Tamkang University Software Engineering Group
淡 江 軟 體 工 程 實 驗 室
http://www.tkse.tku.edu.tw/

Software Engineering

教材：
1. Roger S. Pressman. Software Engineering: a practitioner's approach, 6th edition. McGRAW-HILL
2. 軟體工程聯盟編撰教材

Presented by：Ying-Hong Wang
E-mail：inhon@mail.tku.edu.tw
Date：2012/3/18

上課用書、參考用書暨相關規定

- 成績評定
  - 出席15%、平時作業40% 、期中報告15%、期末報告20%
  - 出席成績僅考核出席與否，故不接受任何請假，但每人每學期有三次免責，全勤者於學期成績另加3分
  - 演講一場：需繳交隨堂聽講報告(10%) ，若無法安排演講，此項分數併入平時作業成績
  - 一般作業遲交每24小時扣10分、隨堂作業、期中與期末報告不可遲交
  - 期中報告繳交時間：4/13 1200前、期末報告繳交時間：6/8 1200前
- 上課方式
  - 投影片爲主、板書爲輔
- 上課規定
  - 手機改設震動或關機、不要私下講話

課程目標

- 瞭解軟體專案開發流程與方法
- 學習軟體專案開發程序、方法與工具
- 輔助軟體工具：**ArgoUML**
  - 下載網站**http://tw.opensourceinstall.org**

調整上課心態與學習態度

- 以前修課的想法可能是求過關就好
- 以前的學習態度可能是可以順利畢業就好
- 現在妳(你)應該要爲即將就業作準備
- 現在的妳(你)應該要知道爲什麼要學、要學什麼
- 現在的妳(你)應該要開始思考如何面對競爭
- 現在的妳(你)應該要準備如何打贏一場又一場的競賽
- 現在的妳(你)應該思考如何成爲企業需要的人才

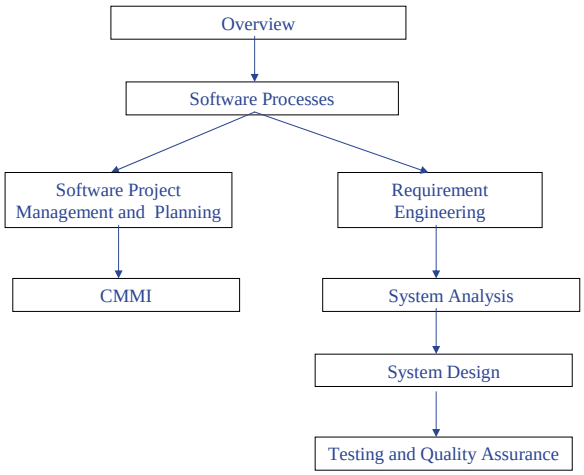
對同學們的建議

- 在專業養成上 (Hard Skills)
  - 奠定紮實的基礎
  - 養成終身學習的習慣
- 在人格養成上 (Soft Skills)
  - 建立正確的態度
  - 處事三態：真誠、負責、合群

TSEG Tamkang Universty Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Contents

Learning Objects Architecture



Contents

Preface

- The objective of this course is to explain the Introduction of software engineering , and provide an easy and practical introduction to the important characteristics of software engineering. After taking this course, students will understand:
  - what is software engineering;
  - why software engineering is important;
  - how to develop software and manage a software project by using the software engineering in detail.

軟體工程學程介紹

- 淡江大學跨院系所『軟體工程學分學程』介紹
  - 設置宗旨
    - 因應國內外軟體系統開發品質之發展，特別是台灣軟體產業處於印度、韓國與中國大陸的競爭條件下，有賴於軟體系統開發程序與品質的技術以提升相關軟體系統開發產業的品質與競爭力，是以本學程設立宗旨在於培育本校軟體系統開發流程、品質控制與專案管理之專業人才。
  - 學程必選修科目暨其學分數及應修學分總數
    - 學程必選修科目如附表，學生修習本學程科目學分，其中至少應有九學分不屬於學生主修系、所、加修學系及輔系之應修科目
  - 最低修業學分數
    - 23學分
  - 參考網址
    - http://www2.tku.edu.tw/~teix/CSIE/index.php?option=com_content&task=view&id=54&Itemid=111#cc

「淡江大學軟體工程學分學程」修業科目表

最低修業學分數：23 學分

科目表

基礎必修	高等程式語言(資訊系開) 商用程式設計(資管系開)	3 學分 4 學分	二選一
	資料結構與處理/演算法(資訊系開) 程式設計與資料結構(資管系開)	6 學分 6 學分	二選一
	網路概論(資訊系開) 網路與通訊(資管系開)	3 學分 2 學分	二選一
專業必選	軟體工程導論(資訊系開)	3 學分	必選
	系統分析與設計(資管系開)	4 學分	必選
進階專業選修	資訊管理導論(資管系開)	4 學分	
	物件導向技術(資管系開)	4 學分	
	軟體可靠度與檢測(資管系開)	2 學分	
	資訊系統之價值分析(資管系開)	2 學分	
	軟體代理人(資管系開)	2 學分	
	物件導向軟體工程(資訊系開)	3 學分	
	網際服務軟體工程(資訊系開)	3 學分	
	元件式軟體發展技術(資訊系開)	3 學分	
	個人軟體程序(資訊系開)	3 學分	
	UML 統一塑模語言(資訊系開)	3 學分	
	軟體測試與品質(資訊系開)	2 學分	最高承認 3 學分
	軟體品質保證(資訊系開)	2 學分	最高承認 3 學分
	資訊安全導論(資管/資工系開)	2 學分	最高承認 3 學分
	軟體型態管理(資訊/資管開)	2 學分	最高承認 3 學分
	軟體專案管理(資訊/資管開)	2 學分	最高承認 3 學分
	網路安全技術(資管系開) 網路安全(資訊系開)	2 學分 3 學分	二選一

※ 非資訊工程學系、資訊管理學系學生另需修習作業系統 2 學分、資料庫或資料庫管理 4 學分。

※ 表列選修課程得依實際情況開設

Contents

An Overview of Software Engineering

Software Crisis 軟體危機

Software Myths 軟體迷思

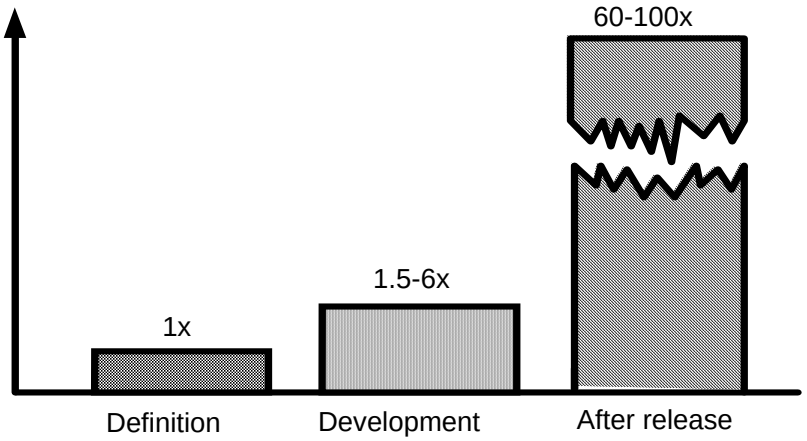
What is Software Engineering? 什麼是軟體工程?

The Evolution of Software Industry 軟體工業的演進

Software Crisis

- What is the Problem ?
- 84 % of all software projects do not finish on time and within budget (Survey conducted by Standish Group)
  - 8000 projects in US in 1995
  - More than 3o % of all projects were cancelled
  - 189 % over budget
- Key issues:
  - Software firms are always pressured to perform under unrealisticdeadlines.
  - The clients ask for new features, just before the end of the project, and unclear requirements.
  - Software itself is awfully complex.
  - Uncertainties throughout the development project.

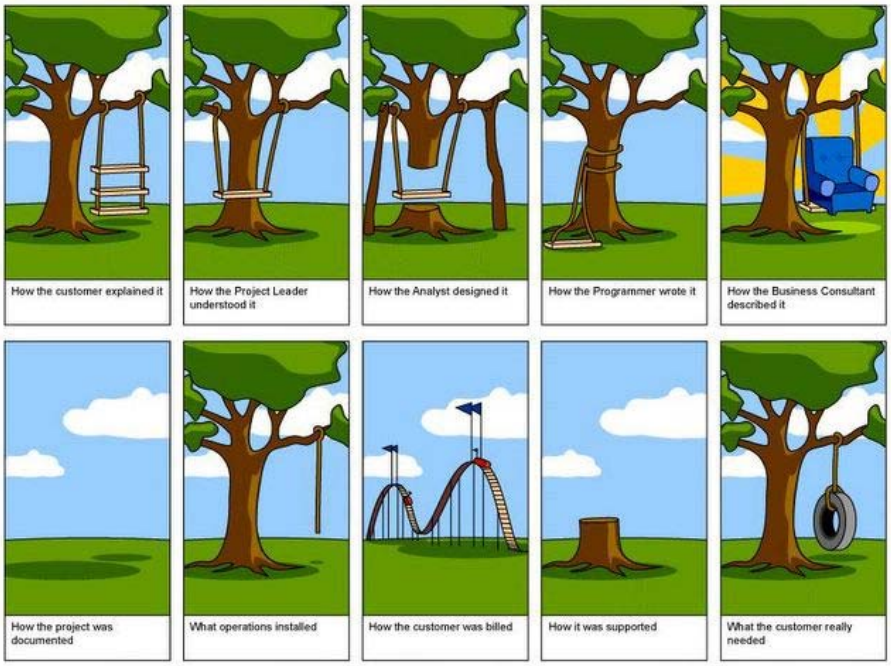
The Cost of Change



軟體越早修改，所需的成本越低

Real Cases

- Bank of America Master Net System
  - Trust business. 1982.
  - Spend 18 months in deep research & analysis of the target system.
    - Original budget: 20 million.
    - Original Schedule: 9 months, due on 1984/12/31.
    - Not until March-1987, and spent 60 million.
    - Lost 600 millions business
  - Eventually, gave up the software system and 34 billion trust accounts transferred.
- Other cases:
  - Explosion of Ariane-5 prototype in 1996 (原型機爆炸事件)
  - Explosion of Boeing's Delta III rocket. (火箭爆炸事件)



Copyright from: United States Department of Agriculture

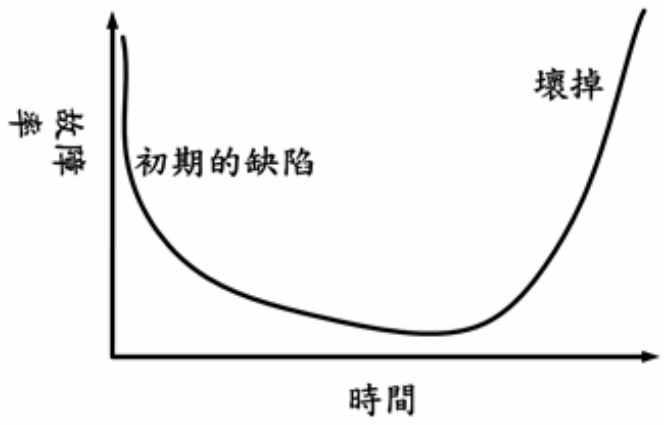
Problems of Software

- General issues 一般議題
  - HW vs. SW 硬體與軟體的比較
  - Productivity: build new programs from scratch
  - Maintenance: maintain existing programs

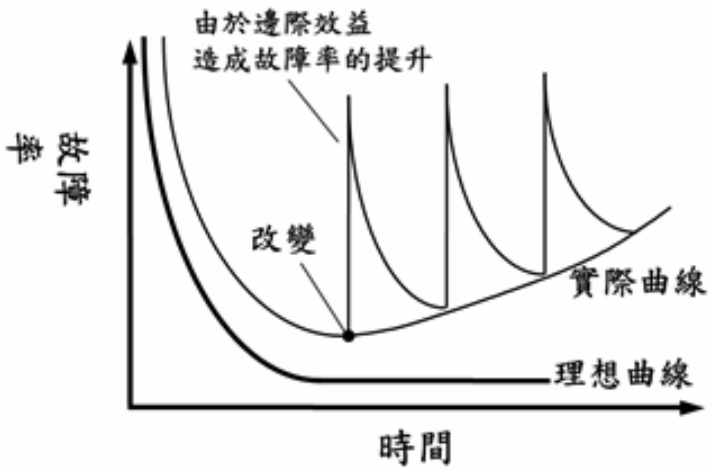
Characteristics of Software

Software	Hardware
logical system element	physical system element
developed/engineered	Manufactured
Not ware out but deteriorate -no spare parts	ware out -yes, with spare parts
Usually custom-built	assembled from existing Component

Failure Curve for Hardware (Ware out)



Failure Curve for Software (Deterioration Not Ware Out)



Software Myths

- Management Myths
  - We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?
  - My people do have state-of-the-art software development tools; after all, we buy them the newest computers
  - If we get behind schedule, we can add more programmers and catch up

