

# Three practices that reduce complexity in systems development

by Niklas Saers, May 2002

## Abstract:

This essay will investigate the work practises of using interpreted languages, prototyping and unit testing and see if they can reduce four kinds of complexity. It will first introduce the problem of complexity and then the three practises. Then follows a discussion on how the practises reduce the complexities and what new complexities are introduced. It concludes that the practises indeed reduce complexity, and that the reduction is greater than the complexities introduced. Therefore these are practises that should be used in software development projects.

## Table of contents

Introduction.....	2
The problem of complexity .....	3
Three practises .....	5
Interpreted languages .....	5
Prototyping .....	6
Unit testing.....	7
Reduction of complexity.....	9
Prototyping .....	9
Interpretive languages.....	12
Unit testing.....	15
References.....	21

## Introduction

In the early days of software development, the users and the software developers were the same people. Therefore, how the users would understand the program was not a question. As the software development industry grew, the focus was on big applications for large corporations such as banks. These systems would be built by in-house programmers and take years to develop as the programmers would build their own development tools and all the modules for a program.

Today, very few companies not in the business of software development have their own programmer staff. At the same time, companies in the business of software development are facing increasing challenges. The company that can offer the shortest time to market at the lowest cost gets the job. However, the company that makes a system that fulfils the customer's needs will have a returning and loyal customer. This means that the companies must both have a rapid production, low costs and be good at understanding the customers.

Reducing time and costs is a matter of reducing organizational complexity. There are many kinds of complexities within an organization. Some examples are the structure, the management policies, the social interactions of employees within departments, within one department or project, the physical location of employees and the ways of interaction with customers. This essay will look at the complexity of the development environment of a project and suggest three practices that will help reduce the complexity within the project: the use of interpretive languages, prototyping and unit testing. After introducing the practices this essay will show how they help reduce complexity supported on literature from the systems development research area.

## **The problem of complexity**

The cost of developing software far exceeds the cost of a computer. At the time of writing, a standard office computer costs around 10.000 NOK, while consulting agencies developing software for customers can charge 1.200 NOK and more pr hour. This is because while the computer industry has developed an economy of scale, the software development industry for the most part is concerned with tailoring software systems to the customer. The customer expects computers to help automate the company's tasks and thus cut costs. He expects the software to fulfil his needs, and to be delivered on time within budget. If the project goes over budget, does not fulfil his needs correctly or is severely delayed, the customer will loose money. As a software engineer, it is our job to understand the customer's needs, give a reliable cost/benefit estimate and supply in time a product that will be useful to the customer. We will define useful as saving more costs than the total cost of the system development, the implementation of the system into the company and the required training of the employees that will use the system.

There are two parameters to the usefulness: how much the software costs and how much it saves. Assuming we make the right product for the customer, the amount it saves is dependent on the use of the product and the customers business. This parameter is usually outside our control. The amount it costs, however, is a variable we can adjust. Since developer time is our main expense, reducing time is a way of reducing costs. The time spent is a matter of reducing the complexity of the job. Thus, through making the project simpler, we will save time, save money, and finally have a greater chance of the product being useful to the customer.

There are four issues of complexity we set out to simplify in this essay. First, the software development process more often than not is full of change, changes that need to be dealt with for the software to become useful. The customers perception of where the software fits into his organization, the features the customer wants, the changes in the organization of the customer, the technological imagination of the customer, all these things are sources of requests for change from the customer. The understanding of the customer by

the development team, introduction of new technology, standards and work practises, new requirements by upper management, new acquired knowledge (through courses or hiring new employees) are just some of the sources of change from the software development company. Therefore it is important that the project can see change coming rather than being caught up by it, and be able to handle change in a way that does not introduce more complexity into the project.

The second issue of complexity is unintended consequences. The issue is well exemplified by the introduction of cane toads into Australia. They had been used with huge success on the Caribbean islands for getting rid of the cane beetle. However, they adapted so well to the Australian fauna that they ate the food of the local frogs. Since they were poisonous and therefore had no natural predators, their numbers could not be held in check. This has become the biggest environmental disaster in Australia. Thus, finding techniques for reducing the risk for and dealing with unintended consequences therefore reduces complexity and makes the product more useful because costs are reduced.

Thirdly, a project has at least a client and a developer organization. The introduction of more clients, outsourcing of parts from the developer organization or hiring of multiple developer organizations by the client will greatly increase the complexity of the project. An example of multiple developer organizations this is when banks build transaction systems they hire multiple companies so that no one company can control a transaction. This reduces their risk of fraud. An example of outsourcing is that Microsoft outsources the development of localized dictionaries for their text editor Word.

