Contents

Chapter 1

Presentation of the research topic

The domain of the proposed thesis is Automated Software Engineering. The thesis will develop methods for the analysis of legacy software systems, focusing on using historical information describing the evolution of the systems extracted from the versioning systems. The methods for analysis will integrate techniques based on computational algorithms as well as data-mining. As proof-of-concept, tool prototypes will implement the proposed methods and validate them by extensive experimentation on several cases of real-life systems.

Chapter 2

Software evolution and maintainability

2.1 Software evolution

2.1.1 Stages

Software has distinctive stages during its life:

Initial development In this stage, the first functional version of the system is developed. Also, many of the main features are developed and documented and, the architecture is defined: components that compose the system, their interactions, and properties.

In this stage, the architecture is defined and must be made in such a manner that the software will easily support new additions. Changes made in the evolution stages must change very little the architecture defined in the initial stage [22]. The architecture must:

- allow unanticipated change in the software without compromising system integrity.
- evolve in controlled ways. [23]

Evolution The evolution stage begins only after the initial stage is passed successfully. In this stage, iterative changes are made. By changes, we mean additions (new software features), modifications (changes of requirements or misunderstood requirements) or deletions. There are two main reasons for the change: the learning process of the software team and new requests from the client.

Servicing - Like we mentioned in the description of the initial stage, the architecture must be made in such a manner that the software may easily embrace changes. But, if not, then new changes are no longer easy to be made, are very difficult and time-consuming. At this point, the software enters the servicing stage (also called aging software, decayed software, and legacy) [26], [6].

Phase out - This stage is also called decline [26]. If in the servicing stage limited changes are implemented, in the phase-out stage the entire development is frozen. In this way the software becomes outdated. The users must make workarounds in order to cover the deficiencies of the software.

Close down - In this stage, the software is completely shut down, and the users redirected to a replacement system if this exists.

The stages presented above are represented in Figure 2.1.1.

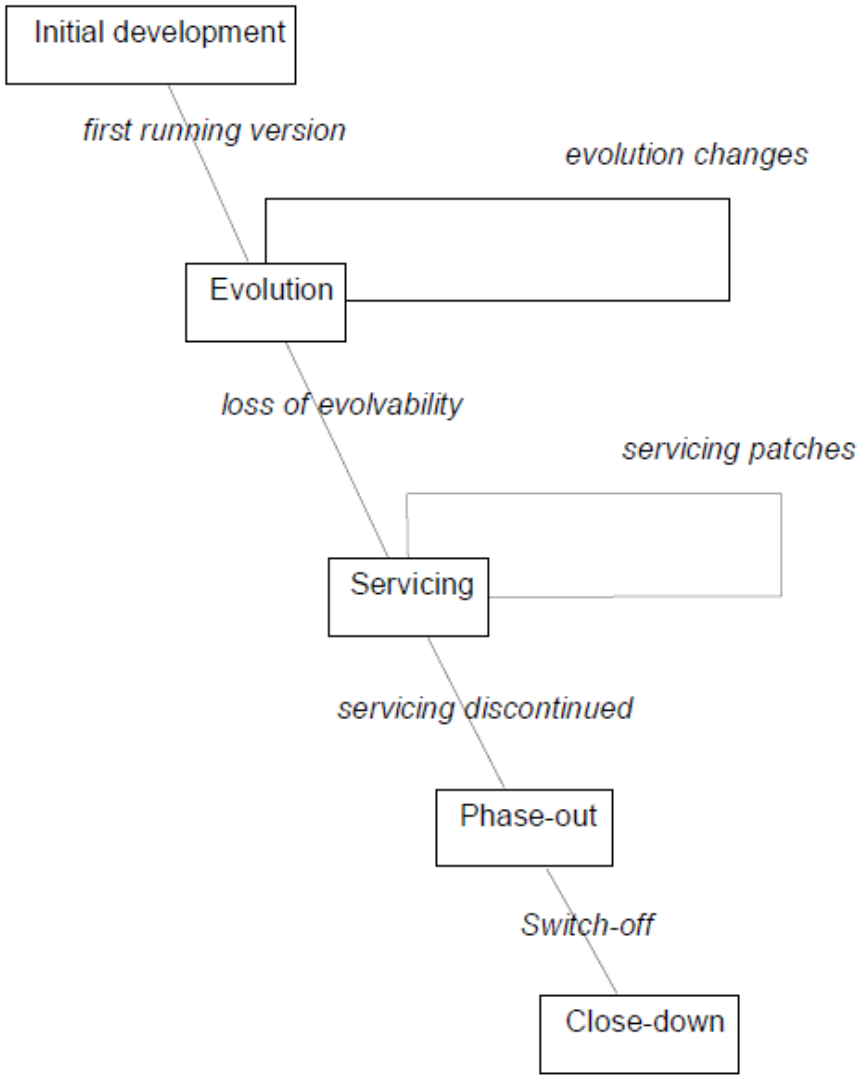


Figure 2-1: The versioned staged model [22]

