Note on Software Engineering, Programming & Testing

Dr. Carsten Sørensen

Software engineers are professionals dealing in software, so obviously we need to understand the tools the software professionals themselves use or do not use. Do they take their own medicine and use all the technologies they so keenly promotes to others? How can we support the software engineering process? Has the changes to software engineering caused by the Internet changed the way the software engineering process is supported.

What is programming about anyway? In order for us to understand a bit about the nuts and bolts of building software, we will devote some time to discuss the programming process and programming environments. Is programming about building components, gluing components together or both? What is a compiler and how is it different from an interpreter? What happens to the code we write once the machine takes over. You won't be an expert after this, but you will have had a glimpse of what's under the bonnet of a modern programming environment.

This note explores software design methods as a whole. What issues can we relate to the view of the systems development process we already have from the IS471 Systems Development course? How can we understand the quality of the software process in terms of documentation, rationalisation, and process maturity? We will briefly discuss the main functional components of the software engineering process and in particular discuss design and testing. The requirements specification, analysis and organisational implementation was discussed in IS471. We will also discuss programming environments and software testing in subsequent sections.

Matthiasen (1998) defines system development as a change process taken with respect to object systems in a set of environments by a development group to achieve or maintain some objectives. Lytinen (1987) defines a system development method as an organised collection of concepts, beliefs, and normative principles supported by material resources. Software engineering has the objective of producing software products, i.e., software systems delivered to a customer with the documentation which describes how to install and use the system (Sommerville, 1995). Software engineering signifies a view to the technical. When we focus on software engineering issues as opposed to system development in general, we allow ourselves for a brief moment to forget the political aspects of developing systems, the end-users, the cultural dimension of organisational life etc. It does, however, not imply that we wholeheartedly accept the positivistic notion of software engineering being the proper and rational approach to building software, based on scientific principles. The software engineering process consists of the following four fundamental process activities (Sommerville, 1995):

- Software specification: The functionality of the software and constraints on its operation must be defined.
- Software development: The software to meet the specification must be produced.
- Software validation: The software must be validated to ensure it does what the customer wants.
- Software evolution: The software must evolve to meet changing customer needs

We will not make any hard distinctions between systems development and software engineering. Academics have for years argued about definitions and a lot of the debate simply relate to the fact that different communities have different perspectives. The four activities was all discussed in IS471 Systems Development. Systems development is often based on an Information Systems tradition with a relatively rich notion of organisational and human aspects of the process. Information Systems Development, as we saw in IS471, deals with the broad range of issues involved in establishing, managing and conducting the development of primarily information systems supporting business operation, as well as the complex set of issues concerning making the systems work in an organisational setting. Software engineering emerged out of the so called “software crisis” in the late 60ies where the rapid increase in hardware capability could not
be met by equal advances in software production. Moores Law states a doubling of computer processing capability every 18 months in terms of the number of transistors per surface unit. The advent of more structured and computer supported ways of developing both standard software components and bespoke systems has not as such solved the problems of developing software. As we might get a hint of by reading most mainstream software engineering textbooks, the software engineering tradition is slightly more functionalistic. It represents a relatively close focus on “building” software products based on an elicitation of requirements. In that sense, the software engineering perspective is relatively introvert in terms of focusing on the core activities for the software engineers, although some efforts to improve the level of maturity of the software process has been driven by the aim of establishing more clearly defined relationships between the customers and producers of software. Being born out of a technical and fairly positivist tradition, software engineering has come a long way and the debate to a large extent, especially in Europe, overlaps the debate within Information Systems Development.

The “software crisis” notion unifies a number of phenomena. Given that there has been a crisis declared for many years, it can not sensibly be a crisis, but rather a condition. The crisis can be now be viewed as caused by the gap between the dramatic increase in demand for software and the only moderate increase in programming productivity. Software engineering methods and practices has consistently been promoted as a means of addressing the software crisis. Tools, methods and techniques in abundance has been seen as the next “silver bullet” as argued by Brooks (1987). As Brooks argue, the nature of software production is such that no silver bullet can be found. There are basically two ways of improving productivity: steady stepwise improvement and standard software. Although recent years has seen an explosive growth in the use of standard application software such as MS Office, and even though software engineering principles increasingly are introduced in software projects, there is still a significant gap between our hopes and desires for productivity increases in software production and what is delivered. Given that Moores Law is estimated to hold for the years to come, the development within hardware will continue to outrun our capability to build software. Brooks would argue that he told us years ago ;-) The development of technologies and methods improving the software development process has been viewed as a means of meeting increasing demand for complex software systems from increasingly demanding customers. Although Curtis et al (1988) argued that domain knowledge was of paramount importance in software projects, the fact that many people work together over a long period of time making very complex decisions, calls for some sort of structured support and even support technologies such as Computer Aided Software Engineering (CASE) tools.

### The Software Crisis
- Low hardware costs
- More mature end-users
- High software costs
- Heavy maintenance load
- Moderate productivity increase
- More application areas
- Larger and more advanced systems

### Technological Innovation
- Commercially available technologies:
  - CASE Technology
  - CSCW Technology

#### Current software development practice

### Software Engineering Issues
There are a number of basic issues involved when characterising the software engineering process and the tools and techniques available to the modern software engineer. The following briefly lists the most important ones and relate them to Sommerville’s (1995) textbook for reference. The issues will, however, be dealt with in more or less a similar way in most software engineering textbooks. Requirements and specification including issues such as requirements engineering, requirements analysis, system models, prototyping and formal specification are presented in chapters 4 to 11. Software design are discussed in chapter 12 to 17. Dependable systems including a discussion of how to specify and ensure software reliability are discussed in chapter 18 to 21. Verification and validation of software is discussed in chapter 22 to 24. CASE is covered in chapter 25 to 27. Management issues including software cost estimation, quality management and process improvement are discussed in chapter 28 to 31. Software evolution including software maintenance,
configuration management and software re-engineering is covered in chapter 32 to 34. In general, software engineering textbooks will present in more technical detail issues touched upon in IS471.

