2AA4 / 2ME4 Course Notes | Farzan Yazdanjou
Note: Lecture headings are hyperlinked with the corresponding lecture slides.

Lecture 2

"Faked" Rational Design Process:

- Problem Statement
- Development Plan
- Requirements (SRS)
- Design Docs (MG and MIS)
- Code
- V&V Report

Embedded systems control physical systems and are rapidly appearing everywhere from cell phones to nuclear power plants.

Challenges for Engineers:

- Must design systems that have safe, correct, high-quality software.
- Produce software that they can guarantee.
- No silver bullets

Opportunities for Engineers:

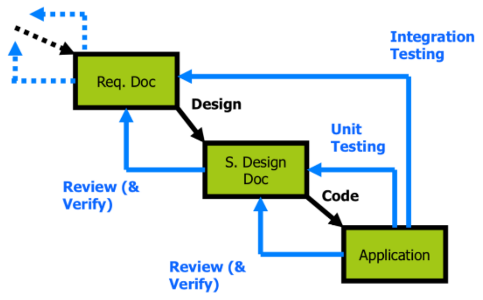
- Software tools can enhance engineers' capabilities.
- Software can increase effectiveness of devices engineers design.

Software Development Process:

A rational development process is needed to produce quality software. Any proposed rational process is necessarily an idealization as:

- Humans inevitably make errors
- Communication is imperfect
- Many things are not understood at the start
- Supporting tech always have limitations
- Requirements change over time

Software Lifecycle



Lecture 3 : Software Quality

Software Documentation:

Every software product should include documentation that presents the product to clients, reviewers, users, and maintainers.

It is useful to produce documentations that outlines a rational process (appear so).

Software Development Process:

- Requirements (SRS): What is the problem that needs to be solved? What are the product requirements that need to be satisfied?
- Design (MG, MIS): How will the problem be solved? How will the product requirements be satisfied?
- Implementation (Code): What is a solution to the problem? What is an executable implementation of the design?
- Verification (V&V): What behavior does the product exhibit? Is the behavior correct?
- Delivery and Maintenance: How will the product be delivered? What needs to be maintained? How will it be maintained?

Software Life Cycle Models

Waterfall model:

- Follows a logical order of the phases given above in a linear fashion.
- Is an idealization of the SDP that is rare.
- Potentially appropriate when requirements are well understood and slow to change.

Other Life Cycle Models:

- Refinement
- Incremental
- Spiral
- Prototyping

2AA4 / 2ME4 Course Notes | Farzan Yazdanjou

Note: Lecture headings are hyperlinked with the corresponding lecture slides.

Software Differs from other Engineer Products

- Intangible: Not physical, hard to visualize
- Malleable: Easy to modify (yet requires care)
- Human intensive: S99.9% engineering, 0.1% manufacturing. Software is essentially documentation.

Definition of Software Qualities

Measure of the excellence or worth of a software product (code or document) or process with respect to some aspect.

- Correctness
- Reliability
- Robustness
- Performance
- Verifiability
- Productivity
- Usability
- Understandability
- Maintainability
- Reusability
- Portability

User Satisfaction = Important Qualities are High + Within Budget

Must be two: Fast, Good, or Cheap.

Software Qualities

External vs Internal Software Qualities (Qs):

- External qualities are visible to the user
- Internal qualities are visible to the developer
- Internal Qs help external Qs be achieved

Product vs Process Qualities (Qs):

- Product Qs concern the product itself
- Process Qs concern how the product is developed
- Process Qs help product Qs be achieved
- Process qualities can also reduce dev. Costs
- Importance of particular software Q varies across software products – external Qs are not as important for embedded systems as for a desktop software.

Correctness vs Reliability vs Robustness:

Correctness: A software product is correct if it satisfies requirement specification. Correctness is extremely difficult to achieve because:

- The requirements specification may be imprecise, ambiguous, inconsistent, based on incorrect knowledge, or nonexistent.
- Requirements often compete.
- It is virtually impossible to produce "bug-free" software.
- It is very difficult to verify or measure correctness.

If the requirements specification is formal, correctness can in theory and possibly in practice be:

- Mathematically defined
- Proven by mathematical proof
- Disproven by counterexample

Reliability: A software product is reliable if it actually does what it is intended to do. Correctness is an absolute quality, while reliability is a relative quality.

- A software product can be both reliable, and incorrect.
- Reliability can be statistically measured.
- Software products are usually much less reliable than other engineering products.

Robustness: A software product is robust if it behaves reasonably even in unanticipated or exceptional situations.

- Missing file, incorrect URL, unanticipated views of the data

A correct software product does not necessarily need to be robust. Robust programs are not necessarily correct, either. Correctness is accomplished by satisfying requirements. Robustness is accomplished by satisfying unstated requirements.

Lecture 4 : Software Quality

Software engineering is quantifiable. A requirements specification is necessary to assess correctness.

All correct programs are reliable, but not all reliable programs are robust - reliability and correctness are defined with respect to the requirements, while robustness is how the software behaves for situations not anticipated in the requirements. A correct program is not necessarily robust.

