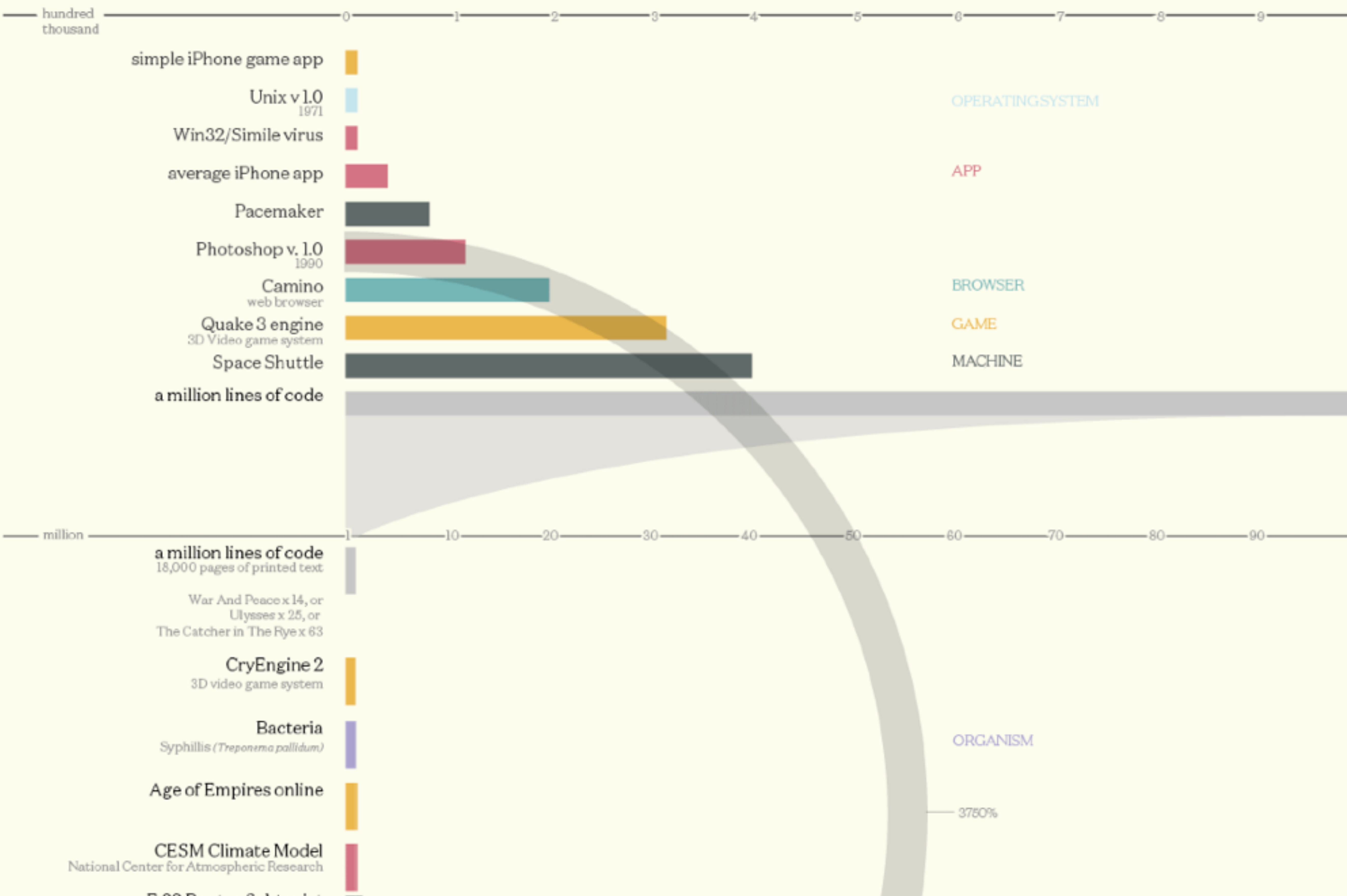
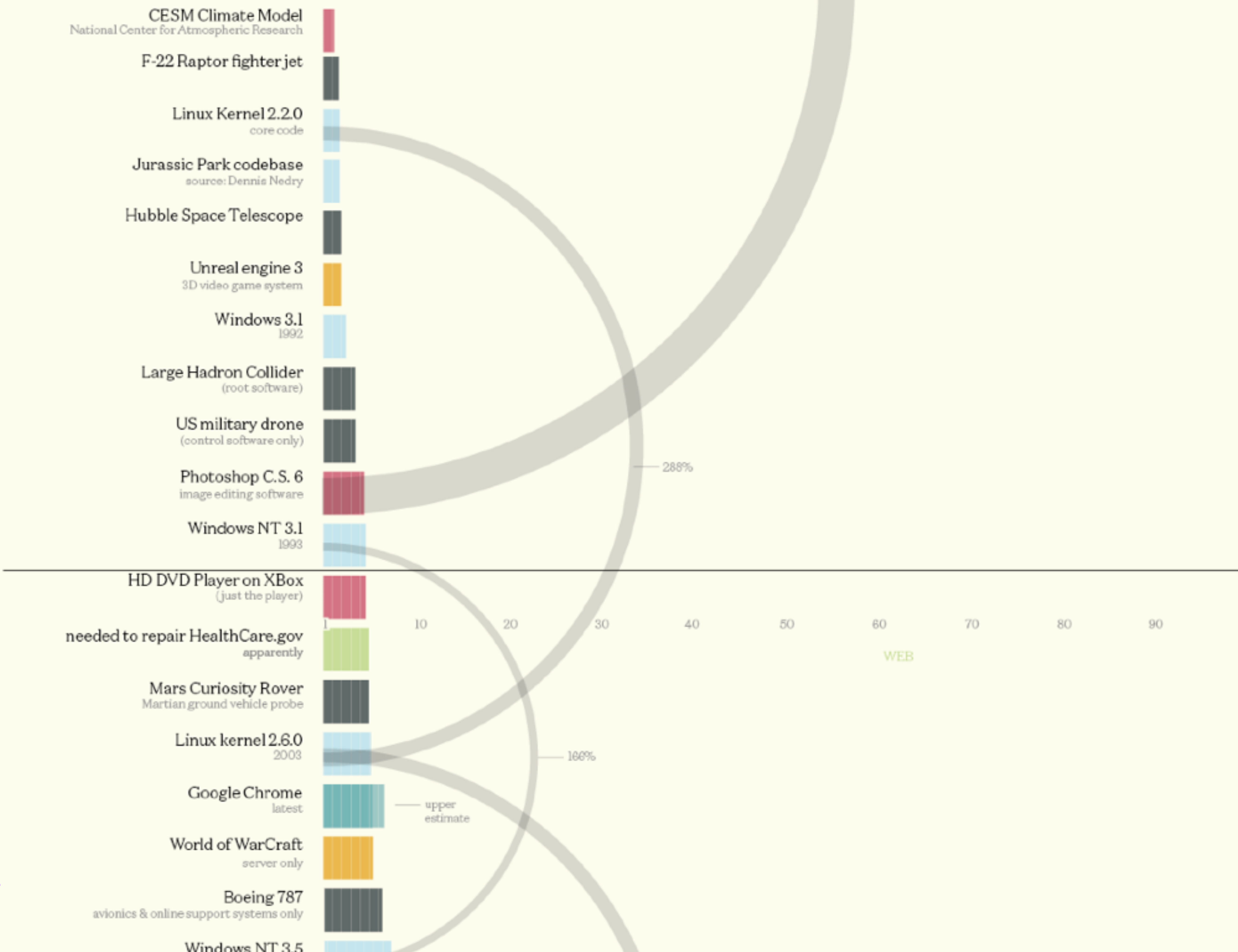
Introduction to Software Evolution