2.1.2 Software change

As mentioned in section 2.1.1, software change in made in evolution and servicing stages. The difference between changes made in the evolution stage and changes made in the servicing stage is the effort of making changes. In the evolution stage, software changes are made easily and do not require much effort while in the servicing stage only a limited number of changes are made and require a lot of effort, so are really time-consuming [23], [35].

The change is triggered either by new feature requests from the client or by misunderstanding the previous requirements and re-implementing them.

The change mini-cycle consists of the following phases [48]:

- Phase 1: The request for change. This usually comes from the users of the software and it can also be a bug report or a request for an additional functionality.
- Phase 2: The planning phase, this includes program comprehension and change impact analysis. Program comprehension is a mandatory prerequisite of the change while change impact analysis indicates how costly the change is going to be. [1]
- Phase 3: The change implementation, restructuring for change and change propagation.
- Phase 4: Verification and validation.
- Phase 5: Re-documentation.

2.2 Software maintenance

In early days software maintenance was a small part of the software life cycle. As time was passing and more software was created, people realized that software does not die. Even if the actual development of the software may be frozen, requests regarding bug fixing and compatibility with new operating systems may appear over time. Sometimes software maintenance requires more effort than to build the system from scratch [46], [20].

Lientz and Swanson categorized maintenance activities into four classes [28]:

- Adaptive maintenance: changes in the software environment.
- Perfective maintenance: new user requirements and documentation improvement.
- Corrective maintenance: debugging and bug fixing.
- Preventive maintenance: prevent future problems.

They also made a survey on the problems of application software maintenance in 487 organizations. The survey showed that most of the maintenance effort was on the first two types (adaptive and perfective maintenance). Many other studies suggest similar results [23]. All these studies underline one thing: *the incorporation of new user requirements is the main problem for software evolution and maintenance* [28], [10].

One may ask why maintenance requires so much effort? There are some main problem factors when it comes to maintenance:

- User knowledge: lack of user training and/or lack of understanding of the system.

- Programmer effectiveness: lack of skills or experience for maintenance, development.
- Product quality: quality of the original system, quality of documentation and specs.
- Machine requirements: increasing storage or decreasing processing time requirements.
- System reliability: data integrity, hardware, and software reliability.

Chapter 3

Dependencies in software systems

A dependency is created by two elements that are in a relationship and indicates that an element of the relationship, in some manner, depends on the other element of the relationship [9], [12].

A developer gains information about a software system by looking at it in two different ways:

- top-down: starting from the highest level of abstraction of the system and going down to the code.
- bottom-up: starting from the code and going up to the highest level of abstraction.

Both ways imply gradually gaining new information about the system and in both, the developer traces the dependencies and connections of the software system [45], [29].

3.1 Structural dependencies

Structural dependencies can be found by analyzing the source code. [38]. There are several types of relationships between these source code entities and all those create *structural dependencies*:

Data Item Dependencies

Data items can be variables, records or structures. A dependency is created between two data items when the value held in the first data item is used or affects the value from the second.

Data Type Dependencies

Data items are declared to be of a specific data type. Besides the built-in data types that every programming language has, developers can also create new types that they can use. Each time the data type definition is changed it will affect all the data items that are declared to be of that type.

Subprogram Dependencies

A subprogram is a sequence of instructions that performs a certain task. Depending on the programming language a subprogram may also be called a routine, a method, a function or a procedure. When a subprogram is changed, the developer must check the effects of that change in all the places that are calling that subprogram. Subprograms may also have dependencies with the data items that they receive as input or the data items that they are computing.

3.2 Logical dependencies

3.2.1 Definition

3.2.2 Current status of research

The current trend recommends that general dependency management methods and tools should also include logical dependencies besides the structural dependencies [32], [3].

