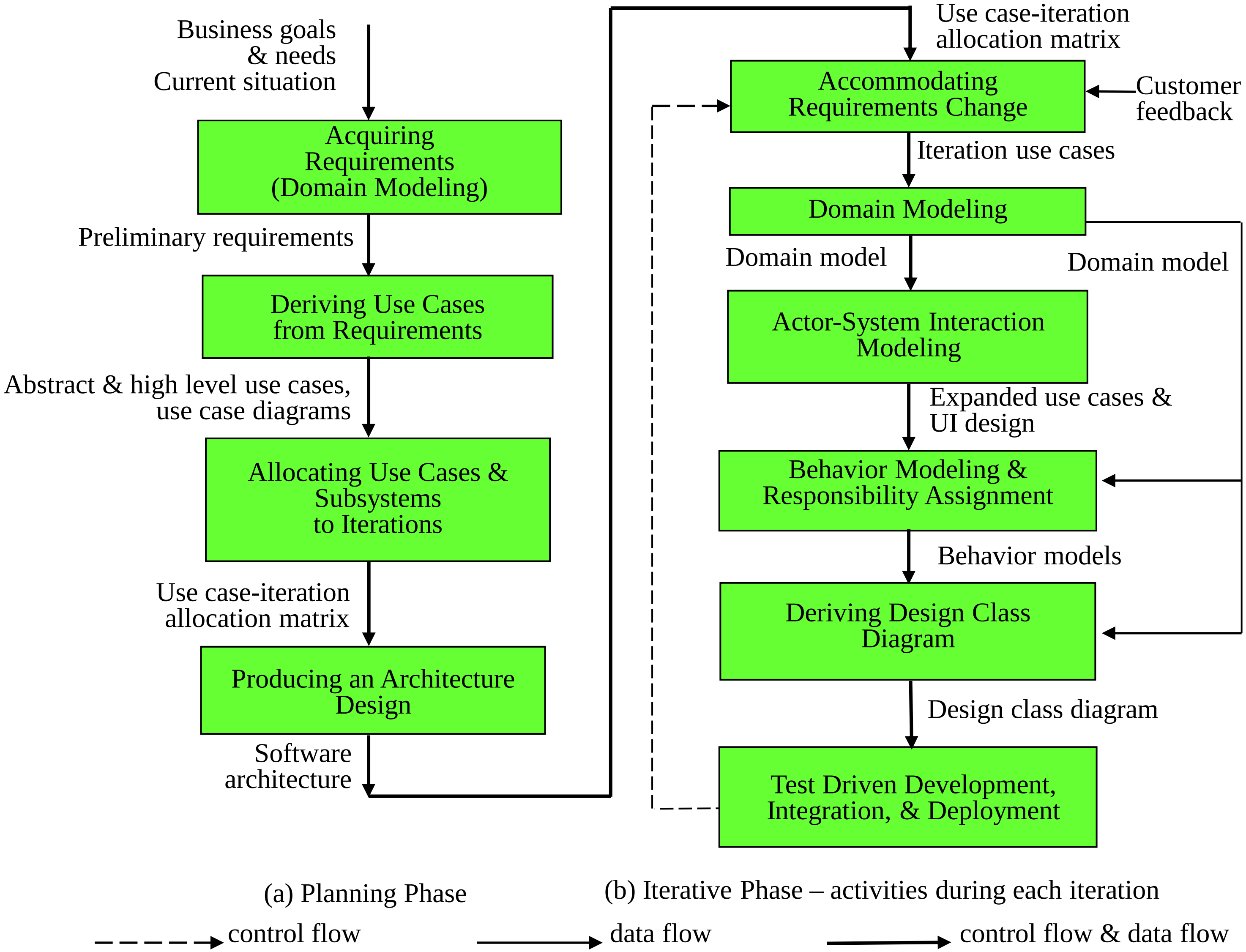
Chapter 19: Software Quality Assurance

Key Takeaway Points

- Software quality assurance encompasses a set of activities to ensure that the software under development or modification will meet functional and quality requirements.

- Software quality assurance activities are life-cycle activities.

SQA in the Methodology Context



What is Software Quality Assurance?

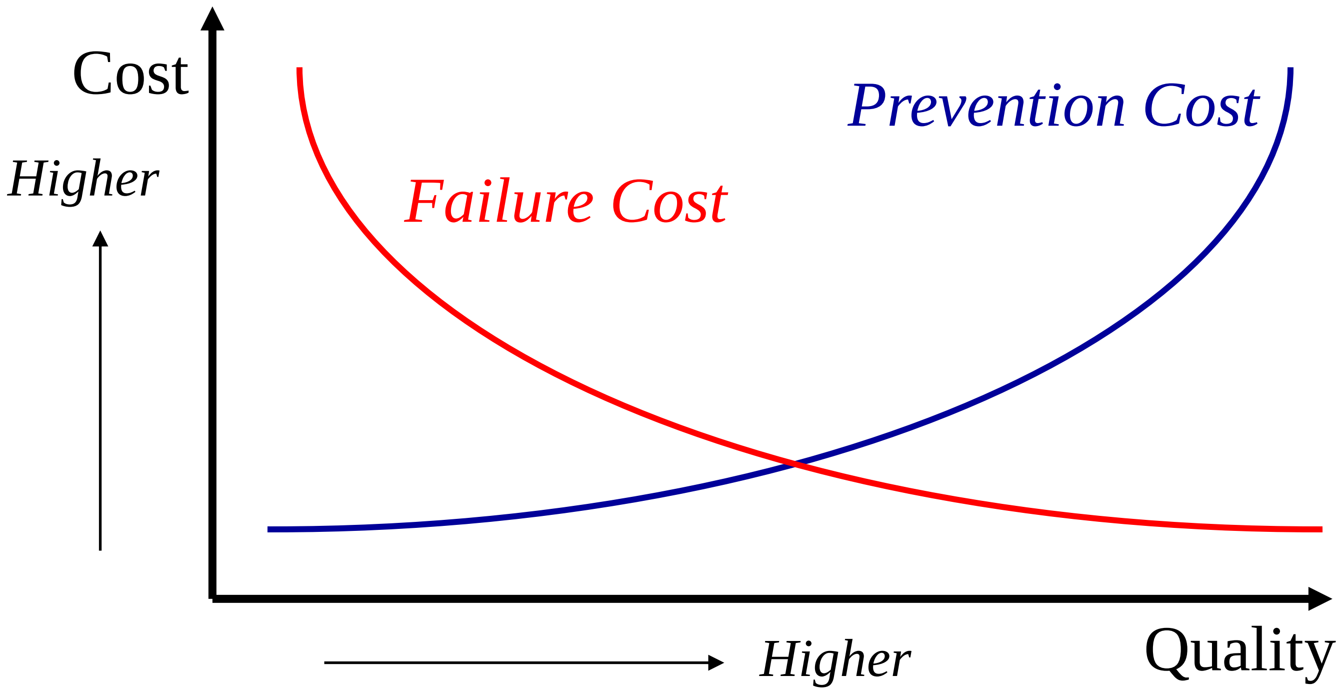
- SQA is a set of activities to ensure that the software product and/or process confirms to established requirements and standards

- The activities include:
  - Verification: *Are we building the product right?*
  - Validation: *Are we building the right product?*
  - Technical reviews
  - Testing: Attempts to uncover program errors

