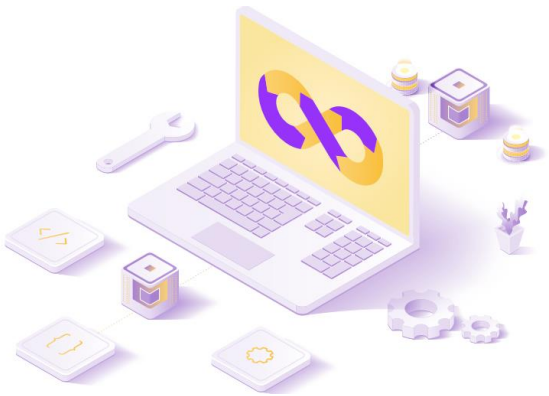
University of Buea,
College of Technology (COT), Department of Computer Engineering - 2023/2024 Academic Year

CEC418: Software Construction and Evolution

Enjoy Your Software Construction Ride!!



Unit 1

Unit 1: Software Construction Fundamentals

1.1 Introduction

- ⇨ The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging.
- ⇨ The Software Construction knowledge area (KA) is linked to all the other KAs, but it is most strongly linked to Software Design and Software Testing because the software construction process involves significant software design and testing.
- ⇨ It is often part of an iterative "**design → construct → test**" cycle at the heart of most development processes.
- ⇨ The process is one of going from a design to implemented code.
- ⇨ Code is generally much more complex than the design and will require a myriad of detailed design or implementation activities.
- ⇨ The process uses the design output and provides an input to testing ("design" and "testing" in this case referring to the activities, not the KAs).
- ⇨ Boundaries between design, construction, and testing (if any) will vary depending on the software life cycle processes that are used in a project.
- ⇨ Although some detailed design may be performed prior to construction, much design work is performed during the construction activity.
- ⇨ Thus, the Software Construction KA is closely linked to the Software Design KA.
- ⇨ Throughout construction, software engineers both unit test and integration test their work.
- ⇨ Thus, the Software Construction KA is closely linked to the Software Testing KA as well.
- ⇨ Software construction typically produces the highest number of configuration items that need to be managed in a software project (*source files*, *documentation*, *test cases*, and so on).
- ⇨ Thus, the Software Construction KA is also closely linked to the Software Configuration Management KA.
- ⇨ While software quality is important in all the KAs, code is the ultimate deliverable of a software project, and thus the Software Quality KA is closely linked to the Software Construction KA.
- ⇨ Since software construction requires knowledge of algorithms and of coding practices, it is closely related to the Computing Foundations KA, which is concerned with the computer science foundations that support the design and construction of software products.

- ⇨ It is also related to project management, insofar as the management of construction can present considerable challenges.
- ⇨ Figure 1.1 gives a graphical representation of the top-level decomposition of the breakdown for the Software Construction KA.

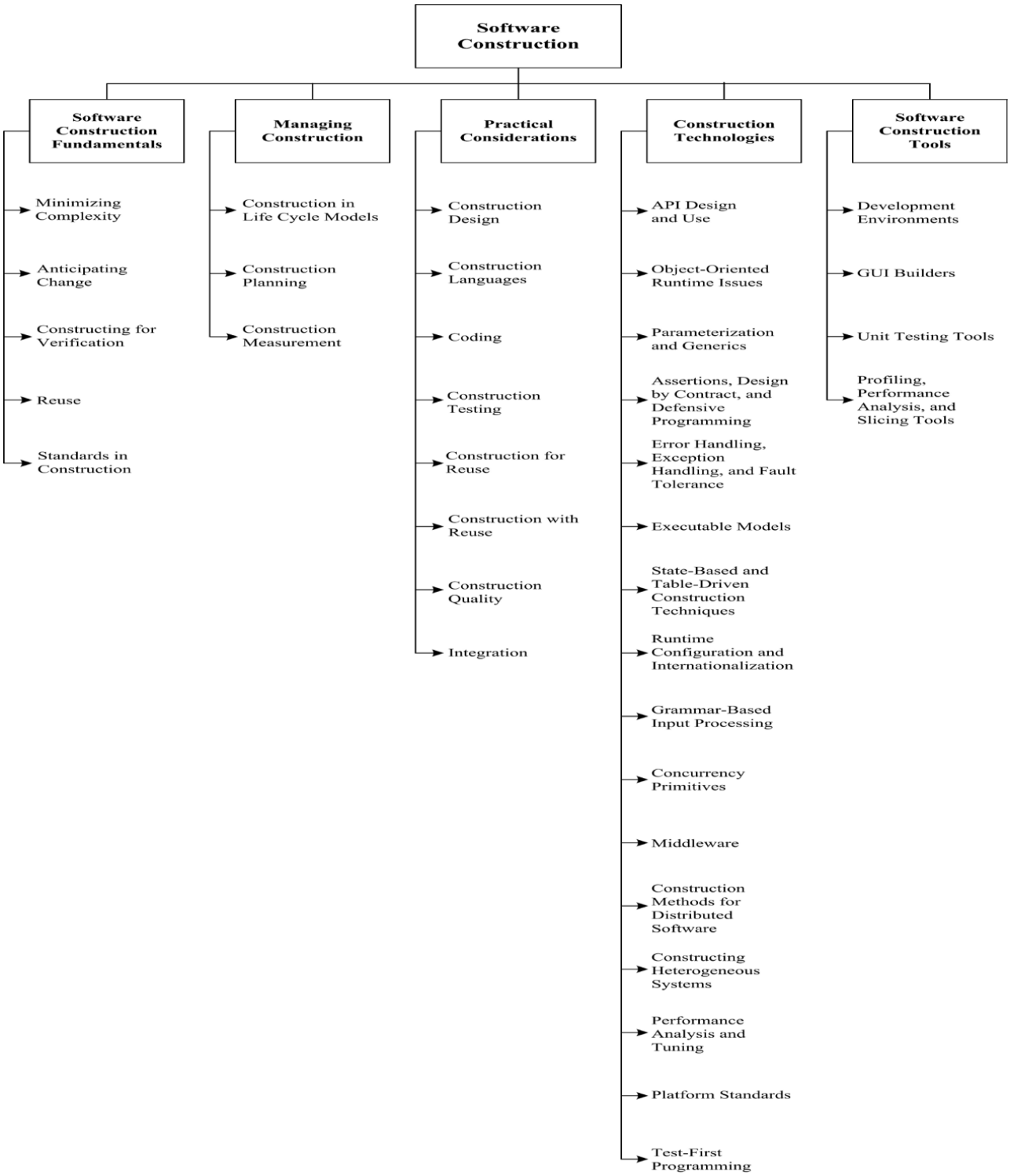


Figure 1.1: Breakdown of Topics for the Software Construction KA.

1.2 Managing and Minimising Complexity

- Most people are limited in their ability to hold complex structures and information in their working memories, especially over long periods of time.
- This proves to be a major factor influencing how people convey intent to computers and leads to one of the strongest drives in software construction: *minimising complexity*.
- The need to reduce complexity applies to essentially every aspect of software construction and is particularly critical to testing of software constructions.
- In software construction, reduced complexity is achieved through emphasizing code creation that is simple and readable rather than clever.
- It is accomplished through making use of standards (see section 1.6, Standards in Construction) modular design, and numerous other specific techniques (we shall elaborate on these aspects in subsequent units).
- It is also supported by construction-focused quality techniques.
- A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes.
- Many things that make writing software fun also make it complex and error-prone:
  - joy of solving puzzles and building things from interlocking moving parts
  - stimulation of a non-repeating task with continuous learning
  - pleasure of working with a tractable medium, 'pure thought stuff'
  - complete flexibility – you can base the output on the inputs in any way you can imagine
  - satisfaction of making stuff that's useful to others
- Software engineering helps to reduce this programming complexity.
- Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition

1.2.1 Mitigating the Impact of Complexity in Software Models

- Safety-critical systems—such as those used in the avionics, aerospace, medical, and automotive domains—are becoming extremely software reliant.
- The size of avionics software has increased enormously in the past few decades: as illustrated in Figure 1.2, the size of military avionics software increased from 135 KSLOC (thousand source lines of code) to 24 MSLOC (million source lines of code) in less than 40 years.

