Challenges in Software Architecting

Prof.dr. Michel Chaudron
March 24, 2023

INAUGURAL LECTURE

Challenges in Software Architecting

TU/e
EINDHOVEN UNIVERSITY OF TECHNOLOGY

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE

PROF.DR. MICHEL CHAUDRON

Challenges in Software Architecting

Presented on March 24, 2023
at Eindhoven University of Technology

1. Introduction

In this inaugural lecture I will explain the topic of my research: Software Architecture. I will explain the challenges that arise when doing software architecting in practice, the challenges in researching software architecting, and describe how my recent and future research relates to this.

1.1. WHAT IS SOFTWARE?

Software is a structured collection of instructions that direct hardware-processors on which actions to perform. Software is built in levels: the lowest level contains very simple instructions that a processor can immediately perform (such as addition and multiplication). More advanced software is built by building layers of higher levels of abstractions on lower levels. In this way, the internet builds on the layers of network protocols, operating systems and user-interaction-libraries.

Today, software is ubiquitous. It is embedded in many devices, such as mobile phones, cars, pace-makers, microscopes, microwave-ovens, solar-panels, and much much more. Altogether, software is indispensable to the running of companies, governments and society.

The increasing capacities of hardware-processors have enabled them to store and execute increasingly larger programs. For example, the software in a DVD player contains around 5 million lines of instructions. The software in a Fighter Jet contains around 25 million lines of code. The software in YouTube contains around 80 million lines of code. The software in a modern consumer car has around 100 million lines of code each with unique instructions. Imagine that you use A4 paper to print all these instructions of the software of a car. The height of that stack would be as tall as the length of four football-fields.

Researchers that have looked empirically at the evolution of software over time (notably M. Lehman et.al. [6]) have formulated a number of ‘laws’ related to this. Paraphrased (and slightly simplified) they state that:

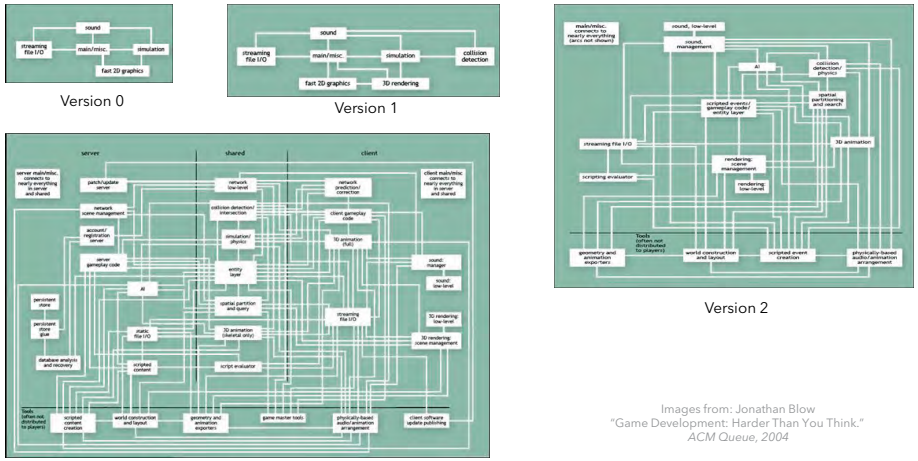
- **Software Changes Continuously**: As society, business, or technology changes, so must the underlying software.
- **Software Always Grows**: The functionality of software system must be continually increased to maintain user satisfaction over its lifetime. Actually, studies show that software tends to grow *exponentially* in size over time.
- **Complexity of Software Increases**: As software evolves, its complexity increases unless work is done to maintain or reduce it.

Figure 1 illustrates these laws. It shows four pictures of the structure of a computer game – each taken around two years apart. They illustrate the growth of the size and complexity of software over a relatively short period of time. From this series of snapshots it is clear that this growth quite quickly leads to many challenges. Indeed, given these continuous growth and continuous changes, one can imagine that the creation, understanding, and maintenance of such an enormous number of parts requires special methods, tool and processes.

The increasing adoption of software together with its continuous growth and maintenance make software development an incredibly large business worldwide. The global software products market grew from 1333 billion USD in 2022 to 1500 billion USD in 2023 at a compound annual growth rate (CAGR) of 12.5%. Also, the number of developers is increasing. According to the last available report from Evans Data Corporation[1], there were 27 million software developers in 2021 – a number that is expected to grow to 29 million in 2024, and 45 million in 2030. In the Netherlands there are around 500.000 people working in IT-jobs. This is more than one in every 20 jobs (data: CBS).

Next, we explain the role that software architecture can play in this.