And finally, although it has been touched upon within change, there is the complexity of new technologies. There are many alternative ways of running projects, and many different products and technologies can be used. The change of new technology is dealt with above, but the organization needs to learn technologies and prove them useful for their projects before they can attempt to integrate them into their projects.

## **Three practises**

We will look at three practises that are believed to be able to reduce the before mentioned complexities. These are the use of interpretive languages, prototyping and unit testing. Then we will support the claim that these practises can indeed reduce the complexities listed above and therefore increase the chance of the product being useful to the customer. The use of interpretive languages is common to the “internet software development models”, unit testing is one of the best-known features of the “eXtreme Programming” software development model, and both the evolutionary software development model and eXtreme Programming advocate the use of prototyping.

### **Interpreted languages**

An interpreted language is a language that must be interpreted before the code can be executed. Many authors call this scripting languages [Oustehout, 1998], but Lutz [Lutz, 1996: 697-708] makes an excellent point that scripting is not well defined. Scripting is often referred to batching jobs together, and indeed the earliest known interpreted language, Job Control Language (JCL), was made for exactly this purpose: sequencing job steps in OS/360. While it is true that one area interpreted languages are widely used are for building scripts of programs that are going to be run in a particular fashion with particular data and parameters, interpreted languages also excel in areas like user-applications where most of the work is building the GUI and web-applications where what is done and outputted is dependent on the users actions. This can no longer be considered a script.

Interpreted languages stand in contrast to traditional programming languages, hereafter called system programming languages, which must be compiled and linked before they can be executed. When compiled and linked, programs can be distributed and run by themselves without the need of a parser. Because of the interpretation, interpreted programs usually run slower than compiled ones. The compile step allows the compiler to

optimize the code for faster running. However, because of the small cost of a faster computer in comparison to development time, this is not a significant factor if the development time can be reduced. The obvious saving in development time is that the cycle from coding to testing the new code is reduced from code-compile-link-test to code-test. Even though the compiling and linking are in the general case an automated process, this overhead can be large for a complex system and the cost high compared with the small amount of code that typically has been added since the last compile and linking. We will see in our discussion that a great time saver is the high line-to-instruction level of interpreted languages.

Interpreted languages are easily extendible and often contain many packages with ready-to-use functionality. This is by no means true only for interpreted languages. Sun, for instance, provides thousands of classes with the release of its Java Development Kit 1.4. However, many much used languages have a very standardized set of libraries that are included (C and C++ are for instance ANSI standards), and include far less functionality by default. Even so, it is common for interpreted languages to be easily extended by making it easy writing interface to libraries and systems built with system programming languages. This also reduces the number of lines of code a programmer has to write to get the job done.

## **Prototyping**

The combination of many packages by default, easy to integrate other packages and fewer lines to code to implement the functionality you want makes an ideal setting for prototyping. According to Budde et. al [1991: 6-9], *prototyping is an approach on evolutionary programming where early versions through experimentation and feedback converge towards the system the customer wants* [Saers, 2002]. Prototyping must therefore not be confused with what is usually understood with a prototype, a working model of how the final product is going to be.

From our definition, we can see that prototyping is an evolutionary approach<sup>1</sup>. This is in contrast to the waterfall model which in turn was based on the stepwise model. These common models are based on a stepwise approach where the team first checks if it is feasible, then makes the plan and requirements specification, then does the design followed by code, integration and implementation, and finally maintenance. These models are document-driven, and are therefore a great tool for management. The main objection to this approach is that you cannot plan everything and the author knows of no projects that have been entirely specified correctly in the start of the project. Because of the need to go back and edit the documents all the time, it becomes very costly. Dr. Royce himself, the man credited for the creation of the stepwise model, said that in his case scenario, the SAGE project, the stepwise model would not have worked had they not dedicated much time to learning and prototyping [Royce, 1970]. Therefore, while these approaches seem like nice and rational plans, they do not make the day. The opposite extreme is Rapid Prototyping, where the prototype is the only valid document in the system development process [Budde et. al, 1991: 8].