Cost of Quality

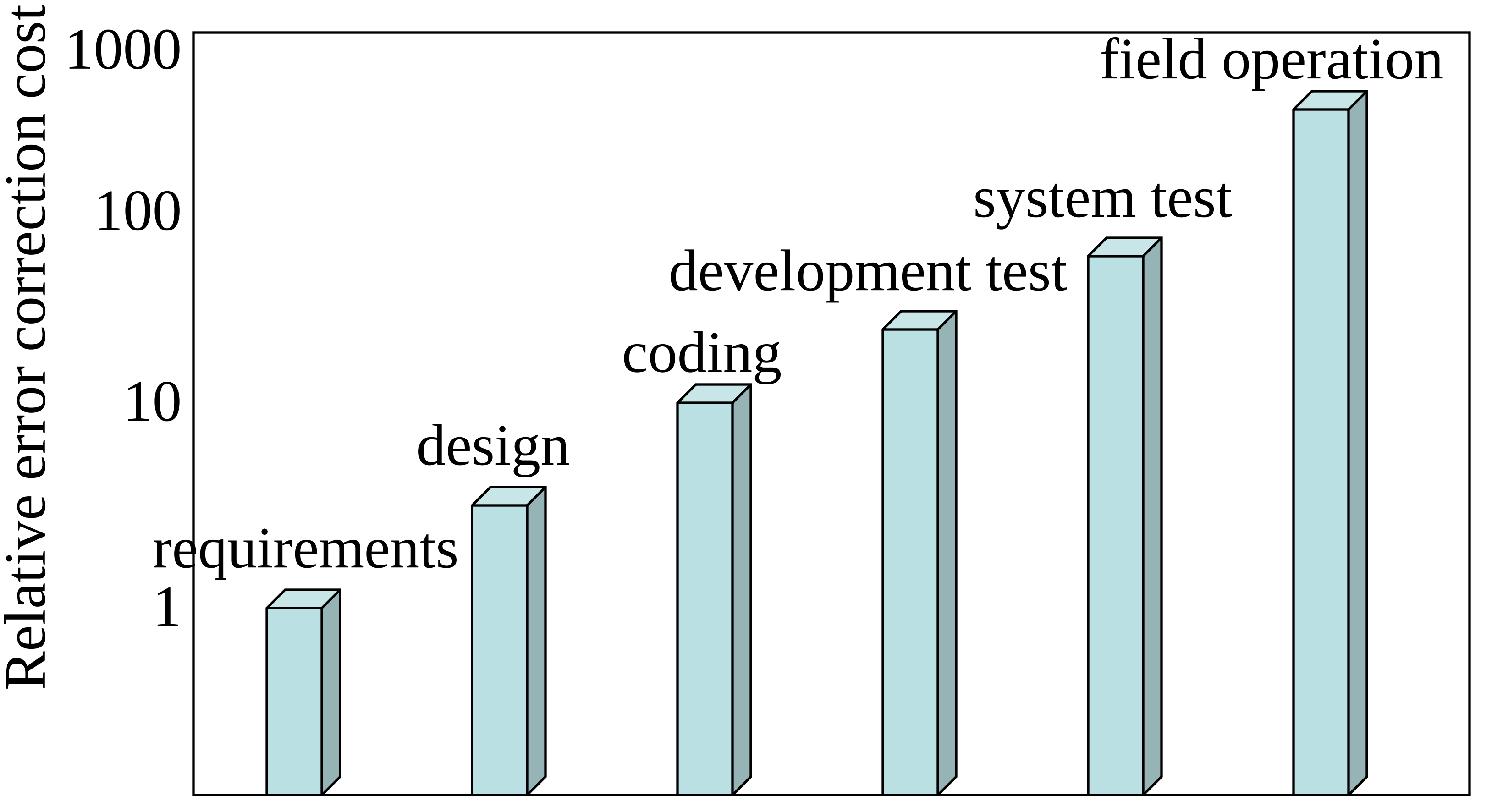
- Cost of quality consists of:
  - Costs of preventing software failures
    - Costs to implement an SQA framework (one time)
    - Costs to perform SQA activities (on going)
    - Costs to monitor and improve the framework (on going)
    - Costs of needed equipment (machines and tools)
  - Costs of failure
    - Costs to analyze and remove failures
    - Costs of lost productivity (developer and customer)
    - Costs of liability, lawsuit, and increased insurance premium
    - Costs associated with damage to company's reputation

Cost of Quality



Software engineering considers costs to accomplish something. Graph shows the ‘Quality’ and ‘Cost’ trade-off.

Costs of Quality



Software Quality Attributes

- Reliability
  - adequacy: correctness, completeness, consistency; robustness
- Testability and Maintainability
  - understandability: modularity, conciseness, preciseness, unambiguity, readability, measurability, assessability, quantifiability
- Usability
- Efficiency
- Portability
- etc.

Quality Measurements and Metrics

- Software measurements are objective, quantitative assessments of software attributes
- Metrics are standard measurements
- Software metrics are standard measurements of software attributes
- Software quality metrics are standard measurements of software quality
- Taking, collecting, and analyzing metrics takes time and money
  - Acting on/making decisions to change based on metrics takes time and money as well

- Class -- why do we need software metrics?
  - ***We cannot improve what we do not measure . . .***

Typical Software Quality Metrics

- Requirements metrics
  - Unambiguity = (# Uniquely Interpreted Req'mts)/(total # of Req'mts)
  - Completeness = (# Unique functions)/(Total # of combinations of states & stimuli)
  - Correctness = (# validated correct req'mts)/(Total # of Req'mts)
  - Consistency = (# of non-conflicting Req'mts)/(Total # of Req'mts)
- Design metrics
  - Fan-in = # of incoming messages to a module
  - Fan-out = # of outgoing messages from a module
  - Modularity = (a * Cohesion + b * Coupling)/(a + b)
  - Module Design Complexity = # of decisions to call +1 = # of integration tests
- Implementation metrics
  - # Defects/KSLOC
  - # Pages documentation/KSLOC
  - # Comment lines/SLOC
  - Cyclomatic complexity = #Edges – #Nodes + 2 = # of closed regions +1
- System metrics
  - Reliability = MTBF = MTTF + MTTR
  - Availability = MTBF/(MTBF + MTTR)

Requirements Metrics

Requirements Unambiguity Q1 =

$$\frac{\text{\#of uniquely interpreted requirements}}{\text{\#of requirements}}$$

Requirements Completeness Q2 =

$$\frac{\text{\#of unique functions}}{\text{\#of combinations of states and stimuli}}$$

