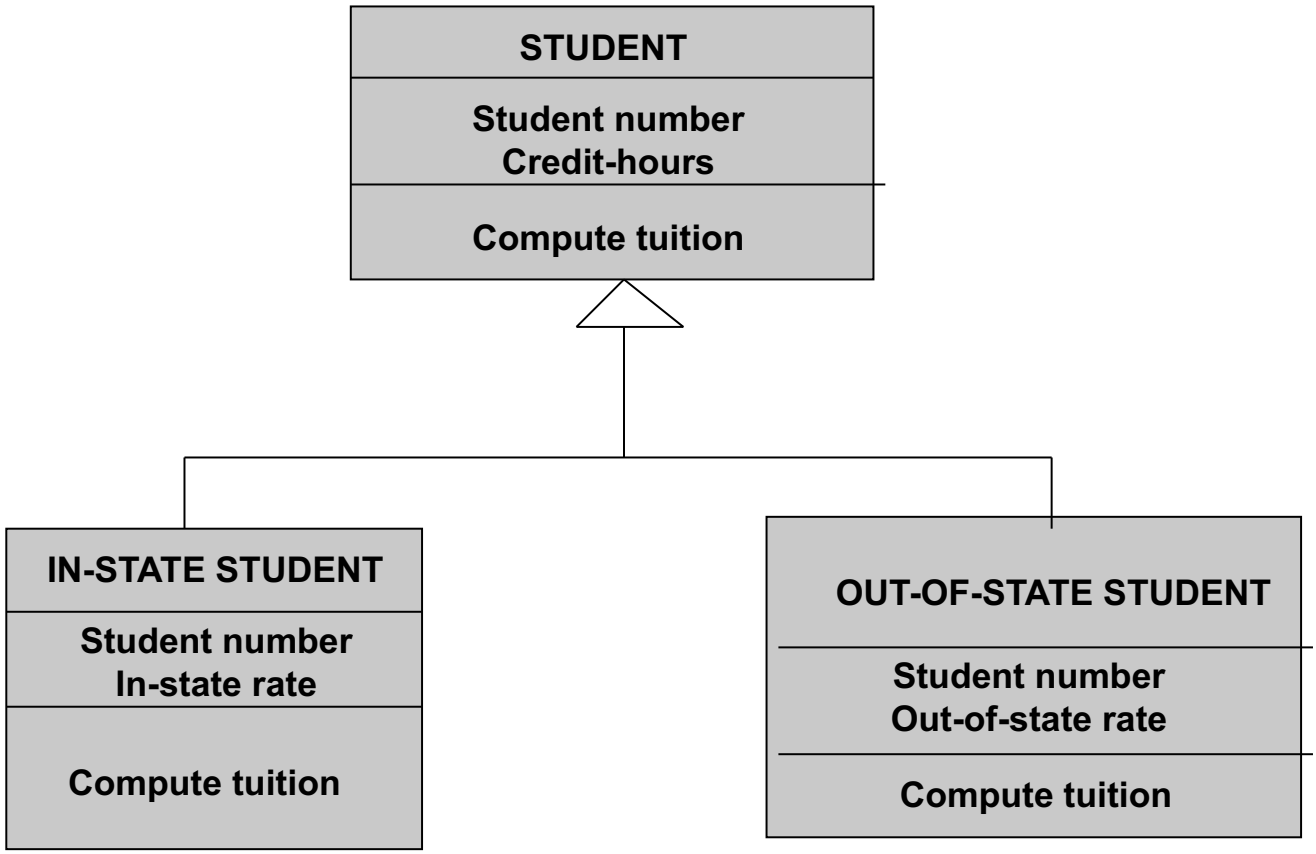
XML (eXtensible Markup Language)

- Subset of SGML (Standard Generalize Markup Language)
- ISO standard for defining and using documents format
- Syntactically, XML superficially resembles HTML
- Three important features make it more useful for documentation:
  - Extensibility
  - Structure
  - validation

Static Description of Requirements

- A description lists the system entities or objects, their attributes, and their relationships with other
- *Static:* views requirements as defining the relationships of entities or objects to each other, doesn't describe how relationships change with time
- Several ways to describe a system statically
  - Recurrence relations (e.g. Fibonacci number)
  - Expression as a language (e.g.: Backus-Naur form)
  - Data abstraction (class, instance)

An abstract class and Object relationships



Dynamic Descriptions –1

- Describes how the system reacts over a period of time to the things that change system behavior
- System is described in terms of states and stimuli

Decision tables

- Describe a system as a set of possible conditions satisfied by the system at a given time, rules for reaching to stimuli when certain sets of those conditions are met, and actions to be taken as a result
- Table could be very large (for *n* conditions, 2^n possible combinations of condition)

Example of Decision Table

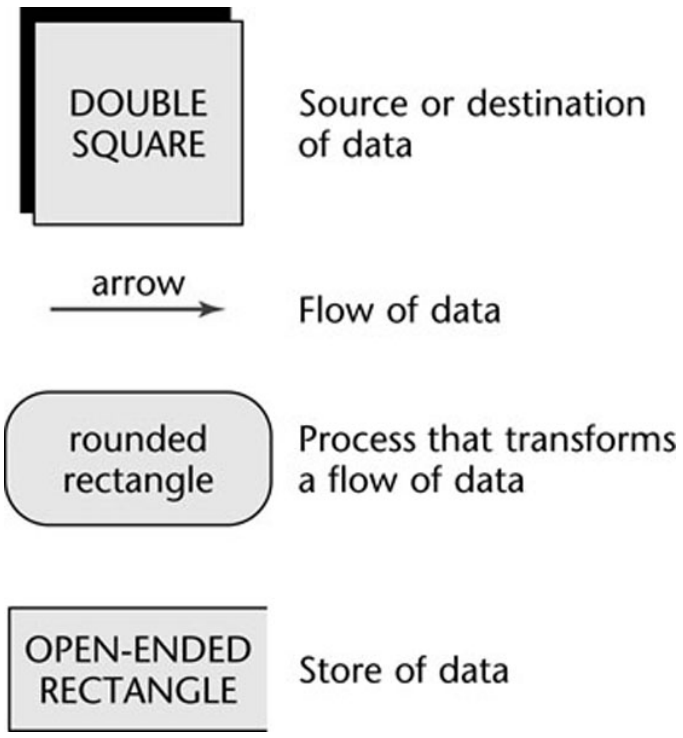
Example: Freshmen Admission Criteria

.	*Rule 1*	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*
High standardized exam scores	T	F	F	F	F
High grades	-	T	F	F	F
Outside activities	-	-	T	F	F
Good recommendations	-	-	-	T	F
Send rejection letter			X	X	X
Send admission forms	X	X			

Dynamic Descriptions –2

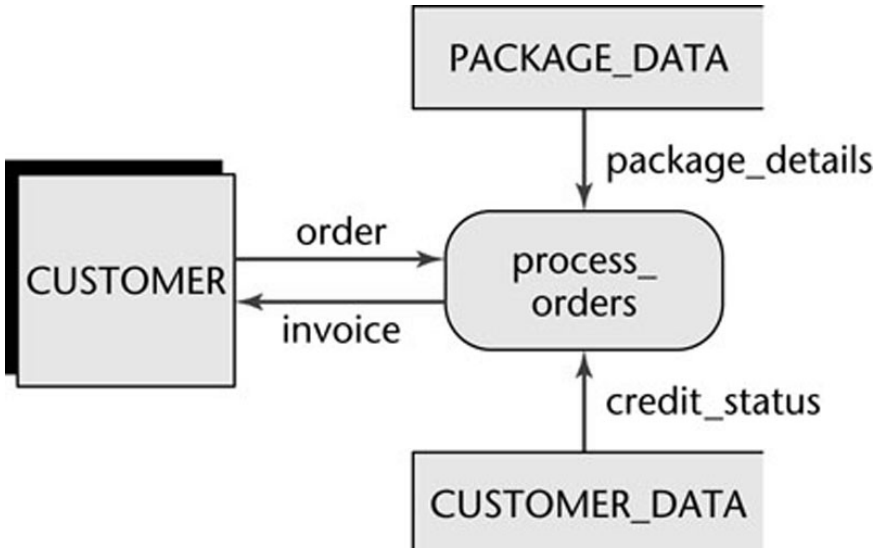
Data Flow Diagrams

- Graphics specification
- The data flow diagram (DFD) shows the logical data flow
  - "What happens, not how it happens"
- Example: Symbols of Gane and Sarsen's structured systems analysis



Data Flow Diagrams Example

- First refinement
  - Infinite number of possible interpretations



Cf: Schach Session 11.3

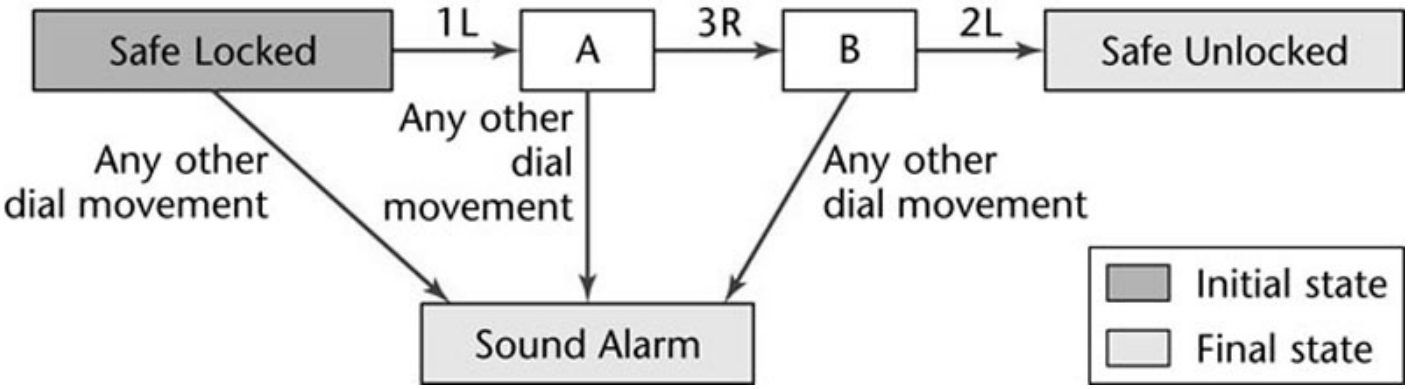
Dynamic Descriptions –3

Finite State Machines (FSM) –1

- FSM consists of five parts:
  - A set of states (J)
  - A set of inputs (K)
  - The transition function: specifies the next state given the current state and the current input (T)
  - The initial state (S)
  - The set of final states (F)

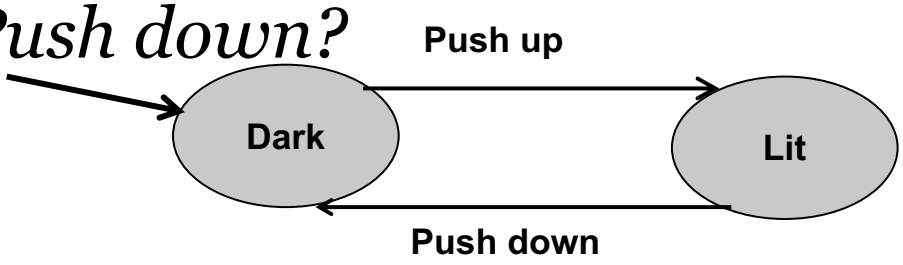
FSM Example: Combination Lock

- A safe has a combination lock that can be in one of three positions, labeled 1, 2, and 3. The dial can be turned left or right (L or R). Thus there are six possible dial movements, namely 1L, 1R, 2L, 2R, 3L, and 3R. The combination to the safe is 1L, 3R, 2L; any other dial movement will cause the alarm to go off
- The set of states J is {Safe Locked, A, B, Safe Unlocked, Sound Alarm}
- The set of inputs K is {1L, 1R, 2L, 2R, 3L, 3R}
- The transition function T is on the next slide
- The initial state J is Safe Locked
- The set of final states F is {Safe Unlocked, Sound Alarm}



Dynamic Descriptions –3

- When read a software description, we should question everything in the description, specially the assumptions and hidden meanings
- E.g. what will happen when in *Dark* state *Push down?*



Dynamic Description –4

- The techniques described thus far are most useful for system whose states and events occur in sequence.

  f (state A, Event) → State S
- When several events occur at once, a system must perform parallel processing. Synchronization is a major problem
- Several events must occur before the system can leave state A to state S

  f (State A, Event 1, Event 2, … Event N) → State S
- In general case, several events may be required to begin a state transition. Once the transition is initiated, the system moves into several states in parallel

  f (State A, Event 1, Event 2, …, Event N) → State 1, State 2, .., State M

Requirements Engineering

- A *requirement* is "a condition or capability needed by a user to solve a problem or achieve an objective" [IEEE 1990]
- *Requirements* for a system are the descriptions of the services provided by the system and its operational constraints [Sommerville04]
- *Requirements engineering*: the process of finding out, analyzing, documenting and checking these services and constraints
- *Requirements Definition*
- *Requirements Specification*

