February 2007

From Communication to Implementation - A framework for understanding and reducing unintentional complexity in software development processes

Matthias Biggeleben
Johann Wolfgang Goethe University, biggeleben@wiwi.uni-frankfurt.de

In: Oberweis, Andreas, u.a. (Hg.) 2007. *eOrganisation: Service-, Prozess-, Market-Engineering*; 8. Internationale Tagung Wirtschaftsinformatik 2007. Karlsruhe: Universitätsverlag Karlsruhe

ISBN: 978-3-86644-094-4 (Band 1)
ISBN: 978-3-86644-095-1 (Band 2)
ISBN: 978-3-86644-093-7 (set)

From Communication to Implementation

A framework for understanding and reducing unintentional complexity in software development processes

Matthias Biggeleben

Chair of Information Systems Engineering
Johann Wolfgang Goethe University
60325 Frankfurt / Main
biggeleben@wiwi.uni-frankfurt.de

Abstract

Since the invention of “software engineering” in 1968, software development has been suffering from efficiency problems. Software development is bridging the gap between verbally formulated requirements and programming languages. This work equates development with communication. Communication efforts refer to either essential or accidental complexity. This work hypothesizes that accidental complexity is inherent in implementation processes. However, it can be mitigated. This paper discusses an implementation framework that attacks accidental complexity. The framework is tested in an experiment in order to study the hypothesized efficiency gains in a daily programming task. Finally, this work discusses potential reasons for the existence of accidental complexity in software development.

1 Introduction

Since the invention of the term “software engineering” in 1968 [NaRa68] and in spite of all invented methods to improve software engineering [Berr04; Somm01], software development (SD) still suffers from efficiency problems. 23% of SD projects are cancelled due to failure and 49% exceed project resources [Stan01], i.e. projects do not meet deadlines or projects are simply too expensive. Similarly, [KeMR00] note, that between 30% and 40% of all software projects exhibit some degree of escalation.

Generally, a SD process starts with an initial customer's need, which is structured, documented and discussed by techniques of requirements engineering [Somm01]. The process results – if successful – in an implemented and running system. Simplified, SD is bridging the *gap* between requirements (the "left side") and a perfected implementation (the "right side"; see Figure 1).

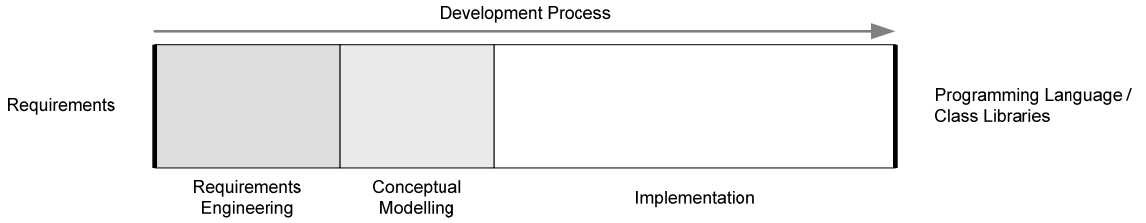

Figure 1: Simplified SD process

SD can be viewed from different perspectives, e.g. programming languages, compilers, operating systems or system architectures. Most perspectives have a technical nature. However, SD can also be understood as a form of communication. Usually, SD is performed within a team. Customers, requirements engineers, modelers, software developer and other relevant stakeholders have to communicate with each other during the course of an SD project. In addition, developers have to communicate with machines through programming languages. Even in the case of a one-man development, there is still a communication process between the developer and the machine. In the following, this work uses the term "*communication*" to unify both social and man-machine communication.

This work concentrates on *shortening* the gap by shifting the very "right side" of the overall development process, leaving other methods of software engineering unaffected. If this shift is realizable, the procedures developers usually follow will be truncated. Consequently, the process will demand less effort and less communication. This work proposes an *implementation framework* to realize that shift (see Figure 2).

