



SOEN 6431

Software Comprehension and Maintenance

Week 3

Program Analysis and
Software Traceability

Introduction

- Prior to implementing any change it is essential to understand and analyze the software product.
- Understanding the software during maintenance involves:
 - Having a general knowledge of **what** the software system does and how it relates to its environments.
 - Identifying **where** in the system changes are to be affected.
 - Having a good knowledge of **how** the parts to be corrected or modified.
- Program understanding and analyzing consumes a significant proportion of maintenance effort and resources.
 - At HP, reading code costs approximately 200 millions \$ a year
 - Other sources claim that about half of the total effort expended on effecting change is used up in understanding programs.
- This expenditure tends to increase when the programmer is different from the maintainer.

Software Comprehension and Analysis

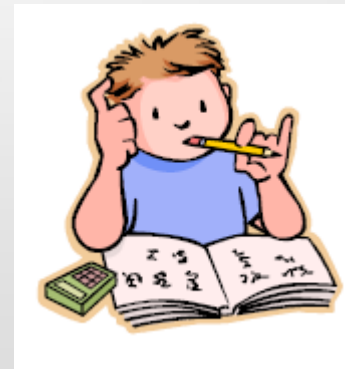
1. Aims of program comprehension and Analysis
2. Maintainers information needs
3. Comprehension process models
4. Program comprehension and analysis strategies
5. Factors that affect understanding
6. Why program comprehension and analysis are difficult

Software Comprehension and Analysis

1. Aims of program comprehension and Analysis
2. Maintainers information needs
3. Comprehension process models
4. Program comprehension and analysis strategies
5. Factors that affect understanding
6. Why program comprehension and analysis are difficult

Aims of Program Comprehension and Analysis

- The ultimate goal of reading and comprehending programs is to be able successfully to implement requested changes.
- This entails acquiring information about certain aspects:
 - Problem domain
 - Execution effect
 - Cause-effect relation
 - Product-environment relation, and
 - Decision support features



Aims of Program Comprehension and Analysis

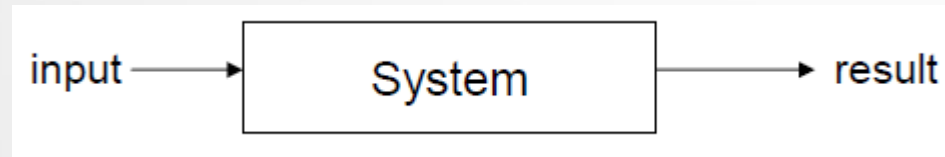
1. Problem Domain:

- In large systems, problems are usually broken down into manageable sub-problems or smaller elements. (example compiler: parser, lexical analyzer, etc.)
- In order to perform a change, knowledge of:
 - the problem domain in general, and
 - the sub-problems in particularis essential so as to direct the choice of algorithms, methodologies and tools.
- Information can be obtained from various sources:
 - System documentation
 - End-users
 - Program source code.

Aims of Program Comprehension and Analysis

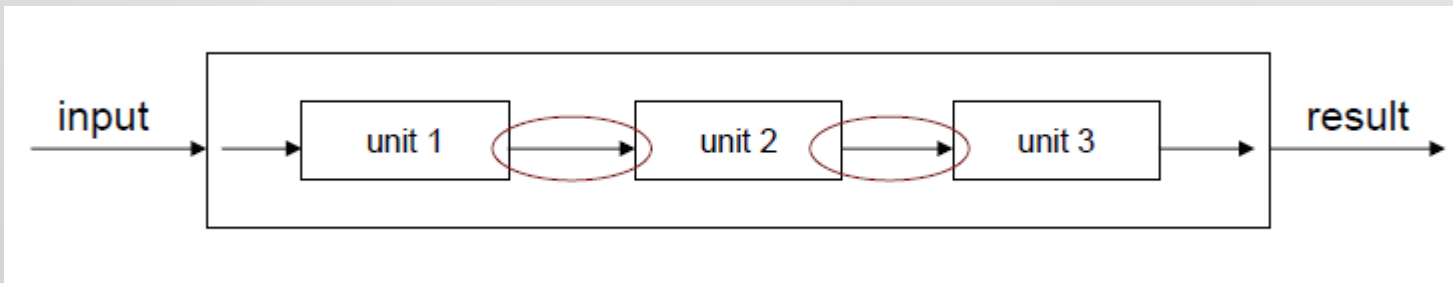
2. Execution effect:

- At a high level of abstraction, the maintenance personnel need to know what results the program will produce for a given input.



It is not important to know how the result was accomplished

- At a low level of abstraction, they need to know the results that individual program units will produce on execution.



Aims of Program Comprehension and Analysis

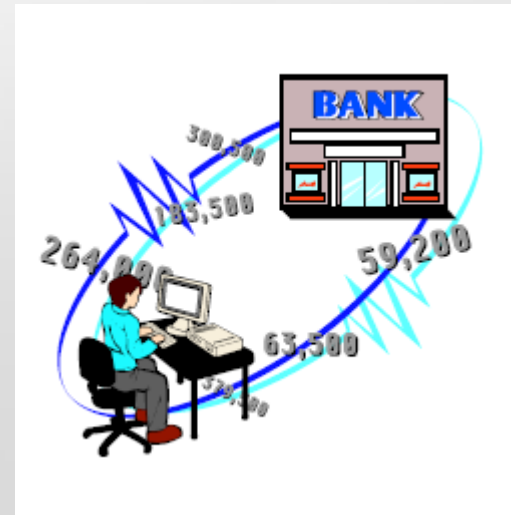
3. Cause-Effect Relation:

- In large and complex systems, knowledge of cause-effect relation is important in a number of ways:
 - It allows the personnel to reason about how components interact during execution.
 - It enables to predict the scope of a change and any effect that may arise from the change.
 - It can be used to trace the flow of information through the program. An interruption in this flow may signal the source of a bug.

Aims of Program Comprehension and Analysis

4. Product-Environment Relation:

- A product is a software system.
- An environment is the totality of all conditions and influences which act upon the product. (example : business rules, government regulations, hardware operation platforms, etc.)
- It is essential for the maintenance personnel to know the nature as well as the extent of the relation.



Aims of Program Comprehension and Analysis

5. Decision-Support Features:

- Software product attributes such as: complexity and maintainability can guide maintenance personnel in decision making processes.
- Examples of these processes are:
 - Budgeting and resource allocation
 - Option analysis
 - decision-making, etc.
- Example:
 - Measures of the complexity of the system can be used to determine which components of the system require more resource for testing.



Software Comprehension and Analysis

1. Aims of program comprehension and Analysis
2. **Maintainers information needs**
3. Comprehension process models
4. Program comprehension and analysis strategies
5. Factors that affect understanding
6. Why program comprehension and analysis are difficult

What Information the Maintainers Need ?

- It is not essential that every member of a maintenance project team understands every aspect of the system being maintained.
- The process of maintaining is driven by the need to know.
- The members of the maintenance team are:
 - Managers
 - Analysts
 - Designers
 - Programmers

What Information the Maintainers Need ?

1. Managers

- Managers need to have decision-support knowledge in order to make informed decisions.
- The level of understanding required will depend on the decision to be taken.
- Example:
 - To be able to estimate the cost of a major enhancement, knowledge of the size of the programs (lines of code or function points) is required.
- Managers do not need to know the architectural design or the low level implementation.

"Nobody can seriously have believed that executives could read programs"

Weinberg



What Information the Maintainers Need ?



2. Analysts

- During Software development, the analyst requires to understand the problem domain in order to:
 - Determine the functional and non-functional requirements
 - Establish the relationship between the system and the elements of its environment.
- During maintenance, analysts are concerned with:
 - Having a global view of the system (interaction between the major functional units)
 - Knowing how changes in this environment would affect the system
 - Determining the implications of change on the performance of the system.
- Like managers, analysts do not need the local view.

What Information the Maintainers Need ?

3. Designers



- The design process of a software system can take place at two levels:
 - Architectural design: production of functional components, and identification of the interconnection between all components.
 - Detailed design: production of detailed algorithms, data structures and interfaces between procedures.
- During maintenance, the designer:
 - Extract this information and determine how enhancements could be accommodated by the existing system (arch., data strut., etc.)
 - Go through the existing code to get an idea about the size of the job, the area that will be affected and the skills that will be needed.
- The use of concepts such as information hiding, modular decomposition, data abstraction, good design notations, etc. can help the designer obtain good understanding.

What Information the Maintainers Need ?



4. Programmers

- Programmers are required to know the execution effect of the system at different levels of abstraction:
 - **At a higher level:** The programmer needs to know the function of individual components of the system and their causal relation
 - **At a lower level:** the programmer needs to understand what each program statement does, the execution sequence, etc.
- This information will assist the programmer in a number of ways:
 - To decide on whether to restructure or rewrite specific code segments
 - To predict more easily the effect of making changes
 - To hypothesize the location and causes of error
 - To determine the feasibility of changes
- The use of tools such as static analyzers, cross-references, and program slicers can facilitate the programmer's task.