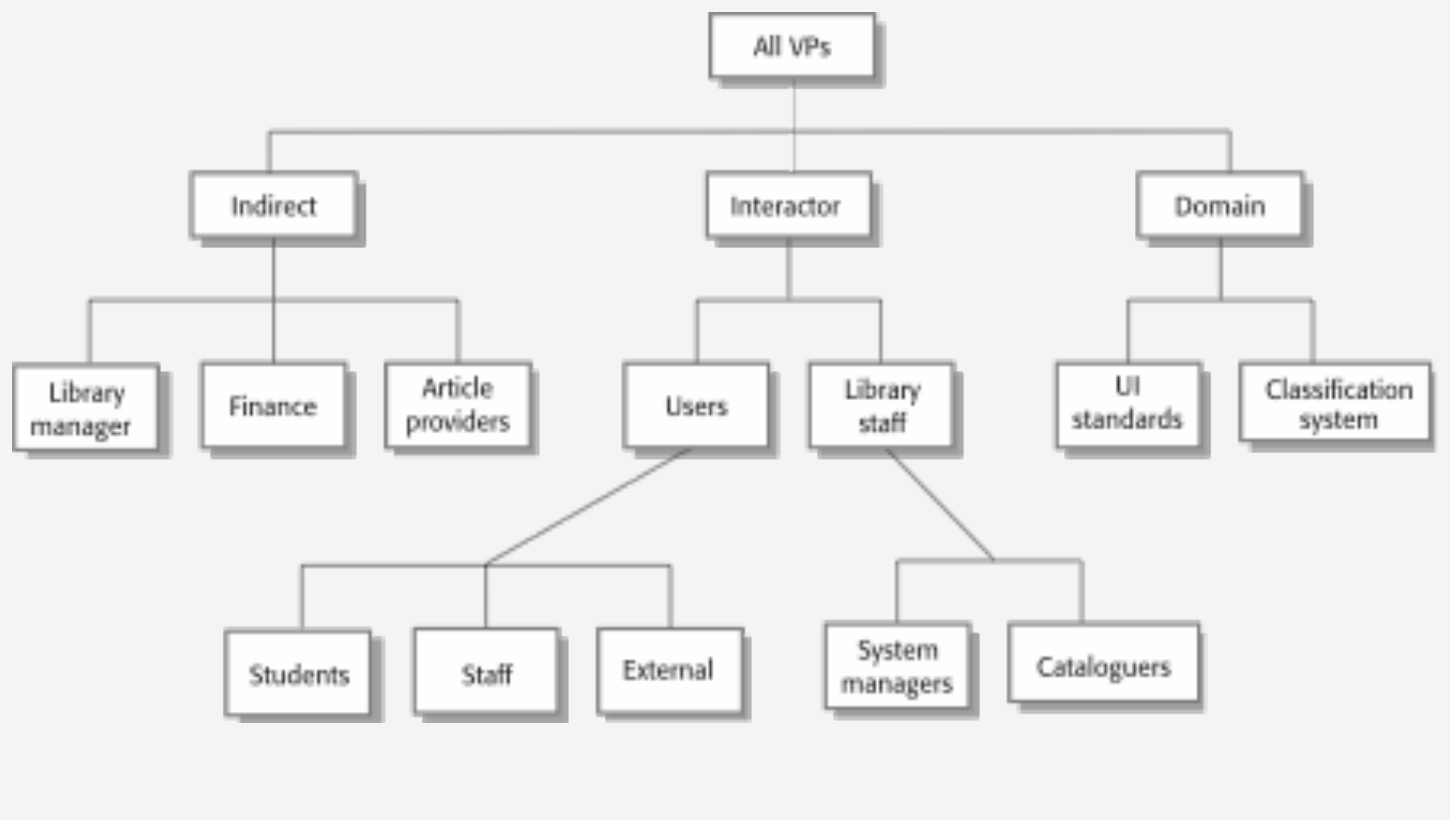
Feasibility Studies

- A feasibility study decides whether or not the proposed system is worthwhile / possible
- A short focused study that checks if:
  - The system contributes to organisational goals and objectives
  - The system can be engineered using current technology and within budget
  - The system can be integrated with other systems that are used

Elicitation

- Also called “analysis” or “discovery”
- Software engineers working with customers to find out about the application domain, the services that the system should provide, and the system’s operational constraints
- May involve end-users, managers, engineers involved in maintenance, domain experts, trade unions, etc. These are called *stakeholders* (e.g.: ATM)

LIBSYS viewpoint hierarchy



Elicitation Techniques – Interviewing

- In formal or informal interviewing, the RE team puts questions to stakeholders about the system that they use and the system to be developed.
- There are two types of interview
  - Closed interviews where a pre-defined set of questions are answered.
  - Open interviews where there is no pre-defined agenda and a range of issues are explored with stakeholders.

Elicitation Techniques – Scenarios

- More focused than generic plans, capturing likely system usage situations
- Helps prioritize requirements by asking a lot of "what if" questions
- They are helpful in requirements elicitation as people can relate to these more readily than abstract statement of system requirements
- Scenarios are particularly useful for adding detail to an outline requirements description

Elicitation Techniques – Scenarios

- Use-cases are a scenario-based technique in the UML which identify and describe the actors in an interaction
- A set of use cases should describe all possible interactions with the system.
- Sequence diagrams may add detail to use-cases by showing the sequence of event processing in the system
- Scenarios form the heart of the ATAM (Architecture Tradeoff Analysis Method, by SEI)

Requirements V&V –1

- Concerned with demonstrating that the requirements define the system that the customer really wants
- Requirements error costs are high, so validation is very important: > 100x as expensive as fixing code

- Validation: correct requirements stated
- Verification: requirements stated correctly

Requirements V&V –2

Things to Check -1

- Validity: Does the system provide the functions which best support the customer's needs?
- Consistency: Are there conflicting requirements?
- Completeness: Are all functions required by the customer included?
- Realism: Can the requirements be implemented given available budget and technology
- Verifiability: Can the requirements be checked?

Requirements V&V –3

Things to Check -2

- Comprehensibility: is the requirement properly understood?
- Traceability: is the origin of the requirement clearly stated?
- Adaptability: can the requirement be changed without a large impact on other requirements?

V&V Techniques

- *Requirements reviews*: systematic manual analysis of the requirements
- *Prototyping*: using an executable model of the system to check requirements
- *Test-case generation*: developing tests for requirements to check testability
- *Automated consistency analysis*: checking the consistency of a structured / formal requirements description

Traceability

- Traceability is concerned with the relationships between requirements, their sources and the system design
- Source traceability
  - Links from requirements to stakeholders who proposed these requirements;
- Requirements traceability
  - Links between dependent requirements;
- Design traceability
  - Links from the requirements to the design

A Traceability Matrix

Req. id	1.1	1.2	1.3	2.1	2.2	2.3	3.1	3.2
1.1		D	R					
1.2			D			D		D
1.3	R			R				
2.1			R		D			D
2.2								D
2.3		R		D				
3.1								R
3.2							R	

CASE Tool Support

- Requirements storage
  - Requirements should be managed in a secure, managed data store.
- Change management
  - The process of change management is a workflow process whose stages can be defined and information flow between these stages partially automated.
- Traceability management
  - Automated retrieval of the links between requirements.

Design 1

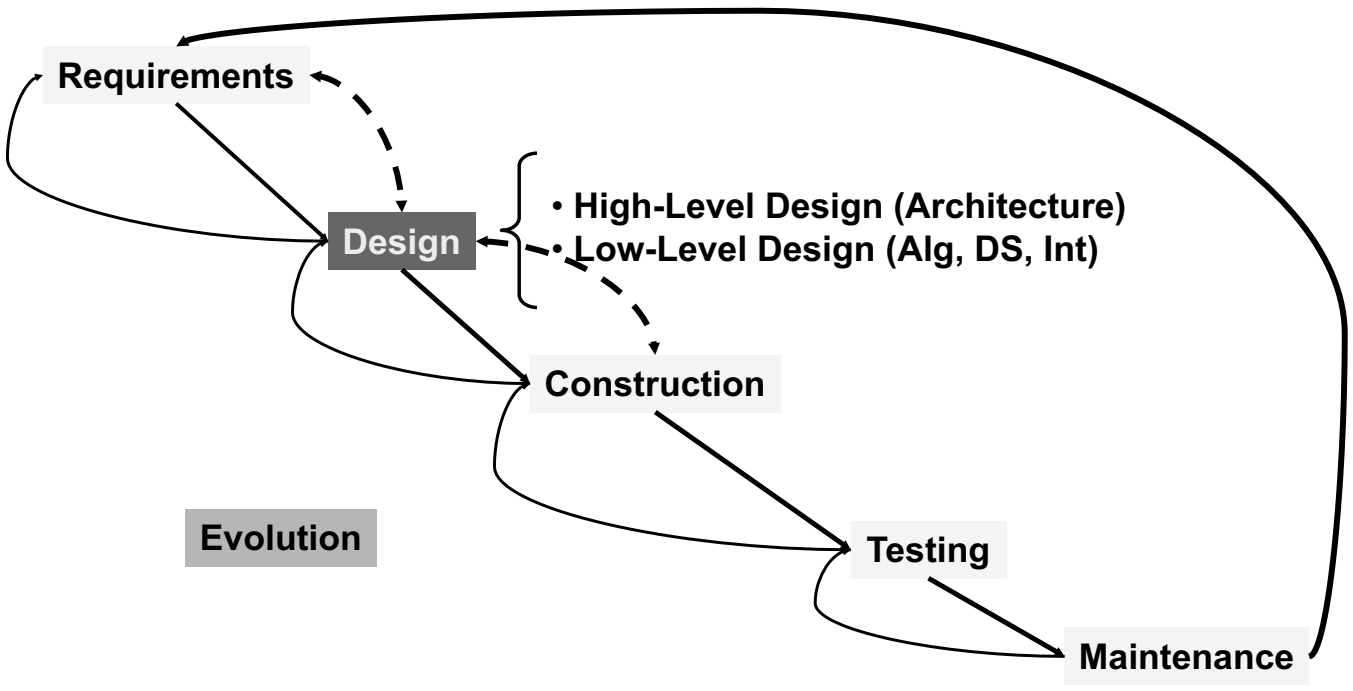
Dr. Shihong Huang

Department of Computer Science & Engineering
Florida Atlantic University

What is Software Design -1

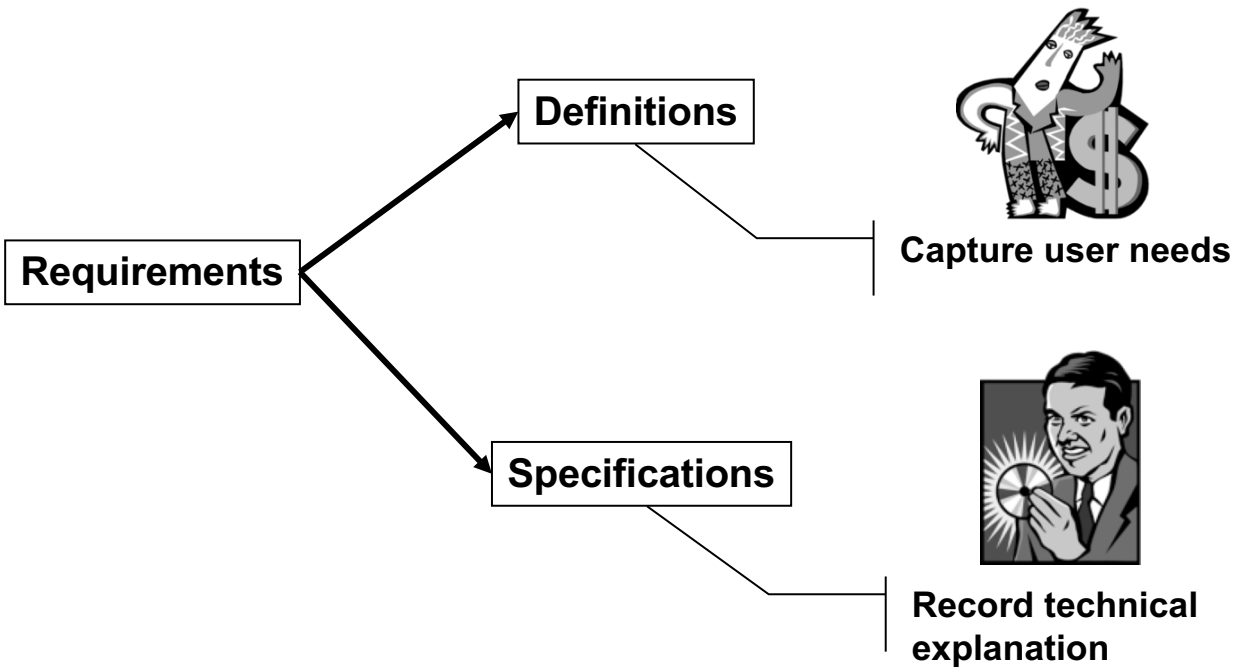
- Design is a bridge between requirements (specifications) and code construction
- Design refines abstract specifications into implementation guidelines
- Design customers are software engineers (programmers and sometimes testers)
- Architectural design process is concerned with establishing a basic structural framework that identifies the major components of a system and the communications be/w these components
- The output of this design process is a description of the software architecture