**ASPECTS OF THE MODERN SOFTWARE ENGINEERING PROCESS**

The software engineering tradition generally adheres to the principle outlined by Parnas and Clements (1986). Although we cannot ensure a rational software engineering process, we can at least pretend and in doing so we can ensure a better documentation and conduct of the process. The arguments against a rational design process are similar to the ones against the feasibility of the waterfall model. Parnas & Clements argue that we must rationalise the process even though we know it cannot be done. In short, it has been argued consistently over many years that the maturity of the software process must be improved in most organisations, and this can be done by applying scientific and management techniques streamlining the process. A relatively large proportion of the software professionals, as for example expressed in the IEEE Software Journal, are engaged in improving the quality of the software engineering process and the software engineering profession. This is based on the assumption that improving the process will lead to improvement of the products. Curtis et al. (1988) argued that one of the main sources of complexity was lack of familiarity with the type of problem presented (Curtis et al., 1988). Humphrey (1989b; 1990) and the Software Engineering Institute (www.sei.cmu.edu) (Paulk et al., 1993) has established an elaborate framework: The Capability Maturity Model (CMM), characterising good software engineering practice. It initiated as a way of establishing sound contracts between customers and suppliers of software and has spawned in all directions. There are now specialised maturity models for various aspects of the software engineering process, such as personal Although being very much born out of scientific management ideas, CMM characterises the main problems in software development of managing risk. It is further argued that going from the initial stage over repeatable, define, managed and eventually ending at an optimised level, must be accompanied specialisation of domain. It is argued that a particular company cannot obtain a high degree of software process maturity if it does not specialise on building a particular type of software for particular domains. Jackson (1999) seems to agree in his promotion of specialisation as a means of establishing a more professional software engineering community.

<table>
<thead>
<tr>
<th>Level</th>
<th>Characteristics</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimized</td>
<td>• Improved feedback into process</td>
<td>Productivity &amp; quality</td>
</tr>
<tr>
<td>5</td>
<td>• Data gathering is automated and used to identify weakest process elements</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Numerical evidence used to justify application of technology to critical tasks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Rigorous defect-cause analysis and detect prevention</td>
<td></td>
</tr>
<tr>
<td>Managed</td>
<td>(Quantitative)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>• Measured process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Minimum set of quality and productivity measurements established</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Process database established with resources to analyze its data and maintain it</td>
<td></td>
</tr>
<tr>
<td>Defined</td>
<td>(Qualitative)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>• Process defined and institutionalized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Software Engineering Process Group established to lead process improvement</td>
<td></td>
</tr>
<tr>
<td>Repeatable</td>
<td>(Intuitive)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>• Process dependent on individuals</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Established basic project controls</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Strength in doing similar work, but faces major risk when presented with new challenges</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Lacks orderly framework for improvement</td>
<td></td>
</tr>
<tr>
<td>Initial</td>
<td>(Ad hoc/chaotic process)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>• No formal procedures, cost estimates, project plans</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• No management to ensure procedures are followed, tools are not well integrated and change control is lax</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Senior management does not understand key issues</td>
<td></td>
</tr>
</tbody>
</table>

Applying a macro view of the software development process, Mathiassen (1998) illustrates the changes to the systems development practice in the following three eras, covering: (I) Early sixties to the mid seventies, (II) the mid seventies to the late eighties, and late eighties until today.
<table>
<thead>
<tr>
<th>Era I</th>
<th>Era II</th>
<th>Era III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Purpose</td>
<td>Productivity and efficiency</td>
<td>Individual and group effectiveness</td>
</tr>
<tr>
<td>Applications</td>
<td>Automation</td>
<td>Support Integrated systems</td>
</tr>
<tr>
<td></td>
<td>Separate systems</td>
<td>Embedded systems</td>
</tr>
<tr>
<td>Technology</td>
<td>Mainframes Batch processing Databases</td>
<td>Distribution PC’s Local networks Graphics Expert systems</td>
</tr>
<tr>
<td>Skills</td>
<td>Programming Management</td>
<td>Analysis Design Collaborational</td>
</tr>
<tr>
<td>Improvement</td>
<td>Methods and tools</td>
<td>Quality assurance CASE</td>
</tr>
<tr>
<td></td>
<td>Project management</td>
<td></td>
</tr>
</tbody>
</table>

"AUTOMATIC PROGRAMMING" WITH CASE?

There are great challenges involved in conducting large complex software engineering projects involving a large number of people, as Brooks (1982) so eloquently illustrates in his classic essays from the early 70s. Although several studies, for example Curtis et al’s (1988), have shown, domain knowledge and experience is a crucial factor in successful software engineering projects. In Mathiassen’s three eras of systems as a critical success factor (Mathiassen, 1998). Nonetheless, humans always seems to seek the “silver bullet” (Brooks Jr., 1987), and somehow always end up biting it ;-)  

The silver bullet of software engineering was, at least at some point in time, Computer Aided Software Engineering (CASE) tools. The dream of solving the software crisis by means of "automatic programming" was at it’s peak in the mid 1980ies. The CASE (Computer Automated Software Engineering) concept was born in 1984. Subsequently the "Automated" was replaced with the more modest "Aided". It was realised that if the code was generated then you would in fact have had done coding during the design phase. It is not so important what you call the phases. It is more important what actually happens. CASE tools can be viewed as yet a step in the direction of moving effort from the later phases of the software engineering process up to the early phases. It places heavy emphasis on supporting analysis and design activities. You have in IS471 been subjected to one example of a CASE tool, i.e., Select Enterprise. It is fairly typical in the sense that it is repository based CASE tools generally support the following activities (Sørensen, 1995):

- Model sharing and distribution: Through a capability known as subsetting, portions of models can be derived from large ones. Members of projects can thus work concurrently without the danger of overlap
- Actual and trial model merging: Project team members can combine two versions of a model based on like names and structures
- Administrative, statistical, and impact analysis reports: These reports are useful in tracking model status and impacts throughout the project life cycle
- Version control: By replicating models and transferring changes, developers can maintain multiple copies of the same model at different stages of development, testing, and production. Version control also provides a mechanism for managing shared information at an object level across multiple projects in different models and at different stages
- Multiple levels of security access: These measures help ensure the integrity of business models at all times