[1] https://evansdata.com/reports/viewRelease.php?reportID=9, visited March 2023



Images from: Jonathan Blow
"Game Development: Harder Than You Think."
ACM Queue, 2004

Figure 1.1. Increasing Size and Complexity of Software

1.2. WHAT IS SOFTWARE ARCHITECTURE?

Software architecture is the overall organisation of a software system. This involves key decisions about the design and the operation of the software, such as: the (hierarchical) decomposition into subsystems, the relationships between subsystems, the ways in which subsystems interact, and the relationship between the system and its environment. Because software systems are not static, but evolve, there are also aspects of the organisation of software systems that go beyond the implementation of the software, such as: the principles and guidelines that govern the extension and evolution of the software.

In the next section we will explain that software architectures can serve many uses during the development of software, and that a well-designed software architecture can lead to a more successful and more efficient development process.

As a slight aside on the history of software architecture: Some of the foundational ideas that have contributed to the emergence of the notion of software architecture can be traced to Edsger W. Dijkstra, who was a professor in Computer Science at TU Eindhoven. In the 1970's, he defined principles for the design of software in terms of layers and abstractions, as well as principles for the grouping

of functionalities across different components ('separation of concerns'). In the late 1990's academics started to look into this topic, and a seminal book on Software Architecture was published by Mary Shaw and David Garlan [9].

1.3. WHY SOFTWARE ARCHITECTURE? USES AND PURPOSES

Software architectures serve a surprising number of uses in the development of software:

- **Supporting the Conception of Design**: Modeling an architecture is a way of supporting the conception of a design. To illustrate this, we can employ an analogy between the architecting of software and the architecting of buildings. Figure 2 shows four representations of different stages of the development of the building that houses the Guggenheim museum in Bilbao. The top-left is very sketch-like, but still captures initial directions and decisions about the building to be. The top-right, then shows a representation in which these ideas are crystallized further. The bottom-left shows very complete and very detailed diagrams. This level of detail and completeness enables the engineering of parts and the planning of the actual construction. The photo on the bottom-right shows the actual built museum on the banks of the Nervio´n river in the city of Bilbao. The first three diagrams show a progression of the ideas and decisions in creating the design of the architecture. Arriving in the last stage would not be possible without first going through a sketchier ideation and exploration stage. Also, this sequence of stages illustrates the fact that designing such large-scale systems requires both creative, analytic and synthetic skills. Moreover, knowledge of the possible technologies available for implementing a system is also of key influence as these constrain the construction and thus the form of the ultimate system.

    - **Understanding of the Domain**. Performing exploratory reasoning about an architecture is a way of making sense of the functionality that is required of the system. Grouping of functionalities can lead to an understanding of the scope of the system, and can help in identifying missing or superfluous functionalities.

    - **Catalyst for Eliciting Requirements and Rationale**. One key aspect of the process of constructing an architecture, is the need to bringing together of all stakeholders for the system and engage them in sharing their needs and wishes about the system. Architecture can be used for purpose such as the elicitation of requirements and rationale for design decisions and constraints. Diagrams of architectures, in particular, can be catalysts for such discussions amongst stakeholders.

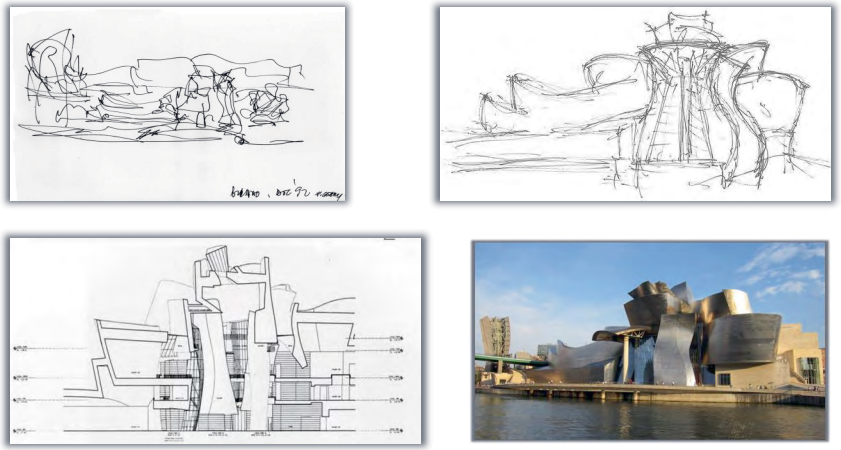


Figure 1.2. Conception/Ideation via Architecture Modeling: Guggenheim museum Bilbao, Architect: Frank Gehry