Software Comprehension and Analysis

1. Aims of program comprehension and Analysis
2. Maintainers information needs
3. Comprehension process models
4. Program comprehension and analysis strategies
5. Factors that affect understanding
6. Why program comprehension and analysis are difficult

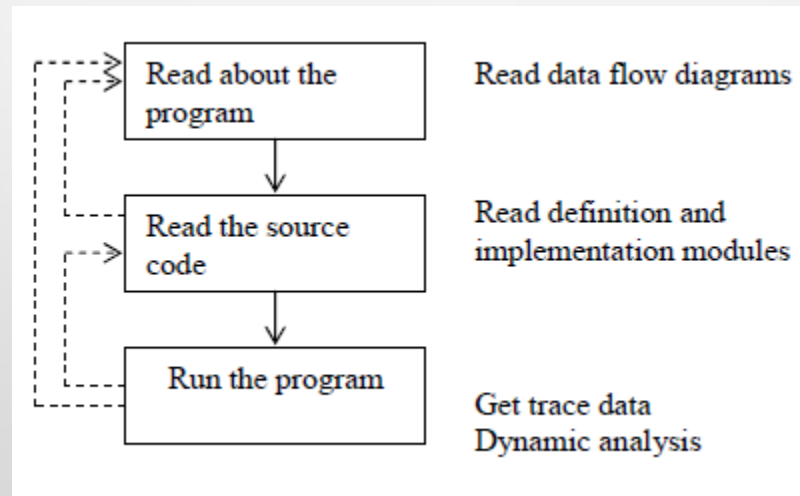
Comprehension Process Models

- Programmers vary in their way of thinking, solving problems and choosing techniques and tools
- Generally, the three actions involved in the understanding of a program are:
 - 1. Reading about the program:** the 'understander' browses different source of information (specification and design documents) to develop an overview of the system
 - 2. Reading the source Code:**
 - During this stage, global and local views can be obtained.
 - The local view allows programmers to focus their attention on a specific part of a system
 - Static analyzer are used to produce cross-reference lists that indicate where different functions and procedures are called.

Comprehension Process Models

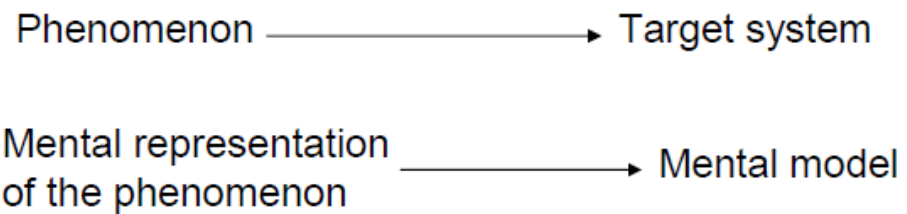
3. Running the program: allows to study the dynamic behaviour of the program in action. For example by obtaining the trace data.

- In practice, the process of understanding a program does not usually take place in such an organized manner.
- There tend to be iteration of the actions and backtracking.



Mental Models

- Our understanding of a phenomenon depends to some extent on our ability to form mental representation.
- The mental representation serves as a working model of the phenomenon to be understood
- A phenomenon can be : how a television set works, the behaviour of liquid, an algorithm, etc.



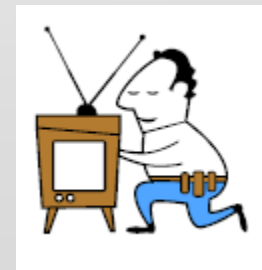
Mental Models



Example:

If you understand how a television works, $\xrightarrow{\text{then}}$ You have a mental model that represents this

- Based on that model you can predict behaviour such as what will happen when a different channel is selected
- Using this model, you can also explain certain observations (e.g. distorted image)
- A technician who repairs the TVs needs a more elaborate and accurate mental model



Mental Models

- The mental model is formed after observation, inference or interaction with target system.
- It changes continuously as more information about the target system is acquired.
- The mental model may contain insufficient, contradictory or unnecessary information about the target system.
- Research in the area of programmer's cognitive behaviour during maintenance suggests that there are variations in the strategies that programmers use to understand programs.



Software Comprehension and Analysis

1. Aims of program comprehension and Analysis
2. Maintainers information needs
3. Comprehension process models
4. Program comprehension and analysis strategies
5. Factors that affect understanding
6. Why program comprehension and analysis are difficult

Program Comprehension Strategies

- A program comprehension strategy is a technique used to form a mental model of the target program.
- The mental model is constructed by combining information contained in the source code and documentation.
- A number of comprehension strategies have been proposed, among them:
 1. Top-Down Model
 2. Bottom-Up Model
 3. Opportunistic Model

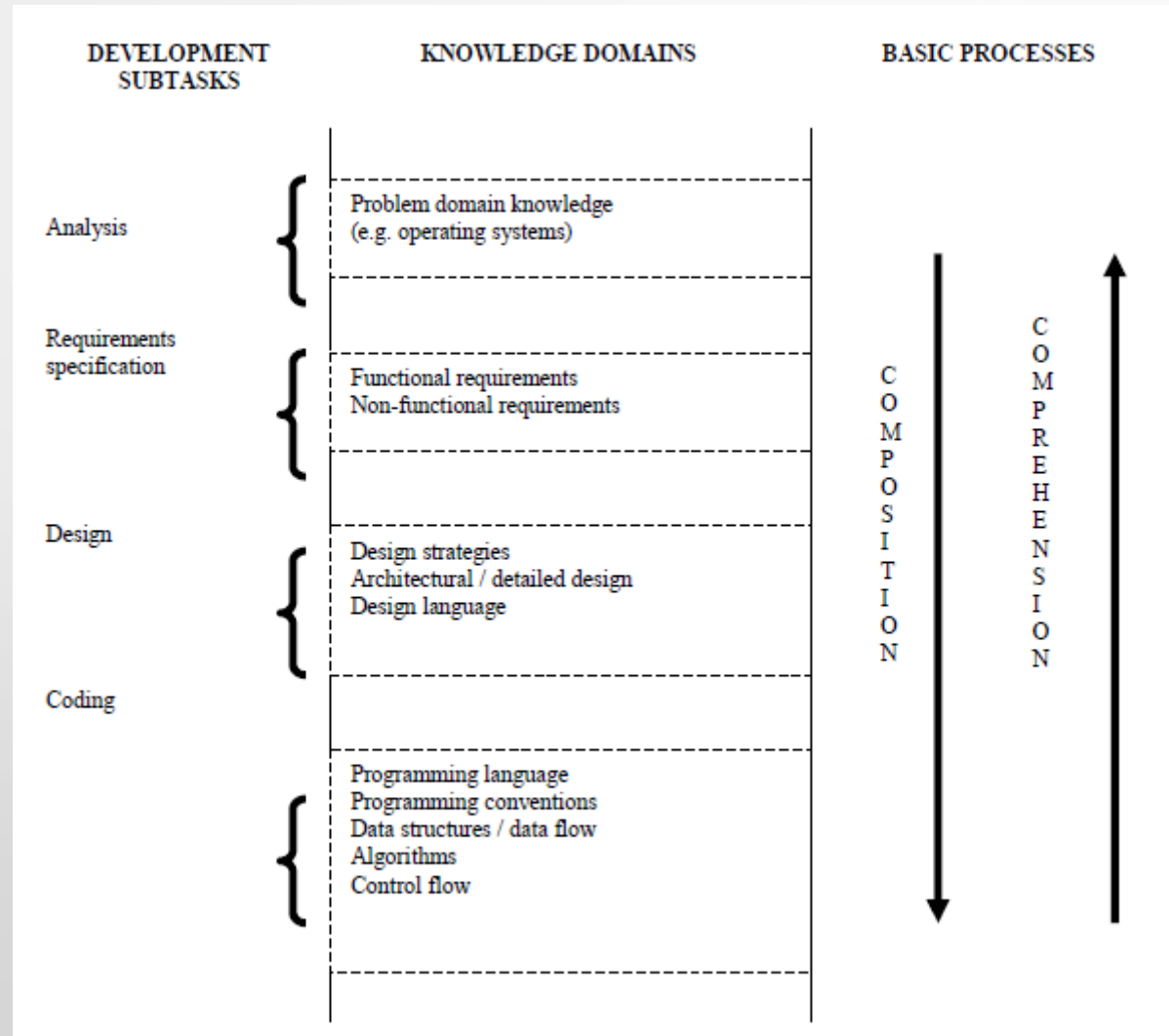
Program Comprehension Strategies

1. Top-Down Model

- The 'understander' starts by comprehending the top-level details of a program.
- Then, gradually works towards understanding the low-level details such as data-types, algorithmic patterns in a top-down fashion.
- Software development in its entirety can be considered as a design task consisting in two processes:
 - **Composition** (production of a design): a transformation from the problem domain to the programming domain.
 - **Comprehension** (understanding of that design): a transformation from the programming domain to the problem domain

Program Comprehension Strategies

1. Top-Down Model (Knowledge Domain):



Program Comprehension Strategies

1. Top-Down Model

- The reconstruction process is concerned with the creation, confirmation and successive refinement of hypotheses.
- It commences with a general hypothesis (the ***primary hypothesis***).
 - The mental model of the program begins to be constructed at the outset, even before the programmer becomes aware of low-level semantic and syntactic details of the program.
- This hypothesis is confirmed and further refined on acquisition of more information from the program code and system documentation.