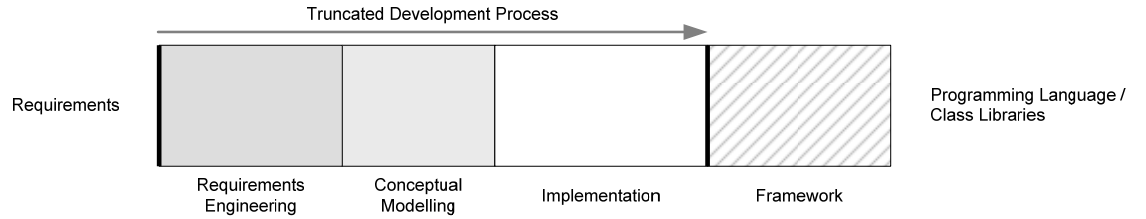

Figure 2: Truncated SD process

The crucial motivation for the framework is to free programmers from anything, which is not in straight line with problem solution or unnecessarily error-prone, i.e. the identification and elimination of *accidental complexity* [Broo87; Broo95]. In addition, the framework provides and integrates established concepts developers might need to know and implement. By doing so, the framework attempts to prevent communication hazards, which are supposed to exist in development processes.

The paper is structured as follows. Section 2 discusses related work. This section also elaborates three types of communication hazards. Section 3.1 proposes an abstract implementation framework that addresses the presented communication hazards. Accidental complexity is the most promising candidate to attack. Yet, accidental complexity strongly depends on technical aspects, i.e. how concepts can be implemented. Thus, section 3.2 summarizes analyzed aspects of SD. This analysis results from the development of various framework prototypes. These developments can be regarded as a collection of experiments. Details of the research methodology and the hypothesis of this paper are presented in section 4.

Afterwards, section 5 will illustrate a *test* of the hypothesis by analyzing a daily programming problem. The test is based on the current framework prototype. Finally, section 6 discusses the test results and attempts to derive a model of *general* existence of accidental complexity in SD and explain how it can be mitigated.

2 Related Work

Requirements are neither clear at the start nor stable during the development process. Following [Jack94] only two things are known about requirements: they will change and they will be misunderstood. This insight is not due to defects of requirements engineering. In fact, changes are natural. [Lehm80] points out, that real world software – once installed and becoming part of the application domain (termed E-type systems) – alters its own requirements. The software co-evolves with its domain and vice versa [LeRa02]. As a consequence, requirements are adapted, some are removed and new ones are added, which is organizationally managed by change requests.

During development, developers continuously gain knowledge about requirements through communication processes. Analogously and according to [BuMo79, 1] who describe the origin of knowledge, “one might begin to understand the world and communicate this as knowledge to

fellow human beings". [GaNe01, 611] describe knowledge as "justified true belief" and state that "belief refers to an individual's or group's idea about what is truth". As a result, truth is socially constructed. Analogously, software requirements are social constructs as well.

However, it is impossible to verbalize requirements in a form that a machine automatically generates an implementation [Holt03]. For this reason, a gap emerges that cannot be bridged instantly. A machine simply does not understand us. In consequence, there has to be at least one person who is capable of mapping *concepts* formulated (or thought) in spoken language to concepts of a machine represented in a programming language. This transformation is inevitable. Here, the term *concept* is used with respect to [KaLo84]. Kamlah and Lorenzen define *concept* as "the meaning of a term […]: the meaning of a term is that which the term makes understood on the basis of its explicit agreement, which, however, can also be made understood by other signs" [KaLo84, 73].

This work identifies two sets of concepts. The first set contains concepts, termed *R-concepts* (requirements), which are used to define requirements and which refer to the "left" side of development processes. The second set contains concepts, termed *I-concepts* (implementation), a machine understands and provides and which refer to the "right" side of development processes. Both sets refer to a corresponding language community [KaLo84]. With respect to their members, both language communities might be disjoint. This emphasizes the importance of a person who is able to collate concepts of both communities.

The transformation of R-concepts into I-concepts is time-consuming and expensive, since software is complex. "Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike" [Broo95, 182]. Thus, developers have to invest a remarkable amount of time in communication, because it is necessary to communicate each software entity. Conversely, there are communication processes which can be identified as unnecessary with respect to problem solution.

Despite extensive use of the word, there is no generally accepted definition of complexity. Complexity is a multi-facetted term which has many possible meanings [FlCa93; Klir85]. Since complexity is something perceived by an observer, the complexity of the system being observed can be described as a measure of the perceived effort that is required to understand and cope with the system [Back02]. Here, complexity can be understood in terms of number of parts and number of relationships [FlCa93], as well as requisite variety [Ashb65].

Following [Broo95, 182] complexity can be divided into the *essence* and the *accidents*. *Essential complexity* is inherent and unavoidable by definition, since any *simpler* solution would not solve the problem. In contrast, *accidental complexity* is avoidable, since it refers to complexity that we create on our own, i.e. it originates from the process or the techniques of solving a problem. It is not part of the solution. Accordingly, communication processes that focus accidental complexity are unnecessary. However, there has to be an alternative method that avoids accidental complexity. If there is one, accidental complexity can be removed and communication efforts will be reduced.