- ⇨ This trend is also true for other domains: automotive systems contain more and more electronic functions and, as space is scarce, most of these functions are implemented using software rather than hardware, which also facilitates system upgrade and maintenance.
- ⇨ Because failure of such systems might have important consequences, software for safety-critical systems must be designed and certified against stringent certification standards as mentioned above.
- ⇨ A problem with developing such large commercial programs is that the complexity and difficulty levels of the programs increase exponentially with their sizes.
- ⇨ Software engineering helps to reduce this programming complexity.
- ⇨ Software engineering principles use two important techniques to reduce problem complexity: abstraction and decomposition.

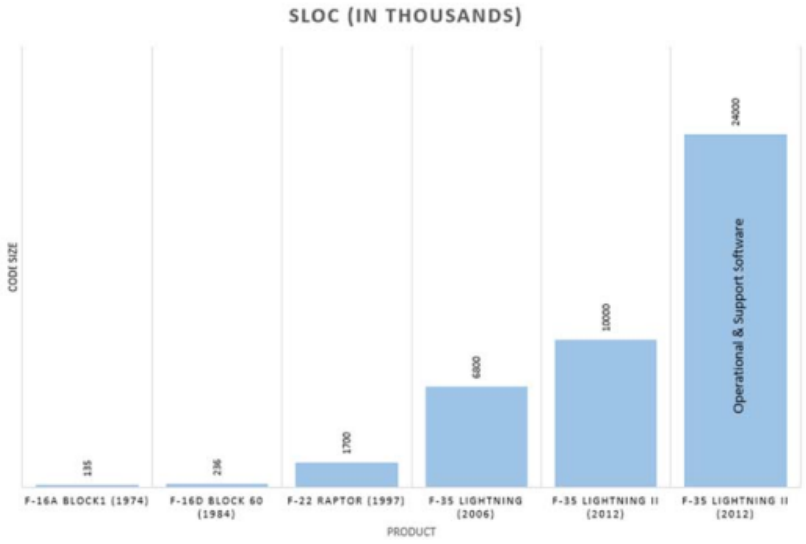


Figure 1.2: Growth of Avionics Software in KSLOC.

- ⇨ An increase of software volume (i.e., number of lines of code [LOC]) implies that more components are executed by different computers and connected through networks using specific protocols.
- ⇨ This increasing use of software also increases its complexity: interconnected components perform various functions, potentially at different criticality levels (e.g., entertainment system vs. cruise control), and designers must ensure isolation among these different criticality levels.
- ⇨ According to certification standards, a system must be free of any design errors and must be tested and validated to demonstrate that a component with low criticality cannot interfere with a component of high criticality.

- ⇨ Compliance with these requirements is expensive: related verification activities—such as code review, unit testing, and integration testing—require significant effort in terms of manpower and are tightly dependent on the software size.
- ⇨ Reducing software size and complexity would not only lower certification costs but also reduce software maintenance effort.
- ⇨ During the past few years, safety-critical software development has been moving toward model-based engineering (MBE) approaches, which consist of abstracting software concerns in models with specific notation.

1.2.2 Toward a Definition of Software Complexity

- ⇨ When applied to a system, the term *complexity* has different dimensions and associated understandings.
- ⇨ One definition of complexity is based on the "ease of understanding" a software model representing a software system design specified by an engineer.
- ⇨ Two components of this dimension are:
  - ➢ ***the notation***: A clear, unambiguous language with a precise semantics eases the understanding of the model.
  - ➢ ***the user***: A well-trained user who is familiar with a notation will take less time to understand and maintain a model.
- ⇨ Obviously, a model that is easy to understand is also easier to use and maintain, which reduces the likelihood that a user will introduce errors when updating it.
- ⇨ For that reason, the notation should be designed to reduce or mitigate complexity.
- ⇨ In addition, tools must be designed so that they guide users not familiar with the notation and help them both understand the model and avoid design mistakes.

1.2.3 Essential and Avoidable Complexity

- ⇨ Evidence shows that software complexity has increased significantly over time not only because of the increase in number of functions but also because of a paradigm shift in which more functions are realised using software rather than hardware.
- ⇨ This prevalence of software usage induces an organic increase of software complexity, a type of complexity required to implement all the necessary functions.
- ⇨ This is the *essential complexity* of a system, and it cannot be mitigated by any means other than changing the design (removing functions or implementing them in hardware instead of in software).
- ⇨ However, complexity can be also introduced through poor design and implementation techniques.

- ⇨ For example, too many interfaces on a component or too many nesting levels make software review and analysis difficult.
- ⇨ This is the *avoidable complexity* of a system, and it can be removed by refactoring the system and using appropriate design patterns.
- ⇨ The total software complexity is the sum of the essential and the avoidable complexity.
- ⇨ The essential complexity is related to the system functions and cannot be mitigated.
- ⇨ On the other hand, the avoidable complexity is not necessary and incurs potential rework, refactoring, or re-engineering efforts.
- ⇨ For that reason, it should be avoided as much as possible and as early as possible

1.2.4 Model-Based Engineering (MBE) on Software Complexity

- ⇨ Traditional software development techniques that implement solutions in embedded systems, control systems, integrated modular avionics, and so forth are reaching scalability limits in both cost and assurance.
- ⇨ The volume, complexity, and cost of software development for safety-critical systems—such as in avionics, aerospace, defense, medical, or automotive systems— are trending upward.
- ⇨ An important reason behind constructing a model is that it helps manage complexity.
- ⇨ Once models of a system have been constructed, these can be used for a variety of purposes during software development, including the following:
  - ➢ Analysis
  - ➢ Specification
  - ➢ Code generation
  - ➢ Design
  - ➢ Visualise and understand the problem and the working of a system
  - ➢ Testing, etc.
- ⇨ In all these applications mentioned above, the UML models can not only be used to document the results but also to arrive at the results themselves.
- ⇨ Since a model can be used for a variety of purposes, it is reasonable to expect that the model would vary depending on the purpose for which it is being constructed.
- ⇨ For example, a model developed for initial analysis and specification should be very different from the one used for design.
- ⇨ A model that is being used for analysis and specification would not show any of the design decisions that would be made later on during the design stage.

- ⇨ On the other hand, a model used for design purposes should capture all the design decisions.
- ⇨ Therefore, it is a good idea to explicitly mention the purpose for which a model has been developed, along with the model.
- ⇨ Figure 1.3 shows the trend in system size for avionic systems.
- ⇨ Software volume, measured by lines of code, is on a trend to double every four years.
- ⇨ The Figure includes software development cost estimates using COCOMO II of approximately $300 per line of code (LOC).
- ⇨ High costs result from the need for highly skilled developers, the level of verification required for certification and a growing test volume.
- ⇨ Although high cost is often attributed to the need for low defect density, it may be more correct to attribute cost to both the methods used to produce the software and the requirements for certification of its quality.

