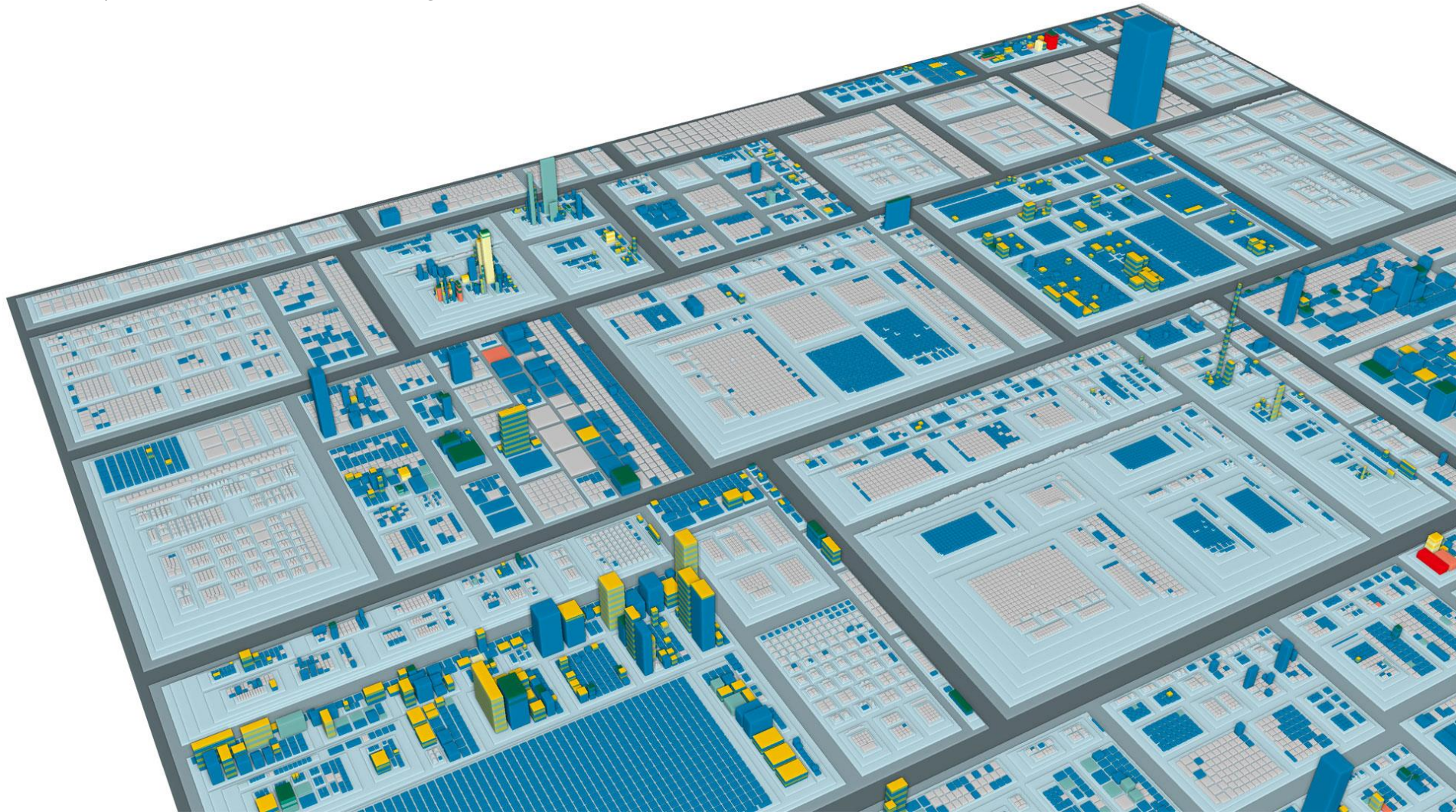
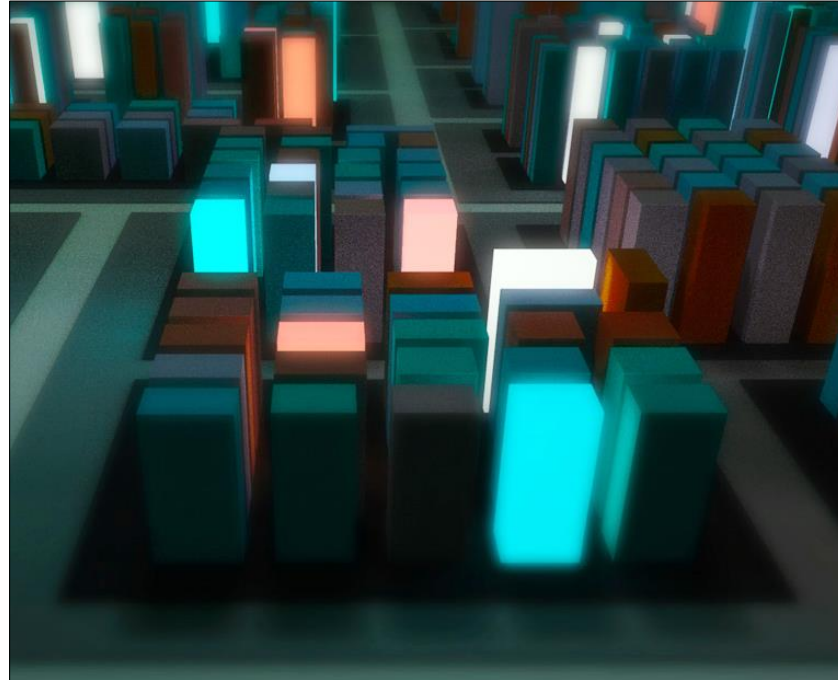


Automated Visual Software Analytics

OpenHPI – Prof. Dr. Jürgen Döllner – 2015



Course Overview



Course Contents
Learning Objectives
Clip Structure
Course Organization

**The purpose of this course is to outline and motivate
'automated visual software analytics'
as a key methodology and technology that aims at
improving effectiveness of software development and software engineering.**

Course Contents

This course outlines 'automated visual software analytics' and includes related topics from

- software engineering;
- information visualization;
- big data, visual, and predictive analytics;
- software analytics.

Prerequisites

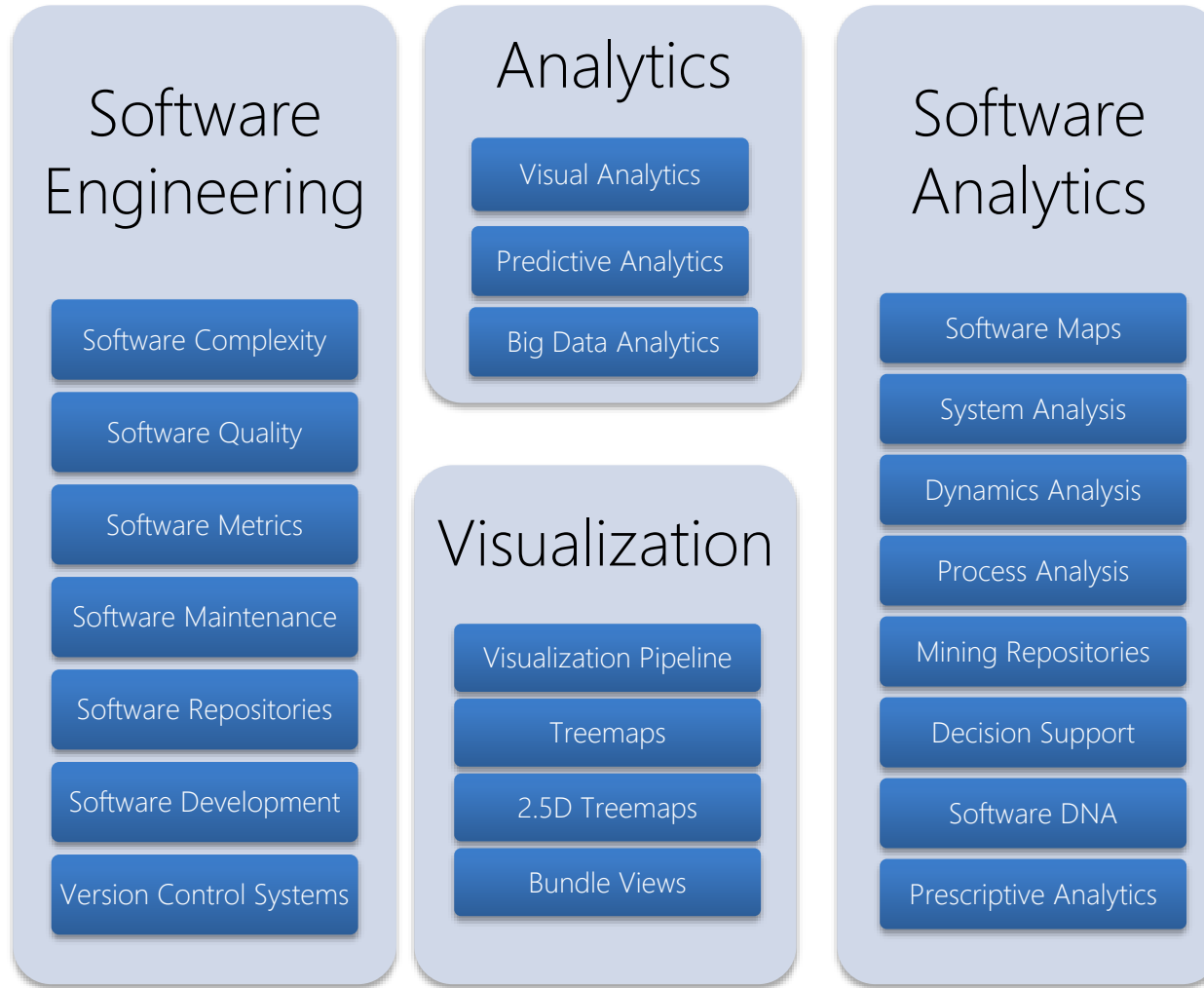
For course participation, it is beneficial and assumed that

- you have basic experience in software engineering and software development projects;
- you have participated in software development projects in one or more roles (manager, team leader, architect, designer, developer, tester, ...);
- you have been exposed to aspects of the long-term handling of software-based projects;
- you want to know more about methodology of IT development.

Prerequisites

What this course does **not** deliver:

- It does not give a first or general introduction to software development and software engineering.
- It does not introduce new modeling approaches, new programming languages, new programming paradigms, or new frameworks.
- It does not provide tricks or best-practices for programming, software design, software architecture, etc.
- It does not give instructions or solutions for solving specific software development issues.