Software engineering practice has shown that sometimes modules which do not present structural dependencies still appear to be related. Co-evolution represents the phenomenon when one component changes in response to a change in another component [49]. Those changes can be found in the software history maintained by the versioning system. Gall [21] identified as logical coupling between two modules the fact that these modules *repeatedly* change together during the historical evolution of the software system. *Logical dependencies* (a.k.a logical coupling) can be found by software history analysis and can reveal relationships that are not always present in the source code (structural dependencies).

The concepts of logical coupling and logical dependencies were first used in different analysis tasks, all related to changes: for software change impact analysis [36], for identifying the potential ripple effects caused by software changes during software maintenance and evolution [33], [32], [34], [24] or for their link to deffects [44], [50].

Different applications based on dependency analysis could be improved if, beyond structural dependencies, they also take into account the hidden non-structural dependencies. For example, works which investigate different methods for architectural reconstruction [17], [15], [16], all of them based on the information provided

by structural dependencies, could enrich their dependency models by taking into account also logical dependencies. However, a thorough survey [18] shows that historical information has been rarely used in architectural reconstruction.

Another survey [40] mentions one possible explanation why historical information have been rarely used in architectural reconstruction: the size of the extracted information. One problem is the size of the extraction process, which has to analyze many versions from the historical evolution of the system. Another problem is the big number of pairs of classes which record co-changes and how they relate to the number of pairs of classes with structural dependencies.

3.2.3 Extracting logical dependencies from co-changes

The software architecture is important in order to understand and maintain a system. Often code updates are made without checking or updating the architecture. This kind of updates cause the architecture to drift from the reality of the code over time.[18] So reconstructing the architecture and verifying if still matches the reality is important.

Surveys show that architectural reconstruction is mainly made based on structural dependencies [40] [18], the main reason why historical information is rarely used in architectural reconstruction is the size of the extracted information.

Logical dependencies should integrate harmoniously with structural dependencies in an unitary dependency model: valid logical dependencies should not be omitted from the dependency model, but structural dependencies should not be engulfed by questionable logical dependencies generated by casual co-changes. Thus, in order to add logical dependencies besides structural dependencies in dependency models, class co-changes must be filtered until they remain only a reduced but relevant set

of valid logical dependencies.

Currently there is no set of rules or best practices that can be applied to the extracted class co-changes and can guarantee their filtering into a set of valid logical dependencies. This is mainly because not all the updates made in the versioning system are code related. For example a commit that has as participants a big number of files can indicate that a merge with another branch or a folder renaming has been made. In this case, a series of irrelevant co-changing pairs of entities can be introduced. So, in order to exclude this kind of situations the information extracted from the versioning system has to be filtered first and then used.

Other works have tried to filter co-changes [32], [3]. One of the used co-changes filter is the commit size.The commit size is the number of code files changed in that particular commit. Ajienka and Capiluppi established a threshold of 10 for the maximum accepted size for a commit [3]. This means that all the commits that had more than 10 code files changed where discarded from the research. But setting a harcoded threshold for the commit size is debatable because in order to say that a commit is big or small you have to look first at the size of the system and at the trends from the versioning system. Even thought the best practices encourage small and often commits, the developers culture is the one that influences the most the trending size of commits from one system.

Filtering only after commit size is not enough, this type of filtering can indeed have an impact on the total number of extracted co-changes, but will only shrink the number of co-changes extracted without actually guaranteeing that the remaining ones have more relevancy and are more logical linked.

Although, some unrelated files can be updated by human error in small commits, for example: one file was forgot to be commited in the current commit and will be commited in the next one among some unrelated files. This kind of situation can

introduce a set of co-changing pairs that are definitely not logical liked. In order to avoid this kind of situation a filter for the occurrence rate of co-changing pairs must be introduced. Co-changing pairs that occur multiple times are more prone to be logically dependent than the ones that occur only once. Currently there are no concrete examples of how the threshold for this type of filter can be calculated. In order to do that, incrementing the threshold by a certain step will be the start and then studying the impact on the remaining co-changing pairs for different systems.