Software Myths

- Customer Myths
  - A general statement of objectives is sufficient to begin writing programs – we can fill in the details later
  - Project requirements continually change, but change can be easily accommodated because software is flexible

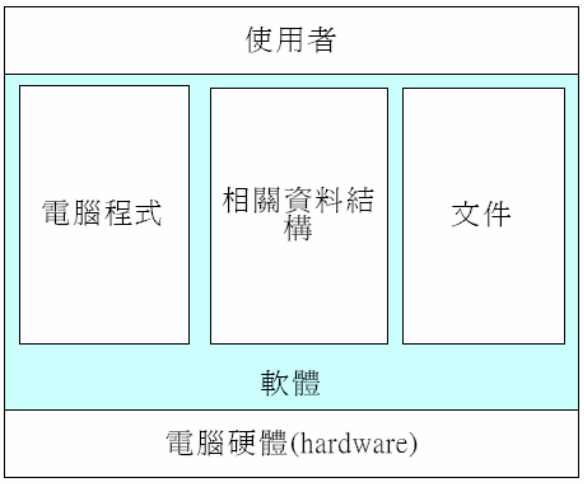
Software Myths

- Practitioner's Myths
  - Once we write the program and get it to work, our job is done
  - Until I get the program "running" I really have no way of assessing its quality
  - The only deliverable for a successful project is the working program

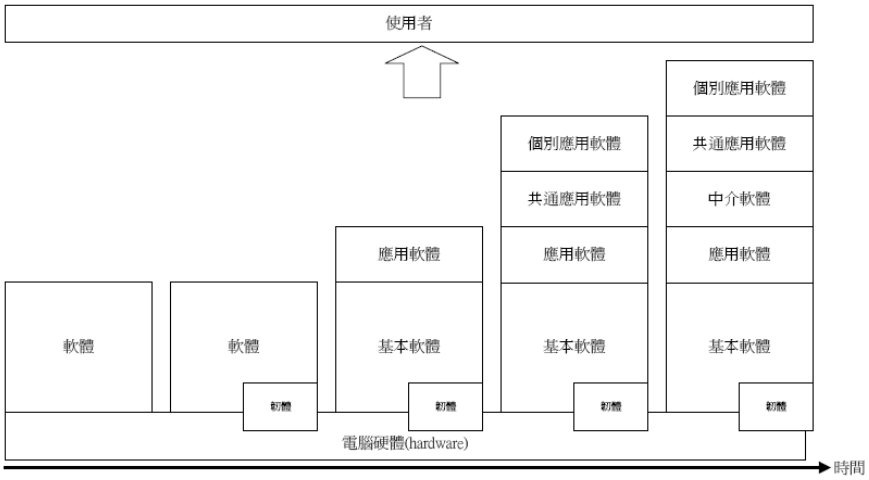
What is Software?

- Software includes
  - computer programs
  - documents
  - data & data structures

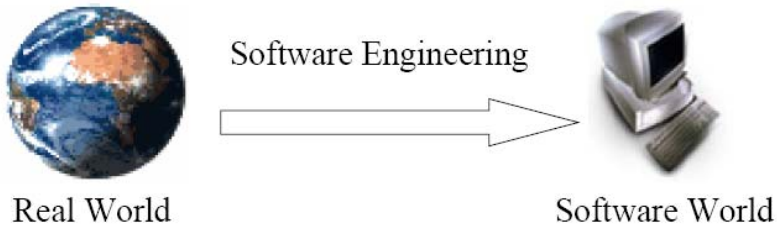
The Definition of Software?



The Different Architectures of Software



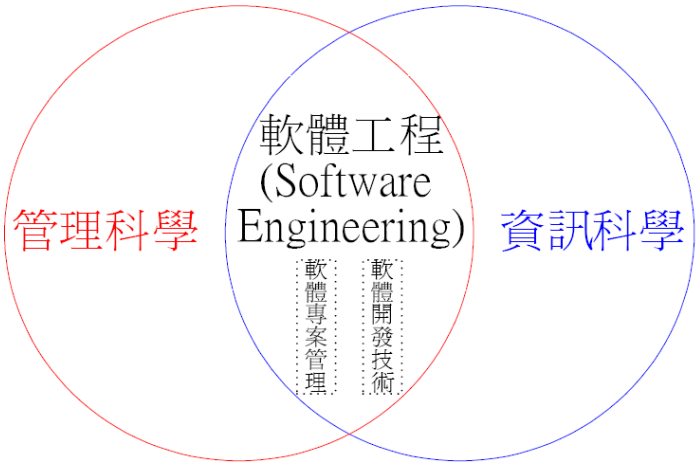
What is Software Engineering?



What is Software Engineering?

- Definition
  - 學者Stephen R. Schach (2002)對軟體工程的定義為”一門專業學門，針對在預算內生產並準時交付無錯誤且滿足客戶需求之軟體”
  - 軟體工程所包括的內容為：軟體工程是一門以研究如何以系統化、規範化、和量化之工程原則和方法來進行俱經濟效益之軟體開發和維護的學科
  - 軟體工程包括了兩方面之內容：軟體開發技術和軟體專案管理

What is Software Engineering?



What is Software Engineering?

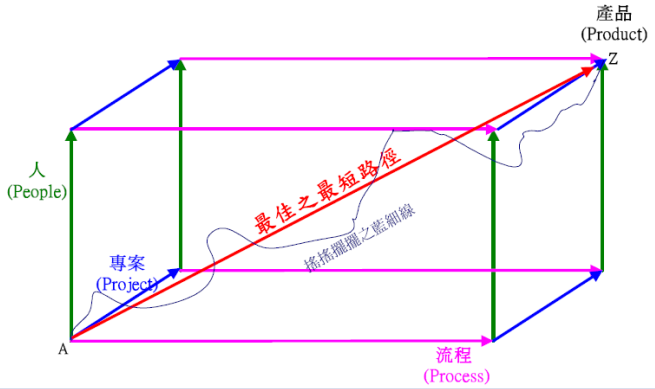
- Software engineering is a discipline that integrates methods, tools, and procedures for the development of computer software.
  - Method: introduce a way to build software
  - Tool: automatic, semi-auto support for methods
  - Procedure: define the sequence in which methods will be applied, the controls that help ensure quality and coordinate changes.

Generic View of Software Engineering

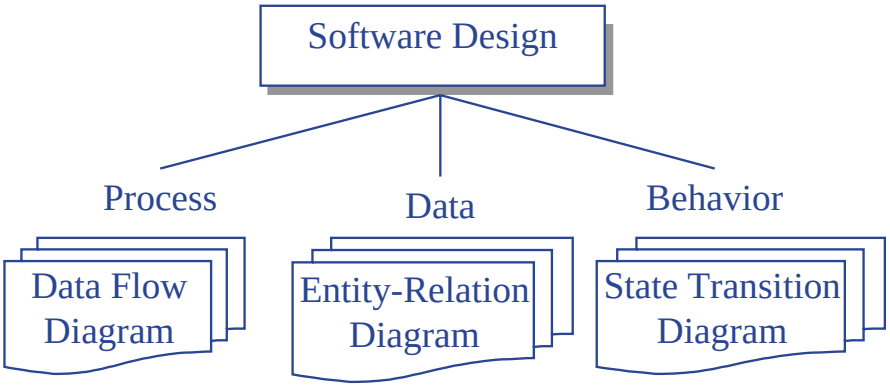
- Definition: What
- Development: How
- Maintenance: Changes

Generic View of Software Engineering

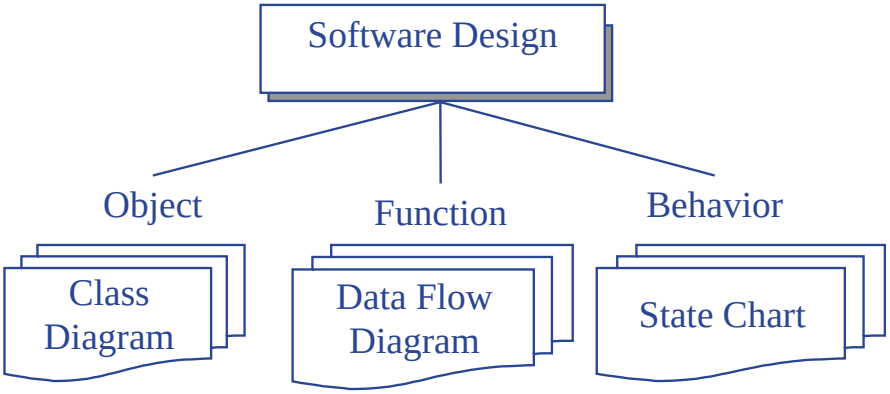
- 軟體工程的四個構面 (4Ps): People, Process, Project, and Product



Traditional Software Engineering



Object-Oriented Software Engineering



The Evolution of Software Industry

- Independent Programming Service (Era 1)
- Software Product (Era 2)
- Enterprise Solution (Era 3)
- Packaged Software for the Mass (Era 4)
- Internet Software and Services (Era 5)

Independent Programming Services (Era 1)

- Feb 1955, Elmer Kubie and John Sheldon founded CUC
  - the First Software Company that devoted to the construction of software especially for hardware company.
- Promoting Software Industry: two Major Projects,
  - SABRE, airline reservation system, $30 million.
  - SAGE, air defense system (1949~1962)
    - 700/1000 programmers in the US. $8 billion.

Software Product (Era 2)

- 1964 Martin Goetz developed Flowchart Software -- Autoflow for RCA, but rejected.
  - Sale to the customer of RCA & IBM.
  - Develop and market software products not specifically designed for a particular hardware platform.
- MARK IV, a pre-runner for the database management system.
- IBM unbundled software from hardware.

Enterprise Solutions (Era 3)

- Dietmar Hopp. IBM Germany
  - Systems, Applications and Products (SAP) $3.3billion (1997)
  - Setting up shop in Walldorf, Germany.
  - Marked by the emergence of enterprise solutions providers.

    e.g. Baan 1978. Netherlands. $680 million (1997)

    Oracle 1977. U.S.

    Larry Ellison.
  - ERP, $45 billion (1997)

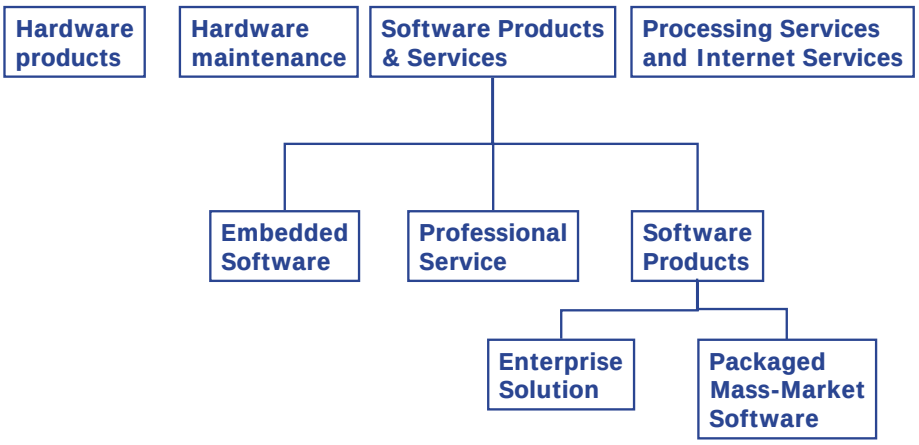
Packaged Software for the Masses (Era 4)

- Software products for the masses. 1979.
  - VisiCalc, Spreadsheet program.
- August 1981: The deal of the century.
  - Bill Gates bought the first version of the OS from a small firm called Seattle Computer Products for $50,000 without telling them it was for IBM.
  - The development of the IBM PC, 1981, initiated a 4th software era.
    - PC-based mass-market software. Few additional services are required for installation.
  - Microsoft reached revenues of $11.6 billion. Packaged Software Products, $57 billion (1997)

Internet Software and Services (Era5)

- Internet and value-added services period, 1994.
  - started with Netscape's browser software for the internet.

IT Market



Software Products and Services

Professional Software Services	Enterprise Solutions	Packaged Mass-Market Software
Anderson Consulting	IBM	Microsoft
IBM	Oracle	IBM
EDS	Computer Associates	Computer Associates
CSC	SAP	Adobe
Science Applications	HP	Novell
Cap Gemini	Fujitsu	Symantec
Hp	Hitachi	Intuit
DEC	Parametric Technology	Autodesk
Fujitsu	People Soft	Apple
BSO Origin	Siemens	The Learning Company

Contents

Software Processes

- Introduction to Software Process
- Software Life Cycle
- Software Process Models
- Comparison of Different Models

What is a Process

- A sequence of steps performed for a given purpose.
- Integrating people, tools and procedures together.
- A set of activities, methods, practices and transformations that people employ to develop and maintain software and the associated products, including project plans, design documents, code, test case and user manuals.