Week 1

Introduction to Software Engineering

- *Course Contents and Learning Objectives*
- Software Dependency
- Software Development
- Software Complexity
- Software Maintenance
- Static Source Code Analysis

Week 2

Software Metrics

- Software Metrics
- Lines-of-Code (LoC, ...)
- Code Duplicates
- Nesting Level (NL)
- Cyclomatic Complexity (McCabe)
- Halstead Complexity
- Module Dependencies
- OO Metrics

Week 3

Analytics

- Analytics
- Visual Analytics
- Visualization Pipeline
- Predictive Analytics
- Software Analytics

Week 4

Automated Data Mining

- Version Control Systems
- Mining SW Repositories

Visualization Techniques

- Treemaps
- 2.5D Treemaps
- Hierarchical Circular Bundle Views

Week 5

Applying Visual Software Analytics

- Software Maps
- Exploring System Implementations
- Discovering Error-Prone Code
- Monitoring Technical Debts
- Involvement and Knowledge Distribution
- Refactoring Planning
- Monitoring Redesign Processes

Week 6

Related Techniques and Outlook

- Tracing
- TraceViews
- Code Usage & Test Coverage
- Software Effectiveness
- Prescriptive Software Analytics

At the conclusion of this course, participants should be able to

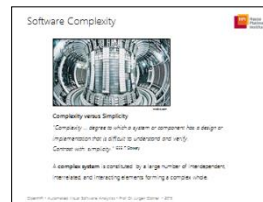
- outline concepts and methods from scientific fields that contribute to software analytics;
- describe objectives underlying “data-driven” software engineering approaches;
- discuss challenges for software maintenance due to software complexity and dependency;
- list, describe, and compare software metrics as quality measures used in software analytics;
- outline and contrast different visualization techniques used by software analytics;
- obtain first experience in using automated visual software analytics tools;
- reflect on using elements of automated visual software analytics in own projects as means to improve effectiveness of software development processes.

Clip Structure

Course Title



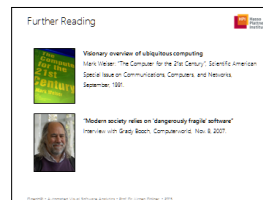
Content



Clip Topic (End)



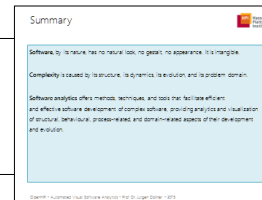
Further Reading



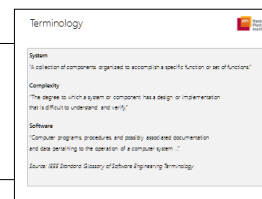
Clip Topic (Start)



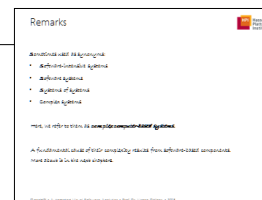
Summary



Terminology



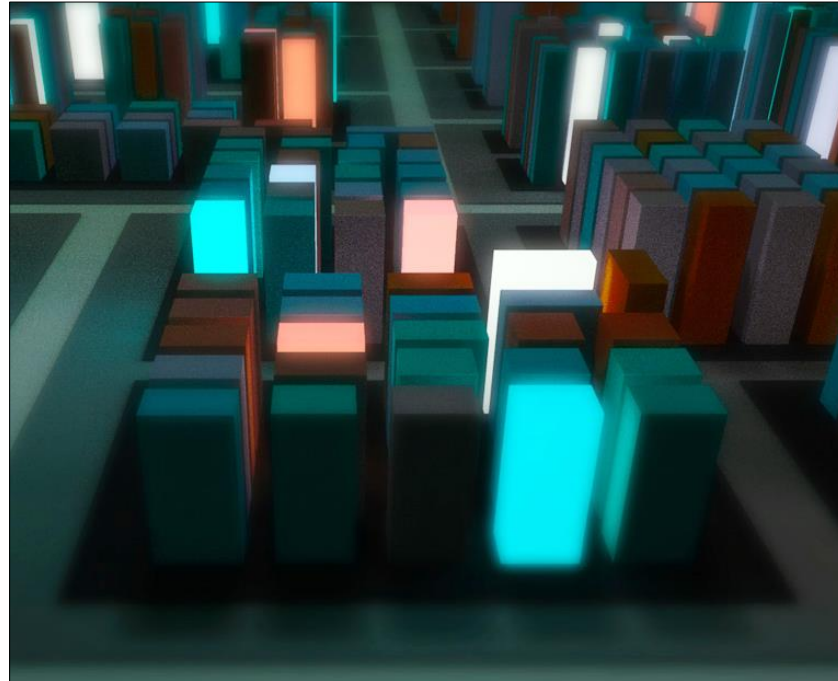
Remarks



- Six **lecture weeks**, final exam in week 7
- Several **lecture units** per week, per unit:
 - Video, slides, non-graded self-test
 - Sometimes mandatory and optional readings
 - Sometimes optional analytics tasks
 - Week finished with a **graded assignment**
- Six graded assignments sum up to max. 90 points
- **Graded final exam** with max. 90 points
- OpenHPI certificate awarded for getting ≥ 90 points in total
- Forum can be used to discuss with other participants and teaching team members
- FAQ is constantly updated

- ALL INFORMATION OF THIS COURSE ARE PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.
- HPI ASSUMES NO RESPONSIBILITY FOR ERRORS OR OMISSIONS IN THE INFORMATION OR OTHER DOCUMENTS WHICH ARE REFERENCED BY OR LINKED TO THIS COURSE.
- IN NO EVENT SHALL HPI BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR ANY DAMAGES WHATSOEVER, WHETHER OR NOT HPI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.
- THIS COURSE COULD INCLUDE TECHNICAL OR OTHER INACCURACIES. CHANGES ARE PERIODICALLY MADE TO THE INFORMATION HEREIN. HOWEVER, HPI MAKES NO COMMITMENT TO UPDATE MATERIALS ON THIS SITE.
- AS PART OF THIS COURSE, WE OFFER ACCESS TO THE SOFTWARE ANALYTICS TOOLS AND SYSTEMS OF HPI'S RESEARCH PARTNERS SOFTWARE DIAGNOSTICS TECHNOLOGY GMBH AND SOFTWARE DIAGNOSTICS GMBH; THEY PROVIDE FREE ACCESS AND USE, RESTRICTED TO THE USE CASES EXPLICITLY CONTAINED AND DEFINED IN THIS COURSE ("USE CASES") AND ARE ALLOWED TO USE PARTICIPANTS' EMAIL AND CONTACT DATA IN THIS CONTEXT. THE SPECIFIC CONDITIONS OF USE FOR THE AFOREMENTIONED SOFTWARE ANALYTICS TOOLS AND SYSTEMS ARE DEFINED SEPARATELY BY THEM.

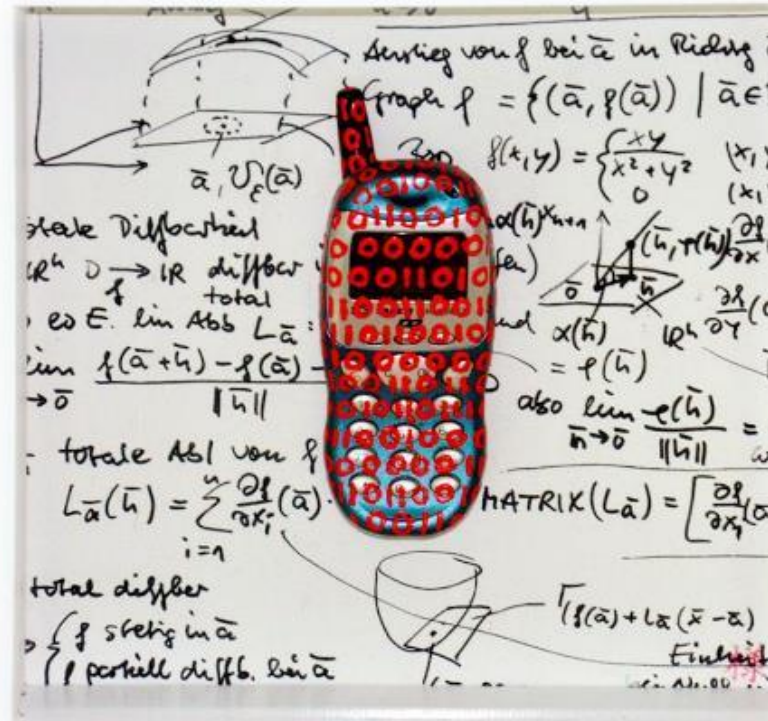
Course Overview



Course Contents
Learning Objectives
Clip Structure
Course Organization

- We explicitly invite participants that are involved in software development projects at a high level of abstraction such as CTOs, CEOs, CIOs,
- We explicitly invite participants that are confronted by seeking future-proof methods to handle large, distributed, complex, or wired software projects
- We explicitly invite participants that work as consultants in the field of IT
- Last but not least: we invite, of course, all professionals in software engineering, IT development, and big data analytics.
- We will get access to a software analytics system of HPI's research partner Software Diagnostics used to practically experience software analytics

