Software Engineering Principles

Dr Zhiquan (George) Zhou
Associate Professor

Software Engineering Principles

To achieve the goals

Software: Its Nature and Qualities
(Goals of Software Engineering practice)

- Software engineering ?

Definitions of SE

- The application of engineering to software
- Field of computer science dealing with software systems
  - large and complex
  - built by teams
  - exist in many versions
  - last many years
  - undergo changes

Definitions of SE

- (1) Application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software;

  (2) The study of approaches as in (1)

  (IEEE 1990)
- Multi-person construction of multi-version software (Parnas 1978) ▪

A shortened history of SE

- In the early days
  - Programmer = user
    - E.g. a physicist writing a program to solve a differential equation

A shortened history of SE

- Later, from late 1950s, "programming" started to attain the status of a profession
  - A programmer can write a program for you
  - A user has to specify the requirements
  - Misinterpretation was possible even in small tasks

A shortened history of SE

- Middle to late 1960s: truly large software systems were attempted to be built commercially
  - OS 360 Operating System for IBM 360 computer family

A shortened history of SE

- Middle to late 1960s: (cont.)
  - People on these projects quickly realized:
    - Building large software systems was significantly different from building smaller systems
      - fundamental difficulties in scaling up the techniques

A shortened history of SE

First used in a NATO conference held in Germany, 1968.

- Middle to late 1960s: (cont.)
  - The term *Software Engineering* was invented around this time.
  - Large software projects were universally over budget and behind schedule.
  - Another term invented at the time was "software crisis."

A shortened history of SE

- The problems being solved were **not well understood**
  - People had to spend a lot of time communicating with each other rather than writing code

A shortened history of SE

- The problems being solved were **not well understood** (cont.)
  - People sometimes left the project, and this affected not only the work they had been doing but also others' work

A shortened history of SE

- The problems being solved were **not well understood** (cont.)
  - Replacing an individual required an extensive amount of training about the "folklore" of the project requirements and design

A shortened history of SE

- The problems being solved were **not well understood** (cont.)
  - Any change in the original requirements seemed to **affect many parts** of the project

A shortened history of SE

- The problems being solved were **not well understood** (cont.)

These problems just did not exist in the early "programming" days.

A shortened history of SE

- The problems being solved were **not well understood** (cont.)

The inherent difficulties of software development are not short-term problems.

There is no magic—no "silver bullet".

A shortened history of SE

- Question:

  Will the importance of SE continue to grow ??

A shortened history of SE

- We can expect the importance of SE to continue to grow
  - Economic reason: worldwide expenditures in software continue to rise
    - This fact alone ensures that SE will grow as a discipline.

A shortened history of SE

- We can expect the importance of SE to continue to grow (cont.)
  - Software is permeating our society. More and more, software is used to
    - control critical machines: aircraft, medical devices, …
    - worldwide critical functions: eCommerce
    - This fact ensures the growing interest of society in dependable software.

A shortened history of SE

- We can expect the importance of SE to continue to grow (cont.)
  - No doubt, it will continue to be important to learn how to build better software better.

Role of software engineer

- Programming skill not enough
- Software engineering involves "programming-in-the-large"
  - understand requirements and write specifications
    - derive models and reason about them
  - master software
  - operate at various abstraction levels
  - member of a team
    - communication skills
    - management skills ▪

SE Principles

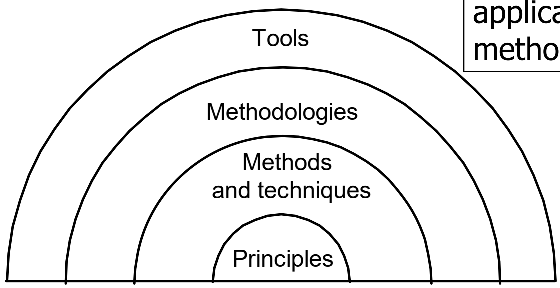
(principles central to successful sw development)

- Principles apply to process and product
- Principles form the basis of *methods*, *techniques*, *methodologies* and tools
- **Seven** important principles that may be used in all phases of software development
- **Modularity** is the cornerstone principle supporting software design

Each layer is based on the layer(s) below it.

changing vs. *enduring*

Sometimes, methods and techniques are packaged together to form a *methodology*. The purpose of a methodology is to promote a certain approach to solving a problem by preselecting the methods and techniques to be used. *Tools*, in turn, are developed to support the application of techniques, methods and methodologies.



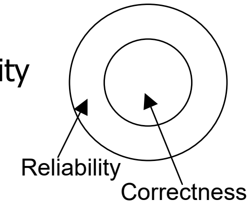
Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Recall software qualities

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

– Correctness, reliability, and robustness
– Performance
– Usability
– Verifiability
– Maintainability
– Reparability
– Evolvability
– Reusability
– Portability
– Understandability
– Interoperability
– Productivity
– Timeliness
– Visibility

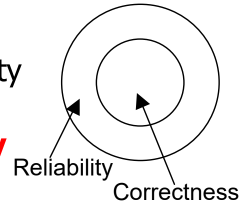


Recall software qualities

If no **reliability** and **evolvability** requirements, the need for SE principles and techniques diminishes greatly.