Software Life Cycle

- Requirement acquisition (problem statements)
  - To describe the problem to be solved and providing a conceptual overview of the proposed system
- Requirement analysis
- Requirement specification
- System analysis
- System design, Detail design
- Coding
- Testing
- Maintenance

Draft of Software Life Cycle

- Requirement analysis
- Design
- Implementation
- Testing
- Maintenance

Generic View of Software Life Cycle

- Definition: What
- Development: How
- Maintenance: Change

Requirement Analysis

- A process of discovering, refinement, modeling and specification.
  - Principles: represent information domain of a problem
    - Information flow: data & control changes
    - Information content: composite term.
    - Information structure: organization
  - Modeling: (graphical & textual description)
    - Modeling methods: SA, OOA, JSD, DSSD, SADT
    - Model component: information, function, behavior
  - Artifact
    - Requirement specification.
      - Capturing: functionality, behavior, and structure

Requirement Analysis

- SA: Structure Analysis
- OOA: Object-Oriented Analysis
- JSD: Jackson System Development
- DSSD: Data Structured Systems Development
- SADT: Structured Analysis & Design Technique

Design

- The problem is decomposed into *modules*
- The interface between modules must be specified
- Define architecture
- Artifact: design model
  - Data design
    - Data abstraction, data structure, data modeling
    - Procedural design: iteration , conditional, sequence
    - Architectural design: program structure, software architecture)

Implementation

- Individual module programming
  - Pseudo-code
- The goals
  - The development of a well-documented
  - The reliable, easy to read, flexible, correct program
- Integration of modules
- Artifact: executable program

Testing

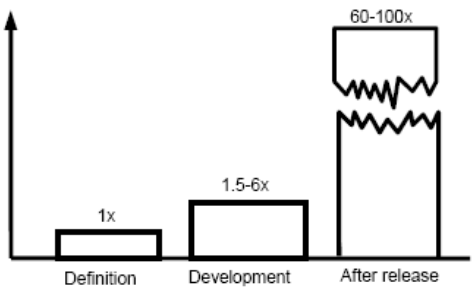
- Test the system *from requirement engineering to implementation*
  - Verification and validation
- Artifact: testing report

Maintenance

- Maintain the user satisfaction
  - Repair errors, requirements changed or extended
- Changes in both the system's environment and user requirements are inevitable
  - Maintenance = Evolution

Maintenance (cont'd)

- Kinds of maintenance activities
  - Corrective
  - Adaptive
  - Perfective
  - Preventive



The Impact of Change

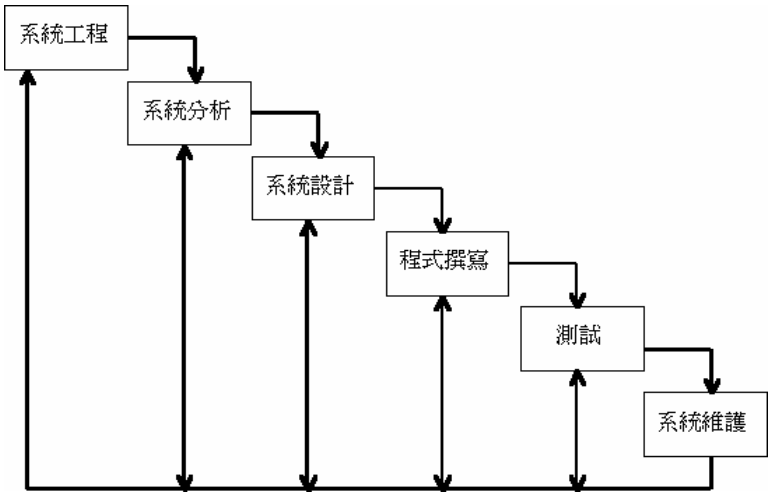
Software Process Models

- Waterfall Model
- Prototyping
- Spiral Model
- Fourth-generation Techniques
- Automatic synthesis Model
- Object-Oriented Approach
- Agile Method

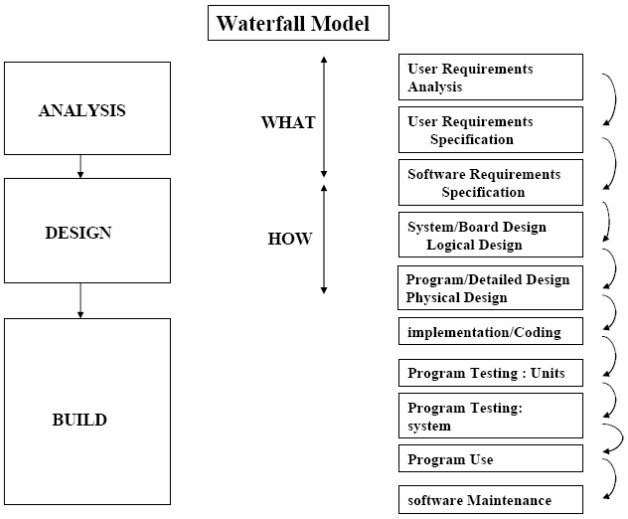
Waterfall Model

- Frequently implemented based on a view of the world interpreted in terms of a functional decomposition.
  - What does the system do?
- Based on functional decomposition.
  - Top-down analysis and design methodology
  - SA/SD
    - Based on data flows : DFD, DD, structure charts.
  - Easily to map to conventional procedural language

Waterfall Model



Waterfall Model



Prototyping Model

- Throwaway : implement only aspects poorly understood.
- Evolutionary : more likely to implement best understood benefits :
  - Improve communication
  - Reduce risk
  - Most feasible way to validate specification
  - For maintenance as well.

TSEG Tamkang Universty Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Prototyping Model

- Throwaway : implement only aspects poorly understood.
- Evolutionary : more likely to implement best understood benefits :
  - Improve communication
  - Reduce risk
  - Most feasible way to validate specification
  - For maintenance as well.

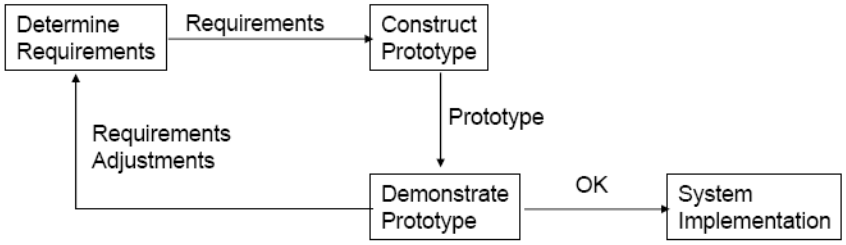
<inhon@mail.tku.edu.tw> March 18, 2012

Prototyping Model

- The roles of prototyping
  - As a means to acquire validate users requirements.
  - As scaled-down version of the final operational system.
  - As a means to validate solution specifications.
  - As a solution specification for design and implementation

TSEG Tamkang Universty Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Prototyping Model

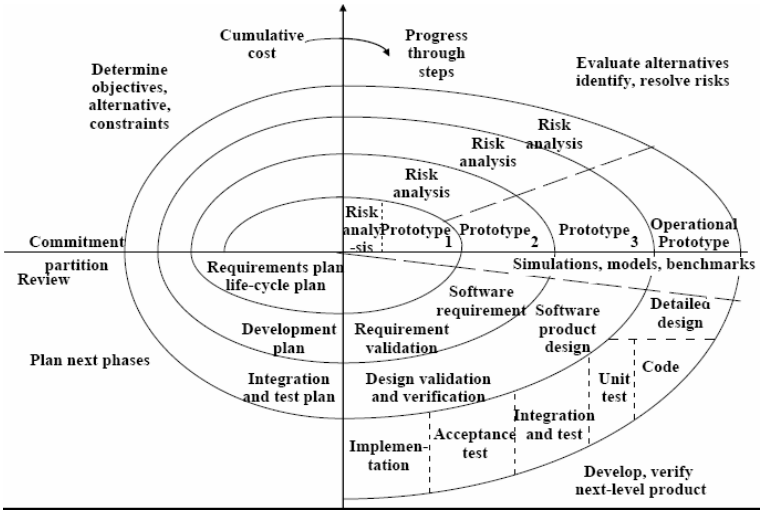


<inhon@mail.tku.edu.tw> March 18, 2012

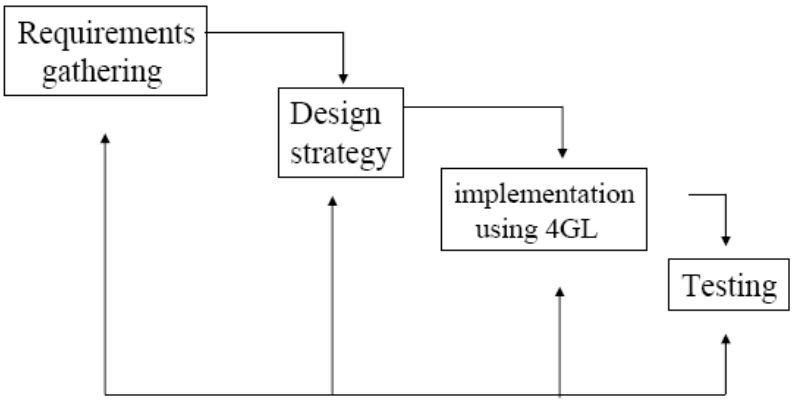
Spiral Model

- Risk driven
- Throwaway prototyping

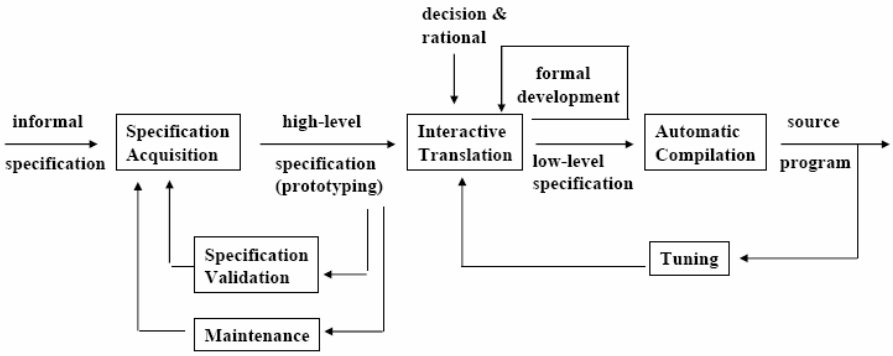
Spiral Model



Fourth Generation Technology



Automatic Synthesis Model



Object-Oriented Approach

- OOA emphasizes on finding and describing the *objects* -or *concepts* -problem domain.
- OOD emphasizes on defining logical software object that will ultimately be implemented in an object-oriented programming language.
- OOP (Programming), implements the designed components in C++, Java.

Object-Oriented Approach

Boat
age

```
Public class Boat
{
 public void sail()
 private Date age;
}
```

OOA → **OOD** → **OOP**

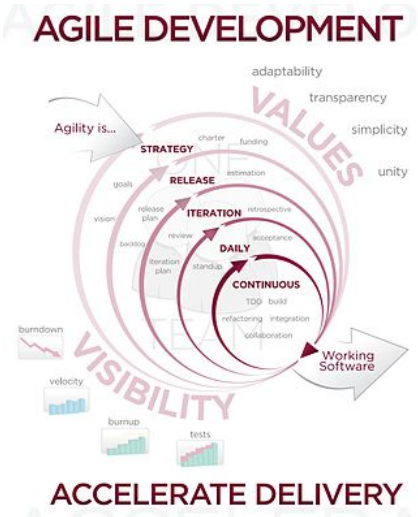
Develop model of requirements — *Add detail and design decisions* — *Develop code*

User's Perspective — **Developer's Perspective**

Agile Method

- **Agile Method (also called Agile software development)** is a group of software development methodologies based on iterative and incremental development
- Requirements and Solutions evolve through collaboration between self-organizing, cross-functional teams.