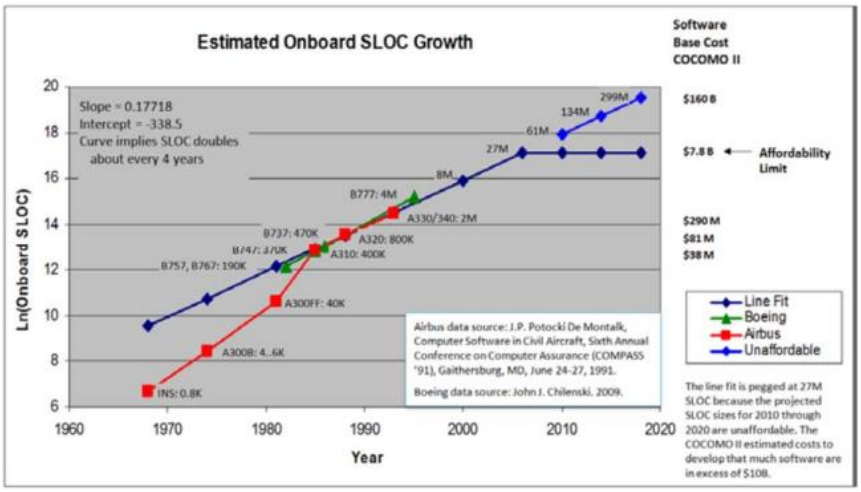


Figure 1.3: Increasing Size of Onboard Avionic Software.

- ⇨ Another aspect of the increasing cost of software is that defect rates, using conventional development methods for high-assurance safety-critical systems, are commonly believed to be approximately 100 defects per MSLOC, or more than a factor of 10 better than leading commercial software.

1.2.5 Complexity Reduction

- ⇨ MBE reduces or hides complexity (such as complexity related to code production and testing activities).
- ⇨ Reduction may result from improved design using formal notation at a higher level of abstraction.
- ⇨ At a minimum, however, MBE hides much of the complexity (cyclomatic complexity) at the code level within the module.

- ⇨ This hidden complexity represents code that need not be built by hand but can be machine generated.
- ⇨ There is substantial evidence that cyclomatic complexity is linearly correlated with product size at least within a domain, as illustrated by Figure 1.4.

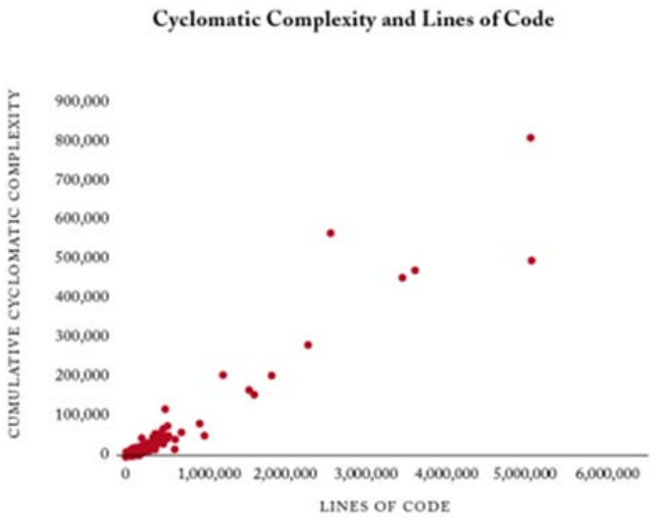


Figure 1.4: Cyclomatic Complexity Increases with Size.

1.2.6 Cyclomatic Complexity

- ⇨ For more complicated programs it is not easy to determine the number of independent paths of the program.
- ⇨ McCabe's cyclomatic complexity defines an upper bound for the number of linearly independent paths through a program.
- ⇨ Also, the McCabe's cyclomatic complexity is very simple to compute.
- ⇨ Thus, the McCabe's cyclomatic complexity metric provides a practical way of determining the maximum number of linearly independent paths in a program.
- ⇨ Though the McCabe's metric does not directly identify the linearly independent paths, but it informs approximately how many paths to look for.
- ⇨ There are three different ways to compute the cyclomatic complexity.
- ⇨ The answers computed by the **three** methods are guaranteed to agree.

Method 1:

Given a control flow graph G of a program, the cyclomatic complexity V(G) can be computed as:

$V(G) = E - N + 2$

where N is the number of nodes of the control flow graph and E is the number of edges in the control flow graph.

For the CFG of example shown in Figure 1.5, E = 7 and N = 6. Therefore, the cyclomatic complexity = 7 - 6 + 2 = 3.

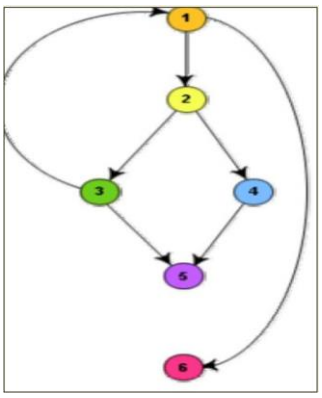


Figure 1.5: Control Flow Diagram.

Method 2:

An alternative way of computing the cyclomatic complexity of a program from an inspection of its control flow graph is as follows:

V(G) = Total number of bounded areas + 1

In the program's control flow graph G, any region enclosed by nodes and edges can be called as a ***bounded area***. This is an easy way to determine the McCabe's cyclomatic complexity.

But, what if the graph G is not planar, i.e. however you draw the graph, two or more edges intersect? Actually, it can be shown that structured programs always yield planar graphs. But, presence of GOTO's can easily add intersecting edges. Therefore, for non-structured programs, this way of computing the McCabe's cyclomatic complexity cannot be used.

The number of bounded areas increases with the number of decision paths and loops. Therefore, the McCabe's metric provides a quantitative measure of testing difficulty and the ultimate reliability. For the Control Flow Graph (CFG) example shown in Figure 1.5, from a visual examination of the CFG the number of bounded areas is 2. Therefore, the cyclomatic complexity, computing with this method is also 2 + 1 = 3. This method provides a very easy way of computing the cyclomatic complexity of CFGs, just from a visual examination of the CFG. On the other hand, the other method of computing CFGs is more amenable to automation, i.e. it can be easily coded into a program which can be used to determine the cyclomatic complexities of arbitrary CFGs.

Method 3:

The cyclomatic complexity of a program can also be easily computed by computing the number of decision statements of the program. If N is the number of decision statement of a program, then the McCabe's metric is equal to N+1.

- ⇨ Apart from the McCabe's metric, there also exist the Halstead Complexity Metrics.
- ⇨ Halstead metrics aim to estimate the programming effort.
- ⇨ The metrics can be applied to preliminary software design artifacts (pseudo code) as well as to the code itself.
- ⇨ The metrics are based on counting various characteristics of the software system
- ⇨ Another metric is the Zage Complexity Metrics which focus on two measures:
  - ➢ D_e: an external metric related to the component's external interfaces and other interactions within the system
  - ➢ D_i: an internal metric related to the component's internal structure (algorithms, internal data flows, etc.)

The global metrics are then the sum of both D_e and D_i.

D_e, the external measure, is defined as follows for a particular component in the system:

$$D_e = e_1 * (\#inputs * \#outputs) + e_2\ (\#fanin * \#fanout),$$

where e_1 and e_2 are weighting factors defined by the user.

- ⇨ D_i, the internal measure, is defined as follows:

$$D_i = i_1 * CC + i_2\ DSM + i_3\ I/O,$$

Where i_1, i_2, and i_3 are weighting factors defined by the user

- ➢ CC is the number of procedure invocations
- ➢ DSM is the number of references to complex data types (pointers, records, arrays, etc.)
- ➢ I/O is the number of input/output operations (read a device, write in a file, receive data from the network, etc.)

- ⇨ The external metric focuses on the external interfaces (what is using or used by the component) and does not consider the data flow in the architecture (in contrast to other metrics, such as McCabe's).
- ⇨ The internal metric is very precise but can be difficult to map into a programming language.
- ⇨ For example, some concepts are not present or are hidden by some languages: the number of I/Os is often abstracted by the language, and the number of references to complex data types is language dependent (e.g., in Java, any object variable can be counted as a pointer).