What is Software Design -2



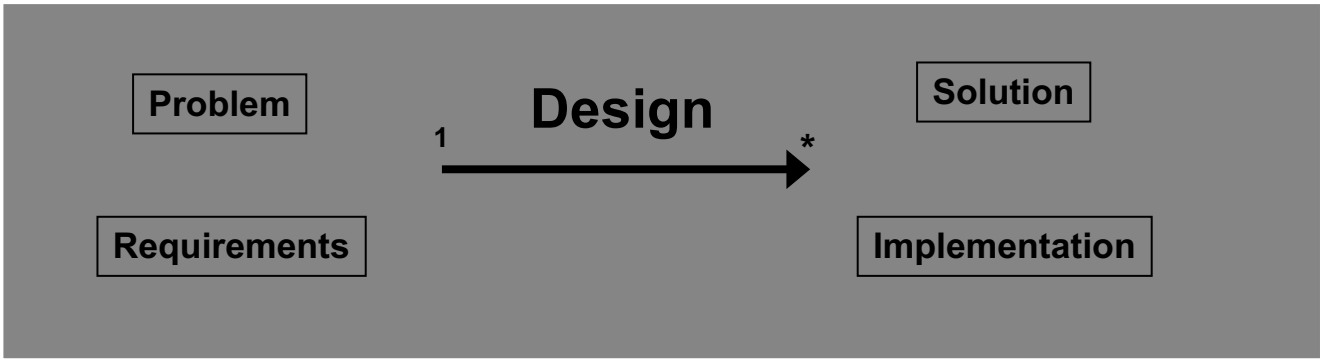
What is Software Design -3

- Requirements: the ***what*** of the system



What is Software Design -4

- Design identifies the ***how***
- Deriving a solution that satisfies both sets of requirements



What is Software Design -5

- The design process involves tradeoff analysis based on the requirements
- Design has three layers: [Shaw and Garlan 1996]
  - High-Level Design (Architecture Design)
    - Associate the system capabilities identified in the requirements speciation w/ the system components that will implement them
    - Components are usually modules, the architecture describes the interconnections among them
  - Low-Level Design (Code Design):
    - Algorithms
    - Data Structures
    - Component Interfaces
  - Executable Design:
    - Address the code design at a lower level of detail
    - Memory allocation, data formats, bit patterns, and so on
- We focus on architectural design

Architectural Design

- A creative process where you try to establish a system organization will satisfy the functional and non-functional system requirements
- Represents the link between specification and design processes.
- Often carried out in parallel with some specification activities.
- It involves identifying major system components and their communications.
- Input: Requirement (Specification)
- Output: Modular decomposition

Advantages of explicit architecture

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible
- Large-scale reuse
  - The architecture may be reusable across a range of systems

Architecture and system characteristics

- Performance
  - Localize critical operations and minimize communications Use large rather than fine-grain components
- Security
  - Use a layered architecture with critical assets in the inner layers
- Safety
  - Localize safety-critical features in a small number of sub-systems
- Availability
  - Include redundant components and mechanisms for fault tolerance
- Maintainability
  - Use fine-grain, replaceable components

Architectural conflicts

- Using large-grain components improves performance but reduces maintainability.
- Introducing redundant data improves availability but makes security more difficult.
- Localizing safety-related features usually means more communication so degraded performance

➢ ATAM

The Essence of Design -1

- The essence of design is problem solving
- It is fundamentally a creative process
- It is difficult to learn by rote (cf. guild)

The Essence of Design -2

Iterative Enhancement

- Identify the *core* of the problem
- Design a solution for it
- Enhance the solution until it meets the original requirements
- Each iteration implemented and tested
- Problems extending the core solution

- Example: a compiler - no scope, no variables, no types, no functions, ...

The Essence of Design -3

Step-Wise Refinement

- *Algorithms + Data Structures = Programs* Nicklaus Wirth (Prentice-Hall, 1985)
- A **systematic** approach to design based on divide-and-conquer (cf. course theme)
- Decomposition into 7 +/- 2 "components" to deal with complexity (& schedule tasks)
- Problem when there are more components than information per component

The Essence of Design -4

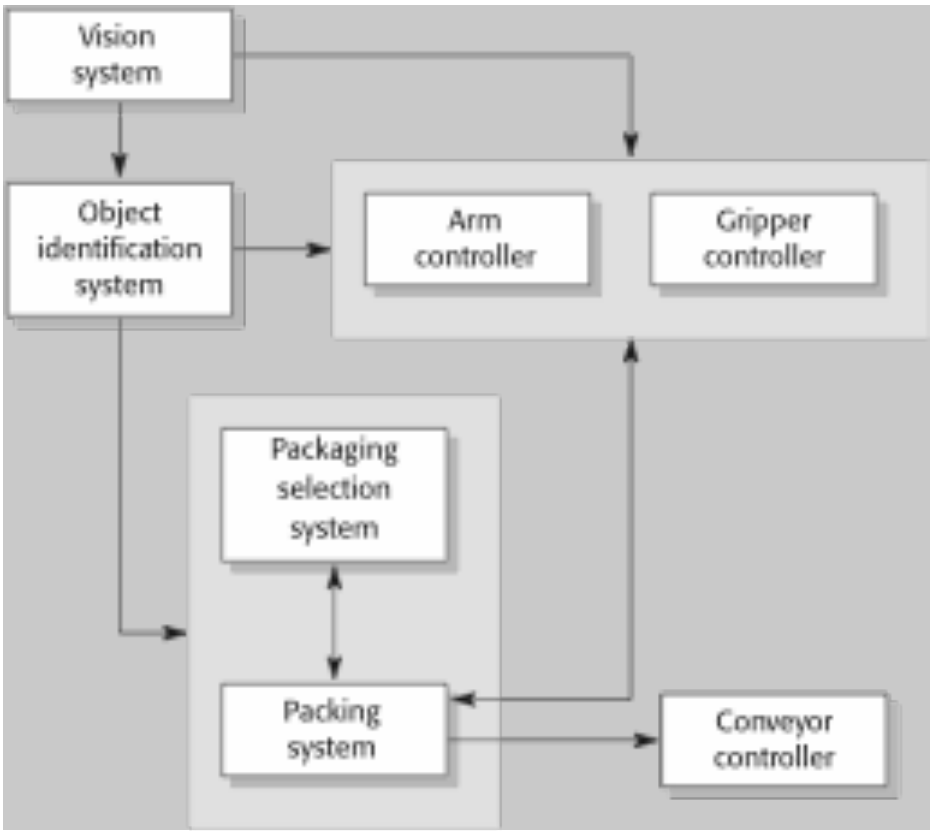
Information Hiding

- Idea espoused by David Parnas
- Plan for change (cf. maintenance)
- Only expose what is needed; design a minimal set of component interfaces
- Direct impact on coupling & cohesion
- One of the central tenets of OO

System structuring

- Concerned with decomposing the system into interacting sub-systems.
- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.
- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

Example: Packing robot control system



Box and line diagrams

- Very abstract - they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system being developed.
- However, a number of common decisions span all design processes

Architectural design decisions

- Is there a generic application architecture that can be used?
- How will the system be distributed?
- What architectural styles are appropriate?
- What approach will be used to structure the system?
- How will the system be decomposed into modules?
- What control strategy should be used?
- How will the architectural design be evaluated?
- How should the architecture be documented?

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
- Application architectures are covered in Chapter 13 and product lines in Chapter 18.

Architectural Styles

- The architectural model of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- However, most large systems are heterogeneous and do not follow a single architectural style

Architectural Models

- Used to document an architectural design
- Static structural model that shows the major system components
- Dynamic process model that shows the process structure of the system
- Interface model that defines sub-system interfaces
- Relationships model such as a data-flow model that shows sub-system relationships
- Distribution model that shows how sub-systems are distributed across computers

System Organization Styles

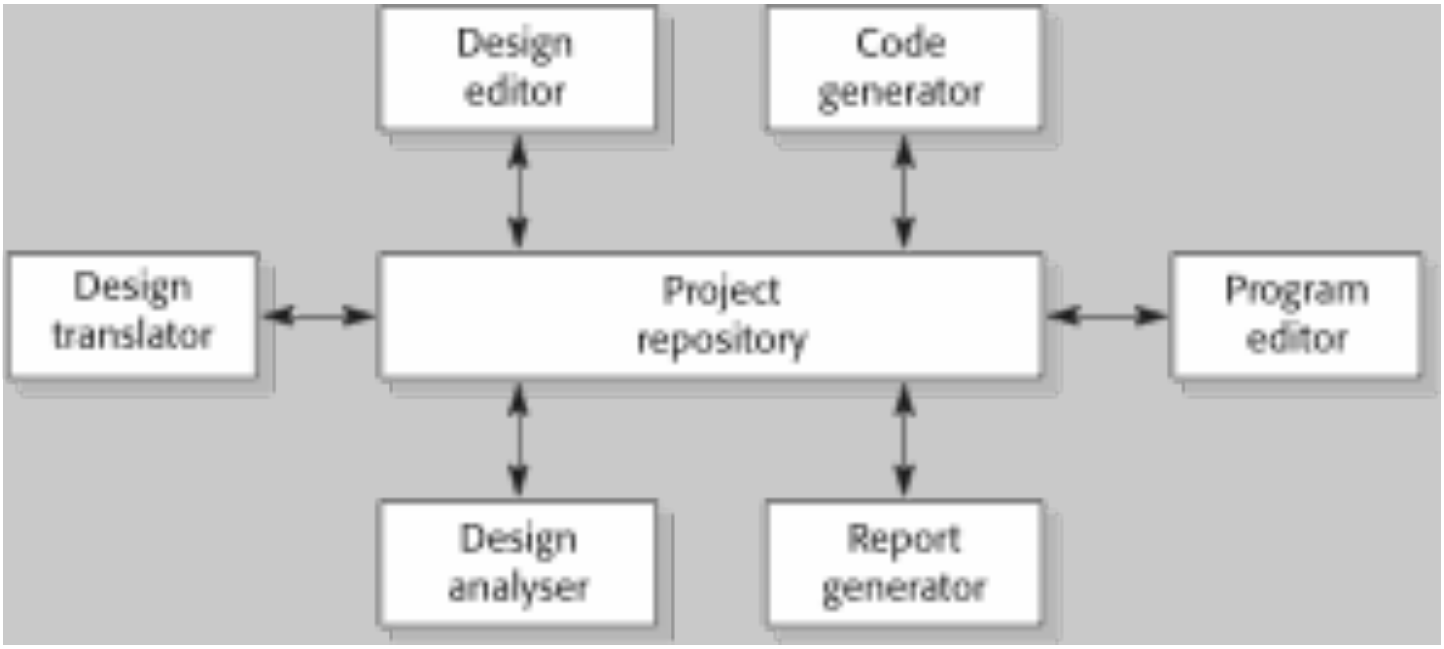
- Reflects the basic strategy that is used to structure a system
- Three organizational styles are widely used:
  - Repository model: a shared data repository style
  - Client-server model: a shared services and servers style
  - Layered Model: an abstract machine or layered style

The Repository Model –1

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems
- When large amounts of data are to be shared, the repository model of sharing is most commonly used

The Repository Model –2

An example: The architecture of an integrated CASE Toolset



From *Software Engineering* (8th Edition)
Ian Sommerville. Addison-Wesley, 2007.