Additionally, there are communication processes that are unnecessary because developers do not know a particular concept. Instead, they have to invest time to develop and express concepts on their own, e.g. concepts like the model-view-controller or object-relational mapping. The linguist Whorf [Whor56] hypothesizes a relationship between the expressive power of a language and the ability to think a particular thought. If required words are unknown, a person will not be able to express the thought and might not even be able to formulate it in other words. This hypothesis can be transferred to SD [McCo04]. For example, if simple concepts like hashes or dictionaries are not known, they will probably not be used within development, although their convenience might even be known from keys and indexes within databases.

As a result, this work identifies three types of communication hazards:

- to communicate accidental complexity,
- to communicate unknown concepts and
- *not* to communicate relevant concepts.

These communication hazards still refer to communication as a unification of both social and man-machine communication. Each communication hazard is regarded as a source of inefficiency.

With respect to the reduction of inefficiency, a major focus of software engineering has been the organization and *acceleration* of the process of bridging the gap. Popular representatives of this idea are the build-and-fix model [Scha02], the waterfall model [Royc70; Somm01], structured programming [DaDH72], extreme programming [BeAn04], domain specific approaches etc. However, past decades have shown that the efficiency problem has not changed and that Brooks' "silver bullet" has not been found [Berr04; Broo87; Broo95]. Consequently, this work

does not directly focus on the process, but the basis of SD, i.e. the quantity and usability of provided *I-concepts* on the implementation level.

3 Implementation Framework

3.1 Abstract Implementation Framework

The implementation framework is based on a 3rd generation programming language. Accordingly, accidental complexity of mapping high-level language constructs to machine or assembler code does not exist. Additionally, the framework will address an object-oriented programming language. However, object-orientation has not removed accidental complexity [Broo87]. The choice for an object-oriented language just results from the popularity and widespread use of these languages, e.g. C++, C# and Java.

A programming language simply consists of essential language constructs, e.g. assignments, arithmetic expressions, loops, conditions etc. Frequent data structures and functions (which do not extend the syntax of a language) are usually provided by a collection of class-libraries. Commonly, class-libraries offer reusable functionality [KiLa92]. Moreover, generality and extensibility are their design objectives [KiLa92]. Accordingly, class-libraries reduce accidental complexity, because they offer already implemented functions.

Though generality and extensibility are design objectives of class-libraries, these features may also emerge as obstacles. For example, an application has to load an XML document from a web server which requires authorization, validate the document against an XML schema definition and extract some data via XPath [W3C99]. This should be three lines of code. Using class libraries straightforwardly developers need a multiple of lines, because class-libraries contain small but highly reusable structures that have to be composed for problem solution.

In consequence, the implementation framework has to identify the most required functions, which normally have to be composed by developers manually. Thus, the framework has to *pre-compose* frequent functionality, so that a particular functionality is utilizable within one line of code. If a pre-composed function requires customization, this will be management by descriptors that are separated from the code level.

Another major source of accidental complexity is bad design [Broo87]. Consequently, the framework supports and even forces a primary segmentation of the implementation.

Accordingly, the framework demands less effort and ability from programmers. Following [RaTo92] *ability* which includes familiarity with relevant concepts and design still has the strongest effect on software developers' performance. Basically, the framework is divided into *four* segments: *data*, *code*, *graphical user interface* (GUI) and *communication*, which is consistent with *three*-tier architectures like the client/server model (the communication segment connects the tiers). However, the segmentation is transparent for developers.

At this point, the framework appears to be equivalent to architectures like *Java 5 EE* (successor of J2EE; [Sun06]) or the *.NET Framework* [Micr06d]. For distinction purposes, a programming language, its class libraries and corresponding architectures are subsumed under the term "*implementation base*". Nevertheless, the proposed implementation framework does not intend to replace or compete with such implementation bases. It is rather intended to put the implementation framework on such an implementation base. This implicates, that – in spite of a compound implementation base – there is much accidental complexity left in SD processes, which legitimates a further *layer* between developers and the implementation base. However, the framework does not encapsulate the implementation base. Thus developers still have direct access (see Figure 3).