Aspects of Performance:

- Speed, Memory, Throughput, etc.

Measure Software Performance:

- Computation time, big O notation, etc.

One can specify performance requirements to make them unambiguous and verifiable with:

- Absolute measured – unambiguous, but often difficult to verify for all inputs.
- Relative measured for specific test cases/benchmarks – unambiguous and verifiable.
- Consider the real requirement – maybe safety? Maybe outperforming the competition?

Performance: Performance of a product is related to external quality requirements for speed and storage. Efficiency is an internal (implementation) quality related to the use of computer resources (memory, time, communication). Thus, efficiency affects performance.

Performance Evaluation:

- Empirical measurements
- Analysis of an analytical model
- Analysis of a simulation model

Poor performance often negatively affects the usability and scalability of the product. Performance requirements can often be stated as relative comparisons to existing products.

Standardization

Importance of standardization varies (Christopher Schankula): For example, having a slightly different sequence of buttons on a microwave is annoying, but you figure it out pretty quickly, only costing you a little bit of time. Standardization would be nice but it's not life-or-death. Standardization on how to operate something safety-critical is much more important for helping to ensure that these things are used safely by the user. So standardization has a varying scale of importance from minorly more efficient use to saving lives.

Usability: The usability of a software product is the ease with which a typical human user can use the product.

- Depends strongly on the capabilities and preferences of the user.
- The user interface of a software product is usually the principle factor affecting the product’s usability.
- Standardization can improve usability.
- HCI is a major subject concerned with understanding and improving interaction between humans and computers.

Verifiability: The ease with which a software product’s properties (such as correctness and performance) can be verified. Often an internal quality, but it can be an external quality.

Maintainability: The ease with which a software product can be modified after its initial release. Maintenance costs can exceed 60% of the total cost of the software product.

Three main categories of maintenance:

- Corrective: Modifications to fix residual and introduced errors.
- Adaptive: Modifications to handle changes in the environment in which the product is used.
- Perfective: Modification to improve the qualities of the software.

Two qualities of maintenance:

- **Repairability**: The ability to correct defects.
- **Evolvability**: The ability to improve the software and to keep it current.

Developers promote maintainability through:

- Documentation
- Traceability
- Separation of concerns
- Modularity
- Design for change
- Design for generality

Reusability: A software product or component is reusable if it can be used to create a new product. Reusability is a bigger challenge in SE than other Engs. Reuse comes in two forms:

- Standardized, interchangeable parts
- Generic, instantiable components

Portability: A software product is portable if it can run in different environments. The environment for a software product includes the hardware platform, the operating system, the supporting software, and the user base.

Portability is often crucial to the success of a product as environments are constantly changing. Some software such as operating systems and compilers are inherently machine specific.

Understandability: The ease with which the requirements, design, implementation, documentation, etc. of a product can be understood.

- Is an internal quality that has an impact on other qualities such as verifiability, maintainability, and reusability.
- Often a tension between understandability and the performance of a product.
- Some useful software products completely lack understandability.

Special Qualities:

- Information systems require information security (privacy, integrity).
- Safety is an important quality of real-time systems that respond to external events.
- Accuracy, reliability, and performance is important for scientific computing. Often sacrificing maintainability and portability for performance. Verifiability is difficult to achieve.
- Usability and other human-oriented qualities are not as important for embedded systems.

Measurement of Quality: A software quality is only important if it can be measured – otherwise, there is no basis for claiming improvement. A software quality must be precisely defined be it can be measured. Most qualities do not have universally accepted measures.

You cannot directly measure maintainability. You can measure it by:

- Modularity of code
- Lines of code
- Cyclomatic complexity
- Documentation
- Traceability, etc.

High Quality Documentation: To achieve external qualities for documentation, there are some agreed on internal qualities that can more likely be directly measured:

- Complete
- Consistent
- Modifiable
- Traceable
- Unambiguous
- Correct
- Verifiable
- Abstract

Interoperability: A software system is interoperable if it can work with other systems. A software product is an open system if parts of the systems – such as interface specs, protocols, and source code – are available to the public. Open systems tend to be more interoperable than non-open systems.

Productivity: The productivity of a software development process is the measure of how efficiently the process produces software.

- Highly depends on the skills and organization of the development team.
- Very hard to measure.
- The number of lines of code per unit time is a terrible measure of productivity.
- Can be greatly increased using development tools, environments, and methods.
- Software reuse decreases productivity in the short term but increases productivity in the long term.

Timeliness: The timeliness of a SDP is the ability to deliver a product on time. Timeliness is difficult to achieve in software development.

Standard project management techniques are difficult to apply to software engineering as:

- It is difficult to define the requirements for software.
- It is difficult to quantify software.
- Requirements for software tend to continuously change as the project progresses.
- Incremental delivery is one technique for achieving timeliness.

Visibility: A SDP is visible if the steps of the process and the product itself are documented. The documentation should be accessible to the whole dev team as well as to management.

Benefits of visibility:

- Promotes communication
- Facilitates planning
- Protects against personnel changes

Lecture 5 : Principles

A **principle** is a general concept that is widely applicable in software engineering.

