Introduction to Software Engineering

Neelam Gupta

The University of Arizona
Department of Computer Science

Definitions

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

- Application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (IEEE 1990)
- Multi-person construction of multi-version software (Parnas 1978)

History

- The field of software engineering was born in 1968 in response to chronic failures of large software projects to meet schedule and budget constraints
    - Recognition of "the software crisis"
- Term became popular after NATO Conference in Garmisch Partenkirchen (Germany), 1968

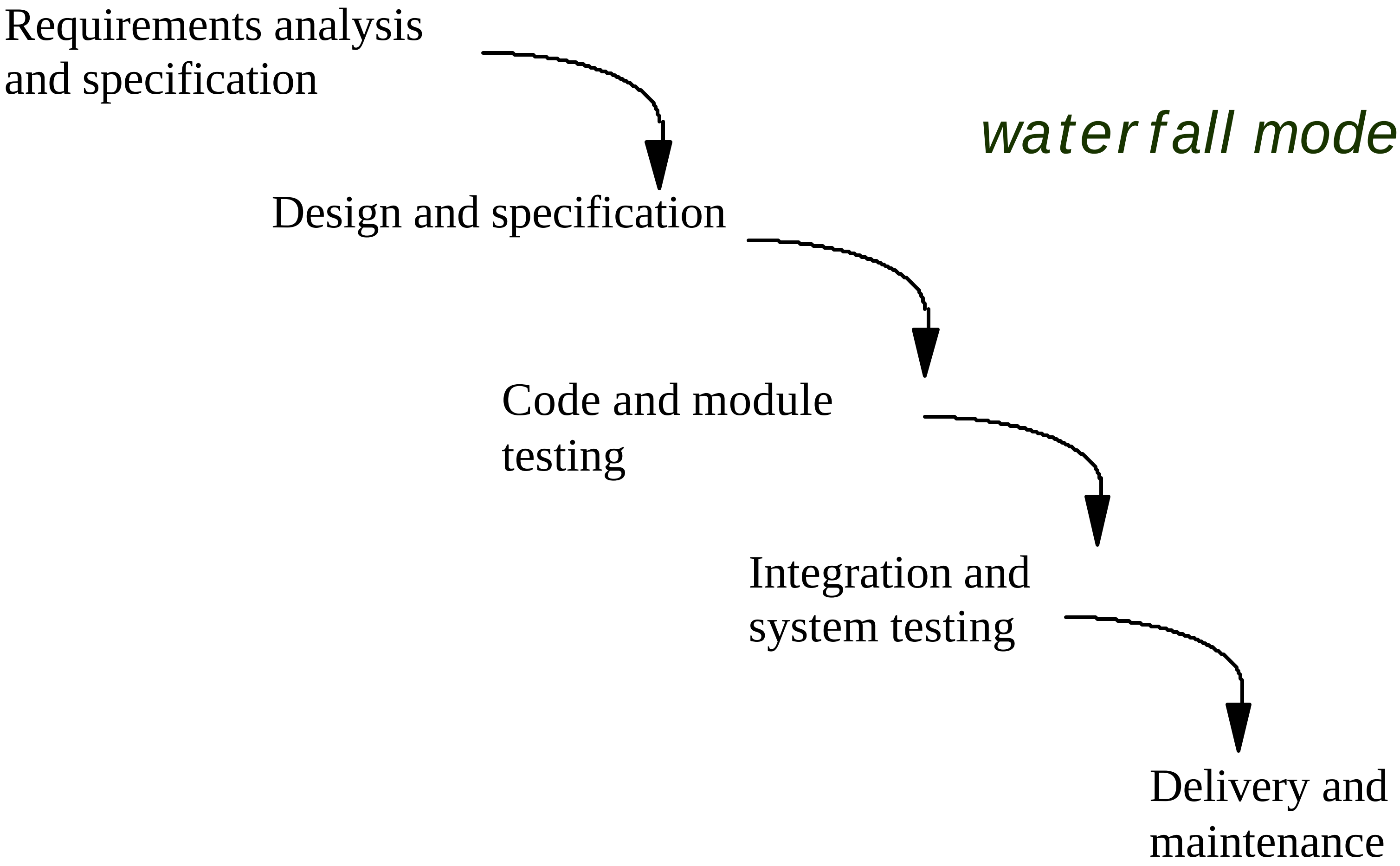
Role of software engineer

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

Software lifecycle- Waterfall Model



Requirements analysis and specification

Design and specification

Code and module testing

Integration and system testing

Delivery and maintenance

waterfall model

Software Qualities

- Software is built to meet a certain functional goal and satisfy certain qualities
- Software processes also must meet certain qualities

Software product

Different from traditional types of products

- **intangible**
  - difficult to describe and evaluate
- **malleable**
- **human intensive**
  - involves only trivial “manufacturing” process

Classification of software qualities

Internal vs. external

- External→ visible to users
- Internal→ concern developers

Product vs. process

- Our goal is to develop software products
- The process is how we do it

Internal qualities affect external qualities

Process quality affects product quality

Correctness

Software is correct if it satisfies the functional requirements specifications

If specifications are formal, since programs are formal objects, correctness can be defined formally

- It can be proven as a theorem or disproved by counterexamples (testing)

Reliability

- informally, user can rely on it
- can be defined mathematically as “probability of absence of failures for a certain time period”
- if specs are correct, all correct software is reliable, but not vice-versa (in practice, however, specs can be incorrect …)

Robustness

- software behaves “reasonably” even in unforeseen circumstances (e.g., incorrect input, hardware failure)

Performance

Efficient use of resources

- memory, processing time, communication

Can be verified

- complexity analysis
- performance evaluation (on a model, via simulation)

Performance can affect scalability

- a solution that works on a small local network may not work on a large intranet

Usability

The ease of use of the system by expected users

Other term: user-friendliness

Rather subjective, difficult to evaluate