Requirements Metrics

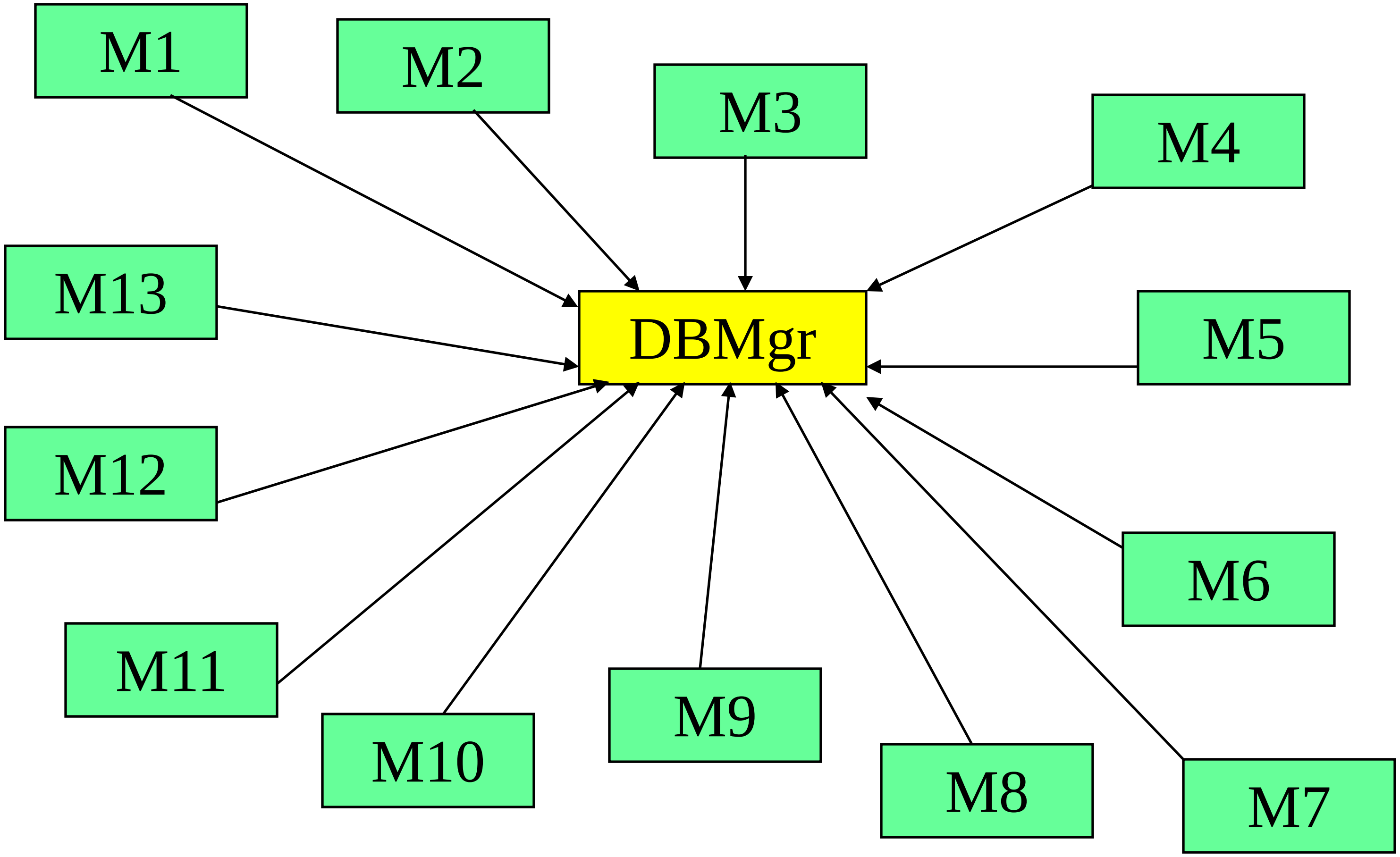
$$\text{Requirements Correctness Q3} = \frac{\text{\#of validated correct requirements}}{\text{\#of requirements}}$$

$$\text{Requirements Consistency Q4} = \frac{\text{\#of non-conflicting requirements}}{\text{\#of requirements}}$$

Design Metric Fan-In

- Number of incoming messages or control flows into a module.
  - Measures the dependencies on the module.
- High fan-in signifies a "*god module*" or "*god class*"
  - The module may have been assigned too many responsibilities
- It may indicate a low cohesion

Design Quality Metrics: Fan-In

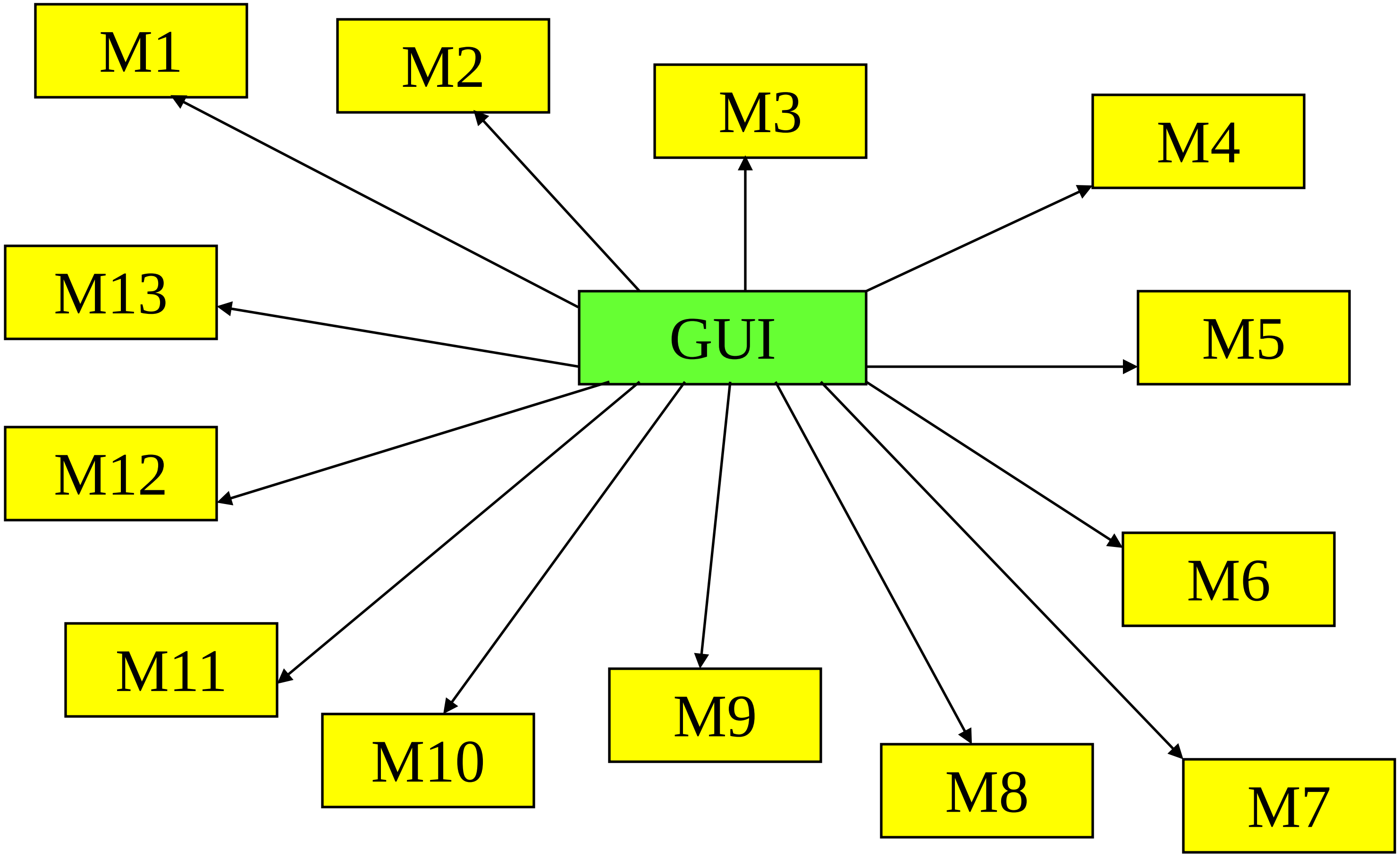


Design Quality Metrics Fan-Out

- Number of outgoing messages of a module
  - Measures the dependencies of this module on other modules

- High fan-out means it will be difficult to reuse the module because the other modules must also be reused

Quality Metrics: Fan-Out



Modularity

- Modularity – Measured by cohesion and coupling metrics.