- Anticipation of change
- Generality
- Incrementality

– Correctness, **reliability**, and robustness
– Performance
– Usability
– Verifiability
– Maintainability
– Reparability
– **Evolvability**
– Reusability
– Portability
– Understandability
– Interoperability
– Productivity
– Timeliness
– Visibility



Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Apply throughout the sw development process.

Not exhaustive .

Key principles

- **Rigor and formality**
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Rigor and formality

Discussion:

- Software development is a creative design activity
  - In any creative process, there is an inherent tendency to be neither precise nor accurate
  - But rather to follow the inspiration of the moment in an unstructured manner
- Rigor: precision and exactness
- Why rigor?

Rigor and formality

- Software development is a creative design activity, BUT Engineering=\=arts
- It must be practiced systematically
- Rigor is a necessary complement to creativity that increases our confidence in our developments
- Formality is rigor at the highest degree
  - a software process driven and evaluated by mathematical laws

Programs are formal objects.

Examples: product

- Mathematical (formal) analysis of program correctness
- Systematic (rigorous) test data derivation

Example: process

- Rigorous **documentation** of development steps helps project management and assessment of **timeliness .**

This slide is not examinable.

Formality: programs are a formal subject

Discussion: Why computers are powerful ?

This slide is not examinable.

Formality: programs are a formal subject

– Why computers are powerful ?

Branch and Loop

How do you write programs/loops?

This slide is not examinable.

Formality: programs are a formal subject

- How to construct loops ?
  – Invariants and bound functions

invariant properties

Reference: Roland Backhouse, *Program Construction: Calculating Implementations from Specifications,* John Wiley & Sons, 2003

This slide is not examinable.

Preliminaries

- How to construct loops ?
  – Invariants and bound functions

A measure of the **size** of the problem to be solved. It should be an **integer**-valued function of the program variables that is guaranteed to be **> 0** when the loop is executed (at the beginning of each iteration). A **guarantee** that the value of the bound function is always **decreased** at each iteration is a **guarantee** that the execution will **halt**/terminate --- the number of iteration times is at most the initial value of the bound function.

Roland Backhouse, *Program Construction: Calculating Implementations from Specifications,* John Wiley & Sons, 2003

This slide is not examinable.

Formality: programs are a formal subject

- How to construct loops ? (cont.)
  - Design principles
    - **Each iteration** of the loop body ***maintains*** the **invariant** whilst making progress to the goal by always ***decreasing*** the bound function .

Roland Backhouse, *Program Construction: Calculating Implementations from Specifications,* John Wiley & Sons, 2003

Key principles

- Rigor and formality
- **Separation of concerns**
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Separation of concerns

- To dominate complexity, separate the issues to concentrate on one at a time
- "Divide & conquer"
- Supports parallelization of efforts and separation of responsibilities

Separation of concerns

- Many concerns about the product
  - Functions to offer
  - Reliability
  - Efficiency (space and time)
  - Environment: hw/sw resources required
  - User interfaces
  - …

Separation of concerns

- Many concerns about the process
  - Development environment
  - Organization and structure of the team
  - Scheduling
  - Control procedures
  - Design strategies
  - Error recovery mechanisms
  - …
- And economic and financial matters … .

Separation of concerns

- Many decisions are strongly related
  - e.g. a **design decision**: swapping some data from main memory to disk

Separation of concerns

- Many decisions are strongly related
  - e.g. a **design decision**: swapping some data from main memory to disk
    - Depend on the size of the memory of the target machine
      - and hence, the _____ of the machine

Separation of concerns

- Many decisions are strongly related
  - e.g. a **design decision**: swapping some data from main memory to disk
    - Depend on the size of the memory of the target machine
      - and hence, the **cost** of the machine
    - May affect the **policy** for ___________

Separation of concerns

- Many decisions are strongly related
  - e.g. a **design decision**: swapping some data from main memory to disk
    - Depend on the size of the memory of the target machine
      - and hence, the **cost** of the machine
    - May affect the **policy** for error recovery

Separation of concerns

- Many decisions are strongly related
  - It would be good if all issues are considered at the same time by the same people
    - But this is often impossible in practice.

Separation of concerns

- We should try to **isolate** issues that are not so closely related to the others
- Then consider these issues **separately**, together with only the *relevant* details of related issues

Separation of concerns

- Way 1: separate them in *time*
  - E.g. a professor:
    - Teaching related activities: class, seminar
      - 9 am to 2pm Monday to Thursday
    - Consultation
      - Friday morning
    - Meeting with research students
      - Friday afternoon
    - Research
      - Rest of the time

Separation of concerns

- Way 1: separate them in time (cont.)
  - Allows for precise planning of activities
  - Eliminates overhead in switching from one activity to another in an unconstrained way
    - That's why we have "consultation hours"!