Dr. Vadim Zaytsev aka @grammarware
UvA, MSc SE, 25 October 2015

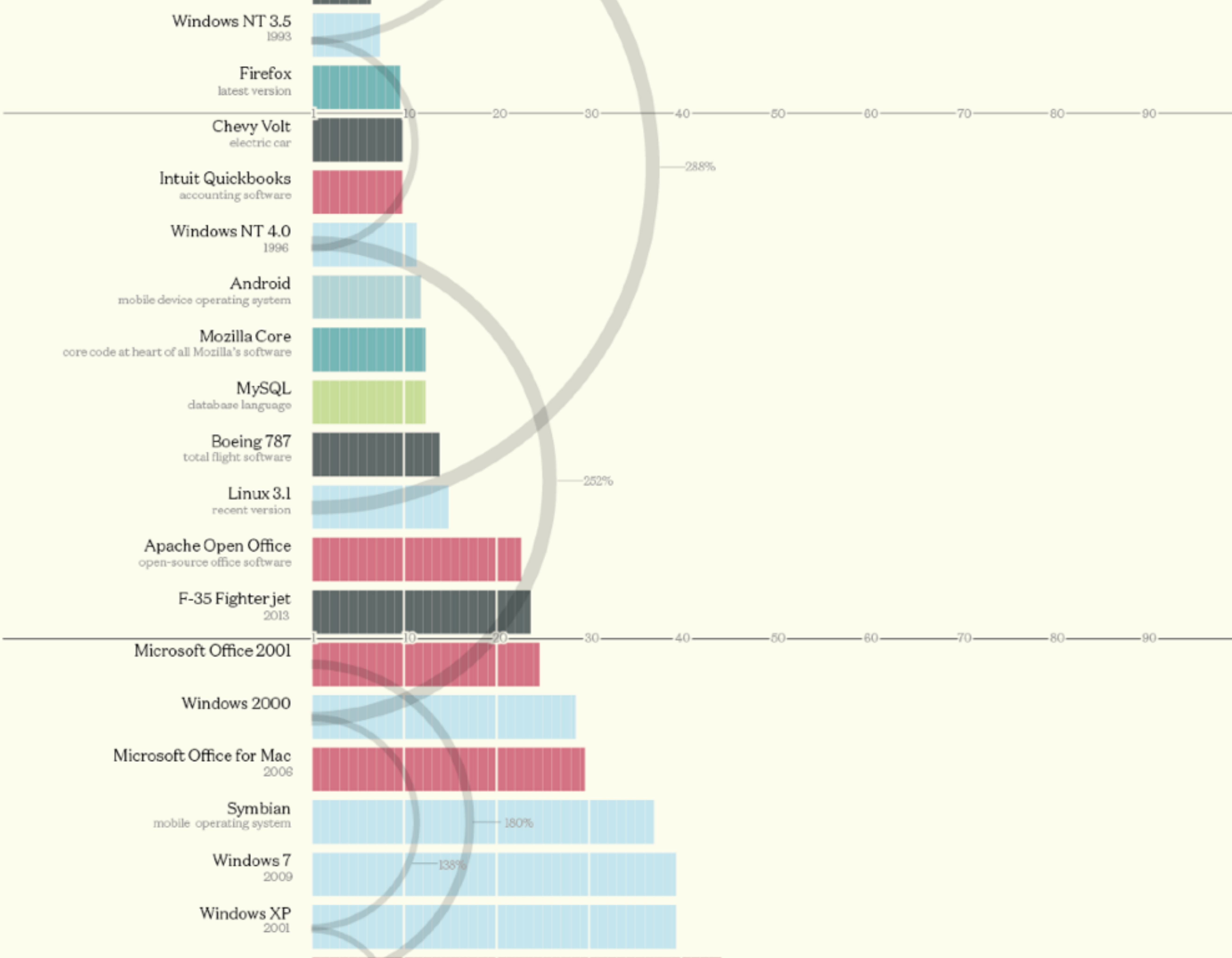
Codebases

Millions of lines of code

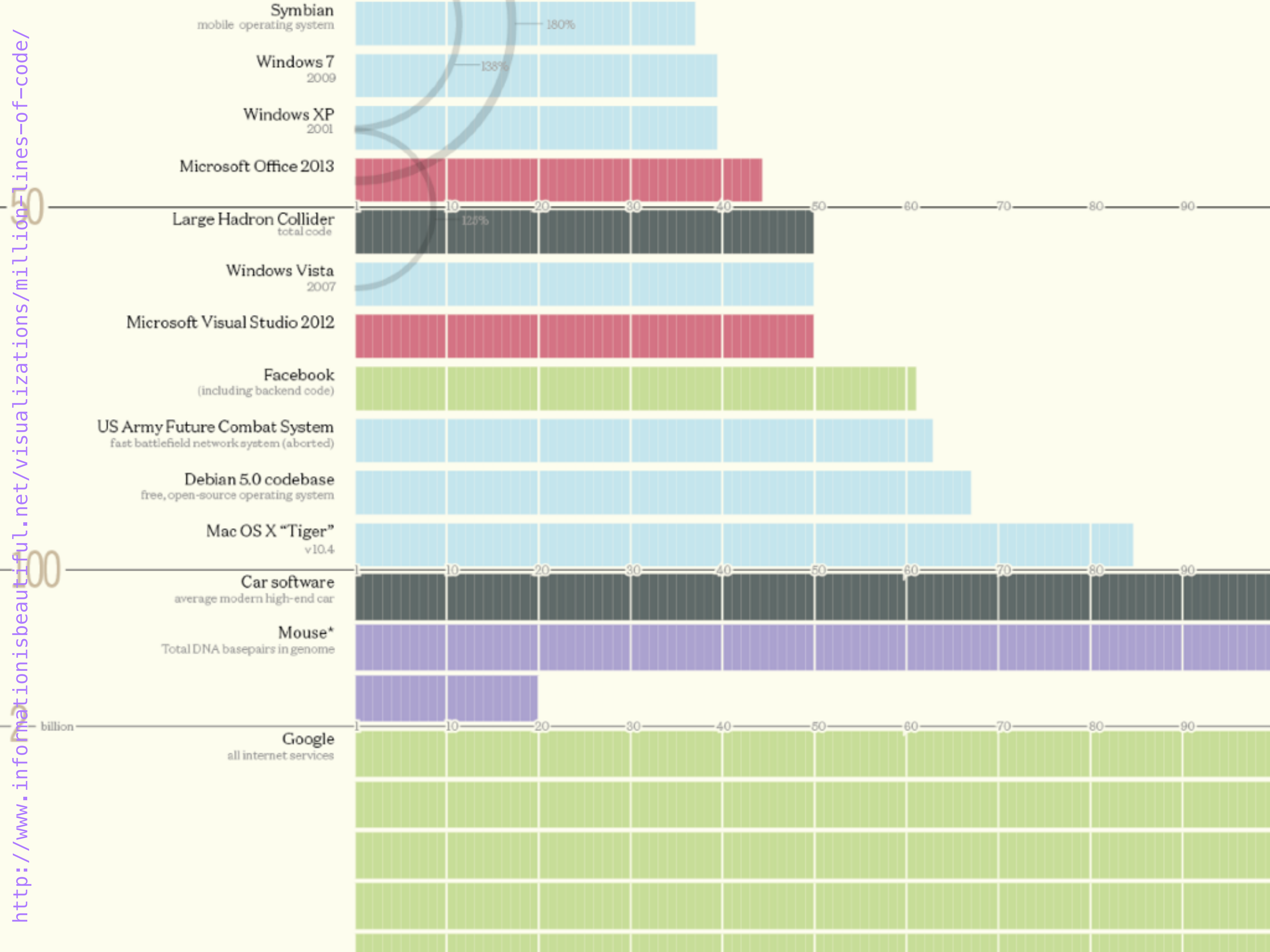




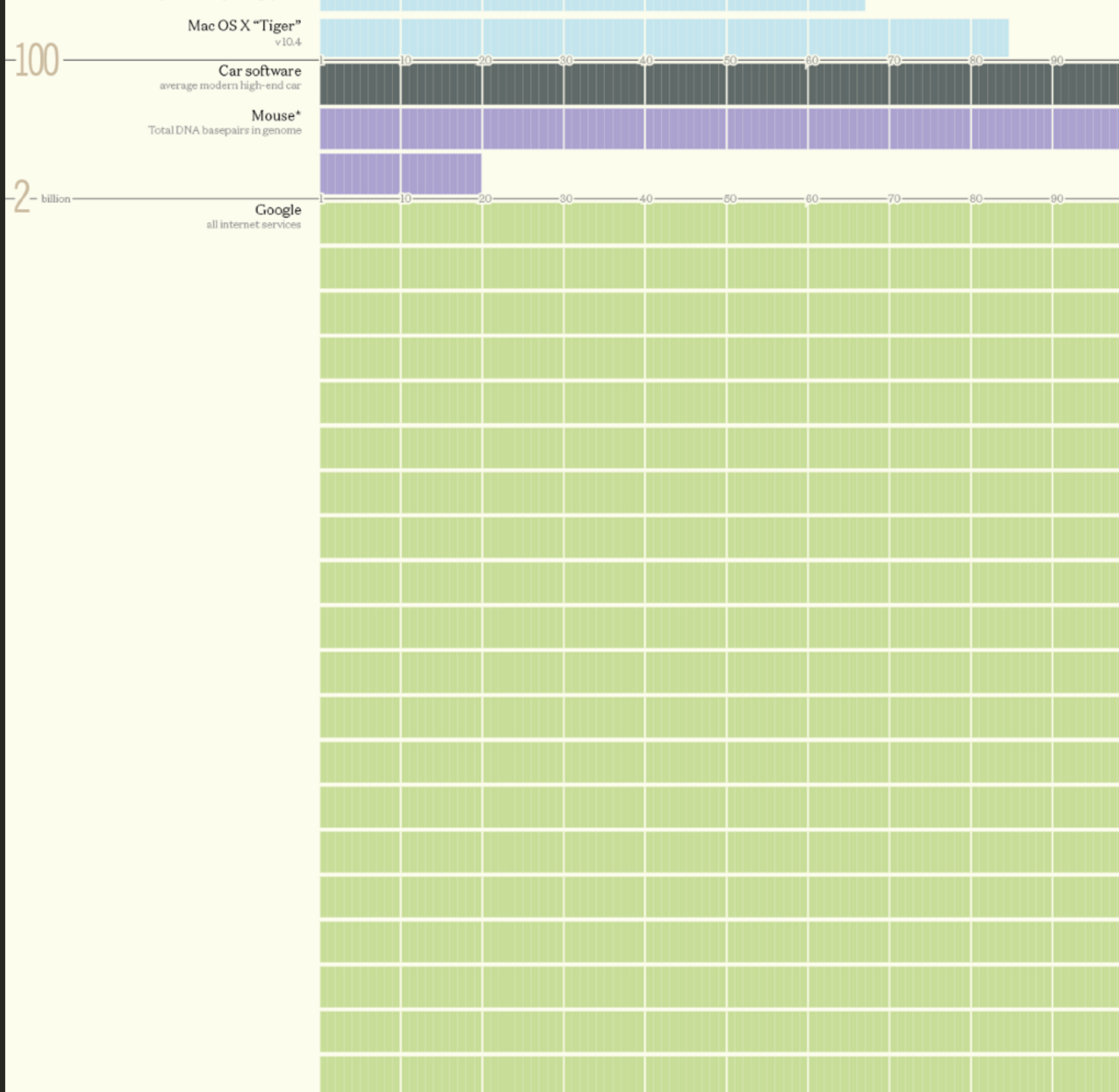
CESM Climate Model
National Center for Atmospheric Research
F-22 Raptor fighter jet
Linux Kernel 2.2.0
core code
Jurassic Park codebase
source: Dennis Nedry
Hubble Space Telescope
Unreal engine 3
3D video game system
Windows 3.1
1992
Large Hadron Collider
(root software)
US military drone
(control software only)
Photoshop C.S. 6
image editing software
Windows NT 3.1
1993
HD DVD Player on XBox
(just the player)
needed to repair HealthCare.gov
apparently
Mars Curiosity Rover
Martian ground vehicle probe
Linux kernel 2.6.0
2003
Google Chrome
latest
upper estimate
World of WarCraft
server only
Boeing 787
avionics & online support systems only
Windows NT 3.5
288%
166%
1
10
20
30
40
50
60
70
80
90
WEB



Windows NT 3.5
1993
Firefox
latest version
10
Chevy Volt
electric car
Intuit Quickbooks
accounting software
288%
Windows NT 4.0
1996
Android
mobile device operating system
Mozilla Core
core code at heart of all Mozilla's software
MySQL
database language
Boeing 787
total flight software
252%
Linux 3.1
recent version
Apache Open Office
open-source office software
F-35 Fighter jet
2013
25
Microsoft Office 2001
Windows 2000
Microsoft Office for Mac
2006
Symbian
mobile operating system
180%
Windows 7
2009
138%
Windows XP
2001
1
10
20
30
40
50
60
70
80
90


Symbian
mobile operating system
180%
Windows 7
2009
138%
Windows XP
2001
Microsoft Office 2013
50
Large Hadron Collider
total code
125%
Windows Vista
2007
Microsoft Visual Studio 2012
Facebook
(including backend code)
US Army Future Combat System
fast battlefield network system (aborted)
Debian 5.0 codebase
free, open-source operating system
Mac OS X "Tiger"
v10.4
100
Car software
average modern high-end car
Mouse*
Total DNA basepairs in genome
billion
Google
all internet services
1
10
20
30
40
50
60
70
80
90

http://www.informationisbeautiful.net/visualizations/million-lines-of-code/



Schedule

W44	Introduction	V.Zaytsev
W45	Metaprogramming	J.Vinju
W46	Reverse Engineering	V.Zaytsev
W47	Software Analytics	M.Bruntink
W48	Clone Management	M.Bruntink
W49	Source Code Manipulation	V.Zaytsev
W50	Legacy and Renovation	TBA
W51	Conclusion	V.Zaytsev

Schedule

W44	Introduction	V.Zaytsev
W45	Metaprogramming	J.Vinju
W46	Reverse Engineering	V.Zaytsev
W47	Software Analytics	M.Bruntink
W48	Clone Management	M.Bruntink
W49	Source Code Manipulation	V.Zaytsev
W50	Legacy and Renovation	TBA
W51	Conclusion	V.Zaytsev

Schedule

W44	Introducti	V.Zaytsev
W45	Metaprogramm	J.Vinju
W	neering	V.Zaytsev
W	lytics	M.Bruntink
W48	Clone Manage	
W49	Source Code Manipulation	V.Zaytsev
W50	Legacy and Renovat	
W51		V.Zaytsev

Series 1:
implement a set of metrics

Series 2:
write a clone detector

Review a paper

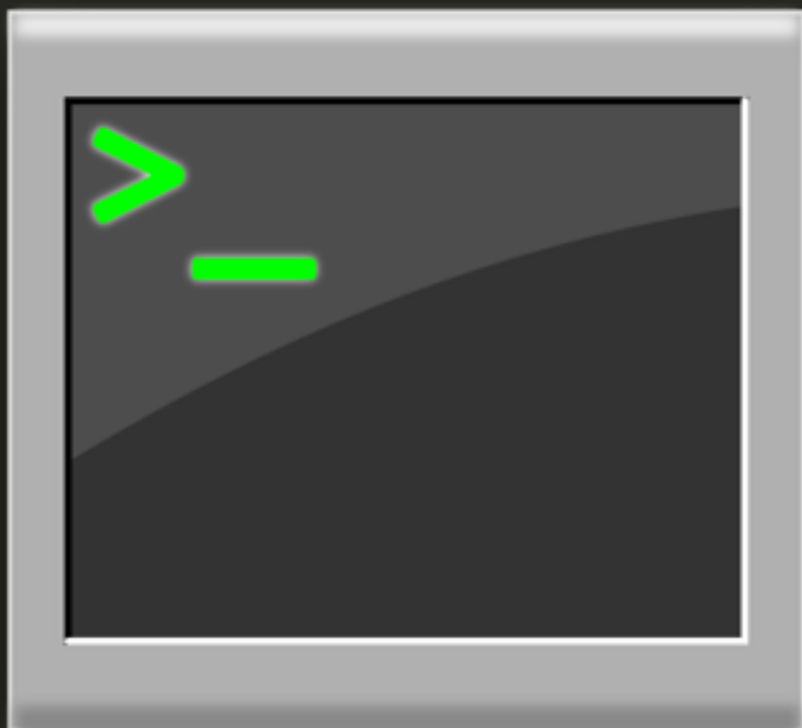
Honours Track

Deadlines & Deliverables

* 2 Nov: Series 0 (Rascal test)
* 17 Nov: Series 1 = ⅓ grade
* 1 Dec: Review = ⅓ grade
* 15 Dec: Series 2 = ⅓ grade

