How Does Your Software Measure Up?

Mastering the DNA of our information society

IN AUGURAL LECTURE BY PROF. DR. IR. JOOST VISSE
Our society functions as it does due to software. Travel, transport, production, retail, banking, government, health care, education, and almost every aspect of our lives is facilitated in some way by software systems. In fact, software is the DNA of our modern information society. But software is different from DNA in one very important aspect: its creation and continuous evolution are the work of people. Many organizations are struggling with the software assets that they create and maintain. This manifests itself in bugs, outages, excessive and unpredictable costs, delayed releases, and frustrated users. Why is it that something we make and adapt ourselves slips from our fingers and takes on a life of its own? What makes software hard to control?

In this inaugural address, Joost Visser reflects on the nature of software and the fundamental challenges that we face when software becomes large and complex. He will discuss the need to develop software measurement models that allow software engineers and laymen alike to perform direct observation of the relevant properties of software-intensive systems. Can software measurement enable us to master the DNA of our information society?

Joost Visser (1971) studied computer science at Delft University of Technology and philosophy at Leiden University. He obtained his PhD from the University of Amsterdam for software analysis research carried out at the Center for Mathematics and Informatics (CWI). He is Head of Research at the Software Improvement Group since 2007, and Professor of Large-scale Software Systems at Radboud University Nijmegen since April 2012.
HOW DOES YOUR SOFTWARE MEASURE UP?
MASTERING THE DNA OF OUR INFORMATION SOCIETY
How Does Your Software Measure Up?
Mastering the DNA of our information society

Rede in verkorte vorm uitgesproken bij de aanvaarding van het ambt van bijzonder hoogleraar Large-scale Software Systems aan de Faculteit der Natuurwetenschappen, Wiskunde en Informatica van de Radboud Universiteit Nijmegen op donderdag 6 december 2012

door prof. dr. ir. Joost Visser
Mijnheer de Rector Magnificus,
Zeer gewaardeerde toehoorders,
Ladies and gentlemen,

Thank you for attending this lecture. Today, I would like to reflect with you on the nature of software and the fundamental challenges that we face when software becomes large and complex. I will also share with you the research directions I hope to pursue to face those challenges and the contributions I hope to make to the education of those that will be building and managing large-scale software systems in the future.

I will be speaking to you on these topics not merely from my own experience and accomplishments. I would not even be aware of these topics if Paul Klint and colleagues at cwi had not pointed them out to me 15 years ago. I would not have been able to think clearly about them without the formation I received at Lawrence University, Delft, and Leiden. And were it not for the unique opportunity to work with my colleagues at the Software Improvement Group who analyze and assess hundreds of large-scale software systems on a yearly basis, I had no reason to be here.

RELEVANCE OF SOFTWARE
Many of you have traveled to this location with the help of navigation software or a public transportation planner. You may have paid electronically for your train fare or for fuel at a gas station. You had money in your account because automated salary systems keep track of what is owed to you and when. But they also keep track of what you owe to the tax authorities, to your pension fund, and health insurance. And these taxes and premiums need to flow to maintain the infrastructure over which you traveled and to restore you to health when you are ill.

In short, you are here thanks to software.

Our society functions as it does, due to software. Travel, transport, production, retail, banking, government, health care, education, publication, and in fact every aspect of our lives is facilitated in some way by software systems that process, store, and transmit our information.

Software is in fact the DNA of our modern information society. DNA is the invisible code of life. It is tucked away in our cells, but determines and predetermines how we grow, function, and behave in interactions with our environment.

Software is the invisible code of digital life. Tucked away inside computers, software is the code that ultimately determines what information flows where and what parts move when.

PROBLEMS WITH SOFTWARE
You are here thanks to software, but hopefully you did not notice. Hopefully the navigation software did not send you into a dead-end street. Hopefully your payment
went through in a matter of seconds. Good software performs its functions unnoticed.

Good software does not get in the way of its users to do what they want to do and in this sense good software remains invisible.

The moment that software gets noticed is the moment when it fails. When your car does not start and needs to be towed to the garage for a reboot. When you make online payments but your balance does not decrease. When you reported your change of address, but mail keeps going to the old house. When you go through all the steps of an online form, you press submit, ... and it tells you to start over.