Modularity = (a * Cohesion + b * Coupling)/(a + b), where a and b are weights of cohesion and coupling

Cohesion

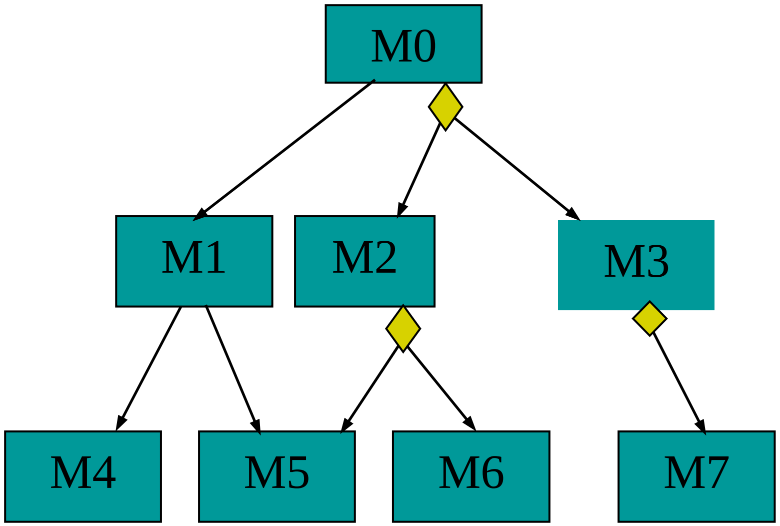
- Each module implements one and only one functionality
- Ranges from the lowest coincidental cohesion to the highest functional cohesion
- High cohesion enhances understanding, reuse, and maintainability

Coupling

- An interface mechanism used to relate modules
- It measures the degree of dependency
- It ranges from low data coupling to high content coupling
- High coupling increases uncertainty of run time effect
  - also makes it difficult to test, reuse, maintain, and change the modules

Module Design Complexity (mdc)

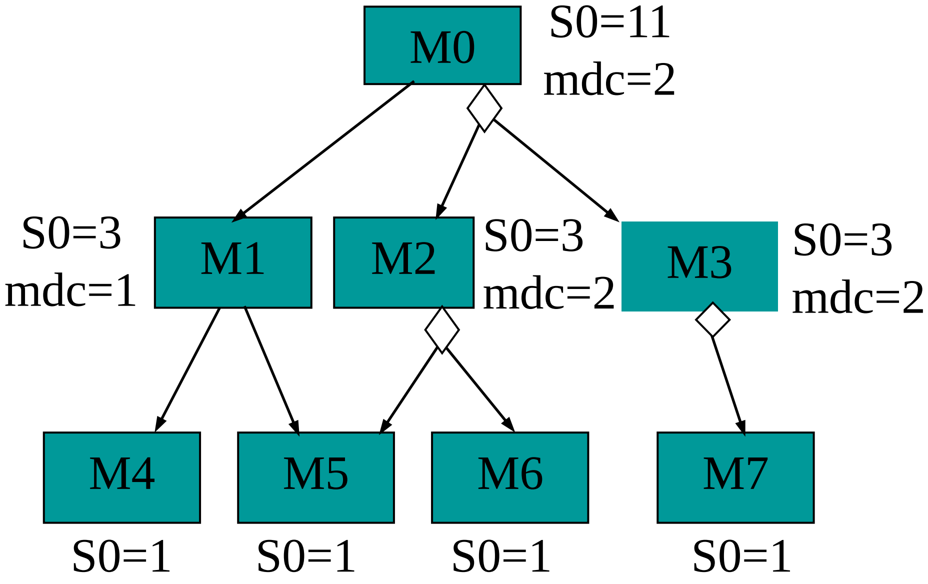
- The number of integration tests required to integrate a module with its subordinate modules.
- mdc is equal to the number of decisions to call a subordinate module (d) plus one



Module Design Complexity mdc=4

Design Complexity S0

- Number of subtrees in the structure chart with module M as the root
- S0(leaf) = 1
- S0(M) = mdc+S0(child1 of M)+...+S0(child*n* of M)



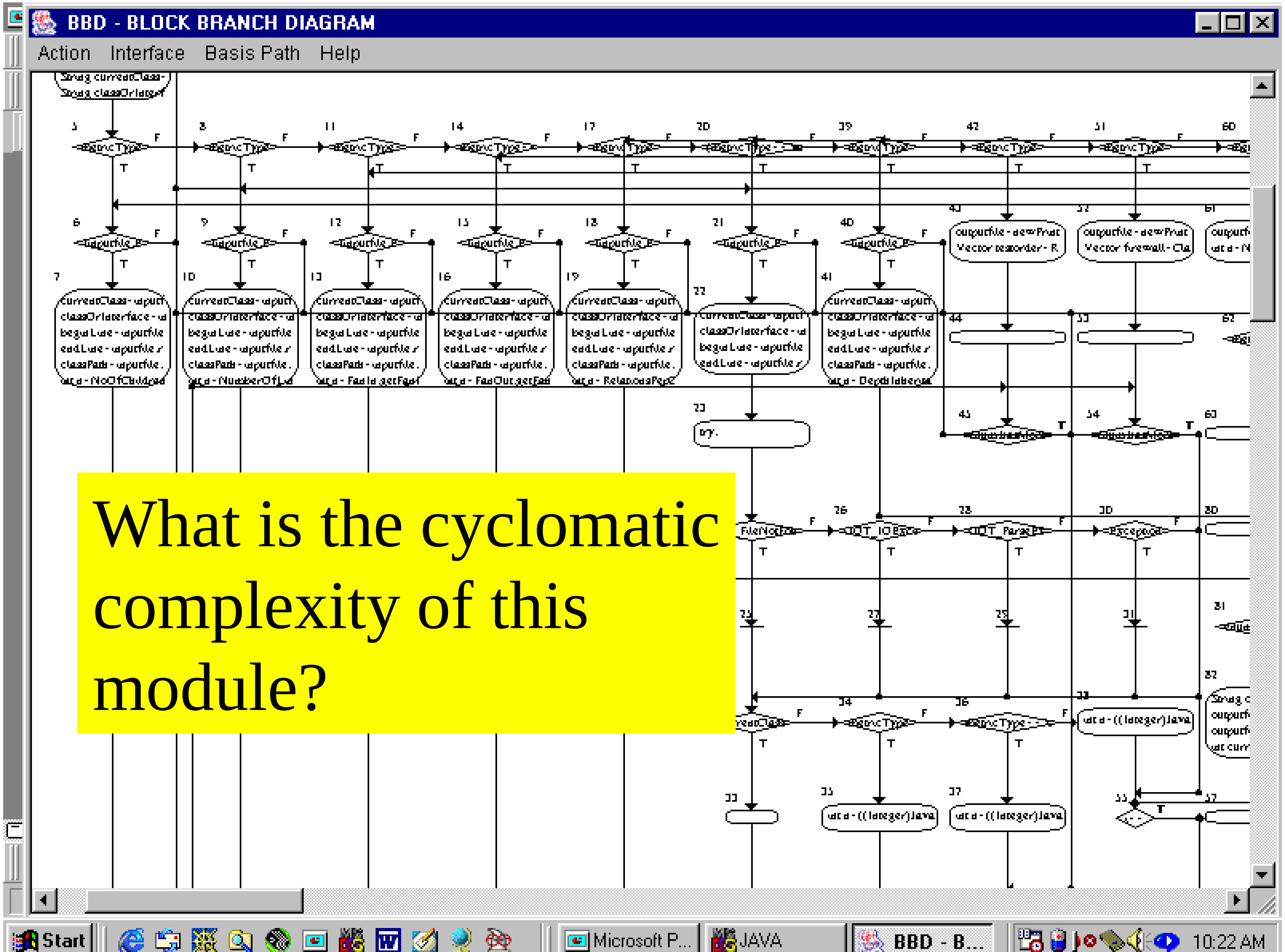
Integration Complexity

- Minimal number of integration test cases required to integrate the modules of a structure chart