Software Dependency



Martin Reiter: Digital Imperative

Computer-Based Systems
System Dependability

Computer-Based Systems

- Applications
- IT Systems
- Embedded Systems

```
> dump $2000 $2200
03-2000: 4083 130D 1311 1314 021F 2023 4C49 5354 0..... #LIST
03-2008: 204B 4559 A807 6C0F 0000 0828 47D1 443A KEY..l...<G.D:
03-2010: 44E0 0360 F187 607C 2034 321D 07EC 0A1D D...`...! 42.....
03-2018: 47D7 F082 44F6 6C05 6D2D 4526 E8C6 F081 G...D.l.m-E&....
03-2020: 4523 F081 F081 2060 4C49 5354 A805 6C2C E#.... `LIST..l.
03-2028: 0000 0A34 C12C 082A 47B3 441C 7171 834B ...4...*G.D.qq.K
03-2030: 2025 B800 7171 834B 2026 321D 0001 B21D %...qq.K &2.....
03-2038: 834B 2025 8000 F020 2001 F403 47B8 6FBC .K %... ..G.o.
03-2040: 0360 F187 607C 2038 321D 07BB 0A1D 47A8 -...! 82.....G.
03-2048: F082 7170 3360 7171 834B 2023 B800 7171 ..qp3`qq.K #...qq
03-2050: 2021 B800 F081 44BD 6C09 6D03 C103 E8C6 ?....D.l.m.....
03-2058: 4504 6D01 C103 E8C6 F081 C103 6CFA 6FF9 E.m.....l.o.
03-2060: 2077 5255 4E20 A805 6C0F 0000 4406 F081 wRUN ..l...D...
03-2068: 0796 086C 4785 F082 0824 4772 7171 834B ...lG....$Grqq.K
03-2070: 2023 B800 F082 C0CE C103 6CE3 F081 208A #.....l....
03-2078: 434F 4E54 A805 6C08 0000 47EF F081 077E CONT..l...G....~
03-2080: 086C 476E F082 C0CE C103 6C02 F081 0887 .lGn.....l.....
03-2088: C0D3 F081 2097 5343 5241 5443 4820 4120 .... .SCRATCH A
03-2090: A808 E8CE 0000 C168 0000 0087 6C3E 20AA .....h.....l> .
03-2098: 5343 5241 5443 4820 4B45 5920 A809 6C07 SCRATCH KEY ..l.
03-20A0: 0000 444D 0761 0834 474B F082 C0CE 449C ..DM.a.4GK....D.
03-20A8: F081 F081 20B5 5343 5241 5443 4820 5020 .... .SCRATCH P
03-20B0: A808 E8CE 0000 0022 6C20 20C0 5343 5241 .....`l .SCRA
```

Computer-Based Systems

Operational Systems and Embedded Systems



Computer-Based Systems

Social Networks and Communication Services



Seamless and Invisible Integration of Computer-Based Systems

- Augment Reality
- Semantic Web
- Internet of Things
- Ambient Intelligence
- Pervasive Computing
- Ubiquitous Computing

Seamless and Invisible Integration of Computer-Based Systems

- Augment Reality
- Semantic Web
- Internet of Things
- Ambient Intelligence
- Pervasive Computing
- Ubiquitous Computing

The Computer for the 21st Century

*Specialized elements of hardware and software,
connected by wires, radio waves and infrared, will be
so ubiquitous that no one will notice their presence*

by Mark Weiser

The most profound technologies
are those that disappear. They
weave themselves into the fabric
of everyday life until they are indistin-
guishable from it.

Key Software Abilities and Qualities

- **Availability:** Readiness for correct service
- **Reliability:** Continuity of correct service
- **Safety:** Absence of catastrophic consequences on user(s) and the environment
- **Integrity:** Absence of improper system alteration
- **Maintainability:** Ability for a process to undergo modifications and repairs



economist.com



jpl.nasa.gov



Consequences

- Deficiencies of computer-based systems directly interfere our daily lives
- Increasing risks, e.g., in terms of “total failures”

Examples

- Airbag software bugs
- “... newly installed trading software went haywire

Software Glitch Costs Trading Firm

\$440 Million in 45 Minutes Knight Capital, 2012

gizmodo.in/wtf/Software-Glitch-Costs-Trading-Firm-440-Million-In-45-Minutes/articleshow/19178948.cms



Stroustrup: We should aim for a greater degree of professionalism in software development. There are things in our society that musn't break, and most of them depend on software. And we need to consider our systems as a whole. Languages and their tool chains are part of it. We need to look into how we educate our developers and how we put emphasis on reliability.

(Bjarne Stroustrup: Interview InfoWorld.com "Why the 35-year-old C++ still dominates 'real' dev")

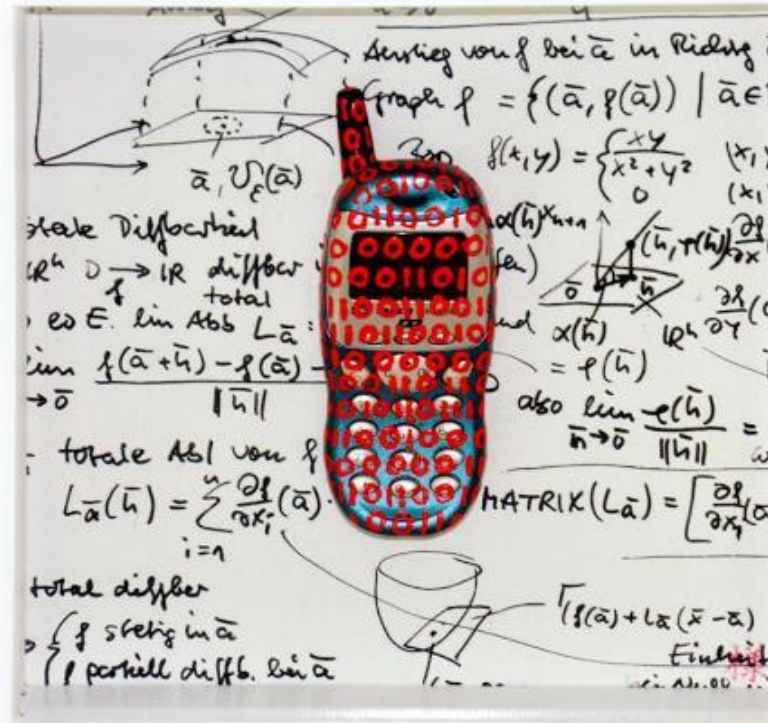
Grady Booch: *"Many systems look dangerously fragile. The bad news is they are fragile. This is another force that will lead us to the next era of how we build software systems"*

(Computerworld, Nov. 8, 2007)

Complex computer-based systems provide a large number of challenges for their planning, construction, and management. One key issue is system dependability due to the general and increasing "software-dependency" in our society.

Automated visual software analytics focuses on complex computer-based systems, in particular on the software-based components. Its goal is to enable software development teams to handle tasks, budgets, risks, and quality.

Software Dependency



Martin Reiter: Digital Imperative

Computer-Based Systems
System Dependability

System

"A collection of components organized to accomplish a specific function or set of functions."

Complexity

"The degree to which a system or component has a design or implementation that is difficult to understand and verify."

Software

"Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system ..."

Source: IEEE Standard Glossary of Software Engineering Terminology

Complex Computer-Based System

A complex computer-based system is a system that is non-simple with respect to its software and hardware components, their structure and organization and that is difficult to explore, analyze, comprehend and visualize for humans.

System Dependability

Dependability is a measure of a system's availability, reliability, and its maintainability.

It covers availability, reliability, safety, integrity, and maintainability of a system.

Dependability can be affected by faults, errors, and failures.

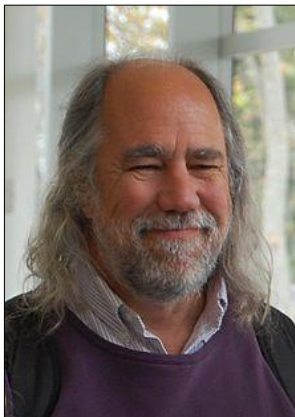
“Software Dependency”

“Software dependency” refers to the fact that almost all segments in private life, economy, and even politics considerably and increasingly are coupled with the dependability of complex computer-based systems.



Visionary overview of ubiquitous computing

Mark Weiser: "The Computer for the 21st Century", Scientific American Special Issue on Communications, Computers, and Networks, September, 1991.



"Modern society relies on 'dangerously fragile' software"

Interview with Grady Booch, Computerworld, Nov. 8, 2007.

Sometimes used as synonyms:

- Software-intensive systems
- Software systems
- Systems of systems
- Complex systems

Here, we refer to them as *complex computer-based systems*.

A fundamental cause of their complexity results from software-based components.

More about it in the next chapters.

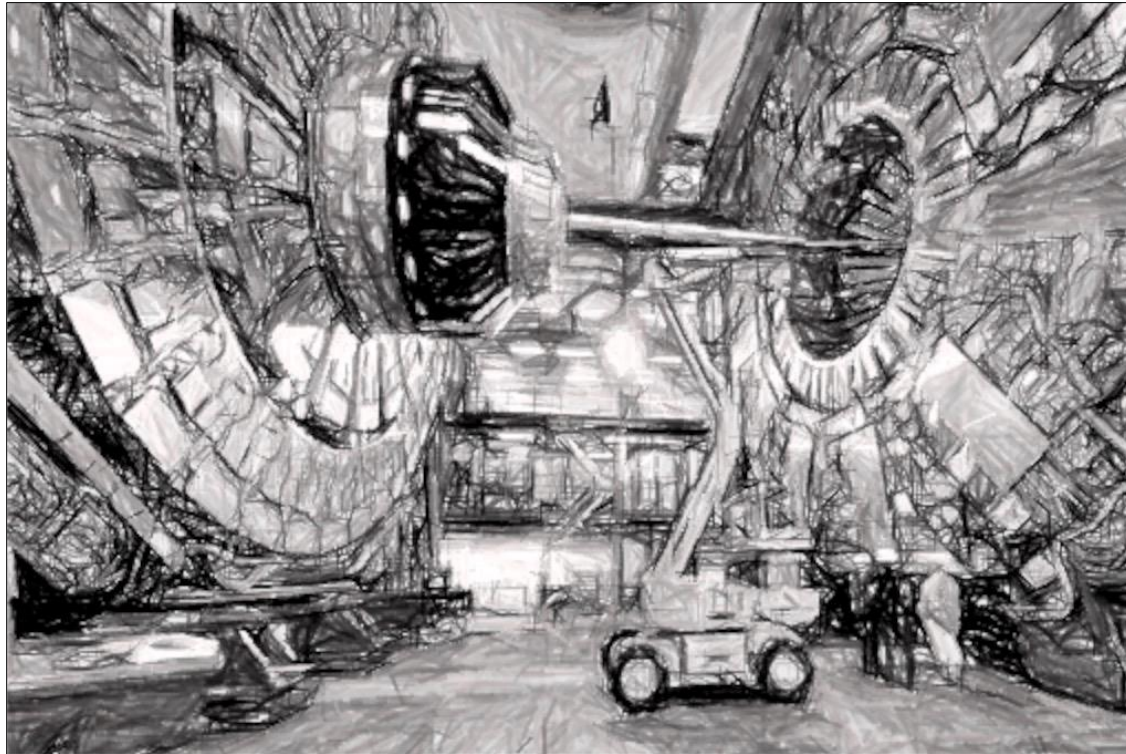
Recent system dependability issue in the news:

FEB. 2, 2015

[Chrysler](#) will recall more than 228,000 vehicles because their airbags could unexpectedly inflate without a crash. The company will upgrade software governing side-curtain and seat-mounted side airbags in Jeep Cherokees from the 2014-15 model years, including more than 168,000 vehicles in the United States. Chrysler engineers initiated an investigation after getting reports of “a small number of inadvertent deployments involving drivers who executed extreme maneuvers.” In these cases, the airbag systems, sensing potential rollovers, automatically activated. The action follows a much larger recall for inadvertent airbag deployments announced on Saturday by regulators at the [National Highway Traffic Safety Administration](#). That recall covers about 2.1 million vehicles made by Chrysler, Toyota and Honda with airbags that might suddenly deploy.