Methods are:

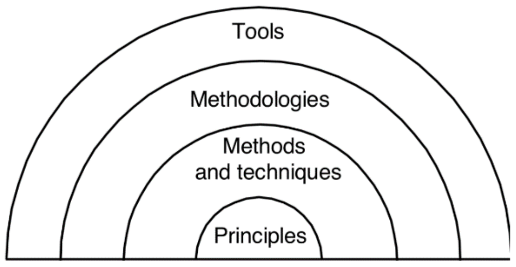
- General guidelines to govern the execution of an activity
- Rigorous, systematic and disciplined approaches
- Example – following a template.

Techniques:

- Provides an approach to develop software, but they are more technical and mechanical than methods.
- Techniques have a more restricted applicability than methods.
- Example: Hoare triple for correctness proof.

A **methodology** is a coherent collection of methods and techniques.

A **tool** is a device that supports the application of a method, techniques, or methodology.



Software Engineering Principles:

Used to reduce complexity. Form the basis for methods, techniques, methodologies, and tool.

- Can be used in all phases of software dev.
- Can be applied to both process and product.
- Purpose is to improve quality, with a special emphasis on reliability and evolvability.
- Are also key principles of mathematics and engineering as a whole.
- SE more explicit in identifying and using principles than other Eng branches.

Key Principles:

- Rigour
- Formality
- Separation of Concerns
- Modularity
- Abstraction
- Anticipation of Change
- Generality
- Incrementality

Rigour: An argument is valid if the conclusion is a logical consequence of the premise. Rigour is precise reasoning characterized by:

- Only unambiguous language is used
- There are no hidden assumptions
- Care is taken to ensure that all arguments are valid.

Achieved through mathematics and logic. Should be systematically employed throughout the SDP.

Formality: Reasoning in a formal system consisting of:

- A language with a formal syntax and a precise semantics.
- A set of syntactic rules.

A formal system enables reasoning to be mechanized:

- Reasoning is performed mechanically with computer assistance.
- Arguments are machine checked.
- Parts of the reasoning are automated.

The use of formality in SD has a high cost:

- The learning curve is very high.
- Tool support and knowledge bases are inadequate.
- The amount of detail involved is often overwhelming.

Formality over Rigour:

- Advantage: Unambiguous, can check for consistency, basis of mechanization.
- Disadvantage: Understandability, high cost.

Every software development project uses at least one formal language – a programming language.

A typical first year mathematics textbook (without software) is not a formal presentation of Calculus. It is rigorous but not formal (could not teach it to a computer as given).

Separation of Concerns

The principle that different concerns should be isolated and considered separately.

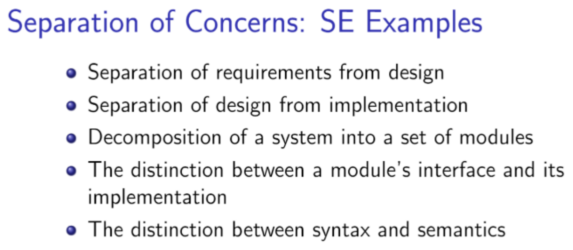
- Goal is to reduce a complex problem to a set of simpler problems.
- Enables parallelization of effort.

Concerns can be separated in various ways:

- Different concerns are considered at different times.
- Software qualities are considered separately.
- A software system is considered from different views.
- Parts of a software system are considered separately.

Dangers include:

- Opportunities for global optimizations may be lost.
- Some issues cannot be safely isolated (eg, security).



Separation of Concerns: SE Examples

- Separation of requirements from design
- Separation of design from implementation
- Decomposition of a system into a set of modules
- The distinction between a module's interface and its implementation
- The distinction between syntax and semantics

Modularity

A modular system is a complex system that is divided into smaller parts called modules. Modularity enables the principle of separation of concerns to be applied in two ways:

- Different parts of the system are considered separately.
- The parts of the system are considered separately from their composition.

Modular decomposition is the top-down process of dividing a system into modules. A "divide and conquer" approach.

Modular composition is the bottom-up process of building a system out of modules. An "interchangeable parts" approach.

Lecture 6 : Principles

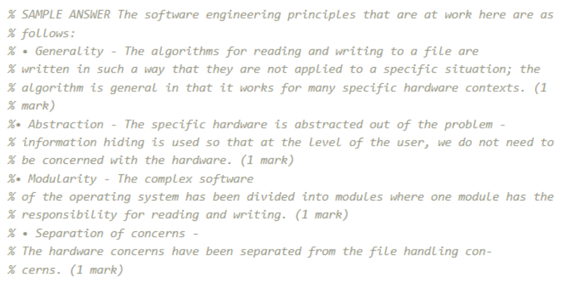
Properties of Good Modules:

- High cohesion: The components of the module are closely related.
- Low coupling: The module does not strongly depend on other modules.

This allows the modules to be treated in two ways: as a set of interchangeable parts, or as individuals.

Without coupling, every module would have to provide all of its own services – need **some** communication. As such, **zero** coupling is not ideal.