- **A Common View for Coordination and Communication**.
As stated before, software continuously grows, and as a consequence so does the number of people that are needed to work on maintaining and extending the software. As an example, let's look at a software development organisation in Torslanda in Gothenburg. This is a car manufacturer that is realizing the growing importance of software. They have around 25 teams and each team consists of around 15 persons, hence 375 software engineers at this location of Volvo. And these are also collaborating with affiliated companies – such as Polestar, with whom they share the software architecture, and with several tens of suppliers of software for various subsystems. For such large teams, there is a need to coordinate the continuously increasing and evolving knowledge about the system. More specifically, these uses include the following:

- **Common Knowledge Base**. One obvious use is that architectures provide a common, ideally consistent, source of reference about what the system should be like. Actually, organisations often have a representation of what the system currently looks like ('as is') and a representation of what they would like it to look like in the future ('to be'). Such a standard reference about the design of the system enables *coordination* of the work of large teams of engineers, and enables concurrent development by assigning work to independent teams/developers.
- **Blueprint for Guiding Construction**. Another common use of software architecture is to serve as a *blueprint for the development and maintenance* of software systems. The architecture shows which components need to be constructed, which functionality must be implemented in which component and how components should work together.
- **Basis for Training/On-boarding**. An architecture reflects the key design decision about a system. As such, architectures are an appropriate means to explain what the system is about to newcomers in a project. Given that software tends to continuously grow, there is also a continuous need for the onboarding of new employees. Moreover, the ICT-industry has a significant turnover of staff that move on to other jobs or retire. As an example, according to the NRC, the Dutch tax-office 'Belastingdienst' needs to replace 30% of its IT-personel due to turn-over and retirements[2].
- **Locus of Communication**. Architectures can improve communication and collaboration between team members. For example, an architecture can point out that there are dependencies between components and therefore suggest that the developers working on these different components talk to each other in order to agree on how their components will interact. As such, architecture aids in guiding communication across teams.

- **A Tool for Reasoning and Analysis**. The design of an architecture determines to what degree the implemented system satisfies important quality properties such as performance, security and maintainability. In order to assess that a system will indeed satisfy such requirements, it must be possible to reason about the architecture. For this, architectures need to be represented in such a way that that they are amenable to various types of *formal analyses and reasoning*.

[2] Kabinetsbeleid loopt vast door ict-problemen bij de Belastingdienst, NRC Feb 21st, 2023

- **A Tool for Project Management**. Because architectures show the decomposition of a system into components, they enable the planning, estimation and management of the cost, time and effort for developing a system.

The key point of this section is that architectures can serve many purposes in the development of systems.

Given that there are so many purposes of architecture, maybe there are also many benefits?

In practice, finding empirical evidence for the benefits of architecting has turned out to be very challenging. We will explain some of the reasons why this has turned out to be challenging in Section 4.

2. A Fairy Tale

Since its introduction, various metaphors have been used to explain software architecture. In the next section, I would like to introduce a new metaphor through means of a fairy tale.

2.1. GARDENING WITH KOBOLDS

Once upon a time, there was a king who had a dream of creating a beautiful garden in which fruits would grow that would feed the people in his city. In reality, there had been droughts for several years, and the fruits in his city's garden did not grow.

The king could not sleep from worrying about how he could help his people. At night he would wander through the forest and through the mountains, trying to think of ideas that could help. One night the king stumbled on a small area full of many different types of berries and fruits. There was a kobold collecting the fruits and eating them. The kobold explained that he knew of a secret way to make the grounds fertile and of many wonderful recipes for preparing the fruits. The king asked the kobold for help to make the city's garden fertile. The kobold proposed a pact: I will make your garden fertile if you allow me to live in the garden. The king agreed and made a pact with the kobold.

The kobold did as promised: he turned the garden into fertile grounds and shrubs and trees quickly started growing and producing fruits. The king called in people from his city to help with the harvesting of the fruits. After a fortnight, the garden had doubled in size as the kobold grew new shrubs and trees in all directions. The kobold had called in a few of his family to help fertilize the new areas of the garden. Both the people and the kobolds were happy with the growing garden: they could harvest enough fruits to live from and even sell some on the markets.

All the time, the garden continued growing and growing. The king directed the people to build roads and irrigation channels. The people saw the growing garden as an opportunity: they realized that they could earn more money on the market when they offered more types of products. So, they invented recipes for fruit-pies,

juices, smoothies and more. They used the new areas of the growing garden to plant new types of fruit-plants and they built different types of kitchens around the garden that specialised in making different types of recipes.