- There are many other metrics to analyse software and to characterise and identify complexity. In some respects, the McCabe, Halstead, and Zage metrics also use these values to provide a better view of complexity.
- However, in some cases, the raw data alone can be a good indicator of software complexity.
- One simple approach is to count the program elements and identify whether there are too few, too many, or just enough elements.
- Some coding guidelines use program counting approaches to evaluate the complexity and quality of a piece of code.
- For example, in the Linux kernel coding style, the code should have fewer than four levels of indentation to keep it simple to understand.
- Many similar techniques can give developers insight into software complexity, such as the number of functions, use of specialised data types, number of arguments for a method or function, and the number of methods within a class.

1.3 Anticipating Change

- Most software will change over time, and the anticipation of change drives many aspects of
- software construction; changes in the environments in which software operates also affect software in diverse ways.
- Anticipating change helps software engineers build extensible software, which means they can enhance a software product without disrupting the underlying structure.
- Anticipating change is supported by many specific techniques (we shall look at details in the Coding section).
- It is impossible to produce systems of any size which do not need to be changed.
- Once software is put into use, new requirements emerge and existing requirements change as the business running that software changes.
- Parts of the software may have to be modified to correct errors that are found in operation, improve its performance or other non-functional characteristics.
- All of this means that, after delivery, software systems always evolve in response to demands for change.
- Software change is very important because organisations are now completely dependent on their software systems and have invested millions of dollars in these systems.
- Their systems are critical business assets and they must invest in system change to maintain the value of these assets.

- ⇨ A key problem for organisations is implementing and managing change to their legacy systems so that they continue to support their business operations.
- ⇨ There are a number of different strategies for software change:

  1. *Software maintenance*: Changes to the software are made in response to changed requirements but the fundamental structure of the software remains stable. This is the most common approach used to system change.

  2. ***Architectural transformation***: This is a more radical approach to software change then maintenance as it involves making significant changes to the architecture of the software system. Most commonly, systems evolve from a centralised, datacentric architecture to a client-server architecture.

  3. ***Software re-engineering***: This is different from other strategies in that no new functionality is added to the system. Rather, the system is modified to make it easier to understand and change. System re-engineering may involve some structural modifications but does not usually involve major architectural change.
- ⇨ The above strategies are not mutually exclusive.
- ⇨ Re-engineering may be necessary to make the software easier to understand before its architecture is changed or components are reused.
- ⇨ Some parts of a system may be replaced by off-the-shelf (COTS) components while other, stable, parts are maintained.
- ⇨ The choice of the most appropriate strategy depends on both the technical quality of the system and its business value.
- ⇨ Equally, different strategies may be applied to different parts of a system or to different programs that make up a legacy information system.
- ⇨ A program which is well-structured and which does not need frequent maintenance may be maintained; another program which is used by many different people in different locations may have its architecture modified so that its user interface runs on a client computer and a third program in the same system, may be replaced by an off-the-shelf alternative.
- ⇨ However, if the system data is re-engineered this will normally require changes to all of the programs in the system.
- ⇨ For some systems, none of these change strategies is appropriate and you have to replace the system.
- ⇨ It may be possible to replace part or all of the system with an off-the-shelf (COTS) system.
- ⇨ However, there may be no suitable COTS alternative and you therefore have to develop a new customised system.

- ⇨ In such a situation, it may be possible to reuse major parts of the original system in the development of the replacement.
- ⇨ Software change inevitably generates many different versions of a software systems and its components.
- ⇨ It is critically important to keep track of these different versions and to ensure that the appropriate versions of components are used in each system version.
- ⇨ The management of changing software products is called ***configuration management***.

1.3.1 Software Maintenance

- ⇨ Software maintenance is the general process of changing a system after it has been delivered.
- ⇨ The changes may be simple changes to correct coding errors, more extensive changes to correct design errors or significant enhancements to correct specification errors or accommodate new requirements.
- ⇨ Software maintenance does not normally involve major architectural changes to the system.
- ⇨ Changes are implemented by modifying existing system components and, where necessary, by adding new components to the system.
- ⇨ There are ***three*** different types of software maintenance:
  1. ***Maintenance to repair software faults***: Coding errors are usually relatively cheap to correct; design errors are more expensive as they may involve the re-writing of several program components. Requirements errors are the most expensive to repair because of the extensive system re-design which may be necessary.
  2. ***Maintenance to adapt the software to a different operating environment***: This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system or other support software changes. The application system must be modified to adapt it to cope with these environmental changes.
  3. ***Maintenance to add to or modify the system's functionality***: This type of maintenance is necessary when the system requirements change in response to organisational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.
- ⇨ In practice, there isn't a clear-cut distinction between these different types of maintenance.
- ⇨ Software faults may be revealed because a system has been used in an unanticipated way and the best way to repair these faults may be to add new functionality to help users with the system.
- ⇨ When adapting the software to a new environment, functionality may be added to take advantage of new facilities supported by the environment.

- Adding new functionality to a system may be necessary because faults have changed the usage patterns of the system and a side-effect of the new functionality is to remove the faults from the software.
- While these different types of maintenance are generally recognised, different people sometimes give them different names.
- Corrective maintenance is universally used to refer to maintenance for fault repair.
- However, adaptive maintenance sometimes means adapting to a new environment and sometimes means adapting the software to new requirements.
- Perfective maintenance sometimes means perfecting the software by implementing new requirements and, in other cases, maintaining the functionality of the system but improving its structure and its performance.
- It is difficult to find up-to-date figures for the relative effort devoted to the different types of maintenance, however, Figure 1.6 presents us one.

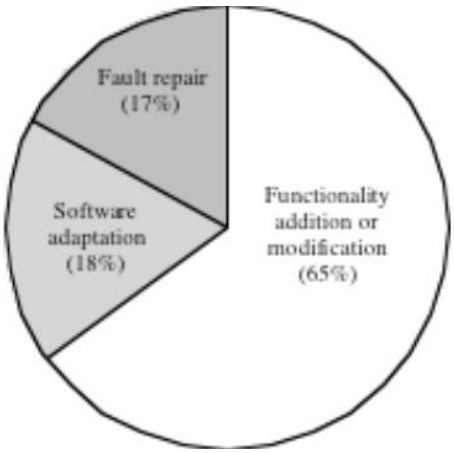


Figure 1.6: Maintenance Effort Distribution.

- It is usually cost-effective to invest effort when designing and implementing a system to reduce maintenance costs.
- It is more expensive to add functionality after delivery because of the need to understand the existing system and analyse the impact of system changes.
- Therefore, any work done during development to reduce the cost of this analysis is likely to reduce maintenance costs.
- Good software engineering techniques such as precise specification, the use of object-oriented development and configuration management all contribute to maintenance cost reduction.
- Figure 1.7 shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system.

⇨ One important reason why maintenance costs are high is that it is more expensive to add functionality after a system is in operation than it is to implement the same functionality during development.

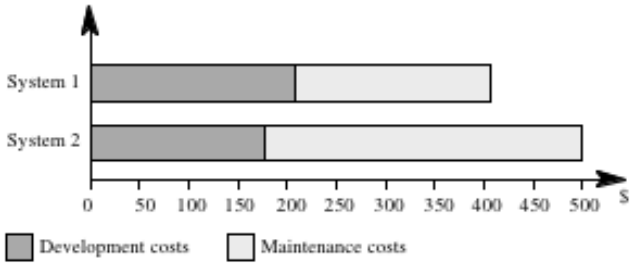


Figure 1.7: Development and Maintenance Costs.