A version of this brief appears in print on February 3, 2015, on page B2 of the New York edition with the headline: Jeep Cherokees to Be Recalled to Upgrade Airbag Software. Order Reprints | Today's Paper | Subscribe

Software Development



Software Engineering
Software Development Processes
Software Development Effectiveness

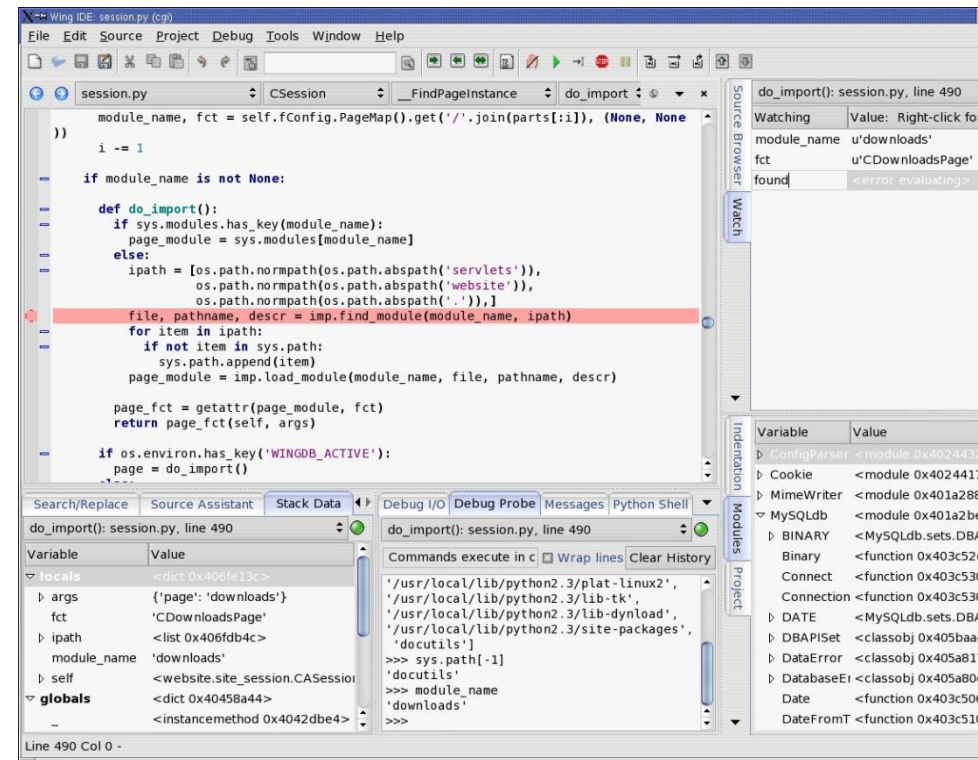


Software Engineering

- Is the study and application of engineering, design, development, and maintenance of software
- Systematic, disciplined, quantifiable approach
- Refers to the application of engineering to software
- State-of-the-art of developing quality software on time and within budget
- Multi-person construction of multi-version software (team-work, continuous development)

Software Engineering Environment

- hardware, software , and firmware used to perform a software engineering effort
- compilers, assemblers, interpreters, ...
- operating systems, libraries, frameworks
- debuggers, simulators, emulators, ...
- test tools, documentation tools, ...
- source code management systems
- integrated development systems (IDEs)



<http://deliveryimages.acm.org/10.1145/1160000/1152921/9138f3.jpg>



How the Senior Team described it



How the organization understood it



How the Integration Team designed it



How the Managers implemented it



How the Consultant described it



How the project was documented



What Operations installed



What it cost



How it was supported



What the organization really needed

Software development and its diverse stakeholder structure.

Software Development Process

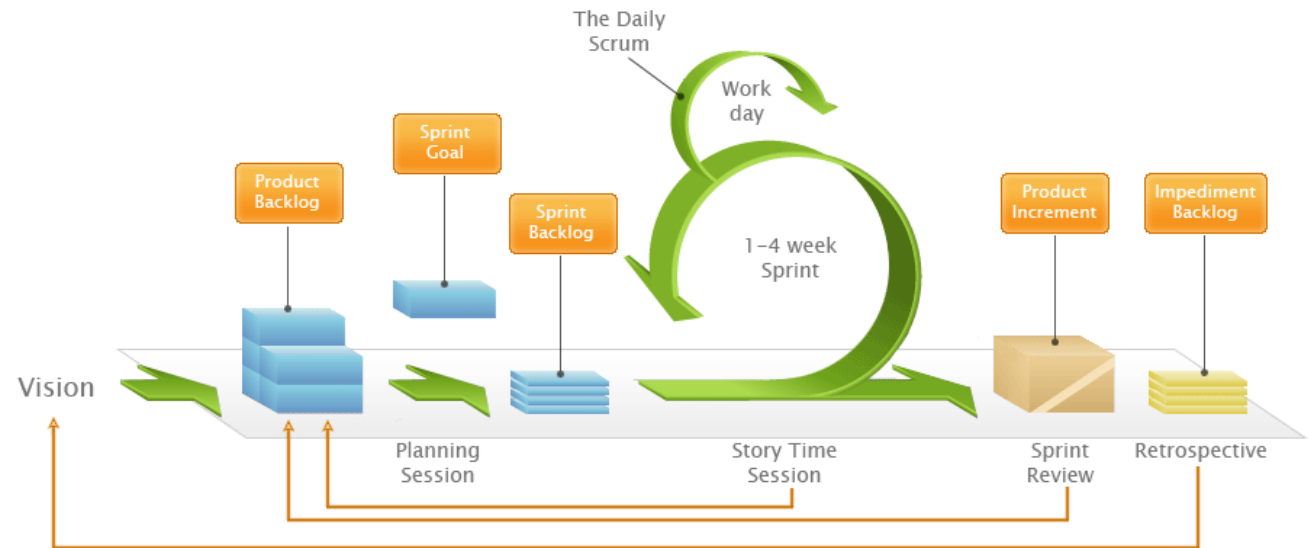
- Needs of users, clients, and customers are translated into a software product.
- Common translation steps include:
 - Requirement Analysis
 - Software Design
 - Implementation
 - Testing
 - Deployment
- Common methodologies include waterfall, prototyping, iterative and incremental development, spiral development, rapid application development, extreme programming, scrum, and agile software development.

Software Expressiveness

- From the same specification, commonly more than one functionally identical or equivalent implementations can be derived.
- Each variant typically differs with respect to run-time performance, errors, reliability, robustness, documentation, etc.
- There is no “unique” or “canonical” solution to most software development implementations.
- Programming languages and their use have a strong impact on how solutions are expressed.

Software Development Process

Scrum Process



blog.intera.si/wp-content/uploads/2011/10/scrum-sprint.gif

Software Development Process

- Process models define distinct stages (for scrum, e.g., sprint review, retrospective, etc.)
- Semi-formal definitions of process activities, inputs, outcomes, ...
- Software development process methodologies state what developers and other stakeholders *shall* do, i.e., they are *normative* approaches.

Effectiveness

- The degree to which **efforts spend in software development activities lead to actually positive consequences and results** in terms of software quality, software functionality, and software evolution.
- Key factor during the long-lasting maintenance phase of aging, complex, computer-based systems.
- Effectiveness depends, among other criteria, on how efficient program understanding and comprehension takes place.

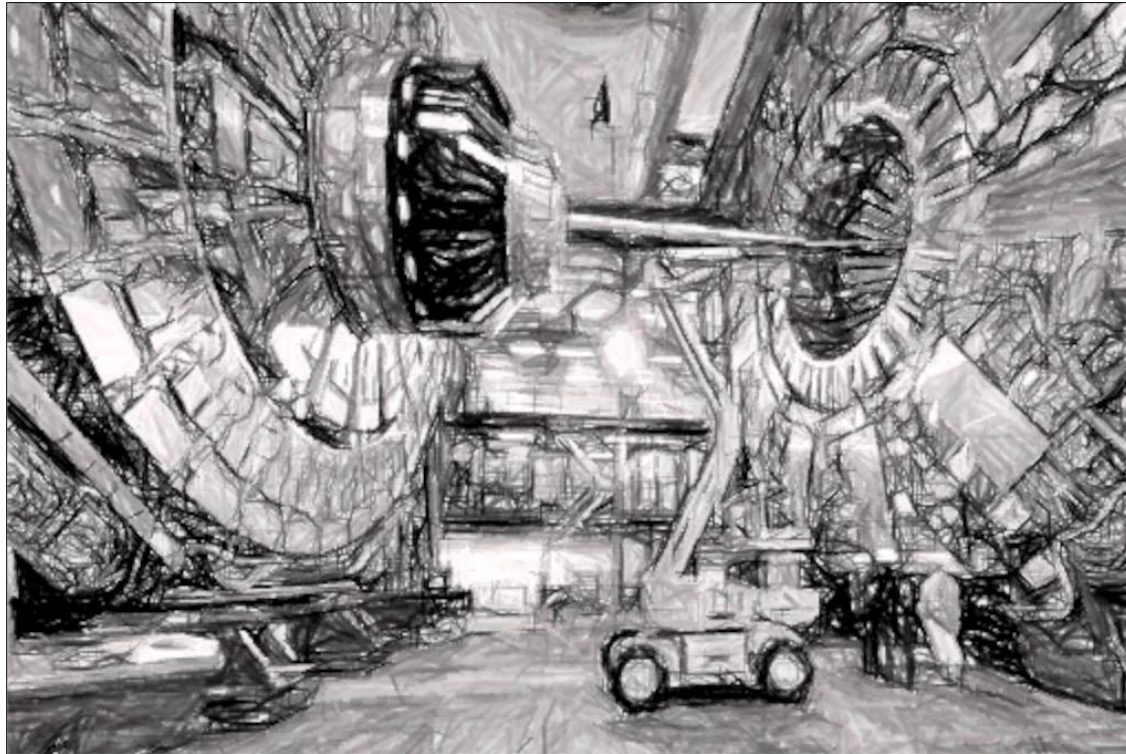
For all complex computer-based systems, a **software development process** methodology is required to set up, model, organize, and steer software development activities. The choice of a methodology depends on the type of system, a company's experience, regulations, or fashions, etc.