Program Comprehension Strategies

1. Top-Down Model

- The understander begins verification of a hypothesis when the hypothesis deals with operations that can be associated with visible details found in the program code.
- The terms *beacons* describes those details that show the presence of a particular structure or operation.
 - E.g., a beacon for the hypothesis “sort”
- Beacons are important because they are the first link between the top-down hypotheses and the actual program text.
- Experiments showed that, the effect of experts recalling beacons lines more successfully than non-beacon lines (77% vs. 47%)

Program Comprehension Strategies

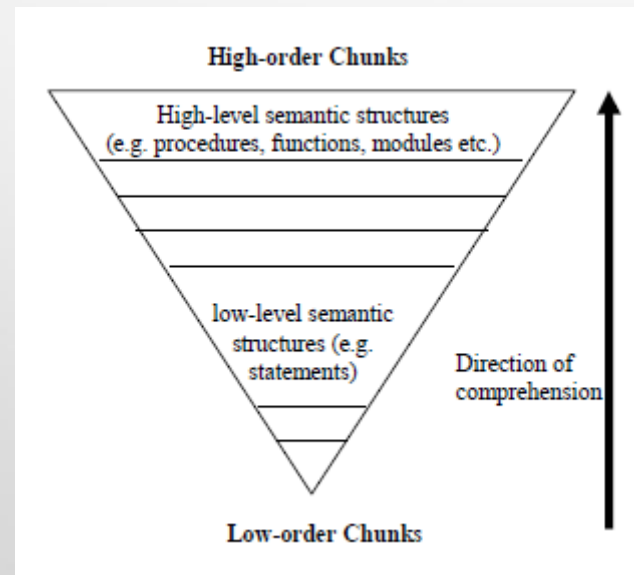
1. Top-Down Model

- Four sources of variants in the act of comprehension
 - The functionality of the program being understood
 - How do programs that perform different computations vary in comprehensibility?
 - The program text
 - Why do programs written in different languages differ in comprehensibility, even for the same problem?
 - The understander's task
 - Why does the comprehension process vary depending on motivations?
 - The understander's individual experience
 - Why does one understander find a program easier to comprehend than does another?

Program Comprehension Strategies

2. Bottom-Up / Chunking Model

- The programmer successively recognizes patterns in the program.
- These patterns are iteratively grouped into high-level, semantically more meaningful structures.
- The high-level structures are then chunked together into bigger structures in a bottom up fashion until the program is understood.



Program Comprehension Strategies

- The *main weaknesses* of top-down and bottom-up comprehension strategies are:
 - These models fail to take into consideration the contribution that other factors such as the available support tools make to understanding.
 - The fact that the process of understanding a program rarely takes place in such a well-defined fashion.

Program Comprehension Strategies

3. Opportunistic Model

- Programmers tend to take advantage of any clues they come across in an opportunistic way.
- The 'understander' makes use of both bottom-up and top-down strategies.
- Comprehension hinges on three key and complementary features:
 - **A knowledge base**: represents the expertise and background knowledge that the maintainer brings to the understanding task.
 - **A mental model**: the programmer's current understanding of the target program.
 - **Assimilation process**: this describes the procedure used to obtain information from various sources.

Reading Techniques



- Reading techniques are instructions given to the software maintainer on how to read in a software product
- An important issue in reading is to focus on the specific context:
 - What is the purpose of this reading exercise ?
 - Is it to find errors ?
 - Is it to analyze for specific characteristics ?
 - Is it for reuse purposes ?
- Experiments have shown that if the reading technique is focused on the goal, it is more effective in achieving that goal.
- The motivation to study and understand reading techniques is to develop defined and effective processes, rather than relying upon traditional *ad hoc* approaches.



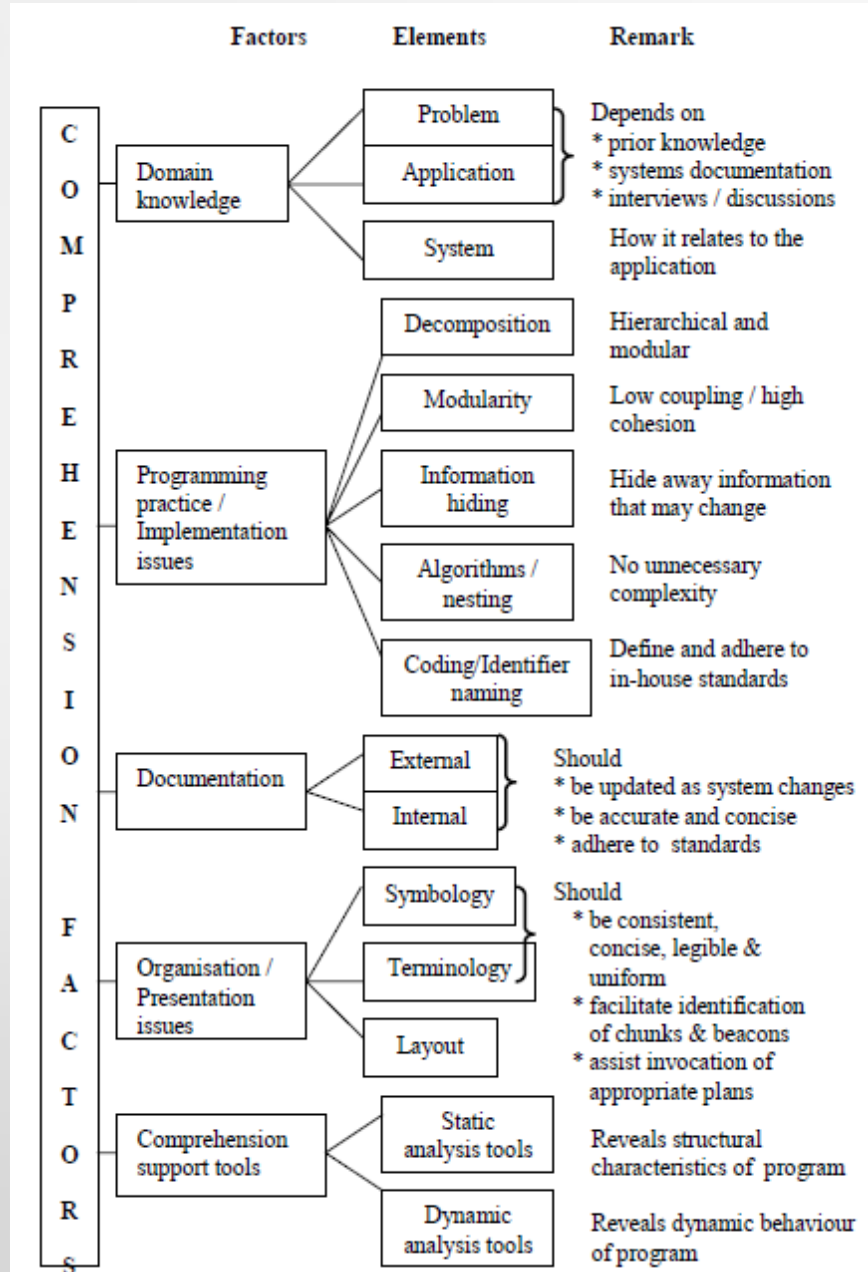
Software Comprehension and Analysis

1. Aims of program comprehension and Analysis
2. Maintainers information needs
3. Comprehension process models
4. Program comprehension and analysis strategies
5. **Factors that affect understanding**
6. Why program comprehension and analysis are difficult