Teachers

Dr. Vadim Zaytsev



Dr. Magiel Bruntink

Prof.Dr. Jurgen Vinju

Davy Landman

Jouke Stoel

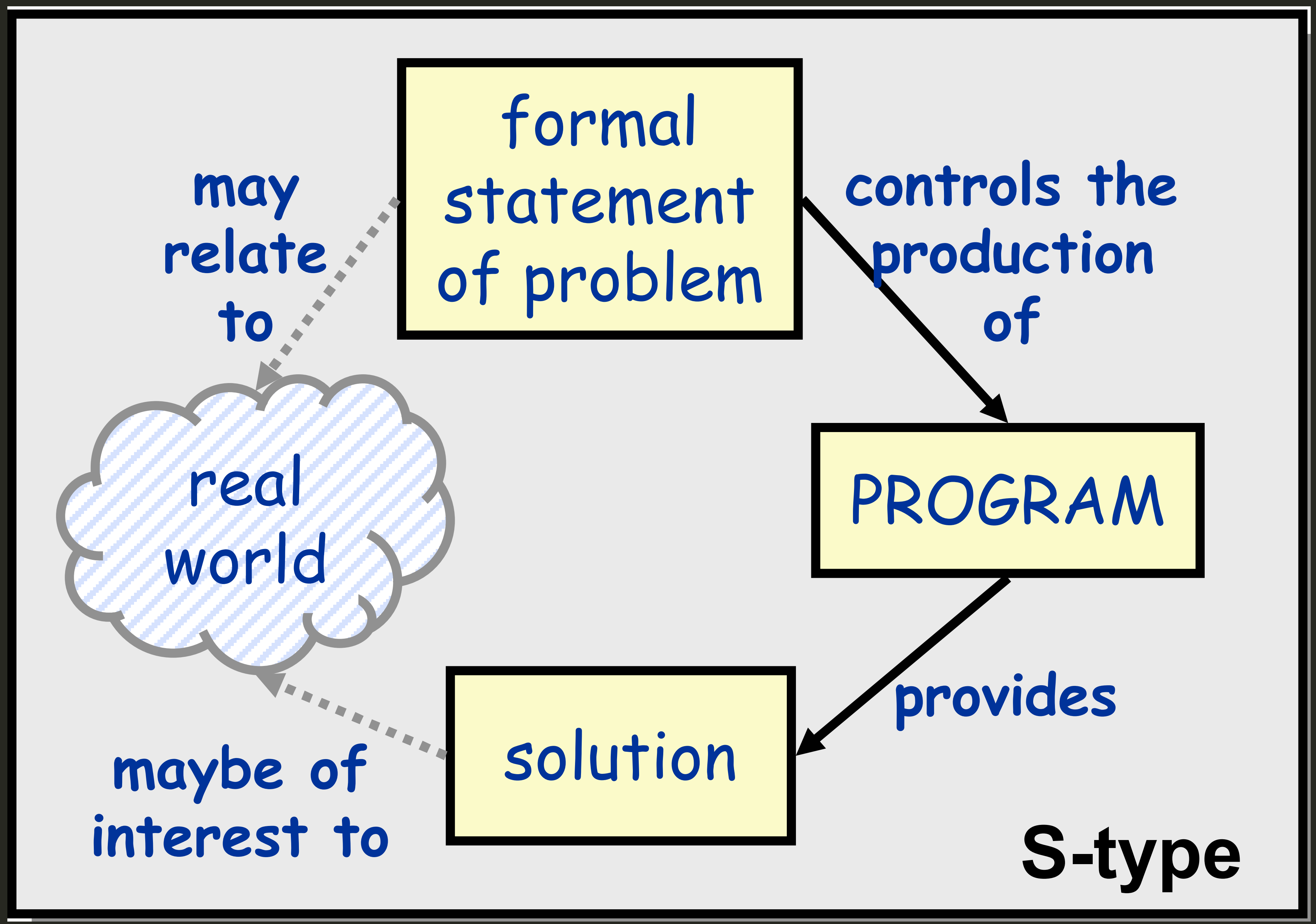
Software Types

Program Types: S

* S–type programs
  * “specifiable”
  * problem formally defined by a spec
  * automated acceptance possible
  * such software does not evolve

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Program Types: S



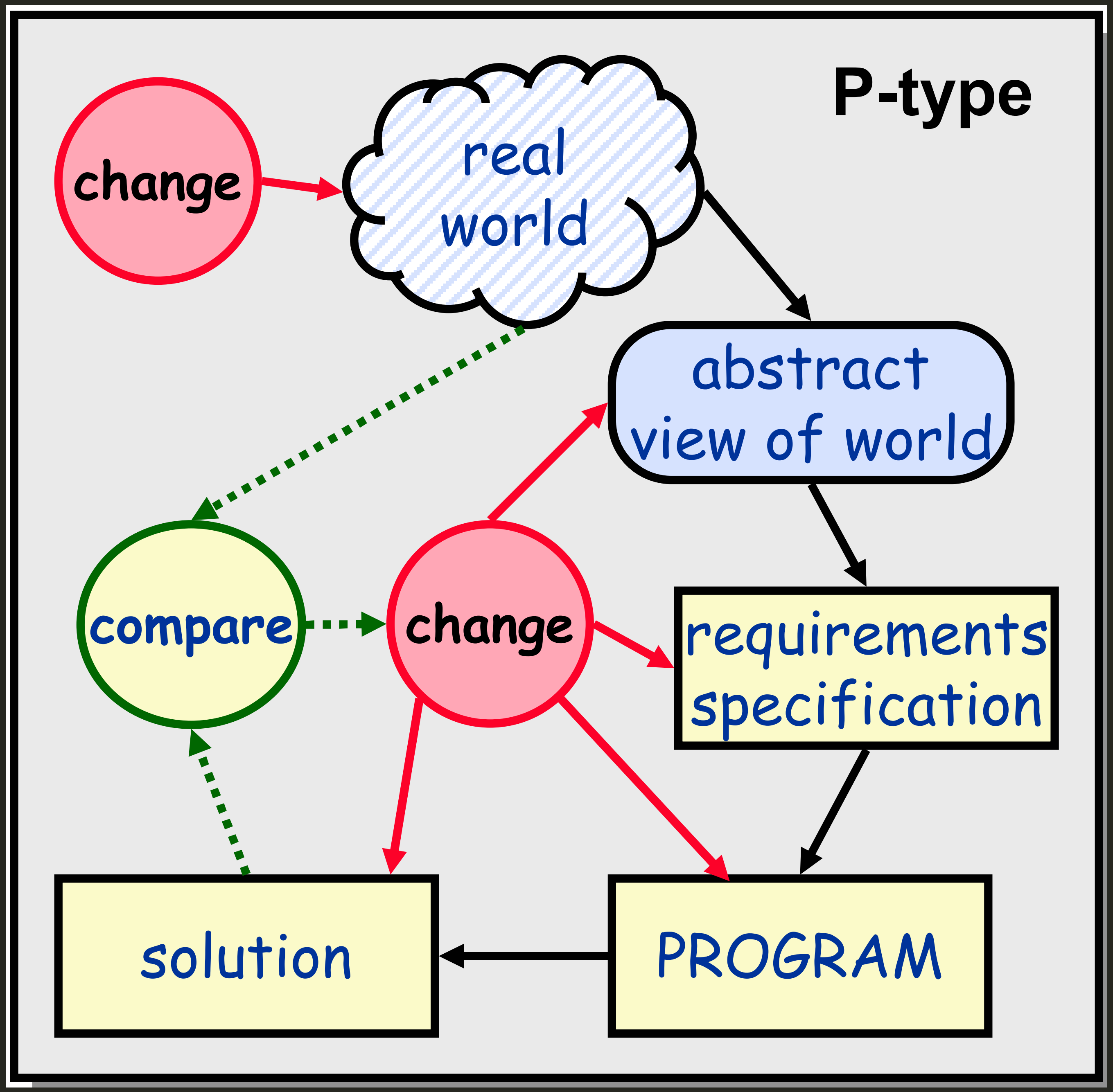
Steve Easterbrook, http://www.cs.toronto.edu/~sme/CSC444F/slides/L20-SoftwareMaintenance.pdf

Program Types: P

* P-type programs
  * “problem-solving”
  * problem models a real-world task
    * imperfectly
  * qualitative acceptance
  * they can evolve continuously

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Program Types: P



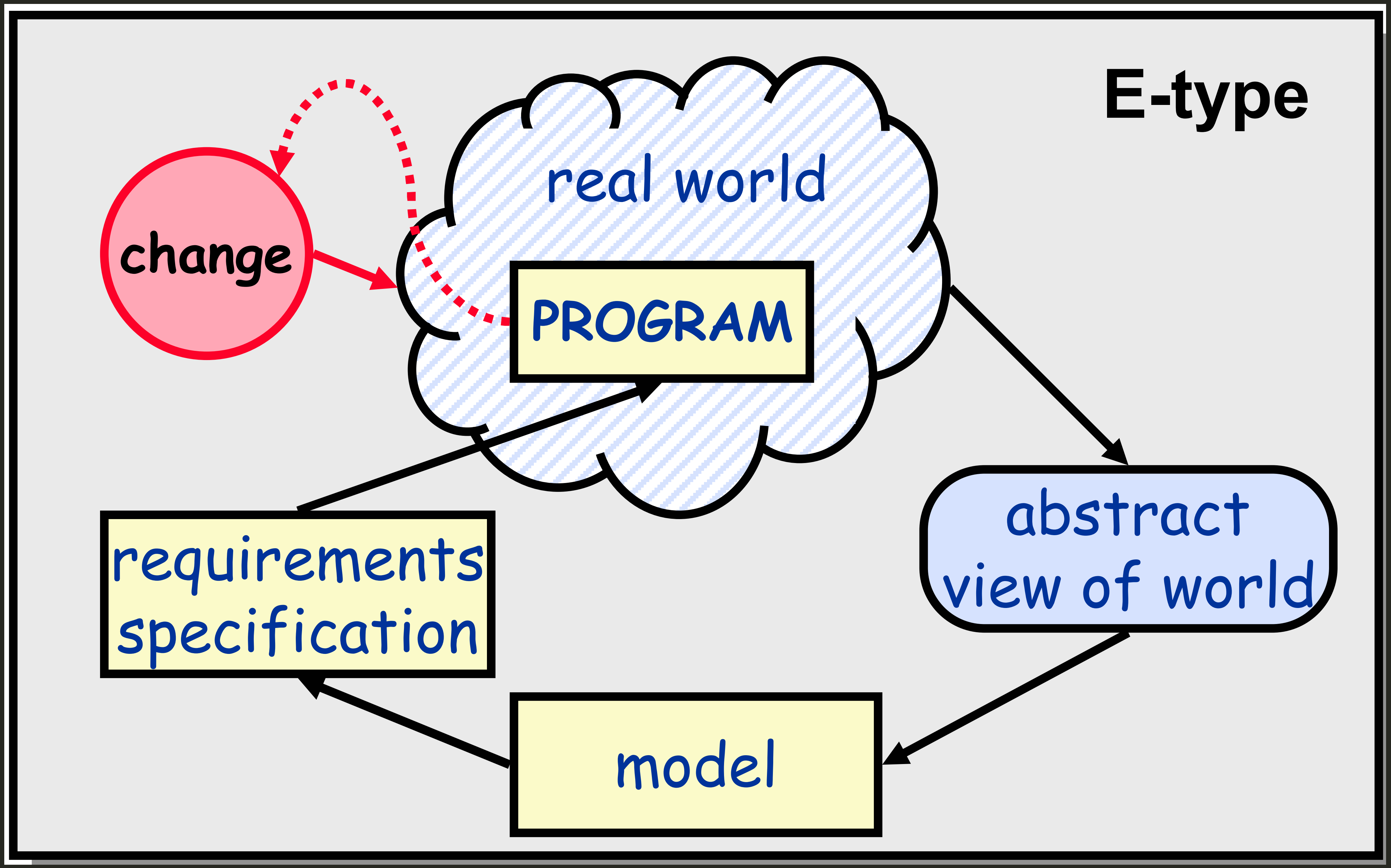
Steve Easterbrook, http://www.cs.toronto.edu/~sme/CSC444F/slides/L20-SoftwareMaintenance.pdf

Program Types: E

* E–type programs
  * “embedded”
  * solution is a part of the world
  * acceptance is subjective
  * they are inherently evolutionary

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Program Types: E



Steve Easterbrook, http://www.cs.toronto.edu/~sme/CSC444F/slides/L20-SoftwareMaintenance.pdf

Lehman’s Laws of Software Evolution

Lehman’s Laws (1/8)

* Continuing Change
  * E-system rots unless adapted
  * the process never stops
  * (true for P-systems as well)

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Lehman’s Laws (2/8)

* Increasing Complexity
  * E-system becomes more complex
  * evolving means complicating
  * (unless we do something)

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Lehman’s Laws (3/8)

* Self-regulation
  * E-system evolution is SRP
  * obeys certain statistical laws
  * (distribution close to normal)

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Lehman's Laws (4/8)

- Conservation of Organisational Stability
  - E-system dev activity is invariant
  - throughout its lifetime
  - (does not depend on resources)

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Lehman's Laws (5/8)

* Conservation of Familiarity
  * E-system changes per release
    * invariant
  * throughout its lifetime
  * (too little: bored;
    too much: overwhelmed)

M.M.Lehman, *Programs, Life Cycles and Laws of Software Evolution*, IEEE 68(9), 1980.

Lehman’s Laws (6/8)

* Continuing Growth
  * E–system must add features over time
  * to keep users satisfied
  * (expectations creep)

M.M.Lehman, J.F.Ramil, P.D.Wernick, D.E.Perry, W.M.Turski,
Metrics and Laws of Software Evolution – The Nineties View, METRICS, 1997.

Lehman’s Laws (7/8)

* Declining Quality
  * E-system perceived quality declines
  * internal as well as external
  * (unless constantly maintained)

M.M.Lehman, J.F.Ramil, P.D.Wernick, D.E.Perry, W.M.Turski,
Metrics and Laws of Software Evolution – The Nineties View, METRICS, 1997.

Lehman's Laws (8/8)

* Feedback System
  * E-system evolution is a
    * feedback system
  * multi-level
  * multi-loop
  * multi-agent

M.M.Lehman, J.F.Ramil, P.D.Wernick, D.E.Perry, W.M.Turski,
Metrics and Laws of Software Evolution – The Nineties View, METRICS, 1997.

Lehman’s Laws

* Continuing Change
* Increasing Complexity
* Self-regulation
* Conservation of Organisational Stability
* Conservation of Familiarity
* Continuing Growth
* Declining Quality
* Feedback System

M.M.Lehman, J.F.Ramil, P.D.Wernick, D.E.Perry, W.M.Turski,
Metrics and Laws of Software Evolution – The Nineties View, METRICS, 1997.

Maintenance Types

Maintenance

* Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment

IEEE 1219, 1993

Maintenance phases

* Introductory
  * user support!
* Growth
  * correcting faults!
* Maturity
  * enhancements!
* Decline
  * technology replacement!

Hans van Vliet, *Software Engineering: Principles and Practice*. Jon Wiley & Sons, 2009.