- Integration complexity $S1 = S0 - n + 1$, where n is the number of modules in the structure chart

Implementation Metrics

- Defects/KLOC
- Pages of documentation/KLOC
- Number of comment lines/LOC
- Cyclomatic complexity
  - equals to number of binary predicates + 1
  - measures the difficulty to comprehend a function/module
  - measures the number of test cases needed to cover all independent paths (basis paths)



What is the cyclomatic complexity of this module?

Object-Oriented Quality Metrics

- Weighted Methods per Class (WMC)
  - $WMC(C) = Cm_1 + Cm_2 + \cdots + Cm_n$, *where:* Cm_i = *complexity metrics of methods of C*
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling Between Object Classes (CBO)
- Response for a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

Weighted Methods per Class

- $WMC(C) = Cm_1 + Cm_2 + \cdots + Cm_n$

  where: Cm_i = complexity metrics of methods of C

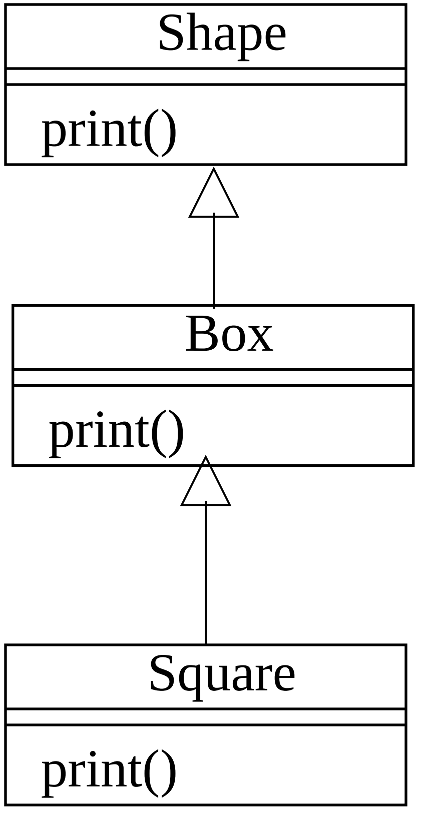
- <u>Significance</u>: Time and effort required to understand, test, and maintain class C increases exponentially with WMC

Depth of Inheritance Tree (DIT)

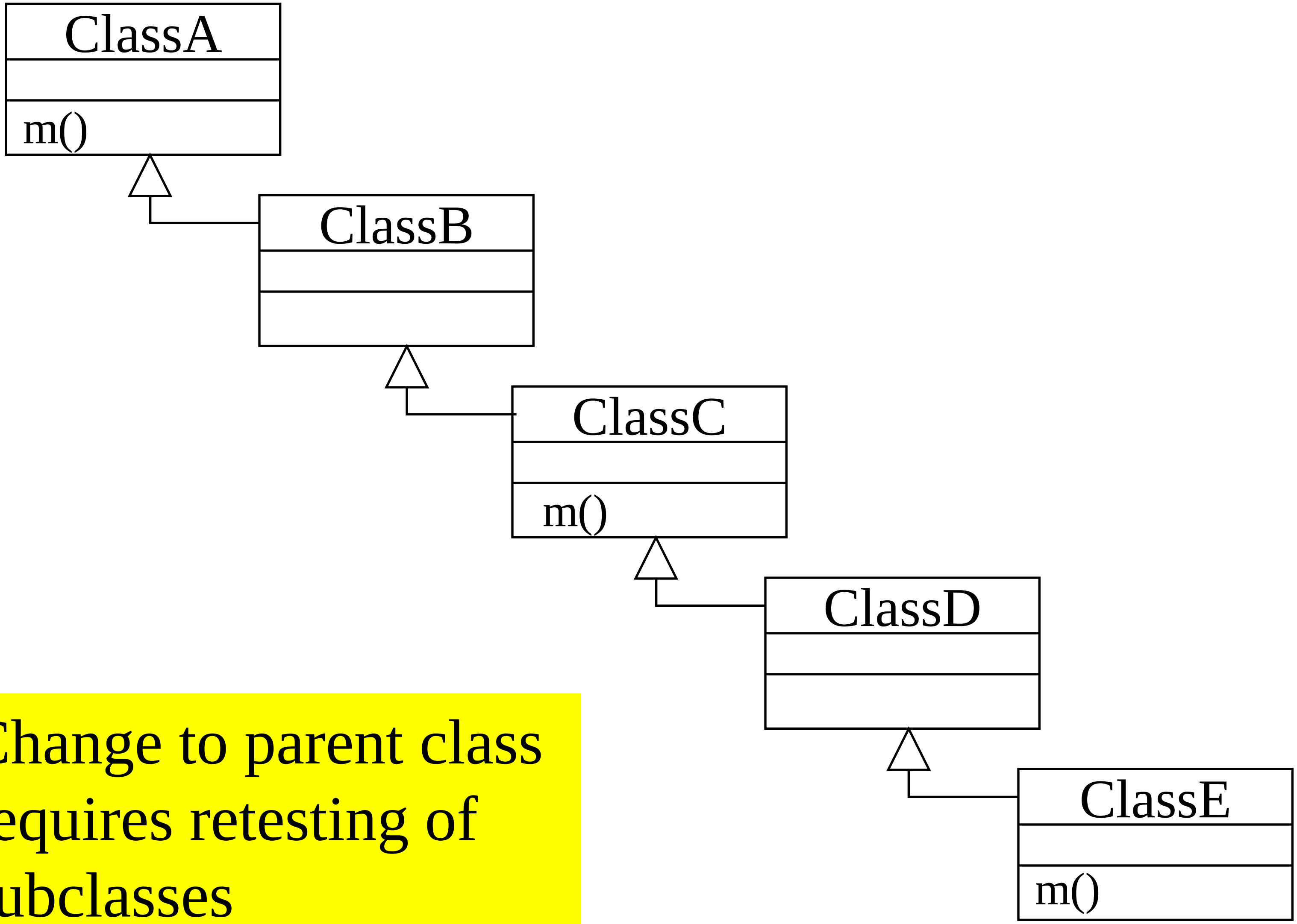
- Distance from a derived class to the root class in the inheritance hierarchy
- Measures . . .
  - the degree of reuse through inheritance
  - the difficulty to predict the behavior of a class
  - costs associated with regression testing due to change impact of a parent class to descendant classes

High DIT Means Hard to Predict Behavior

- All three classes include print()
- It is difficult to determine which "print()" is used:




```
public static void main (...) {
   Shape p; ….
   p.print();    // which print()?
   …
}
```



Change to parent class requires retesting of subclasses

Number of Children (NOC)

- NOC(C)