Taking into account also structural dependencies from all the revisions of the system was not made in previous works, this step is important in order to filter out the old, out-of-date logical dependencies. Some logical dependencies may have been also structural in previous revisions of the system but not in the current one. If we take into consideration also structural dependencies from previous revisions then the overlapping rate between logical and structural dependencies could probably increase. Another way to investigate this problem could be to study the trend of concurrences of co-changes: if co-changes between a pair of classes used to happen more often in the remote past than in the more recent past, it may be a sign that the problem causing the logical coupling has been removed in the mean time.

Also, logical dependency can be also a structural dependency and vice-versa, so studying the overlapping between logical and structural dependencies while filtering is important since the intention is to introduce those logical dependencies among with structural dependencies in architectural reconstruction systems. Current studies have shown a relatively small percentage of overlapping between them with and without any kind of filtering [3]. This means that a lot of non related entities update together in the versioning system, the goal here is to establish the factors that determine such a small percentage of overlapping.

Filtering after number of occurrences

Chapter 4

Applications of software dependencies

Examples of areas where dependencies are used:

4.1 Architecture reconstruction

The term reverse engineering was first defined by Chikofsky and Cross [13] as the *"process of analyzing a system to (i) identify the systems components and their interrelationships and (ii) create representations of the system in another form or at a higher level of abstraction."*

Reverse engineering is viewed as a two step process: information extraction and abstraction. [11] The firs step, information extraction, is made by source code analysis which generates dependencies between software artifacts. So, reverse engineering uses dependencies in order to create new representations of the system or provide a higher level of abstraction. [8]

Currently, the software systems contain tens of thousands of lines of code and are updated multiple times a day by multiple developers. The software architecture is important in order to understand and maintain a system. Often code updates are made without checking or updating the architecture. This kind of updates cause the architecture to drift from the reality of the code over time. So reconstructing the architecture and verifying if still matches the reality is important. [18],[16], [4] ,[39], citemodel-bennett.

4.2 Identifying clones

Research suggests that a considerable fraction (5-10%) of the source code of large-scale software is duplicate code (clones). Source code is often duplicated in part, or in whole, for a variety of reasons. Programmers may simply be reusing a piece of code by copy and paste or they may be reinventing the wheel [31], [30]. Detection and removal of clones can decrease software maintenance costs [5], [25].

4.3 Code smells

Code smells are a metaphor to describe patterns that are generally associated with bad design and bad programming practices. Originally, code smells are used to find the places in software that could benefit from refactoring [43]. Examples of code smells are: large class: classes with many fields, feature envy for methods that access more methods and fields of another class than of its own class, switch statements where inheritance should be used for specialization, data class for classes that do not contain functionality, only fields, refused bequest for classes that leave many of the fields and methods they inherit unused, and data clumps for clusters of data that are

often seen together as class members or in method signatures but are not grouped in a class [19].

4.4 Comprehension

Program comprehension is the process of acquiring knowledge about a computer program. Increased knowledge enables such activities as bug correction, enhancement, reuse, and documentation [37]. Program comprehension is an emerging interest area within the software engineering field. Software engineering itself is concerned with improving the productivity of the software development process and the quality of the systems it produces. However, as currently practiced, the majority of the software development effort is spent on maintaining existing systems rather than developing new ones. Estimates of the proportion of resources and time devoted to maintenance range from 50% to 75% [27]. The greatest part of the software maintenance process, in turn, is devoted to understanding the system being maintained. These involve reading the documentation, scanning the source code, and understanding the changes to be made.

4.5 Fault location

Debugging software is an expensive and mostly manual process. Of all debugging activities, locating the faults, or fault localization, is the most expensive [14]. This expense occurs in both the time and the cost required to find the fault. Because of this high cost, any improvement in the process of finding faults can greatly decrease the cost of debugging.[2] [14]

4.6 Program evolution

[7], [47]

4.7 Empirical software engineering research

Empirical research seeks to explore, describe, predict, and explain natural, social, or cognitive phenomena by using evidence based on observation or experience. It involves obtaining and interpreting evidence by, e.g., experimentation, systematic observation, interviews or surveys, or by the careful examination of documents or artifacts. [41]

Chapter 5

Problems in Dependency Tracing

Chapter 6

Research content and stages of research

6.1 Proposed research stages

The research will be made by following the next stages of implementation:

A. DEVELOPMENT OF CONTENT AND TOOLS

Stage 1: Build tool to extract structural dependencies from code and co-changes from git for a given set of projects.

Stage 2: Find filters for the co-changes extracted, the filters can be the ones already mentioned in previous works or new ones. Establish different thresholds for those filters.