Agile Method



Agile Method

- 敏捷式的精神
  - 原則上敏捷式開發主要的精神在於較短的開發循環（建立在反覆式開發方式上）以及漸進式開發與交付。換句話來說，專案的成果，包含計畫、各類的需求細節、設計等都會隨著專案的進行而漸漸完整，而非在一開始將所有的計畫與需求擬定完成。
  - 在敏捷式開發中有個很重要的觀點是塑模（Modeling）的目的在於增加開發者了解軟體的程度，進而使得軟體更接近於使用者的需求，而非使用塑模之後產生的文件。

Agile Method

- 草稿與藍圖
  - 開發者使用塑模的時機，是當使用這個技術有助於開發者更了解被開發的軟體時才使用，例如某些具關鍵性的議題或者高風險性的項目，而非將軟體所有範圍的設計都加以塑模留下文件。
  - 塑模在敏捷式開發的精神下是一種類似草圖或者草稿的作用。也就是說，用以在團隊開發時討論以及研究議題的一種工具，在過程中利用塑模的技術來讓問題得到解決，一開始的動機並非謂了留下設計圖讓程式設計師去實作。

Agile Method

- **Agile Alliance**
  - 2001年支持敏捷式開發的社群組成了Agile Alliance(http://www.agilealliance.com/)，並且發表了Agile宣言與原則。
- The Agile Manifesto (敏捷宣言)
  - 獨立的工作成員與人員互動 勝於 流程與工具的管理
  - 工作產生的軟體 勝於 廣泛而全面的文件
  - 客戶的合作 勝於 契約的談判
  - 回應變動 勝於 遵循計畫

Agile Method

- The Agile Principles (敏捷原則)
  - 最為優先的事情是透過早期與持續交付有價值的軟體來使客戶滿意。
  - 歡迎需求的變動，即使是在開發的晚期。敏捷式流程駕馭變動來作為客戶的競爭優勢。
  - 頻繁的交付工作產生的軟體，自數週至數月，週期越短越好。
  - 領域專家與開發成員必須一同作業，並貫穿整個專案開發時期。

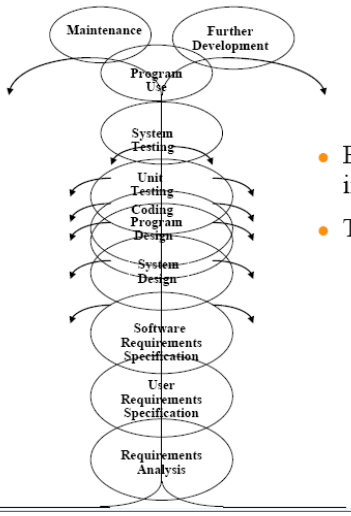
Agile Method

- The Agile Principles (敏捷原則)
  - 使用積極的工作成員來建構專案，給予他們環境以及支援所需的一切，然後信任他們能夠完成工作。
  - 在開發團隊中最快也最有效的傳遞資訊方法就是面對面的溝通。
  - 工作產生的軟體是衡量進度最主要的依據。
  - 敏捷式流程倡導水平一致的軟體開發
  - 專案發起者，開發人員以及使用者都必須持續的維持專案進度。

Agile Method

- The Agile Principles (敏捷原則)
  - 持續重視技術的優勢以及設計品質
  - 最好的架構、需求以及設計會出現在能夠自我管理的團隊裡
  - 在規律的反覆之間，團隊會反省與思考如何更有效率，然後相對的來調整與修正團隊的開發方式。
- 參考網址—苗圃-台灣敏捷方法Agile Method
  - http://140.115.155.41/

Top-Down vs. Bottom-Up



- Bottom-up: develop an individual class
- Top-down analysis

Comparing Various Models

- Waterfall model problems
- Prototyping
- Language Comparison

Waterfall model problems

- Traceability/languages in different phases
- Process is too linear
  - Requirement acquisition and validation
- Maintainability : due to the use of functional decomposition

Waterfall model problems

- Assume fully elaborated documentation at the early stage of the life cycle.
  - Reusability : top-down design
  - Communication
- Based on functional decomposition
  - Strongly dependent on detailed functional breakdown
  - Not consider evolutionary changes.
  - Not encourage reusability

Prototyping

- Benefits
  - Improve communication
  - Reduce risk
    - Communication between developments
    - Determine a proposed design's unknown properties
    - Address requirement acquisition and validation limitation
    - Provide a basis for assessing the feasibility and performance of alternative designs
    - Most feasible way to validate specification.
    - For maintenance as well

Prototyping

- Limitation
  - Quick and direct approach without considering issues such as quality and maintainability.
  - partial implementation

Language Comparison

Verify correctness and completeness of design or implementation

Interconnections during architecture and module design

	executable	supporting formal reasoning	containing quantifies	detail algorithm and data structure
prototyping lang.	yes	low priority	no	no
specification language	not necessary	must	yes	no
Design language	not			no
programming language	efficient			

Language Comparison

- Main Features of Languages
  - Specification language: abstract of system functionality
  - Design language: abstract of system structure
  - Prototype language: both specification and design
  - Programming language: optimization

Requirements Engineering

- Requirements engineering
- Requirements analysis
- Object-oriented (OO) software engineering
- Data modeling and OOA

Requirements Engineering

- Software requirements
- Characteristics of requirements
- Requirements engineering
- Requirements elicitation

Requirements Engineering

- Requirement
  - Functional requirement describes system services or functions
  - Non-functional requirement is a constraint or a goal on the system or on the development process
- User (Customer) requirement
  - A statement in natural language plus diagrams of the services the system provides and its operational constraints
- Requirements specification
  - A structured document for detail description of the system services
  - Written as a contract between client and developer

Characteristics of Requirements

- Incomplete Requirements
  - Most software systems are complex, that developer can never fully captured during the system development, therefore, requirements are always incomplete.
- Inconsistent Requirement
  - Different users have different requirements and priorities. There is a constantly shifting compromise in the requirements.
  - Prototyping is often required to clarify requirements.

Requirements Engineering

- Requirements elicitation
  - Determine what the customer requires
- Requirements analysis
  - Understand the relationships among various customer requirements
- Requirements negotiation
  - Shape the relationships among various customer requirements to achieve a successful result
  - Research on requirements trade-off analysis (formulating as goals)
- Requirements specification
  - Build a form of requirements

Requirements Engineering

- Software modeling
  - Build a representation of requirements that can be assessed for correctness, completeness and consistency.
- Requirements validation
  - Review the model
- Requirements management
  - –Identify, control and track requirements and the changes

Requirements Elicitation

- Two sources of information for the requirements elicitation process
  - User (customer)
  - Application domain
- Asking
  - Ask users what they expect from the system
  - Interview, brainstorm and questionnaire
- Task analysis
  - High-level tasks can be decomposed into sub-tasks
- Scenario-based analysis
  - Study instances of tasks
  - A scenario can be real or artificial

Requirements Elicitation

- Form analysis
  - A lot of information about the domain can be found in various forms (examples in ERD slides)
  - Forms provide us with information about the data objects of the domain, their properties, and their interrelations
- Natural language description
  - –with background information to be used in conjunction with other elicitation techniques such as interviews
- Derivation from an existing system
  - Take the peculiar circumstances of the present situation into account (examples in ERD slides)
- Prototyping

Requirements Analysis

- Software modeling
- The analysis process
- Entity-Relationship diagram (ERD)
- Extended entity-relationship diagram (EERD)
- Components of structured analysis

TSEG Tamkang Universty Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Requirements Analysis

- Information domain analysis
  - information flow: data transformation
  - data content: data dictionary
  - data modeling
- Functional and behavioral representation
  - function: process transformation
  - behavior: state transition diagram
- Interfaces definition
  - function/process interface
- Problem partition and abstraction
  - –at different levels of abstraction
  - –classification and assembly structure

<inhon@mail.tku.edu.tw> March 18, 2012

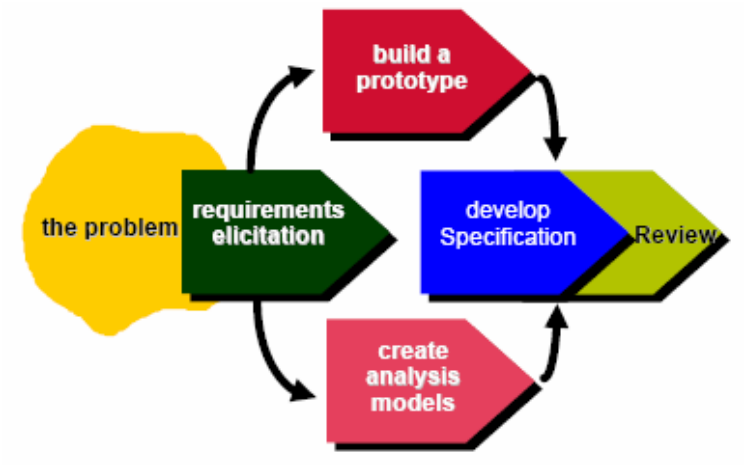
Software Modeling

- Purpose
  - focus on those qualities of an entity that are relevant to the solution of an application problem
  - abstract away those that are irrelevant
- Model: an abstraction for
  - Understanding before (actually) building
  - Communication
  - Visualization
  - Reducing complexity
- Methodology: build (analyze) a model of an application domain

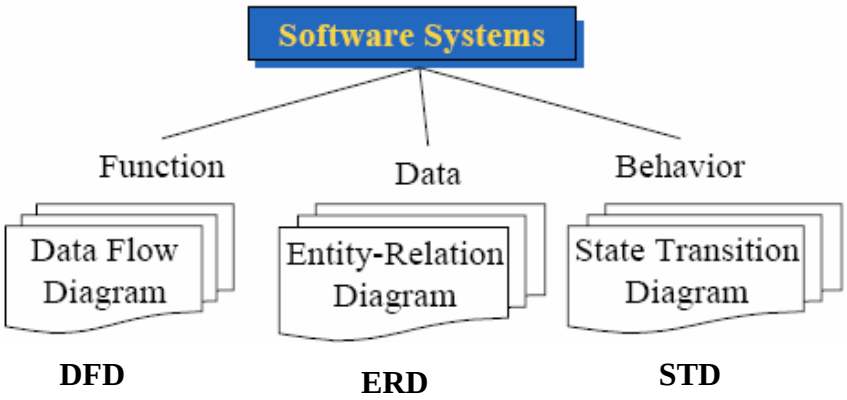
Application and Solution Domain

- Application domain (requirements analysis)
  - The environment in which the system is operating
- Solution domain (system design, object design)
  - The available technologies to build the system

The Analysis Processes

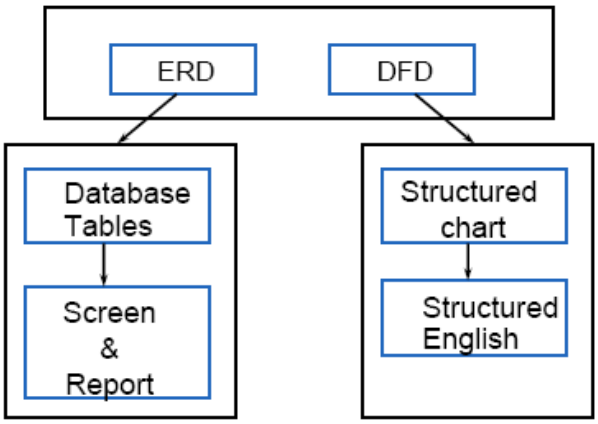


Traditional Software Engineering



Traditional Software Engineering

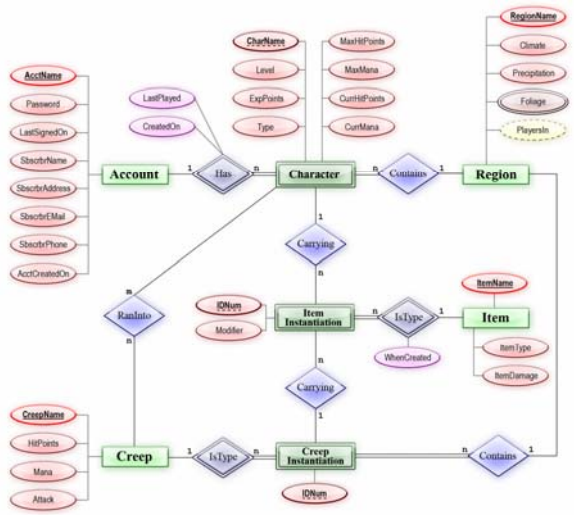
- From Analysis to Design



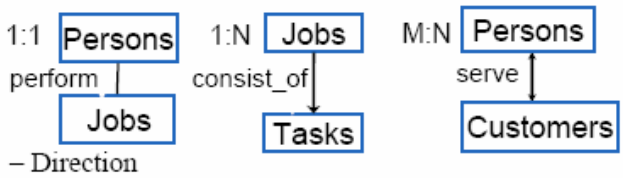
Entity-Relationship Diagram

- Entity
  - Primary things an organization collects and records information about. (noun) (▭)
    - E.g. persons, products, places, etc.
- Relationship
  - Linkage between entities. (verb) (—)
    - E.g. Persons perform jobs, Jobs consist-of tasks
- Cardinality
  - Identify how many instances of one entity are related to how many instances of another entity.

Overview



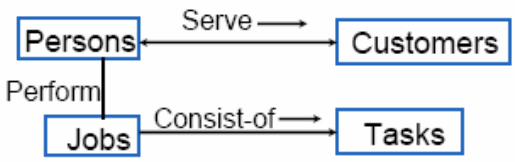
Entity-Relationship Diagram



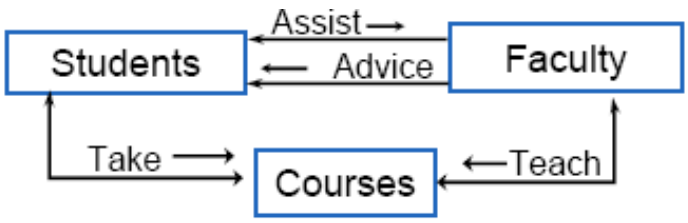
– Direction

- using an arrow pointing to the object of the action.

Examples

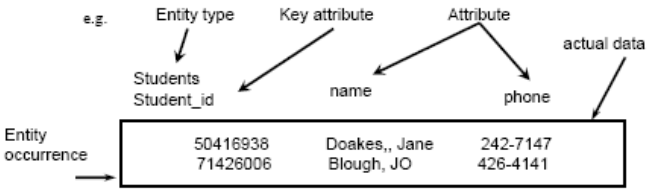


Entity-Relationship Diagram



- Attributes: Properties to describe an entity
  - key attribute (key, identifier) to characterize the specific entity (to retrieve a single entity occurrence (instance))
    - **unique**: to ensure that no other record has the same identifier.
    - **unchanging**: to ensure that it always refers to the same thing.