Verifiability

How easy it is to verify properties

- **mostly an internal quality**
  - **use of monitors to verify constraints on traffic between components**
- **can be external as well (e.g., security critical application)**

Maintainability

Maintainability: ease of maintenance

Maintenance: changes to software after release

Maintenance costs exceed 60% of total cost of software

Three main categories of maintenance

- ***corrective*: removing residual errors (20%)**
- ***adaptive*: adjusting to environment changes (20%)**
- ***perfective*: quality improvements (>50%)**

Maintainability

Can be decomposed as

- **Repairability**
  - ability to correct defects in reasonable time
- **Evolvability**
  - ability to adapt sw to environment changes and to improve it in reasonable time

Reusability

Existing product (or components) used (with minor modifications) to build another product

- (Similar to evolvability)

Also applies to process

Reuse of standard parts measure of maturity of the field

Portability

Software can run on different hardware platforms or ssoftware environments

Remains relevant as new platforms and environments are introduced (e.g. digital assistants)

Relevant when downloading software in a heterogeneous network environment

Understandability

Ease of understanding software

Program modification requires program understanding

Typical process qualities

Productivity

- denotes its efficiency and performance

Timeliness

- ability to deliver a product on time

Visibility

- all of its steps and current status are documented clearly

Software Engineering Principles

- Principles form the basis of methods, techniques, methodologies and tools
- Seven important principles that may be used in all phases of software development
- Apply to the software product as well as the development process

Key principles

1. Rigor and formality
2. Separation of concerns
3. Modularity
4. Abstraction
5. Anticipation of change
6. Generality
7. Incrementality

1. Rigor and formality

Software engineering is a creative design activity,

BUT

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

Examples:

Product:

Formal- Mathematical analysis of program correctness

Rigorous- Systematic test data derivation

Process:

Rigorous- detailed documentation of each development step in waterfall model

Formal- automated transformational process to derive program from formal specifications

2. 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

Example:

Process: Go through phases one after the other as in waterfall Model

- Does separation of concerns by separating activities with respect to time

Separation of concerns

Examples:

Process: Go through phases one after the other as in waterfall Model

- Does separation of concerns by separating activities with respect to time

Product: Keep different types of product requirements separate

- Functionality discussed seperately from the performance constraints

3. 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**

- when dealing with a module we can ignore details of other modules

Cohesion and coupling

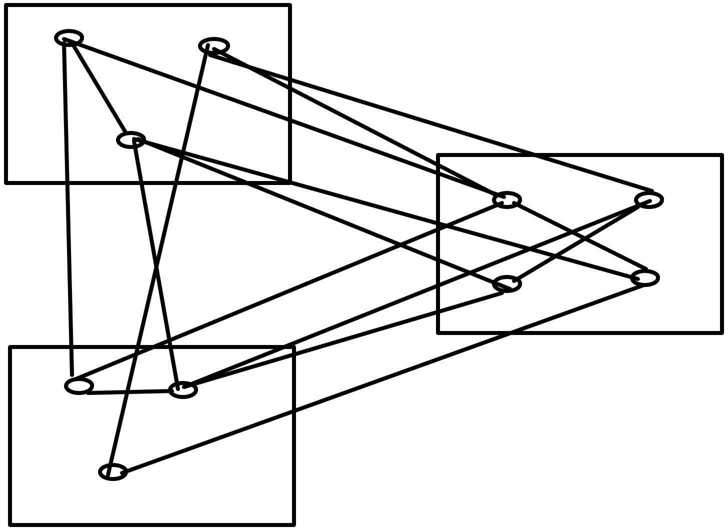
Each module should be *highly cohesive*

- module understandable as a meaningful unit
- Components of a module are closely related to one another

Modules should exhibit *low coupling*

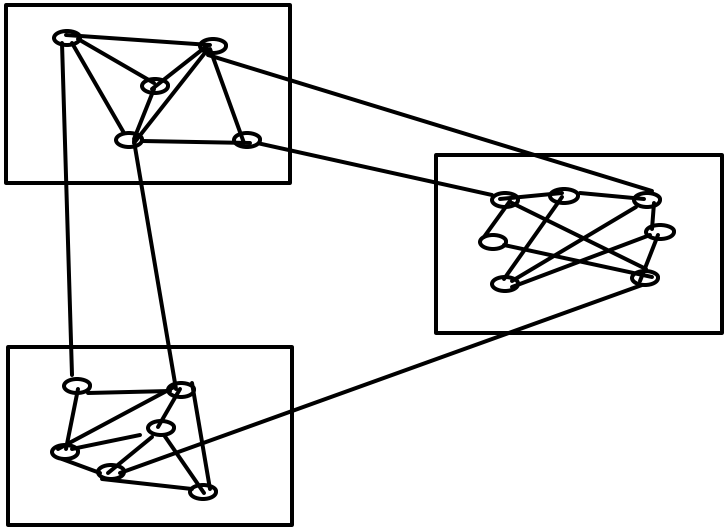
- modules have low interactions with others
- understandable separately

An Example



(a)

high coupling low cohesion



(b)

low coupling high cohesion

4. 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

Example : the user interface of a watch (its buttons) abstracts from the watch's internals for the purpose of setting time; other abstractions needed to support repair

Abstraction ignores details

Example: equations describing complex circuit (e.g., amplifier) allows designer to reason about signal amplification

Equations may approximate description, ignoring details that yield negligible effects (e.g., connectors assumed to be ideal)

Abstraction yields models

For example, when requirements are analyzed we produce a model of the proposed application

The model can be a formal or semiformal description

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

Abstraction in process

When we do cost estimation we only take some key factors into account

We apply similarity with previous systems, ignoring detail differences

5. Anticipation of change

Ability to support software evolution requires anticipating potential future changes

- **It is the basis for software evolvability**

6. 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

Sometimes a general problem is easier to solve than a special case

- Carefully balance generality against performance and cost

7. Incrementality

Process proceeds in a stepwise fashion (*increments*)

Examples (process)

- **deliver subsets of a system early to get early feedback from expected users, then add new features incrementally**
- **deal first with functionality, then turn to performance**