Sample question about USBs (generality, abstraction, modularity, and sep. of concerns):



```
% SAMPLE ANSWER The software engineering principles that are at work here are as
% follows:
% • Generality - The algorithms for reading and writing to a file are
% written in such a way that they are not applied to a specific situation; the
% algorithm is general in that it works for many specific hardware contexts. (1
% mark)
%• Abstraction - The specific hardware is abstracted out of the problem -
% information hiding is used so that at the level of the user, we do not need to
% be concerned with the hardware. (1 mark)
%• Modularity - The complex software
% of the operating system has been divided into modules where one module has the
% responsibility for reading and writing. (1 mark)
% • Separation of concerns -
% The hardware concerns have been separated from the file handling con-
% cerns. (1 mark)
```

Abstraction

The process of focusing on what is important while ignoring what is irrelevant. Abstraction is a special case of separation of concerns.

Produces a model of an entity in which the irrelevant details of the entity are left out:

- Many different models of the same entity can be produced by abstraction.
- Abstraction models differ from each other by what is considered important and what is considered irrelevant.
- Repeated application of abstraction produces a hierarchy of models.

Refinement is the opposite of abstraction.

Over-abstraction produces models that are difficult to understand because they are missing so many details.

Requirement specification should be abstract – to provide flexibility for the designers.

Anticipation of Change

The principle that future change should be anticipated and planned for. Also called “design for change”. Techniques include:

- Configuration management: Manage the configuration of the software so that it can be easily modified as the software evolves.
- Information hiding: Hide the things that are likely to change inside of modules.
- Little languages: Create little languages that can be used to solve families of related problems.

Crucial principle for the SDP as software is constantly changing. Also as an anticipation of change; important knowledge should not be with just one person – make documentation.

Generality

The principle of generality is to solve a more general problem than the problem at hand whenever possible.

- The more general a solution is, the more likely that it can be reused.
- Sometimes a general problem is easier to solve than a specific problem.
- A general solution may be less efficient than a more specific solution.
- A general problem may cost more to solve than a more specific problem.

Abstraction is often used to extract a general solution from a specific solution.

Incrementality

The principle of incrementality is to attack a problem by producing successively closer approximations to a solution. Enables the development process to receive feedback and the requirements to be adjusted accordingly.

Techniques for incremental software dev:

- Rapid prototyping: Produce a prototype that is “thrown away” later.
- Refinement: A high-level artifact is incrementally refined into a low-level artifact.

In modular design, abstraction, anticipation of change, and separation of concerns principles are used.

Lecture 7 : Module Intro

Design refers to:

- Activity
  - Bridge between requirements and implementation
  - Structure of the activity
- Result of activity
  - System decomposition into modules (model guide)
  - Module interface specification (MIS)

Design for change: Designers tend to concentrate on current needs. Special effort needed to anticipate likely changes. Changes can be in the design or in the requirements. Too expensive to design for all changes, but should design for likely ones.

Product families: Think of the current system under design as a member of a program family. Analogous to product lines in other engineering disciplines. Examples include cellphones, cars, etc. Design the whole family as one system, not each family member separately.

Examples of likely software changes:

- Algorithms – like replacing inefficient sorting algorithm with a more efficient one.
- Change of data representation
  - From BT to Threaded Tree
- Change of underlying abstract machine
  - New release of OS
- Change of peripheral devices
- Change of "social" environment
  - New tax regime
  - New language
  - Corresponds to requirement changes
- Changes due to dev process (prototype transformed into product)

Product Families are different versions of the same system. For example, a family of mobile phones – commonalities include use of communication, interface, screen, keyboard, etc. Another example would be reservation system – commonalities include reserving something for a period of time, user interface, etc. Context varies from hotel, school, office.

Should anticipate definition of all family members. Identify what is common to all of them and delay decisions that differentiate among different members.

Components of a Module:

- An interface that enables the module's clients to use the service the module provides.
- An implementation of the interface that provides the services offered by the module.

The Module Interface

Can be viewed as:

- A set of services
- A contact between the module and its clients
- A language for using the module's services

The interface is exported by the module and imported by the module's clients. An interface describes the data and procedures that provide access to the services of the module.

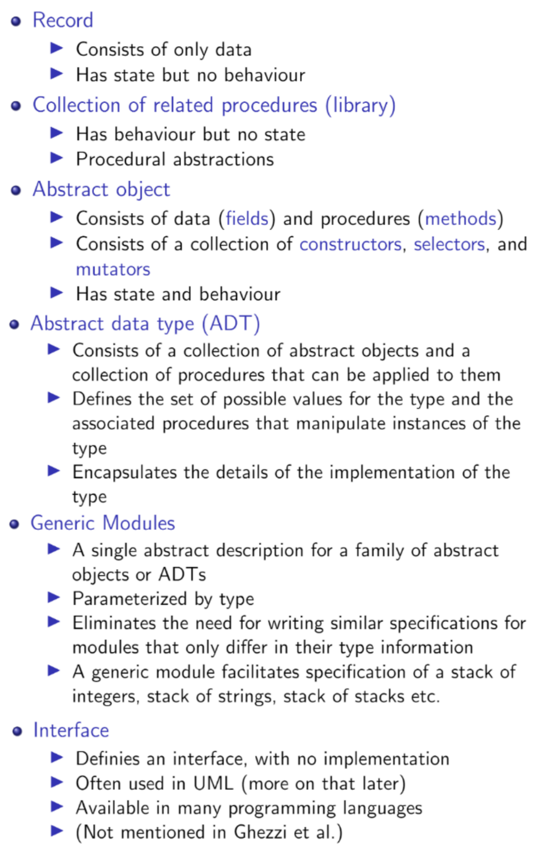
The Module Implementation: An implementation of the module's interface. The implementation is hidden from other modules. The interface data and procedures are implemented together and may share data structures. The implementation may utilize the services offered by other modules.