Repository Model –3

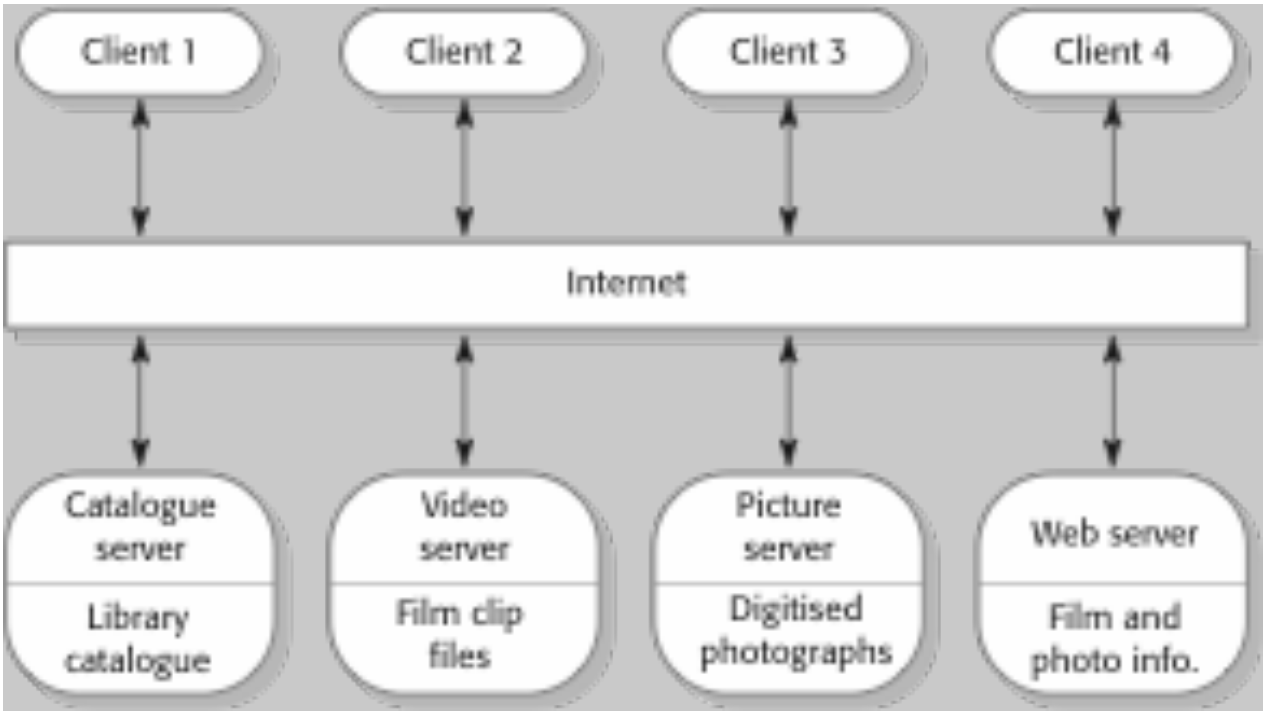
- Advantages
  - Efficient way to share large amounts of data
  - Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security
  - Sharing model is published as the repository schema.
- Disadvantages
  - Sub-systems must agree on a repository data model. Inevitably a compromise
  - Data evolution is difficult and expensive
  - No scope for specific management policies
  - Difficult to distribute efficiently

Client-Server Model –1

- Distributed system model which shows how data and processing is distributed across a range of components
- Set of stand-alone servers which provide specific services such as printing, data management, etc
- Set of clients which call on these services.
- Network which allows clients to access servers

Client-Server Model –2

An Example: A film and picture library system



From *Software Engineering* (8th Edition)
Ian Sommerville. Addison-Wesley, 2007.

Client-Server Model –3

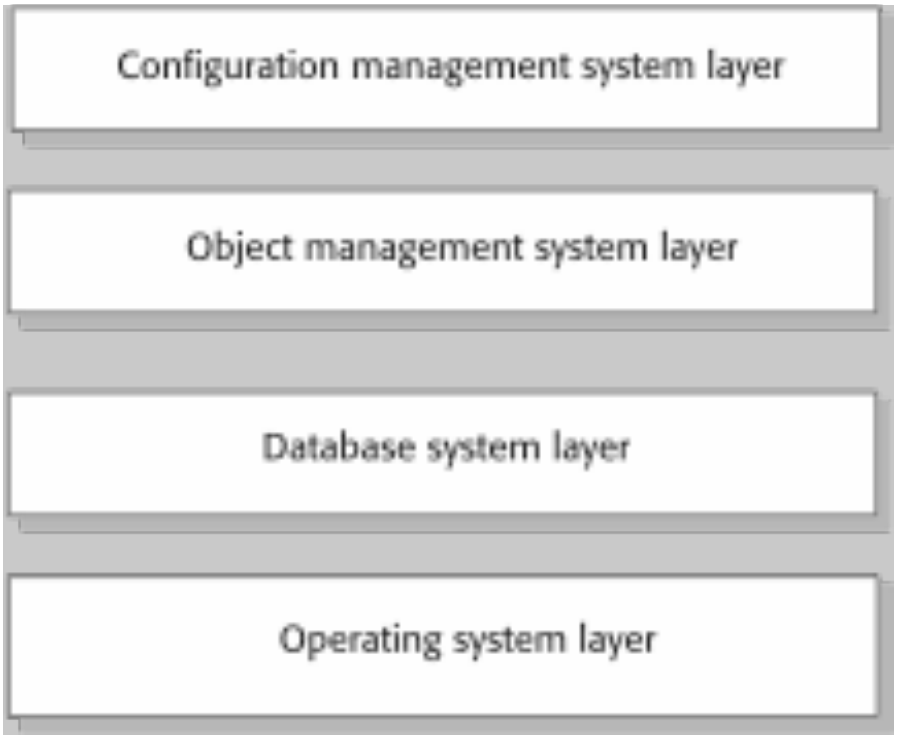
- Advantages
  - Distribution of data is straightforward
  - Makes effective use of networked systems. May require cheaper hardware
  - Easy to add new servers or upgrade existing servers
- Disadvantages
  - No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
  - Redundant management in each server
  - No central register of names and services - it may be hard to find out what servers and services are available

Abstract Machine (Layered) Model –1

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected
- However, often artificial to structure systems in this way

Abstract Machine (Layered) Model –2

An Example: Version Management System



From *Software Engineering* (8th Edition)
Ian Sommerville. Addison-Wesley, 2007.

Abstract Machine (Layered) Model –3

Advantages

- Support incremental development of systems
  - As a layer is developed, some of the services provided by that layer may be made available to users
- Architecture is changeable and portable
  - So long as its interface is unchanged, a layer can be replaced by another, equivalent layer
  - If new interfaces change or new facilities are added to a layer, only the adjacent layer is affected
- As layered system localize machine dependencies in inner layers, makes it easier to provide multi-platform implementation; only the inner, machine-dependent layers need to be reimplemented to take account of the facilities of a different OS or DB

Abstract Machine (Layered) Model –4

Disadvantage

- Difficult to structure
  - E.g. file management is needed at all layer
- Performance can be a problem
  - Multiple levels of command interpretation may be required
  - A service request from a top layer may have to be interpreted several times in difference layers before it's proceed
  - ☞ Applications may have to communicate directly with inner layers rather than use the services provided by the adjacent layer

Summary

- Design is problem solving
- It is a process of disciplined creativity
- Three timeless guidelines:
  - Iterative enhancement
  - Step-wise refinement
  - Information hiding
- Three architectural styles:
  - Repository model
  - Client-server model
  - Layered model

Design 2

Dr. Shihong Huang

Department of Computer Science & Engineering
Florida Atlantic University

Outline

- Design Quality Attributes
- Coupling
- Cohesion

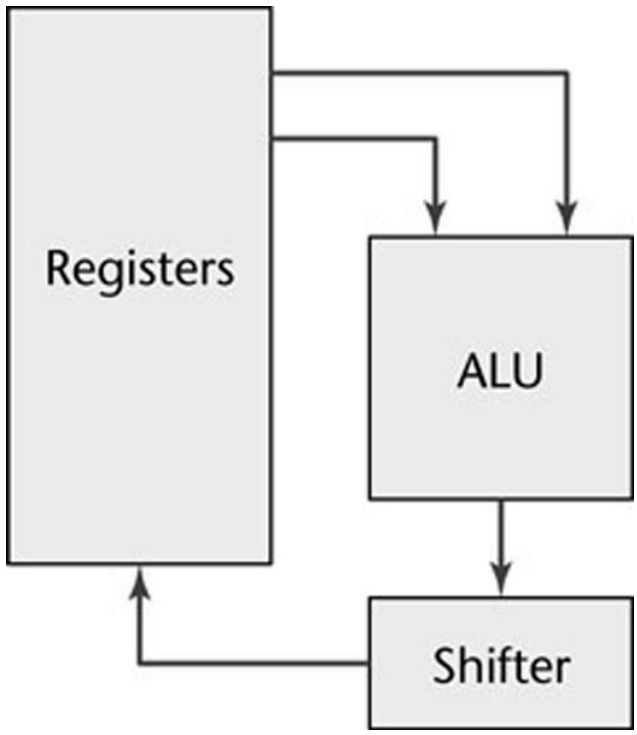
- ☐Decomposition Techniques

What is a module?

- A lexically contiguous sequence of program statements, bounded by boundary elements, with an aggregate identifier [Yourdon and Constantine 1979]
  - "Lexically contiguous"
    - Adjoining in the code
  - "Boundary elements"
    - `{ ... }`
    - **`begin ... end`**
  - "Aggregate identifier"
    - A name for the entire module
- According to this definition, e.g.: procedures and functions; object, and method w/in an object

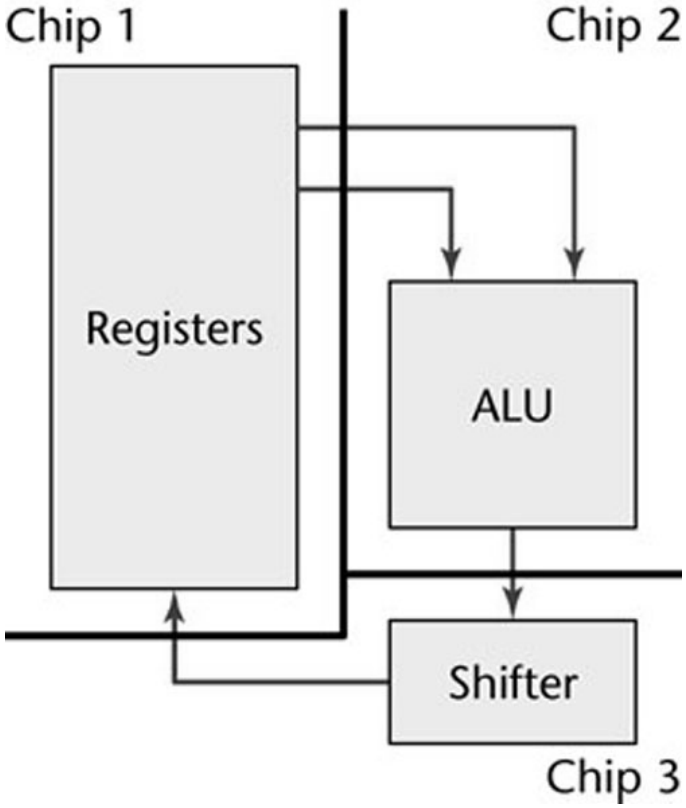
Example: Design of Computer

- A highly incompetent computer architect decides to build an ALU, shifter, and 16 registers with AND, OR, and NOT gates, rather than NAND or NOR gates.



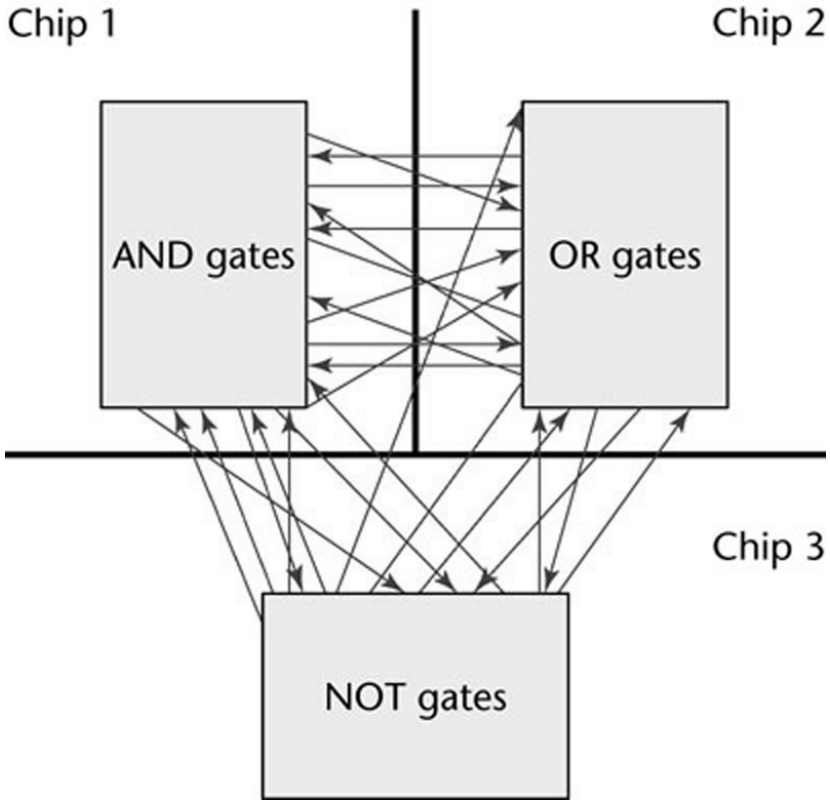
Example: Design of Computer (contd)

- The architect designs 3 silicon chips



Example: Design of Computer (contd)

- Redesign with one gate type per chip
- Resulting "masterpiece"



Example: Design of Computer (contd)

- The two designs are functionally equivalent
  - The second design is
    - Hard to understand
    - Hard to locate faults
    - Difficult to extend or enhance
    - Cannot be reused in another product

- Modules must be like the first design
  - Maximal relationships within modules, and
  - Minimal relationships between modules

Characteristics of Good Design –1

- Design quality is an elusive concept
- A “good” design may be the most efficient, the cheapest, the most maintainable, the most reliable, etc. => tradeoffs
- Quality characteristics are equally applicable to function-oriented and object-oriented designs