As the garden continued to grow, the king realized that it needed organization to manage its ever increasing complexity and to make it possible to continue to grow. He asked the kobolds to dedicate different sections to different types of plants. In this way, it would be easier to navigate the garden and to efficiently harvest and transport the right kinds of fruits to the appropriate kitchens.

The kobolds did not stick to the plans of the king. Some kobolds did not understand the plan, some did not like the plan, and sometimes the plan was not a good fit with the new landscapes into which the garden grew. And new kobolds that had just arrived did not even know of any plan. So some kobolds just mixed plants in sections, created new paths, a few tunnels, removed barriers, and rearranged the layout between sections as they saw fit. All the time more and more kobolds and people were needed to manage the garden.

Figure 2.1. Kobolds picking fruits in the garden

While the king valued the talents of the kobolds in growing fruits, he made drawings of his plans for organizing the garden and wrote down the principles that guided the design ('each section shall produce only one type of fruit', 'every section shall be connected by a path to the relevant kitchen'). As the garden continued to grow, the king needed higher and higher places to see an overview of his garden: the top of a high tree, the highest tower of the castle, and then even the top of a hill. From such height, the details of the garden could no longer be seen.

All the time, the king kept updating his map to be able to instruct the kobolds and people of his city. Moreover, the king installed guards to ensure that the plans and principles were actually followed. In the end, the garden thrived under the careful guidance and wise leadership of the king.

Similarly, in software development, architects and programmers must work together according to a common plan that allows for autonomy and creativity, yet also ensures sustainable growth. By finding this balance, they can create software that is both beautiful and maintainable.

3. Challenges in Applying Software Architecture

Section 1.3 described that software architectures can serve a variety of purposes. However, it turns out that it is very challenging for organizations to effectively perform software architecting practices such that they cater for all their needs. Some of the reasons for this include the following:

1. **A Lack of Skilled Architects and Developers**. Software architects need both excellent soft skills (e.g., listening, presenting, negotiating) as well as a variety of technical skills (analysing, abstracting). One of the distinguishing characteristics of architects is that they can understand the system at all levels of abstraction – from implementation and technology, to architecture, to business domain. Architects typically have broad knowledge, yet also know relevant technical details. Individuals that have all of these qualities are rare.
2. **Diversity of Stakeholders Backgrounds and Disciplines**. Often many stakeholders that are involved in developing a system have not been trained in software architecting. Software architects need to collaborate with a variety of specialists, such as electrical engineers, mechanical engineers, user-interface designers, and with people from the application domain, as well as with project managers that may have limited technical knowledge. As such architects and architecture models must try to bridge all of these different backgrounds and disciplines.
3. **Tailoring of architecting practices to a plurality of purposes**. A plethora of methods and techniques are available for software architecting. Organisations must select which combination of these techniques to use, and how to tailor these techniques to their specific context. Examples of tailoring include: which notation to use (standardized notation or domain-proprietary notation), the level(s) of abstraction to cover, the process of maintaining the architecture (including versioning and how to communicate updates). This tailoring is a further challenge because in any project, software architectures serve a mix of stakeholders and purposes. Moreover, the degree to which architectures serve their purposes changes over the course of a project [1]. Yet, the purposes and uses of architectures are key forces in determining how architectures should be represented and maintained.

4. **Integration of Architecting in the Development Process**. Many organisations are optimistic about the design of a software architecture at the start of a project. Once the initial phases of development have passed – and development teams are typically set up to work on their own components, attention starts to drift to the implementation work and awareness of the architecture starts to fade away. At this stage, the correspondence between architecture and implementation starts to drift apart – making the software architecture less and less useful for day-to-day development. This drift is due to a poor integration of architecting practices in the day-to-day working processes of software projects.
5. **Priority-setting**. Under time-pressure, managers and developers typically prioritise the production of source code, and de-prioritize other practices – such as architecting. Continuous attention to architecting practices, requires management support, disciplined embedding of architecting practices in the development process, and proper priority-setting in the activities of a software development team.
6. **Tools Immaturity**. One particular factor that challenges the adoption of architecting practices is the *immaturity of the tooling for architecting* and the poor integration of this tooling with other software development tools.

    - **Immature Tools for Architecting**. Modern software development is organized in sprints in which developers focus on a user-visible feature, and move this through several activities going from requirements, to architecture/design, implementation, testing and deployment. There is a trend to integrate implementation, testing and deployment in one highly automated toolchain: the DevOps movement. However, the earlier stages, requirements and architecture are not well integrated into this. We lack tools for linking architectures to requirements as well as to source code and to tests. Also, there are established Software Engineering practices, such as quality-assurance and version-management that are commonly applied to source code, but not to architectures. A further challenge is that the implementation and the architecture tend to drift apart over time. However, there are no mature tools that can automatically monitor (or better, maintain) the consistency between architectures and their implementations.

