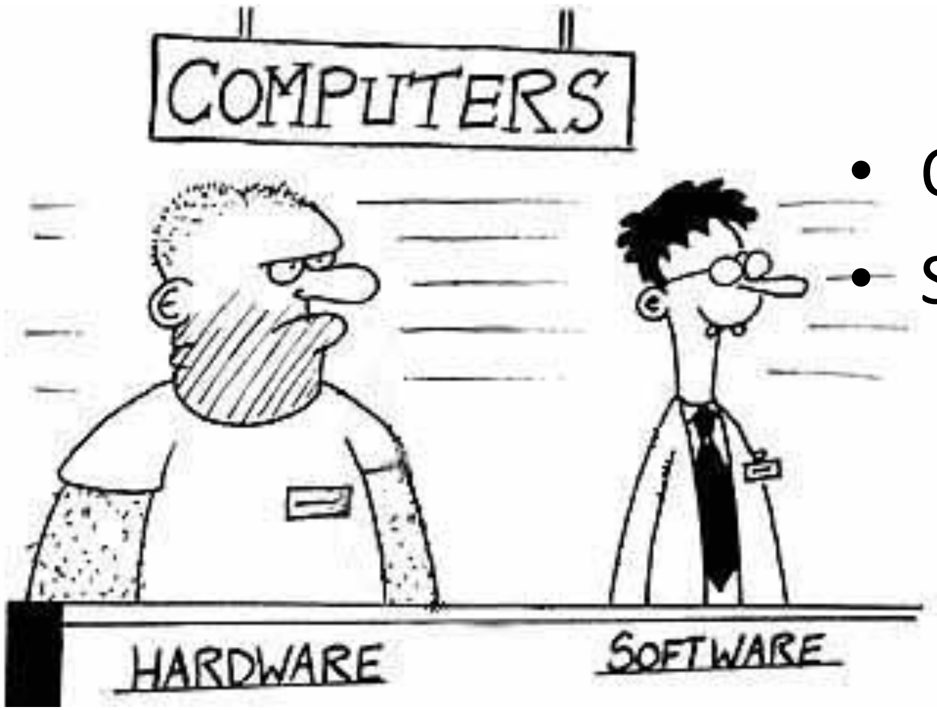
Week 7^{th}

Prepared by Dr Syed Khaldoon Khurshid

PART 2

Software

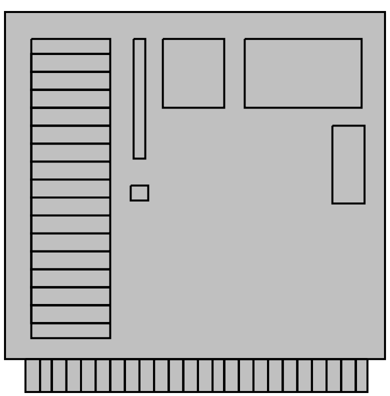


- Operating Systems
- Software Engineering
  - Algorithms
  - Programming Languages

- **Note:** ***hardware & software are logically equivalent!***

Hardware/software tradeoffs - ACM Digital Library

Hardware and software are logically equivalent. Any operation performed by **software** can also be built directly into the **hardware** and any instruction executed by the **hardware** can also be simulated in **software.**



PART 2

Software

- **Hardware & software logically equivalent:**
  - Any operation performed by software can also be built directly into hardware
  - Any instruction executed by hardware can also be simulated in software

CHAPTER 6

Software Engineering

Reference: Computer Science an Overview
Author: J. Glenn Brook Shear
6th Edition

- **Building LARGE / complex software systems**

6.1: Engineering Example

- **Design**
- **Re-design**
- **Construction**
- **Integration of parts**
- **Materials**
- **Transportation**
- **Financing**
- **Time assessment**
- **Personnel**
- **Politics**
- **Drawings/Documentation**
- **...**

Basílica de la Sagrada Família
Started construction in 19th March 1882
More than 140 years in Construction

6.1: Software Engineering

- Building large software systems is engineering effort too, incl.
  - division of problem into manageable parts
  - integration of separately developed units
  - cost assessment (time / money, …)
  - personnel management…
- But it’s not exactly identical
- In traditional engineering pre-defined components
  - **Off the shelf components:** In engineering design, the off-the-shelf (OTS) components are **hardware products that are ready-made and available for sale to the general public**.

6.1: Software vs. Real-world Engineering

- SE differs from real-world engineering:
  - reuse of pre-fabricated parts often not possible
    - so, large systems often built from scratch
  - software is either correct or incorrect
    - no tolerances, as in real-world ‘objects’
  - ‘quality’ of software is hard to define / measure
    - real-world measure: how well does object endure strain over time?
  - software does not wear out…
    - … but it can become outdated

Research in Software Engineering

- **Two Levels**
  - **Practitioners:** Work toward developing techniques for immediate applications.
  - **Theoreticians:** Search for underlying Principles and theories on which more stable techniques can someday be constructed.
- Both needed.
- **ACM & IEEE**.