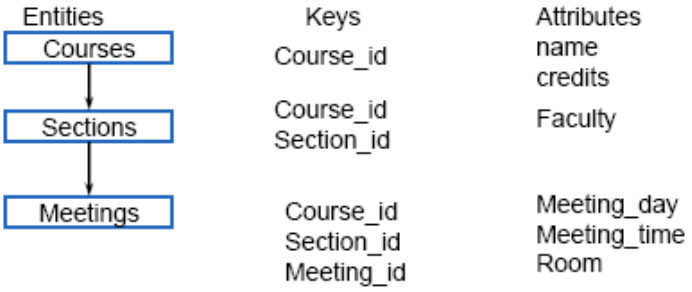
Entity-Relationship Diagram



- Student is an entity about which a university stores info such as the Student_id, name, and phone.
- Compound keys: made up of a number of different subkeys to produce a unique identifier
  - e.g. course number + section number + term
- The difference between an entity and an attribute is that attributes are atomic.
  - i.e. Attributes have no further attributes that describe them. Entities can be further described by their attributes.

Advance Features

- Kernel and characteristic entities
  - Entities can be described by other subsidiary entities in a hierarchical fashion.
    - to store related values of one of the attributes of an entity



Advance Features

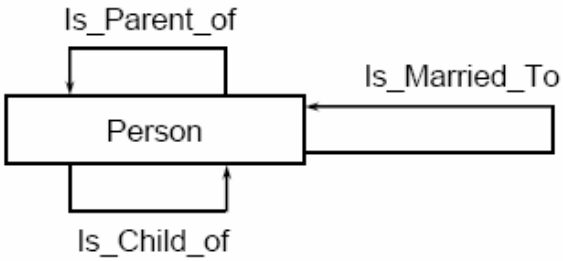
- The highest entity type in the hierarchy is called a kernel entity, which has a unique identity that does not depend on the existence of any other entity type.
- Characteristic entities: to record the repeated characteristics of the kernel entity.
  - e.g. Course is a kernel entity, Sections and Meetings are characteristic entities describing the characteristics of Courses.
  - The unique identifier for the characteristic entities is a multiple key.
  - e.g. Course_id + Section_id are needed to uniquely identify a section.

Advance Features

- The highest entity type in the hierarchy is called a kernel entity, which has a unique identity that does not depend on the existence of any other entity type.
- Characteristic entities: to record the repeated characteristics of the kernel entity.
  - e.g. Course is a kernel entity, Sections and Meetings are characteristic entities describing the characteristics of Courses.
  - The unique identifier for the characteristic entities is a multiple key.
  - e.g. Course_id + Section_id are needed to uniquely identify a section.

Advance Features

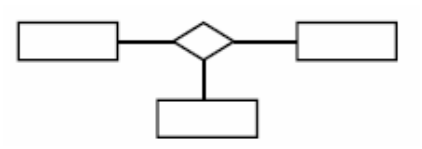
- Recursive relationships: an entity is related to itself
  - E.g.



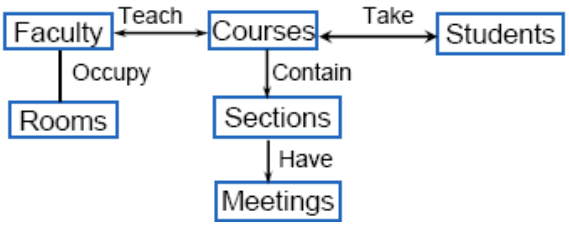
  - for retrieving all family relationships

Advance Features

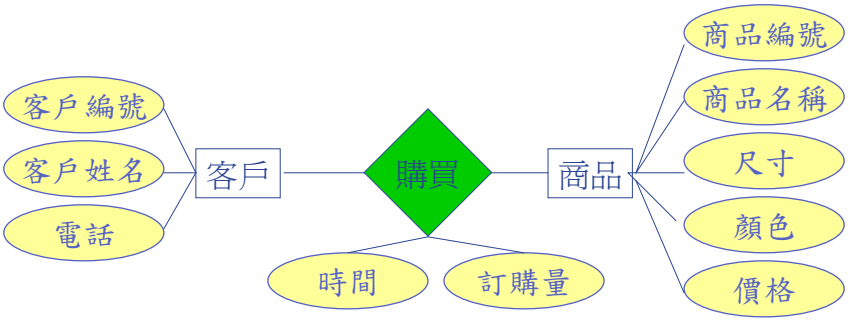
- N-ary Relationships
  - E.g.



  - For Example

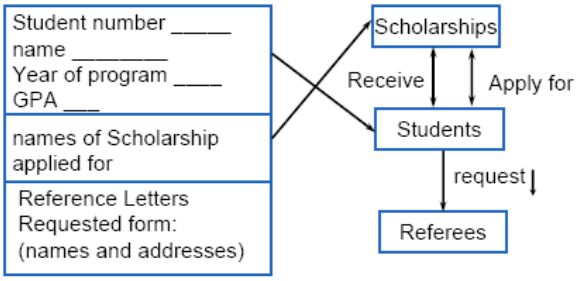


Example



Where to look for Information

- Existing forms
  - forms organize the data and remind what to collect.
  - It is common in manual systems to provide large amounts of redundant data.
  - e.g. Scholarship Application Form



Where to look for Information

- Existing file structures
  - Frequently organizations have a collection of application programs that do not link to each other. They may require complex programs to transform data used by one application into a form used by another one.
  - e.g. existing student record system

Accounts: Account_id, Label
Buildings: Building_id, Name, Address.
Courses: Course_id, Course_Name, Credits.
Course_Program: Course_id, Program_id
Departments: Department_id, Department_Name.
Enrolled: Student_id, Course_id, Section_id, Year, Term, Grade.
Faculty: Faculty_id, Name, Address, Birthday.
Fee_Payments: Student_id, Account_id.
Prerequisites: Course_id, Pre1, Pre2, Pre3.
Programs: Program_id, Program_Name.
Rooms: Building_id, Room_id, Size, Type.
Sections: Course_id, Section_id, Year, Term, Faculty_id (Meeting_id, Building_id, Room_id, Day, Time) ...
Students: Student_id, Name, Address, Birthday.

Where to look for Information

- Kernel entities: single keys, such as: Accounts, Buildings, Courses, Departments, Faculty, Prerequisites, Programs, and Students
- Characteristic entity vs. Relationship (rules)
  - **kernel entity** (or characteristic entity)'s key + not part of the key of any entity

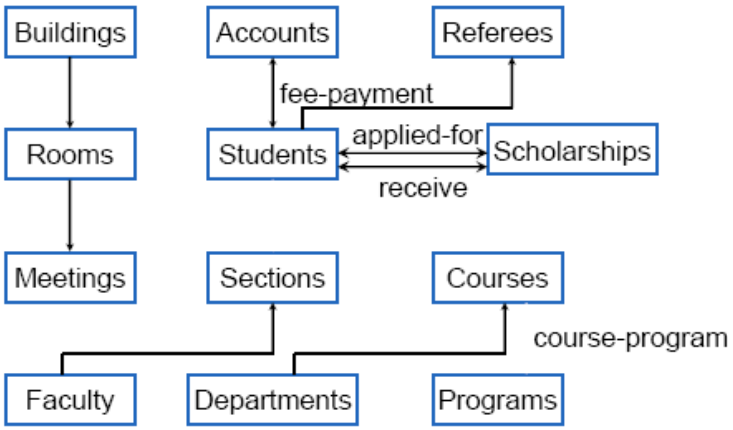
  => characteristic entity
  - **multiple keys** are combinations of keys for other entities => M:N relationships
  - e.g. Course_program: course_id, program_id...

    Enrolled: students_id, course_id,...
  - e.g. Rooms: building_id <----from building's

    room_id <---not identified precisely

    characteristic entity

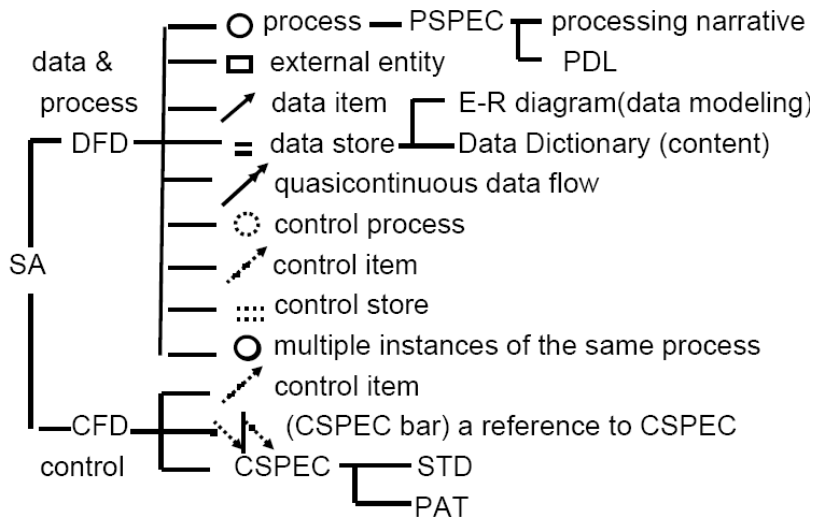
ERD from existing Forms and Files



Testing ERD

- No identification key for entity
- Two or more entities have the same key
- Many relationships to a single entity
- Two or more relationships between the same entities
- N-ary relationship
- An entity has no relationship

Components of Structured Analysis (SA)



Components of Structured Analysis

- Modeling Technique
  - model: describe information (data & control), flow, content.
  - control-oriented applications
  - data-intensive applications

Deficiency

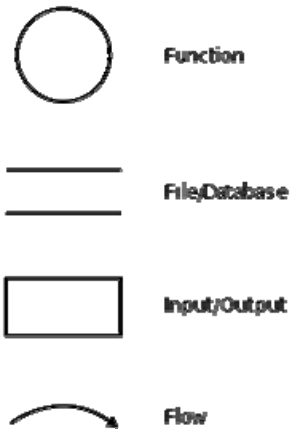
Data Flow Diagram (DFD)

- A **data flow diagram** (**DFD**) is a graphical representation of the "flow" of data through an information system. DFDs can also be used for the visualization of data processing (structured design)
- On a DFD, data items flow from an external data source or an internal data store to an internal data store or an external data sink, via an internal process.

Data Flow Diagram (DFD)

- A DFD provides no information about the timing of processes, or about whether processes will operate in sequence or in parallel. It is therefore quite different from a flowchart, which shows the flow of control through an algorithm, allowing a reader to determine what operations will be performed,
- but not what kinds of data will be input to and output from the system, nor where the data will come from and go to, nor where the data will be stored (all of which are shown on a DFD).

Data Flow Diagram (DFD)



Basic Notations of DFD

Data Flow Diagram (DFD)

- DFD can be used to represent a system at any level of abstraction. (refine)
  - –level 0: context model (a single bubble)
  - –information flow continuity: I/O to each refinement must remain the same. (balancing)
- No explicit indication of the sequence of processing is supplied by the DFD.
  - –Explicit procedural representation delayed until design.