Stage 3: Study the impact of those filters and the coresponding thresholds on the remaining quantity of co-changes for each system. Study the overlappings between the remaining pairs of co-changing entites and the structural dependencies extracted. [42]

Stage 4: Establish a dynamic way to determine the thresholds for filters in order to fit the best each studied sistem. Main focus on the threshold for number of occurences of co-changing pairs. Use plots or other visual instruments in order to see the highests and the lowest rates for the numbers of occurrences among co-changing pairs. Also filter those rates into normal and abnormal ones and study what was the cause of the highest rates (code or human related).

Stage 5: Take into account also structural dependencies from all the revisions of the system to filter out the old, out-of-date logical dependencies. Study how this affects the remaining number of logical dependencies.Here an extra check is needed, it can be a case in which old structural dependencies that were also logically linked to continue to be logically linked even after the structural dependency was removed.

B. USAGE

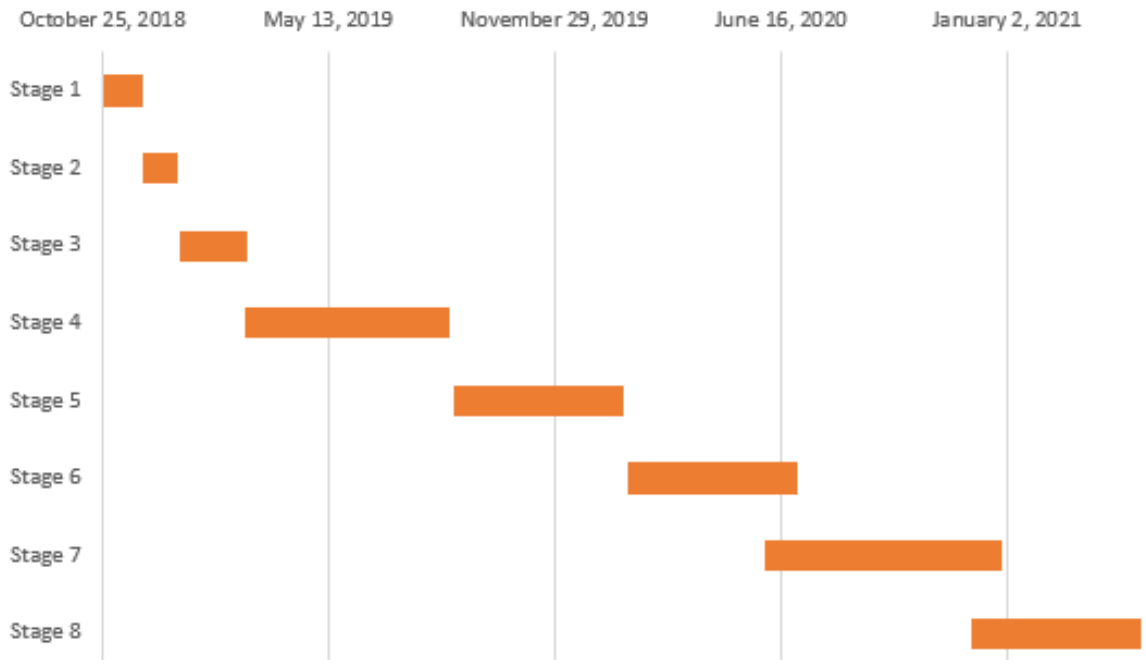
Stage 6: Export the remaining co-changes whom at this step we can call logical dependencies and use them among structural dependencies in tools for architectural reconstruction to evaluate the improvement.

Stage 7: Compare the number of logical dependencies with metrics like Fan Out, Fan In, Efferent Coupling (Ce), Afferent Coupling (Ca) and study their connections.

Stage 8: Identify other tools that use historical information and evaluate the impact of co-changes filtering into logical dependences for them.

6.2 Gantt chart

Gantt Chart				
START DATE	END DATE	DESCRIPTION	PAPER OUTPUT	DURATION (days)
10/25/18	11/30/18	Stage 1	No	35
11/30/18	12/30/18	Stage 2	No	30
1/1/19	3/1/19	Stage 3	Yes: ENASE, SACI	60
3/1/19	9/1/19	Stage 4	No	180
9/1/19	2/1/20	Stage 5	Yes: ICSME	150
2/1/20	7/1/20	Stage 6	Yes: ENASE	150
6/1/20	1/1/21	Stage 7	Yes: ICSE	210
12/1/20	5/1/21	Stage 8	No	150



6.3 Proposed contents of the thesis

CHAPTER I - theoretical notions

CHAPTER II - dependencies in software systems

CHAPTER III - co-changes

CHAPTER IV - co-changes filtering

CHAPTER V - usage of dependencies

Bibliography

[1] S A. Bohner and R S. Arnold. Software change impact analysis. *IEEE Computer Society*, 1, 01 1996.