⇨ The key factors that distinguish development and maintenance and which lead to higher maintenance costs are:

1. ***Team stability***: After a system has been delivered, it is normal for the development team to be broken up and people work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. A lot of the effort during the maintenance process is taken up with understanding the existing system before implementing changes to it.
2. ***Contractual responsibility***: The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write the software so that it is easy to change. If a development team can cut corners to save effort during development it is worthwhile for them to do so even if it means increasing maintenance costs.
3. ***Staff skills***: Maintenance staff are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers. It is seen as a less skilled process than system development and is often allocated to the most junior staff. Furthermore, old systems may be written in obsolete programming languages. The maintenance staff may not have much experience of development in these languages and must learn these languages to maintain the system.
4. ***Program age and structure***: As programs age, their structure tends to be degraded by change and so they become harder to understand and change. Old systems may not have been subject to configuration management so time is often wasted finding the right versions of system components to change.

- ⇨ The first three of these problems stem from the fact that many organisations still make a distinction between system development and maintenance.
- ⇨ Maintenance is seen as a second-class activity and there is no incentive to spend money during development to reduce the costs of system change.
- ⇨ The only long-term solution to this problem is to accept that systems rarely have a defined lifetime but continue in use, in some form, for an indefinite period.
- ⇨ Rather than develop systems, maintain them until further maintenance is impossible and then replace them, we have to adopt the notion of evolutionary systems.
- ⇨ Evolutionary systems are systems that are designed to evolve and change in response to new demands.
- ⇨ They can be created from existing legacy systems by improving their structure through re-engineering and by evolving the architecture.
- ⇨ The last issue in the list above, namely the problem of degraded system structure is in some ways, the easiest problem to address.
- ⇨ Re-engineering techniques may be applied to improve the system structure and understandability.
- ⇨ If appropriate, architectural transformation (discussed later in this chapter) can adapt the system to new hardware.
- ⇨ Preventative maintenance work (essentially incremental re-engineering) can be supported to improve the system and make it easier to change.
- ⇨ Maintenance processes vary considerably depending on the type of software being maintained, the development processes used in an organisation and the people involved in the process.
- ⇨ The maintenance process is triggered by a set of change requests from system users, management or customers.
- ⇨ The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.
- ⇨ If the proposed changes are accepted, a new release of the system is planned.
- ⇨ During release planning, all proposed changes (fault repair, adaptation and new functionality) are considered.
- ⇨ A decision is then made on which changes to implement in the next version of the system.
- ⇨ The changes are implemented and validated and a new version of the system is released.
- ⇨ The process then iterates with a new set of changes proposed for the new release. Figure 1.8 shows an overview of this process.

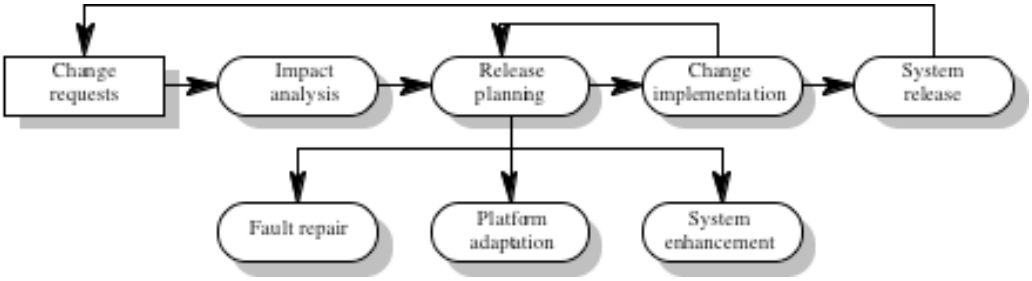


Figure 1.8: An Overview of the Maintenance Process.

1.3.2 Architectural evolution

- During system maintenance, most individual changes which are made are localised and do not affect the architecture of the system.
- There are a number of different drivers that contribute to this change:
  1. ***Hardware costs***: The costs of buying and maintaining a distributed client-server system are usually much less than the costs of buying a mainframe computer of equivalent power.
  2. ***User interface expectations***: Many legacy mainframe systems provide form-based, character interfaces. However, most users now expect graphical user interfaces and easier interaction with the system. These interfaces require much more local computation and can only be provided effectively in a client-server system.
  3. ***Distributed access to systems***: Companies are, increasingly, physically distributing their organisation rather than maintaining all facilities on a single site. Their computer systems may have to be accessed from different locations and from different types of equipment. Customers and staff may access systems from their homes and this has to be supported.
- By migrating to a distributed architecture, organisations can dramatically reduce hardware costs, can develop a system which has a more effective interface and a more modern 'look and feel' and can support distributed working.
- In the process of migration, there will inevitably be some conversion of the system to an object-oriented model and this is likely to reduce the costs of future system maintenance.
- However, modifying the architecture of a legacy system is a major challenge and is a very expensive process.
- Key factors which influence this decision are shown in Table 1.1.

Table 1.1: Factors Influencing System Distribution Decisions.

Factor	Description
Business importance	Returns on the investment of distributing a legacy system depend on its importance to the business and how long it will remain important. If distribution provides more efficient support for stable business processes then it is more likely to be a cost-effective evolution strategy.
System age	The older the system the more difficult it will be to modify its architecture because previous changes will have degraded the structure of the system.
System structure	The more modular the system, the easier it will be to change the architecture. If the application logic, the data management and the user interface of the system are closely intertwined, it will be difficult to separate functions for migration.
Hardware procurement policies	Application distribution may be necessary if there is company policy to replace expensive mainframe computers with cheaper servers. .

- ⇨ Before embarking on architectural migration, organisations should make a careful assessment of their legacy systems to ensure that they will gain real business value from the architectural transformation.
- ⇨ A fundamental difficulty in migrating many centralised legacy systems to a distributed architecture is that the systems are not structured in such a way that the basic architectural components can be identified and separated from other components.
- ⇨ Ideally, we would like legacy systems to have a structure as shown in the diagram on the left of Figure 1.9.

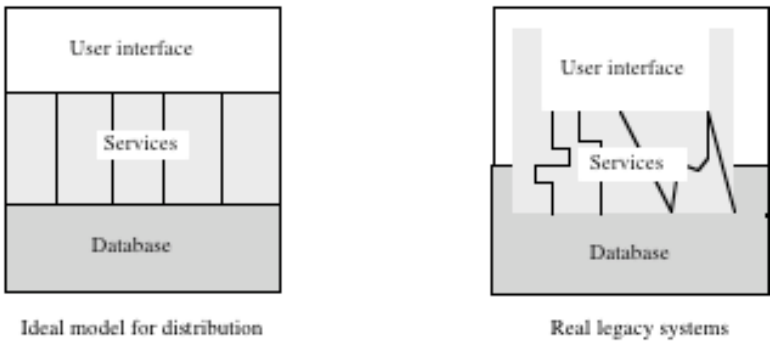


Figure 1.9: Ideal and Realistic Legacy System Structures.

- ⇨ In this case, the user interface, the services provided by the system and the database are clearly separated.
- ⇨ Individual services are well-defined.
- ⇨ Within the service layer, it is possible to distinguish between the different services.

- ⇨ With this type of structure, the distributable elements can be identified in the system and can be rewritten to run on client computers.
- ⇨ In practice, most legacy systems are more like the right side of Figure 1.9 where user interface facilities, services and data access are intermingled. Services may
- ⇨ overlap.
- ⇨ Different parts of the service are implemented in different system components.
- ⇨ User interface and service code are integrated in the same components and there may not be a clear distinction between the system services and the system database.
- ⇨ In these cases, it may not be possible to identify the parts of the system which can be distributed.
- ⇨ In situations where it is impractical to separate the legacy system into distributable components and implement these components on a distributed system an alternative approach can be used.
- ⇨ The legacy system may be frozen and the complete system packaged (wrapped) as a server.
- ⇨ The user interface is re-implemented on the client and special-purpose middleware translates requests from the client into interactions with the unchanged legacy system.
- ⇨ This situation is illustrated in Figure 1.10.