The idea of a CASE tool is basically to record design decisions during the system development process in a repository, as the figure from Fournier (1991) illustrates:
The term Upper-CASE (pardon the pun, but it wasn't my idea!) denotes CASE tools primarily support the high-level, early phases in the waterfall model, such as strategic modelling of the need for systems, requirements analysis, systems analysis, and systems design. Lower-CASE implies a tool primarily supporting the later stages in the waterfall model, i.e., from design over technical or architecture design, implementation, testing and perhaps the re-engineering of existing source code into high-level models (an ambition not properly fulfilled since the high level re-engineered version of spaghetti-code is a spaghetti pattern of boxes with lines between them.)

In order to sort out some of the conceptual confusion concerning what CASE tools actually were all about, Henderson & Cooprider presents the following functional model of CASE tools (Henderson and Cooprider, 1990):

**CASE Tool Support and Software Process Maturity**

How does computer supporting the software engineering process fit with the attempts to increase the maturity of the software process? The tools support the process of rationalising the software engineering process by focusing on supporting the process of producing internally consistent specifications. This is in line with Parnas & Clement's (1986) cry for second-order rationality in the design process, given that first order rationality is not obtainable. If we consider the Capability Maturity Model from the Software Engineering Institute (Humphrey, 1989b; Humphrey, 1990; Paulk et al., 1993; Chapter 31, Sommerville, 1995), then how can match the use of CASE tools with attempts to establish a mature software process? Some of the CMM proponents have argued that there is no sense in introducing CASE tools before the software process is
defined (Humphrey, 1989a; Curtis, 1992). Otherwise it is not possible to know what exactly the tool should support. This rational argument is deconstructed by Mathiassen and Sørensen (1996), arguing that if over 80% of software engineering companies are at level 1, and they all buy CASE tools, either this assumption is wrong or the companies are doing something silly. It is, argued, using March’s idea of the technology of foolishness (March, 1976), that organisations need to experiment, and given that CASE will only succeed in organisational settings if it is accompanied changes to the way software is developed, it may be possible to leverage the quality of the software process by interjecting technology into the equation. The statistical model formulated in (Sørensen, 1993), highlights the importance of changing the way software is developed when introducing CASE toolsto support the production and documentation of software systems.

For more reading on CASE and the implications of attempting to make software engineers use them, see for example: (Aaen and Sørensen, 1991; Sørensen, 1993; Ivari, 1996; Mathiassen and Sørensen, 1996; Mathiassen and Sørensen, 1997; Mathiassen, 1998).

**The Programming Process**

Let us her at the end turn towards the programming process itself, as opposed to looking at the entire software engineering process or the analysis and design activities. It can be quite a challenge to attempt characterising programming. On the surface it may seems quite straight forward. You write a series of statement in a programming language. These statements is subsequently transformed into executable commands. It can not be all that difficult to understand? Well, actually it is a bit tricky. If we ask what programming "really" is we may get different answers depending on who we ask. Obviously, the different answers can be attributed to different perspectives, and with that in mind, let us look at programming from different perspectives. We here primarily look at programming as an individual activity and at the main products it produces.

The world’s first programmer was a woman, or let’s at least pretend so. Ada Lovelace documented the work of Charles Babbage when he designed and constructed the Analytical Engine. He also designed the mechanical general purpose machine the Difference Engine, but due to mechanical precision standards at that time, did not work. For a very interesting speculation of what the world would look like if Babbage had succeeded, read and William Gibson and Bruce Stirling’s book "The Difference Engine" (Gibson and Sterling, 1991). Weizenbaum explains the nature of the compulsive programmer and argues that this person should not be confused with hard-working professional programmers. There absolute power over an abstract world you are given as a programmer corrupts some people and they embark upon building gigantic castles in the sky. This aspect is even covered in Gibson and Stirlings book.

When characterising the products of the programming process, we must distinguish between programs, program products, program systems and program product systems Brooks Jr (1982)
No Silver Bullet. Brooks (1987) argued that there, unlike in the case of the silver bullet destroying werewolves, is no silver bullet, which in one strike will solve the problems of developing more and more complex software. Only hard work and steady improvement will increase productivity. Brooks recognises the importance of reusing software, i.e., code once run everywhere as in Windows, Office, Notes, SAP R/3 etc etc. Programming is no longer primarily a craft as such. More and more software is "manufactured" It is developed by businesses who subsequently mass produce it and sells it to whoever want it. Even very complex interorganisational systems are standard products (SAP R/3, Lotus Notes + Domino etc).

Logics vs. interaction: But what is at the core of all this. How can we characterise "the thing in itself"? Dijkstra (1989) and may with him claims that programming is all about logic. Algorithms transforming data by means of sequence, selection and iteration. To ensure correct and elegant programs, all programmers must study math! Wegner (1997) argues against this. The world of programming is not any longer one transaction, but one of interaction. The basic model for understanding computers has been an algorithmic model with an emphasis on algorithmic transformations. The Turing Machine with its infinite tape, the control box and the read-write head that is inaccessible whilst processing it dead. Interactive systems with memory to the fore!

Formal vs. informal: Programming as theory building. Naur (1992) has a slightly different slant on it all. Naur looked at the relationships between the programs and the process of bringing them about. He said to the people who believe in correctness of algorithms as the main quality parameter were naive. Usability, relevance etc were just as important properties. Furthermore, Naur argued, we should not look at the product of programming, but also on the process that brings it about. Naur argued that we can view the programmers’ process as a process of creating a theory. The program is the theory and for others subsequently to understand and appreciate the qualities of this theory, they need to study it carefully and perhaps even re-experience what the programmer experienced when formulating the theory. Re-use of code therefore is problematic. which you experience when using JBuilder. Programming, although dealing with formal constructs is inherently an informal process.