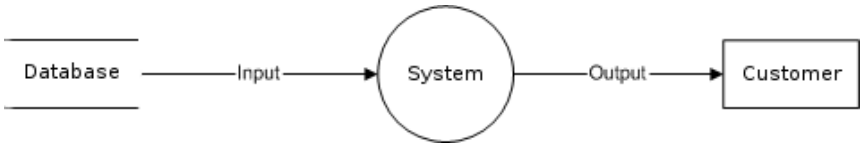
Data Flow Diagram (DFD)

- Content of data (implied by the arrow or described by the store)
  - –a collection of items: using data dictionary. (DD) (only content)
  - –a need to represent the relationship between complex collections of data. (E-R diagram for data modeling)

Data Flow Diagram (DFD)

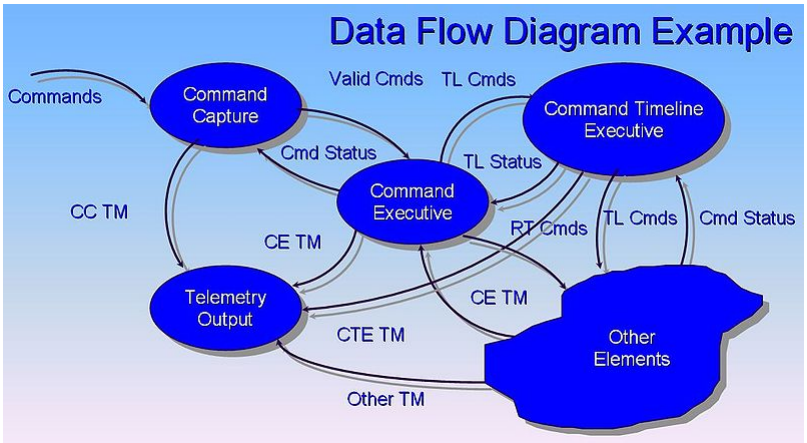
- Processing narrative: describe (usually natural language) a process bubble.
  - To specify the processing details in the bubble.
    - inputs to the bubble
    - algorithm applied to the input
    - Output
    - Restrictions & limitations imposed on the process.
    - Performance characteristics related to the process.
    - Design constraints

Data Flow Diagram (DFD)



Examples- Level 0

Data Flow Diagram (DFD)

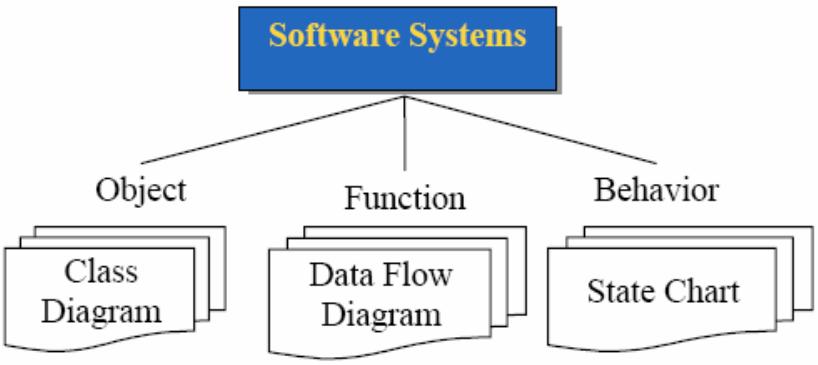


Examples-Detail level

Object-oriented (OO) Software Engineering

- Steps of analysis: an example using OO approach
- Concepts and phenomena
- Class
- Class identification
- Pieces of an object model

Object-oriented (OO) Software Engineering

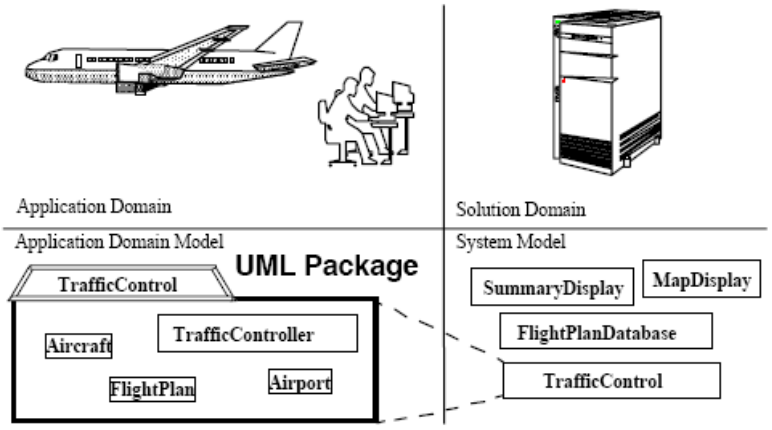


Steps of analysis

- Define use cases
- Extract candidate classes
- Establish basic class relationships
- Define a class hierarchy
- Identify attributes for each class
- Specify methods that service the attributes
- Indicate how classes/objects are related
- Build a behavioral model

Steps of analysis

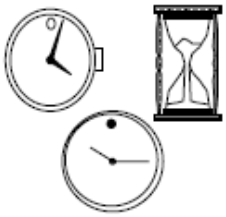
- Application and Solution Domain



Concepts and Phenomena

- *Phenomenon (object)*: An object instance in the world of a domain,
  - E.g. My black watch
- *Concept (object class)*: Describes the properties of phenomena that are common,
  - E.g. Black watches
- A concept is a 3-tuple:
  - Its *Name* distinguishes it from other concepts.
  - Its *Purpose* are the properties that determine if a phenomenon is a member of a concept.
  - Its *Members* are the phenomena which are part of the concept.

Concepts and Phenomena

Name	Purpose	Members
Clock	A device that measures time.	

- Modeling: Development of abstractions to answer specific questions about a set of phenomena while ignoring irrelevant details.

Classes

- Class:
  - An abstraction in the context
  - encapsulates both state (variables) and behavior (methods)
  - Can be defined in terms of other classes using inheritance
- Criteria of selecting classes
  - Retained information
  - Needed services
  - Multiple attributes
  - Common attributes
  - Essential requirements

Classes Identification

- Identify the boundaries of the system
  - What object is inside, what object is outside?
- Identify the important entities in the system
  - Learn about problem domain: Observe your client
  - Take the flow of events and find participating objects in use cases (Scenarios and use cases)
  - Apply design patterns
  - Nouns are good candidates for classes

TSEG Tamkang Universty Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Classes Identification

- Identify the boundaries of the system
  - What object is inside, what object is outside?
- Identify the important entities in the system
  - Learn about problem domain: Observe your client
  - Take the flow of events and find participating objects in use cases (Scenarios and use cases)
  - Apply design patterns
  - Nouns are good candidates for classes

<inhon@mail.tku.edu.tw> March 18, 2012

Pieces of an Object Model

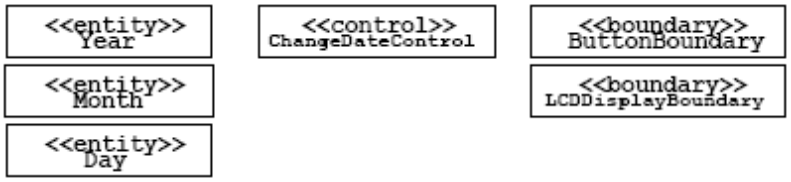
- Classes
- Associations (Relations)
  - Part of-Hierarchy (Aggregation)
  - Kind of-Hierarchy (Generalization)
- Attributes
  - Application specific
  - Attributes in one subsystem can be classes in another subsystem, turning attributes to classes

Pieces of an Object Model

- Service
  - ***Domain Methods***: Dynamic model, Functional model
  - ***Operation*:** A function or transformation applied to objects in a class. All objects in a class share the same operations *(Analysis Phase)*
  - ***Signature:*** Number & types of arguments, type of result value. All methods of a class have the same signature *(Object Design Phase)*
  - ***Method*:** Implementation of an operation for a class *(Implementation Phase), Polymorphic operation*: The same operation applies to many different classes.

Object Types

- Entity Objects: represent the persistent information (Application domain objects, “Business objects”)
- Boundary Objects: represent the interaction between the user and the system
- Control Objects: represent the control tasks performed by the system



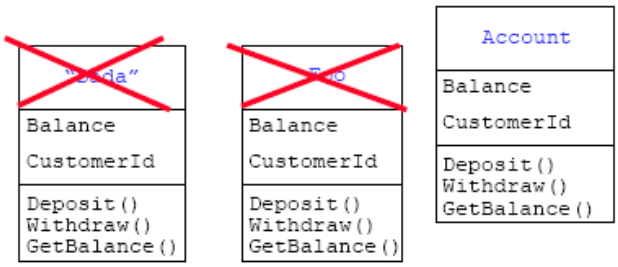
Model Behavior

- Indicate different states of the system
- Specify events that cause the system to change state

Modeling Example: A Banking System

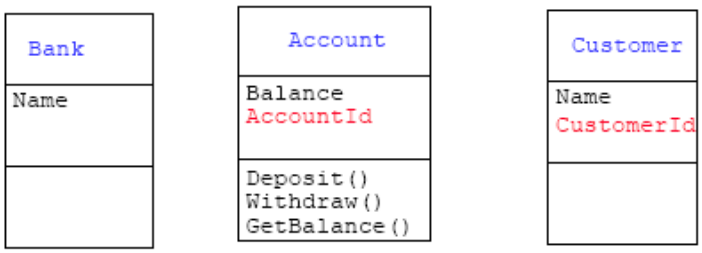
- Class Identification: Name of Class, Attributes and Methods

Modeling Example: A Banking System



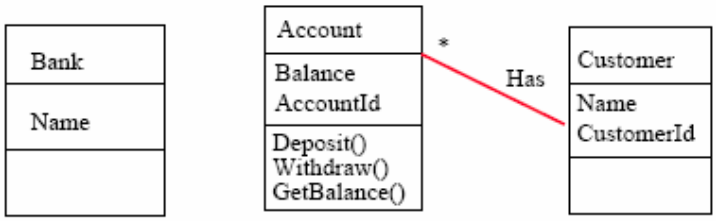
- Naming

Modeling Example: A Banking System



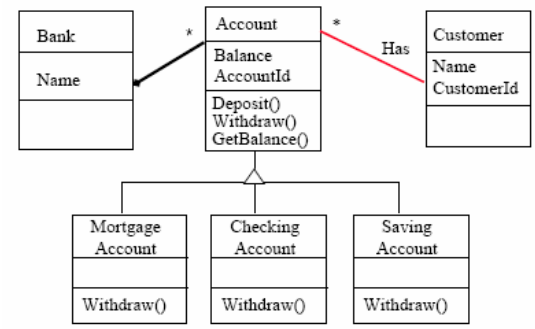
- Finding New Objects
  - Iterate on Names, Attributes and Methods

Modeling Example: A Banking System



- Finding New Objects
  - Iterate on Names, Attributes and Methods
  - Find Associations between Objects
  - Label the associations
  - Determine the multiplicity of the associations

Modeling Example: A Banking System



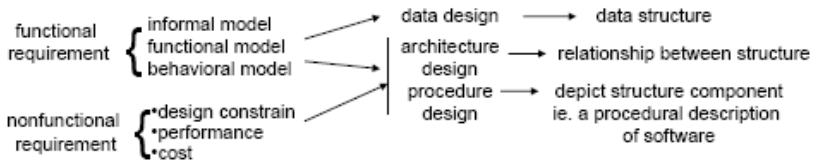
- Categorize
  - Don't put too many classes into the same package: 7 ± 2 (or even 5 ± 2)

Software Design

- Design Fundamentals
- Effective Modular Design
- Architecture Design
- Data Design
- Procedural Design

Design Fundamentals

- Design Step Procedures



- Software Design vs. Requirement Analysis
  - –Software design: requirement →a representation of software
  - –requirement analysis: create a model to represent to represent the requirements

Design Fundamentals (Cont.)

- Common Characteristics
  - A mechanism for the translation of information domain representation into design representation
  - A notation for representing functional components and their interfaces
  - Heuristics for refinement and partitioning

Design Fundamentals (Cont.)

- Fundamental Concepts
  - Abstraction

Level of abstraction	Language used
Highest level	Program-oriented terminology
Low level	Procedural-oriented terminology
Lowest level	Implementation-oriented terminology

  - Procedural abstraction
    - a named sequence of instruction that has a specific function
  - Data abstraction
    - a named collection of data–that describes a data object
    - can refer all the data by stating the name of the data abstraction
    - original abstraction data type is used as a template or generic data structure from which the data structure can be instructed.

Fundamental Concepts

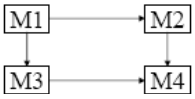
- Refinement
  - Top-down design strategy
  - A hierarchy is developed by decomposing a statement of function (a procedural abstraction) in a stepwise fashion until programming statements are reached
    - Every refinement step implies some design decisions
  - A process elaboration
    - Statement of function (description of information) without the internal working of the function (internal structure of the information)
    - Providing more and more details as each successful refinement occurs

Fundamental Concepts (Cont.)

- Modularity
  - Modules: software is divided into separately named and addressable components
  - Modules vs. interfaces between modules
    - Avoid over modularity & under modularity; notice that the relationship between modules (that is, the number of interfaces increase exponentially with the number of modules)
- Software architecture
  - Hierarchy structure of procedural components
  - Structure of data
  - It relates elements of a software solution to parts of a real-world problem

Fundamental Concepts (Cont.)

- Control hierarchy (program structure)
  - The organization of program components & implies a hierarchy of control
    - Does not represent procedural aspects of software (iteration, condition, sequence)
  - Depth / width / fan-in / fan-out / superordinate/ subordinate
  - Two characteristic of software architecture
    - Visibility: program components may be invoked or used as data by given component (indirectly) (e.g. M4 is visible to M1)
    - Connectivity: program components are directly invoked or used as data by given component (e.g. M2 is connected to M1)
      - A module that directly causes another module to begin execution is connected to it



Fundamental Concepts (Cont.)

- Software procedure focuses on the processing details of each module individually (sequence, iteration, condition)
  - a procedural representation of software is layered
- Information hiding
  - modules should be specified and designed so that information (procedure & data) contained within a module are inaccessible to other modules that have no need for such information.
  - abstraction defines the procedural entities
  - hiding defines access constraints to both procedural detail within a module and local data structure used by the modules

Effective Modular Design

- Effective modular design
  - A modular design reduces complexity, facilitates changes, results in an easier implementation.

monolithic program — module program — object-oriented program

code-and-fix

- Classification (based on temporal aspect)

sequential	/ incremental	/ parallel
subroutines	/ coroutines	/ conroutines
Without apparent interruption	/ Can be interrupted before completion	/ execute simultaneously with another module

Effective Modular Design (Cont.)