Hardware/software tradeoffs
delivery.acm.org
https://dl.acm.org/citation.cfm?id=1296938

ACM DL DIGITAL LIBRARY

University of Engg. and Technology (UET) Lahore

SIGN IN SIGN UP

SEARCH

Check out the beta version of the *next* ACM DL

Hardware/software tradeoffs: a general design principle?

Full Text: Pdf

Author: Brian Randell The University of Newcastle upon Tyne

1985 Article

Published in:

· Newsletter
ACM SIGARCH Computer Architecture News archive
Volume 13 Issue 2, June 1985
Pages 19 - 21
ACM New York, NY, USA
table of contents doi>10.1145/1296935.1296938

Bibliometrics

- Citation Count: 0
- Downloads (cumulative): 74
- Downloads (12 Months): 4
- Downloads (6 Weeks): 1

Tools and Resources

TOC Service:
Email RSS

Save to Binder

Export Formats:
BibTeX EndNote ACM Ref

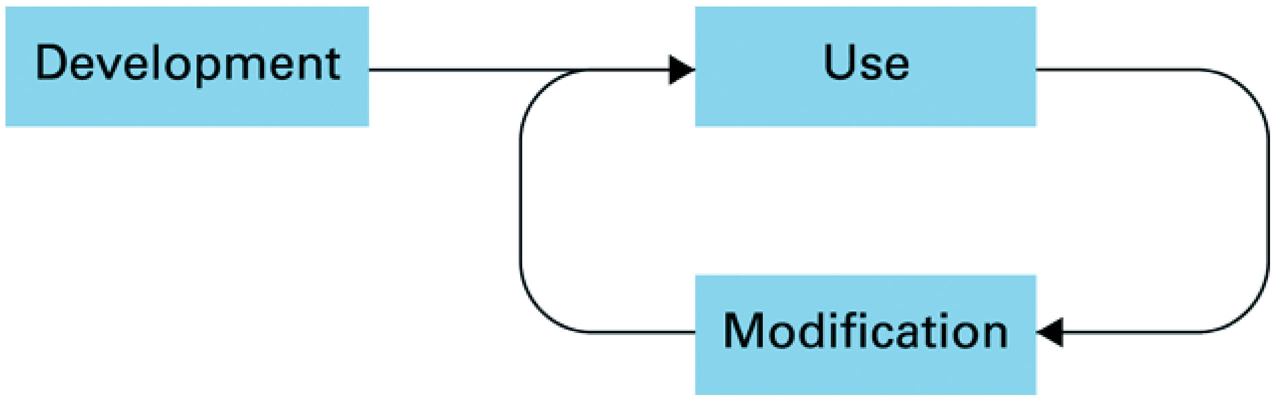
Share:

9:53 AM
25-Nov-19

6.1: Large/complex software systems

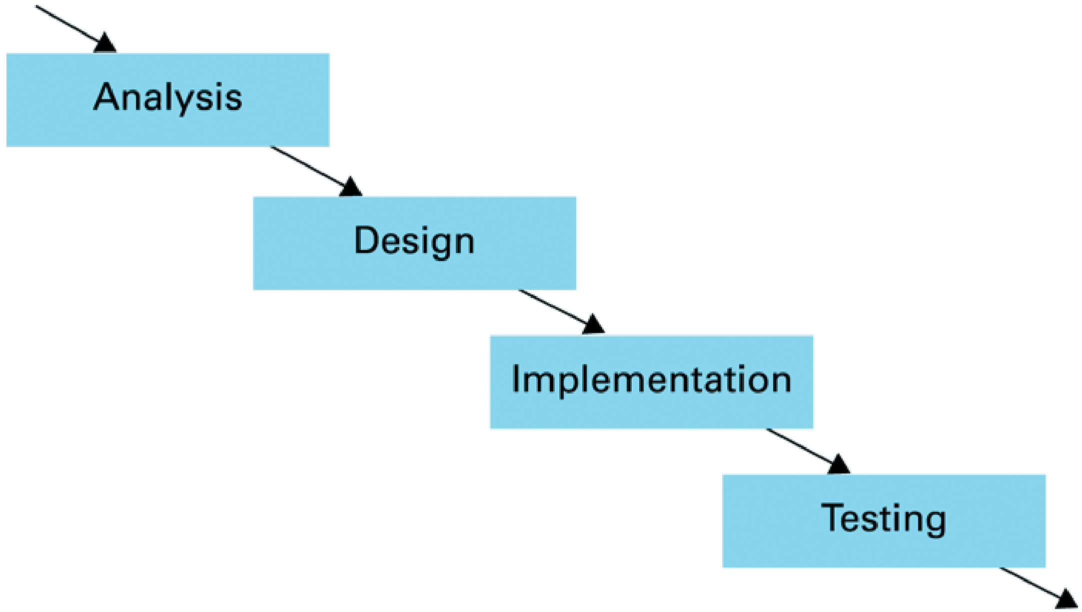
- **Common idea/mistake:**
  - Large/complex = many lines of code
- **More realistic:**
  - Large/complex = many interrelated entities that need to work together as a single system
- **Note:**
  - goal of software engineering is to make such systems ‘manageable’