  $= | \{ C' : C' \text{ is an immediate child of } C \}|$
- The dependencies of child classes on class C increases proportionally with NOC
- Increase in dependencies increases the change impact, and behavior impact of C on its child classes
- These make the program more difficult to understand, test, and maintain

Coupling between Object Classes (CBO)

- $CBO(C) = |\{ C' : C \text{ depends on } C' \}|$

- The higher the CBO for class C the more difficult to understand, test, maintain, and reuse class C

Response for a Class (RFC)

- RFC (C) = |{ m : m is a method of C, or m is called by a method of C }|

- The higher the RFC, the more difficult to understand, test, maintain, and reuse the class due to higher dependencies of the class on other classes

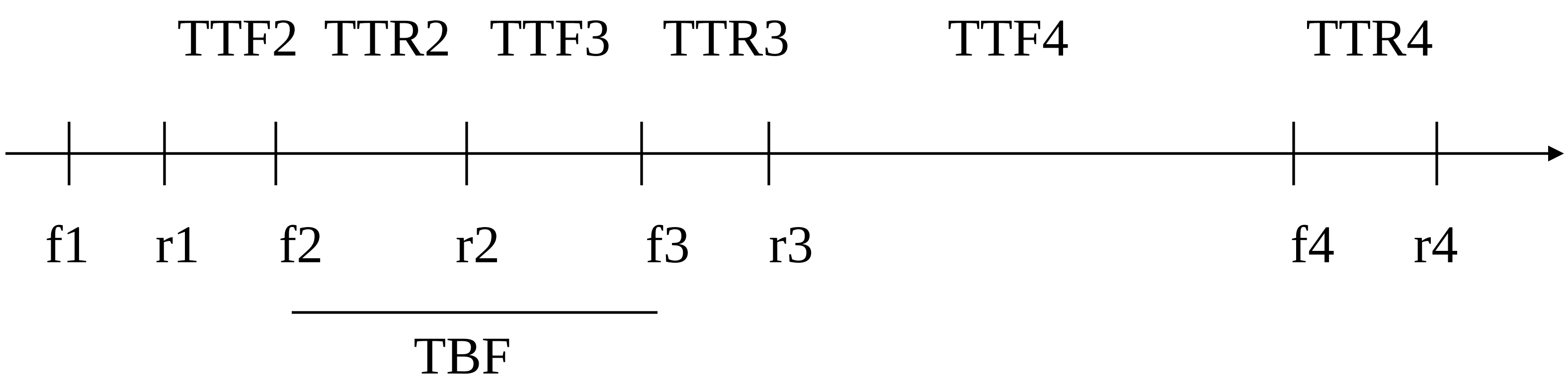
Lack of Cohesion in Methods

- LCOM (C) = n * (n-1) / 2 - 2 * $|\{ (m_i, m_j) : m_i \text{ and } m_j \text{ share an attribute of C } \}|$
- LCOM measures the number of pairs of methods of C that do not share a common attribute

- Class exercise:
  - Is it possible to derive a metric called "cohesion of methods of a class?"
  - If so, what would be the formula?

Reliability and Availability

- Mean time to failure (MTTF)
- Mean time to repair (MTTR)
- Mean time between failure (MTBF) = MTTF + MTTR
- Availability = MTTF/MTBF X 100%

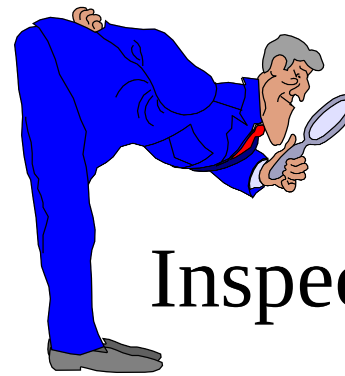


Usefulness of Quality Metrics

1. Definition and use of indicators
2. Directing valuable resources to critical areas
3. Quantitative comparison of similar projects and systems
4. Quantitative assessment of improvement including process improvement
5. Quantitative assessment of technology

Software Verification and Validation Techniques

Desk checking



Inspection

Walkthrough

Peer review

Testing

Software Inspections

- Inspector examines the representation with the aim of detecting anomalies and defects
- Inspection does not require execution of code/product so Inspections can be used before implementation
- Inspections can be applied to any representation of the system (requirements, design, code, test data, etc.)
- It is an effective technique for error detection

Program Inspection

- An approach for detecting defects, not correcting defects
- Defects may be logical errors, anomalies in the code that might indicate an erroneous condition (e.g. an uninitialized variable) or non-compliance with standards
- Step by step reading the program
- Check against a list of criteria
  - common errors
  - standards
  - consistency rules

Inspection Pre-condition

- A precise specification must be available
- Team members must be familiar with the organisation standards
- Syntactically correct code must be available
- An error checklist should be prepared
- Management must accept that inspection will increase costs early in the software process
- Management must not use inspection for performance evaluation

Inspection Procedure

- System overview presented to inspection team
- Code and associated documents are distributed to inspection team in advance for their review
- Inspection meeting takes place and discovered errors are formally noted
- Modifications/changes are made to repair discovered errors
- Re-inspection may or may not be required
- Inspection metrics are collected and retained

Fagan Inspection Technique

- A checklist of inspection items
- It includes design and code inspections
- 4 - 5 inspectors: moderator, designer, programmer, tester
- It is good for real-time systems because errors are not repeatable
- Different programmers tend to have different error patterns

Design Inspection

- Precondition: functional requirements and design specification must exist
- Design inspection checks for
  - completeness
  - consistency, and
  - correctness

Walkthrough

- The developer loudly reads through the program
- The developer provides explanation if he/she deems necessary
- Other team members may ask questions and stimulate doubts
- The developer answers questions and justifies answers

Difference Between Walkthrough and Inspection

Walkthrough

- use simple test data
- team is led through manual simulation
- check product step by step while reading aloud
- stimulate doubt and discussions

Inspection

- use a list of criteria
  - common errors
  - standards
  - consistency rules
- team inspects the work product for common errors and compliance to standards and consistency rules