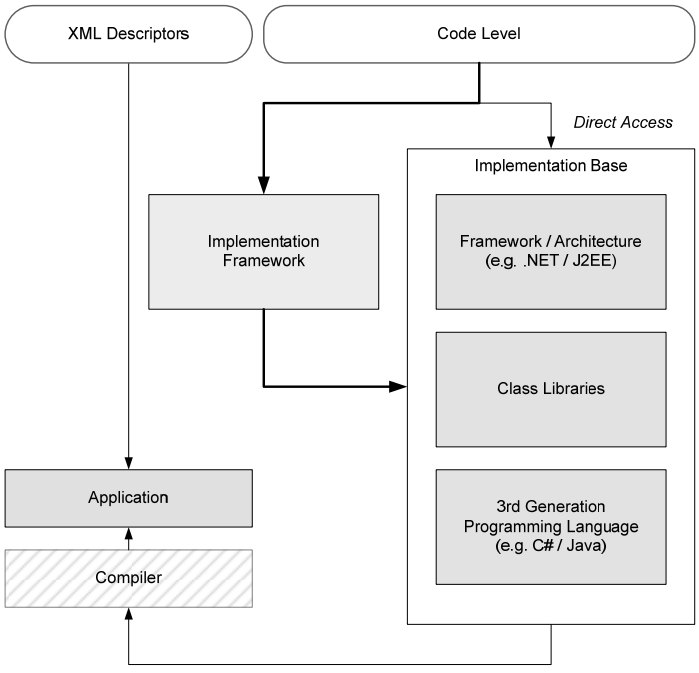

Figure 3: Integration of the implementation framework and an implementation base

For the legitimization of a further layer, the *abstract* implementation framework has to be fulfilled by a *general* explanation for the existence of accidental complexity. In particular, the existence of accidental complexity and ways of mitigation strongly depend on technical aspects.

3.2 Technical Aspects

For abbreviation purposes, this section does not present technical aspects in detail, but gives an impression which aspects of SD have been technically analyzed. This analysis examined differences between *typed* and *untyped* programming languages with respect to productivity [McCo04; Oust98; Prec05], the coherency between a language and productivity in general [BlSW96; MaWD96], the lines-of-code paradox [Jone94], arguments for typed languages [Oust98; RySB05], arguments for untyped languages [Oust98], future features of programming languages ("*Anonymous Classes*", "*implicitly typed variables*", etc. [Micr06b]), data processing based on XML, GUI description based on XML (*XML application markup language* [Micr03]), GUI element access via the *document object model* [W3C04], late-binding, code reflection, promising approaches to integrate script-languages [Hugu04; Micr06c], object-relational mapping [Micr06a; Sun05] and finally standardizing communication via web services [W3C02].
Summarizing, the technical analysis encouraged the assumption that accidental complexity still exists in daily programming scenarios. Moreover, the analysis provided ideas to mitigate this complexity.

4 Research Methodology

This work *hypothesizes*, that accidental complexity denotes a communication hazard that negatively impacts the efficiency of SD processes. Using the proposed framework in SD processes reduces accidental complexity and positively impacts the efficiency.
With respect to software engineering, [DBOB03, 53] state, that "it is certainly arguable whether a positivist approach can ever be appropriate for a discipline so dependent on people and the environment, where carefully controlled and repeatable experiments, which change only one variable at a time, are often difficult or impossible to design and implement". For that reason, this work predominantly addresses a *subjective* and *interpretive understanding* [Lee91].

A collection of simple controlled experiments have been made to provide first data for the *subjective understanding*. These experiments were derived from a superordinate exemplary SD project. With respect to comprehensibility, the project goal was to develop a simplified replica of the software "*EndNote*" [Thom06], which is a tool for publishing and managing bibliographies. The choice for replicating this tool was made due to a lot of experienced crashes, the lack of database support and the missing support for a lot of citation styles (actually *EndNote* is not able to handle the citation style of this paper). Besides, this software type is regarded as popular within the research community.

The SD project was analyzed for frequent programming issues. The dominant issues are database queries (database support), XML (for importing existing EndNote libraries), XSLT (for describing and generating output styles), GUI handling and client-server communication (between the replica and e.g. *Microsoft Word*). Relating to the identification and mitigation of accidental complexity, each issue was subject to a controlled experiment.

All experiments have been conducted by the author. Other persons were not involved. First, a programming issue was isolated from its context. Afterwards the issue was implemented. With respect to the experimental context [KPPJ02], the author is especially familiar with C, C#, Java, JavaScript, Perl, PHP, SQL, XML and XSLT with more than two decades of programming experience. All issues were implemented in C# using Microsoft's .NET Framework (version 1.1) and Visual Studio .NET 2003. Run-time benchmarks have been performed under Microsoft Windows XP Professional (SP2) on a Pentium 4 machine (2.8 GHz; 512 MB RAM).