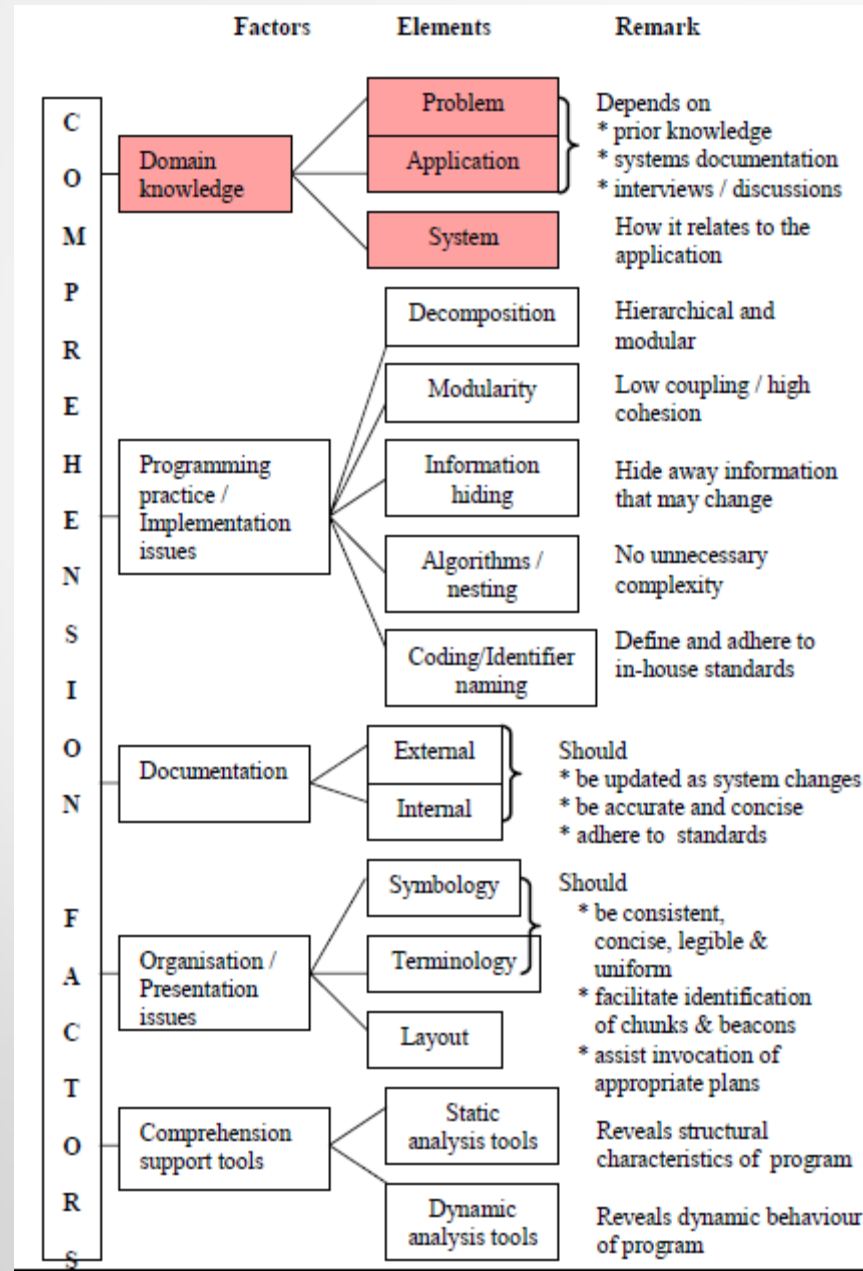
Factors that Affect Understanding

- A number of factors can affect the formation of mental models of a program, their accuracy, correctness and completeness.
- These factors include:
 - Expertise
 - Programming practice and implementation issues
 - Documentation
 - Program organization and presentation
 - Support tools
 - Evolving Requirements

Factors that Affect Understanding



Factors that Affect Understanding



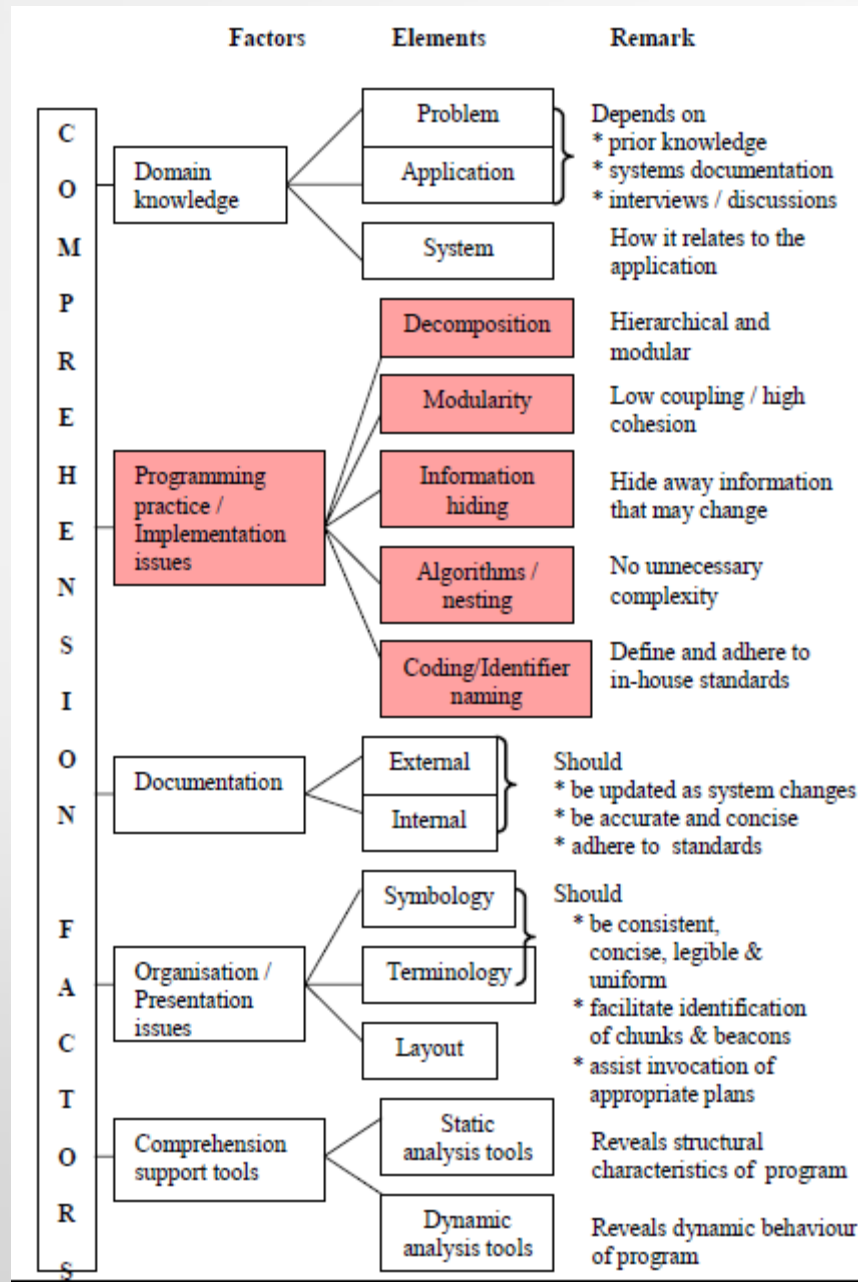
Factors that Affect Understanding

1. Expertise:

- Programmers become experts in a particular application domain thanks to the repertoire of knowledge and skills they acquire from working on that domain.
- Many studies revealed that experts tend to perform better than novices.
- Petre explains this by the following:
 - Experts differ from novices in both their breadth and their organization of knowledge.
 - Experts store information in larger chunks organized in terms of underlying abstractions.
 - This organization facilitates quick recognition of problem types.



Factors that Affect Understanding



Factors that Affect Understanding

2. Implementation issues:

- Naming style:
 - Meaningful identifier names can provide clues that assist programmers to invoke appropriate plans during understanding
 - Some experimental studies reveal that: identifier names were found to affect the comprehension of high-level procedural language programs by novices but not experts.
 - Another study, which investigate the presence of beacons in a sort program, only expert programmers were observed to recall the beacon lines better than the non-beacon lines.