Separation of concerns

- Way 1: separate them in time (cont.)
  - E.g. process
    - go through **phases** one after the other (as in waterfall)
    - is the underlying motivation of the software process models, each of which defines a sequence of activities that should be followed in sw production .

Separation of concerns

- Way 2

Question:

- During development, so many **qualities**, how to address all of them all together??
- During verification, how to verify them all together?

Separation of concerns

- Way 2: separate them in *qualities*
  - E.g. deal with efficiency and correctness separately
    - First, **design** software in such a structured way that its correctness is expected to be guaranteed (e.g. using formal methods)
    - Next, restructure the program partially to improve its efficiency
    - In **verification** phase, first check the functional correctness, then its performance.

Separation of concerns

- Way 2: separate them in qualities (cont.)
  - E.g. product: keep product requirements separate
    - functionality
    - performance
    - user interface and usability ▪

Exercise

- Show in a simple program of your choice how you can deal separately with correctness and efficiency.

Separation of concerns

- Way 3: in different views
  - Real-life example?
  - How many kinds of **flows** does a program have?

Separation of concerns

- Way 3: in different views
  - E.g. when we analyze the requirements of an application
    - It may be helpful to concentrate separately on the *data flow* from one activity to another in the system and the *control flow* that governs the way different activities are synchronized.
  - Neither way gives a complete view, but both views help ▪

Separation of concerns

- Way 4: deal with *parts* of the system separately—separation in terms of size
  - This is a fundamental concept that we need to master to dominate the complexity of software production
  - It is so important that we will detail it under modularity .

Separation of concerns

- Discussion: disadvantages?

An inherent disadvantage of Separation of Concerns

- We might miss some global optimization that would be possible by tackling them together

An inherent disadvantage of Separation of Concerns

- We might miss some global optimization that would be possible by tackling them together
  - However, our ability to make "optimized" decision in the face of complexity is rather limited.
  - If we consider too many concerns simultaneously, we are likely to be overwhelmed

Separation of concerns

- An important decision in design: which aspects to consider together and which separately
  - System designers and architects often face such trade-offs

Separation of concerns

- If two issues are intrinsically intertwined, i.e., the problem is not immediately decomposable, then
  - it is often possible to make some overall design decisions first
  - and then effectively separate the different issues.

Separation of concerns

- Example: consider a system in which online transactions access a database concurrently.

  In a first implementation, we could introduce a simple locking scheme that requires each transaction to lock the entire database at the start of the transaction and unlock it at the end.

  Suppose now that a preliminary performance analysis shows that some transaction, say t_i (which might print many records from the DB), takes so long that we cannot afford to have the DB unavailable to other transactions.

  Thus, the problem is to revise the implementation to improve its **performance** yet maintain the overall **correctness** of the system.

  Clearly, the two issues: functional correctness and performance, are **strongly related**.

Separation of concerns

- Example (cont.)

  So a first design decision must concern both of them: t_i is no longer implemented as an atomic transaction, but is split into several sub-transactions t_{i1}, t_{i2}, …, t_{in}, each of which is atomic itself.

  The new implementation may affect the correctness of the system, because of the interleaving that may occur between the executions of any two sub-transactions.

  Now, however, we have separated the two concerns of checking the functional correctness of the system and analyzing its performance; we may, then, do the analyses independently, possibly even by two different people with different expertise.

Separation of concerns

- Way 5: separate problem-domain concerns from implementation-domain concerns.
  - Problem-domain properties hold in general, regardless of how it is implemented
    - E.g. "requirements analysis/elicitation" and "requirements specification"

Separation of concerns

Example:

In designing a personnel management system, we must separate issues that are true about employees in general from those which are consequence of our implementation of the employee as a structure or an object. In the problem domain, we may speak of the *relationship* between employees, such as "employee A reports to employee B", and in the implementation domain we may speak of one object pointing to another. These concerns are often intermingled in many projects.

Separation of concerns

- Separation of concerns may result in separation of responsibilities
  - Thus, the principle is the basis for dividing the work on a complex problem into specific assignments, possibly for different people with different skills.

Examples ?

Separation of concerns

Example:

By separating managerial and technical issues in the process, we allow two types of people to cooperate in a software project.

Or, having separated requirements analysis from other activities in a software life cycle, we may hire specialized analysts with expertise in the application domain, instead of relying on internal resources. The analyst, in turn, may concentrate separately on functional and nonfunctional system requirements.

Key principles

- Rigor and formality
- Separation of concerns
- **Modularity**
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Modularity

- What is it?
- Examples in other engineering disciplines?
- Why do we need it?
- Does it support separation of concerns?
- What benefits does it bring to us?
- Relation to evolvability?

Modularity

- A complex system may be divided into simpler pieces called *modules*
- A system that is composed of modules is called *modular*
- Supports application of separation of concerns **(Why?)**
  - when dealing with a module we can ignore details of other modules

Modularity

- Is an important property of most engineering processes and products
  - E.g. cars:
    - assembling parts that are designed and built separately.
    - Furthermore, parts are often reused from model to model, perhaps after minor changes.