- A typical control hierarchy may not be encountered when coroutinesor conroutines are used
- Functional independence (independent effective modules)
  - Effective modularity: function compartmentalized and interfaces simplified
- Cohesion: relative functional strength of a module

low ↓ high

- coincidental : tasks relate to each other loosely
- logical : performing tasks related logically (all output)
- temporal : all tasks executed with the same span of time
- procedural : must be executed in a specific order
- communicational : concentrate on one area of a data structure
- Sequential
- Functional

Effective Modular Design (Cont.)

- Coupling: interconnection among modules
  - depends on the interface complexity between modules, entity point or reference to a module, what data passes across the interface

low ↓ middle ↓ high

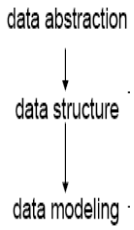
- No direct coupling : module subordinate to different modules
- Data coupling : data passed via argument list
- Stamp coupling : data structure passed via argument list
- control coupling : passes control data
- external coupling : modules are tied to an external environment
- common coupling : refer to a global data area
- content coupling :
  - one module makes use of data or control information maintained within the boundary of another module .
  - branches are made into the middle of a modular

Design Step Procedures

- Data Design
- Architecture Design
- Procedural Design

Data Design

- Define data abstractions
- Select an appropriate data structure to implement the abstraction
- Using entity-relationship modeling depict relationship between objects



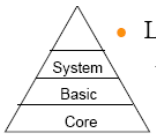
Architectural Design

- Program architecture, domain specific software architecture
  - Develop a modular program structure
  - Represent the control relationship between modules
    - Control hierarchy connecting modules
  - Define interface that enable data to flow throughout the program
  - System organization
  - DSSA
    - Pipes & filters
      - I: a stream of inputs
      - O: a stream of outputs

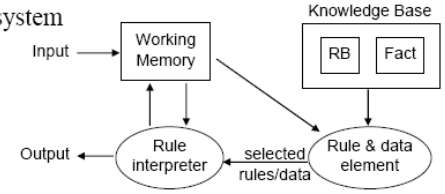
Program Architecture

Domain-Specific Software Architecture

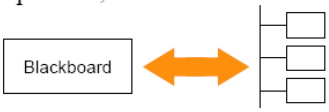
Architectural Design



- Layered systems:
  - System organized hierarchically with each layer providing service to the layer about it
- Rule-based system



- Blackboard systems: a central data structure represents the current state of the computation, a collection of knowledge sources.



Architectural Design

- Structure Chart
  - A **Structure Chart** (SC) in software engineering and organizational theory is a chart, which shows the breakdown of the configuration system to the lowest manageable levels.
  - This chart is used in structured programming to arrange the program modules in a tree structure. Each module is represented by a box, which contains the module's name. The tree structure visualizes the relationships between the modules.

TSEG Tamkang University Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

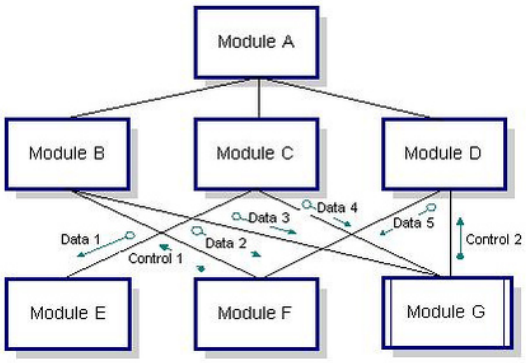
Architectural Design

- Structure Chart
  - A structure chart is a top-down modular design tool, constructed of squares representing the different modules in the system, and lines that connect them. The lines represent the connection and or ownership between activities and subactivities as they are used in organization charts
  - A structure chart is also used to diagram associated elements that comprise a run stream or thread. It is often developed as a , but other representations are allowable. The representation must describe the breakdown of the configuration system into subsystems and the lowest manageable level.

Architectural Design

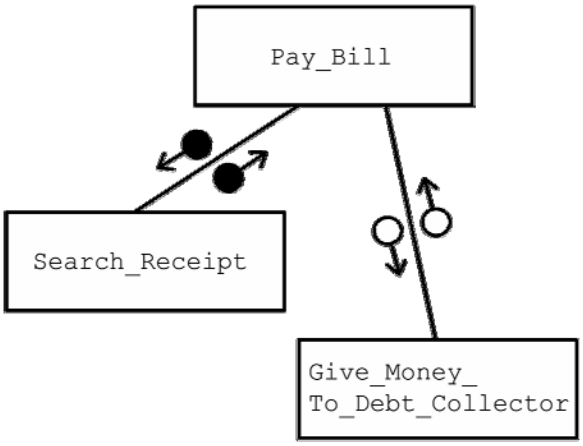
- Structure Chart: Example

CROSSING LINES ON A STRUCTURE CHART



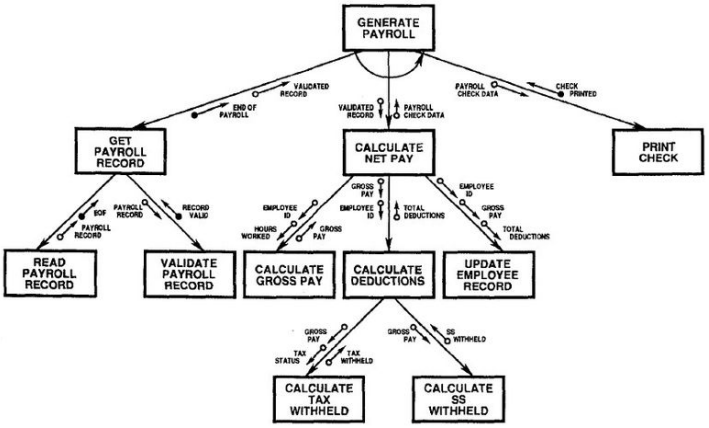
Architectural Design

- Structure Chart: Example



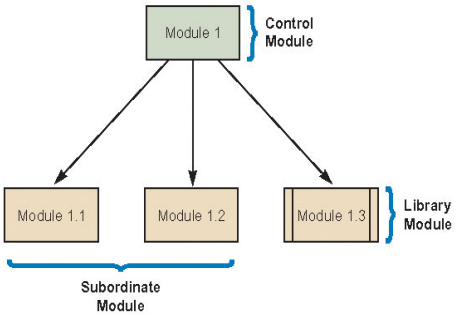
Architectural Design

- Structure Chart: Example



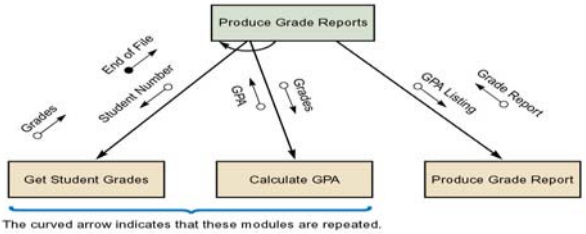
Architectural Design

- Structure Chart: 符號說明
  - 結構圖 (structure chart)
    - 模組
      - 控制模組(Control Module)
      - 程式館模組 (Library Module)
    - 資料關聯
    - 控制關聯 (Control Couple)
      - 旗號 (flag)
      - 一個模組利用旗號來傳送特定情況或動作的訊號給另一個模組



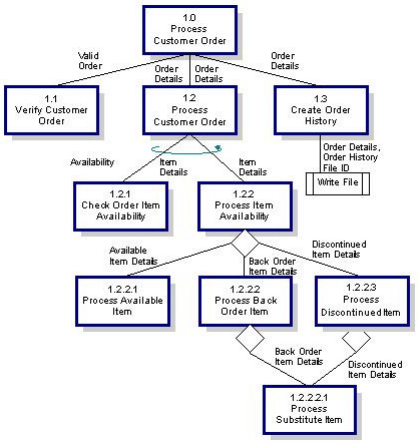
Architectural Design

- Structure Chart: 符號說明
  - 條件
    - 一個條件 (condition) 線段表示出按照特定的條件一個控制模組可以決定該呼叫那一個附屬模組
  - 迴圈
    - 迴圈 (loop) 指出一個或多個模組的重複執行



Architectural Design

- Structure Chart: Another Example



Procedural Design

- Structure programming
  - logical constructs: sequence, conditional, iteration
  - to limit the procedural design to a small number of predictable operations
    - reduce program complexity
    - lead to more readable, testable code
- Flow charts
  - Graphical representation for procedural design
  - Limitation
    - Introduce inefficiency when an escape from a set of nested loops or nested conditions is required
    - Additional complications of logical tests

Procedural Design

- Program design language (PDL)
  - Structured English (pseudocode)
    - Uses the vocabulary of English
    - Overall syntax of a structure programming language
  - Difference between PDL and programming language
    - The use of narrative text embedded directly within PDL statements
    - PDL cannot be compiled but can be translated into a graphical structure
- Why design language
  - can be embedded with source code
  - can be a derivative of the high order language .eg. Ada PDL
  - easy to review
  - can be represented in great detail

PDL

- Data structure
  - TYPE <var-name> IS <q-1> <q-2>
    - eg. TYPE table IS INSTANCE OF symboltable
- Abstract data type: data abstraction
  - Generic data structure (template) from which other data structures can be instantiated
  - e.g. TYPE table IS INSTANCE OF symboltable
  - Block structure

```
BEGIN
        < ..... >
END
```


PDL

- Conditional

```
-IF <      >
   THEN <      >
   ELSE <      >
 ENDIF
```

```
-CASE OF <      >
   WHEN <      > SELECT <      >
      .
      .
   DEFAULT: <      >
 ENDCASE
```

- Iteration

```
-REPEAT UNTIL <      >
      <      >
 ENDREP
-DO WHILE <      >
      <      >
 ENDDO
-DO FOR <      >
      <      >
 ENDFOR
```

Object-Oriented Software Development

- Object orientation
- Object-oriented analysis
- Object-oriented Design
- Design Pattern

TSEG Tamkang Universty Software Engineering Group 淡江軟體工程實驗室 http://www.tkse.tku.edu.tw/

Object-Oriented Software Development

- Object orientation
- Object-oriented analysis
- Object-oriented Design
- Design Pattern

<inhon@mail.tku.edu.tw> March 18, 2012

Object orientation

- Real-world objects vs. Software objects
  - communications

Problem domain → Solution domain

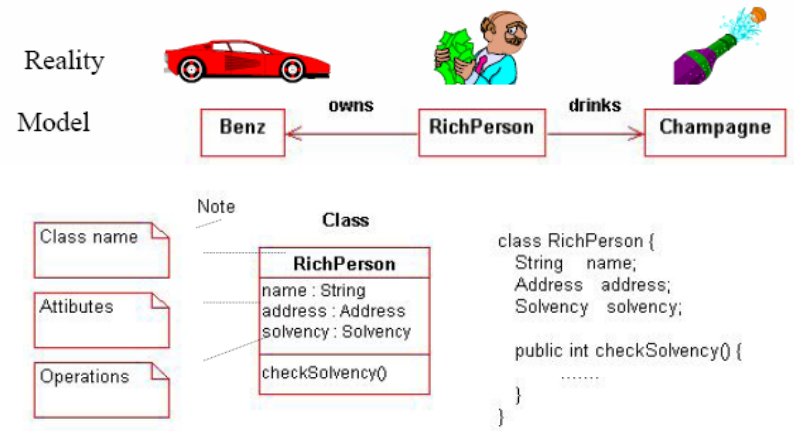
software realization of the real-world problem

Object necessary for describing a solution - problem space

Object required for implementation a solution - solution space

Object orientation

- Modeling the reality of the world.



Object orientation

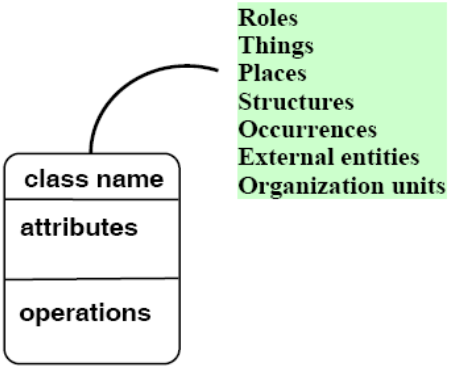
- The essential differences between structured and object-oriented methods:
  - ***All objects to work on***. The class concept is used to work with units of data and operations
  - ***Better possibilities of abstraction***. The reality of the world is visually modeled.
  - ***Methodological uniformity***. The results of an activity *i* can easily be taken over into activity *i+1*, that is, iterative development and reverse engineering are more easier to be carried out.
  - ***Evolutionary development***. A complex system is not built in one go.