Characteristics of Good Design –2

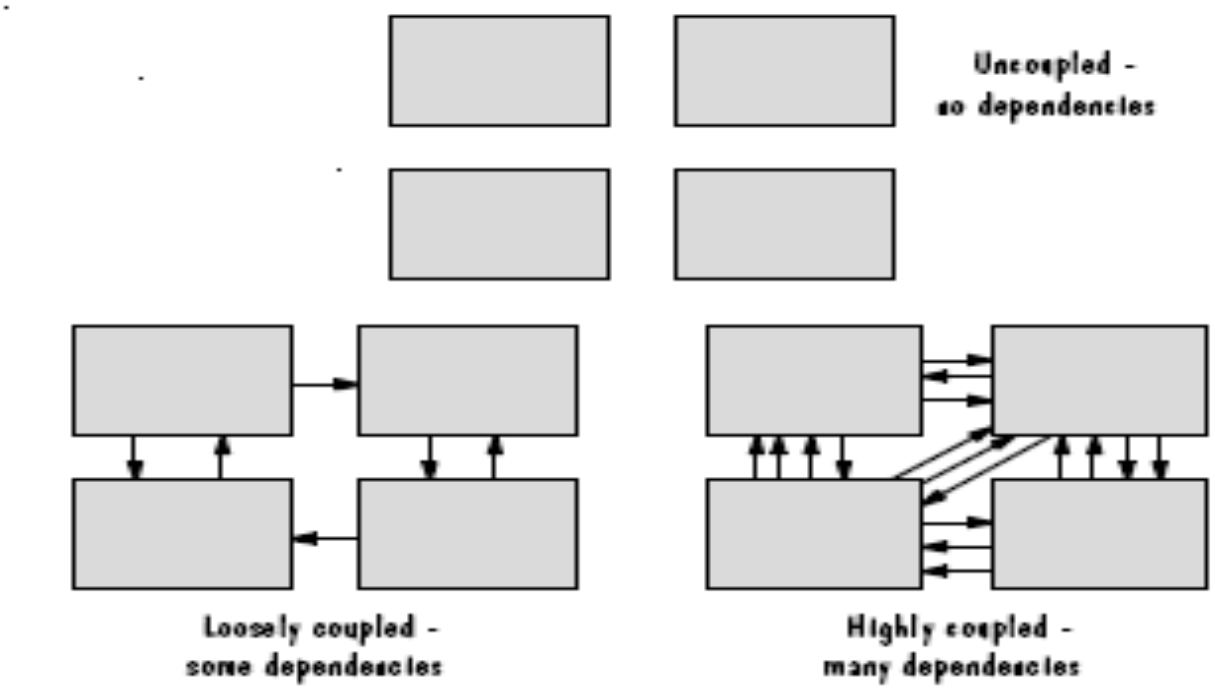
- Component Independence
  - Coupling: the degree of interaction between two modules
  - Cohesion: the degree of interaction within a module
- ✓ Good design has
  - high cohesion and low coupling

- ➢ Exception identification and handling
- ➢ Fault prevention and fault tolerance

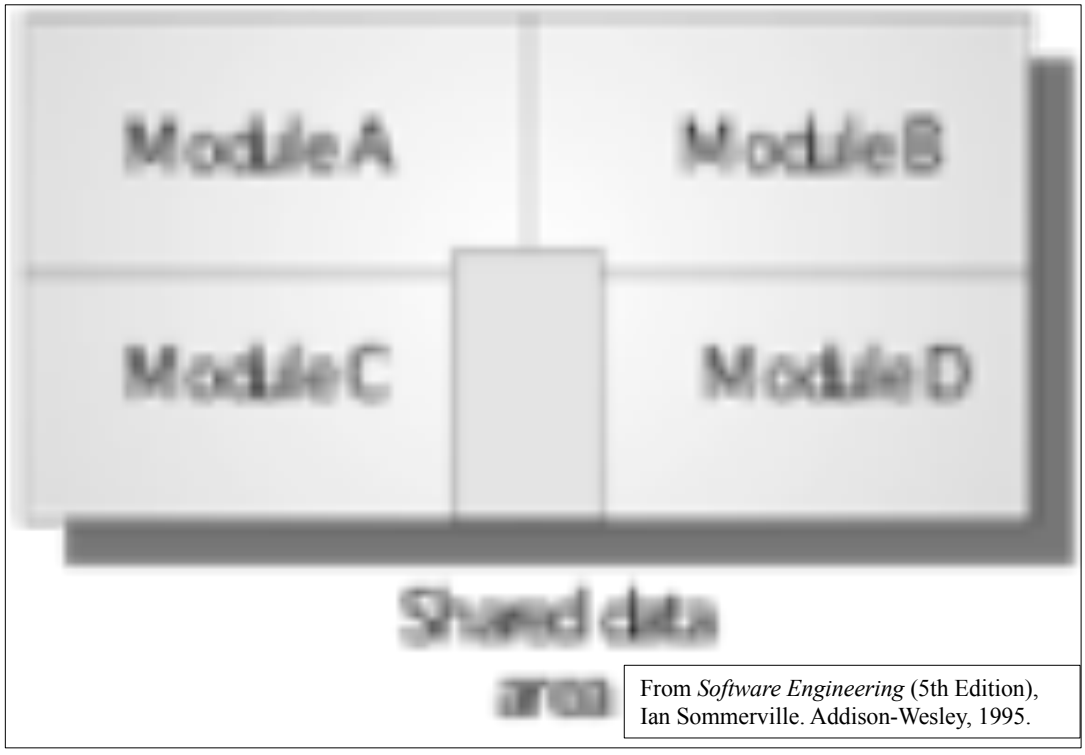
Coupling –1

- An ***inter***-component measure of the strength of the connections between system entities
- Loose coupling is a desirable quality: it means component changes are unlikely to affect others
- Shared variables or control information exchange lead to tight coupling
- Loose coupling can be achieved by state decentralization and component communication via parameters or message passing

Coupling –2

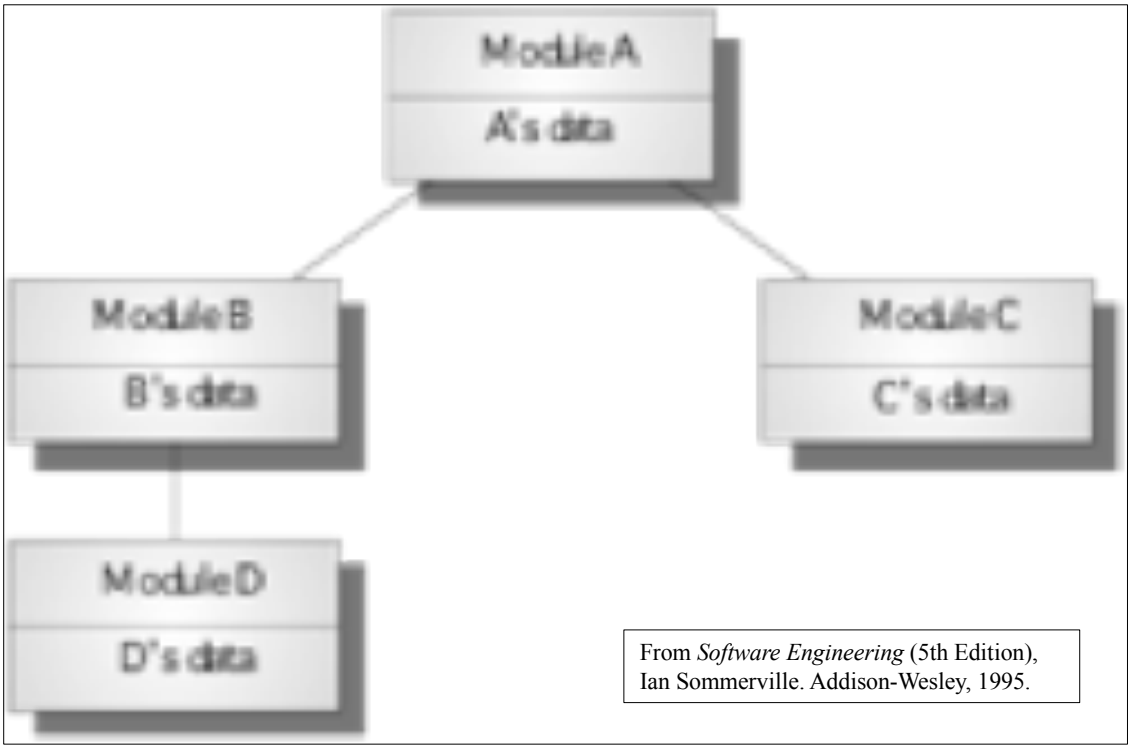

Uncoupled -
no dependencies
Loosely coupled -
some dependencies
Highly coupled -
many dependencies

Coupling –3: Tight



From *Software Engineering* (5th Edition), Ian Sommerville. Addison-Wesley, 1995.

Coupling –3: Loose



From *Software Engineering* (5th Edition), Ian Sommerville. Addison-Wesley, 1995.

Five categories or levels of coupling (non-linear scale)

5.	Data coupling	(Good)
4.	Stamp coupling	
3.	Control coupling	
2.	Common coupling	
1.	Content coupling	(Bad)

Content Coupling

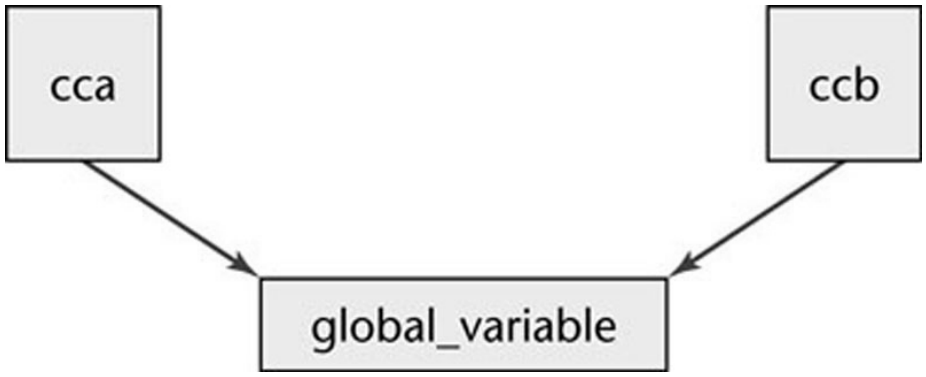
- Two modules are content coupled if one directly references contents of the other
- Example 1:
  - Module `p` modifies a statement of module `q`
- Example 2:
  - Module `p` refers to local data of module `q` in terms of some numerical displacement within `q`
- Example 3:
  - Module `p` branches into a local label of module `q`

Why Is Content Coupling not good?

- Almost any change to module `q`, even recompiling `q` with a new compiler or assembler, requires a change to module `p`
- `It is impossible to reuse module p in some new product w/o reuse module q`

Common Coupling

- Two modules are common coupled if they have write access to global data



- Example 1
  - Modules `cca` and `ccb` can access ***and change*** the value of `global_variable`
- Problem: it can be difficult to determine which component is responsible for having set a variable to a particular value.

Other Couplings:

- Two modules are *control coupled* if one passes an element of control to the other
- Two modules are *stamp coupled* if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure
- Two modules are *data coupled* if all parameters are homogeneous data items (simple parameters, or data structures all of whose elements are used by called module)
- Examples of data coupling:
  - `display_time_of_arrival (flight_number);`
  - `compute_product (first_number, second_number);`
  - `get_job_with_highest_priority (job_queue);`

The Importance of Coupling

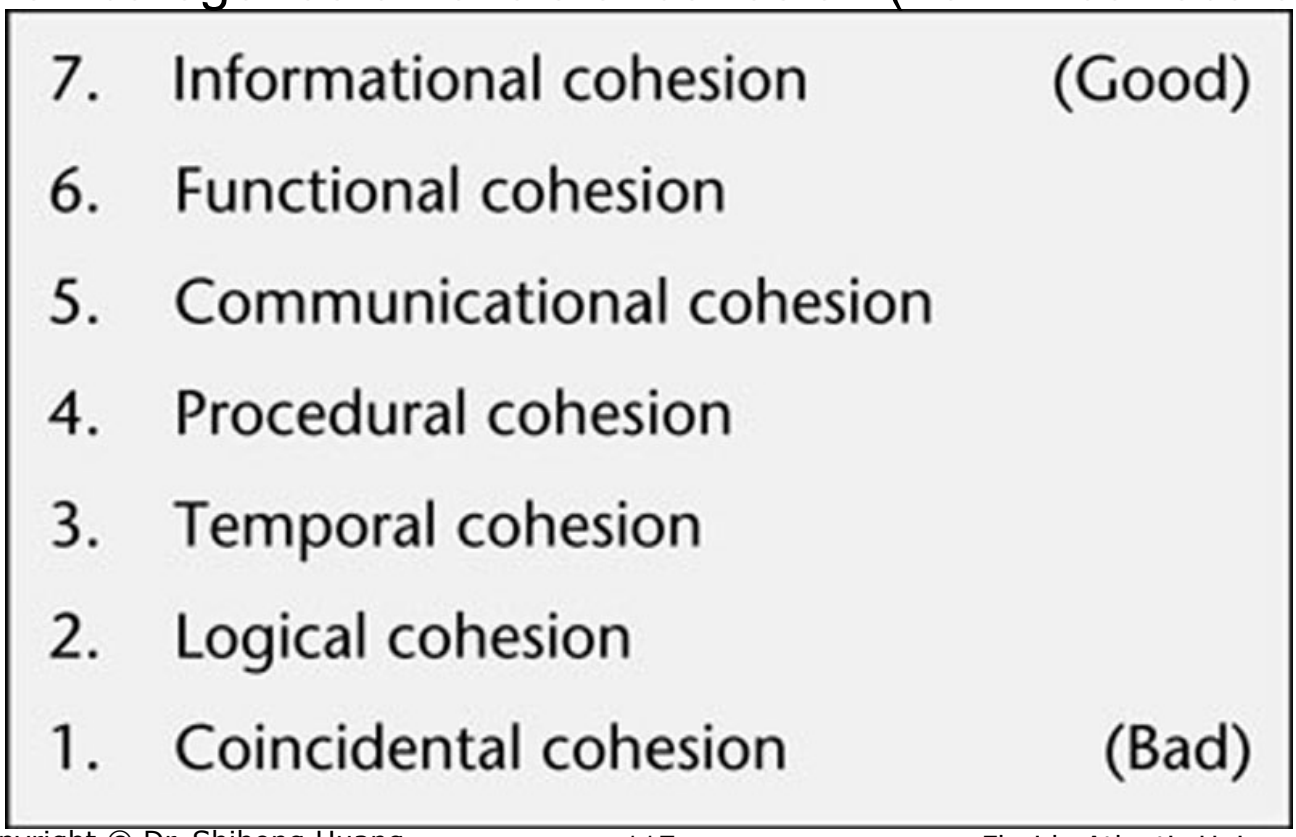
- As a result of tight coupling
  - A change to module `p` can require a corresponding change to module `q`
  - If the corresponding change is not made, this leads to faults