..

Modularity

- Main benefits
  - The capability of decomposing a complex system into simpler pieces
  - The capability of composing a complex system from existing modules
  - The capability of understanding a system in terms of its pieces, and
  - The capability of modifying a system by modifying only a small number of its pieces

Modularity

- Decomposing a complex system into simpler pieces
  - Top down
  - Latin motto *divide et impera* (divide and conquer)
    - Used by ancient Romans to dominate other nations: divide and isolate them first, and then conquer them individually

Modularity

- Composing a complex system from existing modules
  - Bottom up
  - Ideally, we wish to assemble new applications by taking modules from a library
  - Such modules should be designed with the goal of being reusable

Modularity

- Understanding AND modifying a system
  - They are **related** to each other
  - Understanding is the first step to applying modification

Relation to evolvability?

Modularity

- Understanding AND modifying a system
  - **Evolvability**: a major quality goal
    - Because of change
    - If the system can be understood only in its entirety, modifications are difficult to apply
    - Modularity helps confine the search to single components
    - Modularity forms the **basis** for software evolution

Cohesion and coupling

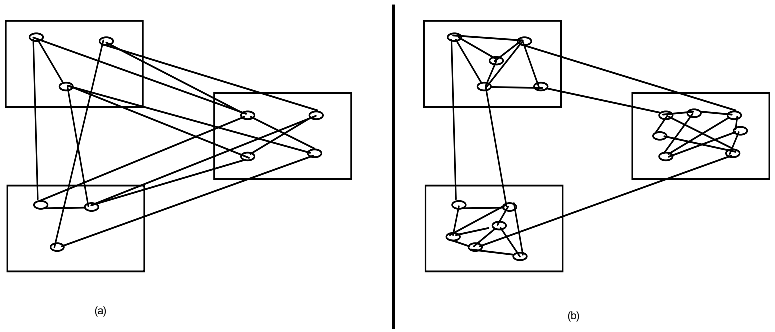
- What are they?

Cohesion and coupling

- Each module should be *highly cohesive*
  - All elements of a module should be related strongly
    - Statements, declarations, etc.
    - Because they are grouped together for a logical reason: the function of the module
- Modules should exhibit *low coupling*
  - should have low interactions with others
  - understandable separately

High cohesion and low coupling: in order to achieve composability, decomposability, understandability and modifiability

A visual representation



high coupling

low coupling

Exercise

- Explain some of the causes of, and remedies for, low cohesion in a software module.
- Explain some of the causes of, and remedies for, high coupling between two software modules.

Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- Incrementality

Abstraction

- What is it?
- Examples?
  - In your daily life?
  - In computer science?
  - In software development?
  - In software verification?
- Relation to separation of concerns?

Abstraction

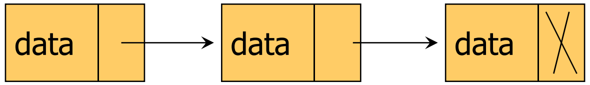
- Identify the important aspects of a phenomenon and ignore its details
  - Special case of separation of concerns
  - The type of abstraction to apply depends on purpose

Abstraction - example

- Programming language semantics described through an abstract machine that ignores details of the real machines used for implementation

```
struct Student{
int studentID;
float height;
float weight;
struct Student *next;
};
```

```
struct Student John, Peter;
John.studentID=1001;
John.next=&Peter;
```



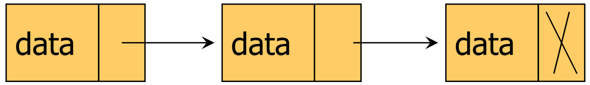
____________________________________ is ignored

Abstraction - example

- Programming language semantics described through an abstract machine that ignores details of the real machines used for implementation

```
struct Student{
int studentID;
float height;
float weight;
struct Student *next;
};
```

```
struct Student John, Peter;
John.studentID=1001;
John.next=&Peter;
```



The specific computer addressing mechanism is ignored

Abstraction - example

- Abstract Data Types:
  - Stacks
  - Queues
  - Trees
  - …

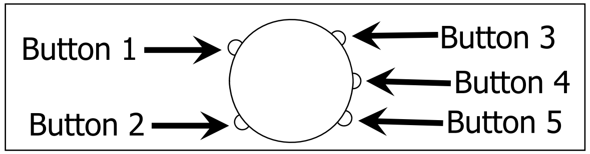
  So we can focus on ___________, rather than ____________.

Abstraction - example

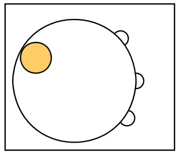
- Abstract Data Types:
  - Stacks
  - Queues
  - Trees
  - …

  So we can focus on the solution (the algorithm), rather than the implementation

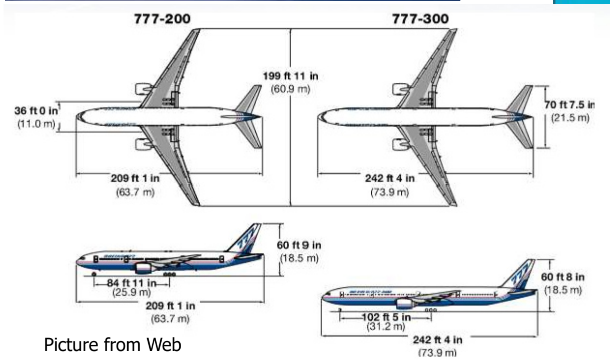
Abstraction for users (clockmakers would need more details!)