Information Hiding

- Basis for design (that is modular decomposition).
- Implementation secrets are hidden from clients.
- Secret can be changed freely if the change does not affect the interface.

Encapsulate changeable design decisions as implementation secrets within module implementation.

Examples of Modules:



- Record
  - Consists of only data
  - Has state but no behaviour
- Collection of related procedures (library)
  - Has behaviour but no state
  - Procedural abstractions
- Abstract object
  - Consists of data (fields) and procedures (methods)
  - Consists of a collection of constructors, selectors, and mutators
  - Has state and behaviour
- Abstract data type (ADT)
  - Consists of a collection of abstract objects and a collection of procedures that can be applied to them
  - Defines the set of possible values for the type and the associated procedures that manipulate instances of the type
  - Encapsulates the details of the implementation of the type
- Generic Modules
  - A single abstract description for a family of abstract objects or ADTs
  - Parameterized by type
  - Eliminates the need for writing similar specifications for modules that only differ in their type information
  - A generic module facilitates specification of a stack of integers, stack of strings, stack of stacks etc.
- Interface
  - Definies an interface, with no implementation
  - Often used in UML (more on that later)
  - Available in many programming languages
  - (Not mentioned in Ghezzi et al.)

2AA4 / 2ME4 Course Notes | Farzan Yazdanjou
Note: Lecture headings are hyperlinked with the corresponding lecture slides.

Key Points:

- One module, one secret
- Secrets are often nouns (algorithm, hardware, data structure, etc.)
- Secretes are sometimes phrased with “How to…”
- Secrets ideally will have a one to one mapping with the anticipated changes for the software.

Lecture 8 : Math for MIS

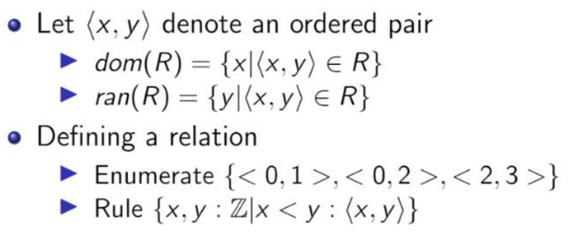
A **set** is an unordered collection of elements.

A **binary** relation is a set of ordered pairs.

A **function** is a relation in which each element in the domain appears exactly once as the first component in the ordered pair.

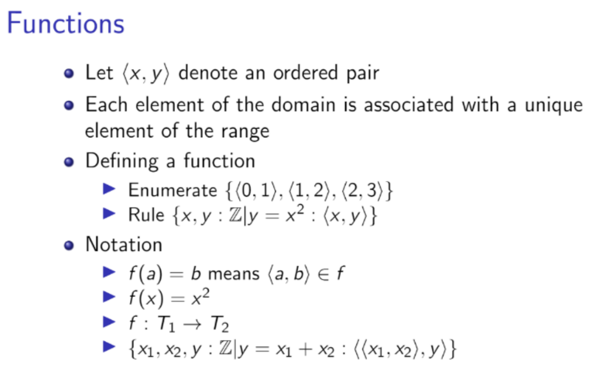
Set example: $S = \{x : Z | x \leq 100 : x^2\}$

Relation examples:



- Let $\langle x, y\rangle$ denote an ordered pair
  - $dom(R) = \{x | \langle x, y\rangle \in R\}$
  - $ran(R) = \{y | \langle x, y\rangle \in R\}$
- Defining a relation
  - Enumerate $\{<0,1>, <0,2>, <2,3>\}$
  - Rule $\{x, y : \mathbb{Z} | x < y : \langle x, y\rangle\}$

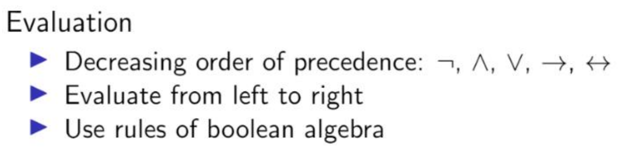
Function examples:



Functions

- Let $\langle x, y\rangle$ denote an ordered pair
- Each element of the domain is associated with a unique element of the range
- Defining a function
  - Enumerate $\{\langle 0,1\rangle, \langle 1,2\rangle, \langle 2,3\rangle\}$
  - Rule $\{x, y : \mathbb{Z} | y = x^2 : \langle x, y\rangle\}$
- Notation
  - $f(a) = b$ means $\langle a, b\rangle \in f$
  - $f(x) = x^2$
  - $f : T_1 \rightarrow T_2$
  - $\{x_1, x_2, y : \mathbb{Z} | y = x_1 + x_2 : \langle\langle x_1, x_2\rangle, y\rangle\}$

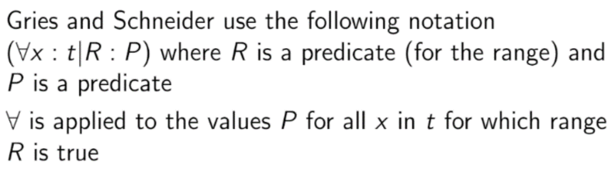
Logic: A logical expression is a statement whose truth values can be determined.