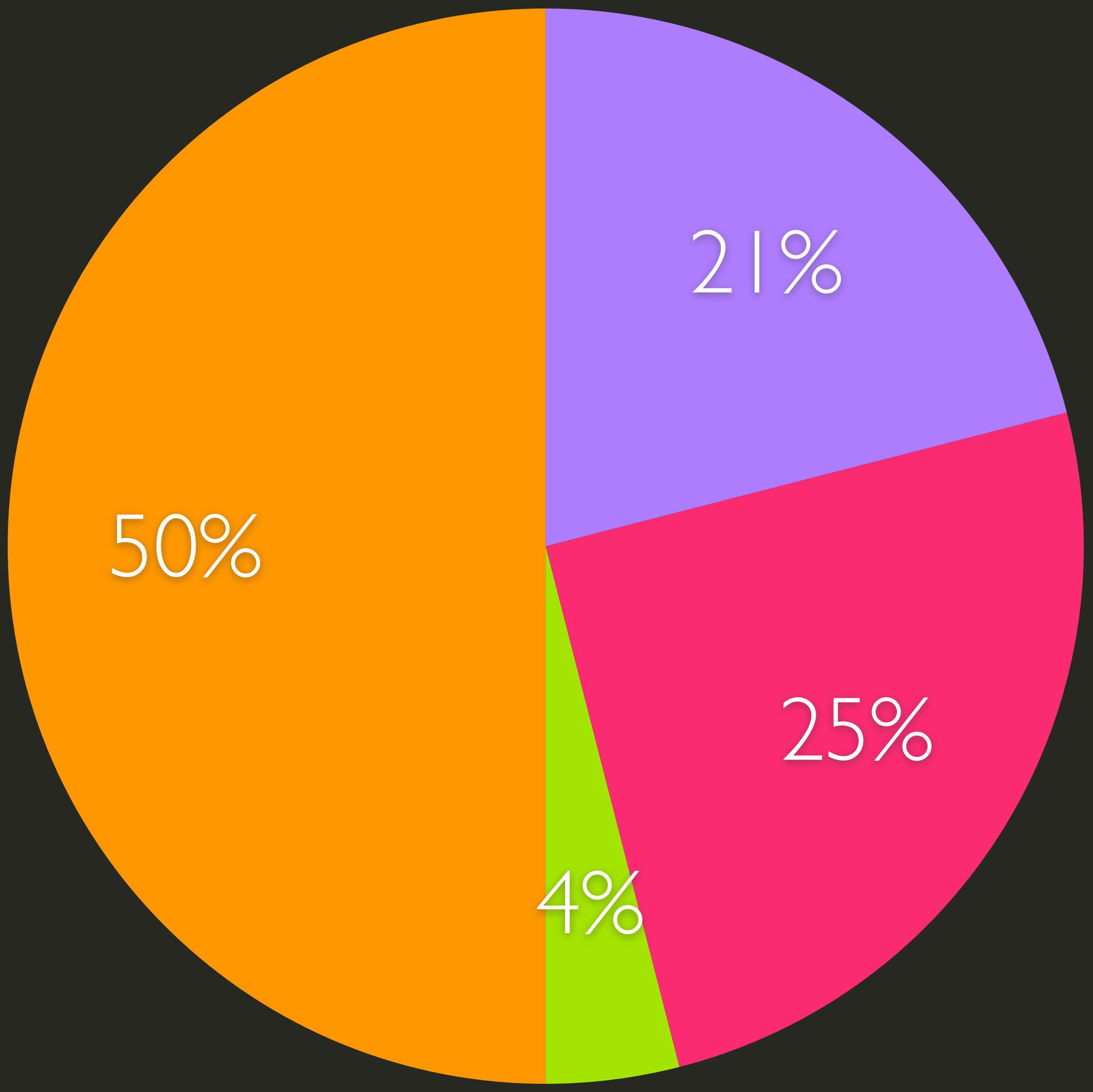
Types of maintenance

* Corrective
* Adaptive
* Perfective
* Preventive

B.P.Lientz, E.B.Swanson, *Software Maintenance Management, A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, 1980.

Types of maintenance

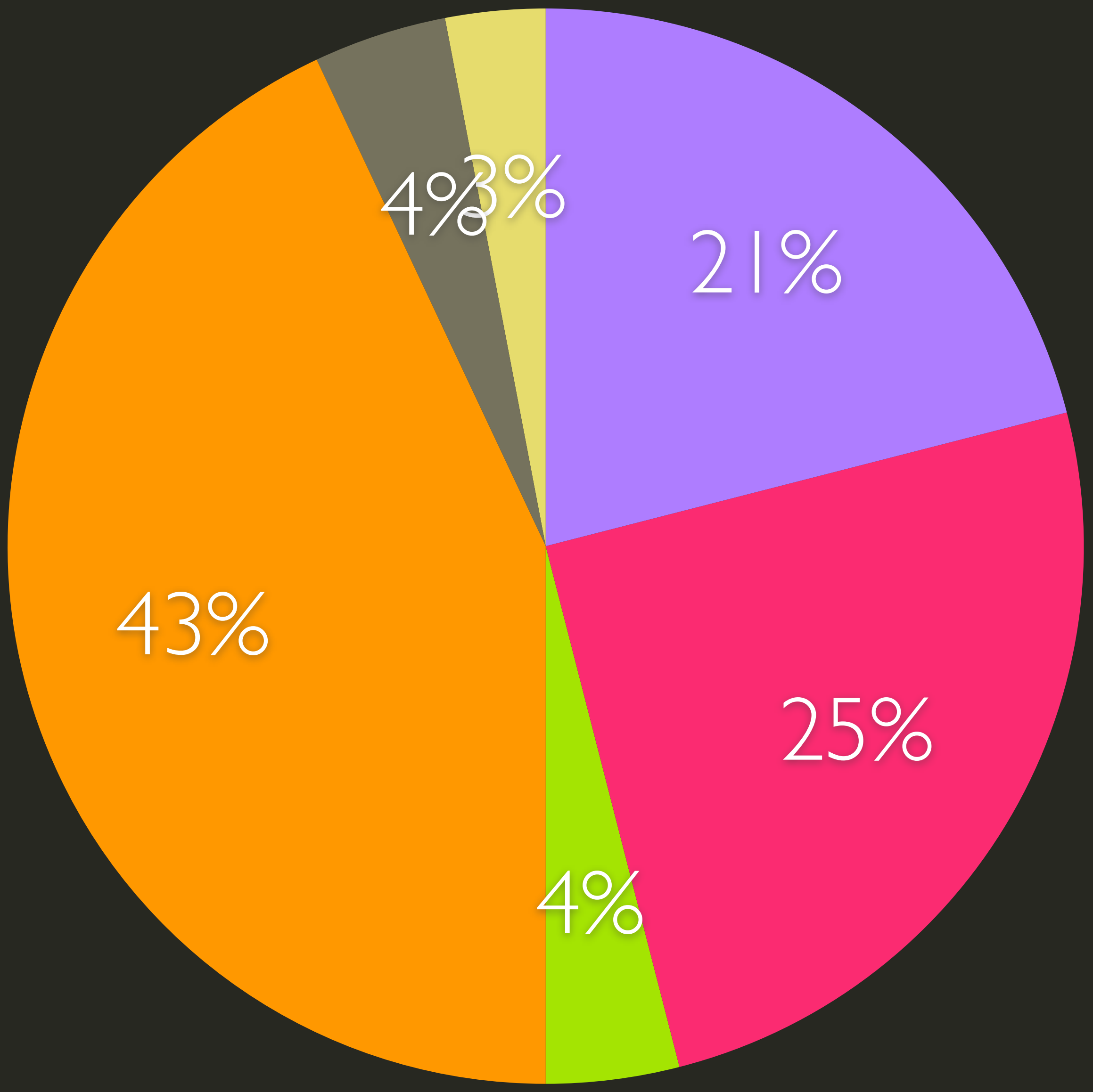
* Corrective
* Adaptive
* Perfective
* Preventive



B.P.Lientz, E.B.Swanson, *Software Maintenance Management, A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, 1980.

Types of maintenance

* Corrective
* Adaptive
* Perfective
  * user enhancement
  * efficiency
  * other
* Preventive



B.P.Lientz, E.B.Swanson, *Software Maintenance Management, A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*, 1980.

Top 5 problems

* Quality of documentation
* User demand for enhancements
* Competing demands for maintainers’ time
* Meeting scheduled commitments
* Turnover in user organisations

S.L.Pfleeger, *Software Engineering: Theory and Practice*, Prentice Hall, 1998.

Is it hopeless?

* Higher quality
  * less (c) maintenance
* Anticipating changes
  * less (a&p) maintenance
* Better tuning to user needs
  * less (p) maintenance
* Less code
  * less (*) maintenance

Maurice ter Beek, http://www.liacs.nl/~mtbeek/se-ma.pdf

Roadmap

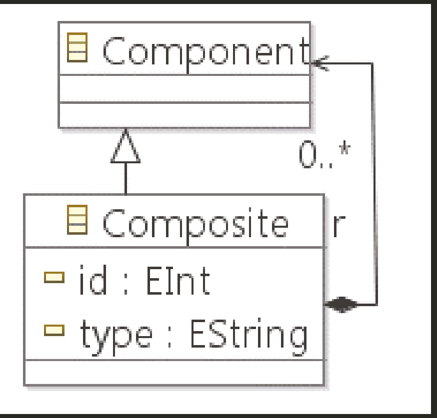
* Metaprogramming
* Reverse engineering
* Software analytics
* Clone management
* Source code manipulation
* Legacy

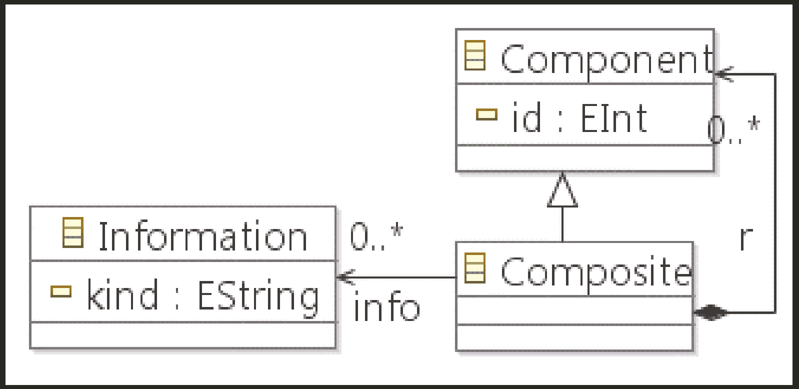
State of the Art

http://bibtex.github.io

Detecting Complex Changes During Metamodel Evolution

* Metamodel evolves:





* Follow user actions
* Detect complex patterns
* Enrich evolution trace

Detecting Complex Changes During Metamodel Evolution

Djamel Eddine Khelladi[1](✉), Regina Hebig[1], Reda Bendraou[1], Jacques Robin[1], and Marie-Pierre Gervais[1,2]

[1] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, F-75005 Paris, France
djamel.khelladi@lip6.fr
[2] Université Paris Ouest Nanterre La Defense, F-92001 Nanterre, France

Abstract. Evolution of metamodels can be represented at the finest grain by the trace of atomic changes: add, delete, and update elements. For many applications, like automatic correction of models when the metamodel evolves, a higher grained trace must be inferred, composed of complex changes, each one aggregating several atomic changes. Complex change detection is a challenging task since multiple sequences of atomic changes may define a single user intention and complex changes may overlap over the atomic change trace. In this paper, we propose a detection engine of complex changes that simultaneously addresses these two challenges of variability and overlap. We introduce three ranking heuristics to help users to decide which overlapping complex changes are likely to be correct. We describe an evaluation of our approach that allow reaching full recall. The precision is improved by our heuristics from 63% and 71% up to 91% and 100% in some cases.

Keywords: Metamodel · Evolution · Complex change · Detection

1 Introduction

In the process of building a domain-specific modeling language (DSML) multiple versions are developed, tried out, and adapted until a stable version is reached. As by one of our industrial partners in the automotive domain, such intermediate versions of the DSML are used in product development, where often further needs are identified. A challenge hereby is that each time the metamodel of the DSML is changed to a next version, already developed models need to be co-evolved too. This is not only the case for DSMLs, but also for more generic metamodels, e.g. the UML officially evolved in the past every two to three years.