**Programming Environments**

Let me introduce you to programming and programming environments. The idea is to let you know your way about without knowing it all. You will hopefully appreciate the complexity of programming environments, but also the relative simple principles involved in conceptualising the process from code to application. What is programming anyway?:

Producing an organised list of instructions that, when executed, causes the computer to behave in a predetermined manner. Without programs, computers are useless. A program is like a recipe. It contains a list of ingredients (called variables) and a list of directions (called statements) that tell the computer what to do with the variables. The variables can represent numeric data, text, or graphical images.

The typical elements of the programming environment are: Programming language (syntax, semantics); Editor; Pretty printer; Compiler; Linker; Debugger; Classes/libraries for Graphical User Interface (GUI) and data-structures; browser for libraries. These are frequently “packaged” as an Integrated Development Environment (IDE). Borland JBuilder 2 is an example of such an IDE.

The programming process was traditionally about moving switches and wires and even structured programming relatively new invention. Previously Goto was the driving programming principle. Goto-less programming was promoted by Boehm & Jacopini, (1966) who argued that any program using Goto could be expressed using the three constructs sequence, selection and repetition and in 1968 Dijkstra (1968) wrote the famous article “Goto considered harmful”. Structured programming was generally taken serious from the 1970ies. In the late 60ies Grace Murray Hopper and others developed COBOL as an easy way to express business rules. This denotes a milestone in going from machine language to “business language” or “domain language”. Now you will use an integrated development environment, where expressing the code in Java syntax is only one element. So why is it difficult to write computer programs? There are a lot of reasons, some of which relates to the requirements process and human capabilities as argued by (Parnas and Clements, 1986). In many ways programming offer us the worst of all worlds. In electronic engineering there are in principle infinite many states to control, but fancy mathematics can help us estimate the states as continuous functions. In logistics we have discrete states so differential equations can not help us easily describe behaviour. On the other hand, there are, relative to the previous example few discrete states and we can therefore just keep track of all the different states or configurations. When programming a computer we can assume infinite many states, and they are all discrete, so continuous mathematics can not help us and there are most often far too many states for us to enumerate ;-( Brooks (1987) argued that programs
inherently are complex to produce, even if we manage to do it as well as possible. We have already last week discussed how to understand the programming process, here, there is just one minor point to be clarified, related to what a program is, or what it should be perceived as. Traditionally, a program has been considered as input-process-output (Dijkstra, 1989). Consequently, Dijkstra argues that way to teach programming is to teach mathematics or logics. Naur (1992), as we discussed last week, argued against this, claiming the important property of the programming process was the programmer building a theory in his or her head about the program. Lately, the way in which we program and the type of programs produced has led to a change in the way we perceive programs. Wegner (1997) argues that we cannot look at the execution of programs as an unbroken input-process-output chain. We now must expand this view, originally formulated by the British mathematician Alan Turing, and view the execution of programs as interaction between the program and its environment. If we follow this, there is put a substantial bomb under the formalised paradigm of “proving” program correctness since the interaction paradigm highlights the importance of users.

**Programming Languages**

A vocabulary and set of grammatical rules for instructing a computer to perform specific tasks. The term programming language usually refers to high-level languages, such as Java, BASIC, C, C++, COBOL, FORTRAN, Ada, and Pascal. Each language has a unique set of keywords (words that it understands) and a special syntax for organising program instructions. Java keywords:

<table>
<thead>
<tr>
<th>abstract</th>
<th>boolean</th>
<th>break</th>
<th>byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>case</td>
<td>char</td>
<td>default</td>
<td>class</td>
</tr>
<tr>
<td>do</td>
<td>double</td>
<td>else</td>
<td>extends</td>
</tr>
<tr>
<td>final</td>
<td>finally</td>
<td>float</td>
<td></td>
</tr>
<tr>
<td>for</td>
<td>implements</td>
<td>import</td>
<td>instanceof</td>
</tr>
<tr>
<td>int</td>
<td>interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td>long</td>
<td>native</td>
<td>new</td>
<td>null</td>
</tr>
<tr>
<td>package</td>
<td>protected</td>
<td>public</td>
<td>return</td>
</tr>
<tr>
<td>short</td>
<td>super</td>
<td>switch</td>
<td>synchronized</td>
</tr>
<tr>
<td>this</td>
<td>throws</td>
<td>transient</td>
<td>true</td>
</tr>
<tr>
<td>try</td>
<td>volatile</td>
<td>while</td>
<td></td>
</tr>
</tbody>
</table>

High-level programming languages, while simple compared to human languages, are more complex than the languages the computer actually understands, called machine languages. Each different type of CPU has its own unique machine language. Between machine languages and high-level languages are languages called assembly languages. Assembly languages are similar to machine languages, but they are much easier to program in because they allow a programmer to substitute names for numbers. Machine languages consist of numbers only.

<table>
<thead>
<tr>
<th>Source code</th>
<th>Assembly Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF COUNT = 10</td>
<td>Compare A to B</td>
</tr>
<tr>
<td>GOTO DONE</td>
<td>If equal go to C</td>
</tr>
<tr>
<td>ELSE</td>
<td>Go to D</td>
</tr>
<tr>
<td>GOTO AGAIN</td>
<td></td>
</tr>
<tr>
<td>ENDIF</td>
<td></td>
</tr>
<tr>
<td>Machine Language</td>
<td>Actual Machine Code</td>
</tr>
<tr>
<td>Compare 3477 2883</td>
<td>100101010010100001010000</td>
</tr>
<tr>
<td>If = go to 23883</td>
<td>101010100100100000001010</td>
</tr>
<tr>
<td>Go to 23943</td>
<td>101001010101010101010101</td>
</tr>
</tbody>
</table>