6.2: The Software Life Cycle



- For real-world objects: modification = repair
- Modification phases combined often much more costly than development phase
  - ‘modification’ often is: ‘redesign from scratch’
  - note: comments in code are essential

6.2: Development Phase



- Compare: 'art of problem solving'
- General cost estimate (in time):

Analysis: 30%	-	Design: 20%
Implementation: 10%	-	Testing: **40%**

Models in Software Engineering

- Water Fall Model
- Incremental Model
- Prototyping
  - Evolutionary Prototyping
  - Throwaway Prototyping
    - Rapid Prototyping
- Agile Model

Models in Software Engineering

- Agile Model
- The meaning of Agile is swift or versatile."**Agile process model**" refers to a software development approach based **on iterative development.** Agile methods break tasks into **smaller iterations**, or parts do not directly involve long term planning. The project scope and requirements are laid down at the **beginning of the development process**. Plans regarding the number of iterations, the duration and the scope of each iteration are clearly defined in advance.
- Each iteration is considered as a short time **"frame"** in the Agile process model, which typically lasts from **one to four weeks**. The division of the entire project into smaller parts helps to **minimize the project risk and to reduce the overall project delivery time requirements**. Each iteration involves a team working through a full software development life cycle including planning, requirements analysis, design, coding, and testing before a working product is demonstrated to the client.