Factors that Affect Understanding

2. Implementation issues:

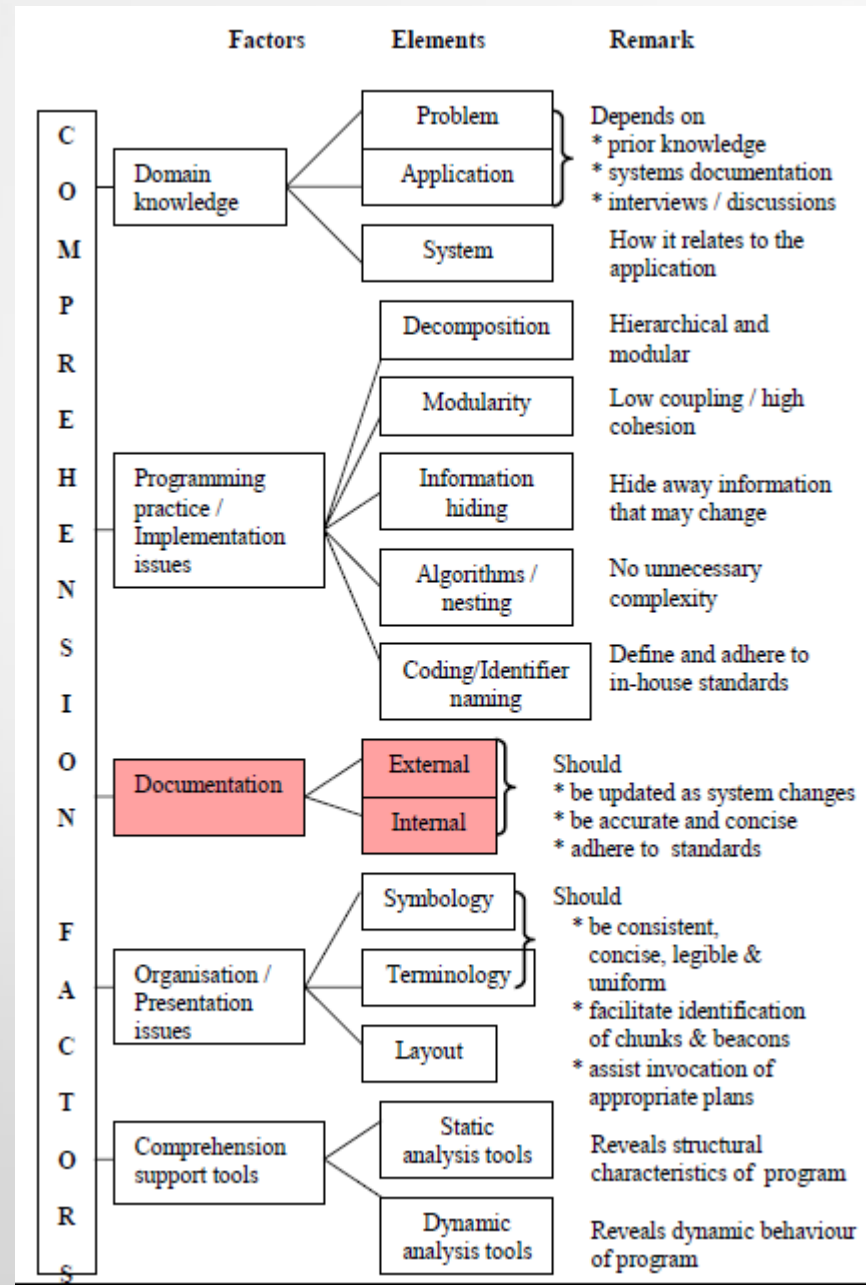
- **Comments:**
 - Program comments usually convey information about :
 - Functionality
 - Design decisions
 - Assumptions
 - Declarations
 - Algorithms
 - Nature of input and output data.
 - Common types of comments are:
 - Prologue comments: precede a program or a module and describe goals
 - In-line comments: describe how these goals are achieved
 - Empirical studies revealed that comments are useful only if they provide additional information.
 - It is the quality of the comment that is important not its presence.

Factors that Affect Understanding

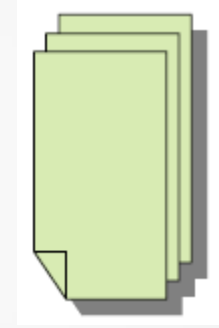
2. Implementation issues:

- **Decomposition mechanism:**
 - One of the important factors that affect comprehension of programs is their complexity.
 - One way to deal with this complexity depends on how the entire software system has been decomposed.
 - Modular decomposition and structured programming can be used.
 - Modular decomposition:
 - Divides large software systems into manageable components.
 - Modularization should be done around the sources of change.
 - Psychologists believe that a modular version is a lot easier to comprehend than a non-modular program
 - Structured programming:
 - Using high-level programming languages in order to reduce the size and complexity of programs to manageable proportions.

Factors that Affect Understanding



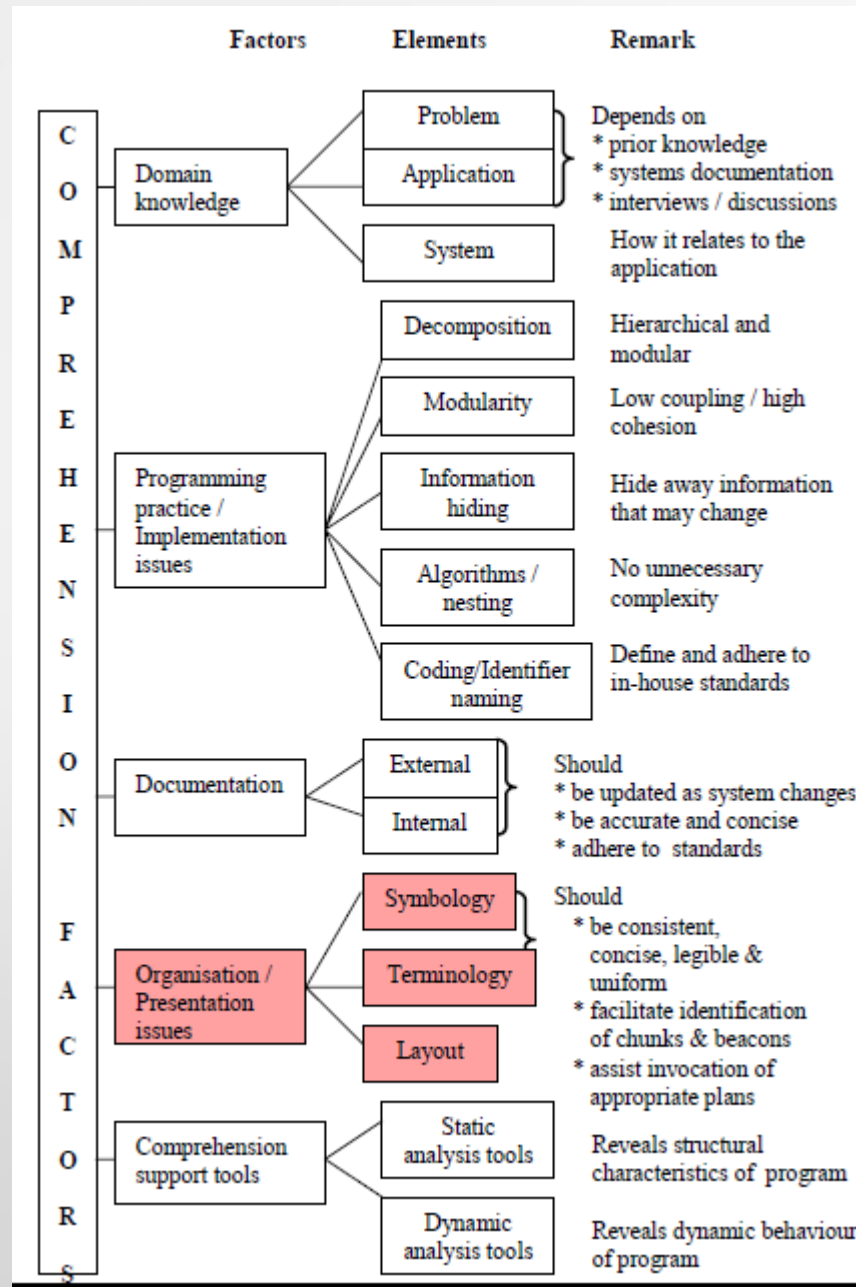
Factors that Affect Understanding



3. Documentation:

- The maintainer must be able to have access to as much information about the system as possible.
- It is not always possible to contact the original authors of the system.
- System documentation can be very useful in this respect.
- Maintainers need to have access to the system documentation to enable them to understand:
 - The functionality
 - The design,
 - The implementation, and other issues.
- If the documentation is inaccurate or non-existent, the maintainer has to resort to the source code documentation (comments).