Fourth-generation languages (usually abbreviated 4GL) are more abstract than high-level languages are languages called fourth-generation languages (usually abbreviated 4GL). The question of which language is best is one that consumes a lot of time and energy among computer professionals. Every language has its strengths and weaknesses. For example, FORTRAN is a particularly good language for processing numerical data, but it does not lend itself very well to organising large programs. Pascal is very good for writing well-structured and readable programs, but it is not as flexible as the C programming language. C++ embodies powerful object-oriented features, but it is complex and difficult to learn. Java combines some of the advantages of other languages. It is object-oriented without some of the complicating factors of C++. It comes with the ability easily to build internet aware network applications. Java attempts through compiling byte-code for a virtual machine to implement a “compile once, run everywhere” strategy. Armstrong (1998) discusses in detail the advantages of Java.
PROGRAMMING LANGUAGE CONCEPTS

Compiletime: When the source file is compiled
Runtime: When the program is executed.
Syntax: The structure of strings in some language. A language’s syntax is described by a grammar (see BNF Note 4).
Static Semantics: The meaning of the program as it can be determined before it is compiled an receives input.
Runtime Semantics: What actually happens when the program is executed.
Statement: The basic element of the source code.
Basic Operations: Sequence, selection and iteration
Compiled versus Interpreted: Does the source code need to be transformed to object code or bytecode by a compiler before execution or is it executed directly by an interpreter.
Weak or Strong Typing: How strict rules does the programming language impose for how types are declared and used. Java has strong typing

TRADITIONAL PROCEDURAL LANGUAGES

The procedural paradigm is regarding to a set of computational languages which the primary concern is how to execute the commands in order to find the solutions. Languages in this paradigm have their commands executed in the same order as they are defined in the source program. Examples of such languages are: Fortran, Cobol and Pascal. Procedural programs are described in terms of procedures, functions, types, variables, and constants and modules often designed based on descriptions of the process. Modules can be analysed using the concepts of coupling and cohesion. Coupling denotes the internal “strength” of the code, i.e., how much it need other modules. Cohesion is the external “strength” between modules. On that basis, the design principle of maximum cohesion within modules and minimum coupling between modules is applied. Parnas (1972) shifted the attention from using a flowchart description of the process as a basis for deciding how to divide code into modules, towards seeking to hide information from the rest of the modules. This in fact led to discussions of information hiding and subsequently encapsulation, i.e., We need to know how to interact with an object, and not concern with how it implements it’s behaviour. Parnas argued that the interface to and not the contents of modules should be the most stable, and the decomposition of modules should be based on hiding what most likely would change in a module from other modules.

The following Turbo Pascal example converts Fahrenheit to Celsius:

```pascal
PROGRAM convert;
VAR
  fahr, cent : INTEGER;
BEGIN
  write('Enter Fahrenheit ');
  readln(fahr);
  cent := (fahr - 32) * 5 / 9;
  writeln('Celsius is ',cent)
END.
```
OBJECT-ORIENTED PROGRAMMING

An evolutionary form of modular programming with more formal rules that allow pieces of software to be reused and interchanged between programs. In fact, object-oriented programming languages support us in doing what Parnas suggested in 1972. SIMULA, invented by the Norwegians Kristen Nygaard and Olle-Johan Dahl, was the original object-oriented language. It was used to model the behavior of complex systems. Xerox’s Smalltalk was the first object-oriented programming language and was used to create the graphical user interface whose derivations are so popular today. C++, defined at AT&T Laboratories by the Dane Bjarne Stroustrup, has become the major commercial OOP language, because it combines traditional C programming with object-oriented capabilities. Encapsulation is the creation of self-sufficient modules that contain the data and the processing (data structure and functions that manipulate that data). These user-defined, or abstract, data types are called classes. One instance of a class is called an object. For example, in a payroll system, a class could be defined as Manager, and Pat and Jan, the actual objects, are instances of that class. Classes are created in hierarchies, and inheritance allows the knowledge in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems. If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited. Object-oriented programming allows, through polymorphism, procedures about objects to be created whose exact type is not known until runtime.

For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement would be written for "cursor," and polymorphism would allow that cursor to be whatever shape is required at runtime. It would also allow a new shape to be easily integrated into the program.

<table>
<thead>
<tr>
<th>PROCEDURAL LANGUAGE</th>
<th>OBJECT-ORIENTED LANGUAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data type + characteristics</td>
<td>1. Class</td>
</tr>
<tr>
<td>2. Variable</td>
<td>2. Instance</td>
</tr>
<tr>
<td>3. Declare a variable</td>
<td>3. Instantiate</td>
</tr>
<tr>
<td>4. Processing code</td>
<td>4. Method</td>
</tr>
<tr>
<td>5. Call</td>
<td>5. Message</td>
</tr>
<tr>
<td>6. Data type + processing</td>
<td>6. Object</td>
</tr>
</tbody>
</table>

FUNCTIONAL

Functional languages are (for the most part) declarative, that is, they specify what ought to be done, rather than in what order things ought to happen. LISP (LIST Processing) is the most famous such language. It is a high-level programming language used for developing AI applications. Developed in 1960 by John McCarthy, its syntax and structure is very different than traditional programming languages. For example, there is no syntactic difference between data and instructions. LISP is available in both interpreter and compiler versions and can be modified and expanded by the programmer. Many varieties have been developed, including versions that perform calculations efficiently. Recursion is the important driving construct. Pure LISP: no state, or sideeffects. Very cumbersome to program most applications.

```
'(this list has (a list inside of it)
(car '(rose violet daisy buttercup))
= rose
(cdr '(rose violet daisy buttercup))
= (violet daisy buttercup)
```

DECLARATIVE/LOGICAL

The paradigm of declarative languages is regarding to a set of computational languages which the primary concern is what to execute in order to find the solutions. Languages in this paradigm have their commands executed in the order necessary to achieve the solution, not the order they order written in the source program. Prolog (PROgramming in LOGic) is the major example of language in this paradigm. A programming language used for developing AI applications (natural language translation, expert systems, abstract problem solving, etc.). Developed in France in 1973, it is used throughout Europe and Japan and is gaining popularity in the U.S. Similar to LISP, it deals with symbolic representations of objects.