## **Unit testing**

Testing is a very important part of software development. According to Brooks [Brooks, 1995: 20] testing and debugging of completed code takes half the project time. Any combination of the  $n$  items in a system can potentially lead to failure. As  $n$  grows higher, the potential for failures rises exponentially. Eric Raymond states that since bugs usually happen from unanticipated interactions between two different lines of code, the number of bugs scale as the square of the number of lines of code [Broersma, 2002]. Therefore, as systems get more complex, the need for testing is greater. The traditional kinds of testing have been white-box, black-box and structural testing. Black-box testing is where certain inputs are given to a piece of code, and the outcome is tested to be as expected. White-box testing (also known as clear-case or glass box testing) is as black-box testing, but the data flow is observed and verified to be correct within the module. The structural

---

<sup>1</sup> A good introduction to evolutionary programming can be found in Chapter 5: Evolution in Computers in Context by Bo Dahlbom and Lars Mathiassen

testing does white-box testing automatically and statistically measures how often different parts of the code are used. This gives an indication of what parts of the code are well tested.

Unit testing is black-box testing, but with a twist. Instead of writing the tests after having written the code, as is done with black-box testing, the tests are written before coding the item. This ensures that the interface is well thought through before beginning and it clearly says when the code for that particular item is finished. Also, rather than becoming a boring task at the end, writing tests becomes a way of thinking through the problem. This ensures that the tests indeed will be written.

The unit tests not only make us think through what we want a particular element of software to do, it also documents it. Unit tests of small parts of the code say what a particular function is supposed to do and how it is supposed to behave. The tests check not only that the code works correctly with correct input, but also that it fails correctly with invalid input. Unit tests do not only test methods, but also combinations of methods. The tests of combinations document the relation between objects and how they interact [Burns, 2001]. Also, the fact that the tests running document for the project manager that the project members actually complete the tasks they set out to do correctly. The English positivists said that what cannot be measured does not exist. This attitude comes as a consequence unit testing, for only with the tests in place do we know how to handle the beast.

## **Reduction of complexity**

Knowing the complexities we set out to solve with the three introduced practises, we will now see if the practises indeed reduce the complexities.

### **Prototyping**

As we will see, it is easier to write and re-write large bits of functionality with interpretive languages. This gives an ideal setting for prototyping where the customer can see and comment on how the program will look and work. Simply displaying a custom user-interface can in some interpreted languages be done with a single line. This helps the customer clearly define what the program is going to do, how the program is going to be used and how it is going to fit into his organization. Working with the customer is taken to its limits with the Scandinavian Tradition where the customers are regarded as co-designers [Ehn, 1993]. In being co-designers, the customer has both more authority and responsibility for the product rather than when the customer is viewed as a participant with opinions that the developers can choose to adhere to or ignore. Carrying this responsibility, however, requires that the customer representative has authority not only from the developer company, but also from the customer company. Prototyping, and especially regarding customer as a co-designer, makes well use of the customer as a resource for knowledge about the organization, which again makes it less likely that the developer company will spend much time writing parts that are not important to the customer. Also, this contact makes it easier for the developer company to adapt the program to changes in the customers company. The early exposing of the customer to the system gives the customer more time to analyse what the introduction of the system will do to his company, thus reducing the chance of unintended organizational consequences. This again requires authority by the participant of the customer company to make the necessary actions if it such consequences should be discovered.

Researchers from the Scandinavian Tradition found that letting representatives from the customer be co-designers gave the customer a technological imagination. Such

imagination can be both seeing technical possibilities and what extra would be required to aid other parts of the organization. However, Braa and Vidgen [Braa et. al: ch. 12] has found that it is not always easy to get the customers to participate this way. Instead of doing this, the customer's participant probably has a well-defined job he needs to attend to. Also, in many professions, there is no room for trial and error. Getting people to work with prototyping can therefore be difficult. Beck [Beck, 1999] objects to the first problem with that if it is not important enough to the customer that they can spare the expense of one full time job for the period of the project, the system will probably not be sufficiently useful to the company in the first place, and the development should therefore not be done. With regards to the little room for trial and error in some profession, a prototyping solution should be negotiated. If there is no room for error while in testing, then this suggests that the customer has not fully understood what to expect. As seen from the SAGE project, even projects traditionally regarded as well structured need at least prototypes if not prototyping. This breaks their intended structure. Being involved in the process through prototyping gives the customer realistic expectations of what they can expect. Also, it boosts working place democracy if those who are going to be affected by and use the system can influence the outcome [Ehn, 1993: 42-45]. This has been an important wish from many worker unions and has proved to increase worker satisfaction. The people who are going to use the system should be the ones who participate in the prototyping as they are most likely to be the experts on the task at hand. [Ehn, 1993]

Developing applications requires the system which the product will be situated in to be quite homogenous. This is even more so when prototyping is used. The developer company can be set for an apparently undoable task when one part of the company wants the system to be one way and another part of the company wants the same part to be completely different. An example is when the Global Information System was built for Veritas, prototypes one office were very happy about were discarded by other offices due to differences in both technology and differences in the way they organized the same work. If this is the case, it is necessary to review whether it is a good idea to build one system for the entire organization or if other solutions should be explored. If the system is

meant to standardize working routines, then the people included in the prototyping should be the people deciding the standard.