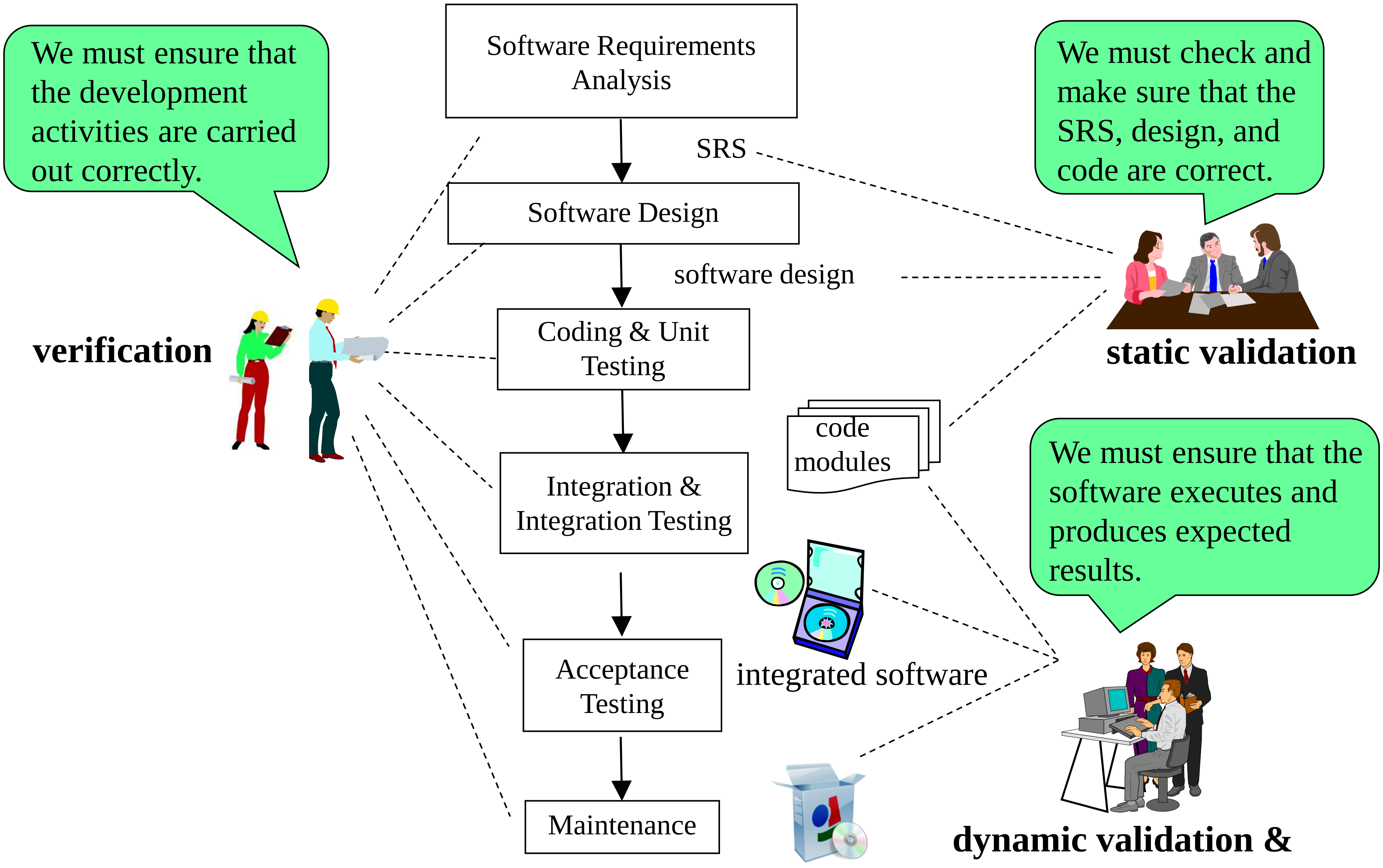
Peer Review

- The product is reviewed by peers, guided by a list of review questions
- The review questions are designed to assess the product
- The reviewers are given one to two weeks to review the work
- The review results may vary due to difference in reviewer’s knowledge, experience, background, and criticality
- A review meeting is usually conducted to discuss the review results

Peer Review Procedure

1. A product overview is presented to the reviewers, who are assigned products to review
2. The reviewers evaluate the product and answer the review questions independently
3. At the review meeting, the reviewers present their comments and suggestions, and the developer answers questions and clarifies doubt
4. After the review meeting, the developer fixes the problems and produces a summary of solutions
   – The reviewers check the changes
5. A second review meeting may be needed

Verification and Validation in the Life Cycle



Verification

- Verification is the process of checking that the "implementation" conforms to the "specification"
- A lower level abstraction is the implementation of a higher level abstraction
- Design verification involves checking that the software design satisfies the requirements specification

Validation

- Validation aimed at checking the correspondence between a model and the real world, e.g., a functional specification corresponds to the real needs of the customer
- A process that seeks to refute a model
- Functional testing is a validation technique

Static and Dynamic Verification & Validation

- Review and inspection are concerned with analysis of the static system representation to detect problems (static verification)

- Testing is concerned with executing the product and observing its behaviour (dynamic verification)

Formal Verification

- A precise mathematical analysis process.
- It checks the correspondence between a specification and an implementation.
- Example:
  - specification: a logical expression of some constraint – bad things should not happen
  - implementation: a state machine model representing the design of a system
  - formal verification: model checking to find a state that violates the logical expression.

Informal Verification

- The correspondence is usually checked by a manual process
- Techniques include desk-checking, inspection, walkthrough, and peer review

Verification Is Consistency Checking

- Internal consistency: a specification must not contain a contradiction
- Consistency between two levels of abstraction:
  - software design specification and the requirements specification
  - leveling in Data Flow Diagram decomposition
- Consistency between a product and standards

V&V in the Requirements Phase

- Activities performed during the requirements phase include:
  - technical review, expert review, and customer review
  - prototyping
  - inspection, and walkthrough
  - formal and informal verifications
  - requirements tracing

Technical Review

- Technical reviews are performed internally by developers looking for:
  - incomplete definitions
  - incomplete formulation of conditions such as an if-condition without an else part, or less than 2**n rules for n binary conditions
  - inconsistencies in requirements, types, and logical expressions
  - ambiguities in definitions and requirements

(*continued on next page*)

Technical Review

- duplicate formulations of concepts, requirements, or constraints
- requirements traceability problems
- unwanted implementation details, e.g.,
  - use of pointers, defining physical data structures
  - use of pseudo-code or programming language code
- potential feasibility, performance, or cost problems

Checking Requirements and Constraints

- Definition completeness
  - every concept mentioned in the requirements has an explicit definition
  - pay attention to important but undefined adjectives (e.g., good customer receives 20% discount)
  - every defined concept has been used at least once

Checking Requirements and Constraints

- Type consistency
  - types of objects involved in each concept match the definition and each use of the concept
- Logical completeness and consistency

Example

Metro Transit provides attractive prices for group travel

- 2-9 people traveling together will receive 25% discount
- 10 people or more will receive a 40% discount

MT discounts also take into account students and military service personnel traveling

- Students are eligible for 50% discount
- Military service-personal pay only 25% of the standard rate

Logical Completeness and Consistency

Rule no.	1	2	3	4	5
Category	Student	Other	Military	Other	Other
Group size	-	1	-	2-9	>=10
0% dis.		x			
25% dis.				x	
40% dis.					x
50% dis.	x				
75% dis.			x		

Is this decision table complete, and consistent?

Ambiguity in Requirements

Example: *Metro Transit discount policy:*

Children between 2 and 12 pay half price.

Children under 2 years old get 90% discount.

It has 4 possible interpretations

Under 2 (90%)	Between 2 and 12 (50%)	?
[0,2)	(2,12)	What about 2
[0,2)	[2,12]	Ok
[0,2]	(2,12)	Ok
[0,2]	[2,12]	Inconsistency at 2

And/Or Ambiguity

Ex. Passengers over 67 or students under 26 and traveling at least 150 km one way receive a 50% discount.

A: the passenger is over 67

B: the passenger is a student under 26

C: the passenger travels at least 150 km one way

$$\begin{tikzcd} & A\vee B\wedge C \quad \text{(literally)} \arrow[dl] \arrow[dr] & \\ (A\vee B)\wedge C & & A\vee(B\wedge C) \end{tikzcd}$$

What Does Logical Consistency Means?