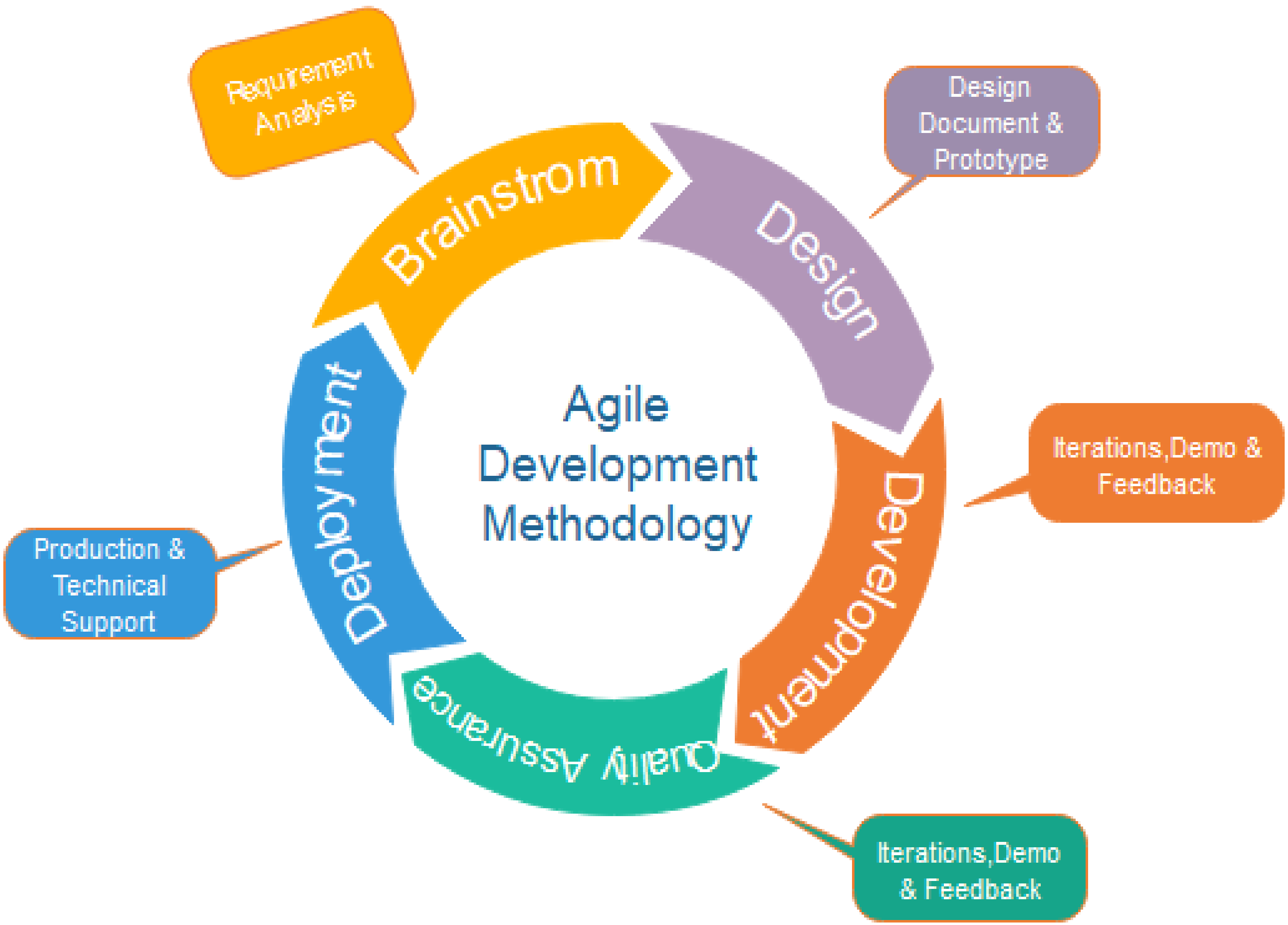
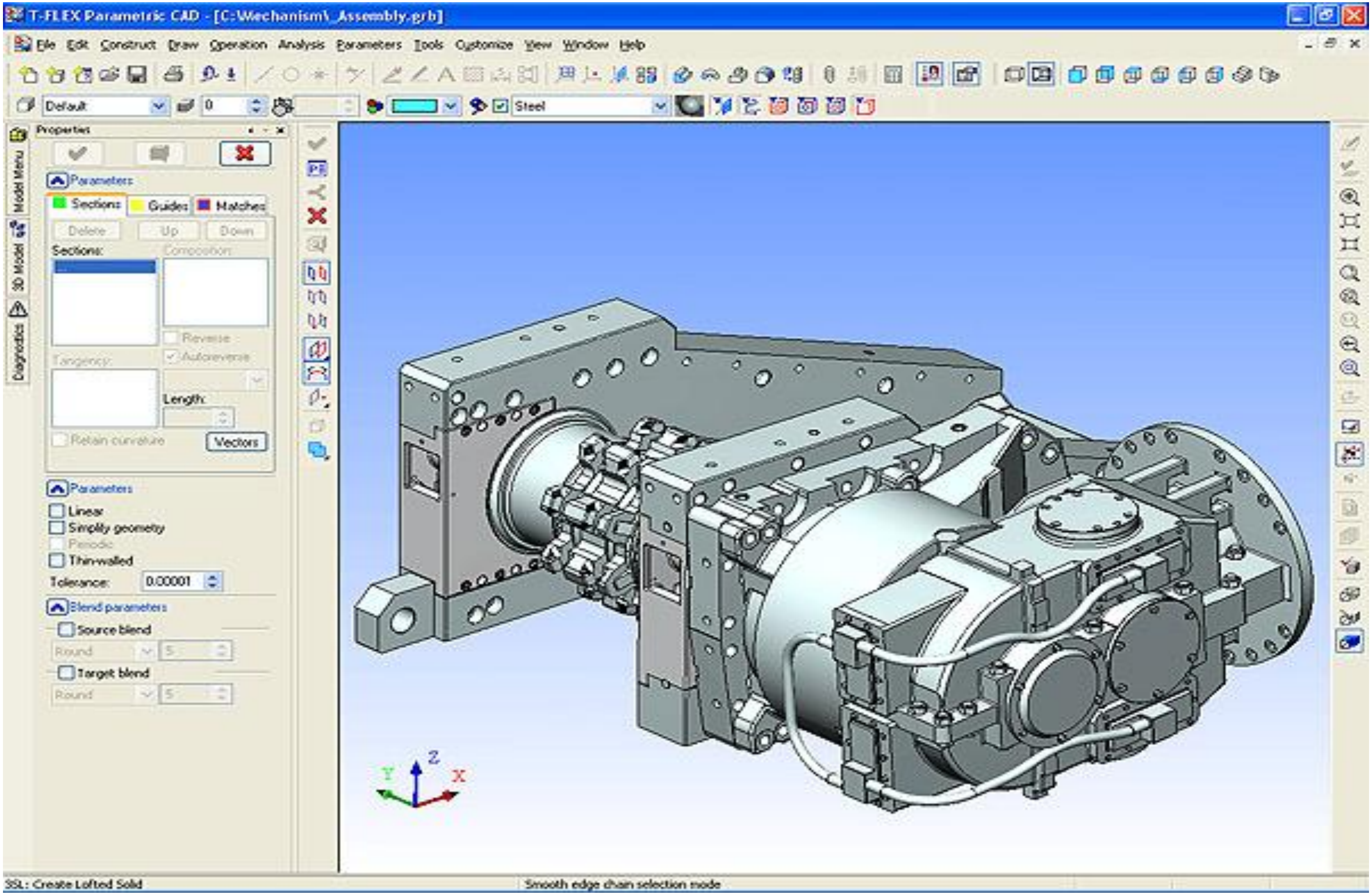


Fig. Agile Model

Trends in Software Engineering

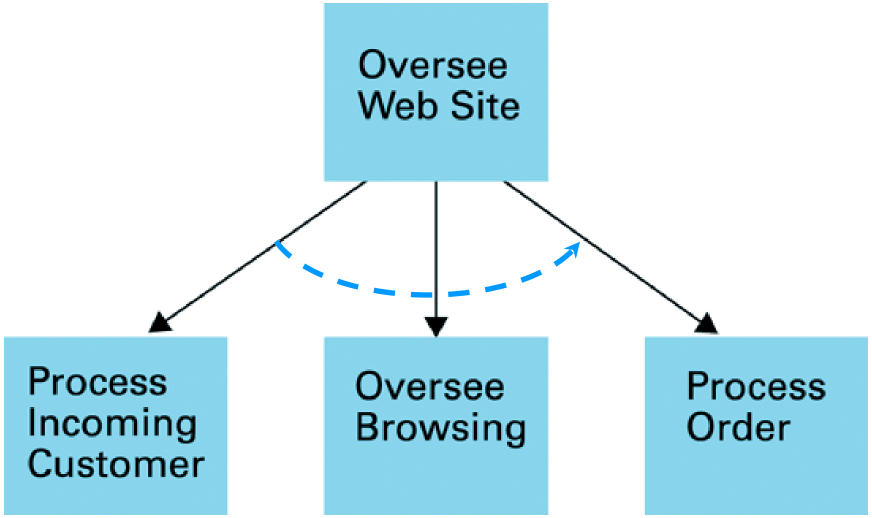
- **CASE (Computer-Aided Software Engineering)**
  - Project Planning Tools
  - Project Management Tools
  - Documentations Tools
  - Prototyping & Simulation Tools
  - Interface Design Tools
  - Programming Tools