Prototyping is good when multiple developing organizations work together because they give better overview than what many pages of text do. Changes in the program are easier to detect when they can be shown than when a new manual has to be compared to a previous one. This is also true when developing work is outsourced.

Thus, we can conclude that prototyping is indeed a powerful tool that can reduce part of the complexities listed.

## **Interpretive languages**

Using interpreted languages makes it easier to write or rewrite large pieces of functionality with less code than with system programming languages. Most interpreted languages have dynamic typing instead of strict typing. Strict typing is where you declare a variable as a given type (for instance an integer) and then use it only as an integer. Using it for other purposes, such as for instance a float, requires conversions. With dynamic typing, a type is given the variable when it is initialized, for instance: `i=b`. If `b` is an number, `i` will now be a number-type, whereas if `b` was an object, `i` will now be an object. When passed to a function, the function determines if the value it was given is of a meaningful type, and therefore no conversion is needed. This reduces the amount of lines that have to be written. That it becomes easier to rewrite functionality makes it thus good for prototyping. But it is also good for writing the actual logic, which verifies that the company has understood what output the input provided is going to produce for the customer. Should misunderstandings have arisen, this can be easily modified. Should changes outside of the control of the customer happen, such as a merger or a change in legislation, the functionality can quickly be adapted to work in the changed environment.

The more work done per instruction is how we define how high-level a language is. Assembly is considered a low-level language as there is a one-to-one mapping between code written and machine instructions. With system programming languages like C, the ratio is around one-to-five. Ousterhout shows that the code-ratio for the language he chose to examine, TCL, was from four to 47 in comparison with C. Since Brooks [Brooks, 1995: 88-94] shows that programmers write roughly the same number of code-lines per year, regardless of language, using higher level languages makes the company more efficient. Fewer lines for writing the same functionality also means less possibility of bugs, which in turn means reduced complexity. According to Eric Raymond [Broersma, 2002], software bugs increase with square the lines of code. This means that the number of unintended consequences due to two lines of code being used in a way not originally thought of is lower for interpretive languages than for lower level languages. Raymond also claims that lines of code quadruples every 18 months, the double of

Moore's law for computing power. If this is true, using interpreted languages gives the developers a competitive advantage in coding time in comparison with developers using system programming languages.

Since interpretive languages are easily extendable, they often have a set of conventions for how extensions should be built. This makes it easy for the programmers to adapt the program to use new technology (such as changing from the CORBA framework to Microsoft's new .NET framework) without knowing the technology intimately. Also, with the conventions on how extensions are built, it is easy for other developing companies, that are engaged through either outsourcing or because the customer has hired multiple developing companies, to understand and use the interfaces their contributions are working with.

The complexity added by the extendibility is the dependence on other libraries and systems that the language interfaces. When writing the system from scratch, you can control the interfaces, functionality and know if the software is installed correctly or not. Being dependent on other modules, you give up this knowledge. To ensure you know how the module is behaving, you can limit the interfaces by being dependent on a special version of the module. This, however, both limits the evolution of the program's usefulness and it can lead to systems having many versions of the same module to please different programs. If these modules are very similar but have for instance a little bit different set of functionality, the functionality being there sometimes but not other times can confuse users. Still, with the requirements of functionality, interoperability, graphical interface and smoothness (as opposed to a crude first version) required by customers makes it impossible even for projects written with system programming languages to write all the functionality from scratch. Instead, they also need to rely on modules. And with the strong typing, this is often more cumbersome with system programming languages than interpreted languages.

Finally, interpretive languages are great for learning new technology. Since it is written in such a high level, one line can be very many lines of lower level code. It is fairly quick to write test applications to try out features of the technology to learn and evaluate how it works and if it is something the project should incorporate as a change. For example, for DCOM, Microsoft's component communication model, learning to program this with C++ requires programmers with long experience to go on week-long training courses, and is still a cumbersome process to get right. However, interpretive languages such as Python have made this close to trivial. Using interpretive languages thus saves the programmers the frustration of not knowing the technology by heart when programming, saves the company training time and makes the company less dependent on the programmers who had that particular training. The programmers can use the training budgets for things they expect to have more use of in their career.