Software development process methodologies are not strictly formal nor can they be applied in perfection – developing software is and remains a **human activity**.

Software analytics is not bound to any specific software development process methodologies. It is a **non-normative approach**, that is, it analyzes, evaluates, and visualizes the as-is status and helps to identify trends and phenomena regardless of the methodology used to achieve that state.

Insofar, software analytics is a **complementary approach to methodologies**, which can be generally applied to all software development projects.

Software Development



Software Engineering
Software Development Processes
Software Development Effectiveness

Software Engineering

“The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.”

Software Development Process

“The process by which user needs are translated into a software product. The process involves translating user needs into software requirements, transforming the software requirements into design, implementing the design in code, testing the code, and sometimes, installing and checking out the software for operational use.”

Source: IEEE Standard Glossary of Software Engineering Terminology

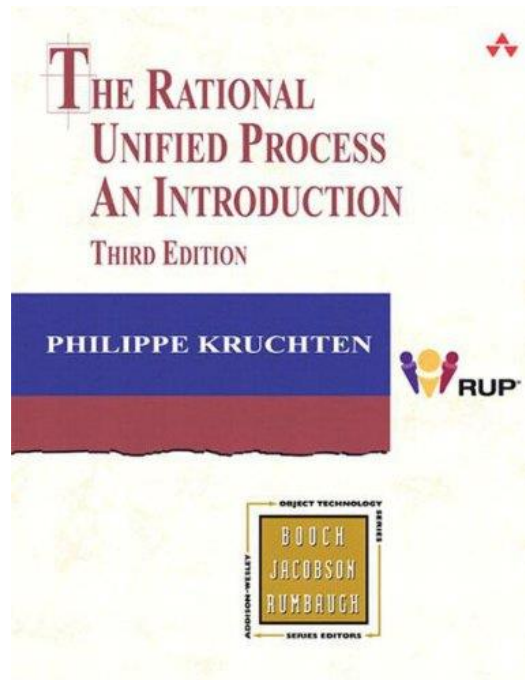
Software Development Effectiveness

The degree to which efforts spent in software development lead to actually wanted results (e.g., corrective, adaptive, preventive and perfective maintenance).

Source: Software Diagnostics Technology, Germany

Further Reading

Philippe Kruchten: *"The Rational Unified Process: An Introduction."*,
3rd Edition, Addison Wesley, 2003.

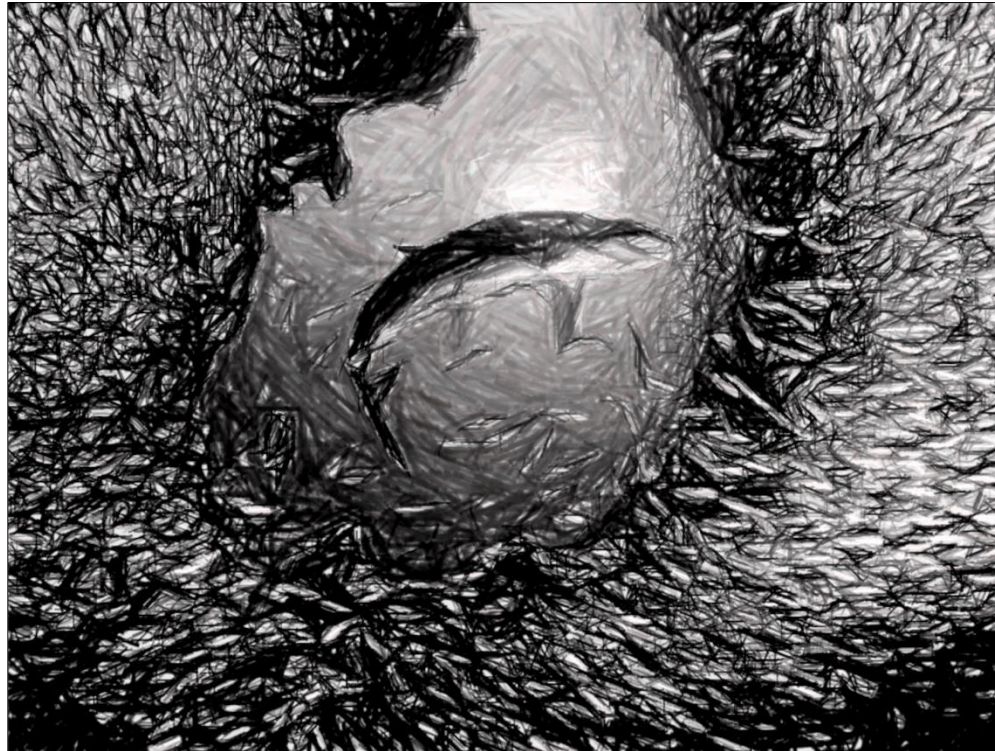


There is no winner in the arena of software development process methodology, although many developer teams today tend to use “dynamic” models such XP or Scrum. But: The methodology does not fully solve the issue of effectiveness for software maintenance.

One important point is that all developers should understand how the applied methodology works.

Regardless of the methodology, software analytics is a complementary instrument to check the real status and the as-is situation, to identify risks, challenges, etc. in advance.

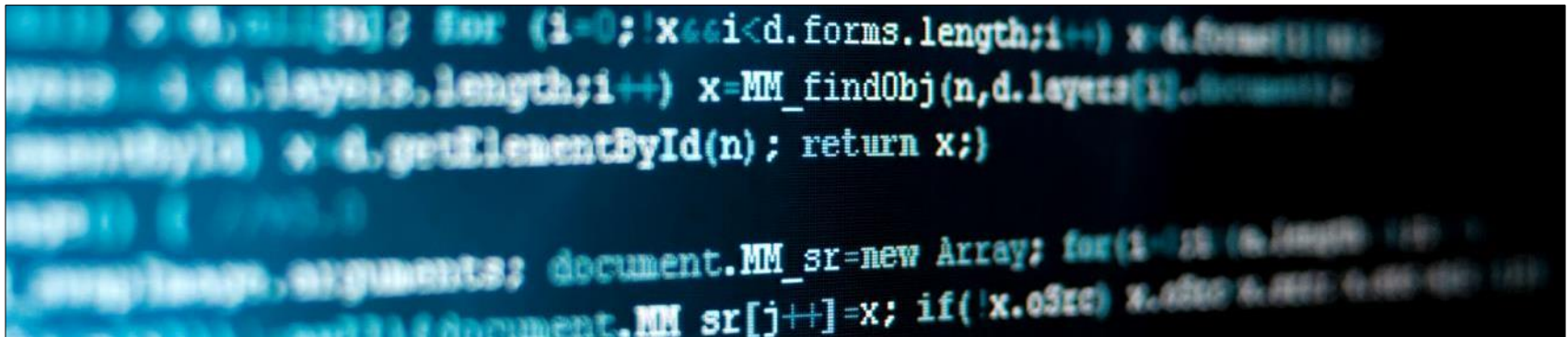
Software Complexity



Software
Software Complexity
Complex Software Systems

What is software?

- *"The entire set of programs, procedures, and related documentation associated with a system and especially a computer system; specifically : computer programs."* Merriam Webster
- Related terms: programs, applications, scripts, and instruction sets, ...
- Written and specified by formal languages
- Major parts are given by source codes



```
for (i=0; !x && i<d.forms.length; i++) x=d.forms[i];  
for (i=0; i<d.layers.length; i++) x=MM_findObj(n,d.layers[i].document);  
if (!x) x=d.getElementById(n); return x;  
...  
document.MM_sr=new Array; for (i=0; i<document.MM_sr.length; i++)  
document.MM_sr[i]=x; if (!x) document.MM_sr[i]=null;
```

hgdlawfirm.com

What is software?

- Key asset of complex computer-based systems
- Intellectual property
- Intangible and virtual

A blurred screenshot of JavaScript code, likely from a web browser's developer console. The code includes a loop that iterates over an array of forms, a function to find an object by ID, and a function to get an element by ID. The text is mostly illegible due to blurring but contains recognizable JavaScript syntax and variable names like 'd', 'forms', 'layers', 'MM_findObj', and 'MM_sr'.

hgdlawfirm.com



nerdist.com

Complexity versus Simplicity

"Complexity ... degree to which a system or component has a design or implementation that is difficult to understand and verify.

Contrast with: simplicity." IEEE IT Glossary

A **complex system** is constituted by a large number of interdependent, interrelated, and interacting elements forming a complex whole.

Software Complexity



nerdist.com

"The fundamental cause of the software crisis is that massive, software-intensive systems have become *unmanageably complex*." Grady Booch, 1994

"Software-intensive systems are perhaps the most intellectually *complex artifacts* created by humans." Grady Booch, 2007



anidea-engineering.com

Structure related characteristics

- Hierarchical structure (system of systems, systems, subsystems, ...) and “relative” primitives
- Large number of entities (structures, functions, classes, files, modules, ...)
- Large number of interactions between entities (coupling, dependencies, ...)
- Advanced use of computer languages (non-imperative languages, template-based programming, ...)
- ...



anidea-engineering.com

Behavior related characteristics

- System behavior during execution (call graph, control flow, ...)
- Nonlinear system behavior, e.g., due to multi-threading and concurrent execution
- Non-documented behavior
- Exception handling, error handling, ...
- ...



anidea-engineering.com

Process related characteristics

- Large number of developers
- Turnover of developers (fluctuation)
- Outsourcing and distributed teams
- Long-term development processes (age)
- Changing functional and non-functional requirements
- Non-consistent system documentation and models
- ...

Complex Software Systems



anidea-engineering.com

Problem-domain related characteristics

- Changing requirements
- Non-formal problem specifications
- Conflicting, inconsistent requirements
- Dependability of legacy systems
- ...