- **Immature tools for Legacy Software**. Many software projects do not start from scratch. Indeed, a complicated landscape of software often exists and needs to be maintained and evolved. For such cases, an architecture needs to be (re-)created for an existing system. For this, some reverse engineering/ architecting tools are proposed that try to reconstruct an architecture from source code. Currently, the results that these reverse engineering tools produce look very different to what is made by human architects. While this topic is not as sexy as AI and blockchains, this is a fundamental problem in software development for which major research efforts are needed.
- **Immature Tools for Distributed Software Sevelopment**. When software grows, so do the organizations that make the software. Such organizations grow across multiple teams and possibly also across multiple geographic locations. In such large distributed organizations, matters are complicated because there is not a single repository that includes all of the source codes that constitutes the system. This complicates creating a single consistent overview of the system.

4. Challenges in Researching Software Architecture

Doing research into software architecture is challenging for a variety of reasons. Some but certainly not all of these reasons include the following:

C1 Architectures are Tied to Context. Software architecture is difficult to study in a university-lab: the practices of software architecting are very much driven by the context in which they are applied. For example, a game for a mobile phone, a banking system, or a medical imaging machine all have very different software architectures. These systems differ in their types of users and in the key forces that drive their architectural design. Moreover, they have different technological ingredients that change at different paces and different organizations that operate under different business models and legal constraints. In addition, the use of architecture makes more sense for larger systems. Larger systems have large numbers of developers working on these systems over longer periods of time. Essentially, there are so many factors that link architectures to context, that recreating this in a lab would either be an oversimplification or extremely expensive.
Software architecting shares these characteristics with disciplines that study people in organisational- and social settings.

C2 Lack of Standardization. There are no standard ways of doing software architecting:

- *Lack of standard representation*. There is no single standard way of documenting or representing software architectures. There does exist a *de facto* standard for representing software: the Unified Modeling Language (UML) [8]. But, like all languages, this language can be used in many ways to describe software systems. Instead there is a zoo of methods, views and notations that projects can chose from. In practice, there are many ways that projects tailor these to fit their own needs and expertise. In representing architecture, projects highlight what is important to their project and their audience – and this tends to differ from project to project.

- *Lack of Comparable Assessments*. There is no universal way of measuring the essential properties of software architectures. This start with the observation that in different projects, architectures serve different purposes, and that the quality of an architecture should be understood relative to its purposes. This also relates to challenge C1. Next, one could look at common quality properties in which stakeholders are interested, such as maintainability/future-proofness and comprehensibility. Comprehensibility turns out to be quite subjective; it depends on the experience, skill and familiarity of developers with the systems and the patterns and technologies used in its construction. The state-of-the-art methods for assessing maintainability properties involve some types of prediction about likely future changes In summary, because architectures are closely tied to their context, so are the methods of assessing architecture. We therefore lack objective methods of assessment of architectures that enable easy comparison across projects.

This lack of standards makes it challenging to compare and contrast architectures for different projects. Also, the diversity of approaches makes it challenging to develop generic architecting tools that could be useful in diverse organizations.

C3 Lack of Data. Recently, we have witnessed an acceleration of AI research. One of the key fuels for this acceleration has been the enormous growth of data that has become available for training AI systems. For research on software architecture, there is very limited data available. Architectures are almost never shared, because organizations consider their contents as sensitive for their performance as a business. Architectures are therefore mostly treated as *company confidential* information and are not shared with researchers. Moreover, studying impact on the quality of the implemented system and the effectiveness of its development would require many complementary types of information about the project context in which they are used, such as e.g. the skill-levels of the developers, their experience with the system, the tools used, the way of testing the software, the way of prioritising new features over maintenance and refactoring, and much more. This challenge is linked to challenge C1: architectures are tightly linked to many contextual factors.

C4 Indirect Impact. While the benefits of architecting are manifold (better communication and knowledge sharing, improved maintainability of the software), these benefits are not directly measurable in tangible artefacts. Also, these benefits are not guaranteed to manifest: the design of the architecture can be very good, but the programmers can make many decisions that lead to a poor implementation. For example, programmers may implement functionality in an inefficient manner, or in a manner that is difficult to maintain, or they may by-pass the delineations of the architecture and thereby increase complexity and decrease maintainability. As an analogy: a good foundation (or plan), does not guarantee a good building (or execution).

