Information Systems on Mathematical Foundations
Turning Software Development into Software Engineering

David Matthys
Promotor(s): Prof. Dr. Ir. Herman Tromp

Abstract—Despite noticeable advances in technology, and numerous
methods available in the field of analysis and design, software projects of-
ten fail to meet their timing schedule and go beyond imposed budget con-
straints. Even worse, they rarely offer their intended functionality.

Several causes lie at the heart of this situation. Requirements can and
will evolve radically in a short period of time. Information systems only
seem to grow in terms of size and scope. What we do lack most, however,
is a mathematical foundation for the entire software construction process,
ranging from initial objective determination to finalized product delivery,
and ongoing maintenance.

Our aim is to investigate whether a similar framework can be elaborated.
This would constitute a major step for software development to become a
fully fledged engineering discipline. If this goal turns out to be unfeasible,
we would like to establish this observation by means of a formal proof.

Keywords—software engineering, process, mathematics, formalization

I. INTRODUCTION

First of all, we will attempt to clarify what we mean when
talking about software projects. Our focal point lies in the
study of software applications used in big companies (e.g. bank-
ing, insurance, automotive) and government institutions (e.g.
health care, child allowance) to support (or even enable!) the
execution of their daily operations. Typical characteristics in-
clude a.o. heavy investments, a large customer base and high
performance demands.

Our attention here is particularly drawn to the fact that such
projects suffer from remarkably high failure rates[1]. Multiple
reasons can be identified as being potential culprits:

• IT specialists1 fail to grasp the stakeholders’ cravings to their
  full extent.

• In turn, the stakeholders often find it very hard to express ex-
  actly what they want.

• Carrying out an accurate estimation about required budgets
  and the probable duration is by no means an easy task.

This list is not exhaustive, but as far as the author is con-
cerned, it constitutes a representative footing, onto which many
other problems can be reduced. For example, it goes beyond
doubt that as a project grows in magnitude and scope, the odds
on a successful termination rapidly decline. One could argue
however, that this element does not represent an issue on its own
that much, but rather aggravates the negative influence exerted
by the items in the enumeration we’ve presented just above.

D. Matthys is associated with the Ghislain Hoffman Software Engineering
Lab (GH-SEL). This research group is part of the Department of Information
Technology (INTEC), Ghent University, Belgium. The author can be reached at
Davyd.Matthys@UGent.be.

1We use the term ‘IT specialist’ to indicate the employees taking care of the
actual implementation of a project: analysts, architects, designers, developers,
database administrators, network specialists etc.

2We use the term ‘stakeholder’ to denote the people who own significant in-
terests in the outcome of a project, without being directly involved in the actual
implementation: end users, program managers, HR staff etc.

II. DEVELOPMENT VS. ENGINEERING

The next question is: ”Why do we keep on encountering these
difficulties?” After all, we have a plentiful arsenal of methods
at our disposal, helping us to fulfill various goals:

• The specification of a piece of functionality.

• The gathering and assembling of requirements.

• The delineation of a design modeling the program.

• The translation of the previous information into code.

But this is where we find ourselves in dire straits: all of those
methods are based on experience, acquired over several tens of
years. At best, you can describe them as heuristics. Not one
is scientifically underpinned. Stated otherwise, we can only as-
sess, not measure, how ‘good’ or ’bad’ they are at performing
the job they were destined to do. We cannot prove our judge-
ment, nor can we calculate the accuracy thereof.

Let’s consider an example originating from the field of civil
engineering, viz the construction of a suspension bridge, span-
ing across some waterway or gorge. We know how to quan-
tify (nearly) each aspect of that structure. Materials science
learns us about the strengths and weaknesses of the compos-
ing substances. We can fairly accurately model and predict the
forces brought to bear on the piles, frames and arches by geol-
ogy, weather and climate.

When we want to build information systems, be it for the in-
dustrial world or public agencies, we find ourselves doing with-
out a mathematical toolbox of the same kind. We are talking
about software development indeed: the manufacturing of soft-
ware applications is more craft than science.

Looking at the illustration of our suspension bridge, let’s as-
sume software to behave as as system. In this respect, we ought
to gain insight into the internal mechanisms that intervene be-
 tween its constituents, as well as penetrate the influences exer-
cised by external forces.