John gives a class

This sentence states a relation give between two objects: John and class. The relation is a more abstract concept than the objects. This means, we can have the same relation relating other objects. For example:
John gives a class.
Paul gives a class.
"Is it true that Paul gives a class?" = Yes
"Is it true that Paul gives a seminar?" = No
If SOMEONE gives a class Then SOMEONE is a lecturer.
Paul is a lecturer
John is a lecturer
"Who is a lecturer?"
Paul, John

In Prolog a set of facts are declared and some generic rules which relates a set of objects, and then combine the set of rules with the set of the facts in order to answer queries. It is based on closed world assumptions and infers based on the rules.

**Scripting**

A high-level programming, or command, language that is interpreted (translated on the fly) rather than compiled ahead of time. A script, or scripting, language is not a general-purpose programming language. Although it can be very extensive, it is usually limited to specific functions used to augment the running of an application or system program. Spreadsheet macros and communications scripts are examples. Microsoft's Visual Basic for Applications (VBA) is a script language version of Visual Basic and is used to automate Microsoft applications. Examples are Perl, Python, JavaScript (Java and JavaScript are unrelated. JavaScript was named so to "ride the wave of popular interest). Perl (Practical Extraction Report Language) is a programming language written by Larry Wall that combines syntax from several UNIX utilities and languages. Especially designed for processing text. Perl is an interpretive language, which makes it easy to build and test simple programs. Perl is designed to handle a variety of system administrator functions. Because of its comprehensive string handling capabilities, it is widely used on Web servers. Osterhout (1998) argues that scripting indeed because of the compressed nature of the code, is indeed the programming language of the future.

---

The Bluffers Guide to Programming Languages

**Ada**: Comprehensive, Pascal-based language used by the Department of Defense.

**ALGOL**: International language for expressing algorithms.

**APL**: Used for statistics and mathematical matrices. Requires special keyboard symbols.

**BASIC**: Developed as a timesharing language in the 1960s. It has been widely used in microcomputer programming in the past, and various dialects of BASIC have been incorporated into many different applications.

**C**: Developed in the 1980s at AT&T. Widely used to develop commercial applications. UNIX is written in C.

**C++**: Object-oriented version of C developed by Bjarne Stroustrup that is popular because it combines object-oriented capability with traditional C programming syntax.

**COBOL**: (COmmon Business Oriented Language) A high-level programming language that has been the primary business application language on mainframes and minis. It is a compiled language and was one of the first high-level languages developed. Formally adopted in 1960, it stemmed from a language called Flowmatic in the mid 1950s. COBOL is a very wordy language.
dBASE: Widely used in business applications. Offshoots of dBASE ("Xbase" languages) are Clipper, Quicksilver, FoxBase and FoxPro.
Delphi: Borland's extended version of Pascal made object oriented and with an extensive IDE.
FORTH: Developed in the 1960s, FORTH is used in process control and game applications.
FORTRAN: Developed in 1954 by IBM, it was the first major scientific programming language. Some commercial applications have been developed in FORTRAN, and it continues to be widely used.
Java: The programming language for the Web. Sun's Java is used to create client/server applications on intranets and the Web.
LISP: Developed in 1960. Used for AI applications. Its syntax is very different than other languages.
Logo: Developed in the 1960s, it is noted for its ease of use and "turtle graphics" drawing functions.
Pascal: Originally an academic language developed in the 1970s. Borland commercialized it with its Turbo Pascal.
Perl: Practical Extraction Report Language) A programming language written by Larry Wall that combines syntax from several UNIX utilities and languages. Perl is designed to handle a variety of system administrator functions. Because of its comprehensive string handling capabilities, it is widely used on Web servers.
PL/I: (Programming Language 1) A high-level IBM programming language introduced in 1964 with the System/360 series. It was designed to combine features of and eventually supplant COBOL and FORTRAN, which never happened. A PL/I program is made up of procedures (modules) that can be compiled independently. There is always a main procedure and zero or more additional ones. Functions, which pass arguments back and forth, are also provided
Python: An interpreted, object-oriented programming language developed by Guido van Rossum. The name comes from one of van Rossum's favorite television shows, Monty Python's Flying Circus. Python is very portable since Python interpreters are available for most operating system platforms.
Simula: A simulation language originating in the late 1960s that was used to model the behavior of complex systems. SIMULA was the original object-oriented language.
Smalltalk: An operating system and object-oriented programming language that was developed at Xerox PARC. As an integrated environment, it eliminates the distinction between programming language and operating system. It also allows its user interface and behavior to be customised. Smalltalk was the first object-oriented programming language to become popular. It was originally used to create prototypes of simpler programming languages and the graphical interfaces that are so popular today.
Tcl: Short for Tool Command Language, and pronounced T-C-L or tickle, a powerful interpreted programming language developed by John Ousterhout. One of the main strengths of Tcl is that it can be easily extended through the addition of custom Tcl libraries. It is used for prototyping applications as well as for developing CGI scripts, though it is not as popular as Perl for the latter.
Visual Basic: Version of BASIC for Windows programming from Microsoft that is very popular.
Web Languages: Languages such as JavaScript, Jscript, Perl and CGI are used to automate Web pages as well as link them to other applications running in servers.

**Programming Language Concepts**

**Compli**-etime: When the source file is compiled

**Runtime:** When the program is executed.

**Syntax:** The structure of strings in some language. A language’s syntax is described by a grammar (see BNF Note 4).

**Static Semantics:** The meaning of the program as it can be determined before it is compiled an receives input.

**Runtime Semantics:** What actually happens when the program is executed.

**Statement:** The basic element of the source code.

**Basic Operations:** Sequence, selection and iteration

**Compiled versus Interpreted:** Does the source code need to be transformed to object code or bytecode by a compiler before execution or is it executed directly by an interpreter.