To replace battery:

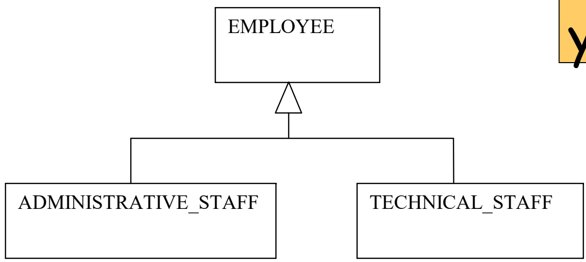


Abstraction yields models



Picture from Web

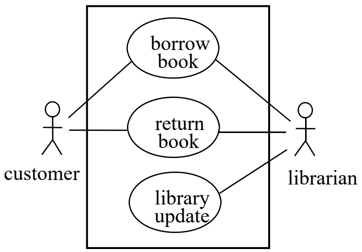
UML representation of inheritance



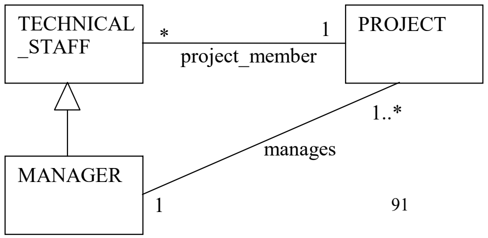
Abstraction yields models

It is then possible to reason about the system by reasoning about the model

UML use-case diagram: defines functions on basis of actors and actions

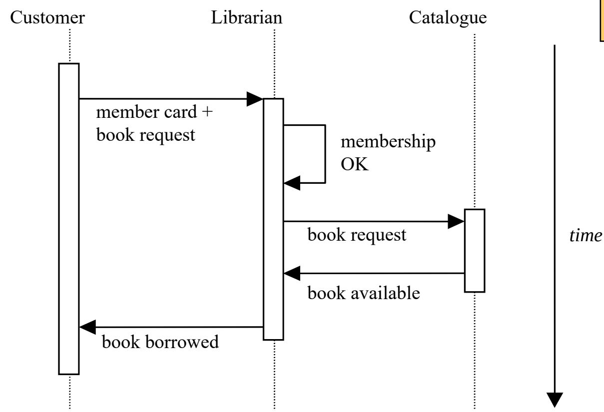


UML associations



UML sequence diagrams

- Describe how objects interact by exchanging messages
- Provide a dynamic view

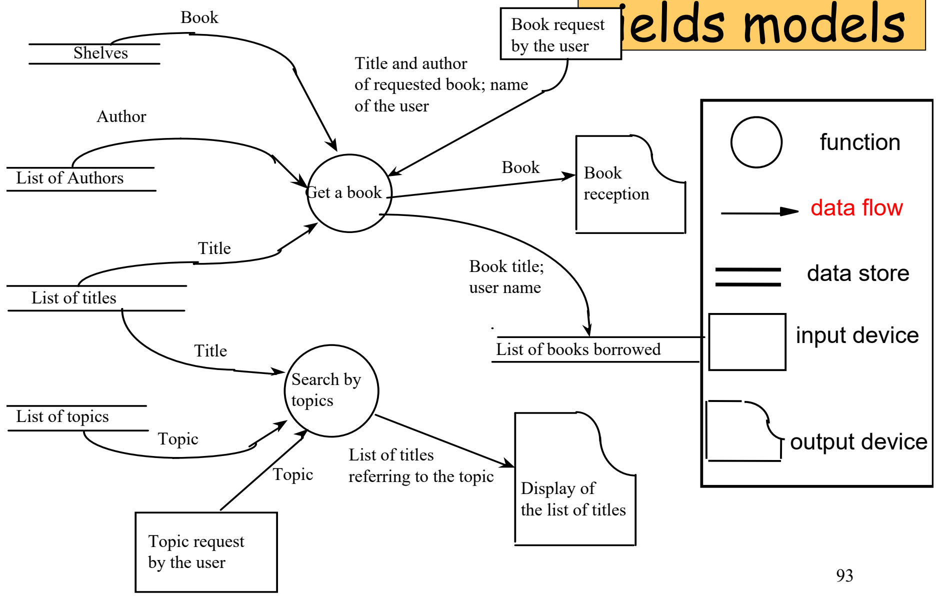


Abstraction yields models

It is then possible to reason about the system by reasoning about the model

DFD: A library example

Abstraction yields models

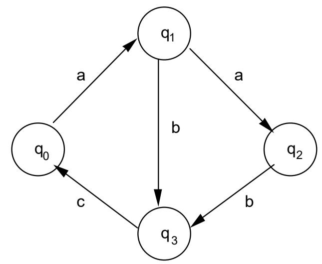


Abstraction yields models

Finite state machines (FSMs)