Complex Software Systems



anidea-engineering.com

Principle causes of software complexity:

Structure related: dimension and extent of software artifacts and their relations

Dynamics related: the problems of characterizing the behavior of discrete systems

Process related: the difficulty of managing a software development process

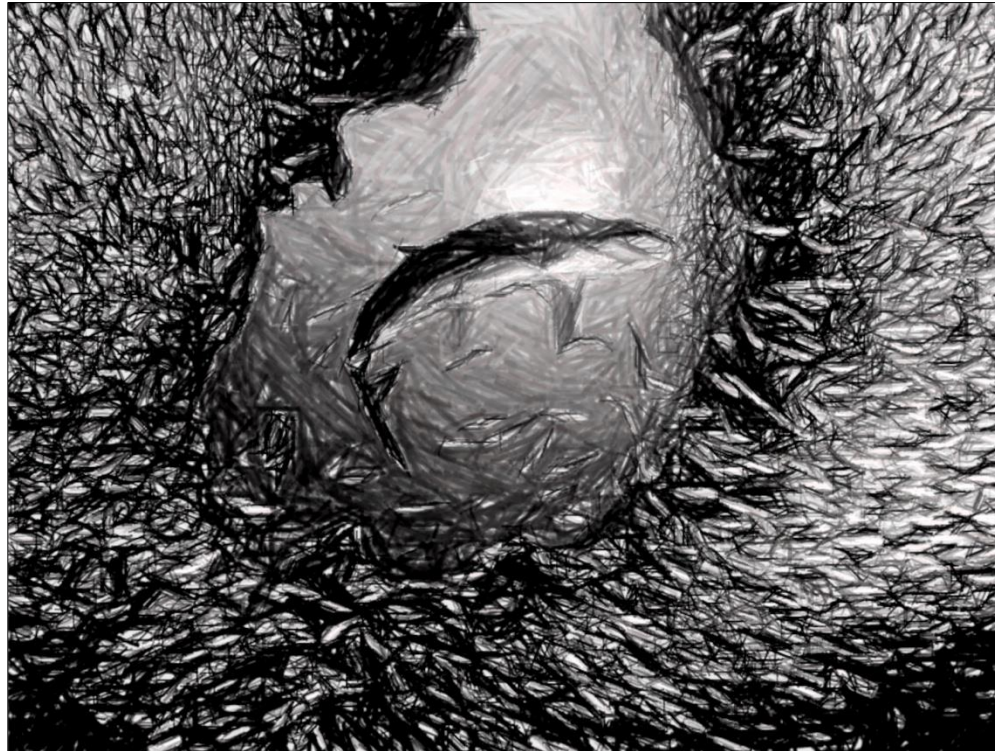
Domain related: the complexity of the problem domain

Software, by its nature, has no natural look, no gestalt, no appearance. It is intangible.

Complexity is caused by its structure, its dynamics, its evolution, and its problem domain.

Software analytics offers methods, techniques, and tools that facilitate efficient and effective software development of complex software, providing analytics and visualization of structural, behavioural, process-related, and domain-related aspects of their development and evolution.

Software Complexity



Software
Software Complexity
Complex Software Systems

Program

"A combination of computer instructions and data definitions that enable computer hardware to perform computational or control functions."

Source Code

"Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translators."

Software

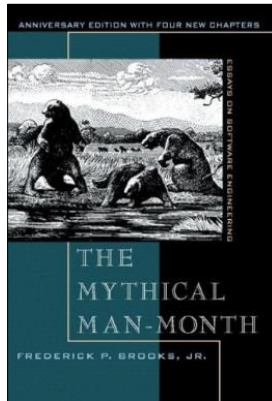
"Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system."

Sources: IEEE Standard Glossary of Software Engineering Terminology

Complex Software System

A software system that is complex with respect to its structural and hierarchical composition, its dynamics and behavior, its evolution and development, and the underlying problem domain.

Further Reading



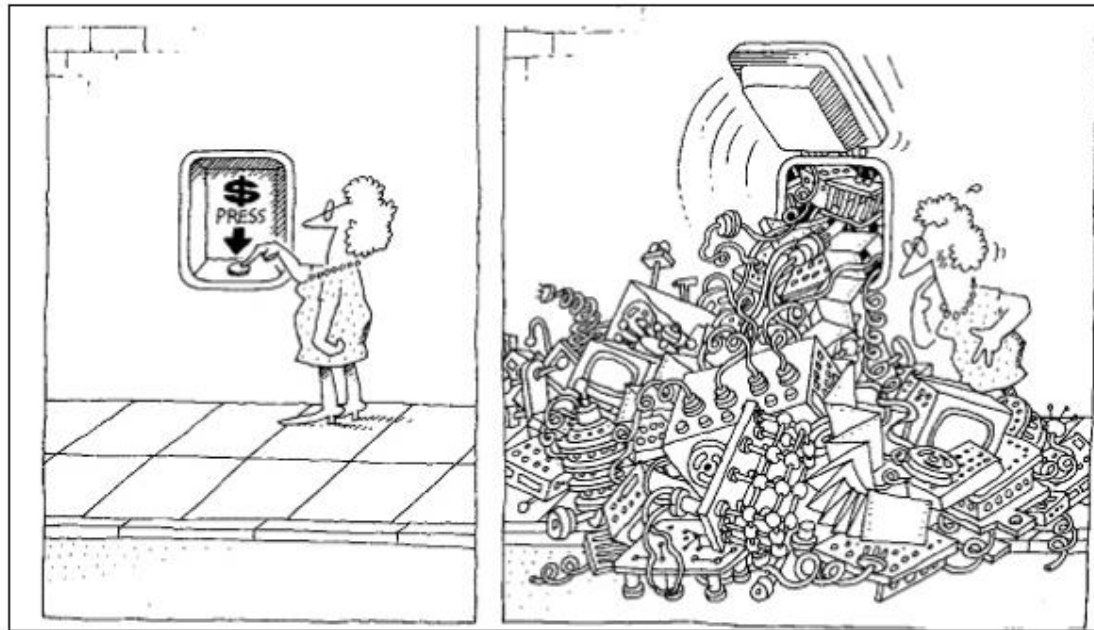
Frederic P. Brooks: "The Mythical Man-Month",
Addison-Wesley, 1995.

Bobbi J. Young, Grady Booch, Jim Conallen, Michael W. Engel,
Kelli A. Houston, Robert A. Maksimchuk:
"Software Complexity: How Do We Bring Order to Chaos?",
www.informit.com, Nov 30, 2007.
<http://www.informit.com/articles/article.aspx?p=726130>

If machines would design, construct, and deploy complex computer-based systems, complexity would possibly have different aspects compared to what is relevant for human developers ...

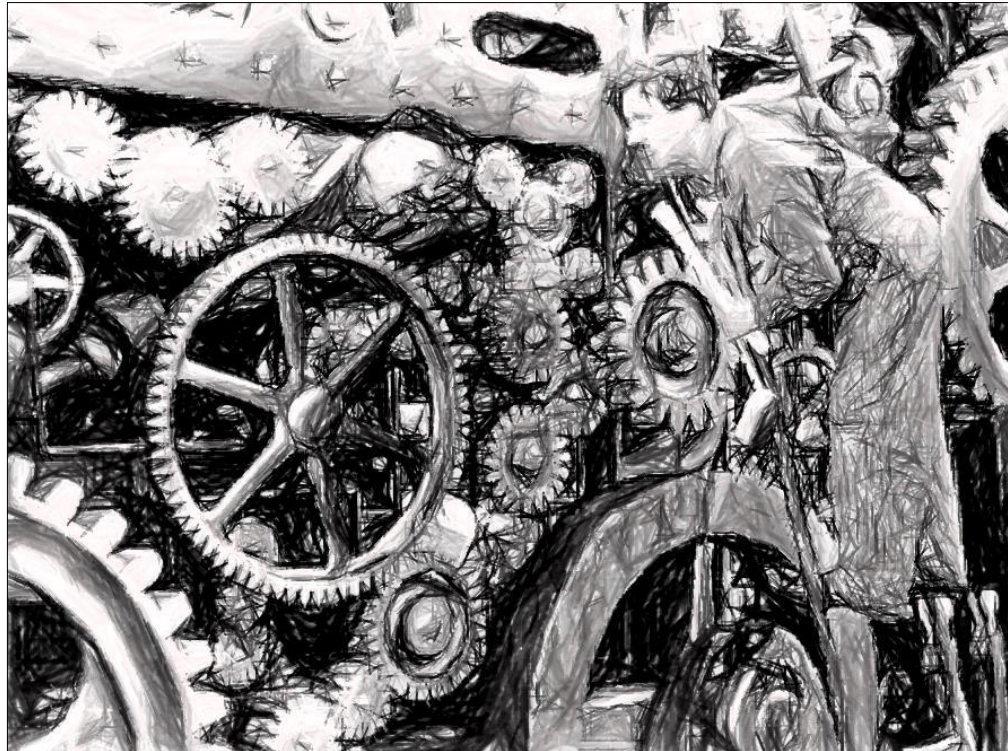
Developers commonly prefer to build new systems, with new tools, new structures, new architectures, new ideas (if they have the choice). During initial development, systems are generally not faced by complexity (small teams, small implementations, memorized models, clear concepts, ...)

"The complexity of software is an essential property, not an accidental one."