When it fails is when we are reminded that it is software on which we rely.

And software failure is not limited to outages and bugs. Sometimes software fails to meet expectations in other ways. When online booking proves more cumbersome than placing a call. When a purchase order that was submitted electronically must be processed manually. When you are working with an old system with limited functionality because the new system – expected to be released last year – is still not available.

**IMPACT OF SOFTWARE PROBLEMS**

These things happen – regularly. Sometimes things go bad in a big way, and you read about it in the papers. More frequently, things go bad in smaller ways and, without making the headlines, substantial time and money is wasted in inefficiencies and lost opportunities.

We all pay for these failures, inefficiencies, and lost opportunities. We pay in irritation and waiting times. We pay in higher prices of the products and services we buy. We pay in lower yields of our pension and investment funds, due to higher than needed operational costs.

Sometimes we pay with our tax money, when ambitious government automation projects run over their initially budgeted costs and schedules. And sometimes the impact cannot be expressed in Euros, for instance, when an IT system hampers rather than helps police officers to fight crime.

Failures, excessive and unpredictable costs, delayed releases. Many organizations are struggling with their software assets.

**EXPLANATIONS FOR SOFTWARE PROBLEMS**

But why?

Software is different from DNA in one very important aspect: it is man-made. And not only its creation, but also its continuous evolution is the work of people. Our software assets are created and updated by us.

How come something we make and adapt ourselves slips from our fingers and takes on a life of its own? What makes software hard to control?

Many explanations have been offered. Some say that incomplete or ambiguous specifications are the cause for buggy programs. Some say that changes in the requirements
of users are the cause of delays and lack of user satisfaction. Some blame inaccurate estimations of time and effort at the start of a project, leading to over-optimistic planning. A communication gap or even cultural divide between users and developers of software – misalignment of business and it. Programming mistakes that introduce new bugs when making changes to fix old bugs or add new functions. A lack of attention to quality, incomplete testing, hasty design. And the list goes on.

Are these truly explanations or are they merely restatements of the problem itself? Don’t they simply echo the overall problem that we are not in control of the thing we create? Should we not probe deeper? Why are specifications incomplete? Why do requirements change? Why are estimations inaccurate? Why do developers and users miscommunicate? Why do fixes lead to bugs? Why is quality elusive in design, programming, and testing?

Can we identify the root cause?

SOFTWARE IS UNIQUE
What is unique about software compared to other man-made artifacts is that it is purely the product of intellectual effort.

Building a bridge or a car, producing breakfast cereals or a concert all surely require a great deal of thought. But in the end they are about joining the right materials and executing the right steps.

SOFTWARE ENGINEERING IS NOT CIVIL ENGINEERING
To explain how software is different, let me revisit the analogy between software engineering and civil engineering that others have made often before me. Are software systems like buildings? Is creating and maintaining software like constructing or renovating a house?

There are definitely many similarities. I was painfully reminded of this when remodeling our house over the summer: a continuously sliding schedule, unexpected defects, costly rework. But let’s instead take a look at the differences.

In civil engineering, design and construction are separate activities. Design is a predominately intellectual exercise, while the construction is foremost physical activity: putting bricks and mortar together. In software engineering, a continuum exists between design and construction where both are thoroughly intellectual activities without a clear divide.

This difference is reflected in the bricks and mortar of software, meaning functions and interfaces. The bricks of a house are all the same – fully exchangeable due to standardization of size, weight, and other material characteristics. Standardized materials are of the essence for large-scale, high-quality civil engineering. But the building blocks of software are all unique. Each function and each interface is different and individually crafted. If the same function appears multiple times, this is a sign of bad rather than good quality.
SOFTWARE ENGINEERING IS NOT LIKE WRITING A BOOK

Being the product purely of intellectual effort, perhaps software programs are more like books. We talk about writing software, and programs, like books, are protected by copyright.

Indeed, software programming is like writing to some extent. Every word and sentence needs thought. Not only when it is first written, but also when additions and adaptations are made. Writing a text and writing software require continuous thinking and re-thinking.