- Can specify control flow aspects
- Defined as

a finite set of states, Q;
a finite set of inputs, I;
a transition function d : Q x I → Q

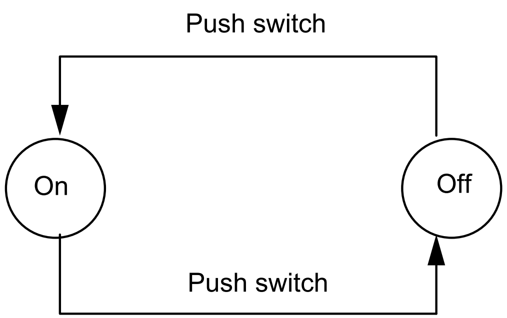


Examples?

Abstraction yields models

Finite state machines (FSMs)

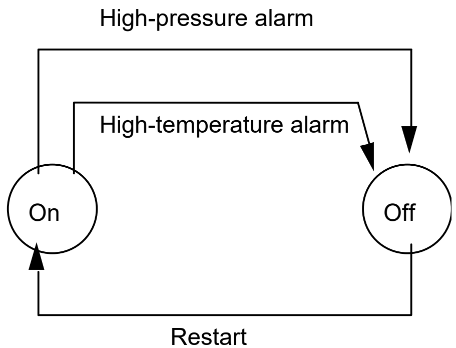
Example: a lamp



Abstraction yields models

Finite state machines (FSMs)

Example: a plant control system



Abstraction in process

- E.g. When we do cost estimation we only take some key factors into account
  - E.g. number of engineers and expected size of the final product, and then
  - extrapolate from the cost of previous similar projects, ignoring detail differences

Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- **Anticipation of change**
- Generality
- Incrementality

Anticipation of change

- Ability to support software evolution requires anticipating potential future changes
  - Correcting errors
  - Old requirements change
  - New requirements

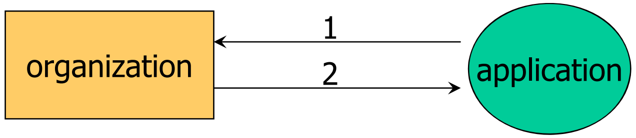
Why changes are unavoidable?

Anticipation of change

- Changes are unavoidable
  - In many cases, software is developed when its requirements are not entirely understood
  - Then after release, on the basis of feedback from the users, …

Anticipation of change

- Changes are unavoidable (cont.)
  – Applications are often embedded in an environment, such as an organizational structure
  – The environment is affected by the introduction of the application, and this generates new requirements that were not known initially



Anticipation of change

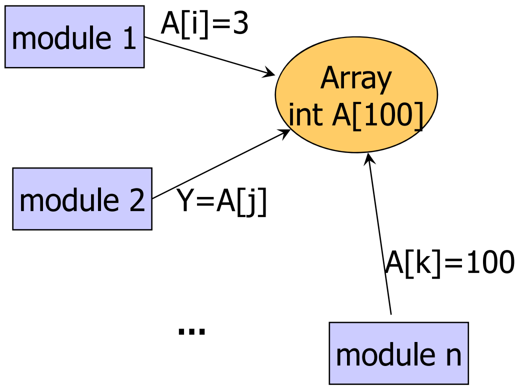
- Changes are unavoidable (cont.)
  – hw/sw environments always evolve
    - e.g. OS: DOS->Windows->Windows 95->Windows XP …
  – Anticipation of change is a principle that we can use to achieve __________.

Anticipation of change

- Changes are unavoidable (cont.)
  – hw/sw environments always evolve
    - e.g. OS: DOS->Windows->Windows 95->Windows XP …
  – Anticipation of change is a principle that we can use to achieve evolvability

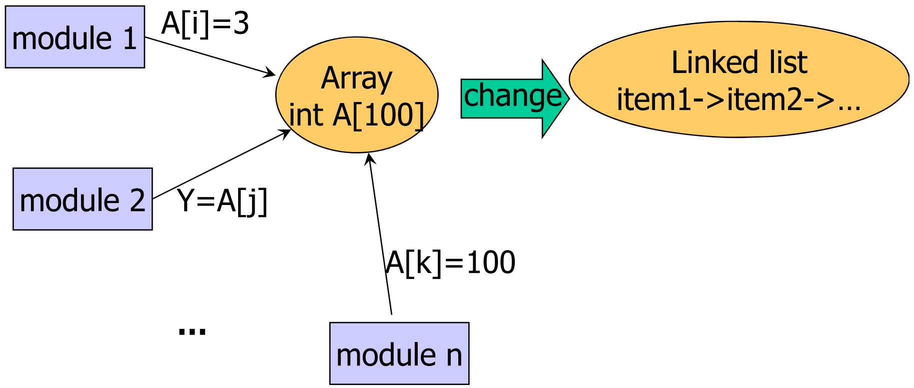
Anticipation of change

- E.g.