- If Requirements R1, R2, ..., R*k* are not logically consistent, then . . .
  – R1 AND R2 AND ..AND R*k* <==> False
- R1,...,R*k* cannot be true at the same time
- It is impossible to produce a system that can satisfy all the requirements R1,..., R*k*

Completeness, Consistency and Unambigutiy

- Completeness: all cases are covered
  - external completeness, a validation problem
  - internal completeness, a verification problem

- Consistency: there exists no contradiction, the system is theoretically possible

- Unambiguity: there exists at most one possible interpretation of the requirements

Expert Review

- Expert reviews are performed by domain experts, who look for . . .
  - inaccuracies in the formulation of domain concepts, regulations, policies, standards
  - conceptualization problems
  - incorrect formulation of business rules
  - other domain specific problems

Customer Review

- These reviews are performed with customer representatives and users, and look for . . .
  - mismatch between requirements and real world
  - problems in user interfaces such as sequence of interaction, look and feel, use of GUI implementation technologies
  - issues relating to non-functional requirements
  - problems in application specific constraints such as operating environment, costs, political considerations

Requirements Tracing

- Performed by the authors of the requirements prior to external reviews
- Ensure that each higher-level requirement corresponds to some lower-level requirements
- Ensure that each lower-level requirement corresponds to some higher-level requirement
- Ensure that lower-level requirements are sufficient to realize the higher-level requirements

V&V in the Requirements Phase

- V&V in the requirements phase also check the requirement-use case traceability matrix
  - ensure that each requirement is realized by some use cases, and
  - each use case realizes some requirements
- Use prototyping to demonstrate capabilities of the system to the customer representatives and user representatives

V&V of Architectural Design

- Activities:
  - peer design review
  - design inspection
  - design walkthrough
- Check for design quality metrics
- Check to ensure that requirements and constraints are fulfilled
- Check to ensure software design principles, and security principles are satisfied

Desirable Properties for Architectural Design

- Subsystems and modules should be functionally cohesive
- Interaction between subsystems and modules should be explicit
- Modules should be small and easy to understand
- Decisions should be confined to a single module
- Modules should be easy to test and maintain
- Implicit, global assumptions should be avoided

V&V in the Implementation Phase

- Activities:
  - detailed design review
  - code review, inspection, and walkthrough
  - testing, and
  - reliability assessment
- Check
  - correspondence between the implementation and design
  - consistency of module interfaces

(*continued on next page*)

V&V in the Implementation Phase

- correctness of the implementation
- conformance of coding standards
- proper use of programming constructs
- un-initialized/improperly initialized variables, structures or pointer
- quality of the code according to various code quality metrics
    - cyclomatic complexity (must not exceed *10*)
    - information hiding
    - cohesion and coupling
    - modularity
    - etc.

Software Quality Assurance Functions

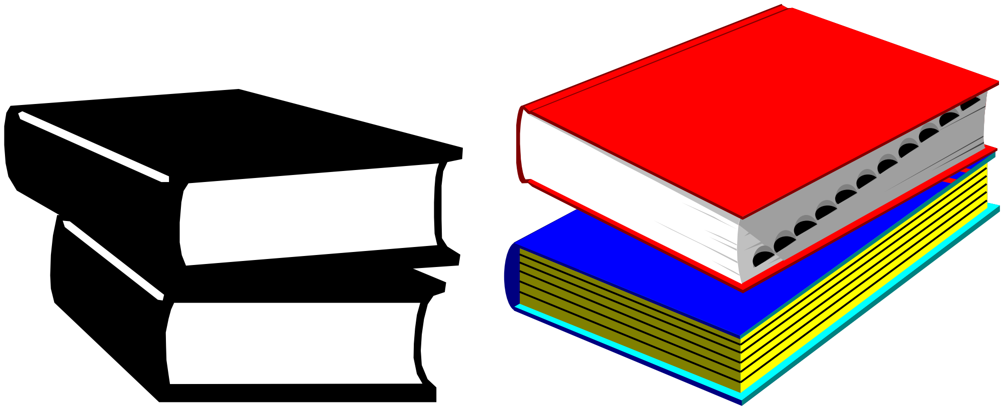
- Definition of Processes and Standards
  - Definition of Process and Methodology
  - Definition of SQA Standards and Procedures
  - Definition of Metrics and Indicators
- Quality Management
  - Quality Planning
  - SQA Control
- Process Improvement

Definition of Processes and Standards

- Definition of a SQA Framework
  - defining development, quality assurance, and management processes and methodologies (see Figure 19.5 for SQA in plan-driven and agile processes)
  - defining SQA standards, procedures, and guidelines
  - defining quality metrics and indicators for quality measurement and assessment

Quality Management Planning

- Quality planning defines a scheduled sequence of activities to be carried out to assure the desired software quality
- Examples are ANSI/IEEE Standard 730-1984, and 983-1986 SQA Plans



Components of an SQA Plan

- Purpose - objectives and scope of the plan as well as the product and its use.
- Management – project organization, and team structure including roles and responsibilities for SQA functions and activities.
- Standards and conventions to be applied.
- Review and audit – types of reviews to be used, problem reporting, and correction procedures.

(*continued on next page*)

Components of an SQA Plan

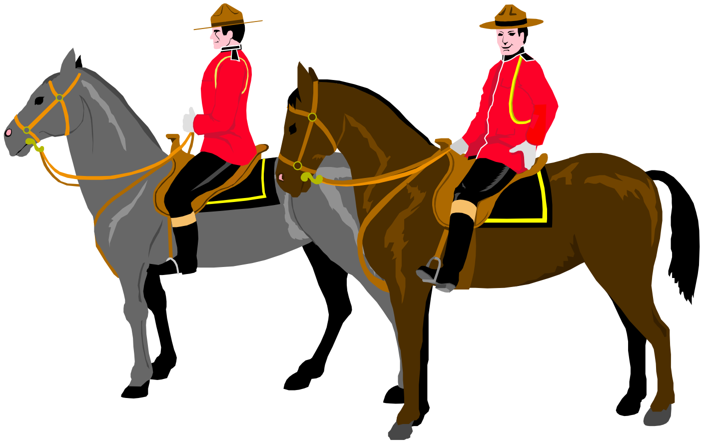
- Software configuration management
  - activities to ensure consistent update, and track changes
- Processes, methodologies, tools, and techniques to be used
- Metrics and indicators to be applied

Watts Humphrey's Quality Plan Outline

- **Product introduction**
  - A description of the product, its intended market, and the quality expectations for the product.
- **Product plans**
  - The critical release dates and responsibilities for the product along with plans for distribution and *product servicing*.
- **Process descriptions**
  - The development and service processes which should be used for product development and management.
- **Quality goals**
  - The *quality goals* and *plans* for the product, including an identification and justification of *critical product quality attributes*.
- **Risks and risk management**
  - *The key risks which might affect product quality and the actions to address these risks*.

SQA Control

- It ensures that the SQA plan is carried out correctly
- It monitors the software development project
- It collects and manages SQA related data
- It reports the data to software process improvement process



Process Improvement Process

- Definition and execution of a process improvement process (PIP)
- It defines metrics and indicators for process improvement
- It gathers data for improving the process
- It computes the metrics and indicators
- It produces process improvement recommendation to the management

Applying Agile Principles

- Good enough is enough
- Keep it simple and easy to comply
- Management support is essential
- A collaborative and cooperative approach between all stakeholders is essential
- The team must be empowered to make decisions
- Values working software over comprehensive documentation

Summary

- We discussed what software quality was and the benefits of establishing quality efforts in software
  - Software Quality attributes, measures, metrics, and indicators
  - "Cost of Quality"
  - Example software quality measures
- Discussed software V&V techniques
  - Inspections, walkthroughs, peer reviews
  - Lifecycle opportunities with V&V
- Discussed the role and functions of SQA
  - Processes, standards
  - Methodologies
  - Planning, control, metrics, process improvement