But, again, the differences are enormous and fundamental. Compared to books, the volume and internal complexity of software systems is overwhelming. The program code of a medium-sized software system may easily fill 3000 pages of A4 paper, but does not form a linear story. Instead, all paragraphs are intricately interlinked by cross-references. Moreover, each software system is dependent on others. For example, an online store depends on a payment system, a database system, an internet browser, several operating systems, and so on. And the bulk of software code is actually created and changed after initial publication [Visser 2012].

THE UNSURVEYABILITY OF SOFTWARE

The overwhelming volume, interconnectedness, and rate of change in a purely intellectual product makes software fundamentally unsurveyable in the sense that we can impossibly take it all in at once. We can only understand one piece at a time. The tapestry as a whole does not fit in our mind.

And this fundamental unsurveyability is the root cause of why we lose control over the artifacts that we create and maintain ourselves.

The lack of surveyability affects users, developers, and IT managers alike. Developers look inside the system and may get a detailed understanding of certain parts of the software. But those parts they have not written or studied recently remain opaque. The larger the system and the faster it evolves, the smaller the portion they can become and remain familiar with.

IT managers observe the activities of designers, developers, and testers through progress reports and status updates. But the products of these activities remain opaque. This gives IT managers a broader, but quite shallow view of the system. The larger the team they manage, the shallower the view of the system.

Users observe the behavior of a system, i.e. what the system does, but not how it does it. To them, the system is a black box. The more incidental their interactions with the system, the more partial their understanding of what it does.

DILEMMA FOR DECISION MAKERS

But, affected the most by the unsurveyability of software are those that need to make investment decisions about software projects and products. These decision makers are
typically experts in domains other than ICT. And the broad organizational scope of their responsibilities often implies that they have many information systems to make decisions about and very limited time for each decision.

This brings us to an interesting dilemma. Those with the greatest power, responsibility, and – let’s not forget – accountability have no means of directly observing the state of the systems about which they need to decide. No means of reconciling conflicting signals they may receive from users, developers, and managers. No means of validating the reports they must depend on. No wonder they do not feel in control.

**Countermeasures**

To counter the lack of surveyability of software, a wide range of solutions has been proposed. Computer scientists have invented abstraction mechanisms, including subroutines, functions, modules, classes, libraries, higher-level languages, domain-specific languages, coordination languages, aspect-oriented languages, model-driven engineering, and modeling languages such as SDL and UML. All of these techniques follow a divide-and-conquer strategy and serve to hide implementation details behind languages or interfaces.

Software engineers introduced structured programming methods, software architectures, object-oriented design methods, and various iterative development methodologies. These methods help to systematize and structure development activities, which in turn enables division of labor between people or organizational units. Integrated environments for design, development, continuous testing, integration, and deployment have increased the span of control and productivity of individual engineers.

On the managerial level, sets of best practices have been defined for managing individual IT projects, IT project portfolios, and IT service activities. Process improvement methods and models have been defined to guide gradual adoption of these practices. These practices help to instill consistent work processes among the numerous people working in IT projects and organizations.

All of these countermeasures have helped to alleviate the unsurveyability of software. At least for those directly involved in its creation and evolution. Abstraction mechanisms help individual workers to limit the area they must survey. Development methods and tools enable them to do the surveying more effectively and efficiently. And various managerial methods coordinate the multitude of people whose forces need to be joined.

But decision makers are not helped by these instruments. The various abstraction mechanisms do not help to decide in which software system to invest and how much. The software engineering methods and tools do not help to understand whether a given system is ready to be taken into production and another one is ready to be taken out. And the adoption of a set of best practices by a supplier of software services does not show that the delivered software products are best in class.

To help decision makers, a different kind of countermeasure is needed.
OBSERVATION INSTRUMENTS FOR LAYMEN

To help decision makers deal with the fundamental unsurveyability of large-scale software, we need to provide them with observation instruments that allow them to get an objective, shared model of reality that allows them to judge the current state, weigh options, and decide about actions to take at the level of entire systems or portfolios of systems.