Case study: compiler

Compiler construction is an area where systematic (formal) design methods have been developed

- **e.g., BNF for formal description of language syntax**

Separation of concerns example

When designing optimal register allocation algorithms (*runtime efficiency*) no need to worry about runtime diagnostic messages (*user friendliness*)

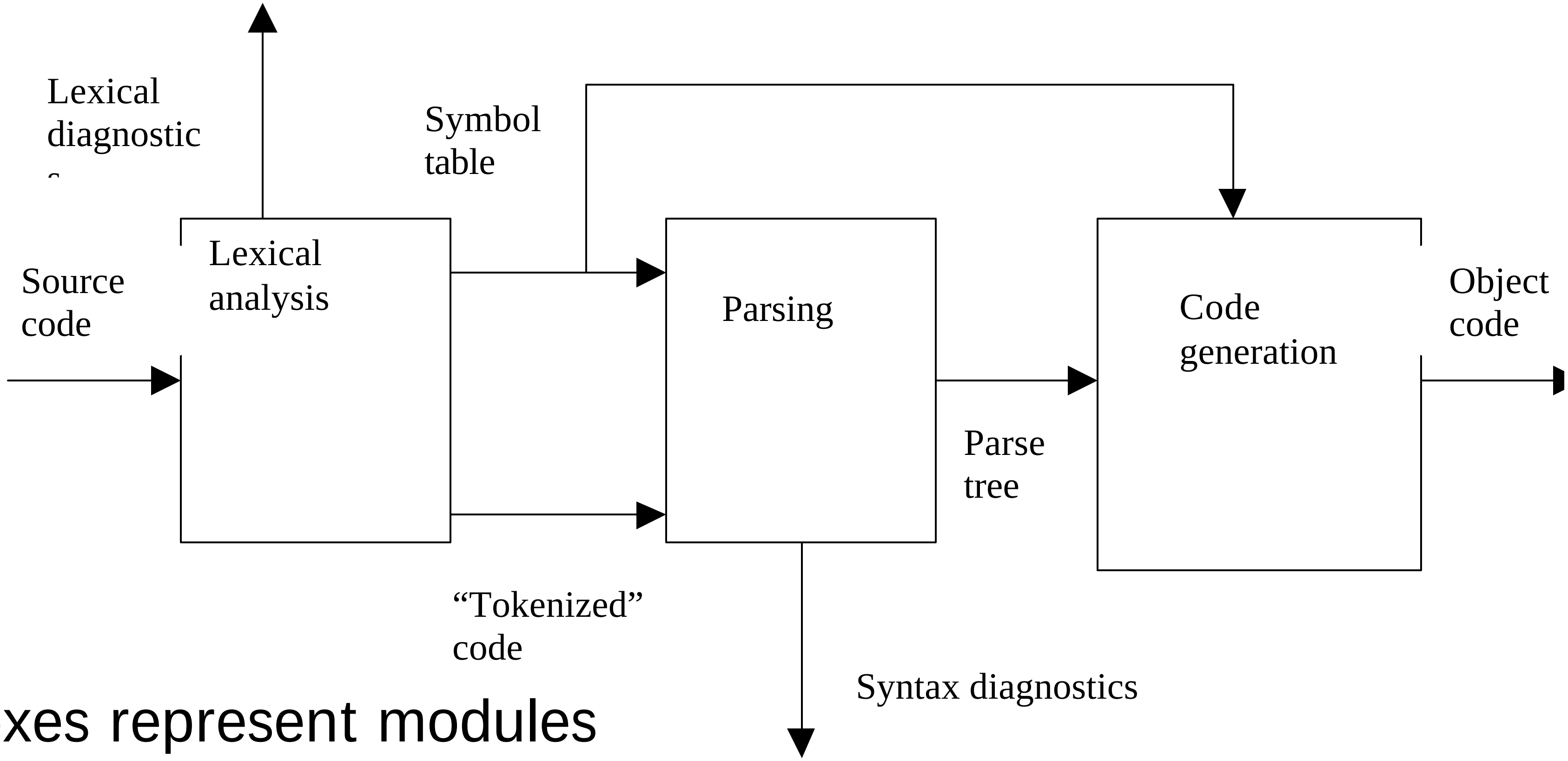
Modularity

Compilation process decomposed into phases

- **Lexical analysis**
- **Syntax analysis (parsing)**
- **Code generation**

Phases can be associated with modules

Representation of modular structure



Lexical
diagnostic

Symbol
table

Source
code

Lexical
analysis

Parsing

Code
generation

Object
code

Parse
tree

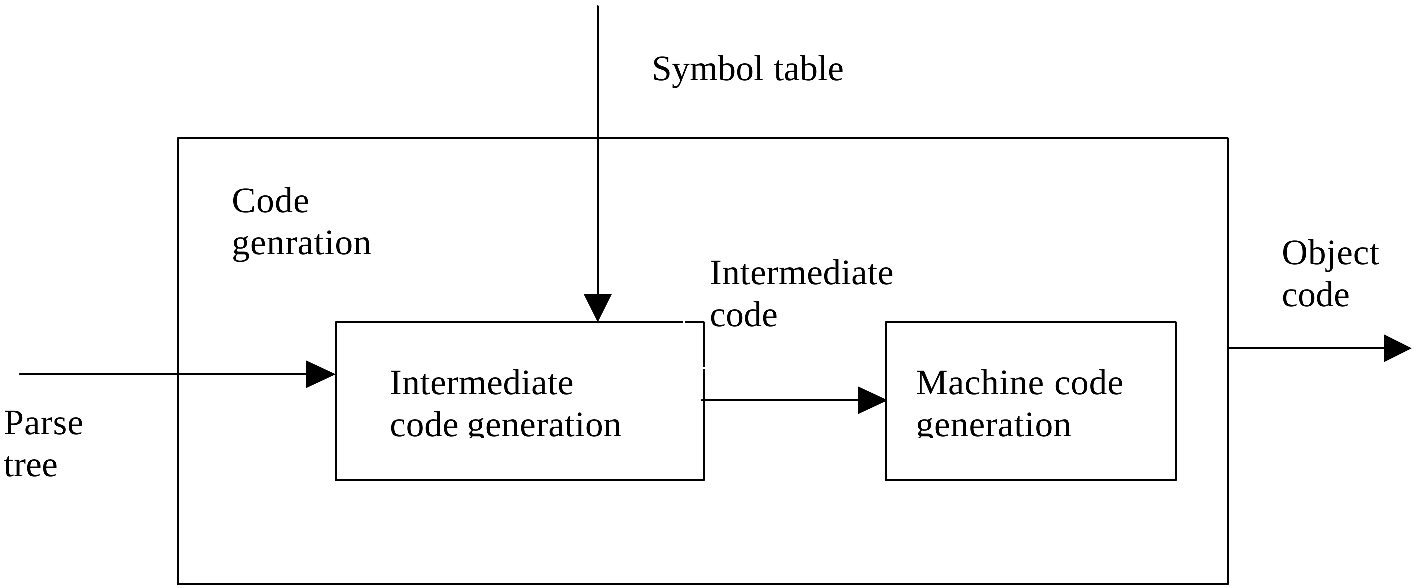
“Tokenized”
code

Syntax diagnostics

boxes represent modules
directed lines represent interfaces

Module decomposition may be iterated

further modularization of code-generation module



Abstraction

Applied in many cases

- **abstract syntax to neglect syntactic details such as** begin...end **vs. {...} to bracket statement sequences**
- **intermediate machine code (e.g., Java Bytecode) for code portability**

Anticipation of change

Consider possible changes of

- source language (due to standardization committees)
- target processor

Generality

Parameterize with respect to target machine (by defining intermediate code)

Develop compiler generating tools (*compiler compilers*) instead of just one compiler

Incrementality

Incremental development

- deliver first a kernel version for a subset of the source language, then increasingly larger subsets
- deliver compiler with little or no diagnostics/optimizations, then add diagnostics/optimizations