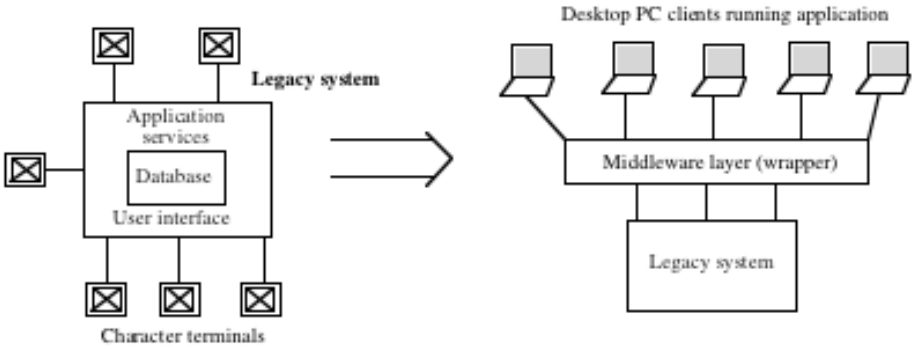


Figure 1.10: Legacy System Distribution.

- ⇨ Although the user interface and services provided by a legacy system are usually integrated, when planning a distribution strategy, it may be helpful to consider them as being organised into a number of logical layers (Figure 1.11).

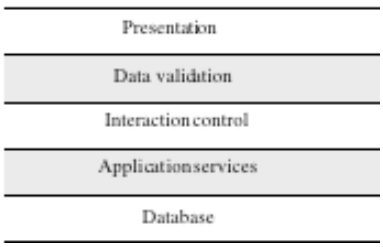


Figure 1.11: Layered Distribution Model.

- ⇨ The layers in this diagram represent potential candidates for distribution.

1. The presentation layer is concerned with the display and organisation of the screens presented to end-users of the system.
2. The data validation layer is concerned with checking the data input by and output to the end-user.
3. The interaction control layer is concerned with managing the sequence of end-user operations and the sequence of screens presented to the user.
4. The application services layer is concerned with providing the basic computations provided by the application.
5. The database layer provides application data storage and management.

⇨ It is impractical to distribute the database for most legacy systems but there is a spectrum of alternative distribution options as shown in Figure 1.12.

⇨ In the simplest option, the client computer is only concerned with the presentation of the user interface and all other functions are retained on the server.

⇨ In the most radical distribution option, the server only manages the system data and all other functionality is distributed to the client.

⇨ Of course, these are not exclusive options. You may decide to start with presentation distribution and distribute other logical layers when time and resources are available.

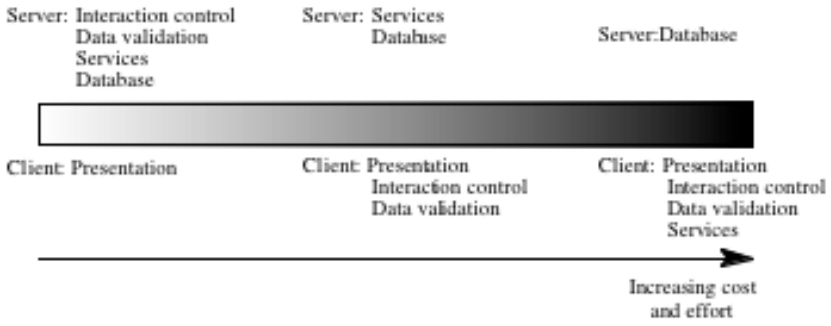


Figure 1.12: Spectrum of Distribution Options.

⇨ When a legacy system is wrapped and accessed through a middleware layer as in Figure 1.10, it is possible to implement a distribution strategy which starts on the left of Figure 1.12 and, over time, moves further and further to the right.

⇨ As new services are implemented, these take over the legacy system functions in the server thus transferring more and more processing to the client.

⇨ Eventually, this gradual distribution of functionality may mean that most of the initial legacy system is unused and it only acts as a database server for the distributed system.

⇨ Once this stage has been reached, you must then decide if it is worth retaining the legacy system or if you should replace it with a database management system.

- Factors that you should consider include the system hardware, the available expertise, whether or not a DBMS is already in use, whether it can cope with the amount of data to be managed and costs of data re-engineering.

1.4 Constructing for Verification

- Constructing for verification means building software in such a way that faults can be readily found by the software engineers writing the software as well as by the testers and users during independent testing and operational activities.
- Ensuring that a software system meets a user's needs.
- Specific techniques that support constructing for verification include: following coding standards to support code reviews and unit testing, organising code to support automated testing, and restricting the use of complex or hard-to-understand language structures, among others.

1.4.1 Verification vs Validation

- ***Verification***: "Are we building the product right"?
  - The software should conform to its specification.
- ***Validation***: "Are we building the right product"?
  - The software should do what the user really requires.
- Verification should check the program meets its specification as written in the requirements document for example.
  - This may involve checking that it meets it functional and non-functional requirements.
- Validation ensures that the product meets the customers' expectations.
  - This goes beyond checking it meets its specification; as we have seen, system specifications don't always accurately reflect the real needs of users.

1.4.2 The Verification and Validation Process

- As a whole life-cycle process - V & V must be applied at each stage in the software process.
  - Has two principal objectives
  - The discovery of defects in a system
  - The assessment of whether or not the system is usable in an operational situation.

1.4.3 Static and Dynamic Verification

- Software inspections Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
- Software testing Concerned with exercising and observing product behaviour (dynamic verification).

- The system is executed with test data and its operational behaviour is observed.

⇨ An Example of Bad Code:

```
public class Temperature {
   // constructor
   public Temperature(double initTemp) {
         x = initTemp;
   }
   // calcTGrd function to calc. the value of a T gradiant
   public double calcTGrd(float ZVAL){
         float a = x * x;
          a = a * ZVAL * 3.8883;
                        return a;
   }
   public double x;
}
```

⇨ System testing is only possible when an executable version of the program is available.

⇨ This is therefore an advantage of incremental development since a testable version of the system is available at a fairly early stage.

⇨ New functionality can be checked as it is added to the system and we can perform regression testing (we will talk about this in a few slides time).

⇨ Real data can be used as input to the system and we try to observe any anomalies in the output.

⇨ Figure 1.13 presents static and dynamic V&V.

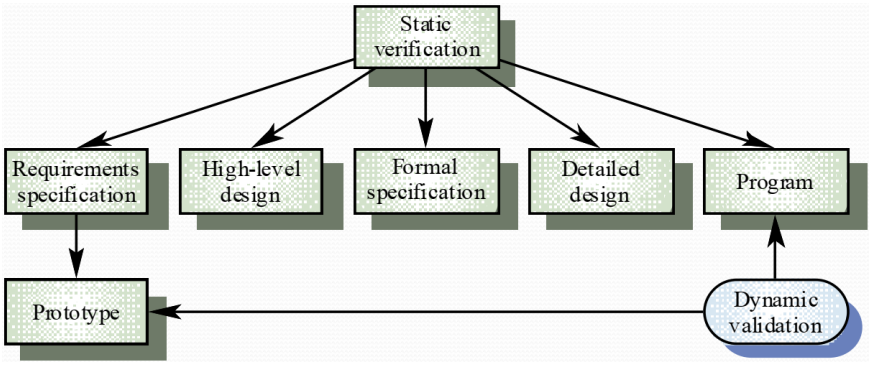


Figure 1.3: Static and Dynamic V&V.

1.4.4 Program Testing

⇨ Can reveal the presence of errors NOT their absence!!!

⇨ A successful test is a test which discovers one or more errors.

⇨ Program testing is the only validation technique for non-functional requirements.