These decision makers are typically not software engineering experts. They may be investors, CEOs, or ministers. For this reason, the observation instruments should not report their findings in terms of software-engineering concepts. Observation should include translation: from software engineering to laymen’s concepts. This is a difficult challenge.

SOFTWARE MEASUREMENT MODELS

My experience at the Software Improvement Group has led me to believe that it is possible to construct software measurement models that can serve as observation instruments for decision makers. But construction and application of such measurement models must be done with care.

Let me first explain what I mean by a software measurement model.

Let me do that by analogy. You are all familiar with a measurement model that determines whether your car is of sufficient technical quality to drive on the public road. Car inspections combine a series of observations by a mechanic – an expert in the domain of car maintenance – about things that I, as a driver, do not truly understand – engine rests, fuel lines, suspension – into a clear report. For instance: change the oil and you are good to go.

Another measurement model you may be familiar with is the Apgar score. Right after birth, the nurse performs 5 observations on the newborn baby that she combines into the 10-point Apgar score. A score of 7 or more means the baby is fine, while lower scores may indicate the need for medical attention.

Analogously, a software measurement model combines a number of observations on software engineering artifacts or activities into a simple indicator that is meaningful to an observer without software engineering expertise [Kuipers 2010].

SOFTWARE METRICS CRISIS

The construction of such models is difficult, but not because of a lack of software engineering metrics. A wide range of metrics for software artifacts and activities has been defined in the literature and many of these metrics are supported by readily available tools.

Unfortunately, the ample availability of metrics has in practice not led to their widespread, effective use. This situation by itself may be sufficient reason to call into doubt whether software metrics are useful at all [Fenton 1999]. This I call the software
metrics crisis. Either we learn to use software metrics effectively, or we should abandon them altogether.

In order to use software metrics effectively, they must be ordered into measurement models that answer the following questions:

- Which metrics shall be combined?
- How shall individual metric values be aggregated?
- How shall aggregated values be mapped onto meaningful outcomes?
- How shall we act upon the measurement outcomes?

Thus, the challenge lies in selecting the right metrics, in aggregating their values, and in translating the aggregated values to meaningful, actionable outcomes.

A MODEL FOR MEASURING MAINTAINABILITY

In the past years, a model for measuring the maintainability of software applications has been developed and refined by the Software Improvement Group [Heitlager 2007]. I was fortunate to be part of that development effort. To explain the challenges involved in constructing and applying software measurement models, I will use this model as an example.

The SIG maintainability model – at least in its current form – uses 8 software metrics as input. These metrics are calculated from the source code of the software application under evaluation.

For a medium-sized application, calculation of a given metric will lead to several thousands of individual metric values. These metric values are then aggregated into scores on a scale of 1 to 5 stars for software engineering properties such as unit complexity or duplication. These property scores are then mapped to maintainability sub-characteristics as defined by the ISO/IEC 25010 international standard for software product quality [ISO/IEC 25010], and finally a single star rating for maintainability is obtained.

The thresholds in the measurement model have been chosen such that about 5% of the software applications will be deemed highly maintainable and receive a 5-star rating. The 5% least maintainable systems will receive a single star only. And the remaining systems are uniformly distributed over the intermediate star levels [Alves 2010, Alves 2011].

Thus, the star-rating that comes out of the model can be interpreted by decision makers as a simple ranking of software systems on a maintainability scale, where 3-star systems can be regarded as good enough, 4-star systems as very good, and 5-star systems as exceptionally good.

The model is currently used for rating hundreds of software systems on an annual basis. The rankings provide input for risk assessments [Van Deursen 2003], progress
monitoring [Kuipers 2004, Kuipers 2007], and ensuing decisions about acceptance, deployment, renovation, or termination.

Every year, the model is re-calibrated and sometimes refined such that the star ratings remain a faithful representation of distribution of maintainability across the evolving population of software systems [Baggen 2011].

Validation experiments have been conducted on the maintainability model to study its statistical properties, to establish its correlation with desirable economic indicators such as programmer productivity [Bijlsma 2011], and to study its usefulness in the hands of practitioners. Apologies to my colleagues for using you as guinea pigs.