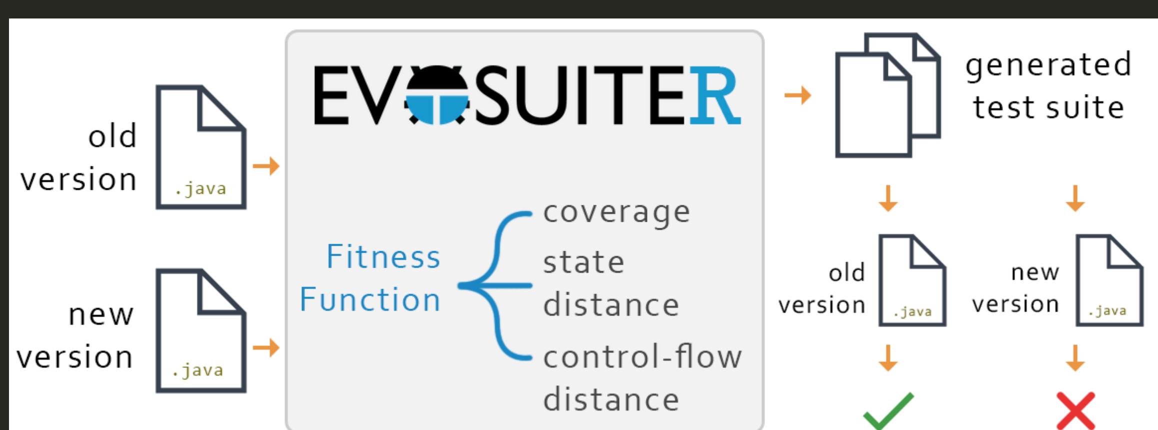
To cope with this evolution of metamodels, mechanisms are developed to co-evolve artifacts, such as models and transformations that may become invalid. A challenging task herein is to detect all the changes that lead a metamodel from a version n to a version $n+1$, called Evolution Trace (ET). Automatically detecting it, not only helps developers to automatically keep track of the metamodels' evolution, but also to trigger and/or to apply automatic actions based on these changes. For instance, models and transformations that are defined based on the metamodel are automatically co-evolved i.e. corrected based on the detected

J. Zdravkovic et al. (Eds.): CAiSE 2015, LNCS 9097, pp. 263–278, 2015.
DOI: 10.1007/978-3-319-19069-3_17

http://bibtex.github.io/CAISE-2015-KhelladiHBRG.html
http://dx.doi.org/10.1007/978-3-319-19069-3_17

Automated Unit Test Generation for Evolving Software

* Software evolves.
* How about test cases?
* Functionality changes?
  * (regression testing)
* Tests that used to work



Automated Unit Test Generation for Evolving Software

Sina Shamshiri
Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello, Sheffield, UK, S1 4DP
sina.shamshiri@sheffield.ac.uk

ABSTRACT

As developers make changes to software programs, they want to ensure that the originally intended functionality of the software has not been affected. As a result, developers write tests and execute them after making changes. However, high quality tests are needed that can reveal unintended bugs, and not all developers have access to such tests. Moreover, since tests are written without the knowledge of future changes, sometimes new tests are needed to exercise such changes. While this problem has been well studied in the literature, the current approaches for automatically generating such tests either only attempt to reach the change and do not aim to propagate the infected state to the output, or may suffer from scalability issues, especially when a large sequence of calls is required for propagation. We propose a search-based approach that aims to automatically generate tests which can reveal functionality changes, given two versions of a program (e.g., pre-change and post-change). Developers can then use these tests to identify unintended functionality changes (i.e., bugs). Initial evaluation results show that our approach can be effective on detecting such changes, but there remain challenges in scaling up test generation and making the tests useful to developers, both of which we aim to overcome.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

Keywords

Automated Unit Test Generation, Genetic Algorithms, Search-Based Testing, Regression Testing

1. INTRODUCTION

Developers evolve software programs by introducing many changes throughout the life-cycle of the software. These changes often range from small refactorings to the addition of large new features. However, some of these changes may affect the originally intended functionality of the software, by introducing unintended bugs – also known as *regression faults*. To avoid regressions in the functionality, engineers write tests as they develop the software, and after making changes developers execute these tests to increase their confidence that the intended functionality of the software is intact. This practice is also referred to as *regression testing* and is commonly used in the industry.

While regression testing can help with early detection of regression faults, developers face several challenges when applying the technique. As the number of tests grows, execution of all tests after every single change can become expensive and impractical. This problem has been well studied in the literature [18] and many techniques such as test selection, prioritization and minimization have been proposed.

The challenges however are not limited to the growing cost of regression testing. Even if all tests are executed, three main problems remain: 1) an existing set of tests is required, 2) the tests are often written without foreseeing future changes, and 3) the effectiveness of the tests in finding regression faults depends on the quality of the written tests. According to the PIE model [15], to reveal a fault, a test has to first execute the fault, infect the state and finally propagate it to the output. While several techniques exist for augmenting existing test suites (e.g., [10, 17]) and generating regression tests (e.g., [2,9,13,14]), the techniques mainly focus on reaching the fault, yet the number of paths to propagate the infected state to the output can explode, which may impose a limit on the scalability of the approach [3].

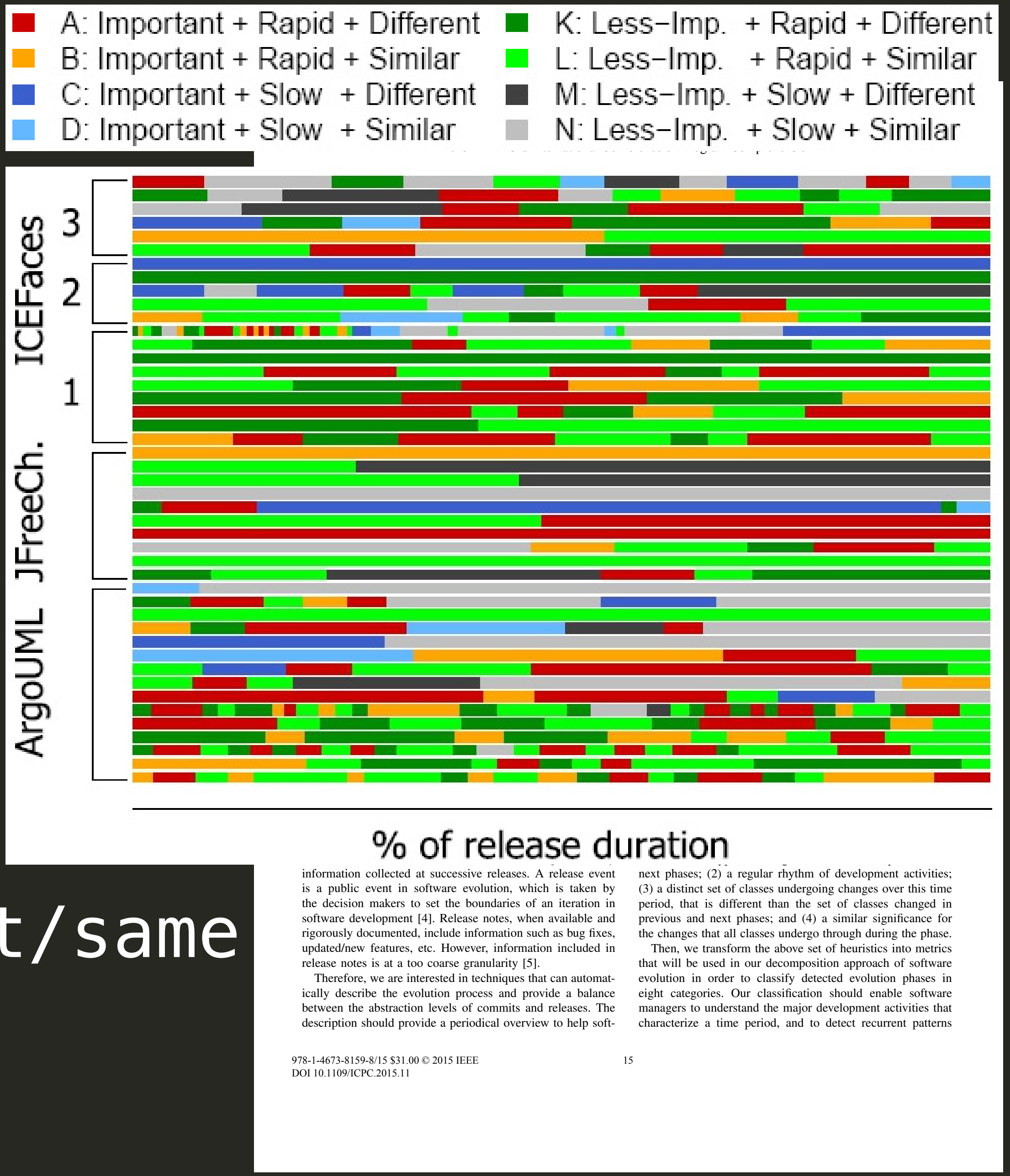
To address the previous shortcomings, we propose a technique for generating a regression test suite (i.e. a set of unit-tests which contain a sequence of calls executing the class under test) without depending on existing tests. Our approach takes two versions of a class under test, and uses a search-based algorithm [8] with the objective of reaching and propagating the changes between the two versions of the program. We have implemented our approach named EvoSuiteR on top of the EvoSuite [5] test generation tool, and our early evaluation of the technique [11] showed encouraging results on examples with propagation issues (i.e. where covering the change alone does not propagate the changed state to the output). Further attempts to evaluate the effectiveness of our approach on detecting real regression faults revealed several challenges. As a part the remaining course of this research we aim to solve these challenges, in addition to evaluating our approach against the state-of-the-art.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2803196

http://bibtex.github.io/ESEC-FSE-2015-Shamshiri.html
http://dx.doi.org/10.1145/2786805.2803196

Detection of Software Evolution Phases based on Development Activities

* Software evol. history:
  * commits – fine-grained
  * releases – coarse
* Something in between?
* 8 kinds of phases:
  * changes: important/not
  * dev: rapid/slow
  * change types: different/same



information collected at successive releases. A release event is a public event in software evolution, which is taken by the decision makers to set the boundaries of an iteration in software development [4]. Release notes, when available and rigorously documented, include information such as bug fixes, updated/new features, etc. However, information included in release notes is at a too coarse granularity [5].

Therefore, we are interested in techniques that can automatically describe the evolution process and provide a balance between the abstraction levels of commits and releases. The description should provide a periodical overview to help soft-

next phases; (2) a regular rhythm of development activities; (3) a distinct set of classes undergoing changes over this time period, that is different than the set of classes changed in previous and next phases; and (4) a similar significance for the changes that all classes undergo through during the phase.

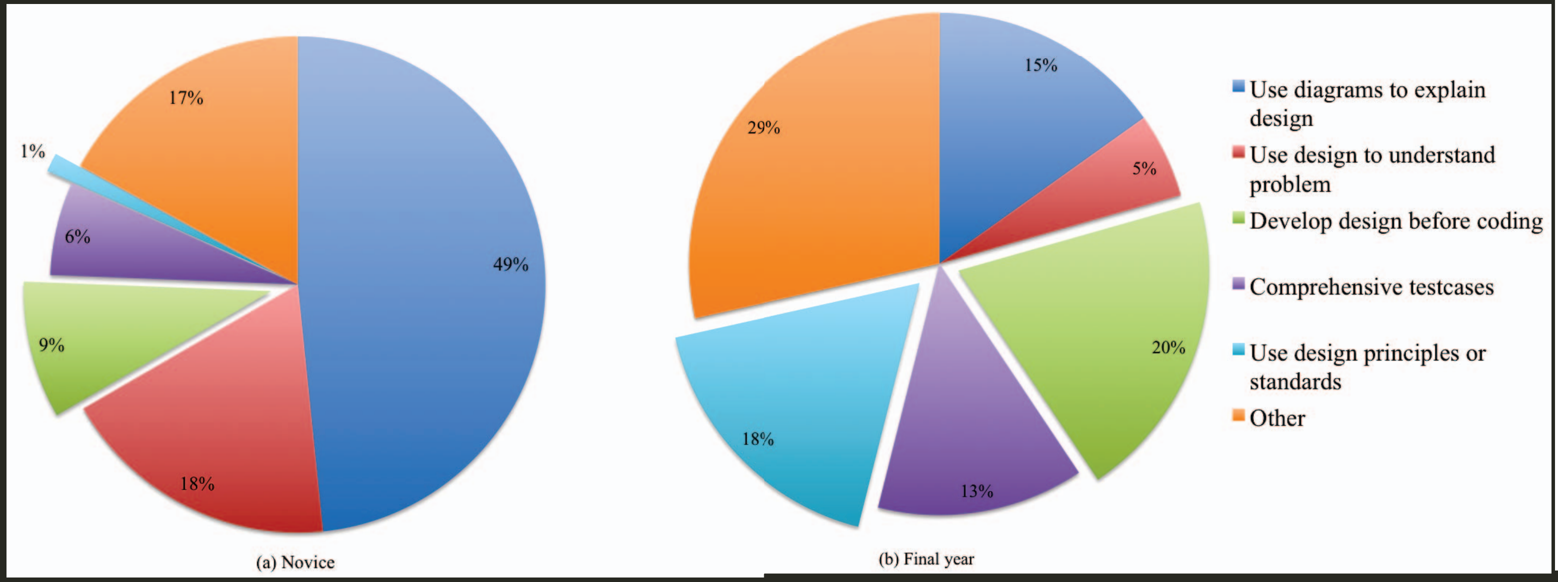
Then, we transform the above set of heuristics into metrics that will be used in our decomposition approach of software evolution in order to classify detected evolution phases in eight categories. Our classification should enable software managers to understand the major development activities that characterize a time period, and to detect recurrent patterns

978-1-4673-8159-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPC.2015.11

http://bibtex.github.io/ICPC-2015-BenomarASPS.html
http://dx.doi.org/10.1109/ICPC.2015.11

Evolution of Software Development Strategies

- Expert/novice devs
- Look at students
  - first year
  - final year



(a) Novice (b) Final year


2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Evolution of Software Development Strategies

Katrina Falkner, Claudia Szabo, Rebecca Vivian and Nickolas Falkner
School of Computer Science
The University of Adelaide,
Adelaide, Australia
Email: firstname.lastname@adelaide.edu.au

***Abstract*—The development of discipline-specific cognitive and meta-cognitive skills is fundamental to the successful mastery of software development skills and processes. This development happens over time and is influenced by many factors, however its understanding by teachers is crucial in order to develop activities and materials to transform students from novice to expert software engineers. In this paper, we analyse the evolution of learning strategies of novice, first year students, to expert, final year students. We analyse reflections on software development processes from students in an introductory software development course, and compare them to those of final year students, in a distributed systems development course. Our study shows that computer science - specific strategies evolve as expected, with the majority of final year students including design before coding in their software development process, but that several areas still require scaffolding activities to assist in learning development.**

I. Introduction

The development of deep learning strategies, self-regulation, abstract thinking and metacognitive strategies are vital in order to assist students in achieving success [1], [2]. A student with self-regulated learning behaviours will set their goals, determine and allocate their resources, as well as manage their time effectively [3]. Without this fundamental level of metacognition, students cannot direct their knowledge in a useful and constructive manner and thus are unlikely to succeed. A significant aspect in the development of self-regulating learning (SRL) strategies is the ability to monitor and reflect upon those strategies within the context of Computer Science (CS) as a discipline, enabling the individual to identify their success or failure, identify strategies to apply in specific contexts, and develop new strategies [4], [5]. Allwood [6] identifies that novices tend to use more general strategies rather than the more powerful specialised strategies employed by experts. According to Robillard [7], expert programmers tend to adopt a systematic planning process, based upon access to the conceptual knowledge required to complete the specified task, enabling a breadth-first search of the problem space. Novice programmers, however, build their planning and design processes upon their knowledge of programming languages, resulting in a depth-first search and a focus on concrete rather than abstract argumentation.