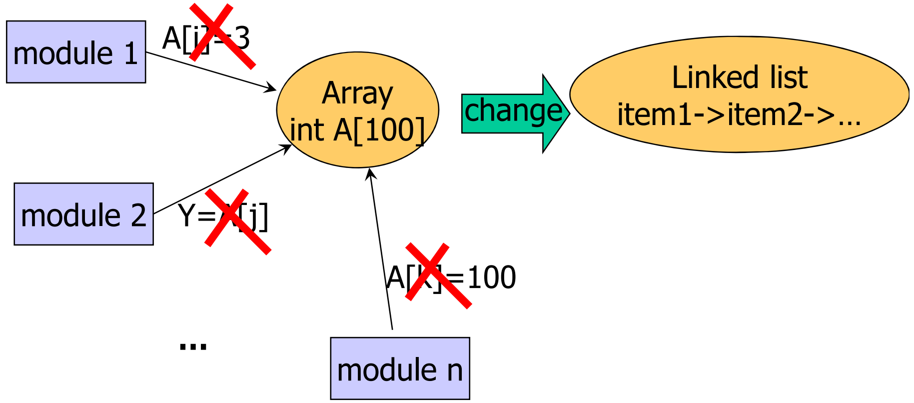
Anticipation of change

- E.g.



Anticipation of change

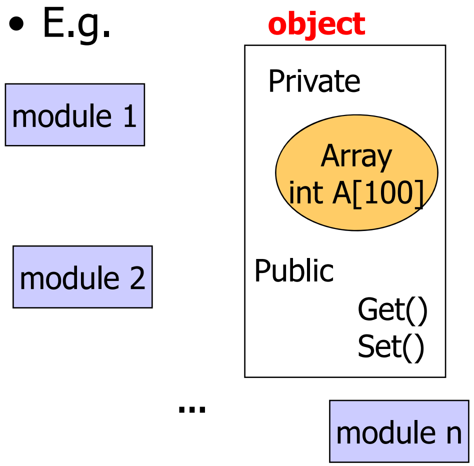
- E.g.



All modules have to change.

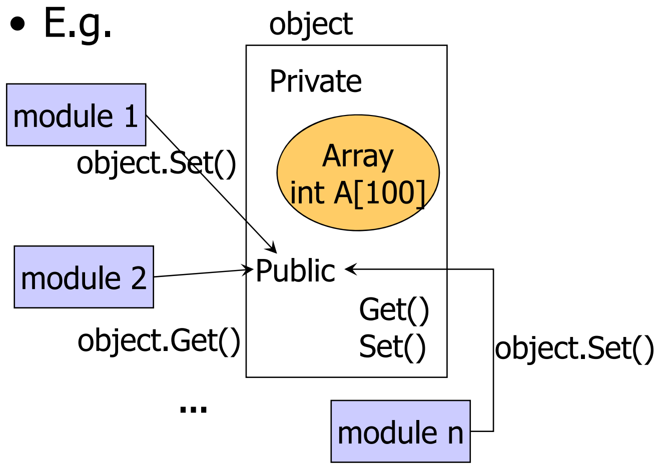
Anticipation of change

- E.g.



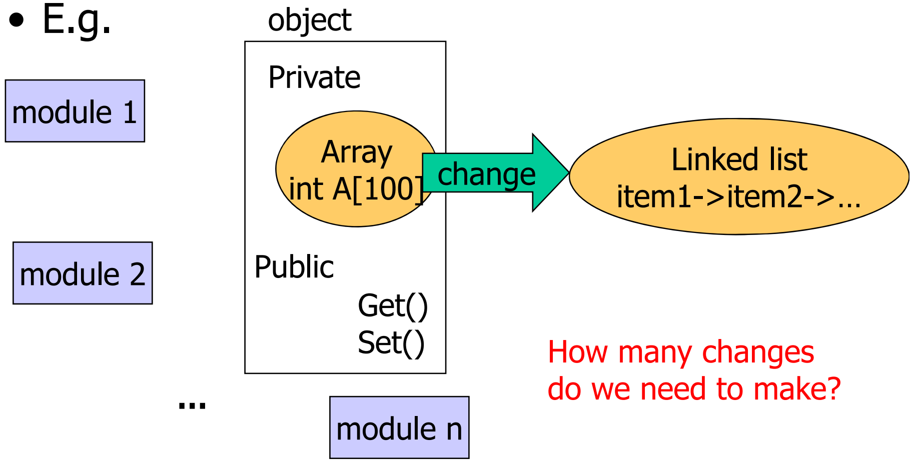
Anticipation of change

- E.g.