**MORE MODELS ARE NEEDED**

The SIG maintainability model has seen wide adoption and acceptance and has proven to be highly effective. Nevertheless, it only scratches the surface of what is needed in the software industry in terms of measurement instruments.

To start with, the model only addresses maintainability, which is one out of 8 product quality characteristics defined by the ISO/IEC 25010 international standard. Measurement models for the other characteristics are likewise needed.

And this is not just a matter of repeating the same trick. For example, another quality characteristic is security, which we can only naively hope to measure by source code analysis alone. Also it requires a different way of aggregating sub-scores into a final score that reflects that the system as a whole can only be as strong as its weakest link.

Apart from models that measure quality characteristics of individual software systems, there is room for models that take different perspectives. I will list a few.

Rather than the product perspective, we can take the project perspective with a model that observes the plans and actual activities in a software development project and aggregates them into a rating that indicates how much risk this project runs to not deliver a satisfactory software product within budget and within time.

In addition to project risk, a model is needed to measure project value. We developed and applied such a model in the context of the large and high-risk IT projects of our central government. An essential design principle of that model is that its outcome is not a financial value calculation, which would belie the non-financial nature of some benefits of governmental systems. Rather, the outcomes can be used by citizens and members of parliament to form an opinion themselves on the value of a project.

To take the perspective of a portfolio of projects or a portfolio of systems, more is needed than an aggregation of measurement outcomes for individual projects or systems. Such a portfolio model will need to take into account the interactions between systems and projects and will need to strike a delicate balance between accuracy and the cost of measuring potentially hundreds of systems and scores of projects.

An economic perspective would be offered by a model that translates quality measurements into the cost and benefits of quality repairs. The basis for such a model
has been created with a model for technical debt and interest that extends the SIG maintainability model [Nugroho 2011].

These models and others are currently being developed by various colleagues.

**Some Requirements for Software Measurement Models**

Thus, we need a wide range of models to cater for a range of usage scenarios and perspectives. But in all their diversity, these models should all satisfy a number of basic requirements.

Firstly, software measurement models must have outcomes that are intuitively understood by their users. These outcomes must therefore be reported on unit scales that are not tool-specific, but commonly used in daily life. So, no lines of code or function points or bits and bytes, but Euros, days, percentages, or rankings.

Secondly, software measurement models must be based on relevant technical observations. Not on opinions or perceptions, but on facts. Facts about artifacts and activities that anyone can see. This ensures that the measurement is objective and grounded in actual technical reality.

Thirdly, measurement models must be simple, or at least as simple as possible. This means that the number of data items to be collected must be limited to a minimum, the aggregation of input data to outcomes should be traceable, and the presentation of outcomes should be as minimalistic as possible.

Fourthly, measurement models must be actionable. This means that their outcomes, either directly or after tracing them back to underlying observations, allow one to take concrete steps for acting on the outcome.

These requirements of understandable outcomes, relevant inputs, simple translation, and actionability are essential for measurement models to serve as instruments of objective observation of technical reality by non-technical stakeholders.

**Additional Desirable Properties of Software Measurement Models**

Additionally, we can define a number of desirable properties that are worth striving for, as they make our observation instruments more reliable and more usable.

Firstly, the cost of applying a software measurement model should be minimized. Data collection, in particular, may in some cases be fully automatable, but in other cases will require manual effort. Limiting this manual effort is desirable, for instance by trading sophisticated metrics for simple ones. But, of course, such tradeoffs should not sacrifice the relevance of the measurements.

Secondly, measurement models should have wide applicability, rather than being limited to software developed for a specific purpose or using specific programming languages. Wide applicability ensures that with fewer models we can do more and it enables comparisons across technologies and domains.
Thirdly, software measurement models should be validated empirically. This means that through experiments or experience, the models must be determined to have sufficient accuracy in a wide range of situations.

Fourthly, software measurement models should be subject to continuous improvement. Through calibration and refinement they must be kept up to date and in line with the continuously evolving state of the art in information technology.