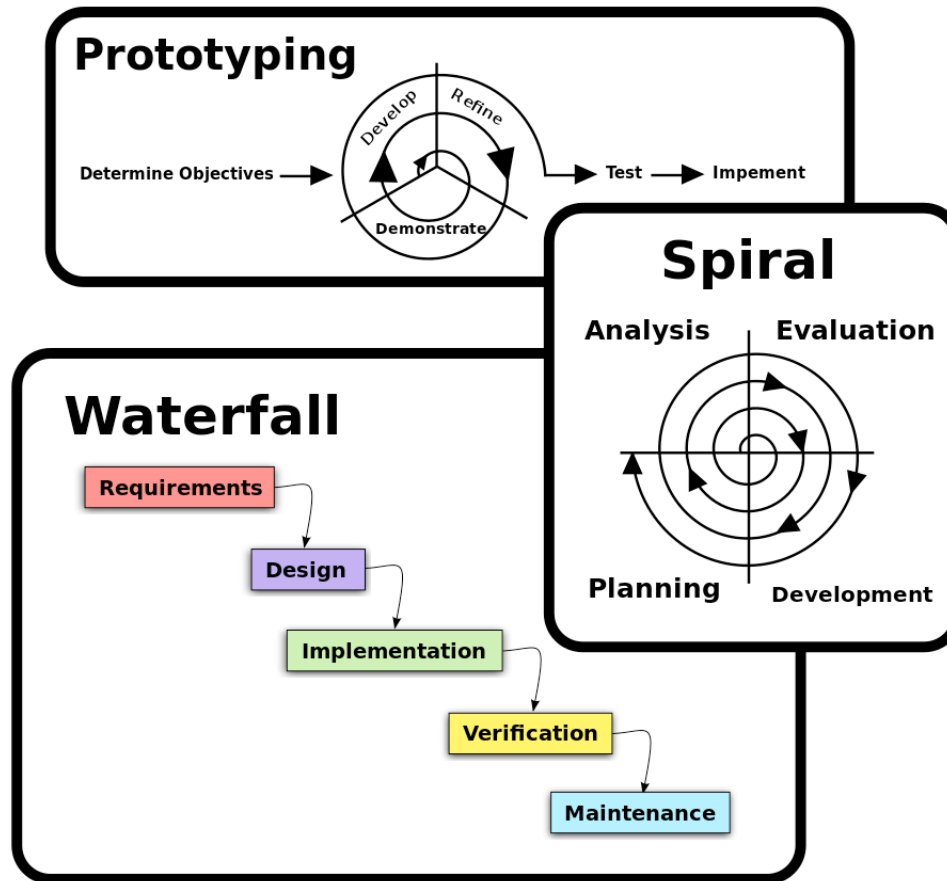
*The task of the software development team
is to engineer the illusion of simplicity.*

Software Maintenance



Software Life Cycle

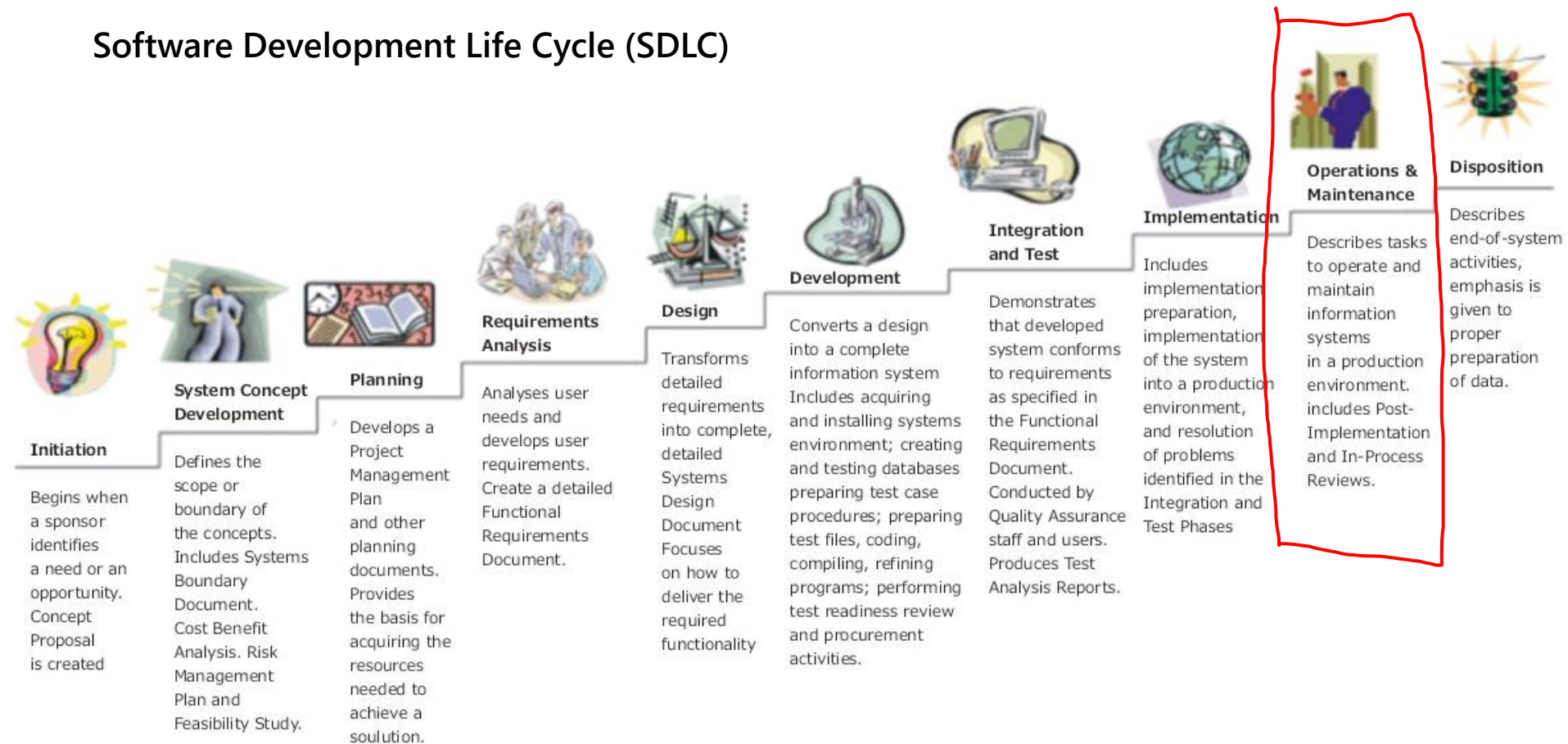
Adaptive, Corrective, Preventive, and Perfective Maintenance



Software Development Methodology Frameworks

Software Life Cycle

Software Development Life Cycle (SDLC)



http://en.wikipedia.org/wiki/Systems_development_life_cycle

IEEE Definition: *"any change to a system after it is operational"*

- **Preventive maintenance:**

Increasing maintainability or reliability to prevent problems in the future

- **Adaptive maintenance:**

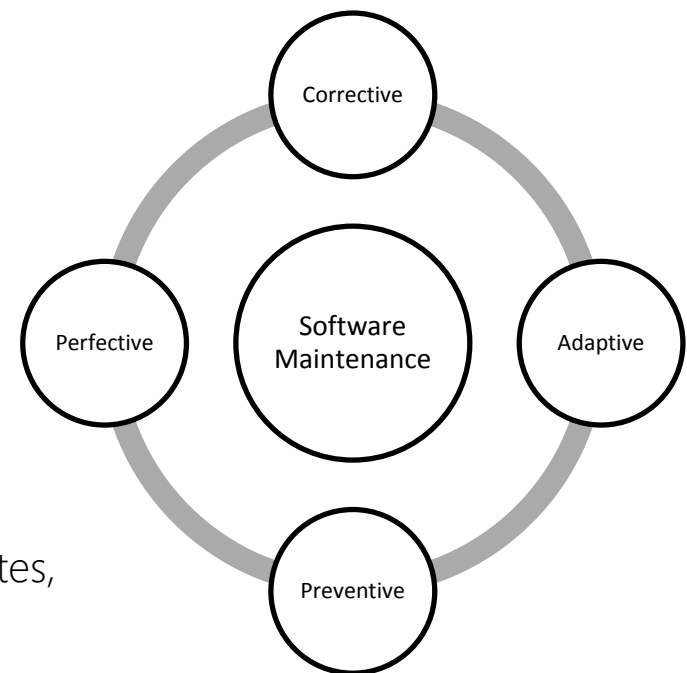
Modifying the system to cope with changing software environments, adding new functionality and adopting new requirements

- **Corrective maintenance:**

Correcting faults, diagnosing and fixing bugs, etc.

- **Perfective maintenance:**

Improve performance, maintainability, or other attributes, i.e., functional enhancements.



Software maintenance as a key factor (costs, time) in software development:

- Maintenance costs are increasingly growing
- Estimates showed that about 90% of software life cycle costs are related to its maintenance phase. Sayed Mehdi Hejazi Dehaghani and Nafiseh Hajrahimi, 2013
- In general, maintenance constitutes the largest part of software life cycle costs (more than two-thirds).
- About 60% of the maintenance costs are generally related to *program understanding and comprehension*.

Program understanding and comprehension

- a crucial aspect of software effectiveness
- a large issue for aging legacy systems
- a key target of software analytics

"Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. After many such changes the original designers no longer understand the product. Those who made the changes never did. In other words, nobody understands the modified product." David L. Parnas, 1994

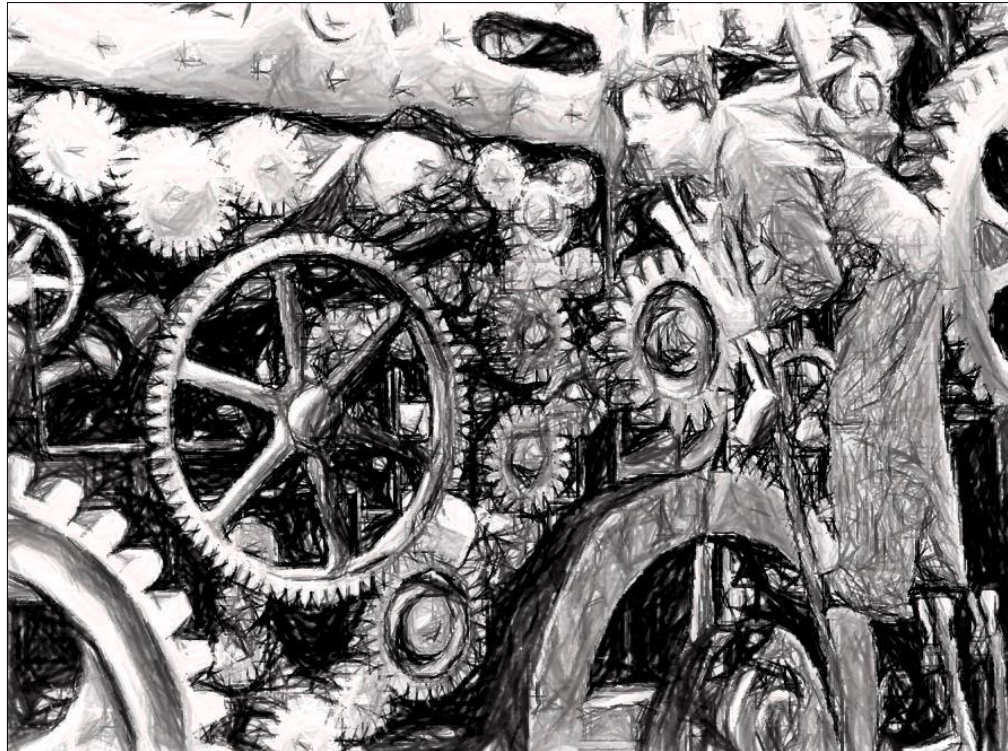
The **software life cycle** –independent from the concrete framework of software development methodology– is faced by the enduring "maintenance" phase.