The transition from novice to expert is assisted by reflection on prior successes and failures [8], followed by analysis of potential areas for improvement. Before we can assist our students in the process of reflection and self-regulation, we must identify and articulate successful SRL strategies for the CS context [9]. Therefore, we must develop an understanding of those discipline specific strategies that can be successfully learnt and adopted by students [10].

In our previous work [11], we analysed students' reflections on their SRL processes as applied to introductory software development. Using a grounded theory model of qualitative analysis, we were able to identify SRL strategies that are specific to software development, expressed in the students' own words and relative to their own experiences. We presented a detailed analysis of the nature of these discipline-specific SRL strategies and how these strategies contribute to the learning of novice students. In this paper, we explore the evolution of discipline-specific SRL strategies through the combined analysis of a cohort of novice students, and a second cohort of final year students. We present an analysis of the evolution of SRL strategies from novice to expert learners, with the aim of identifying points where the development and application of identified SRL can be improved. The analysis of this evolution is fundamental for the development of targeted scaffolding and support to address identified issues. Our findings show that students develop a range of sophisticated strategies, that can be readily scaffolded within the curriculum.

II. Related Work

Self-regulated learners have been defined by Zimmerman [2] as those that "plan, set goals, organise, self-monitor and self-evaluate". The development of SRL strategies has been found to be a complex issue, associated with the perceived purpose of engagement with the activity, the students self-perception of their ability, and the situated context of the activity - these three factors impact upon the self-regulation strategies that the student then considers relevant for application [12]. Lichtinger and Kaplan [13] call for the identification of domain and context-specific purposes of engagement, and the articulation of types of SRL strategies that would be desirable for students within that domain. Further, the development of effective domain-specific SRL strategies, such as domain-based design and planning, can assist in the application and development of other SRL strategies [14]. Kramer [15] describes the difference between novice and expert software engineering students as their 'ability to perform abstract thinking and to exhibit abstraction skills'.

The categorisation of 'expert-novice' [16], a subset of novices who are able to progress quickly in their development of discipline knowledge, presents us with an interesting

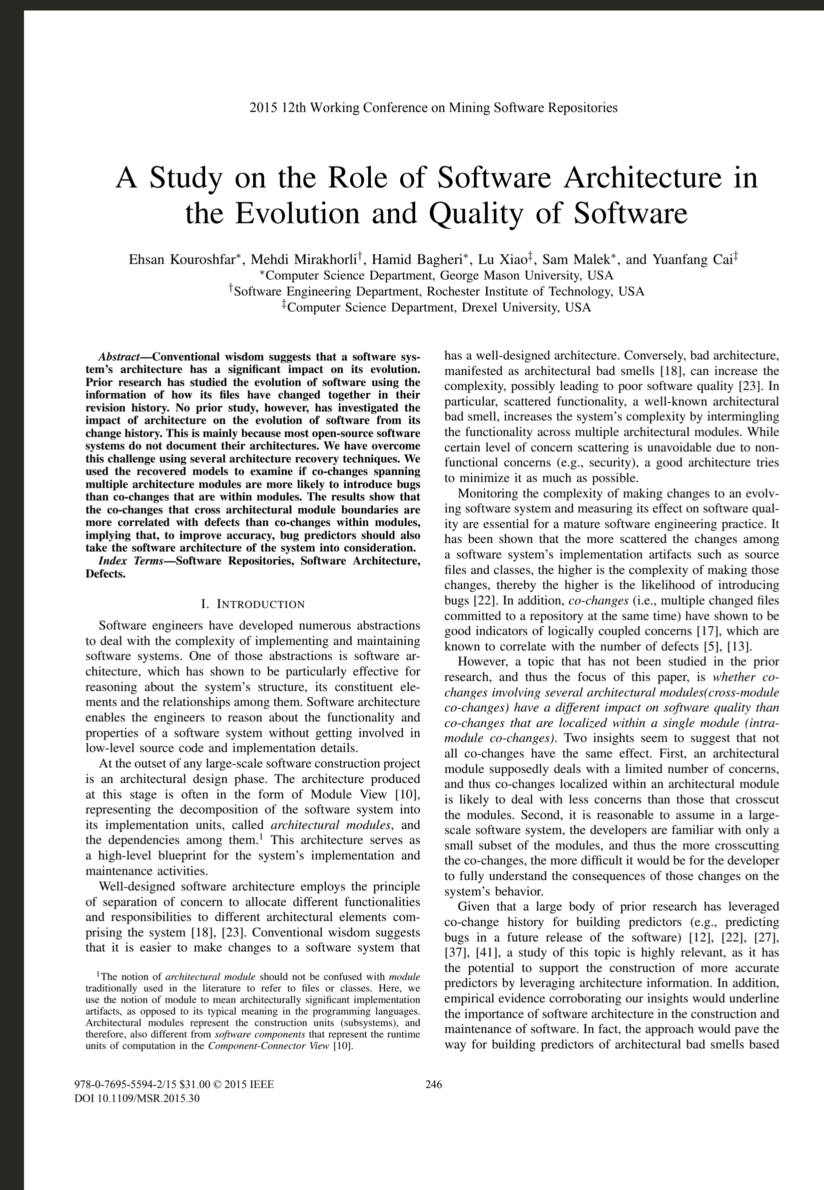
978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.153

ICSE 2015, Florence, Italy
Joint SE Education and Training

http://bibtex.github.io/ICSE-2015-FalknerSVF.html
http://dx.doi.org/10.1109/ICSE.2015.153

A Study on the Role of Software Architecture in the Evolution and Quality of Software

* Impact of architecture on evolution?
* Problem: documentation
* Reverse engineer!
  * heuristics, IR, DM, …
  * e.g. modularisation
* Architectural bad smells
  * x-module co-change
  * 20% defects, 2× time2fix



2015 12th Working Conference on Mining Software Repositories

A Study on the Role of Software Architecture in the Evolution and Quality of Software

Ehsan Kouroshfar*, Mehdi Mirakhorli†, Hamid Bagheri*, Lu Xiao‡, Sam Malek*, and Yuanfang Cai‡
*Computer Science Department, George Mason University, USA
†Software Engineering Department, Rochester Institute of Technology, USA
‡Computer Science Department, Drexel University, USA

***Abstract*—Conventional wisdom suggests that a software system's architecture has a significant impact on its evolution. Prior research has studied the evolution of software using the information of how its files have changed together in their revision history. No prior study, however, has investigated the impact of architecture on the evolution of software from its change history. This is mainly because most open-source software systems do not document their architectures. We have overcome this challenge using several architecture recovery techniques. We used the recovered models to examine if co-changes spanning multiple architecture modules are more likely to introduce bugs than co-changes that are within modules. The results show that the co-changes that cross architectural module boundaries are more correlated with defects than co-changes within modules, implying that, to improve accuracy, bug predictors should also take the software architecture of the system into consideration.**

***Index Terms*—Software Repositories, Software Architecture, Defects.**

I. Introduction

Software engineers have developed numerous abstractions to deal with the complexity of implementing and maintaining software systems. One of those abstractions is software architecture, which has shown to be particularly effective for reasoning about the system's structure, its constituent elements and the relationships among them. Software architecture enables the engineers to reason about the functionality and properties of a software system without getting involved in low-level source code and implementation details.

At the outset of any large-scale software construction project is an architectural design phase. The architecture produced at this stage is often in the form of Module View [10], representing the decomposition of the software system into its implementation units, called *architectural modules*, and the dependencies among them.[1] This architecture serves as a high-level blueprint for the system's implementation and maintenance activities.

Well-designed software architecture employs the principle of separation of concern to allocate different functionalities and responsibilities to different architectural elements comprising the system [18], [23]. Conventional wisdom suggests that it is easier to make changes to a software system that has a well-designed architecture. Conversely, bad architecture, manifested as architectural bad smells [18], can increase the complexity, possibly leading to poor software quality [23]. In particular, scattered functionality, a well-known architectural bad smell, increases the system's complexity by intermingling the functionality across multiple architectural modules. While certain level of concern scattering is unavoidable due to non-functional concerns (e.g., security), a good architecture tries to minimize it as much as possible.

Monitoring the complexity of making changes to an evolving software system and measuring its effect on software quality are essential for a mature software engineering practice. It has been shown that the more scattered the changes among a software system's implementation artifacts such as source files and classes, the higher is the complexity of making those changes, thereby the higher is the likelihood of introducing bugs [22]. In addition, *co-changes* (i.e., multiple changed files committed to a repository at the same time) have shown to be good indicators of logically coupled concerns [17], which are known to correlate with the number of defects [5], [13].

However, a topic that has not been studied in the prior research, and thus the focus of this paper, is *whether co-changes involving several architectural modules(cross-module co-changes) have a different impact on software quality than co-changes that are localized within a single module (intra-module co-changes)*. Two insights seem to suggest that not all co-changes have the same effect. First, an architectural module supposedly deals with a limited number of concerns, and thus co-changes localized within an architectural module is likely to deal with less concerns than those that crosscut the modules. Second, it is reasonable to assume in a large-scale software system, the developers are familiar with only a small subset of the modules, and thus the more crosscutting the co-changes, the more difficult it would be for the developer to fully understand the consequences of those changes on the system's behavior.

Given that a large body of prior research has leveraged co-change history for building predictors (e.g., predicting bugs in a future release of the software) [12], [22], [27], [37], [41], a study of this topic is highly relevant, as it has the potential to support the construction of more accurate predictors by leveraging architecture information. In addition, empirical evidence corroborating our insights would underline the importance of software architecture in the construction and maintenance of software. In fact, the approach would pave the way for building predictors of architectural bad smells based

[1]The notion of *architectural module* should not be confused with *module* traditionally used in the literature to refer to files or classes. Here, we use the notion of module to mean architecturally significant implementation artifacts, as opposed to its typical meaning in the programming languages. Architectural modules represent the construction units (subsystems), and therefore, also different from *software components* that represent the runtime units of computation in the *Component-Connector View* [10].

978-0-7695-5594-2/15 $31.00 © 2015 IEEE
DOI 10.1109/MSR.2015.30

http://bibtex.github.io/MSR-2015-KouroshfarMBXMC.html
http://dx.doi.org/10.1109/MSR.2015.30

Modelling the Evolution of Development Topics using Dynamic Topic Models

* Tasks evolve with sw
* Can be grouped by topic
  * Strength evolution
  * Content evolution
  * (never together)
* Use unstructured repos
* Visualise!

Modeling the Evolution of Development Topics using Dynamic Topic Models

Jiajun Hu*, Xiaobing Sun*‡, David Lo†, Bin Li*‡

*School of Information Engineering, Yangzhou University, Yangzhou, China

†School of Information Systems, Singapore Management University, Singapore

‡State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

jiajunhu.yzu.edu@gmail.com, xbsun@yzu.edu.cn, davidlo@smu.edu.sg, lb@yzu.edu.cn

***Abstract*—As the development of a software project progresses, its complexity grows accordingly, making it difficult to understand and maintain. During software maintenance and evolution, software developers and stakeholders constantly shift their focus between different tasks and topics. They need to investigate into software repositories (e.g., revision control systems) to know what tasks have recently been worked on and how much effort has been devoted to them. For example, if an important new feature request is received, an amount of work that developers perform on ought to be relevant to the addition of the incoming feature. If this does not happen, project managers might wonder what kind of work developers are currently working on.**

Several topic analysis tools based on Latent Dirichlet Allocation (LDA) have been proposed to analyze information stored in software repositories to model software evolution, thus helping software stakeholders to be aware of the focus of development efforts at various time during software evolution. Previous LDA-based topic analysis tools can capture either changes on the strengths of various development topics over time (i.e., strength evolution) or changes in the content of existing topics over time (i.e., content evolution). Unfortunately, none of the existing techniques can capture both strength and content evolution. In this paper, we use Dynamic Topic Models (DTM) to analyze commit messages within a project's lifetime to capture both strength and content evolution simultaneously. We evaluate our approach by conducting a case study on commit messages of two well-known open source software systems, *jEdit* and *PostgreSQL*. The results show that our approach could capture not only how the strengths of various development topics change over time, but also how the content of each topic (i.e., words that form the topic) changes over time. Compared with existing topic analysis approaches, our approach can provide a more complete and valuable view of software evolution to help developers better understand the evolution of their projects.