Interpreted languages are thus another factor that can reduce complexity. While it is true that complexity is added, the added complexity is far less than the complexity reduced, and will as the number of code lines steadily increases be further diminished.

## **Unit testing**

According to Latour, objects can have meaning inscribed into them. That unit tests are written before the code, greatly increases the number of tests written for the product. To write a test first, however, implies that you know what the interface will look like, or you work out the interface as you write the tests. The interfaces are what bind the design together. Therefore, inscribed in the practise of unit testing is that design, coding and testing should be done all at once. It also implies that these roles should not be divided into multiple positions, and therefore makes the administration of people in a project easier as people are more involved in the project and have fewer projects they need to attend to than if they worked with for instance design alone.

At first thought, people who do not like test-writing will resist using unit testing. However, Erich Gamma [Beck et al, 1998] shows that programmers that start writing tests tend to become test-infected. That is, they will not write code before they have the test. Resistance to writing tests will fall as the other programmers notice the benefits of testing: certainty that the code you wrote actually does the job and much reduced debugging time. The programmers that write tests are able to restructure the code more radically and still being certain that the program works. Radical restructuring can be needed if new features that are similar to existing features need to be implemented in a way coherent with the existing ones, or if the developer organization has misunderstood the customer needs or the needs of the customer have changed and have to be reflected in the product. Being certain that the job you do is correct and good decreases stress, less debugging time decreases frustration. Being able to make changes that you find are necessary or that the client representative has decided through the prototyping gives you satisfaction. These are things that people that do not write tests will notice, and will therefore want to write tests. Therefore, they will no longer be marginalized and unit testing will become a standard work practise.

According to Brooks [Brooks, 1995: 20], testing takes roughly 50% of the project resources. Not being able to deliver the system on time has traditionally been a result of

under-estimation of the testing. Lam [Lam, 2001] criticizes the traditional black box testing in that it does *not provide any mechanism for engineers to test their components during the development stage*. Testing the component cannot be done until it is written, and testing interaction between components cannot be done until they are written as well. Lam states that the majority of problems are introduced during the early stages of the development, and because the black-box testing cannot be done until the system is finished, they remain undetected until what the development team expects to be the end. The result is more time being used tracking down and fixing bugs at a stage where the project manager probably has both expected and reported the project to be finished. This in turn leads to mistrusts from the client and upper management, and the project can be labelled a failure. Unit testing, on the other hand, writes tests from the earliest to the latest of code, from stand-alone parts to parts that tightly integrate many pieces of code. This makes a collection of tests that guarantees that the system actually works. If a coder is in doubt whether the system works before he integrates his piece of code into the code base, he can run the tests. If he wonders if the code he integrated breaks any parts of the system, for example through Raymond's unanticipated interactions, he can run the tests. This way, the problem of scale has been reduced from a squared scaling problem to a linear problem.

The time taken from writing the tests to having them running and repeated code breaks after the integration of some employees' code tells the project manager if there is a problem. The time to fix the breaking code and the slip of the estimated time for implementation of the functionality the tests were written for are measures for how serious the situation is. This guides the project manager in who is doing a good job and what employees would need further training or coaching. Having converted the problem with a hard quantifiable gut feeling to easily measurable items as well as being able to measure success in a proper way both motivates the project manager and members and makes running the project a less complex task. Also, tests written for a superclass should be applicable for all subclasses. Thus, tests can ensure that standards, defined in the company or industry standards, are being followed.

Unit testing is not only good for resolving unintended consequences within the code-base of a company, it also aids developing in virtual teams if the customer has chosen multiple developing companies, and it helps in integration if certain parts of the work has been outsourced. The tests document how the code works by actually demonstrating the code. Therefore, the time spent communicating, delays and inconveniences because of time zones, different working hours and work schedules, and time spent figuring out how the code is intended to work is greatly reduced. Coding by example rather than coding from a manual is often more detailed and still more concise, which is why most programmers prefer this.

The fact that unit tests document how classes, interfaces and components should work and be used together makes knowledge sharing across the organization, between multiple vendors in a project or between the developer company and companies that work is outsourced to, easier to share. Rather than having to read theoretical documents about how things should be done, they provide examples of how it is done, including if there are workarounds that comes from mismatches between theory and implementation.