- ⇨ Should be used in conjunction with static verification to provide full V&V coverage.
- ⇨ ***Types of Testing***:
  - ➢ Defect testing
    - Tests designed to discover system defects.
    - A successful defect test is one which reveals the presence of defects in a system.
  - ➢ Statistical testing
    - Tests designed to reflect the frequency of user inputs.
      - Used for reliability estimation.

1.4.5 Verification & Validation Goals

- ⇨ Verification and validation should establish a degree of confidence that the software is fit for purpose.
- ⇨ This does NOT mean completely free of defects.
- ⇨ The degree of confidence required depends upon several different factors.
- ⇨ ***Software function***: A much higher level of confidence that the system is fit for purpose is required for safety critical systems that for prototype systems for example
- ⇨ ***User expectations***: Users sometimes have a low expectation of software and are willing to tolerate some system failures (although this is decreasing).
- ⇨ ***Marketing environment***: Competing programs must be considered and the required schedule for introducing the product to market. Cheaper products may be expected to have more faults.

1.4.6 Testing and Debugging

- ⇨ Defect testing and debugging are distinct processes. Verification and validation is concerned with establishing the existence of defects in a program.
- ⇨ Debugging is concerned with
  - ➢ locating and
  - ➢ repairing these errors.
- ⇨ Debugging involves
  - ➢ formulating a hypothesis about program behaviour.
  - ➢ then testing these hypotheses to find the system error.
- ⇨ There is no simple process for debugging and it often involves looking for patterns in test outputs with defects and using a programmer's skill to locate the error
- ⇨ ***Question***: Recall the programs you have written in C/C++/Java/Python for example so far. Were there errors in early versions?
- ⇨ How did you discover them and fix them? Were they syntactic or semantic errors?

- Interactive debuggers provide a special run-time environment with access to the symbol table and program variables to aid error location. You can also "step-through" the program line by line.
- More Incorrect Code!

```
public class Temperature {
  // calcTGrd function to calc. the value of a T gradient
  public double calcTGrd(float ZVAL) {
    int a = (int) x * x
    if(a = 1)
        x = ZVAL * 3.8883;
    return a;
  }
  public double x;
}
```

Syntax error (missing semicolon)

Semantic error (should use double equals)

- **Syntax and Semantic Errors**
  - A ***syntax error*** should be caught by the compiler which will (usually) indicate the location the error occurred in and the type of error.
  - A ***semantic error*** (also called a logical error) can occur in a program which compiles and runs, but produces incorrect output on some (or all) input (e.g. An incorrect algorithm or mistake in a formulae etc.)
  - ***Semantic errors*** are often harder to detect since the compiler may not be able to indicate where/what the problem is.
- Once errors are located, it is necessary to correct the program code and re-test the program.
- Regression testing – after fixing a defect, it is advisable to retest the program with all previous test data to make sure the "fix" has not introduced new problems.
  - This is not always feasible due to costs
- Experience has shown that a large proportion of fault repairs introduce new errors or are incomplete.
- The debugging process is shown in Figure 1.14.

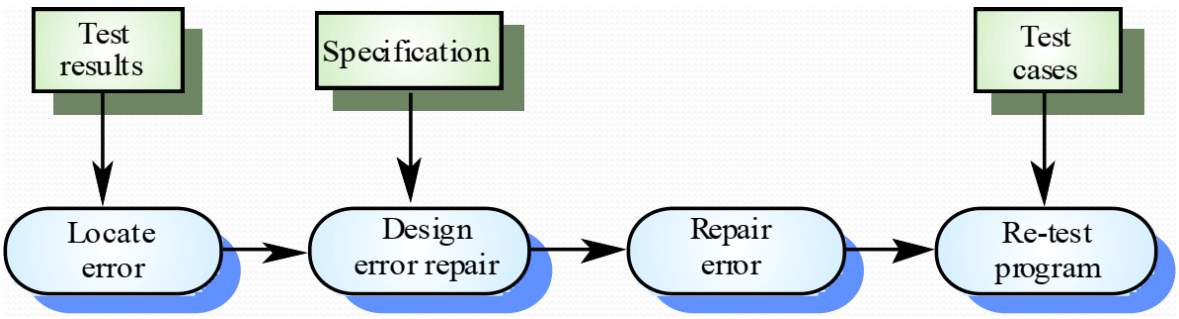


Figure 1.14: The Debugging Process.

1.4.7 V & V Planning

- ⇨ Careful planning is required to get the most out of testing and inspection processes.
- ⇨ Planning should start early in the development process.
- ⇨ The plan should identify the balance between static verification and testing.
- ⇨ Test planning is about defining standards for the testing process rather than describing product tests.
- ⇨ This is very visible in the V-Model development life cycle shown in Figure 1.15.

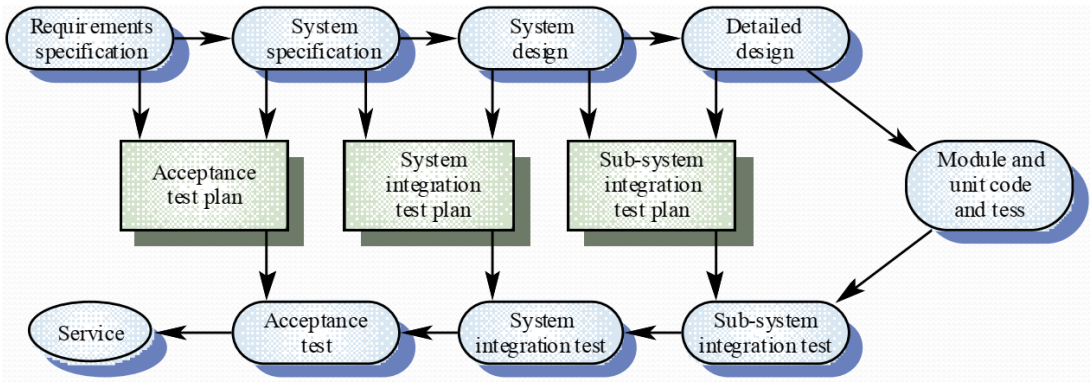


Figure 1.15: The V-Model of Development.

- ⇨ This diagram shows how test plans should be derived from the system specification and design.
- ⇨ The Structure of a Software Test Plan:
  - ➢ ***The testing process***: a description of the major phases of the testing process.
  - ➢ ***Requirements traceability***: testing should ensure that all requirements are individually tested.
  - ➢ ***Tested items***: Specify the products of the software process to be tested.
  - ➢ ***Testing schedule***: An overall schedule for the testing of the software is required and resources (time and personnel) must be allocated as part of the general project schedule.
  - ➢ ***Test recording procedures***: The results of tests must be systematically recorded, it is not enough to simply run the tests. This allows an audit of the testing process to check it has been carried out correctly (imagine a safety critical system; procedures for auditing the tests are often necessary).
  - ➢ ***Hardware and software requirements***: This part of the document sets out a list of software tools required and the estimated hardware utilisation.
  - ➢ ***Constraints***: Any constraints affecting the testing process should be anticipated in this section.

1.4.8 Software Inspections

- ⇨ Involve people examining the source representation with the aim of discovering anomalies and defects.
- ⇨ Does not require execution of a system so it may be used before the implementation phase.
- ⇨ May be applied to any representation of the system (requirements, design, test data, etc.).

- ⇨ Very effective technique for discovering errors.
- ⇨ Incomplete versions of the system can be inspected without additional costs – specialised test harnesses that work on only a part of the program are not required.
- ⇨ As well as program defects, inspections can consider broader quality attributes such as compliance with standards, portability and maintainability.
- ⇨ Poor programming style and inefficiencies can be found which could make the system difficult to maintain and update.

1.4.9 Inspection Success

- ⇨ Many different defects may be discovered in a single inspection, there is no "interaction" between errors to be concerned with.
- ⇨ In testing, one defect, may mask another, so several executions are required.
- ⇨ They reuse domain and programming knowledge so reviewers are likely to have seen the types of error that commonly arise.