Given the above challenges, the most promising approach to study architecting is in it's 'natural habitat', i.e., inside organizations that develop software. Often, qualitative case studies do the most justice to the highly context-dependent nature of software architecting. While this type of research method is becoming more familiar in the field of software engineering, these methods also run into resistance in the field of computer science where many researchers come from fields that are more grounded in formal mathematical methods. For further advancement of this field, researchers in software architecture need to learn how to perform case study research and in particular how to aggregate, combine and generalise results across case studies.

5. Directions in Researching Software Architecture

In this section, I will describe the research that I have done, am doing, and plan on doing – especially in relation to the challenges mentioned in the preceding sections.

- **Education in Software Architecting and Design**

  The 'personnel' factor is [..] the most important pillar for the growth of tech-companies.[3]

  I would like to develop courses and (online) teaching materials on the topics Software Engineering and Software Architecture. I am actively researching how to best teach software architecting and designing. Some recent and ongoing efforts in this direction include the following: My PhD student Dave Stikkolorum defended his thesis on didactic methods for software design. With my SET-colleague Tom Verhoeff and Claire Stevenson from the Psychological Methods group of the University of Amsterdam, I am now supervising a MSc student who is developing methods for studying if we can test abstracting skills and how this relates to software design. With colleagues at universities in Leiden and Antwerp and at McGill in Montreal we are developing techniques for the automated evaluation of software design models – in such a way that students get constructive feedback while they are in the process of creating a design.

- **Studying the Impact of Architecture**
  A popular way of studying software engineering is through 'mining' software repositories. Such studies focus on finding artefacts in software repositories. In a way, these artefacts form a footprint of the activities that take place in software development projects. However, far from all significant activities that take place in a software development project are visible via this footprint. Indeed, for software architectures, we may be able to see what they look like and when they were made. But one of the key uses of architectures is to share knowledge about the system. Typical in this is that architectures are *made once* but

[3] translated from NRC, Feb 3rd, 2023

used often. Yet this use of architectures cannot be derived from any of the artefacts in project repositories. Hence, in addition to the study of architecture-artefacts, I would like to perform more empirical research on the *use* of architectures. Examples of questions I would like to look into include:

- How are architectures used to share knowledge about software systems? Who uses the architecture for what purpose? We know that programmers use architectures differently to designers and to testers. They probably need different types of information from the architecture for their tasks and navigate the architecture in different ways.
- *Longitudinal studies* How do the uses of architecture and its impact change over time? How do architectures evolve over time? What patterns occur in the evolution of architectures?

As an example of a study in this direction, my PhD student Ana Fernandez performed a year-long in-vivo case study at KLM [2]. Her research showed an intricate network of different actors and artefacts in which actors contribute and consume different pieces of architecture-information for a variety of tasks in software development. Based on this research, we aim to develop methods that can aid organizations in systematically analysing the business case for software architecting and to develop methods to aid organisations in tailoring architecting practices to their context.

- **Better Datasets for Empirical Studies on Software Architecture and Design**
  In 2015 I stared a collaboration with professor Gregorio Robles from Madrid, Spain. He is an expert in large scale mining of software repositories. Repository mining studies have focussed on mining source code and related artefacts. I reached out to Gregorio to ask if he could help in mining software designs – especially in the form of UML diagrams. In a joint 18-month effort with a team of colleagues and students from both Madrid and Gothenburg, we managed to mine GitHub to create a dataset of close to 100,000 UML models from around 20,000 software projects [3]. In 2016, this was the first large scale dataset of models of software designs. In a recent impact analysis we found that more than 35 papers have performed studies using this dataset and another 25 have built on the research methods and techniques that we developed. Still, there are many ways in which we can improve the dataset and make it useful for empirical studies: for example by offering better functionality for searching and selection and richer meta-data.

Moreover, such a dataset has provided to be an important source for various types of AI and machine learning studies, such as for creating 'intelligent modeling and design assistants'.

- **Improving Tools for Software Architecting**
  One of the challenges described in section 3 was that of immature tooling for software architecting. I aim to improve on this by working in the following directions:

  - In a joint project with ThermoFisher we aim to develop methods for *monitoring and maintaining consistency between architecture and implementation*. In this project we aim to uncover patterns in the abstractions that are bidirectional mappings between the implementation and the architecture. We will consider both structural and behavioural views of the system. We have two vacancies for PhD students to start working on this.
  - In the last month, a number of different practising architects have expressed the same need: software developers need support for *seamless navigation across abstraction levels*. Sometimes developers need to look up information in the architecture model and then move to the implementation to perform some change. Currently there are separate tools for finding information in the architecture models and for finding information in the source code. The project with ThermoFisher feeds into this by (re) constructing mappings between implementation and architecture. Yet there is an additional need to visualise the software at multiple abstraction levels so as to easily navigate between them.
  - *Inferring higher-level semantics for software*. Nowadays, tools exist that can analyse the implementation of software: the source code. It is our ambition to infer higher-level design characteristics of components in software designs. Recent examples of such research include: the identification of responsibility-stereotypes [5] and the use of graph-mining for discovering recurring patterns in software designs [7] with Frank Takes and Xavier Rademakers from Leiden University. This sensemaking of higher-level design concepts can aid developers in comprehending software. A long term goal here is to create software that can automatically explain a given software system and answer questions about why this software is designed the way it is.