Software development effectiveness takes effect during maintenance, that is, it is less relevant to the initial construction phase.

Software analytics is a key instrument and approach for the efficient management and handling of software maintenance.

Software analytics enables and simplifies program understanding and comprehension as required by almost all maintenance tasks.

Software Maintenance



Software Life Cycle

Adaptive, Corrective, Preventive, and Perfective Maintenance

Software Life Cycle

“The period of time that begins when a software product is conceived and ends when the software is no longer available for use. The software life cycle typically includes a concept phase, requirements phase, design phase, implementation phase, test phase, installation and checkout phase, operation and maintenance phase, and, sometimes, retirement phase.”

Software Product

“The complete set of computer programs, procedures, and possibly associated documentation and data designated for delivery to a user.”

Source: IEEE Standard Glossary of Software Engineering Terminology

Software Maintenance

“(1) The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment. [...]

(2) The process of retaining a hardware system or component in, or restoring it to, a state in which it can perform its required functions. See also: preventive maintenance.”

Source: IEEE Standard Glossary of Software Engineering Terminology

Preventive Maintenance

Software maintenance that aims at increasing maintainability or reliability to prevent problems in the future.

Adaptive Maintenance

Software maintenance that aims at modifying the system to cope with changing software environments, adding new functionality and adopting new requirements

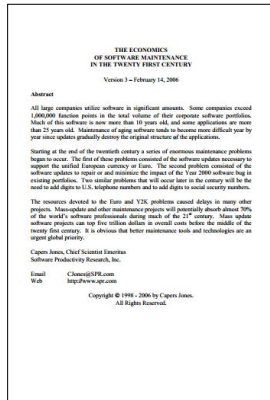
Corrective Maintenance

Software maintenance that aims at correcting faults, diagnosing and fixing bugs, etc.

Perfective Maintenance

Software maintenance that aims at improve performance, maintainability, or other attributes, i.e., functional enhancements.

Further Reading



Capers Jones: "The Economics Of Software Maintenance In The Twenty First Century" (PDF), 2006.

<http://www.compaid.com/caiinternet/ezone/capersjones-maintenance.pdf>

The term "maintenance" is ambiguous and has a number of connotations that sometimes hide the fact that it represents the major time span in the life of a complex computer-based system.

Developers commonly prefer to build new systems (with new tools, new structures, new architecture, new programming language features, new libraries, new coding styles, ...) – if they have the choice.

There are more people already modifying existing computer-based systems than writing new ones.

Tools and techniques for efficient maintenance become more and more key to industry – like software analytics.

```
c = getchar ();  
if (c == 'x') return 0;
```

Source Code
Static Source Code Analysis
Tools

```
QModelIndex start;  
if (currentIndex().isValid())  
    start = currentIndex();  
else  
    start = d->model->index(0, 0, d->root);  
  
bool skipRow = false;  
bool keyboardTimeWasValid = d->keyboardInputTime.isValid();  
qint64 keyboardInputTimeElapsed = d->keyboardInputTime.restart();  
if (search.isEmpty() || !keyboardTimeWasValid  
    || keyboardInputTimeElapsed > QApplication::keyboardInputInterval()) {  
    d->keyboardInput = search;  
    skipRow = currentIndex().isValid();  
} else {  
    d->keyboardInput += search;  
}
```

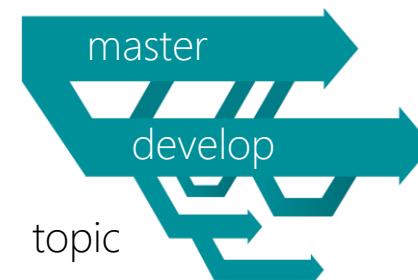
Woboq Codebrowser - <http://woboq.com/blog/codebrowser-introduction.html>

- Formal and complete description of a program that is automatically translated or interpreted by programs or machines (e.g., compilers, interpreters).
- Source code generally is written by (or at least mainly set up) by human developers (sometimes by automated code generation).



Version Control Systems (also: Revision Control Systems, Repositories)

- Allow (large) teams to collaborate on the implementation of complex software systems
- Record any changes made to a system's implementation by an individual developer
- Central repository that manages file updates, change annotations, and file merging
- Examples:
 - CVS: Concurrent Versions System
 - SVN: Apache Subversion System
 - Git: distributed version control system (Linux)



Principles of Static Code Analysis

- Aims at verifications that ensure coding quality and reliability
- Analyzes the source code or binary code
without running the system or considering a specific input
- Checks if reasonable, recommended or common programming practices are violated
- Represents a “white box” testing method (considering the implementation, ignoring the functionality)
- Most common static source code analysis “tool”: **compilers**,
which check source code as core part of their functionality:

```
$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
    return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                           ~~~~~^~~~~
```

Goals of static source code analysis: Find and identify

- unused or unreachable code segments
- “uncommon” usage of language
- error-prone statements and bug-suspect code segments
 - uninitialized variables
 - unused functions and methods
 - assignments in conditions
 - fall-through in case statements
 - ...
- conformance to coding guidelines and formatting rules
- non-portable language usage
- ...

Example C program:

Splint's results (www.splint.org):

```
01 #include <stdio.h>
02
03 int main()
04 {
05     char c;
06     while (c != 'x');
07     {
08         c = getchar();
09         if (c = 'x')
10             return 0;
11
12         switch (c)
13         {
14             case '\n':
15             case '\r':
16                 printf("line break\n");
17         }
18         return 0;
19 }
```

Variable c used before definition

Suspected infinite loop.

No value used in loop test (c) is modified by test or loop body.

Assignment of int to char: c = getchar()

Test expression for if is assignment expression: c = 'x'

Test expression for if not boolean, type char: c = 'x'

Fall through case (no preceding break)

Example C program:

→ Revised C program:

```
01 #include <stdio.h>
02
03 int main()
04 {
05     char c;
06     while (c != 'x');
07     {
08         c = getchar();
09         if (c == 'x')
10             return 0;
11
12         switch (c)
13         {
14             case '\n':
15             case '\r':
16                 printf("line break\n");
17         }
18     }
19 }
```

```
01 #include <stdio.h>
02
03 int main()
04 {
05     int c = 0;
06     while (c != 'x');
07     {
08         c = getchar();
09         if (c == 'x')
10             return 0;
11
12         switch (c)
13         {
14             case '\n':
15             case '\r':
16                 printf("line break\n"); break;
17             default:
18                 printf("%c", c); break;
19         }
20     }
21 }
```

Example: **GNU C++ compiler flags** for static source code analysis:

- Compatibility to language standards
 - "-std=c90" ... "-std=c++11"
- Warnings
 - "-Wall"
 - "-Wunused-parameter"
 - "-Wctor-dtor-privacy"
 - "-Wdelete-non-virtual-dtor"
 - "-Wnarrowing"
 - "-Wreorder"
 - "-Wffc++"
 - "-Wold-style-cast"
 - ...

-Wall turns on the following warning flags:

- Waddress
- Warray-bounds=1 (only with -O2)
- Wc++11-compat
- Wc++14-compat
- Wchar-subscripts
- Wenum-compare (in C/ObjC; this is on by default in C++)
- Wimplicit-int (C and Objective-C only)
- Wimplicit-function-declaration (C and Objective-C only)
- Wcomment
- Wformat
- Wmain (only for C/ObjC and unless -ffreestanding)
- Wmaybe-uninitialized
- Wmissing-braces (only for C/ObjC)
- Wnonnull
- Wopenmp-simd
- Wparentheses
- Wpointer-sign
- Wreorder
- Wreturn-type
- Wsequence-point
- Wsign-compare (only in C++)
- Wstrict-aliasing
- Wstrict-overflow=1
- Wswitch -Wtrigraphs
- Wuninitialized
- Wunknown-pragmas
- Wunused-function
- Wunused-label
- Wunused-value
- Wunused-variable
- Wvolatile-register-var

Static analysis is faced by a number of weaknesses:

- Too many *useless spurious* warnings and errors are reported; developers start to ignore the outputs and results (“developer confusion” and false positives)
- Tools take too long to check the source code basis and, therefore, their application is skipped.
- Static source code analysis are strong for all compiler-based languages but for dynamic languages (Ruby, Python, Perl etc.) it is much more difficult to find reasonably deep checking levels (remember: compilers actually are the first and most important static source code “checkers”).

C/C++ Tools (excerpt)

- **Splint** – the successor of the original “lint” tool www.splint.org
- **Cpplint** – Google checker for C++ coding styles sourceforge.net/projects/cpplint
- **Cppcheck** – “a tool for static C/C++ analysis” cppcheck.sourceforge.net
- **Coverity** – online service, surfacing quality and security defects scan.coverity.com
- ...

Java Tools (excerpt)

- **FindBugs** – a static Java source code analysis tool findbugs.sourceforge.net
- **Checkstyle** – help to write code that adheres to a coding standard checkstyle.sourceforge.net
- **PMD** – finds common programming flaws pmd.sourceforge.net
- ...

Static source code analysis is an effective way of improving software quality and reliability, in particular, if applied regularly and if complemented by project or company specific coding conventions and guidelines. It analyzes and evaluates the use of language elements and identifies potentially dangerous codings.

Software analytics uses static source code analysis as one key data source to analyze, evaluate, and visualize the as-is status of a software system. It provides base measures and metrics information about the implementation.

Static source code analysis contributes to the information used by software analytics.

```
c = getchar ();  
if (c == 'x') return 0;
```

Source Code
Static Source Code Analysis
Tools

Source Code

“Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator.”

Statement

“In a programming language, a meaningful expression that defines data, specifies program actions, or directs the assembler or compiler.”

Object code

“Computer instructions and data definitions in a form output by an assembler or compiler. An object program is made up of object code.”

Static analysis

“The process of evaluating a system or component based on its form, structure, content, or documentation.”

Sources: IEEE IT Glossary



Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, John Penix: "Using Static Analysis to Find Bugs", IEEE Software, Sep./Oct. 2008, 22-29. (Reports on how "FindBugs" work, a Java static source code analysis tools.)

Syntax highlighting, context-sensitive code editors, and integrated development environments (IDE) provide a number of static code analysis functionality by now. Make sure to turn them on and to read their results.

Today's C and C++ compilers perform a broad range of static code analysis (as built-in functionality). Look at the exact list of options and configuration flags of your compiler. For example, in certain intervals, a software project could be compiled and built with "strict" compiler options (see for example the GNU compiler warning flags).