**Weak or Strong Typing:** How strict rules does the programming language impose for how types are declared and used. Java has strong typing

**Compiling and Interpreting**

There are basically two ways of translating your program into machine language so that the computer can understand it. The most common is to compile the program; the other method is to pass the program through an interpreter. But before we go any further, we need to establish a couple of concepts describing the process of translating programs to zeros and ones our computer will understand.
**Compiletime:** When the source file is compiled

**Runtime:** When the program is executed.

**Syntax:** The structure of strings in some language. A language's syntax is described by a grammar (see BNF Note 4).

**Static Semantics:** The meaning of the program as it can be determined before it is compiled and receives input.

**Runtime Semantics:** What actually happens when the program is executed.

**Statement:** The basic element of the source code.

**Basic Operations:** Sequence, selection and iteration

**Compiled versus Interpreted:** Does the source code need to be transformed to object code or bytecode by a compiler before execution or is it executed directly by an interpreter.

**Weak or Strong Typing:** How strict rules does the programming language impose for how types are declared and used. Java has strong, Perl has weak typing.

A compiler is a program that translates source code into object code. The compiler derives its name from the way it works, looking at the entire piece of source code and collecting and reorganising the instructions. Compilers require some time before an executable program emerges. However, programs produced by compilers run much faster than the same programs executed by an interpreter.

Because compilers translate source code into object code, which is unique for each type of computer, many compilers are available for the same language. For example, there is a Java compiler for PCs and another for Apple Macintosh computers. In addition, the compiler industry is quite competitive, so there are actually many compilers for each language on each type of computer. More than a dozen companies develop and sell C compilers for the PC.

**The Interpreter**

An interpreter translates high-level instructions into an intermediate form, which it then executes. In contrast, a compiler translates high-level instructions directly into machine language. Compiled programs generally run faster than interpreted programs. The advantage of an interpreter, however, is that it does not need to go through the compilation stage during which machine instructions are generated. This process can be time-consuming if the program is long. The interpreter, on the other hand, can immediately execute high-level programs. For this reason, interpreters are sometimes used during the development of a program, when a programmer wants to add small sections at a time and test them quickly. Both interpreters and compilers are available for most high-level languages. However, languages such as Perl, Tcl, JavaScript, Visual Basic, BASIC and LISP are especially designed to be executed by an interpreter.
STAGES IN THE COMPILATION PROCESS

The compilation process consists of the following elements:

- **Lexical Analysis:** The first stage of processing a language. The stream of characters making up the source program or other input is read one at a time and grouped into lexemes (or "tokens")

- **Parser:** An algorithm or program to determine the syntactic structure of a sentence or string of symbols in some language. A parser normally takes as input a sequence of tokens output by a lexical analyser

- **Abstract Syntax Tree:** (AST) A data structure representing something that has been parsed often used as a compiler or interpreter's internal representation of a program while it is being optimised and from which code generation is performed

- **Code generation**

- **Linking**

- **Running**

SPECIFYING GRAMMERS

BNF: Bacus-Naur Form

A formal metasyntax used to express context-free grammars. BNF is one of the most commonly used notations for specifying the syntax of programming languages, command sets, and the like.
Consider this BNF for a US postal address:

```
<postal-address> ::= <name-part> <street-address> <zip-part>
<personal-part> p ::= <name> | <initial> "." |
:name-part> ::= <personal-part> <last-name> [{jr-part}] <EOL> |
<street-address> ::= [<apt>] <house-num> <street-name> <EOL>
<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>
```

This translates into English as:

"A postal-address consists of a name-part, followed by a street-address part, followed by a zip-code part. A personal-part consists of either of a first name or an initial followed by a dot. A name-part consists of either a personal-part followed by a last-name followed by an optional "jr-part" (Jr., Sr., or dynastic number) and end-of-line (EOL), or a personal-part followed by a name-part (this rule illustrates the use of recursion in BNFs, covering the case of people who use multiple first and middle names and/or initials). A street-address consists of an optional apartment specifier (apt), followed by a street number (house-num), followed by a street-name. A zip-part consists of a town-name, followed by a comma (""), followed by a state-code, followed by a ZIP-code followed by an end-of-line (EOL)."

Note that many things (such as the format of a personal-part, apartment specifier, or ZIP-code) are left unspecified. These lexical details are presumed to be obvious from context or specified somewhere nearby.

**BUGS!**

"That was back on Mark I. It was in 1945. We were building Mark II—and Mark II stopped. We finally located the failing relay and, inside the relay, beaten to death by the relay contact, was a moth about three inches long. So the operator got a pair of tweezers and carefully fished the bug out of the relay and put it in the log book. He put Scotch tape over it and wrote, "First actual bug found." And the bug is still in the log book under the Scotch tape and it is in the museum of the Naval Surface Weapons Center at Dahlgren, Virginia." Grace Murray Hopper

"The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer's own creation." E. Dijkstra

"The lay public, familiar with only a few incidents of software failure, may regard them as exceptions caused by inept programmers. Those of us who are software professionals know better: the most competent programmers in the world cannot avoid such problems. (...) Software is released for use, not when it is known to be correct, but when the rate of discovering new errors slow down to one that management considers acceptable. (...) It is not unusual for software modifications to be made in the field. Programmers are transported by helicopter to Navy ships: debugging notes can be found on the walls of trucks carrying computers that were used in Vietnam. It is only through such modifications that software becomes reliable." David L. Parnas