The above research feeds into the plans for a Software Engineering Lab, and a collaborative research infrastructure for software research:

- **TU Eindhoven Software Engineering Lab**
  At the TU Eindhoven, we are in the process of building a Software Engineering Lab. This lab will be used to develop our research tools, to offer team-work projects to students and to demonstrate projects of the Software Engineering and Technology group to stakeholders. In developing this lab, we are designing it such that it can be one hub in a collaborative research infrastructure and also can connect to other labs at the TU Eindhoven, such as the Digital Twins-lab and the High-Tech-Systems-Lab.

- **Collaborative Research Infrastructure for Software Research**
  There are a large number of academic tools for software research – going well beyond software architecture. By and far, these tools are developed by individual research groups. I believe that there is much value in establishing a joint infrastructure that enables collaboration across research groups. My vision on this is described in more detail in this paper [4]. A joint research infrastructure such as this would enable researchers to better build on the tools developed at other research groups. Moreover, it would enable transparency and the replication of scientific workflows. I envision such research infrastructure as including the following:

  - A variety of tools for extracting information from various artefacts (such as source code, UML models, requirements, execution traces) in software repositories.
  - An integrated knowledge-based representation of software systems. This knowledge-based representation would act as a hub for integrating knowledge from a variety of sources.
  - An online collaborative editor for defining scientific analysis workflows on top of the data.
  - A collection of tools for interactive visualisation and exploration of the results of these workflows.

With my PhD student Satrio Rukmono, I have started working on this vision. And I am happy that several colleagues are collaborating on various parts of this vision: on the topic of information- extraction from source code, I am co-supervising two M.Sc. students together with Prof. Jurgen Vinju of the CWI. On the topic of software visualisation, I have co-supervised some M.Sc.-projects together with Stef van der Elzen of the Visualisation group. To all colleagues in the field, I express my invitation to work on this jointly. I believe that we can achieve more by working together.

6. Concluding words

In this inaugural lecture, I explained the topic of my research: Software Architecture. I explained the challenges that arise when doing software architecting in practice, and the challenges in researching software architecting. Also I described how my recent and future research relates to this.

I enjoy doing research in this field of software engineering and software architecting because it combines many different facets:

- Software architecting builds on mathematical/logical/analytical skills, such as in distilling the essence of a problem, and in modeling a system in a systematic and consistent manner.
- Methods and tools in software engineering need to be aware of people's cognitive skills and processes: the essence of software development is knowledge sharing and knowledge-processing by large teams. We look at the best ways to form software so that it can most easily be understood. We look at the best ways to represent software through diagrams so that they best convey the design of software yet are also easy to create and maintain.
- Software engineering requires creativity and synthetic skills: as a software designer you create new systems out of 'thin air' – and because the laws of physics hardly constrain the design of software, one is limited only by one's imagination. These aspects of software architecting link to the arts and artistic aspects of design.
- Software engineering is a social/team activity: only when developers work together effectively as a team can they solve problems together.

As a Software Engineering community, I believe that we should try to work together and build on each other's expertise and tools. Other disciplines (astronomy, physics, biology) would never have been able to achieve what they have had they not established large national and international consortia to establish research infrastructures. The time is ripe to also do this for software engineering.

7. Acknowledgements

I would like to thank the TU Eindhoven and in particular the Software Engineering group for their confidence in me – especially the former dean Professor Johan Lukkien. The journey here would not have been possible and would certainly not have been as enjoyable without the collaboration with many students and especially my PhD students.I thank you all: Giovanni Russello, Mohammad Mousavi, Christian Lange, Werner Heijstek, Ariadi Nugroho, Ramin Etemaadi, Ana Fernández-Sáez, Bilal Karasneh, Hafeez Osman, Ho Quang Truong, Rodi Jolak, Arif Nurwidyantoro, Dave Stikkolorum, Satrio Rukmono. I would like to thank my own Ph.D.-supervisor: Frans Peters, who made my visit to the Programming Research Group of Oxford University possible as a M.Sc. student and for hiring me as a Ph.D. student. Both have been instrumental to my choice for a career in academia and to my development as an academic.