Factors that Affect Understanding



Factors that Affect Understanding

4. Organization and presentation of programs:

- Reading the program source code is an important aspect of maintenance
- Program should be organized and presented in a manner that facilitates browsing, visualization and ultimately understanding.
- Presentation can improve understanding by:
 - Facilitating a clear and correct expression of the mental model
 - Emphasizing the control flow of the program's hierarchic structure
 - Visually enhancing the source code through the use of :
 - Indentation
 - Spacing
 - Boxing, and
 - Shadowing

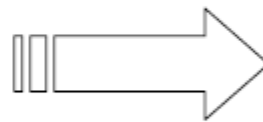


Factors that Affect Understanding

4. Organization and presentation of programs:

```
FOR i:=1 TO NumEmployees DO
LowPos:=i
Smallest:=Employees[LowPos].EmployeeAge;
FOR j:=i+1 TO NumEmployees DO
IF Employee[j].EmployeeAge < Smallest THEN
LowPos:=j;
Smallest:=Employee[j].Employee.Age
END (*IF*)
END (*FOR j *)
TempRec:=Employee[LowPos];
Employee[SmallPos]:=Employee[i];
Employee[i]:=TempRec
END (*FOR i *)
```

A: No blank lines



```
FOR i:=1 TO NumEmployees DO
LowPos:=i
Smallest:=Employees[LowPos].EmployeeAge;
```

```
FOR j:=i+1 TO NumEmployees DO
```

```
IF Employee[j].EmployeeAge<Smallest THEN
LowPos:=j;
Smallest:=Employee[j].Employee.Age
END (*IF*)
```

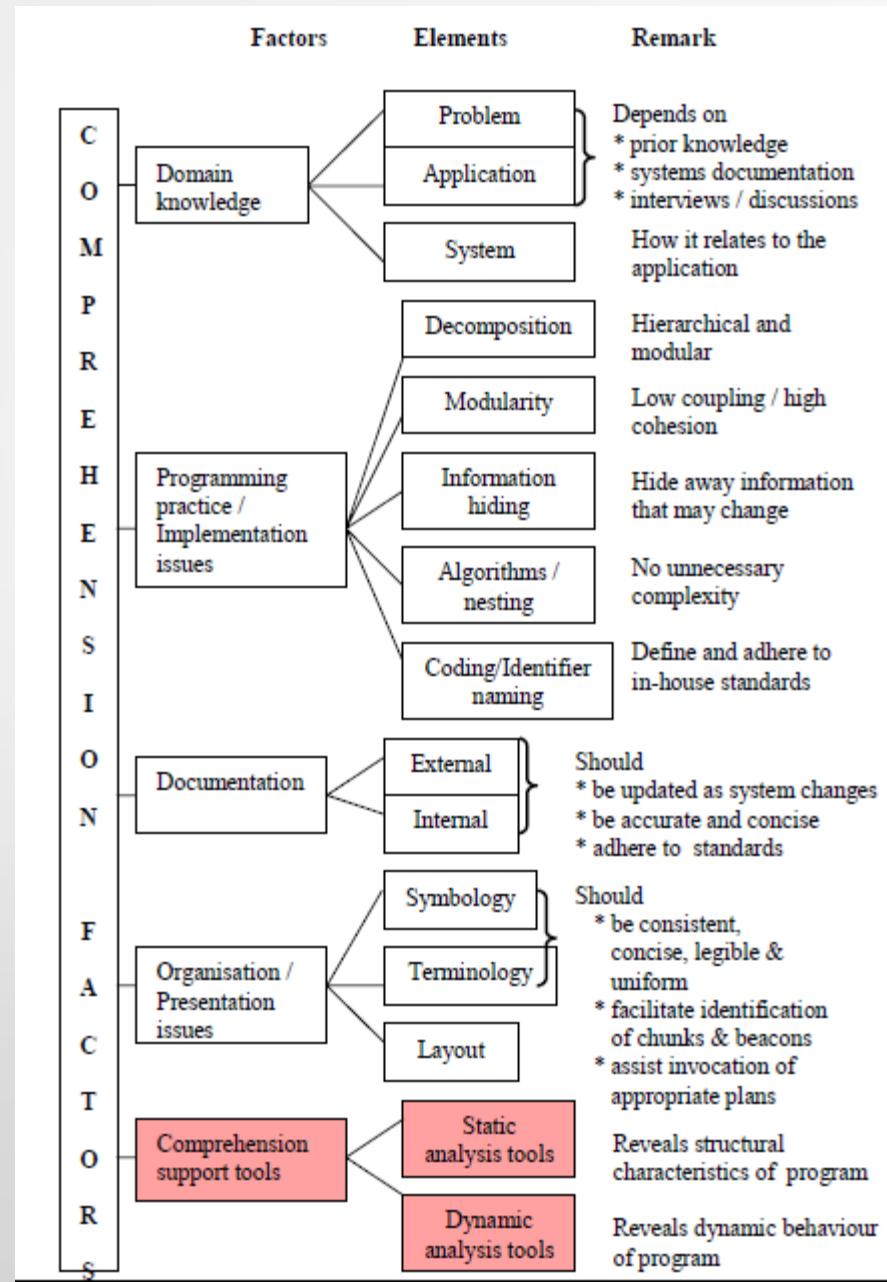
```
END (*FOR j *)
```

```
TempRec:=Employee[LowPos];
Employee[SmallPos]:=Employee[i];
Employee[i]:=TempRec
```

```
END (*FOR i *)
```

B: Blank lines / indentation / boxes / shading

Factors that Affect Understanding



Software Comprehension and Analysis

1. Aims of program comprehension and Analysis
2. Maintainers information needs
3. Comprehension process models
4. Program comprehension and analysis strategies
5. Factors that affect understanding
6. Why program comprehension and analysis are difficult

Why program comprehension and analysis are difficult?

Program comprehension and analysis are difficult because it must bridge the following gaps:

1. The gap between a problem from some application domain and a solution to it in some programming language.
2. The gap between the concrete world of physical machines and computer programs and the abstract world of high level design descriptions.
3. The gap between what the original designers had envisioned (coherent and structured description of a system) and the actual system (whose structure may have disintegrated over time).

Why program comprehension and analysis are difficult?

Program comprehension and analysis are difficult because it must bridge the following gaps:

4. The gap between hierarchical world of computer programs and associational nature of human cognition.
5. The gap between the bottom-up analysis of the source code and top-down synthesis of the description of the application.

Computer programs and abstract descriptions

- The program reader is faced with vast amount of details.
- The reader must create an abstract representation of the program from the mass of concrete details; Must decide on the important concepts.
- The abstraction process is not linear, i.e. a given section of a program may be part of several abstractions.
- The abstractions are *interleaved*.

Coherent models and incoherent artifacts

- When a program is originally constructed, the design stage creates a coherent structure of details.
- Design representations may become out-of-date by the time program comprehension is required.
 - Original structure may have deteriorated through bug fixing, adaptability, etc.
- The program reader must understand the current purpose the program serves (even if the original purpose has been changed).
- The program may serve to accomplish several purposes by the time of program comprehension.

Bottom-up program analysis and top-down model synthesis

- When we look at a program, we detect patterns that indicate the intent of some section of code.
- Low level patterns are part of higher level constructs intended to accomplish larger purposes
- This way, analysis proceeds bottom-up.
- At the same time, we have some idea of the overall purpose of the program and how it might be accomplished; The overall concept is refined into more lower level details.
- This synthesis process, proceeds top-down.
- Analysis and synthesis need to proceed at the same time in a relative synchronized fashion.

Model of comprehension

- Schneiderman views the comprehension of programs as consisting of three levels:
 1. Low-level comprehension of the function of each line of code.
 2. Mid-level comprehension of the nature of the algorithms and data.
 3. High-level comprehension of overall program function.
- It may be possible to do [1] but not achieve [3] (or vice versa).
- [2] involves knowledge of control- and data structures; it is possible to be achieved without [1] or [3].
- Detailed comprehension involves all three levels of understanding.

Summary

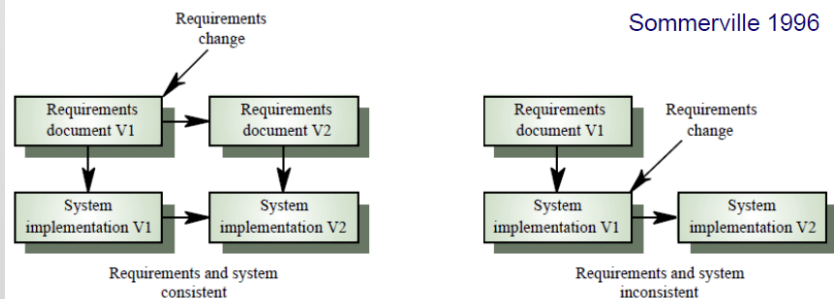
- The **process of understanding** software products is at the heart of all maintenance activities and accounts for over half the effort spent on effecting change.
- **Information needs** of maintenance personnel vary with their responsibilities.
- During comprehension of a system, the 'understander' forms an internal representation (**mental model**). The model may be incomplete but becomes more complete and accurate as additional information is obtained.
- Three principal strategies of program comprehension are **top-down**, **bottom-up** and **opportunistic**.
- Several **factors** affect program comprehension (expertise, programming practice, availability of documentation, organization and presentation of programs, etc.)