Cohesion –1

- An ***intra***-component measure of how a entity "fits together"
- Implement a single logical requirement; otherwise, "interleaving" dirties design
- High cohesion is a desirable design quality attribute for components: when a change is made, it is localized in a single unit
- Different levels of cohesion identified

Cohesion --1

Seven categories or levels of cohesion (non-linear scale)



Coincidental Cohesion (weak)

- Parts of a component are simply bundled together
- It performs multiple, completely unrelated actions
- Example:
  - `print_next_line, reverse_string_of_characters_comprising_second_ parameter, add_7_to_fifth_parameter, convert_fourth_parameter_to_ floating_point`
- Such modules arise from rules like
  - "Every module will consist of between 35 and 50 statements"

Why Is Coincidental Cohesion So Bad?

- It degrades maintainability
- A module with coincidental cohesion is not reusable
- The problem is easy to fix
  - Break the module into separate modules, each performing one task

Logical Cohesion (Weak)

- Components performing similar functions are grouped
- It performs a series of related actions, one of which is selected by the calling module
- Example 1:

```
function_code = 7;
new_operation (function_code, dummy_1, dummy_2, dummy_3);
// dummy_1, dummy_2, and dummy_3 are dummy variables,
// not used if function code is equal to 7
```

- Example 2:
  - An object performing all input and output
- Example 3: A module that edits insertions, deletions, and modifications of master file records

Why Logical Cohesion is bad?

- The interface is difficult to understand
- Code for more than one action may be intertwined
- Difficult to reuse

Why Logical Cohesion is bad (contd)

E.g.: A new tape unit is installed

- What is the effect on the laser printer?

1. Code for all input and output
2. Code for input only
3. Code for output only
4. Code for disk and tape I/O
5. Code for disk I/O
6. Code for tape I/O
7. Code for disk input
8. Code for disk output
9. Code for tape input
10. Code for tape output
⋮ ⋮ ⋮
37. Code for keyboard input

Temporal Cohesion (weak)

- A module has temporal cohesion when it performs a series of actions related in time
- E.g.: a component is used to initialize a system or a set of variables. Such component performs several functions in sequence, but the functions are related only by the timing involved
- Example:
  - `open_old_master_file, new_master_file, transaction_file, and print_file; initialize_sales_district_table, read_first_transaction_record, read_first_old_master_record (a.k.a. perform_initialization)`

Why Temporal Cohesion is bad?

- The actions of this module are weakly related to one another, but strongly related to actions in other modules
  - Consider `sales_district_table`
- Not reusable

Procedure Cohesion (weak)

- A module has procedural cohesion if it performs a series of actions related by the procedure to be followed by the product
- Component elements form single control sequence
- Example:
  - `read_part_number_and_update_repair_record_on_master_file`

Communicational Cohesion (medium):

- A module has communicational cohesion if it performs a series of actions related by the procedure to be followed by the product, but in addition all the actions operate on the same data
- Elements of a component operate on the same input or produce the same output
- Example 1:

```
update_record_in_database_and_write_it_to_audit_trail
```

- Example 2:

```
calculate_new_coordinates_and_send_them_to_terminal
```

Functional cohesion (strong)

- A module with functional cohesion performs exactly one action
- Each component part needed for single execution
- Examples:
  - `get_temperature_of_furnace`
  - `compute_orbital_of_electron`
  - `write_to_floppy_disk`
  - `calculate_sales_commission`

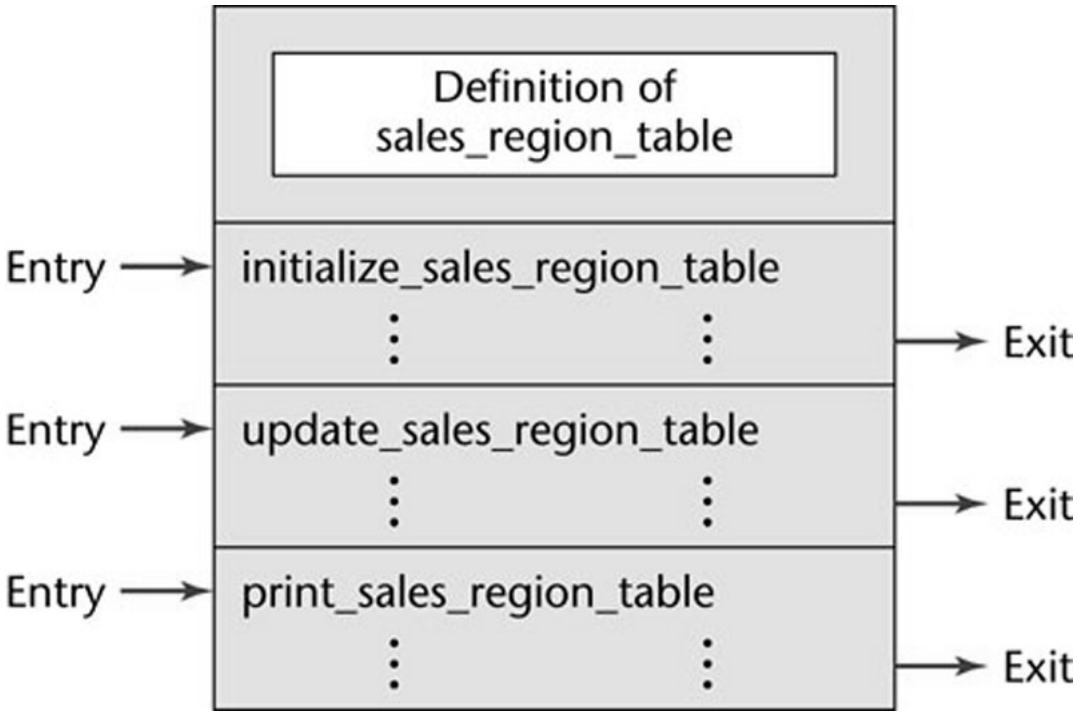
Why Is Functional Cohesion So Good?

- More reusable
- Corrective maintenance is easier
  - Fault isolation
  - Fewer regression faults
- Easier to extend a product

Informational cohesion

- A module has informational cohesion if it performs a number of actions, each with its own entry point, with independent code for each action, all performed on the same data structure

Why Is Informational Cohesion So Good?



- Essentially, this is an abstract data type --> an Object

Exception Identification and Handling

- Design defensively – anticipate situations that might lead to system problems
- Typical exceptions
  - Failure to provide a service
  - Providing the wrong service or data
  - Corrupting data
- For identified exception, we handle it with:
  - Retrying: restore the system to its previous state and try again to perform the service using a different strategy
  - Correct: restore the system to its previous state, correct some aspect of the system, try again by using the same strategy
  - Report: restore system to its previous state, report the problem to an error handling component, do not provide service

Summary

- High quality design should lead to quality products
- Ease of (understanding, implementation, testing, maintenance)
- Attributes that reflect design quality include coupling, cohesion, exception detection and handling

References

- E. Yourdon and L. Constantine, “Structured Design: Fundamentals of a Discipline of Computer Program and System Design”, Prentice-Hall, 1979
- S. Schach, “Object-Orientd & Classical Software Engineering” 6th Ed. 2005 (Session 7.2 “Cohesion”; 7.3 “Coupling”)
- S. Pfleeger, “Sofware Engineering Theory and Practice” 2nd ed. 2001

Design 3

Dr. Shihong Huang

Department of Computer Science & Engineering
Florida Atlantic University

Objectives

- Understand concepts of decomposition and modularity
- Understand decomposition techniques

Outline

- System Design
- System Module
- Decomposition Techniques

System Design and Modular

- To design a system is to determine a set of components and intercomponent interfaces that satisfy a specific set of requirements [DeMarco 1982]
- A system is *modular* when each activity of the system is performed by exactly one component, and when the inputs and outputs of each component are well-defined

Decomposition Techniques

- Although it is sometimes suggested that one approach to design is superior, in practice, an object-oriented and a functional approach to design are often complementary
- Good software engineers should select the most appropriate approach for whatever subsystem is being designed

Modular decomposition styles

- Styles of decomposing sub-systems into modules.
- No rigid distinction between system organization and modular decomposition.

Sub-systems and modules

- A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems.
- A module is a system component that provides services to other components but would not normally be considered as a separate system.

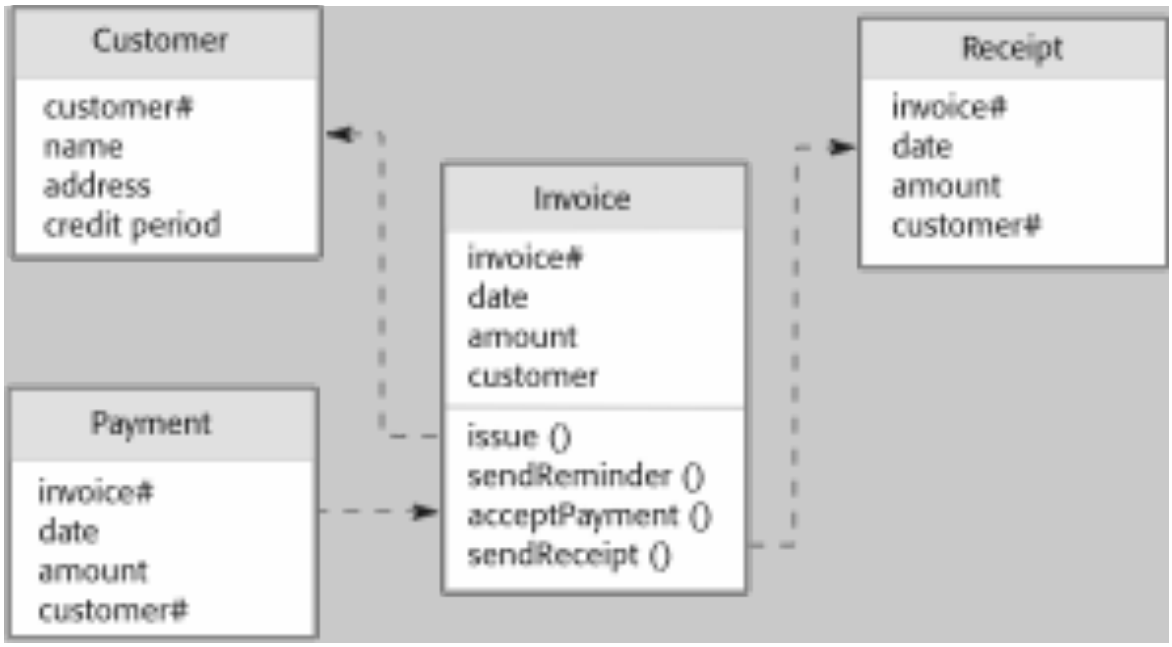
Modular decomposition

- Another structural level where sub-systems are decomposed into modules.
- Two modular decomposition models covered
  - An object model where the system is decomposed into interacting object;
  - A pipeline or data-flow model where the system is decomposed into functional modules which transform inputs to outputs.
- If possible, decisions about concurrency should be delayed until modules are implemented.