I would like to thank Dieter Hammer for hiring me into his group and introducing me to software architecture. I thank my colleagues and friends in Gothenburg: Regina, Imed, Eric, Jan-Philip, Miroslaw and Jan. I am grateful for the friends I have made through their willingness to collaborate with me in research and in organizing conferences: Gregorio, Yann-Gaël, Jon, Ivan, Engineer, Xavier, Foutse, Stefan, Rafael, Peter v.d. Putten, Guus, Claire, Niklas, Kenneth, Stefan, Rick, Onur, Carmine, Helena, Andreas. I would like to thank my family and friends. Annemieke, David and Vincent: I am very proud of you.

AI image generation system Midjourney was used for creating the illustrations of the fairy tale.

8. References

1. Michel R. V. Chaudron, Ana Fernandes-Saez, Regina Hebig, Truong Ho-Quang, and Rodi Jolak. Diversity in UML modeling explained: Observations, Classifications and Theorizations (Invited Keynote Lecture). In A Min Tjoa, Ladjel Bellatreche, Stefan Biffl, Jan van Leeuwen, and Jirí Wiedermann, editors, *SOFSEM 2018: Theory and Practice of Computer Science - 44th International Conference on Current Trends in Theory and Practice of Computer Science, Krems, Austria, January 29 - February 2, 2018, Proceedings*, volume 10706 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 2018.
2. Ana M. Fernández-Sáez, Michel R. V. Chaudron, and Marcela Genero. An industrial case study on the use of UML in software maintenance and its perceived benefits and hurdles. *Empirical Software Engineering Journal*, 23:3281–3345, 2018.
3. Regina Hebig, Truong Ho Quang, Michel R.V. Chaudron, Gregorio Robles, and Miguel Angel Fernandez. The quest for open source projects that use UML: mining GitHub. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 173–183, 2016.
4. Truong Ho-Quang, Michel R.V. Chaudron, Gregorio Robles, and Guntur Budi Herwanto. Towards an infrastructure for empirical research into software architecture: Challenges and directions. In *2019 IEEE/ACM 2nd International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 34–41, 2019.
5. Truong Ho-Quang, Arif Nurwidyantoro, Satrio Adi Rukmono, Michel R. V. Chaudron, Fabian Fröding, and Duy Nguyen Ngoc. Role stereotypes in software designs and their evolution. *Journal of Systems and Software*, 189:111296, 2022.
6. M.M. Lehman. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software*, 1:213–221, 1979.
7. Xavyr T. Rademaker, Michel R. V. Chaudron, and Frank W. Takes. Automatic identification of component roles in software design networks. In Luca Maria Aiello, Chantal Cherifi, Hocine Cherifi, Renaud Lambiotte, Pietro Lió, and Luis M. Rocha, editors, *Complex Networks and Their Applications VII*, pages 145–157, Cham, 2019. Springer International Publishing.
8. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, Boston, MA, 2 edition, 2004.
9. Mary Shaw and David Garlan. *Software Architecture: perspectives on an emerging discipline*. Prentice- Hall, Inc., 1996.

Curriculum Vitae

Prof.dr. Michel R.V. Chaudron was appointed full-time professor of Software Engineering at the Department of Mathematics and Computer Science at Eindhoven University of Technology (TU/e) on July 1, 2020.

Michel Chaudron obtained MSc and PhD degrees in Computer Science from Leiden University. After working as a professional system/software engineer with CMG Advanced Technology in The Hague (now part of CGI), he joined the group of Professor Dieter Hammer at Eindhoven University of Technology to work in the field of Software Architecting. He was the principle investigator in several ITEA projects that developed methods and frameworks for dynamic distributed component-based systems for low-resourced systems. He moved to Leiden University (LIACS) to direct the ICT in Business MSc program from 2008 to 2012. In 2012 he became full professor and group leader of the Software Engineering division at the department of Computer Science and Engineering, which is shared between Chalmers and Gothenburg University in Sweden. In 2020 he returned to Eindhoven University of Technology where he now is a professor in the Software Engineering and Technology cluster.

Colophon

Production
Communication Expertise Center

Cover photography
Bart van Overbeeke Photography, Eindhoven

Design
Grefo Prepress, Eindhoven

Digital version:
www.tue.nl/lectures/

Visiting address
Building 1, Auditorium
Groene Loper, Eindhoven
The Netherlands

Navigation
De Zaale, Eindhoven

Postal address
PO Box 513
5600 MB Eindhoven
The Netherlands
Tel. +31 (0)40 247 9111
www.tue.nl/map

TU/e
EINDHOVEN
UNIVERSITY OF
TECHNOLOGY