In short, software engineering calls for a process telling us
what kind of steps to take, and in what order, to end up with
a working program. Above all yet, this mode of operation, to-
gether with the product(s) it delivers, should be embedded into
a formal groundwork, as is the case in other engineering disci-
plines.

For the remainder of this paper, we will break up this process
into four different stages, and devote one section to each (III-
VI). We do not claim that those stages are entirely distinct, nor
do we pretend that another subdivision is not possible. This is
just our personal approach to try and tackle the problem at hand.

The objective of this research is either to draw up a mathe-
matical framework, aimed at the formalization of the software
development process, or to prove that such an undertaking is not
achievable.
III. Domain Model

When information scientists are solicited to create a new software application for any particular company, they tend to focus their attention exclusively on that application, and forget everything else around them. For all that, it is vitally important for them to obtain a thorough understanding of the problem domain. This will allow them to speak the language of their clients; the other way around is not an option! Furthermore, it will help them to appreciate how and where their work fits into the larger picture.

In our endeavor towards formalization, we esteem ontology as a prime candidate to deal with the aspect of domain modeling[2]. On the other hand, some question marks have to be raised here, notably about our (in)ability to define entities. If, for instance, we were to confront a random number of people with the question to give us a definition of a dog, a lot of varying descriptions are bound to emanate. Some of the properties stated will be contradictory, others will overlap[3]. This remark only serves to illustrate the point we’ve made at the end of the previous section: is formalization feasible at all, and if so, to which degree?

Regardless of the outcome, many technicians and managers are likely to neglect domain modeling beforehand: it gnaws at an already tight schedule, it absorbs precious amounts of money, and in return brings zero value to the customers (who are domain experts by definition).

The more superfluous the effort seems to the stakeholders, the more significant and compelling it becomes for the IT specialists. Let’s refer back to section I on page 1, where we mentioned three major problems to explain the large number of failing software projects. Two of them are chiefly communication issues, which could be solved by laying down a common and agreed upon vocabulary and terminology. Additionally, the knowledge earned in these activities begs to be reused in the shaping of ‘reference’ ontology standards for entire industrial branches, see again [2].

IV. Requirements Analysis

Arriving at the stage of requirements analysis, we can afford concentrating on the mere application, instead of investigating the environment in which it eventually resides. We persevere into a business viewpoint, thinking about what we want our program to accomplish. The author wants to stress here that functional requirements3 and non-functional4 requirements are complementary concerns, that deserve equal notice and introspection.

This is a suitable place to embark our own experiences harvested in the graduate course "Object Oriented Analysis". This subject is followed by final year students in computer science, at this very faculty. They are taught to steer away from the position of the programmer, and think in terms of user requirements, using solely UML as a medium of communication. Our intention is to repeat those exercises using formal notations exclusively, or in combination with UML and other languages as suggested in [4]. Accessory pointers appealing for further exploration are handed by [5].

V. Design and Implementation

Now the time has come for architecture to prominently step forward as a key concept in our exposition. Architecture allows us to attain a design and corresponding implementation in which the software is extendable and modifiable.

Today’s enterprises target smooth integration of old and new technologies into their business processes. In this light, interest in service-oriented software architectures is gradually augmenting. Fiadeiro claims category theory to be a perfect fit to represent these structures[6], so we propose to put this statement into practice using cases present within our research group.

VI. Follow-up and Maintenance

To conclude our plea, we suffice with placing a brief comment. Software should be instrumented, as to record its behavior under operational conditions. The collected data can be compared with our initial requisites, therefore aiding us to learn from our mistakes and draw lessons for the future ahead.

VII. Wrap up

Recapitulating our objectives, we want to see whether a mathematical framework for the entire process of software development can be elaborated. If this goal turns out to be unfeasible, we would like to establish this observation by means of a formal proof.

We motivated our inquiry by identifying obstacles in contemporary software development. We put forward to break up the process into four distinct parts, and elucidated each one.

Acknowledgments

The author would like to thank prof. dr. ir. Herman Tromp for making this PhD possible. The other members of the GH-SEL group (Bram Adams, Kris De Schutter and Stijn Van Wonterghem) provided me with ample useful suggestions, as well as bringing in multiple corrections. Furthermore, we are none the less grateful towards ir. Koenraad Vandenbore, for coming up with this interesting subject, and sharing his ideas.

References