Evaluation

- Decreasing order of precedence: $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$
- Evaluate from left to right
- Use rules of boolean algebra

Quantifiers: Variables are often used inside logical expressions. Variables have types. A type is a set of values from which the variable can take its value. Often quantify a logical expression over a given variable:



Gries and Schneider use the following notation $(\forall x : t | R : P)$ where R is a predicate (for the range) and P is a predicate

$\forall$ is applied to the values P for all x in t for which range R is true

Types, Sets, Sequences, Tuples are not included here and can be found in the lecture slides.

Lecture 9 : MIS

Overview of MIS:

The MIS precisely specifies the module’s observable behavior – what the module does.

The MIS does not specify the internal design.

The idea of an MIS is inspired by the principles of software engineering.

Advantages:

- Improved many software qualities.
- Programmers can work in parallel.
- Assumptions about how the code will be used are recorded.
- Test cases can be decided on early, and they benefit from a clear specification of behavior.
- A well designed and documented MIS is easier to read and understand than complex code.
- Can use the interface without understanding details.

MIS template covered in the hyperlinked PDF.

Lecture 10 : ADTs

An abstract object is a module where there is only one instance (singleton pattern).

- The MIS is more abstract than the code
- The MIS works with mathematical and CS concepts that may not exist in the given programming language
- The MIS may use symbols that are not available in the programming language
- Not all languages support the object-oriented and/or functional programming paradigms.

Types for Access Programs

- The translation of the types of the inputs and outputs should be unambiguous.
- 1: Types should have a trivial translation to a programming language (like integer, real, Boolean, etc.)
- 2: Or the type should be defined in another module (found via the uses clause).
- A sequence type is okay, since we assume it will translate to a list or array type.
- In cases of potential ambiguity, instructions can be given to the programmer.
- Semantics section uses a combination of mathematical types and types with defined interfaces.

Lecture 11 : Functional Programming

Functional Programming:

Computation is treated as the evaluation of mathematical function (not code subroutines):

- No state
- No mutable data
- Programming with expressions, not statements
- Easier to reason about than imperative or object oriented code.

Functions are a first order data type:

- Can pass functions as arguments
- Can return functions

Python is dynamically typed – cannot inspect the type of a function

List Comprehension:

- Modified version: $\{x : T | R \wedge P : E\}$
  - ► P is a predicate (filter)
- Python code: `[E for x in R if P]`
  - ► R is a sequence (list)

A **filter** is a function that takes a predicate and a list and returns the list of elements that satisfies the predicate.

Lecture 12 : Functional Programming

Reduce takes a binary function and applies it to the first two elements. Takes the result of that and uses it in the binary function along with the next element.

All list comprehensions can be implemented via for loops. Not all for loops can be implemented by a list comprehension. List comprehensions can be implemented by maps and filters.

Lecture 13 : Modules & External Interaction

- Records: Defines types – spec has no state.
- Libraries: Defines functions – spec has no state.
- Abstract Object: Not template module – spec has state.
- Abstract Data Types: Template module – spec has state.
- Generic Module Types: Generic abstract object or ADT – spec has generic parameter.
- Interface: Spec has no semantics.
- Inheritance: Spec has inheritance.

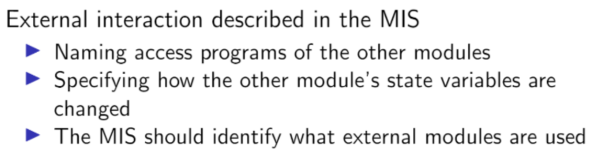
Modules with External Interaction:

Some modules may interact with the environment or other modules. Environment might include the keyboard, the screen, the file system, motors, sensors, etc.

Sometimes, the interaction is informally specified using natural language (prose). You can also introduce an environment variable that has a name, type, and interpretation.

Environment variables include the screen, the state of a motor, the position of a robot, etc.

Some external interactions are hidden, meaning they are present in the implementation but not in the MIS.



External interaction described in the MIS
- Naming access programs of the other modules
- Specifying how the other module's state variables are changed
- The MIS should identify what external modules are used

Lecture 14 : Generic MIS

The distinction between abstract objects and ADTs is that for the life of the program there will only be one abstract object for a given module, but for an ADT there could be any number of instances of modules that follow the module template, only distinguished from another by the values of their state variables.

Uses for Abstract Objects:

- Creating a single abstract object corresponds to the singleton design pattern.
- Provides "global" variables.
- It is a shared resource, meaning you do not have to pass an object reference.
- Problematic for testing if high coupling and frequent state changes – since many test cases will depend on the state of the abstract object.