After the implementation, the resulting code was analyzed with respect to accidental complexity. For example, the essence of a database query is the SQL statement, the corresponding database and the query results. Any further line of code is regarded as an unnecessary communication effort and a possible source of error and therefore accidental.

The results of each analysis were *interpreted* and thus they delivered insights and synergies which helped constructing and adapting the implementation framework. Thus, the experiments were repeated using the first prototypes of the framework. Again, the resulting code was tested for accidental complexity and possible negative side-effects on other issues. With respect to a comparison, the communication efforts of both series, i.e. implementing a solution with or without the framework, have been counted. Here, a communication effort refers to function calls, assignments, loops, object instantiations, type definitions etc. All communication efforts have been weighted equally.

5 Testing the Hypothesis

To test the presented hypothesis, accidental complexity as a communication hazard has to be revealed and mitigated. Accordingly, a common task of SD is analyzed in the following: *database queries*. This refers to one of the experiments derived from the *EndNote* replica development. The analysis addresses the question if this task contains accidental complexity. This section is based on the current prototype of the implementation framework.

The requirements are as follows. The program has to respond to an AJAX request (asynchronous JavaScript and XML) [Garr05; Paul05] or a web-service. The corresponding method has to connect to a database (using a native driver), execute an SQL query, collect its results, serialize them into an XML document, transform the XML document into an AJAX/web-service compatible schema and finally serialize the XML document to a plain string. The program has to support two different database management systems (MySQL 5.0 & Microsoft SQL Server 2005).

The required communication efforts have been classified and counted. A developer has to spend a total of 78 communication efforts (see Figure 4).

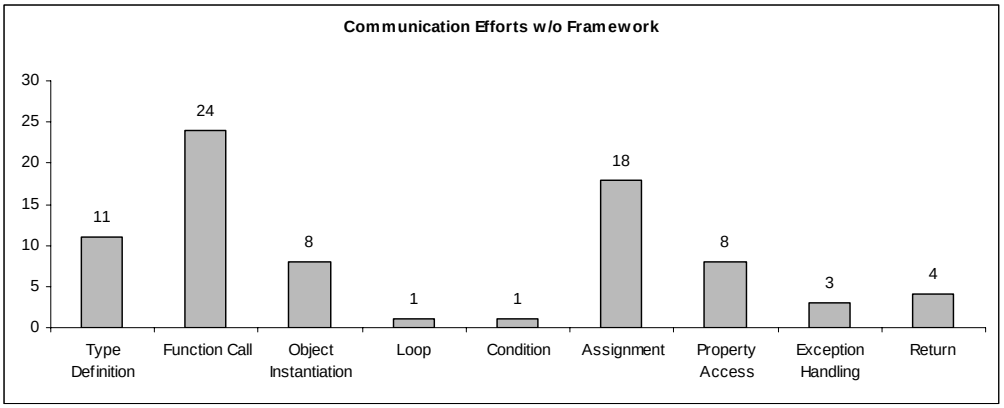

Figure 4: Communication efforts (without framework)

These efforts refer to the composition of basic language constructs and the recomposition of "atomic" elements provided by class-libraries. The latter implies that he or she knows how to implement database access, XML document creation and XSLT transformation. Using the implementation framework, the total effort and thus accidental complexity can be minimized (see Figure 5).

```
public string DemoRequest()
{
    // load global configuration
    Global.LoadConfiguration("configuration.xml");
    // prepare & execute the statement (uses default connection),
    // SQL query is given by name, transform result set to xml, process xslt,
    // return xml as string (implicit conversion)
    return Pool.PrepareByName("Query1").XmlResultSet.Xslt("ajax.xsl");
}
```