Finally, measurement models should come with a body of knowledge of how best to apply them. This knowledge can be transmitted in evaluation guidelines, application examples, or patterns of success and failure [Bouwers 2012]. This helps those that use the models to understand their limitations and apply them with maximal effect.

IMPACT

Of course, measuring software does not automatically make it better. Creating software measurement models is therefore a starting point, not an end in itself. They are a starting point for developers, users, managers, and decision makers to take back control over the software systems that have grown to overwhelming size and complexity. Let’s review some of the things that become possible when adequate measurement models are in place.

Measurement models can be used to track the progress of software development, to detect quality issues, or to recognize and counter risks that may occur during or before the start of software projects. Runaway projects can be prevented or stopped. Effort and time can be spent more efficiently when creating or adapting software. This leads to earlier delivery against lower costs. Ultimately this makes users and owners of software more satisfied and more effective.

Measurement models make it possible to objectively distinguish bad software from good software. This is an essential precondition for a mature software industry and a functioning marketplace. Currently, software services are typically rewarded on the basis of effort, rather than results. When quality becomes measurable, suppliers and clients can reach enforceable agreements that set attainable expectations and provide incentives for producing better software.

Measurement models can help to curb the increasing hunger for energy-resources of our IT systems. Hardware consumes energy, because software tells it to. To make software greener, we need to be able to measure its energy footprint [Grosskop 2013, Visser 2011].

Measurement models can provide transparency about IT expenditure and IT risks in large organizations. Such transparency allows executives to optimize costs and manage risks, not on gut feeling, but on the basis of actual, accurate data. This allows them to take true responsibility for the systems on which their organizations rely. Shareholders and supervisory boards would be served with such transparency to enforce accountability.
In the realm of governmental IT, measurement models can offer transparency to citizens and their representatives in parliament about how their tax money is spent on it. This may lead not only to substantial savings, but also to more timely delivery of software systems that make the lives of citizens better.

In short, let’s develop the instruments by which software engineers and laymen alike can perform direct observation of the relevant properties of software-intensive systems. This will allow us to get a grip on this vast tapestry of software that we are weaving.

Software measurement enables us to become and remain the master of this DNA of our society.

CONTRIBUTION TO RESEARCH
I hope to be able to contribute to this development in several ways. With the help of my colleagues in research and practice I hope to continue building measurement models for various aspects of software quality and management. But building these models is not enough. They must be calibrated, continuously refined, and most importantly, they must be validated. But how do we know we are building the right models and that we are building them in the right way? How can we put our models to the test? What experiments can we run to discover the strengths and weaknesses of our measurement instruments, so we can make them increasingly better and learn to use them better? This calls for the articulation of construction principles for software measurement models and of guidelines for experimentation and application.

CONTRIBUTION TO EDUCATION
I also hope to pass on to others some of the insights in software measurement that I have gained. Through lectures and projects, I will share some of the realities of large-scale software systems. Through the Open Code Clinic of SIG [Open Code Clinic], students are already getting acquainted with software measurement, as their own code is held against the criteria of our maintainability model. As they are getting prepared to contribute to the DNA of our information society, they learn to measure it and master it.

ACKNOWLEDGEMENTS
I would like to express my gratitude to several people that have taught and supported me on the way here.

Firstly, I would like to thank the Executive Board of the University as well as the members of the Advisory Appointment Committee for instituting the chair and appointing me as professor. I am honored and I highly appreciate the trust you put in me.

I’d like to thank Paul Klint and Ralf Lämmel. As PhD supervisor and co-supervisor they had a big hand in launching me into a scientific career and in particular, into the area of analyzing large-scale software systems.
Tobias Kuipers and Marjo Wildvank managed to pull me off the academic path into industry. Not once, but twice. They are exceptionally inspiring colleagues with great vision and perseverance.

Throughout the years I have worked together on articles and innovation projects with a large number of smart and dedicated people. Students, co-authors, colleagues at sig, Universidade do Minho, and cwi. I am sure that without their ideas and insights then, I would have had none to speak of now.

Finally, to my family, who have tolerated my preoccupations and absences: thanks.

*Ik heb gezegd.*
REFERENCES