**SOFTWARE TESTING**

The understanding of bugs in programs has changed since the 1940's, although the idea of having an error in a program is as unpleasant as having a bug inside the computer. The metaphor describing a software error as a bug is confusing. An error can cause reactions as if it was a living insect that should be removed by using poison while programming, but it is, of course, created by the programmer and should as such be regarded as a software error. The word bug is stuck in our language and it is probably as difficult to get rid of as errors in software. The art devoted to finding software errors is called software testing. Myers (1976; 1979), for example, defines software testing simply as being the process of executing a program with the intent of finding errors. The view to software testing has been changing during the last decades. The early view of programming and testing was that you "wrote" a program and then you "checked it out." Later testing has been defined as evaluation of software or prevention of problems. It is widely accepted that software testing is one of the corner stones of modern software engineering and thus should be an integrated part of any quality program. The field of software testing spans mathematical theory, the art and practice of validation, and methodology of software development (Hamlet, 1988). Software testing literature mostly addresses the relationship between the testing and the software engineering process (i.e., the use of testing methods and tools). A broad array of testing methods and techniques are available today, e.g., black and white box testing techniques providing a systematic approach to the design of test cases (Sommerville, 1995). A fast growing flux of automated software testing tool products influences the field today. The vast majority of software organisations have substantial room to improve the manner in which testing is
conducted and software testing is often the poorest scheduled part of programming. If the importance is not recognised, the project planning will not include enough time. The complexity of the testing work and a tight schedule influence on the decisions determining whether or not a software product meets its requirements. That is, there is no systematic way to search, no way to judge points selected, and no way to decide when to stop (Hamlet, 1988). Since exhaustive testing is impossible (Myers, 1976; Myers, 1979; Parnas and Clements, 1986), a common understanding of the status of a software product can only be established by means of negotiations. These situations are not easy to cope with and will often result in work settings including many co-operating actors. The actors in an ensemble developing and testing software become interdependent.

**SOFTWARE TESTING IN THEORY AND IN REAL LIFE**

Software testing is often a complex process potentially involving a large number of geographically distributed people with different perspectives and competencies. Software testers, software developers and project managers engage in discussions about the software errors found, they negotiate the relative importance of the bugs, they allocate responsibilities and resources, they co-ordinate who is doing what, etc. They talk about bugs. In order to co-ordinate and manage talking about bugs, a number of means for co-ordination are applied.

The software testing workflow (Carstensen et al., 1995)

- Registration and Classification
- Diagnosis
- Correction
- Verification
- Monitoring
THE CATHEDRAL OR THE BAZAR

The organisational context of program production has been expanded in recent years, amongst others in ways that fundamentally change the ways we collaborate in building software. From the early days of computing and until the personal computer emerged, can primarily be characterised as the era of bespoke systems. With the increasing industrialisation of the software industry, the software has turned from process to product. We do not buy developer time when we buy software we buy a product. Companies such as IBM, Microsoft, Lotus (now part of IBM), SAP and many others have made it a lucrative business to sell standard application packages. In fact, these days it seems futile in most cases to develop systems from scratch instead of "gluing" existing systems together using dedicated scripting languages (Osterhout, 1998). The companies write applications and as such the source code for these applications and the people who are experts in the source code represent the primary capital of the firm. In the mid-eighties, a movement began which now is causing a lot of discussion in the software industry. Richard Stallman wrote in 1985 the GNU Manifesto (Stallman, 1985), and initiated the Open Source movement. Systems such as the Apache Web-server, the Linux operating system, and most recently, the Netscape web browser and parts of the Macintosh operating system are characterised by some degree of open and public source code. The idea was that software ought to be free and available to everybody. The software industry clearly did not see it as a viable strategy to give away their primary capital, but recent events have made some of them reconsider. Thirteen years after Stallman's manifesto, Eric Raymond (1998) finally found the words to win non-believers over to what he called "open source." Netscape CEO Jim Barksdale allegedly decided to give away the source code for the company's browser after reading this. The main idea behind open source software development according to Raymond, is to look at software development not as building a cathedral (Brooks Jr., 1982; Brooks Jr., 1995), but as a chaotic bazaar with no obvious centralised control. One of the main reasons for the success of open source software in the late 1990ies has been the fact that 1000
programmers distributed all over the Globe represent a mighty force compared to a couple of hundreds in a particular software company. Although it is a highly complex task to manage the development process involving many people, the process of finding errors in software can benefit from thousands of people helping to find the errors and perhaps even suggest and build extensions. Brooks' notion of software development as the process of building a cathedral thus changes to one of many independent participants each contributing their bit to the whole (Raymond, 1998). Development projects such as Linux, and Apache are examples of this. Donald Knuth [, 1989 #86] documents with great elegance the long process of debugging the typesetting language TEX, involving many volunteers over long time. The paper is highly recommendable reading for anyone who wishes to peek inside the process of software testing. Concluding, we can observe that most of the lessons learned more than 20 years ago by Brooks (1982; 1995) still can tell us important things about software development. However, we must be aware of the possibilities offered by global competition, and by a global networked community in terms of pooling resources. Clearly, at the same time as this major change in the organisational conditions for developing software provides new and interesting possibilities, it also imposes new challenges to devise economic models

**Reading**

There are a number of core software engineering texts around that you can read. The most famous general text books that contain everything you need to know about the basic software engineering techniques are written by Sommerville (1995) as listed above, and Pressman (1997). The most famous book on software engineering is beyond doubt written by Brooks (1982) and is highly recommended for anyone with an interest in the topic. (Curtis et al., 1988) on the importance of domain knowledge in large software projects (IS471 study pack). The initial chapter in Planet Internet (Braa et al., 2000) outlines some of the issues related to the Internet. In general, there are at least the following four journals of interest when studying modern software engineering from a "semi-technical" perspective. IEEE Computer and Communications of the ACM both provide relatively broad overview papers. IEEE Software is more specifically aimed at software engineering researchers and professionals and quite a lot of the material published here deals with issues related to the improving software engineering practices. Issues of all four journals have themes so when you browse the back issues it is reasonable easy to find a number of papers on a particular subject. For the particular interested, the November/December issue of IEEE Software deals with professional software engineering. The articles discusses how and on what basis to establish a software engineering profession with a recognised code of practice and a standardised set of tools and body of knowledge on software engineering. One of the projects described is the Software Engineering Body of Knowledge project that seeks to systematise the body of knowledge on software engineering (Bourque et al., 1999). The project website (www.swebok.org) provides an abundance of documents and information, but just as with the Software Engineering Institute web site (www.sei.cmu.edu), there is a risk of drowning in all the information.

**References**


http://iris.informatik.gu.se/sjis/magazine/vol7no2/CARSTE~2.htm#E37E3