Figure 5: The exemplary development task using the implementation framework

The given example demonstrates that the task can be solved within two lines code. The first line loads the configuration (see Figure 6), i.e. all required XML descriptors. It is possible and planned to create and maintain this descriptor by an external application, so that its maintenance demands marginal efforts.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<pif>
    <configuration>
        <system name="Development">
            <database>
                <connections default="testdb">
                    <connection name="testdb">
                        <type>mysql</type>
                        <host>localhost</host>
                        <database>foobar</database>
                        <user></user>
                        <password></password>
                    </connection>
                </connections>
                <queries>
                    <query name="Query1">
                        <sql type="mysql">
                            SELECT proprietary_function(foo)
                            FROM bar
                        </sql>
                        <sql type="mssql">
                            SELECT another_proprietary_function(foo)
                            FROM bar
                        </sql>
                    </query>
                </queries>
            </database>
        </system>
    </configuration>
</pif>
```

Figure 6: Exemplary configuration file

At this time, a default database connection is defined, which also defines the database system type, e.g. MySQL. If the connection fails, a debug dialog will open. The user can immediately kill the process within the debug dialog. Moreover, the debug dialog does some checks to help the developer (send ping to host, connect without a particular database) and it shows the

corresponding configuration lines and the location of the configuration file. If the configuration file is changed, the developer has the possibility to reload the configuration during run-time.
The second line (see Figure 5) prepares and executes the SQL query, defined by its name. All corresponding result sets are serialized to XML. This XML data is immediately transformed by an XSL transformation. The results are implicitly converted to a string and returned.
Using the implementation framework, the communication efforts decrease to a total of 5. The results of the exemplary development task are compared in Tab. 1. The comparison considers communication efforts (CE), lines of code (LOC) and effective lines of code (ELOC), i.e. without braces, empty lines and comments, as well as a run-time benchmark (10.000 iterations). In addition, the comparison presents rounded ratios.

	With & Without Comparison	
	w/	w/o
CE	5	78
LOC	9	121
ELOC	2	48
CE ratio	1 :	16
LOC ratio	1 :	13
ELOC ratio	1 :	24
Benchmark (Total)	21.565 ms	120.482 ms
Benchmark	464 iterations/s	83 iterations/s
Performance ratio	6 :	1

Tab. 1: Comparison of the exemplary development task
both with ("w/") and without ("w/o") the implementation framework

The comparison clearly revealed that the implementation framework reduces communication efforts within the experimental setting. All measures concerning programming effort, i.e. CE, LOC and ELOC, decreased. Especially, the effective lines of code present a ratio of 1:24. In contrast, the run-time benchmark shows that the implementation framework provides better performance (6:1) based on code optimizations and suitable caching mechanisms.

Finally, the ratio of CE and ELOC reveal accidental complexity. While both cases (i.e. with and without the implementation framework) solve the same problem, the framework is able to eliminate approx. 95% of the initial complexity on the implementation level.

6 Discussion

The presented experiment demonstrated the existence of accidental complexity in SD and that this complexity can be mitigated. Thus, the existence of communication hazards could be confirmed, which corroborates the given hypothesis. The development of the implementation framework prototype has already revealed lots of further accidental complexity. At this point, it is interesting to ask if there is a *general explanation* for accidental complexity.

As stated, a 3rd generation programming language is usually provided with class libraries. Analogous to relational database design, class libraries give the impression of *normalization*. In general, the goal of normalization is reduction of redundancy [SiKS06]. This is consistent with class libraries' generality and extensibility.