Interfaces prevent potential abuse:

- Provide no mutators in the interface, just a constructor and selectors.
- Assume information hiding will be respected.
- Assume no other references to state variables.

A more robust implementation would have state variables storing copies of objects rather than references using Python's copy library.

Generic Modules

Rather than duplicating modules for different types, we can parameterize one generic module with respect to type(s).

Advantages:

- Eliminate chance of inconsistencies between modules.
- Localize effects of possible modifications.
- Reuse

Lecture 15 : Object Oriented Design (OOD)

The following types of modules are ordered by increasing abstraction: records, abstract objects, abstract data type, generic abstract data types.

Generic modules are implemented in Python via dynamic typing. However, nothing special needs to be done to implement Generic ADTs or Generic Abstract Objects because:

- Python is a dynamically typed language.
- Python uses Duck typing.

Record:

- Just data, not behavior
- Could document as an ADT with a tuple for the state variable or getters and setters for each field.
- They are more work than necessary since programming languages provide records.

In the MIS, define the syntax of the **interface**, not the semantics.

Object Oriented Design:

- A class exports operations (procedures) to manipulate instant objects (methods).
- Instance objects accessible via references.
- Can have multiple instances of a class (can roughly be thought of as a type).

In python, all classes inherit the "object" class.

Inheritance:

Inheritance is used to show a module must implement the given interface.

- A way of building software incrementally.
- Useful for long lived applications because new features can be added without breaking the old applications.
- A subclass defines a subtype.
- A subtype is substitutable for the parent type.

Polymorphism: A variable referring to type "A" can refer to an object of type "B" if "B" is a subclass of "A".

Dynamic Binding: The method invoked through a reference depends on the type of the object associated with the reference at runtime.

- All instances of the sub-class are instances of the super-class, so the type of the sub-class is a subtype.
- Eg: All instances of AdminStaff and TechnicalStaff are instances of Employee.

Languages like C and Java use static type checking. Object oriented languages use dynamic type checking as the default:

- Types are known at compile time
- The class of an object may be known only at run time.

Inheritance is represented by UML (**Unified Modelling Language**) notation which is the standard notation for representing Object Oriented code. Classes are described by boxes.

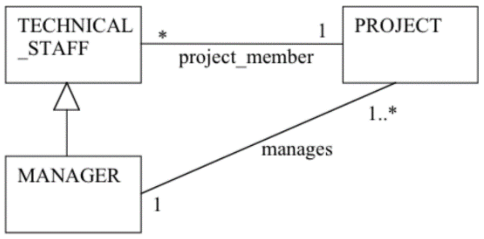
Class diagrams are close to code since syntax of methods is closer to actual syntax.

>> What information the MIS and Class Diagram have in common: Access program names and state variables.

>> What information the MIS adds: Semantics and exceptions (usually).

>> What information the Class Diagram adds: A richer set of uses relations.

UML Associations: Associations are relations that the implementation is required to support. They can have multiplicity constraints.



Lecture 16 : Module Interface Design

Assumptions versus Exceptions:

The assumptions section lists assumptions the module developer is permitted to make about the programmer's behavior.

- Assumptions are expressed in prose (natural language).
- Use assumptions to simplify the MIS and to reduce the complexity of the final implementation.
- Interface design should provide the programmer with a means to check so that they can avoid exceptions.
- When an exception occurs no state transitions should take place, any output is *don't care*.

2AA4 / 2ME4 Course Notes | Farzan Yazdanjou
Note: Lecture headings are hyperlinked with the corresponding lecture slides.

Exception Signaling

- Useful to think about exceptions in the design process.
- Caused by errors made by programmers, not by users.
- Exceptions will be particularly useful during testing.

Will need to decide how exception signaling will be done:

- A special return value, a special status parameter, a global variable.
- Invoking an exception procedure.
- Using built-in language constructs.

Quality Criteria

Consistent:

- Name conventions
- Ordering of parameters in argument lists
- Exception handling, etc.

Essential: Omit unnecessary features.

General: Cannot always predict how the module will be used.

As implementation independent as possible.

Minimal: Avoid access routines with two potentially independent services.

High Cohesions: Components are closely related

Low Coupling: Not strongly dependent on other modules.

Opaque: Information hiding

Check available so programmer can avoid exceptions.

If you find that an interface is not minimal, it should **not** always be fixed so that it is minimal.

The use of functional programming makes the interface more general.

Modular Decomposition:

We need to decompose the system into modules, assign responsibilities to those modules and ensure that they fit together to achieve our global goals.

We need to produce a software architecture – the architecture is summarized in a Software Design Document. Parnas version is the Module Guide (MG).

Lecture 17 : Modular Decomposition

Software Architecture

Shows gross structure and organization of the system to be defined.

Its description includes the description of:

- Main components of the system.
- Relationship among those components.
- Rationale for decomposition into its components.
- Constraints that must be represented by any design of the components.

Guides the development of the design.