I. Introduction

Mining unstructured software repositories (e.g., bug reports, mailing lists, commit messages, etc.) has emerged as a research direction over the past decade, which has achieved substantial success in both research and practice to support software maintenance [1]–[3]. These studies have shown that interesting and practical results can be obtained from mining these software repositories, thus allowing maintainers or managers to better understand how software evolves.

Unlike structured contents in software repositories (e.g., source code, execution traces, change logs, etc.), unstructured contents are often harder to analyze because the data is often vague and noisy [4], making it time-consuming for project stakeholders to manually analyze software repositories. One of recent advanced techniques is to use topic analysis tools (topic models), such as Latent Dirichlet Allocation (LDA) [5], to automatically extract topics from textual repositories to explore and organize the underlying structure of software documents [6]–[16]. Topic models can be used to discover a set of ideas or themes (aka., topics) that well describe the entire corpus. Topics are collections of words that co-occur frequently in the entire corpus and usually have a close semantic relationship. More specifically, a topic model can represent a set of documents as a set of topics, where each document contains one or more of these topics, and each topic is composed of a set of words that appear in the repository.

Understanding how development topics evolve, i.e., change, in a software repository over time can help project stakeholders to understand and monitor activities performed in their project at various time points during project's lifetime. For example, project managers can understand what tasks have recently been worked on and how much effort has been devoted to each task by retrieving revision control systems [10], while developers can understand the evolution of certain features of source code by mining source code repository [6], [9].

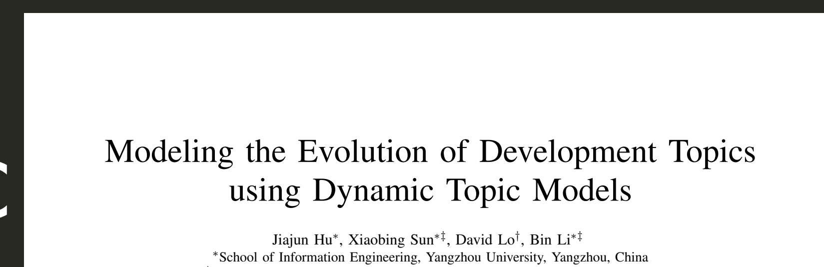
To help developers understand software evolution, a number of LDA-based approaches have been proposed. Thomas et al. applied *the Hall model* [17] to analyze the entire history of source code documents to recover information on how the strengths (i.e., popularity) of various topics change over time [6], [9]. They ran *LDA* once on all versions of a software project to get a set of topics, and then computed several metrics to represent the strength of a topic for each of the version. In such a way, their approach can capture the *strength evolution* of the development topics. However, the content of a topic (i.e., the set of words that form a topic), never changes across the versions. On the other hand, Hindle et al. applied *the Link model* [18] which runs LDA for each time window separately and then used a post-processing phase to link topics which are similar enough across successive time windows [10]. Their approach can capture changes in the content of each topic over time (i.e., *content evolution*). Unfortunately, it cannot recover the strength of a topic across all time windows – for some time windows, some topics do not exist and are expressed as combinations of other topics. Thus, none of existing approaches can capture both strength and content evolution.

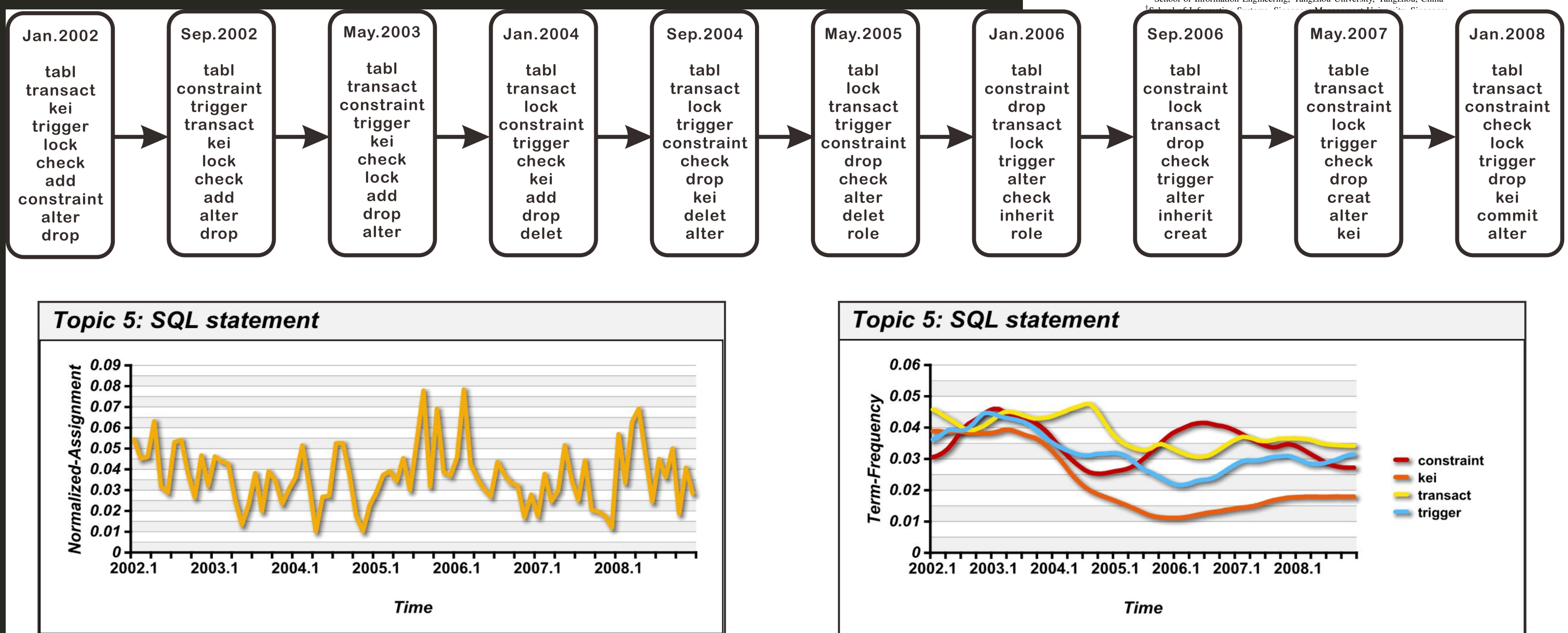
http://bibtex.github.io/SANER-2015-HuSLL.html

http://dx.doi.org/10.1109/SANER.2015.7081810

Modelling the Evolution of Development Topics using Dynamic Topic Models

- Tasks evolve with sw
- Can be grouped by topic





http://bibtex.github.io/SANER-2015-HuSLL.html
http://dx.doi.org/10.1109/SANER.2015.7081810

Mining Software Contracts for Software Evolution

- Version control systems
  - git, svn, cvs
- Program contract
  - pre- & postcond, inv
  - “requires”, “ensures”…
- Bugfixes & contracts coevolve
- Some things easier to RE
  - from contracts
- http://github.com/ybank/inv-research

2014 IEEE International Conference on Software Maintenance and Evolution

Mining Software Contracts for Software Evolution

Yan Yan, Massimiliano Menarini, William Griswold
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, USA.
{yayan, mmenarini, wgg}@cs.ucsd.edu

***Abstract*— Maintenance and evolution are important parts for all successful software projects. In recent years, version control systems have played a key role in software development process. Not only do they provide a means to coordinate programmers, organize and manage source code, but they also persist the evolution history of the source code into their software repositories. Mining software repositories has provided many insights on the evolution of software, both for researchers and practitioners. In this paper we propose that versioned software contracts – mined from software repositories – can be a powerful tool for better understanding and supporting software evolution. Tooling support is critical, due to the complexities of configuring, compiling, and running the software to produce meaningful inferred contracts. This paper contributes both techniques and tool support for downloading, building, and analyzing open source software from social coding sites like GitHub. The tool automatically produces a description of software evolution represented by versions of program invariants.**

***Keywords*— software evolution; version control; contracts; program analysis; software testing**

I. Introduction

Modern software projects relay on Version Control Systems (VCS) to help developers communicate and manage bug fixes and software releases. Well-known VCS include CVS [1], SVN [2], and Git [3]. VCS have a central repository where all authorized programmers can upload their changes. This repository maintains all versions of the software code; it also enables analyzing differences among various software releases. Most industrial software projects update their products when a new stable version (or a milestone) is available to release. Between releases programmers work on new features or fix reported bugs, and push their changes to the repository. When the new version is ready, a snapshot of the repository is taken and the code is compiled to produce the next version of the program. In certain cases the development teams can even revert their product back to a previous version that was persisted in the repository to remove problems introduced in the latest version.

Therefore, VCS repositories provide a historical view of the software evolution. Such information can be helpful in tackling many challenging software engineering problems, such as: reusing software components, debugging, predicting future code changes, and more.

In this paper we leverage this data by first using Daikon [4] framework for creating contracts for each method implemented in the different versions of source code, and then identifying how these contracts change across different versions. To this end we created a semi-automatic tool that downloads source code from Git repositories, builds the project using a modified version of Apache Maven [5] project management tool, and uses Daikon to extract changes in method automatically. In our experiments, we demonstrate two potential benefits of this approach by showing how certain refactorings and bug fixes are identified by our approach.

We structure the remainder of this paper as follows. In the next section, we analyze the research related to mining the source code history for discovering useful information from certain facets of the code. Then in Section 3 we discuss the tools and practices that support our approach to understand the evolution of software systems. In section 4 we present some experimental results that we discuss in section 5. Finally we present our conclusions and outline future work.

II. Related Work

One frequently visited problem is how developers can make the most use of the legacy code. In [6] researchers mine into software repository and find relevant API usage examples based on call-graph information. Similarly, [7] finds code snippets based on the interaction patterns. Then researchers apply certain data mining algorithms [8] based on the information found. The historical information is also helpful for debugging. [9] improves traditional static bug finding process by searching into source code change histories for previously fixed bugs. [10] shows how versioning can help make debugging more effective and efficient. Researchers generally inspect the source code and relate snippets with types of bugs they defined, then use this information to assist future debugging [9]. There are more research scenarios where the source code evolution information helps. For example, researchers can use it to predict future code changes [11], [12]. In addition to helping with coding activities, analyzing software repositories can help understand a wider range of aspects of software evolution, like developers’ efforts. For example, [13] takes as input the commit logs and bug reports to detect “hotspots” where higher development activities are indicated.

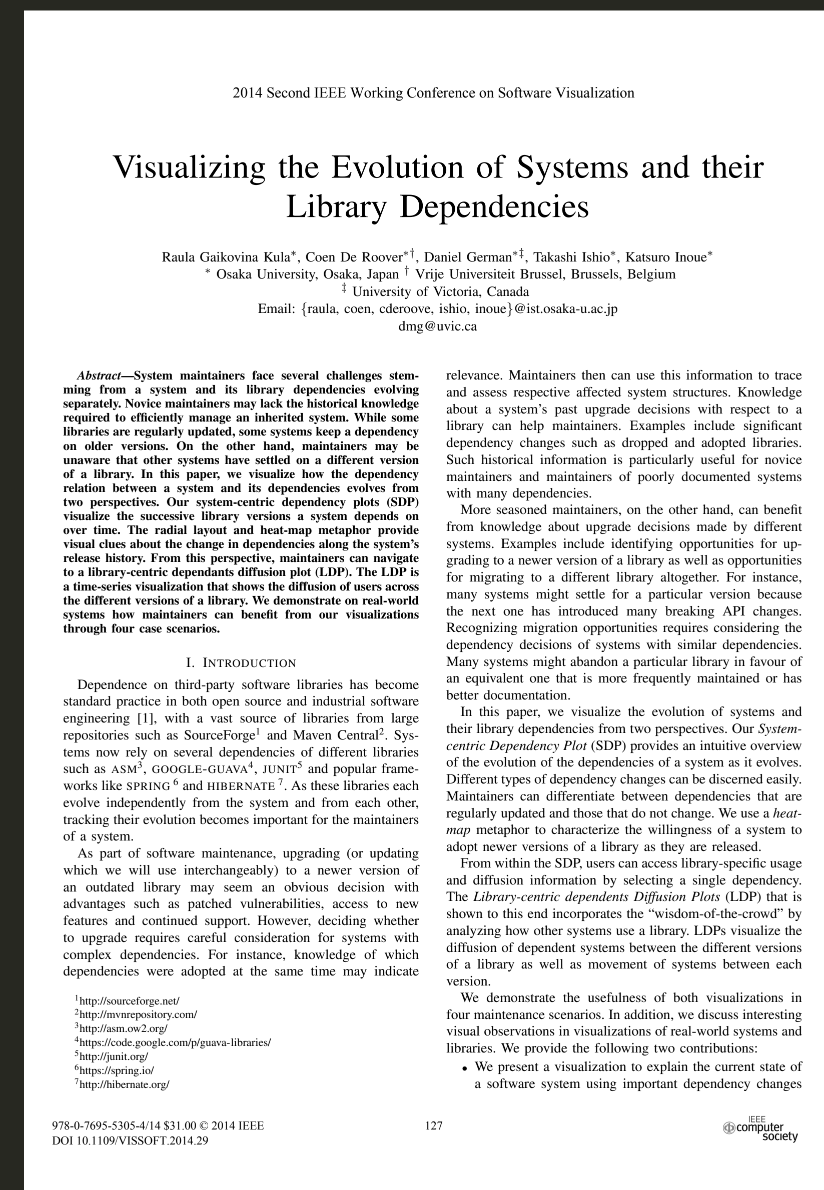
In summary, most researchers analyze code histories in ways that are tightly coupled with, but contribute the most to