Object-Oriented Design

- System designed as a collection of interacting objects
- Structure the system into a set of loosely coupled objects with well-defined interfaces.
- Object-oriented decomposition is concerned with identifying object classes, their attributes and operations.
- When implemented, objects are created from these classes and some control model used to coordinate object operations

Example: Object Model of Invoice Process System



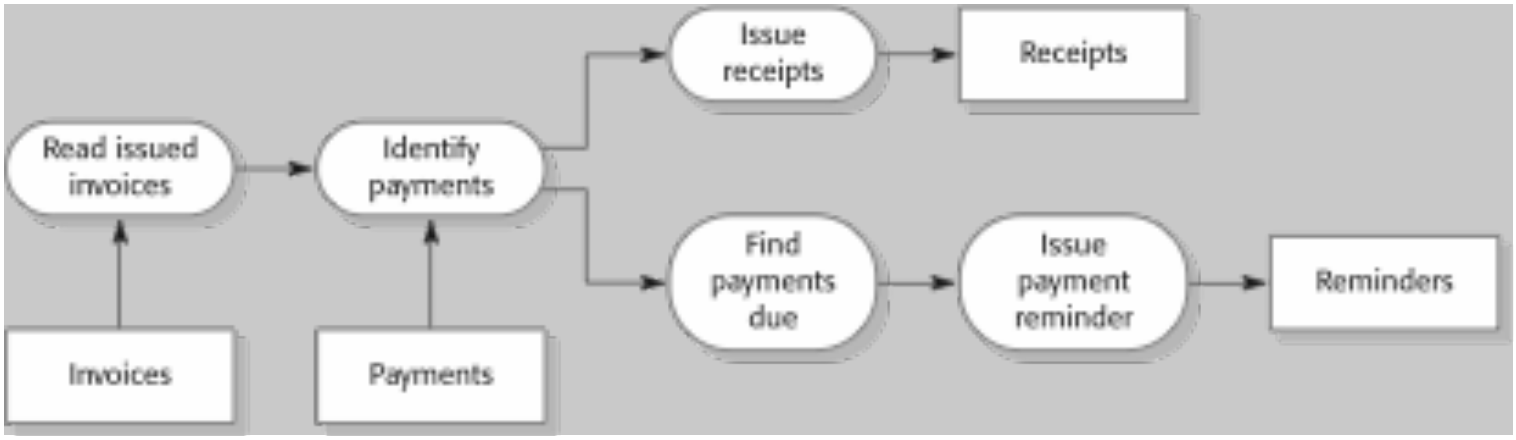
Object-Oriented Design Advantage

- Objects are loosely coupled so their implementation can be modified without affecting other objects.
- The objects may reflect real-world entities.
- OO implementation languages are widely used.
- However, object interface changes may cause problems and complex entities may be hard to represent as objects.

Function-Oriented Design

- Also called data-flow model
- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems

Invoice processing system



Data Flow Analysis (DFA) –1

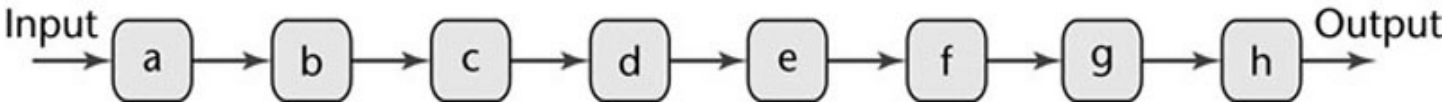
- Consider the system as a transformer of data
- DFA diagram shows data that flow into the system, how they are transformed, and how they leave the system
- Use it with most specification methods (Structured Systems Analysis here)
- Key point: We have detailed action information from the DFD
- Every product transforms input into output

Data-flow model advantages

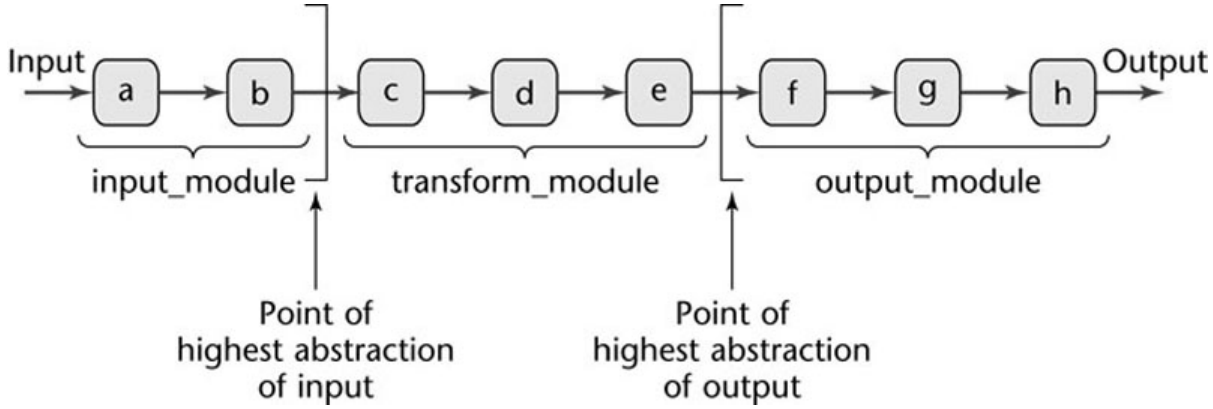
- Supports transformation reuse.
- Intuitive organization for stakeholder communication.
- Easy to add new transformations.
- Relatively simple to implement as either a concurrent or sequential system.
- However, requires a common format for data transfer along the pipeline and difficult to support event-based interaction.

Data Flow Analysis (DFA) –2

Example: A DFD showing flow of data and operations



Data Flow Analysis (DFA) –3



- Decompose the product into three modules
- Repeat stepwise until each module has high cohesion
  - Minor modifications may be needed to lower the coupling

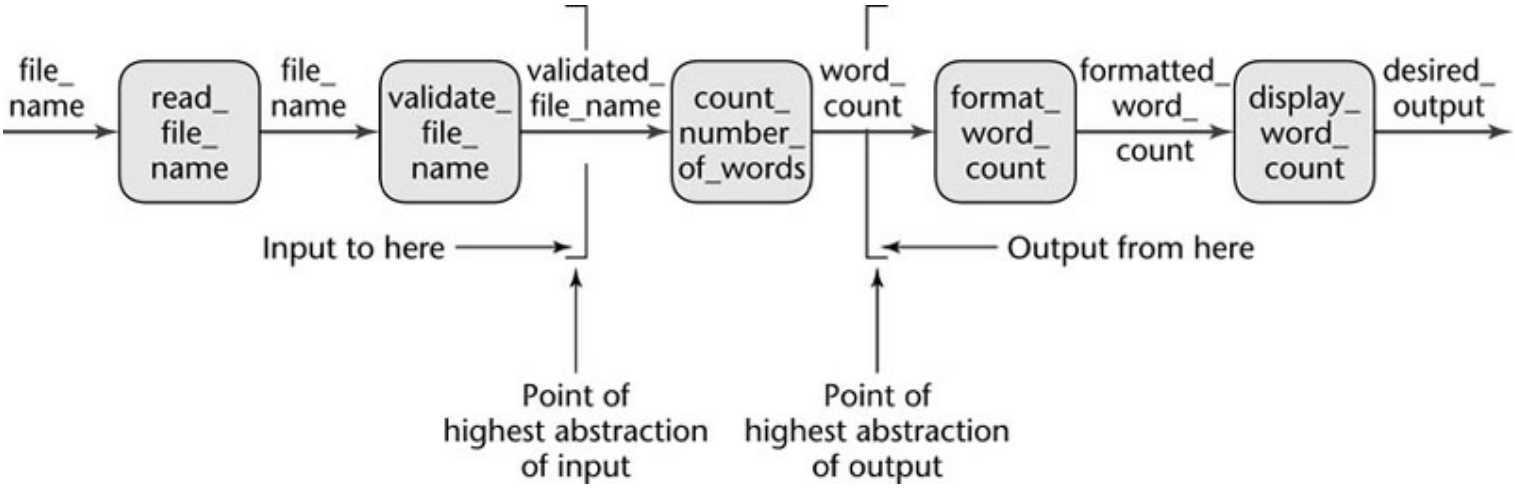
Word Counting Example –1

- The requirement:

  "Design a product which takes as input a file name, and returns the number of words in that file (like UNIX *wc*)"