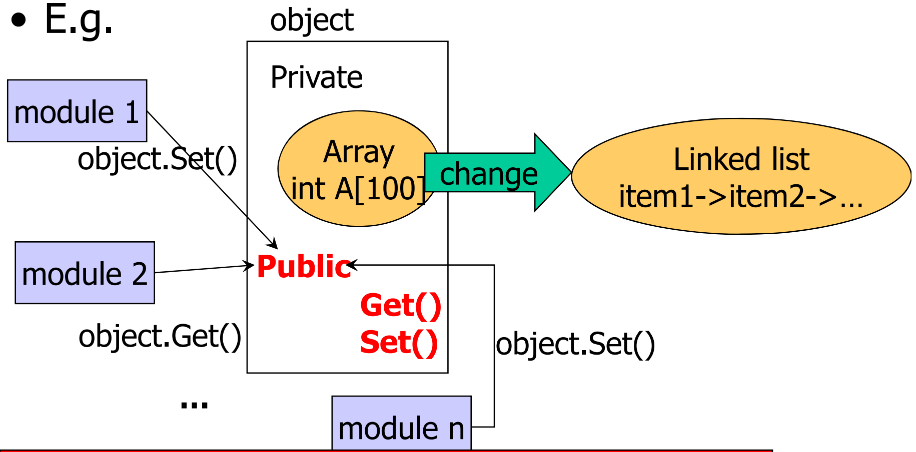
Anticipation of change

- E.g.



Anticipation of change

- E.g.



"information hiding":
We need only change object.Get() and object.Set().

Anticipation of change in process

- E.g.
  - Managers should anticipate the effects of changes in personnel, resources, etc
  - When designing life cycle of an application, it is important to take maintenance into account
  - Depending on the anticipated changes, managers must estimate costs and design the organizational structure that will support software evolution .

Exercise

- Take a sorting program from any textbook. Discuss the program from the standpoint of reusability. Does the algorithm make assumptions about the type of the elements to be sorted? Would you be able to reuse the algorithm for different types of elements?

  How would you modify the program to improve its reusability under these circumstances?

Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- **Generality**
- Incrementality

Generality

- To find the ith largest element in a set?
- Sorting?
- Developing a simple search engine to process txt files?

Generality

- While solving a problem, try to discover if it is an instance of a more general problem whose solution can be reused in other cases
- Advantages and disadvantages?

Generality

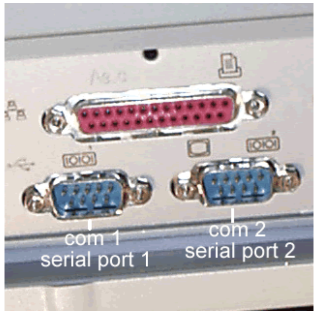
- While solving a problem, try to discover if it is an instance of a more general problem whose solution can be reused in other cases
- Carefully balance generality against performance and cost

Generality – example 1

- You are asked to merge two sorted files
  - Two files do not contain records with identical key values
- If you generalize the solution to accept identical key values, you provide a program with higher reusability

Generality – example 2

- You are asked to write a program to transfer data between computers through the serial port



Generality

- Allows us to develop general tools for the market
  - Spreadsheets, databases, word processors, …
- General trend in software
  - For every specific application area, general packages that provide standard solutions to common problems are increasingly available

Generality

- If the problem at hand can be restated as an instance of the general problem, it is often convenient to adopt the package instead of implementing a specialized solution
  - E.g. use macros to specialize a spreadsheet
  - E.g. writing compilers:
    - YACC - yet another compiler-compiler
    - LEX - generate programs for lexical tasks

    Read the source program and discover its structure .
  - E.g. file operations: use C or Shell ?

Exercise

- Suppose you are writing a program that manipulates files. Among the facilities you offer is a command to sort a file in both ascending and descending order. Among the files you manipulate, some are kept automatically sorted by the system. thus, you might take advantage of the fact: if the file is already sorted, you do not take any action; or you apply a reverse function if the file is sorted in the opposite order. Discuss the pros and cons of using such specialized solutions instead of executing the standard sort algorithm every time the sort command is issued.

Key principles

- Rigor and formality
- Separation of concerns
- Modularity
- Abstraction
- Anticipation of change
- Generality
- **Incrementality**

Incrementality

- Motivation
  - In most practical cases there is no way of getting all the requirements right before an application is developed.
  - Rather, requirements emerge as the application, or parts of it, is available for practical experimentation.

Examples?

Incrementality

- Process proceeds in a stepwise fashion (*increments*)
- Examples (process)
  - deliver subsets of a system early to get **early feedback** from users, then add new features incrementally
    - E.g. a subset of a computer game

Incrementality

- Examples (cont.)
  - deal first with functionality, then turn to performance
  - deliver a first prototype and then incrementally add effort to turn prototype into product
    - E.g. a Web site

Incrementality

- Can it turn into undisciplined development ?

Incrementality

- Evolutionary development requires special care in the management of documents, programs, test data, etc, developed for various versions.
  - Each incremental step must be recorded
  - Documents must be easily retrieved
  - **Changes** must be applied in a **controlled** way
- Otherwise it will quickly turn into undisciplined development and all advantages will be lost .

Exercise

- Discuss the relationships between generality and anticipation of change
- Discuss the relationships between generality and abstraction.