References

- **[Rug95]** Spencer Rugaber, *Program comprehension*, 1995.
- **[May95]** Anneliese von Mayrhauser, *Program comprehension during software maintenance and evolution*, IEEE Computer, August 1995, pp. 44-55.
- **[BCLS96]** Victor Basili, Gianluigi Caldiera, Filippo Lanubile, and Forrest Shull, *Studies on Reading Techniques*, Proceedings of the 21th Annual Software Engineering Workshop, December 1996.
- **[SLBoo]** Forrest Shull, Filippo Lanubile, and Victor R. Basili, *Investigating Reading Techniques for Object-Oriented Framework Learning*, IEEE Transactions on Software Engineering, Vol. 26, No. 11, November, 2000.
- **[BFG07]** Tibor Bakota, Rudolf Ferenc, and Tibor Gyimothy, *Clone Smells in Software Evolution*, ICSM, 2007.

Requirements Evolution

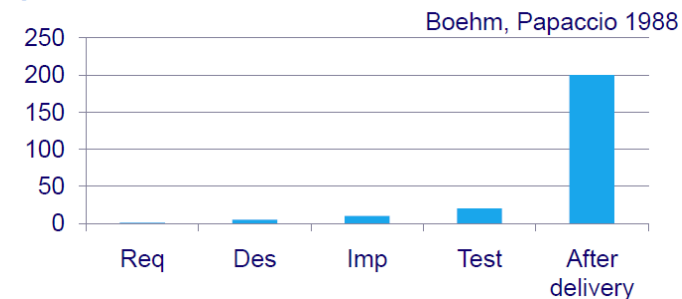
- Requirements are “just a piece of paper”...
- Tend to be forgotten during the evolution



This is how it should be...

this is how it often is.

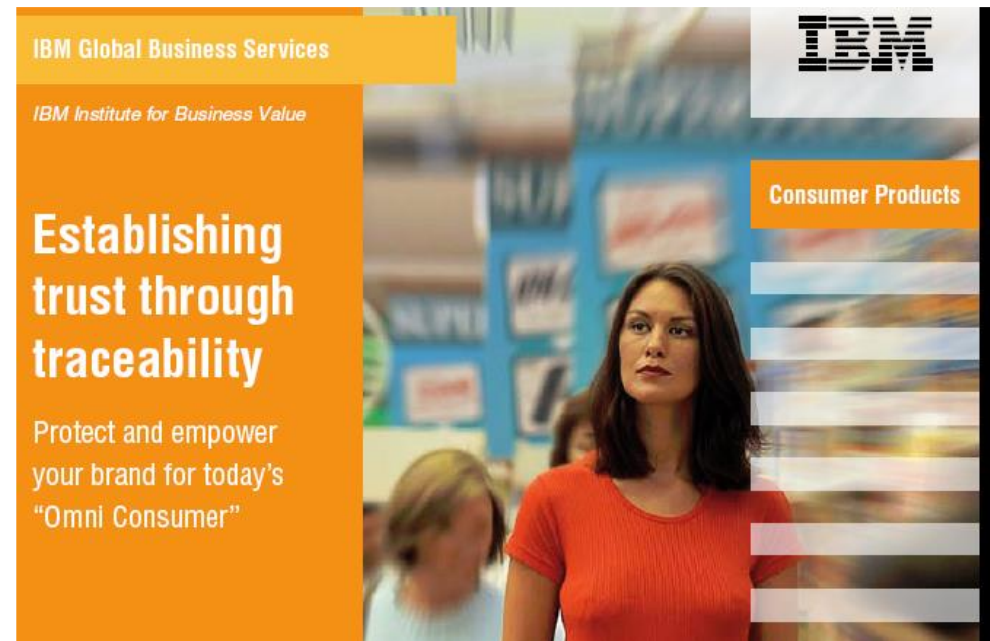
- Errors in requirements are:
 - **common:**
 - 25% of all the errors (Jones '91)
 - 1 error per function point (~ 80 LOC Java; Jones '95)
 - **expensive**




Traceability ...

already a major factor in our daily lives

Traceability ...
already a major factor
in other domains



To ensure safety of the food supply and other consumer products, as well as enhance credibility with consumers, consumer products companies can turn to full value traceability to track product ownership throughout the supply chain.

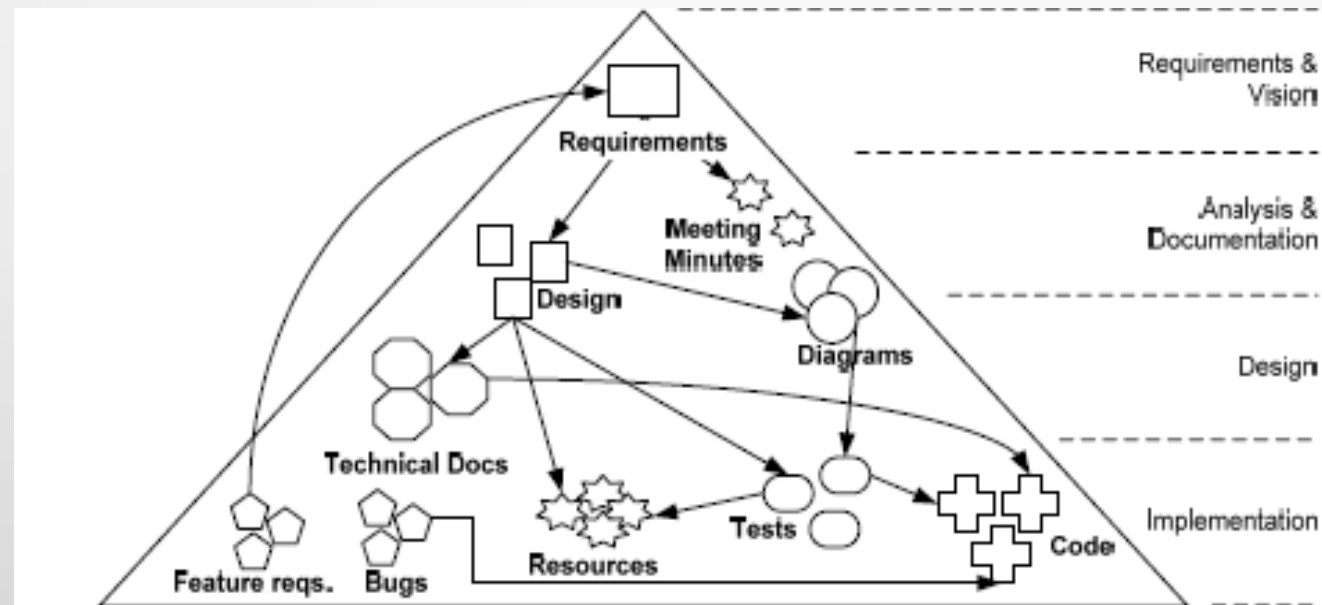


You can't manage what you can't trace.
[Watkins and Neal, 1994]

Robert Watkins, Mark Neal: Why and How of Requirements Tracing. IEEE Software 11(4): 104-106 (1994)

Definitions: Software Traceability

- ❑ “Traceability [...] a document in question is in agreement with a predecessor document to which it has a hierarchical relationship. [DoD 1988]
- ❑ A Software Requirements Specification (SRS) is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.” [IEEE 1998].



[IEEE 1998] IEEE, Std. 830: Guide to Software Requirements Specification,” 1998.

[DoD 1988] DoD, Std-2167A: U.S. Department of Defense Military Standard: Defense System Software Development, Washington, D.C. 1988.



Traceability gives essential assistance in understanding the relationships that exist within and across software requirements, design, and implementation. [Palmer, 2000]

J. D. Palmer. Traceability. In R. H. Thayer and M. Dorfman, editors, Software Requirements Engineering, Second Edition, pages 412-422. IEEE Computer Society Press, 2000.

Traceability Quotes (1)



Requirements traceability refers to the ability to describe and follow the life of a requirement, in both forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of ongoing refinement and iteration in any of these phases)".¹



A software requirements specification is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation.²



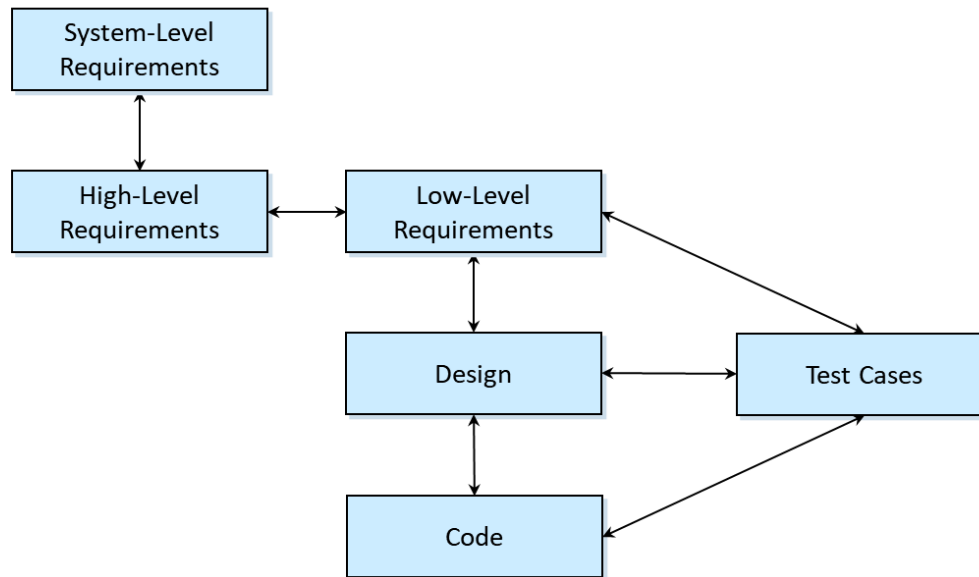
Traceability gives essential assistance in understanding the relationships that exist within and across software requirements, design, and implementation.³



A link or relationship defined between entities.⁴

DOD-178B

At all stages of the process, traceability is required.