Given a particular collection of class libraries, there is almost no redundancy, since each construction is *decomposed* into reusable parts, so that the degree of reusability is maximized. In consequence, a developer has to *recompose* "atomic" elements to a sequence of instructions. Thinking in relational database design, there is no functional dependency within any potential sequence. But there might be sequences, which we now call "frequent functions". Accordingly, a *recomposition* of a frequent function is equal to accidental complexity. It can be avoided by "*pre-composed"* functions, i.e. adding *redundancy*, on a framework level. This redundancy is also able to attack conceptual essence [Broo95, 196], since it provides mechanisms developers no longer have to communicate, implement and test and therefore they no longer have to conceptualize these mechanisms either.

Reducing accidental complexity ahead of a project to reduce costs might be criticized as a self-fulfilling prophecy. However, the costs of the development and customization of the framework have to be added to the total project costs. In general, the framework approach pays if its development costs are less than the project costs savings. Still, if this is not the case, it can be an advantageous investment in the long run.

7 Future Research

Future research will concentrate on completing the implementation framework. With respect to the presented hypothesis, a series of controlled experiments with students will follow. These experiments will be based on a test group, which uses the implementation framework, and a control group, which does not use the framework. Both groups will implement identical requirements, which allows a comparison of the development productivity of each group. The results will be helpful to elaborate a hypothesis about the degree to which the framework approach compresses the project schedule. Furthermore, it is decided to perform a case study with an enterprise that is engaged in SD.

Finally, the author would like to thank his reviewers for their recommendations on this work. Furthermore, the author would like to thank the German Federal Ministry of Education and Research, which funded this work under record no. 01AK706.

References

[Ashb65] *Ashby, W. R.*: An introduction to cybernetics. London 1965.

[Back02] *Backlund, A.*: The concept of complexity in organisations and information systems. In: Kybernetes 31 (2002) 1, pp. 30-43.

[BeAn04] *Beck, Kent; Andres, Cynthia*: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, 2004.

[Berr04] *Berry, Daniel M.*: The Inevitable Pain of Software Development: Why There Is No Silver Bullet. In: Proceedings of the Radical Innovations of Software and Systems Engineering in the Future, Venice, Italy, 2004, pp. 50-74.

[BlSW96] *Blackburn, Joseph D.; Scudder, Gary D.; Van Wassenhove, Luk N.*: Improving Speed and Productivity of Software Development: A Global Survey of Software Developers. In: IEEE Transactions on Software Engineering 22 (1996) 12, pp. 875-895.

[Broo87] *Brooks, Fred P.*: No Silver Bullet: Essence and Accident in Software Engineering. In: IEEE Computer 20 (1987) 4, pp. 10-19.

[Broo95] *Brooks, Fred P.*: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, Reading, MA 1995.

[BuMo79] *Burrell, G.; Morgan, G.*: Sociological Paradigms and Organisational Analysis. Aldeshorst et al. 1979.

[DaDH72] *Dahl, O.-J.; Dijkstra, E. W.; Hoare, C. A. R.*: Structured Programming. Academic Press, London 1972.

[DBOB03] *Dawson, Ray; Bones, Phil; Oates, Briony J.; Brereton, Pearl; Azuma, Motoei; Jackson, Mary Lou*: Empirical methodologies in software engineering. In: Proceedings of the Eleventh Annual International Workshop on Software Technology and Engineering Practice, 2003., Amsterdam, The Netherlands, 2003, pp. 52-58.

[FlCa93] *Flood, R. L.; Carson, E. R.*: Dealing with complexity. Plenum Press New York, 1993.

[GaNe01] *Galliers, Robert D.; Newell, Sue*: Back to the future: from knowledge management to the management of information and data. In: Proceedings of the European Conference on Information Systems, Bled, Slovenia, 2001, pp. 609-615.

[Garr05] *Garret, Jesse James*: Ajax: A New Approach to Web Applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, 2005, Last access 2005-03-29.

[Holt03] *Holten, Roland*: Integration von Informationssystemen. Theorie und Anwendung im Supply Chain Management. 2003.

[Hugu04] *Hugunin, Jim*: IronPython - A fast Python implementation for.NET and Mono. http://www.ironpython.com/, 2004, Last access 2006-07-05.

[Jack94] *Jackson, M.A.*: The Role of Architecture in Requirements Engineering. In: Proceedings of the Proceedings of the IEEE International Conference on Requirements Engineering, Colorado Springs, 1994, p. 241.

[Jone94] *Jones, C.*: Assessment and Control of Software Risks. Yourdon Press, Englewood Cliffs, N.J. 1994.