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.76

IEEE computer society

http://bibtex.github.io/ICSME-2014-YanMG.html
http://dx.doi.org/10.1109/ICSME.2014.76

Visualising the Evolution of Systems and Their Library Dependencies



2014 Second IEEE Working Conference on Software Visualization

Visualizing the Evolution of Systems and their Library Dependencies

Raula Gaikovina Kula*, Coen De Roover*†, Daniel German*‡, Takashi Ishio*, Katsuro Inoue*
* Osaka University, Osaka, Japan † Vrije Universiteit Brussel, Brussels, Belgium
‡ University of Victoria, Canada
Email: {raula, coen, cderoove, ishio, inoue}@ist.osaka-u.ac.jp
dmg@uvic.ca

***Abstract*—System maintainers face several challenges stemming from a system and its library dependencies evolving separately. Novice maintainers may lack the historical knowledge required to efficiently manage an inherited system. While some libraries are regularly updated, some systems keep a dependency on older versions. On the other hand, maintainers may be unaware that other systems have settled on a different version of a library. In this paper, we visualize how the dependency relation between a system and its dependencies evolves from two perspectives. Our system-centric dependency plots (SDP) visualize the successive library versions a system depends on over time. The radial layout and heat-map metaphor provide visual clues about the change in dependencies along the system's release history. From this perspective, maintainers can navigate to a library-centric dependants diffusion plot (LDP). The LDP is a time-series visualization that shows the diffusion of users across the different versions of a library. We demonstrate on real-world systems how maintainers can benefit from our visualizations through four case scenarios.**

I. Introduction

Dependence on third-party software libraries has become standard practice in both open source and industrial software engineering [1], with a vast source of libraries from large repositories such as SourceForge[1] and Maven Central[2]. Systems now rely on several dependencies of different libraries such as ASM[3], GOOGLE-GUAVA[4], JUNIT[5] and popular frameworks like SPRING[6] and HIBERNATE[7]. As these libraries each evolve independently from the system and from each other, tracking their evolution becomes important for the maintainers of a system.

As part of software maintenance, upgrading (or updating which we will use interchangeably) to a newer version of an outdated library may seem an obvious decision with advantages such as patched vulnerabilities, access to new features and continued support. However, deciding whether to upgrade requires careful consideration for systems with complex dependencies. For instance, knowledge of which dependencies were adopted at the same time may indicate relevance. Maintainers then can use this information to trace and assess respective affected system structures. Knowledge about a system's past upgrade decisions with respect to a library can help maintainers. Examples include significant dependency changes such as dropped and adopted libraries. Such historical information is particularly useful for novice maintainers and maintainers of poorly documented systems with many dependencies.

More seasoned maintainers, on the other hand, can benefit from knowledge about upgrade decisions made by different systems. Examples include identifying opportunities for upgrading to a newer version of a library as well as opportunities for migrating to a different library altogether. For instance, many systems might settle for a particular version because the next one has introduced many breaking API changes. Recognizing migration opportunities requires considering the dependency decisions of systems with similar dependencies. Many systems might abandon a particular library in favour of an equivalent one that is more frequently maintained or has better documentation.

In this paper, we visualize the evolution of systems and their library dependencies from two perspectives. Our *System-centric Dependency Plot* (SDP) provides an intuitive overview of the evolution of the dependencies of a system as it evolves. Different types of dependency changes can be discerned easily. Maintainers can differentiate between dependencies that are regularly updated and those that do not change. We use a *heat-map* metaphor to characterize the willingness of a system to adopt newer versions of a library as they are released.

From within the SDP, users can access library-specific usage and diffusion information by selecting a single dependency. The *Library-centric dependents Diffusion Plots* (LDP) that is shown to this end incorporates the "wisdom-of-the-crowd" by analyzing how other systems use a library. LDPs visualize the diffusion of dependent systems between the different versions of a library as well as movement of systems between each version.

We demonstrate the usefulness of both visualizations in four maintenance scenarios. In addition, we discuss interesting visual observations in visualizations of real-world systems and libraries. We provide the following two contributions:

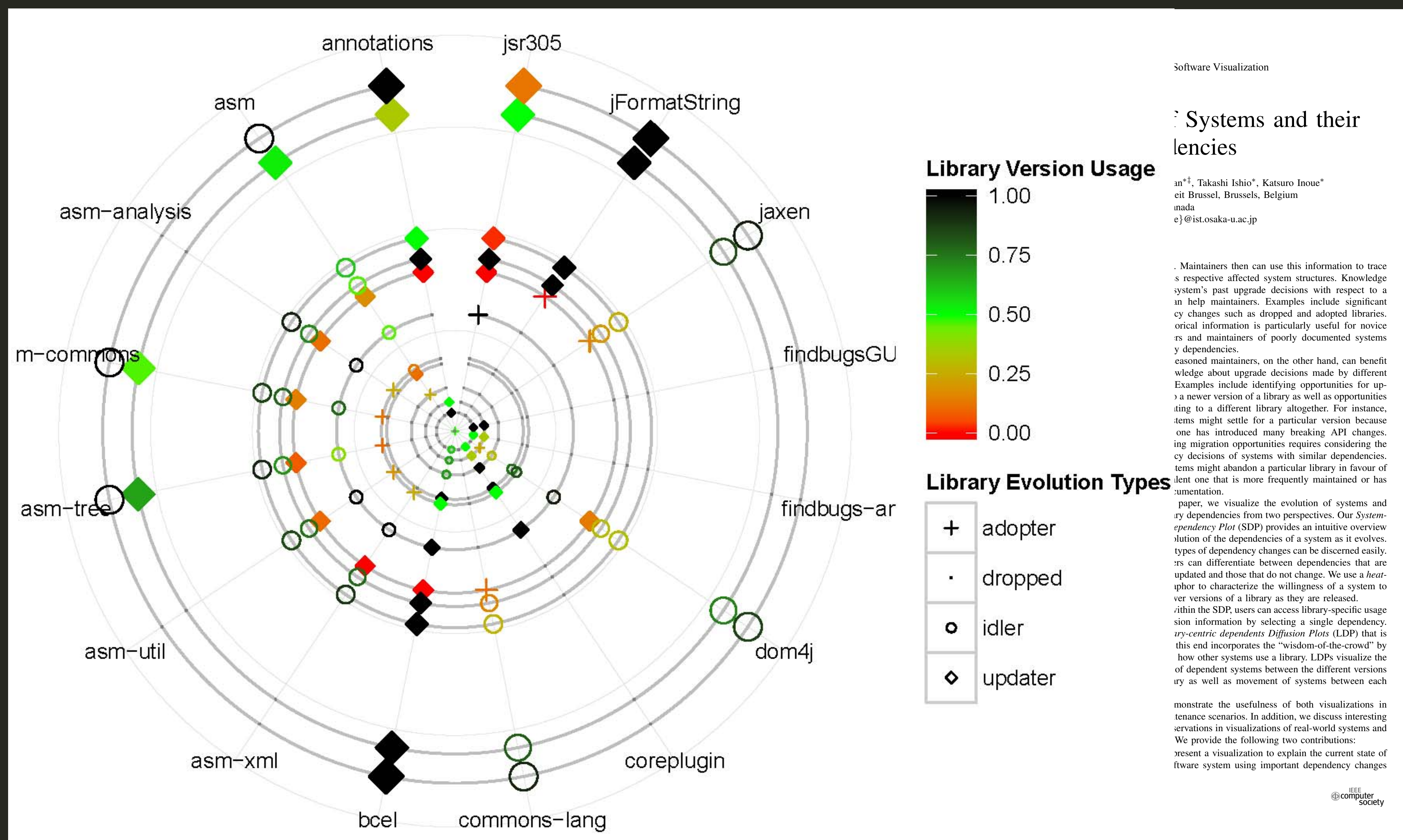
- We present a visualization to explain the current state of a software system using important dependency changes

[1] http://sourceforge.net/
[2] http://mvnrepository.com/
[3] http://asm.ow2.org/
[4] https://code.google.com/p/guava-libraries/
[5] http://junit.org/
[6] https://spring.io/
[7] http://hibernate.org/

978-0-7695-5305-4/14 $31.00 © 2014 IEEE
DOI 10.1109/VISSOFT.2014.29

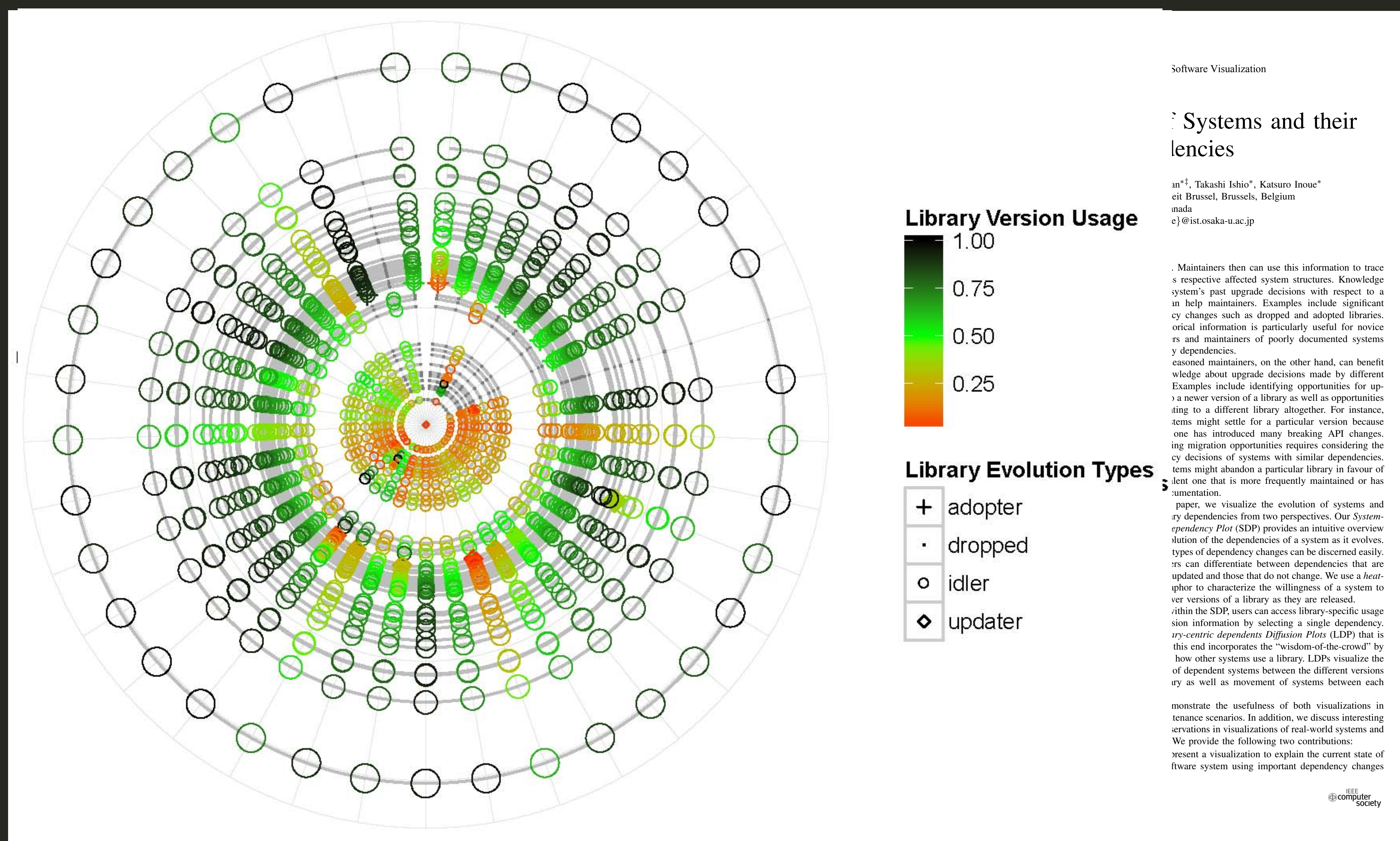
http://bibtex.github.io/VISSOFT-2014-KulaRGII.html
http://dx.doi.org/10.1109/VISSOFT.2014.29

Visualising the Evolution of Systems and Their Library Dependencies



http://bibtex.github.io/VISSOFT-2014-KulaRGII.html
http://dx.doi.org/10.1109/VISSOFT.2014.29

Visualising the Evolution of Systems and Their Library Dependencies



http://bibtex.github.io/VISSOFT-2014-KulaRGII.html
http://dx.doi.org/10.1109/VISSOFT.2014.29

Conclusion

* Software evolves
* Software evolution obeys certain laws
* Software rots in time (quality, complexity…)
* 70% of software engineers do maintenance
* Many software systems are legacy
* Forward, reverse and re-engineering
* Actively researched field
* Learn to build tools

Start with !

Questions?