Computer Aided Design



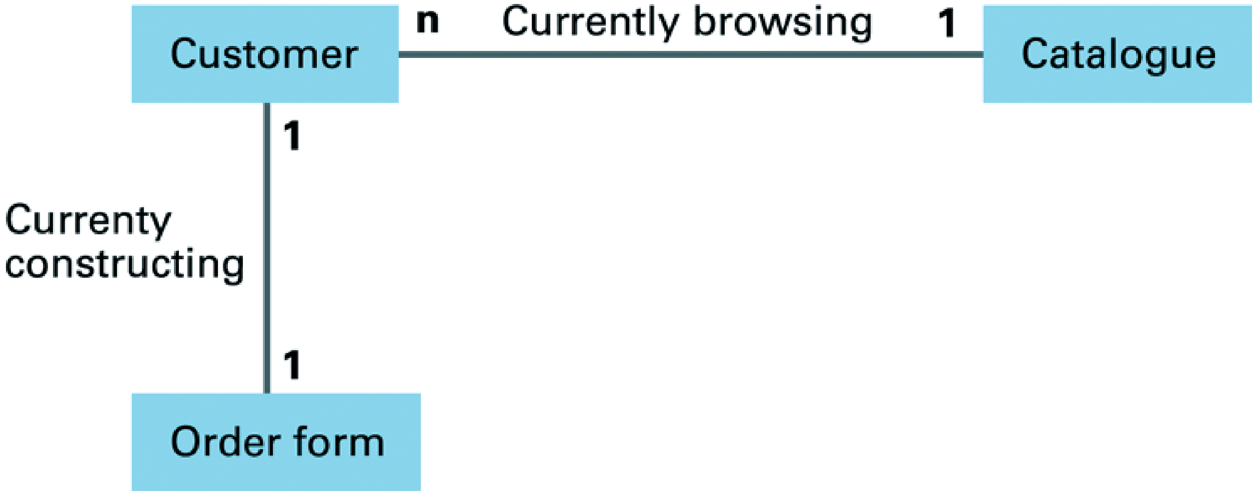
Modularity

- Modularity:
  - division of software into manageable parts, each of which performs a subtask only
    - e.g.: procedures, objects, …
- Representation of procedural modularity
  - structure chart:



6.3: Modularity in OO systems

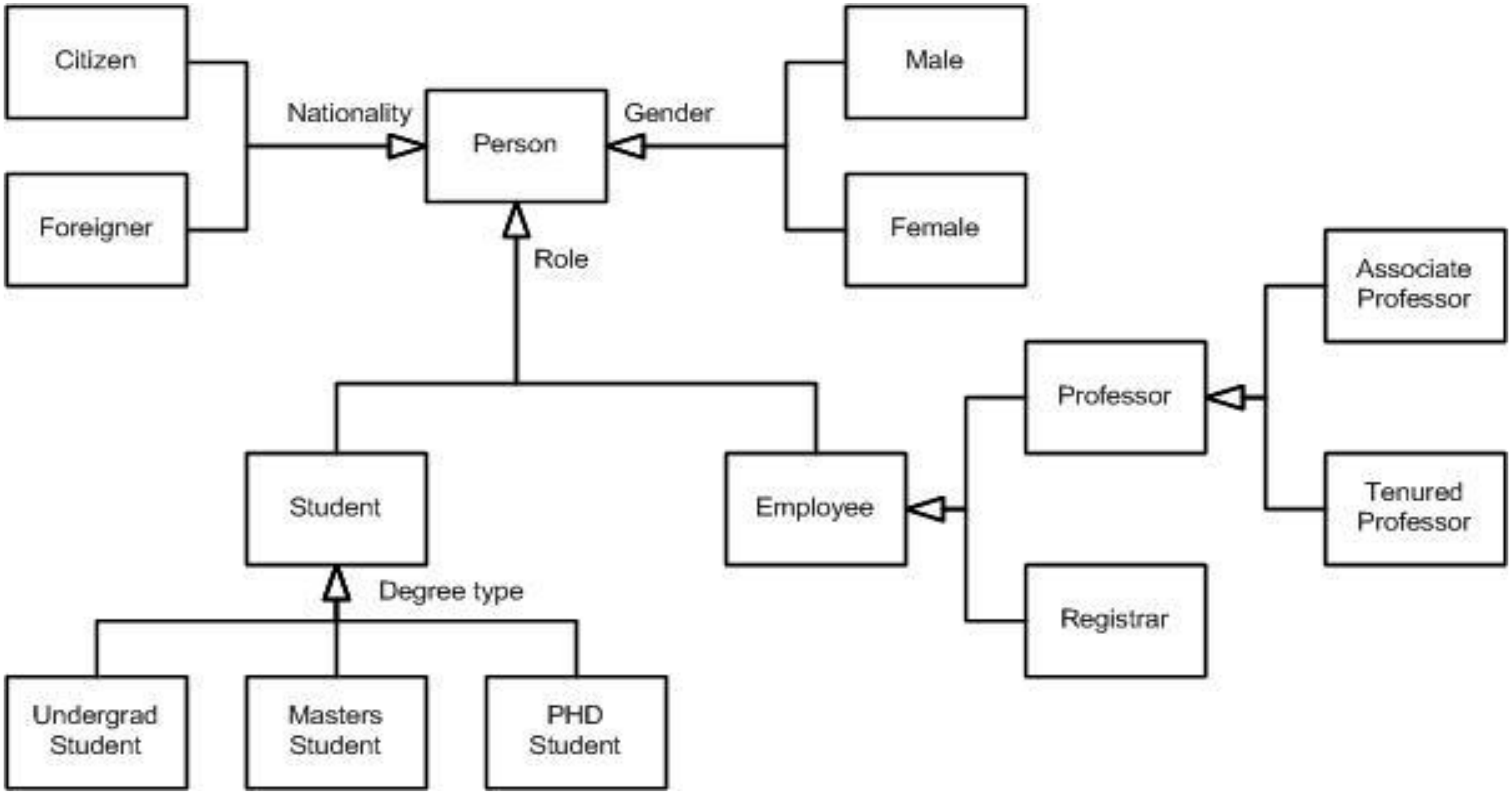
- Representation of object-oriented modularity
  - class diagram:



- Objects related by ‘relationships’ (i.e.: methods)
  - here: *one-to-one* and *one-to-many*

UML Conventions

The Unified Modeling Language (UML) is a general-purpose, developmental, modeling language in the field of software engineering that is intended to provide a standard way to visualize the design of a system.

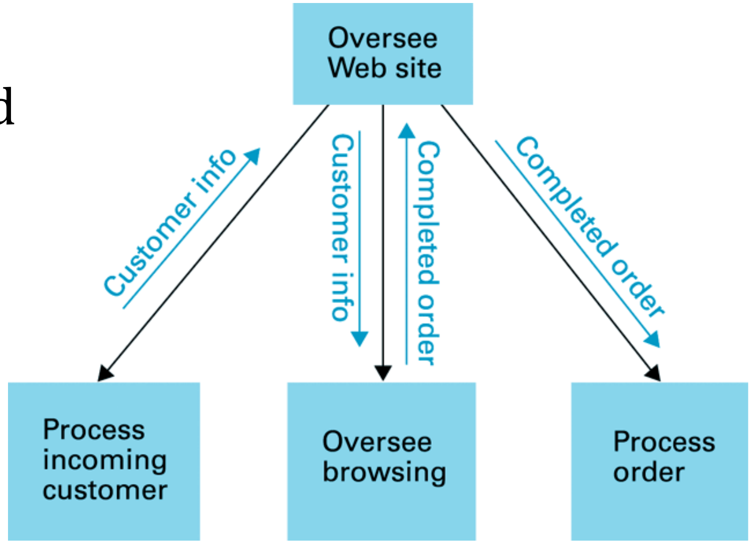


6.3: Inter-Module Dependencies (1)

- Modularity is to obtain *maintainable* software
  - future modifications will only affect few modules
- Note:
  - only when dependence between modules in minimized
- Unfortunately:
  - some dependency always needed to form a coherent system

6.3: Inter-Module Dependencies (2)

- Dependency between modules known as
  - ‘coupling’
- Two forms:
  - **‘control coupling’**
    - passing of control from one module to another
    - i.e.: sequence of procedures called
  - **‘data coupling’**
    - sharing of data between modules
    - i.e.: data passed as parameters