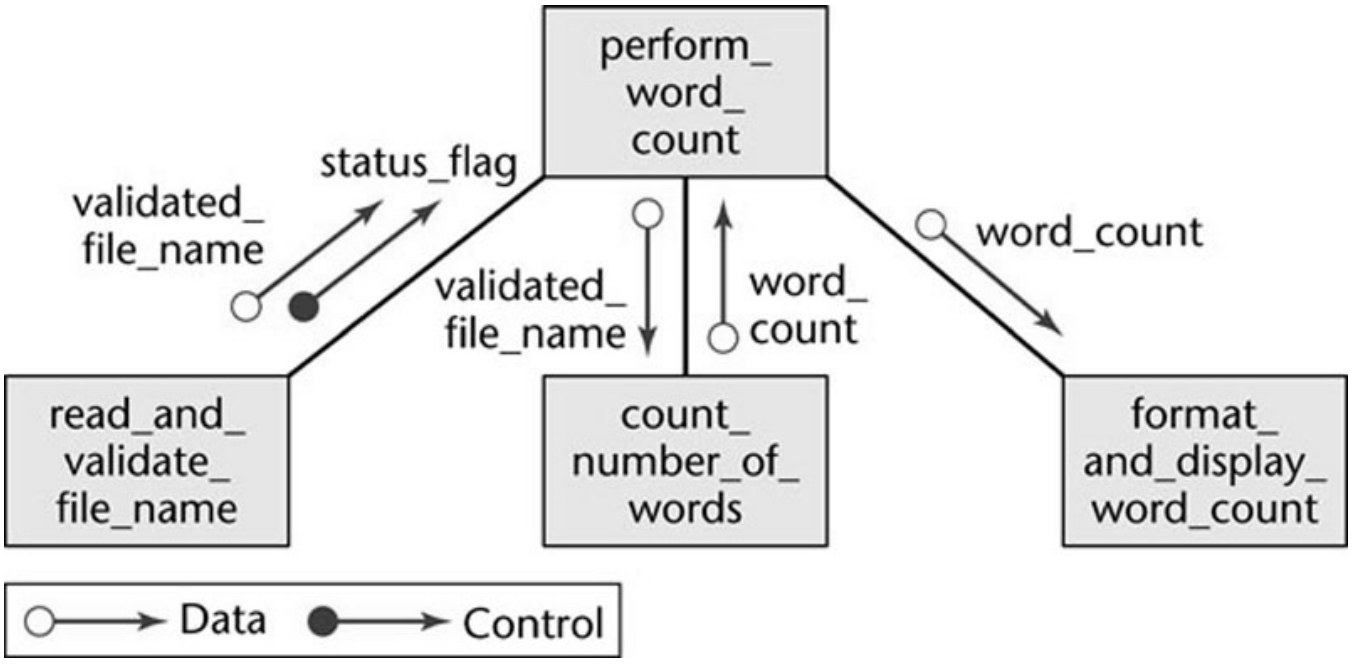
Word Counting Example –2

The initial DFD



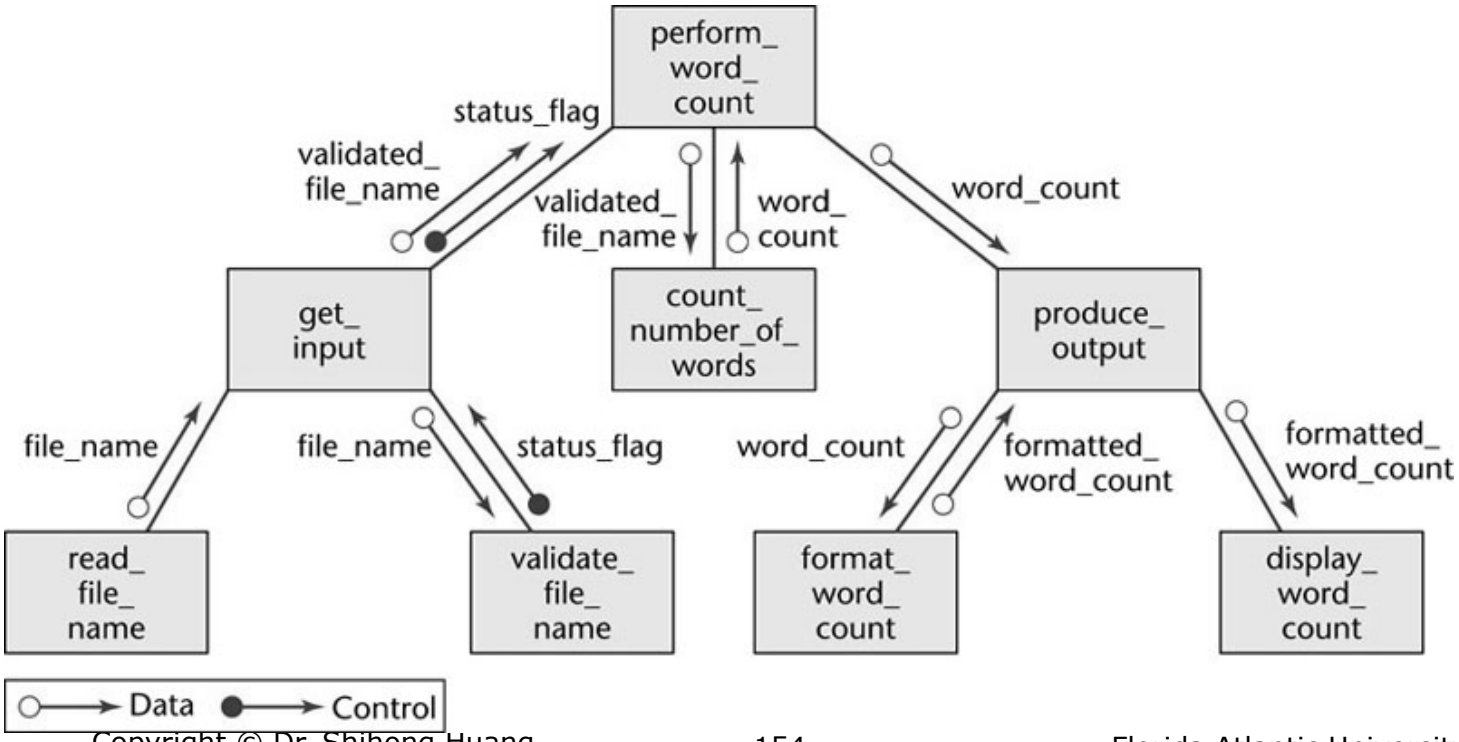
Word Counting Example –3

- First refinement



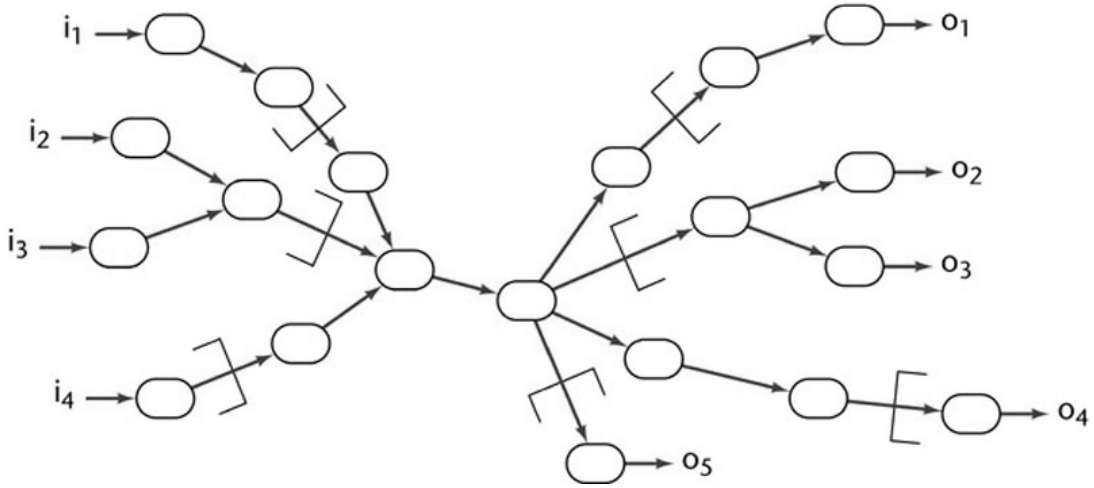
Word Counting Example –4

- Second refinement: all 8 modules now have functional cohesion



Data Flow Analysis (DFA) –4

- In real-world products, there is
  - More than one input stream, and
  - More than one output stream
- Find the point of highest abstraction for each stream



- Continue until each module has high cohesion
  - Adjust the coupling if needed

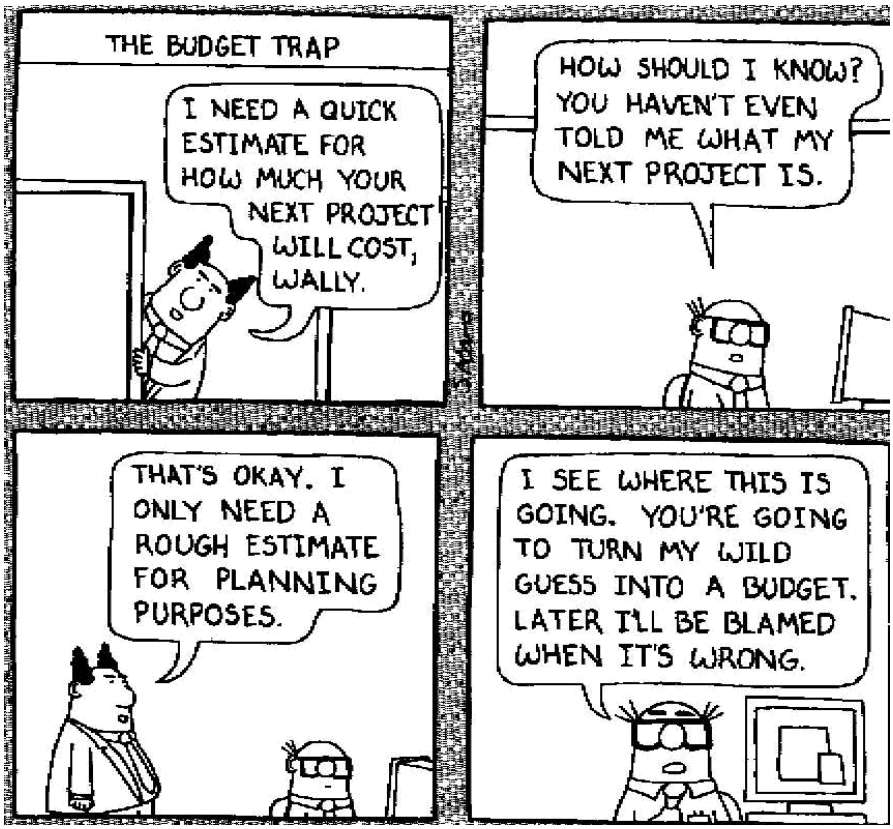
Summary

- There are more than one way to create good designs
- However, each design method involves some kind of decomposition
- Main (modern) decomposition are functional and object-oriented

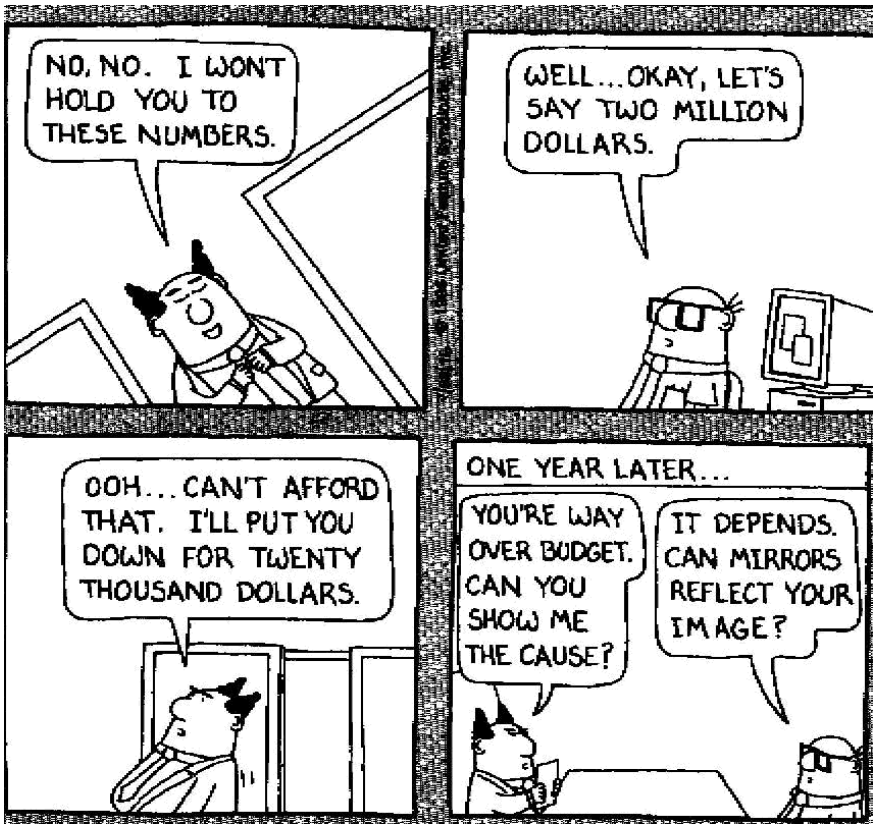
References

- "Software Engineering" 7th edition; Ian Sommerville; Addison-Wesley June 2004
- Software Engineering: Theory and Practice 2nd edition; Shari L. Pfleeger; Prentice Hall 2001
- "Object-Oriented & Classical Software Engineering" 6th edition; Stephen R. Schach; Mc-Graw Hill 2004

“Dilbert” on Cost Estimation –1



“Dilbert” on Cost Estimation –2



Cost & Effort Estimation Techniques

- Expert judgment
- Estimation by analogy
- Parkinson's Law
- Pricing to win
- Top-down estimation
- Bottom-up estimation
- Algorithmic cost modeling

COCOMO Model

- COCOMO: COnstructive COst MOdel (Barry Boehm, USC; 1981)
- An empirical model based on project experience.
- Well-documented, 'independent' model which is not tied to a specific software vendor.
- Long history from initial version published in 1981 (COCOMO-81) through various instantiations to COCOMO 2.
- COCOMO 2 takes into account different approaches to software development, reuse, etc. Published in 2000

COCOMO 81

- Exists in three stages:
  - Basic: estimate based on product attributes
  - Intermediate: modifies basic estimate using project and process attributes
  - Advanced: estimates project phases and parts separate
- Assume waterfall model is used in software development, and using standard imperative program languages (C, FORTRAN)

COCOMO 81

Project complexity	Formula	Description
Simple	$PM = 2.4\ (KDSI)^{1.05} \times M$	Well-understood applications developed by small teams.
Moderate	$PM = 3.0\ (KDSI)^{1.12} \times M$	More complex projects where team members may have limited experience of related systems.
Embedded	$PM = 3.6\ (KDSI)^{1.20} \times M$	Complex projects where the software is part of a strongly coupled complex of hardware, software, regulations and operational procedures.

- Multiplier M reflects product, project and team characteristics

COCOMO Model Effort Multipliers

	Rating					
Cost Drivers	**Very Low**	**Low**	**Nominal**	**High**	**Very High**	**Extra High**
Product Attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Database size		0.94	1.00	1.08	1.16	
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
Execution time constraint			1.00	1.11	1.30	1.66
Main storage constraint			1.00	1.06	1.21	1.56
Virtual machine volatility*		0.87	1.00	1.15	1.30	
Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel Attributes						
Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Programmer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience*	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

*For a given software product, the underlying virtual machine is the complex of hardware and software (operating system, database management system) it calls on to accomplish its task.

Professionalism –1

- Software is more than technical issues: ethical, social, professional, ..
- Personal choices:
  - Military systems development
  - Chicken little (cf. Y2K)
  - Community involvement
- Developers and maintainers need to be:
  - Hardworking
  - Intelligent
  - Sensible
  - Up-to-date, and above all..
  - Ethical

Professionalism –2

- Software engineering involves wider responsibilities than simply the application of technical skills
- Software engineers must behave in an honest and ethically responsible way if they are to be respected as professionals
- Ethical behaviour is more than simply upholding the law

Professionalism –3

- Confidentiality
  - Engineers should normally respect the confidentiality of their employers or clients irrespective of whether or not a formal confidentiality agreement has been signed
- Competence
  - Engineers should not misrepresent their level of competence. They should not knowingly accept work which is beyond their competence
- Intellectual property rights
  - Engineers should be aware of local laws governing the use of intellectual property such as patents, copyright, etc. They should be careful to ensure that the intellectual property of employers and clients is protected
- Computer misuse
  - Software engineers should not use their technical skills to misuse other people's computers. Computer misuse ranges from relatively trivial (game playing on an employer's machine, say) to extremely serious (dissemination of viruses).

Professionalism –4

Code of Ethics

- IEEE-CS ACM Software Engineering Code of Ethics and Professional Practice (www.computer.org/tab/seprof/code.htm)
- Members of these organisations sign up to the code of practice when they join
- The Code contains eight principles related to the behaviour of and decisions made by professional software engineers, including practitioners, educators, managers, supervisors and policy makers, as well as trainees and students of the profession

Professionalism –5

- SWEBOK: An attempt to codify the "Software Engineering Body of Knowledge"
- SEEK: A companion to SWEBOK, focusing on software engineering education

Summary –1

- Software is a critical and complex part of everyday life that impacts everyone
- Software engineering is a young, inter-disciplinary, and evolving field of study
- Technical and non-technical aspects are equally important to a successful project

Summary –2

- The software process includes all activities of software engineering (not just coding)
- There are several process models, sharing the common and essential activities of requirements, design, construction, testing, and maintenance

Summary –3

- Software engineering is not (yet) a generally recognized profession, but there are a code of ethics to follow
- To be a good software engineer, you need:
  - Inquisitive problem solving skills
  - To like to take things apart, to see what "makes them tick"
  - Agility and self-discipline

References

- ACM/IEEE *Software Engineering Body of Knowledge* project: www.swebok.org