Traceability Quotes (2)

[1-2] Watkins and Neal, 1994; [3] Kotonya and Sommerville, 1998; [4] Greenspan, McGowan, 1978

- Traceability is often mandated by contracts and standards.¹
 - E.g., military and aerospace
- One cannot manage what cannot be traced.²
- Traceability information helps assess the impact of changes to requirements, connecting these requirements as well as requirements for other representations of the system.³
- Traceability is a property of a system description technique that allows changes in one of the three system descriptions – requirements, specifications, implementation – to be traced to the corresponding portions of the other descriptions. The correspondence should be maintained through the lifetime of the system.⁴

Importance of Traceability (1)

- Requirements cannot be managed effectively without requirements traceability

A requirement is traceable if you can discover:

- who suggested the requirement, why the requirement exists,
- what requirements are related to it and
- how that requirement relates to other information such as systems designs, implementations and user documentation

Importance of Traceability (2)

Benefits of traceability

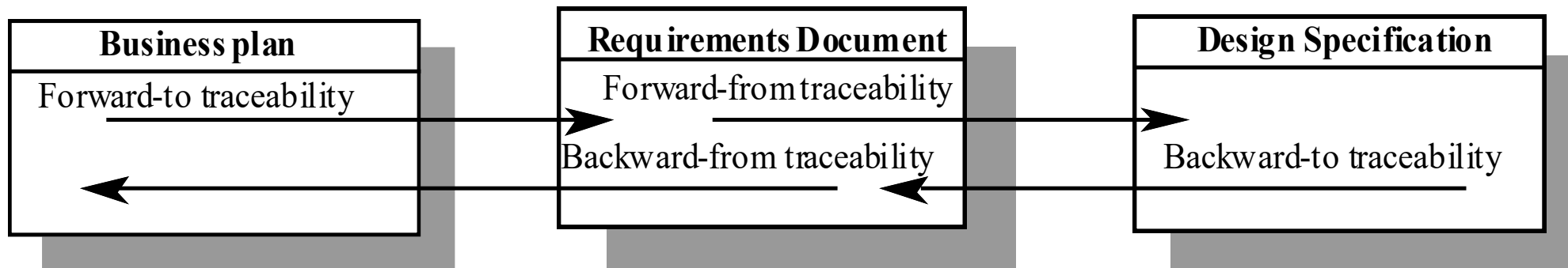
- Prevents losing knowledge
- Supports the verification process (certification, localization of defects)
- Impact analysis
- Change control
- Process monitoring (e.g., missing links indicate completion level)
- Improved software quality (make changes correctly and completely)
- Reengineering (define traceability links is a way to record reverse engineering knowledge)
- Reuse (by identifying what goes with a requirement: design, code...)
- Risk reduction (e.g., if a team member with key knowledge leaves)

Traceability Difficulties

- Various stakeholders require different information
- Huge amount of requirements traceability information must be tracked and maintained
- Manual creation of links is **very** demanding
 - Likely the most annoying problem
- Specialized tools must be used
- Integrating heterogeneous models/information from/to different sources (requirements, design, tests, code, documentation, rationales...) is not trivial
- Requires organizational commitment (with an understanding of the potential benefits)

Backward and Forward Traceability (1)

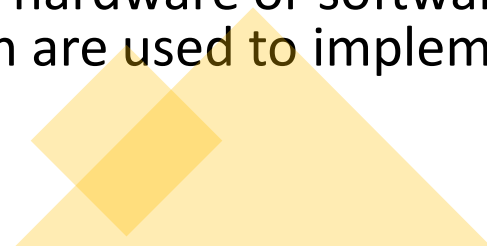
- **Backward traceability**
 - To previous stages of development
 - Depends upon each element explicitly referencing its source in earlier documents
- **Forward traceability**
 - To all documents spawned by a document
 - Depends upon each element in the document having a unique name or reference number



Source of figure: Kotonya and Sommerville



Types of Traceability (1)

- Requirements – source traceability
 - Links requirements with a person or document
 - Requirements – rationale traceability
 - Requirements – requirements traceability
 - Links requirements with other requirements which are, in some way, dependent on them
 - Requirements – architecture traceability
 - Links requirements with the subsystems where these requirements are implemented (particularly important where subsystems are being developed by different subcontractors)
 - Requirements – design traceability
 - Links requirements with specific hardware or software components in the system which are used to implement the requirement
- 

Types of Traceability (2)

- Requirements – interface traceability
 - Links requirements with the interfaces of external systems which are used in the provision of the requirements
- Requirements – feature traceability
- Requirements – tests traceability
 - Links requirements with test cases verifying them (used to verify that the requirement is implemented)
- Requirements – code traceability
 - Generally not directly established, but can be inferred

Representation – Traceability Table

- Show the relationships between requirements or between requirements and other artifacts
- Table can be set up to show links between several different elements
- Backward and forward traceability
- Difficult to capture different types of links

User Requirement	Functional Requirement	Design Element	Code Module	Test Case
UC-28	catalog.query.sort	Class Catalog	catalog.sort()	search.7 search.8
UC-29	catalog.query.import	Class Catalog	catalog.import(), catalog.validate()	search.12 search.13 search.14

Representation – Traceability Matrix

Depends-on

	R1	R2	R3	R4	R5	R6
R1			*	*		
R2					*	*
R3				*	*	
R4		*				
R5						*
R6						



Define links
between pairs of
elements

E.g., requirements to
requirement, use
case to requirement,
requirement to test
case...



Can be used to
defined
relationships
between pairs

E.g., specifies/is
specified by,
depends on, is
parent of,
constrains...



More amenable to automation
than traceability table

The chain of success

increased traceability



more effective
communication



increased flexibility



higher success rates



leads to

Controversial statement(s)

Agile development versus Traceability?

- Back to the Waterfall model !!!??

Should we mandate/legislate traceability?

Traceability vs. privacy and security?

Open source and traceability?

Incomplete documents/artifacts

- No complete requirements
- No source code (COTS)

- Back to requirements evolvability.....

Volatile requirements ?

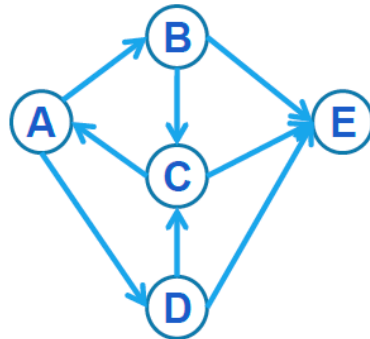
- At requirements engineering time:
 - Try to find a more stable alternative
 - Put special attention to traceability (backwards – rationale, forwards – design, implementation, tests)
 - Anticipate and record responses for future changes
- At design time:
 - Encapsulate volatile requirements in separate modules

- “More stable requirements should not depend on less stable ones”
- A affects B (B depends on A) if changing A might require changing B.



Requirements Dependency analysis - Traceability

- Traceability graph
 - Vertices: requirements
 - Arcs: dependency relations
- What do you think about the requirements document right?
- What does this mean for evolution?
- How would the quality information influence your interpretation?
- What are the limitations of the traceability graph approach?



Inconsistent requirements due to:

- Different **kinds** of requirements: use cases, process models, natural language requirements
- Different **sources** of requirements: multiple documents, multiple stakeholders
- Different types of inconsistencies:
 - **Terminological** (synonyms, different interpretations of the same term): data dictionary, glossary, ontology
 - **Logical (strong)** – conjunction of the requirements is *false*
 - **Logical (weak)** – under some condition conjunction of the requirements is *false*

Traceability

To support:

- (Co-)evolution
 - Change impact analysis
 - Maintenance
 - Reengineering
- Certification
- Project tracking
- Reuse
- Risk reduction
- Testing

Conclusions:

- Requirements often evolve due to environmental changes
- Suitability for evolution: quality, volatility, dependencies, inconsistency
- Evolution:
 - Continuous change and growth
 - System architecture is “almost” stable
- Co-evolution
 - Need for backward and forward traceability