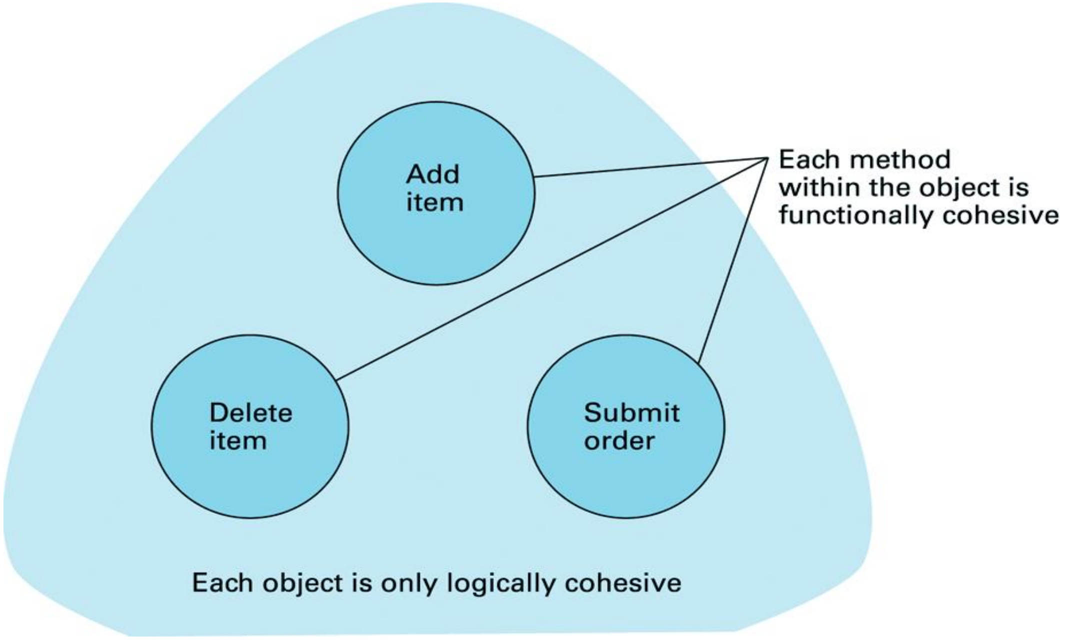


6.3: Inter-Module Dependencies (3)

- **Note: main benefit of OO-design:**
  - data coupling is minimized
  - inter-object dependencies by method invocations (i.e.: control coupling)
- **Danger: ‘implicit coupling’**
  - by ‘global’ data that are accessible by all modules
  - difficult to trace global data accesses and updates
- **So: minimize use of global data!**

6.3: Intra-Module Dependencies

- **Also important:**
  - maximize bindings (or ‘cohesion’) *within* a module
  - ‘put together what belongs together’
- **In OO:**



Cohesion forms:

- **Logical Cohesion**: Within a module induced by the fact that its internal elements perform activities logically similar in nature.
- Elements of component **are related logically and not functionally.**

Example:

- A component reads inputs from tape, disk, and network.
    a. All the code for the functions are in the same component.
    b. Operations are related, but the functions are significantly different.

- **Functional Cohesion:** All the parts of a module are geared towards the performance of a single activity.
- **Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module**. Focused (strong, single minded purpose) and no element doing unrelated activities.
- **Examples:**
  1. Compute cosine of angle
  2. Read transaction record
  3. Assign seat to airline passenger

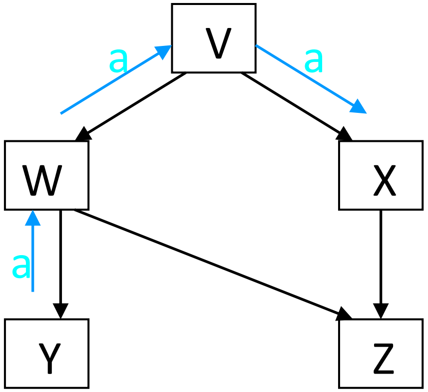
Chapter 6 - Problem 13

Which is an argument for *coupling*, and which for *cohesion*:
a) For a student to learn, a subject should be presented in well-organized units with specific goals.
b) A student does not really understand a subject until the subject's overall scope and relationship with other subjects has been grasped.

- **Coupling => b**
  - relationships among subjects are like data coupling
- **Cohesion => a**
  - well-organized internal subject structure similar to logical cohesion

Chapter 6 - Problem 16

Answer in relation to the following structure chart:



a) To which module does module Y return control? => W

b) To which module does module Z return control? => W, X

c) Are modules W and X linked via control coupling? => no

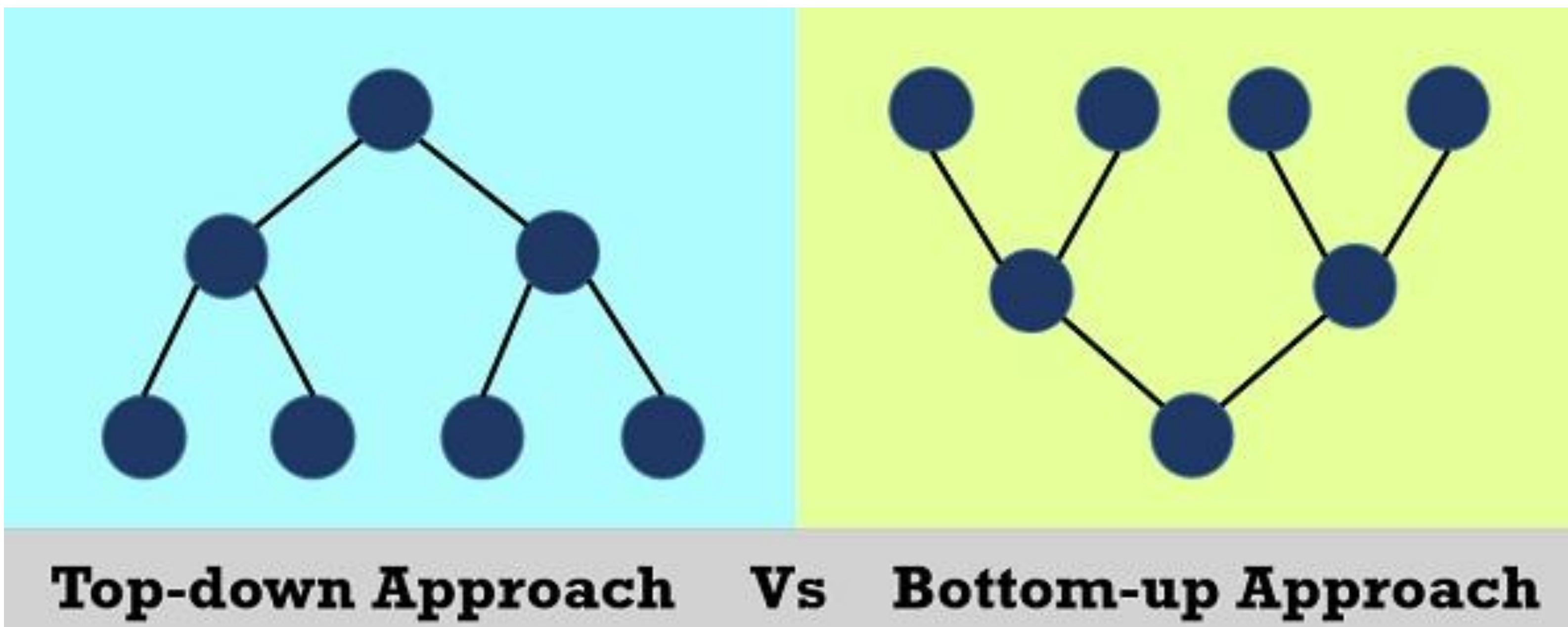
d) Are modules W and X linked via data coupling? => yes

e) What data is shared by both module W and module Y? => parameter ‘a’

f) In what way are modules Y and X related? => data coupling

Design Methodologies

- **Top-down Approach** starts with the big picture. It breaks **down** from there into smaller segments.
- **Bottom-up Approach** is the piecing together of systems to give rise to more complex systems, thus making the original systems sub-systems of the emergent system.



Tools of the Trade

- **Dataflow Diagram**
- **Entity-Relationship Diagram**
  - Example: Professor, Classes and Students
  - Object-Oriented Design Environments
- **Data dictionary**
  - Avoiding misunderstanding
  - Reducing Redundancy and contradiction
- **CRC (Class-Responsibility-Collaboration)**
  - Traditional index card on which description of object is being written.

CRC Examples:

Class Name	
Responsibilities	Collaborators

Student	
Student number Name Address Phone number Enroll in a seminar Drop a seminar Request transcripts	Seminar

Class Broker	***Collaborators*** • Client • Server • Client-side proxy • Server-side proxy • Bridge
Responsibilities • Register and unregister servers • Provide APIs • Transfer messages • Error recovery • Interoperate with other broker systems through bridges • Locate servers	

Design Patterns

- Inspiration from Architecture
- In 1977, Pattern Language by Christopher Alexander et al.
  - *templates for universal problems*
  - *Quiet backs*
- In software engineering, **a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern isn't a finished design** that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

  https://sourcemaking.com/design_patterns
- Software Researchers are applying design patterns as means of providing generic building blocks with which software can be constructed.
  - *JAVA, API and JDK*

Design Patterns: In software engineering

- A **general reusable solution** to a commonly occurring problem within a given context in software design.
- Patterns are formalized best practices that the programmer can use to solve common problems when **designing an application or system**.
- **Object-oriented design patterns** typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Pareto principle

- The **Pareto principle** (also known as the **80–20 rule** and the **law of the vital few**) states that, for many events, roughly 80% of the effects come from 20% of the causes.
- in 1896, published his first paper "Cours d'économie politique." Essentially, Pareto showed that approximately **80% of the land in Italy was owned by 20% of the population**; Pareto developed the principle by observing that 20% of the pea pods in his garden contained 80% of the peas.
- It is a common rule of thumb in business; e.g., "**80% of your sales come from 20% of your clients”**.

Pareto principle: In Software Engineering

- In computer science, the Pareto principle can be applied to **optimization** efforts.
- For example, Microsoft noted that by fixing the top 20% of the most-reported bugs, 80% of the related errors and crashes in a given system would be eliminated.
- In software engineering, Lowell Arthur expressed a corollary principle: "**20 percent of the code has 80 percent of the errors. Find them, fix them**!”

Testing

- **Complete testing impossible.**
- **Pareto principle:**
  - small number of modules within a large software system is more problematic than the rest.
- **Glass-Box Testing:**
  - software interior visible to tester
  - Basis Path Testing
    - Every line must be executed at least once.
- **Black-Box Testing: User point of view**
  - Boundary Value Analysis: Extreme ranges and demanded activities
  - Applying redundancy: two software applying same task.
  - Shrink wrapped: beta testing
    - Benefits: feedback & marketing

Documentation

- **User Documents**
- **Good documentation**
  - books and technical writers
- **Manual and Packages**
- **Internal composition of Software**
- **High level language**
  - Comments
  - Indentions
  - Conventions
- **Design Documents**
- **Updating by CASE**


Any
Questions