Object orientation

- Class
  - Object-oriented thinking begins with class
    - template
    - generalized description
    - pattern
    - "blueprint": describing a collection of similar items
  - A meta-class (also called a superclass) is a collection of classes
    - Once a class of items is defined, a specific instance of the class can be defined

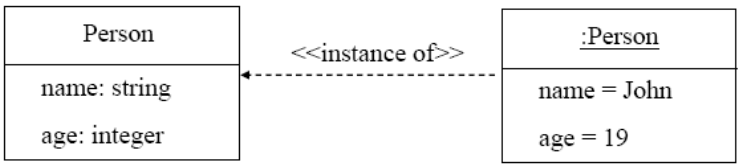
Object orientation

- Class



Object orientation

- Class diagrams
  - describes classes
- Object diagrams
  - describes instances

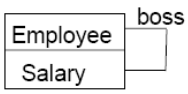


Object orientation

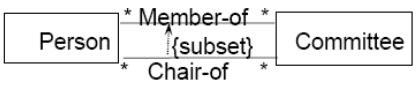
- Attributes
  - a data value held by the objects.
  - e.g. name, age, weight, etc.
  - attributes name may be followed by type or default value.
- Operations/methods/service
  - a function that may be applied to or by objects in a class (i.e. perform or suffer)
  - the same operations takes on different forms in different classes: polymorphism.
  - the implementation of an operation is called a method. (by a different piece of code)

Object orientation

- A ***constraint*** restricts the values that entities can assume, which is defined as functional relationships between entities of an object model.
  - entity : objects, classes, attributes, links, and associations.
  - e.g. No employee's salary can exceed the salary of the employee's boss

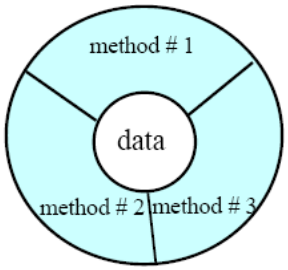


{ Boss Salary ≥ Salary }



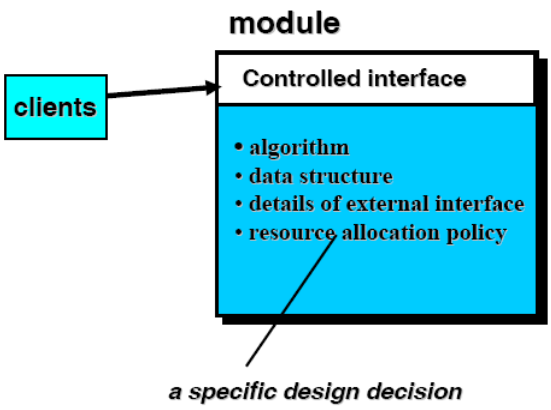
Object orientation

- **Encapsulation/Information Hiding**
  - Separate the external aspects of an object from the internal implementation details of the object, which are hidden from other objects
  - The object encapsulates both data and the logical procedures required to manipulate the data
  - The advantage
    - The user of the data does not need to know how, where, or in what form



Object orientation

- **Encapsulation/Information Hiding**

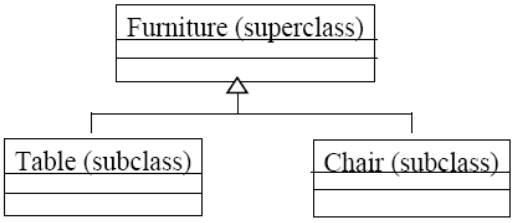


Object orientation

- **Object Identity**
  - Explicit object identifiers are not required in an object model.
  - Each object is a discrete and distinguishable entity.
  - Each real-world object is unique due to its existence.
  - Two objects are distinct even if all their attribute values are identical.
  - Object-oriented Database are often based on system-generated ID numbers and corresponding pointers.

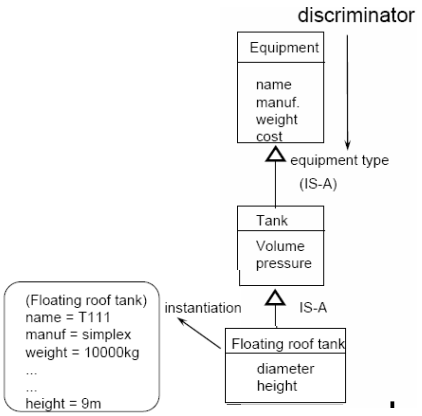
Object orientation

- **Inheritance:**
  - The sharing of attributes and operations among classes based on a hierarchical relationship
  - Each subclass inherits all of the properties of its super class and adds its own unique properties (called extension)
  - For reusability (black box reuse)



Object orientation

- **Classification:**
  - objects with the same attributes and operations are grouped into a class
  - each object is said to be an instance of its class
  - e.g. Bicycle object -------> Bicycle class

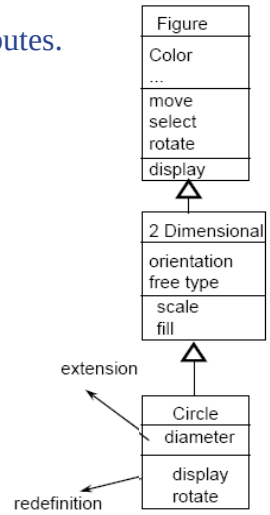


Object orientation

- Generalization : is-a relationship
  - (superclass, subclass)
- Each subclass can inherit all the features of its ancestors and add its own specific attributes and operations.

Object orientation

- Extension: add a new operation and attributes.
- Redefinition (Overriding)
  - –retain the original interface
  - –but recode a particular method
- Restriction :
  - –inherit a subset of operations / or tighten
  - –the type of attribute or operation output



Object orientation

- Reason for overriding:
  - to specify behavior that depends on the subclasses
  - to tighten the specification of a feature.
  - for better performance

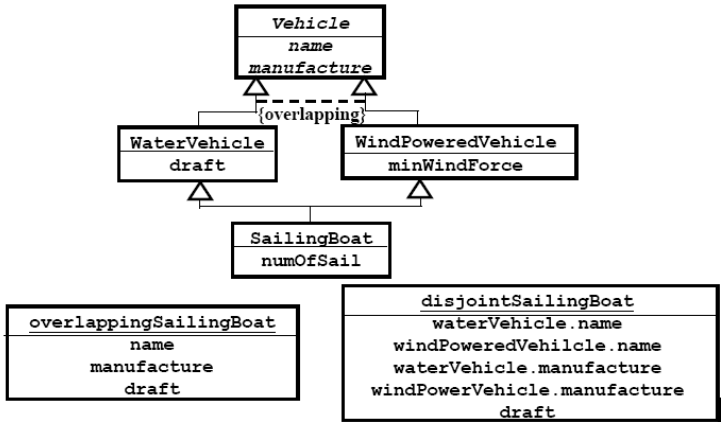
Specialization	class relationship	type relationship
Extension	subclass	subtype
Redefinition	subclass	same type
Restriction	subclass	supertype

Object orientation

- **Multiple Inheritance**
  - A class has more than one superclass and inherits features from all parents.
    - –Disjoint (default):
      - Inherit all attributes of all class
    - Overlapping
      - To ensure that these attributes are only inherited once
      - Need conflicts resolution
    - if a class can be refined on several distinct and independent dimensions, then use multiple generalizations. (inheritance)
    - generalization : conceptual level
      - inheritance : implementation level. (mechanism)

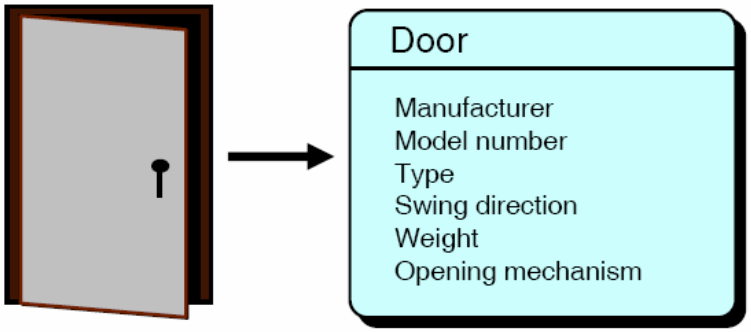
Object orientation

- **Multiple Inheritance**



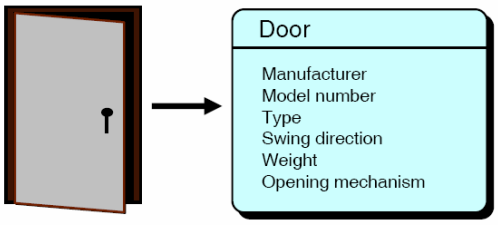
Object orientation

- **Abstraction**



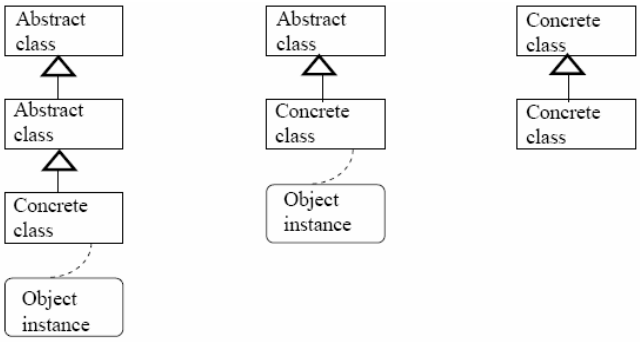
Object orientation

- **Abstraction**
  - Abstraction: focus on the essential, inherent aspects of an entity and ignoring its accidental properties.
  - abstraction during analysis: deciding only with application-domain concepts, not making design and implementation decisions.



Object orientation

- Abstract class : has no direct instances.
- Concrete class : can be leaf (instantiable) in the inheritance tree

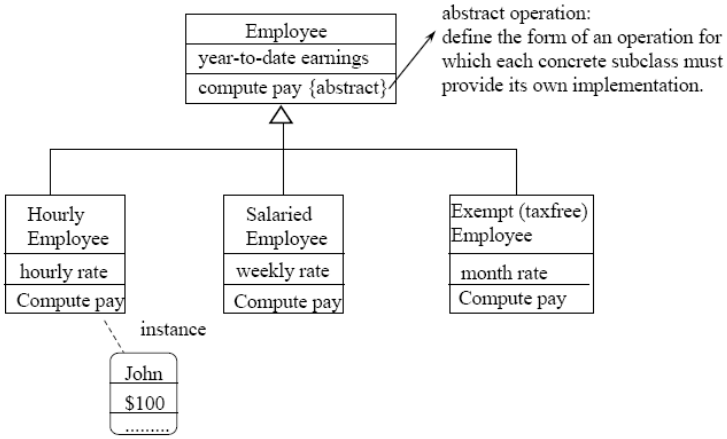


Object orientation

- Origin class : the topmost defining class, which defines the protocol of the feature
  - the type of an attribute
  - the number and type of arguments
  - result type for operations
  - the semantic intent

Object orientation

- Abstraction

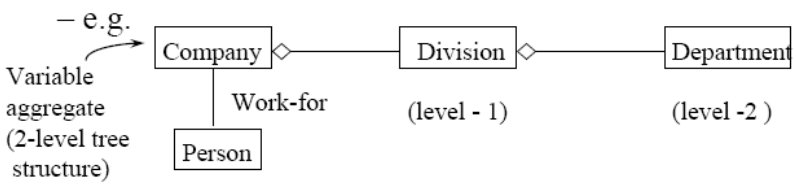


Object orientation

- **Association/Aggregation**
  - Message exchange
  - Association
    - a structural relationship that describes a set of links, in which a link is a connection among objects
  - Aggregation
    - A special form of association that specifies a whole-part (a-part-of) relationship between the aggregation (whole) and a component (part).

Object orientation

- Aggregation
  - aggregation is a special form of association.
    - Part-whole (Object1,Object2) --> aggregation.
    - Independent (Object1,Object2) --> association.
  - e.g.

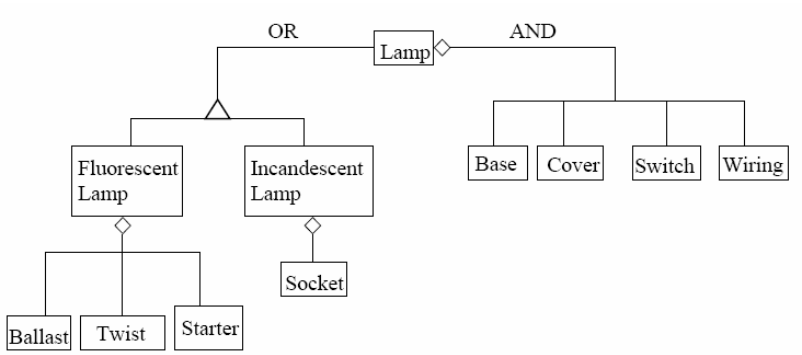


Object orientation

- Aggregation or generalization
  - an aggregation tree is composed of object instances that are all parts of a composite object.
    - parts explosion (distinct object).
  - a generalization tree is composed of classes (structuring and description of a single object).

Object orientation

- Aggregation / Association



Object orientation

- Polymorphism:
  - the same operation may behave differently on different classes
    - method: a specific implementation of an operation
    - a polymorphic operation is an operation that have more than one method implementing it.
- Static polymorphism
  - (overloading): an invocation can be operate on arguments of more than one type.

Object orientation

- Dynamic polymorphism: an object has more than one type
  - object reference can refer to an instance of any of descendants of its class
  - a instance of the class created, and is also an instance of each that class's ancestors

Object-Oriented Analysis and Design

- OMT Methodology
  - Object Model Technology
- UML
  - Unified Modeling Language