1.4.10 Inspections and Testing

- ⇨ Inspections and testing are complementary and not opposing verification techniques.
- ⇨ Both should be used during the V & V process.
- ⇨ Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- ⇨ Also, inspections cannot check non-functional characteristics such as performance, usability, etc. (Emergent properties).

1.5 Reuse

- ⇨ Software reuse is the process of creating software systems from existing software rather than building software systems from scratch. That is, using existing assets in solving different problems.
- ⇨ This simple yet powerful vision was introduced in 1968.
- ⇨ In software construction, typical assets that are reused include libraries, modules, components, source code, and commercial off-the-shelf (COTS) assets.
- ⇨ Reuse is best practiced systematically, according to a well-defined, repeatable process.
- ⇨ Systematic reuse can enable significant software productivity, quality, and cost improvements.
- ⇨ Reuse has two closely related facets: "construction for reuse" and "construction with reuse."
- ⇨ The former means to create reusable software assets, while the latter means to reuse software assets in the construction of a new solution.

- ⇨ Reuse often transcends the boundary of projects, which means reused assets can be constructed in other projects or organisations.
- ⇨ Application system reuse
  - ➢ The whole of an application system may be reused either by incorporating it without change into other systems (COTS reuse) or by developing application families.
- ⇨ Component reuse
  - ➢ Components of an application from sub-systems to single objects may be reused.
- ⇨ Object and function reuse
  - ➢ Software components that implement a single well-defined object or function may be reused.
- ⇨ Types of reuse include:
  - ➢ Ad-hoc reuse;
  - ➢ Intra-project reuse;
  - ➢ Inter-project reuse;
  - ➢ Enterprise level reuse;
  - ➢ Horizontal reuse and;
  - ➢ Vertical reuse.
- ⇨ Table 1.2 presents to us some of the benefits of reuse.

Table 1.2: Benefits of Software Reuse.

Benefit	Explanation
Increased dependability	Reused software, which has been tried and tested in working systems, should be more dependable than new software. Its design and implementation faults should have been found and fixed.
Reduced process risk	The cost of existing software is already known, whereas the costs of development are always a matter of judgment. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as subsystems are reused.
Effective use of specialists	Instead of doing the same work over and over again, application specialists can develop reusable software that encapsulates their knowledge.

Benefit	Explanation
Standards compliance	Some standards, such as user interface standards, can be implemented as a set of reusable components. For example, if menus in a user interface are implemented using reusable components, all applications present the same menu formats to users. The use of standard user interfaces improves dependability because users make fewer mistakes when presented with a familiar interface.
Accelerated development	Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time may be reduced.

⇨ Table 1.3 presents to us some of the problems with reuse.

Table 1.3: Problems with Reuse.

Problem	Explanation
Increased maintenance costs	If the source code of a reused software system or component is not available then maintenance costs may be higher because the reused elements of the system may become increasingly incompatible with system changes.
Lack of tool support	Some software tools do not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account. This is particularly true for tools that support embedded systems engineering, less so for object-oriented development tools.
Not-invented-here syndrome	Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.
Creating, maintaining, and using a component library	Populating a reusable component library and ensuring the software developers can use this library can be expensive. Development processes have to be adapted to ensure that the library is used.
Finding, understanding, and adapting reusable components	Software components have to be discovered in a library, understood and, sometimes, adapted to work in a new environment. Engineers must be reasonably confident of finding a component in the library before they include a component search as part of their normal development process.

1.5.1 The Reuse Landscape

- Although reuse is often simply thought of as the reuse of system components, there are many different approaches to reuse that may be used.
- Reuse is possible at a range of levels from simple functions to complete application systems.
- The reuse landscape covers the range of possible reuse techniques presented in Figure 1.16.

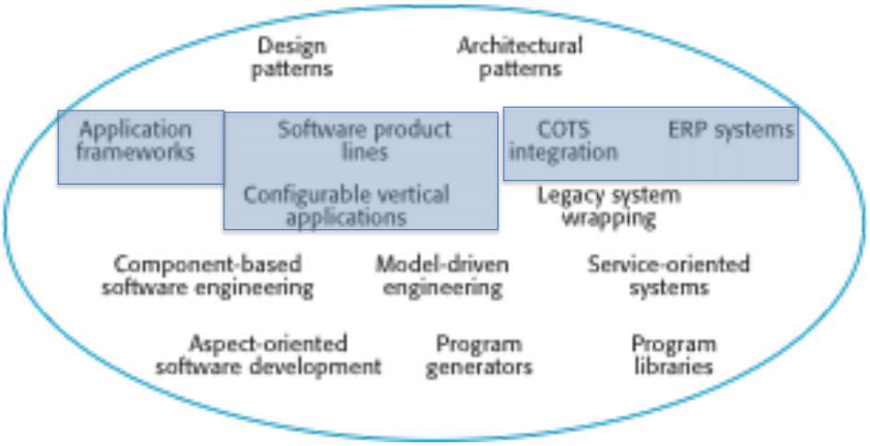


Figure 1.16: The Reuse Landscape.

1.5.2 Reuse Planning Factors

- The development schedule for the software.
- The expected software lifetime.
- The background, skills and experience of the development team.
- The criticality of the software and its non-functional requirements.
- The application domain.
- The execution platform for the software.

1.5.3 Application Frameworks

- Frameworks are moderately large entities that can be reused. They are somewhere between system and component reuse.
- Frameworks are a sub-system design made up of a collection of abstract and concrete classes and the interfaces between them.
- The sub-system is implemented by adding components to fill in parts of the design and by instantiating the abstract classes in the framework.

1.5.4 Framework Classes

- System infrastructure frameworks
  - Support the development of system infrastructures such as communications, user interfaces and compilers.

- ⇨ Middleware integration frameworks
  - ➢ Standards and classes that support component communication and information exchange.
- ⇨ Enterprise application frameworks
  - ➢ Support the development of specific types of application such as telecommunications or financial systems.

1.5.5 Web Application Frameworks (WAFs)

- ⇨ Support the construction of dynamic websites as a frontend for web applications.
- ⇨ WAFs are now available for all of the commonly used web programming languages e.g. Java, Python, Ruby, etc.
- ⇨ Interaction model is based on the Model-View-Controller composite pattern.

1.5.6 Reusability Standards

- ⇨ Name generalisation
- ⇨ Operation generalisation
- ⇨ Exception generalisation
- ⇨ Component generalisation.

1.6 Standards in Construction

- ⇨ Applying external or internal development standards during construction helps achieve a project's objectives for efficiency, quality, and cost.
- ⇨ Specifically, the choices of allowable programming language subsets and usage standards are important aids in achieving higher security.
- ⇨ Standards that directly affect construction issues include:
  - ➢ communication methods (for example, standards for document formats and contents).
  - ➢ programming languages (for example, language standards for languages like Java and C++).
  - ➢ coding standards (for example, standards for naming conventions, layout, and indentation).
  - ➢ platforms (for example, interface standards for operating system calls).
  - ➢ tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language)).
- ⇨ The purpose of software construction standards includes:
  - ➢ Define common practice.
  - ➢ Guide new engineers.
  - ➢ Make software engineering processes comparable.
  - ➢ Enable certification.

1.6.1 Use of External Standards

- ⇨ Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between the Software Construction KA and other KAs.
- ⇨ Standards come from numerous sources, including hardware and software interface specifications (such as the Object Management Group (OMG)) and international organisations (such as the IEEE or ISO).

1.6.2 Use of Internal Standards

- ⇨ Standards may also be created on an organisational basis at the corporate level or for use on specific projects.
- ⇨ These standards support coordination of group activities, minimising complexity, anticipating change, and constructing for verification.