Specific Techniques for Design for Change

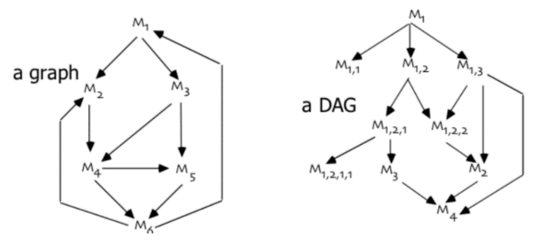
- Anticipate definition of all family members
- Identify what is common, delay decisions that differentiate family members.
- Configuration constants: Factor constant values into symbolic constants.

Conditional complication: Compile time binding, works well when there is a preprocessor

Relationships between modules: Uses, inheritance, association, aggregation, composition. Desire properties between these relations: hierarchy and low coupling.

Uses relation: Inheritance, Association, and Aggregation imply “uses”. A uses B means that A requires the correct operation of B. This relation is “statically” defined.

Relations can be represented as graphs. A hierarchy is a DAG (Directed Acyclic Graph):



Desired Properties:

- USES should be a hierarchy:
    - Hierarchy makes software easier to understand.
    - We can proceed from the leaf nodes upwards.
    - They make software easer to build.
    - They make software easer to test.
- Low coupling

Fan-in is considered better than Fan-Out:

- Fan-In means many modules use one module. Implies reuse.
- Fan-Out means one module uses many modules. Implies that all used modules need to be implemented to run the module.

A DAG is not a tree. A tree is a DAG. You would NOT prefer your uses relation to be a tree.

Hierarchy

- Organizes the modular structure through levels of abstraction.
- Each level defines an abstract (virtual) machine for the next level.

If there is only one place where two modules are used, it does not mean the uses relation depends on the dynamic execution of the program. The uses relation is static, it should include both modules.

A USES relation is **not** a DAG if there is a cycle because of the association relations.

Product Families

Attempt to recognize modules that will differ in implementation between family members.

New program family member should start at the documentation of the design, not with the code.

Decomposition by secrets, not by sequence of steps.

Prototyping

Once an interface is defined, implementation can done.

- You may even have prototypes that "fake" behavior.
- Create the first quickly but inefficiently, then progressively turn into the final version.
- Initial version acts as a prototype that evolves into the final product.

Lecture 18 : MG

Upside Down

Advantage: Does have Fan-In

Disadvantages:

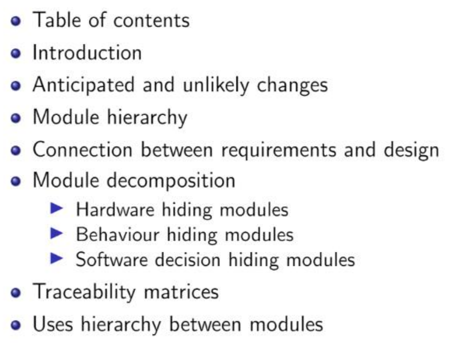
- Still limited Fan-In
- Would like multiple modules to provide services at the bottom level, not just one.
- Confusing since most modules are at the top level.
- Design cannot be viewed from the top down.

Module Guide:

- When decomposing the system into modules, we need to document the module decomposition so that develops and other readers can understand and verify the decomposition.
- Helps future maintainers find appropriate module.
- Decomposition is usually 3-5 steps deep

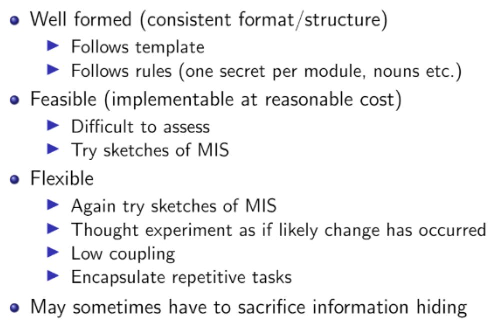
- The MG consists of a table that documents each module's service and secret.
- Conceptual modules will have broader responsibilities and secrets.
- Following a particular branch, the secrets at lower levels "sum up" to the secret at higher levels.
- Only the leaf modules (that represent code and have more precise services and secrets) are actually implemented.
- The MG should list the likely and unlikely changes on which the design is based.

MG Template:



- Table of contents
- Introduction
- Anticipated and unlikely changes
- Module hierarchy
- Connection between requirements and design
- Module decomposition
  - Hardware hiding modules
  - Behaviour hiding modules
  - Software decision hiding modules
- Traceability matrices
- Uses hierarchy between modules

Verification:



- Well formed (consistent format/structure)
  - Follows template
  - Follows rules (one secret per module, nouns etc.)
- Feasible (implementable at reasonable cost)
  - Difficult to assess
  - Try sketches of MIS
- Flexible
  - Again try sketches of MIS
  - Thought experiment as if likely change has occurred
  - Low coupling
  - Encapsulate repetitive tasks
- May sometimes have to sacrifice information hiding

List of Possible Anticipated Changes:

- The specific hardware on which the software is to run.
- The format of the initial input data.
- The format of the input parameters.
- The constraints on the input parameters.
- The format of the output data.
- The constraints on the output results.