Finally, tests are good for implementing new technology. This is because new technology at least provides the same functionality as the old technology. Therefore, the tests that were true for the old technology need to be running for the new technology. For instance, IPv6, the internet protocol planned to replace the existing one (IPv4), supports all the features of IPv4, so by replacing this in your product, the same tests should run. Another example is switching component model from CORBA to Microsoft DCOM. It would probably not make sense to switch component model if the new one could not support the features that the application already relied upon.

In conclusion, unit testing is a practise that documents the work, makes programmers more confident that the job they are doing is correct, and reduces complexity. Complexity within the project is reduced both within the organization and cooperating organizations like other developer organizations or subcontractors. The confidence of the programmers make them dare to make more drastic changes if the need for this is discovered, thus

increasing the chance that the customer will get a useful product.

## Conclusion

Handling organizational complexity is an important way of reducing costs and businesses that do this successfully will therefore be more competitive. We have looked upon four types of complexity, broken two of them down into sub-problems, and through our discussion shown how the three practices can help reduce these complexities. This is summarized in the following table:

	Interpretive Languages	Prototyping	Unit Testing
<b>Changes</b>			
Where does the software fit into the customer's organization?		X	
Features wanted	X	X	X
Organizational changes	X	X	X
Technological imagination by customer		X	
Understanding of the customer by developers		X	X
Following standards and work practises	X		X
Acquiring new knowledge	X		X
<b>Multiple actors</b>			
Multiple clients		X	
Outsourcing of development	X	X	X
Multiple developers	X	X	X
<b>New technology</b>	X		X
<b>Unintended consequences</b>	X	X	X

Interpretive languages let the programmer write more functionality in less time. Because they include guidelines on how extensions should be written, the programmer spends less time learning and understanding the technology upon which the product is built, and can spend more time making the actual product.

Prototyping is making frequent releases that the customer can give feedback to and see how the process is going. With Extreme Programming and especially with the Scandinavian Tradition, customer representatives and the developers work closely together as equals. This makes good use of the customer's knowledge, gives the customer

an even greater say than with traditional prototyping, and makes sure misunderstandings are detected early and the right product is built.

Unit testing makes programming easier, less stressful and debugging easier. It makes changes easier to implement. It documents the code and how pieces of the code should be integrated and therefore reduces the complexity added of communication and code integration in co-development and outsourcing easier. It greatly reduces costs in testing before release and the expenses of writing the tests are therefore easily measurable compared to projects that do not use unit testing.

The extra complexities added of using these practises are far less than the reduction of complexity. We must therefore conclude that these are good and complementary practises and that software developing companies that do not use these already should implement them in their company to draw from their benefits.

## References

- Beck, Kent (1999) *eXtreme Programming eXplained, Embrace Change*, Addison-Wesley: USA
- Beck, Kent; and Gamma, Erich (1998) *Test Infected: Programmers Love Writing Tests*, <http://members.pingnet.ch/gamma/junit.htm>
- Braa, Kristin; Sørensen, Carsten; and Dahlbom, Bo (2000) *Planet Internet*, Studentlitterator: Sweden
- Broersma, Matthew (2002) *Eric Raymond: Linux will rule the desktop*, ZDNet: <http://zdnet.com.com/2100-1104-871395.html>
- Budde et. al (1991) *Prototyping – An Approach to Evolutionary System Development*, Springer Verlag
- Burns, Tim (2001) *Effective Unit Testing*, Association for Computer Machinery (ACM): Ubiquity
- Boehm, Barry W (1988) *A spiral model of software development and enhancement*, IEEE Computer 21: 169-208
- Brooks, Frederick P. Jr (1995) *The Mythical Man-Month: Essays on software engineering, 8<sup>th</sup> edition*, Addison-Wesley
- Ehn, Pelle (1993) *Scandinavian Design: On Participation and Skill*, Chapter 4 Participatory Design: Principle and Practice, Lawrence Erlbaum
- Lam, Gordon (2001) *Run-Time White-Box Testing*, [http://www.jollytech.com/products/autotest\\_whitepaper.html](http://www.jollytech.com/products/autotest_whitepaper.html)
- Lutz, Mark (1996) *Programming Python*, O'Reilly & Associates, California
- Ousterhout, John K (1998) *Scripting: Higher Level Programming for the 21st century*, IEEE Computer, 31: 23-30
- Dr. Royce, Winston W (1970) *Managing the Development of Large Software Systems*, IEEE WESCON: 1-9
- Saers, Niklas J (2002) *Process models for modern software development, a comparison of eXtreme Programming and the Spiral Model*, <http://niklas.saers.com/articles>