[KaLo84] *Kamlah, Wilhelm; Lorenzen, Paul*: Logical Propaedeutic. Pre-School of Reasonable Discourse. University Press of America, Lanham 1984.

[KeMR00] *Keil, Mark; Mann, Joan; Rai, Arun*: Why Software Projects Escalate: An Empirical Analysis and Test of Four Theoretical Models. In: MIS Quarterly 24 (2000) 4, pp. 631-664.

[KiLa92] *Kiczales, Gregor; Lamping, John*: Issues in the design and specification of class libraries. In: ACM SIGPLAN Notices 27 (1992) 10, pp. 435 - 451.

[Klir85] *Klir, G. J.*: Complexity: some general observations. In: Systems Research 2 (1985) 2, p. 131–140.

[KPPJ02] *Kitchenham, Barbara A.; Pfleeger, Shari Lawrence; Pickard, Lesley M.; Jones, Peter W.; Hoaglin, David C.; Emam, Khaled El; Rosenberg, Jarrett*: Preliminary Guidelines for Empirical Research in Software Engineering. In: IEEE Transactions on Software Engineering 28 (2002) 8, pp. 721-734.

[Lee91] *Lee, Allen S.*: Integrating positivist and interpretive approaches to organizational research. In: Organization Science 2 (1991) 4, pp. 342-365.

[Lehm80] *Lehman, M. M.*: Programs, Life Cycles and Laws of Software Evolution. In: Proceedings of the Proceedings of the IEEE, 1980, pp. 1060-1076.

[LeRa02] *Lehman, M. M.; Ramil, Juan F.*: Software Evolution and Software Evolution Processes. In: Annals of Software Engineering 14 (2002) 1-4, pp. 275 - 309.

[MaWD96] *Maxwell, Katrina D.; Van Wassenhove, Luk; Dutta, Soumitra*: Software Development Productivity of European Space, Military and Industrial Applications. In: IEEE Transactions on Software Engineering 22 (1996) 10, pp. 706-718.

[McCo04] *McConnel, Steve*: Code Complete. Microsoft Press, Washington 2004.

[Micr03] *Microsoft*: Chapter 1: The "Longhorn" Application Model. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnintlong/html/longhornch03.asp, 2003, Last access 2006-07-03.

[Micr06a] *Microsoft*: ADO.NET 2.0. http://msdn.microsoft.com/data/ref/adonet/, 2006, Last access 2006-07-05.

[Micr06b] *Microsoft*: C# Version 3.0 Specification. http://download.microsoft.com/download/5/8/6/5868081c-68aa-40de-9a45-a3803d8134b8/CSharp_3.0_Specification.doc, 2006, Last access 2006-07-03.

[Micr06c] *Microsoft*: IronPython. http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython, 2006, Last access 2006-07-05.

[Micr06d] *Microsoft*: Microsoft.NET Framework 3.0 (formerly WinFX). http://msdn.microsoft.com/winfx/, 2006, Last access 2006-07-05.

[NaRa68] *Naur, P.; Randell, B.*: Software Engineering: Report an a Conference Sponsored by the NATO Science Commission. In: Proceedings of the Garmisch, Germany, 1968.

[Oust98] *Ousterhout, John K.*: Scripting: Higher-Level Programming for the 21st Century. In: IEEE Computer 31 (1998) 3, pp. 23-30.

[Paul05] *Paulson, Linda Dailey*: Building Rich Web Applications with Ajax. In: IEEE Computer 38 (2005) 10, pp. 14-17.

[Prec05] *Prechelt, Lutz*: An Empirical Comparison of Seven Programming Languages. In: IEEE Computer 33 (2000) 10, pp. 23-29.

[RaTo92] *Rasch, Ronald H.; Tosi, Henry L.*: Factors Affecting Software Developers' Performance: An Integrated Approach. In: MIS Quarterly 16 (1992) 3, pp. 395-413.

[Royc70] *Royce, W. W.*: Managing the Development of Large Software Systems: Concepts and Techniques. In: Proceedings of the Proceedings of WesCon, 1970.

[RySB05] *Ryder, Barbara G.; Soffa, Mary Lou; Burnett, Margaret*: The Impact of Software Engineering Research on Modern Programming Languages. In: ACM Transactions on Software Engineering and Methodology 14 (2005) 4, p. 431–477.

[Scha02] *Schach, S. R.*: Object-oriented and Classical Software Engineering. McGraw-Hill, New York 2002.

[SiKS06] *Silberschatz, Avi; Korth, Hank; Sudarshan, S.*: Database System Concepts. McGraw-Hill, New York 2006.

[Somm01] *Sommerville, Ian*: Software Engineering. Pearson Education Limited, Essex 2001.

[Stan01] *Standish Group International, Inc.*: Extreme CHAOS. In: Research report, ordering information available at www.standishgroup.com (2001).

[Sun05] *Sun Microsystems Inc.*: J2EE - Enterprise JavaBeans Technology. http://java.sun.com/products/ejb/, 2005, Last access 2005-03-29.

[Sun06] *Sun Microsystems Inc.*: Java EE At a Glance. http://java.sun.com/javaee/, 2006, Last access 2006-07-07.

[Thom06] *Thomson Corporation*: EndNote. http://www.endnote.com/, 2006, Last access 2006-07-14.

[W3C02] *W3C*: SOAP Version 1.2 Part 0: Primer. http://www.w3.org/TR/2002/WD-soap12-part0-20020626/, 2002, Last access 2005-03-29.

[W3C04] *W3C*: Document Object Model. http://www.w3.org/DOM/, 2004, Last access 2004-12-15.

[W3C99] *W3C*: XML Path Language (XPath). http://www.w3.org/TR/xpath, 1999, Last access 2005-03-29.

[Whor56] *Whorf, Benjamin*: Language, Thought and Reality. MIT Press, Cambridge, MA 1956.