[2] James A. Jones and Mary Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. pages 273–282, 01 2005.

[3] Nemitari Ajienka and Andrea Capiluppi. Understanding the interplay between the logical and structural coupling of software classes. *Journal of Systems and Software*, 134:120–137, 2017.

[4] L Bass, P Clements, and Rick Kazman. Software architecture in practice 2nd edition. 01 2003.

[5] Ira Baxter, Andrew Yahin, Leonardo de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. volume 368-377, pages 368–377, 01 1998.

[6] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, Jan 1995.

[7] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution:

A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[8] David Binkley. Source code analysis: A road map. pages 104–119, 06 2007.

[9] Grady Booch. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2004.

[10] Frederick P. Brooks, Jr. No silver bullet essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987.

[11] Gerardo Canfora and Massimiliano Di Penta. New frontiers of reverse engineering. pages 326 – 341, 06 2007.

[12] Marcelo Cataldo, Audris Mockus, Jeffrey A. Roberts, and James D. Herbsleb. Software dependencies, work dependencies, and their impact on failures. *IEEE Transactions on Software Engineering*, 35:864–878, 2009.

[13] E.J. Chikofsky, Cross , and II . Reverse engineering and design recovery: A taxonomy. *Software, IEEE*, 7:13–17, 02 1990.

[14] Holger Cleve and Andreas Zeller. Locating causes of program failures. pages 342– 351, 06 2005.

[15] Ioana Şora. Software architecture reconstruction through clustering: Finding the right similarity factors. In *Proceedings of the 1st International Workshop in Software Evolution and Modernization - Volume 1: SEM, (ENASE 2013)*, pages 45–54. INSTICC, SciTePress, 2013.

[16] Ioana Şora. Helping program comprehension of large software systems by identifying their most important classes. In *Evaluation of Novel Approaches to Software Engineering - 10th International Conference, ENASE 2015, Barcelona, Spain, April 29-30, 2015, Revised Selected Papers*, pages 122–140. Springer International Publishing, 2015.

[17] Ioana Şora, Gabriel Glodean, and Mihai Gligor. Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In *Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on*, pages 259–264, May 2010.

[18] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, July 2009.

[19] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. 01 1999.

[20] Steven Fraser, Frederick Brooks, Jr, Martin Fowler, Ricardo Lopez, Aki Namioka, Linda M. Northrop, David Parnas, and Dave Thomas. "no silver bullet" reloaded: retrospective on "essence and accidents of software engineering". pages 1026–1030, 01 2007.

[21] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the International Conference on Software Maintenance*, ICSM '98, pages 190–, Washington, DC, USA, 1998. IEEE Computer Society.

[22] K H. Bennett, Dh Le, and Vaclav Rajlich. The staged model of the software lifecycle: A new perspective on software evolution. 05 2000.

[23] Keith H. Bennett and Vaclav Rajlich. Software maintenance and evolution: a roadmap. pages 73–87, 05 2000.

[24] H. Kagdi, M. Gethers, D. Poshyvanyk, and M. L. Collard. Blending conceptual and evolutionary couplings to support change impact analysis in source code. In *2010 17th Working Conference on Reverse Engineering*, pages 119–128, Oct 2010.

[25] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28:654– 670, 08 2002.

[26] Franz Lehner. Software life cycle management based on a phase distinction method. *Microprocessing and Microprogramming*, 32:603–608, 08 1991.

[27] Bennet Lientz, E Burton Swanson, and Gerry E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21:466–471, 06 1978.

[28] Bennet P. Lientz and E. Burton Swanson. Problems in application software maintenance. *Commun. ACM*, 24(11):763–769, November 1981.

[29] P. K. Linos and V. Courtois. A tool for understanding object-oriented program dependencies. In *Proceedings 1994 IEEE 3rd Workshop on Program Comprehension- WPC '94*, pages 20–27, Nov 1994.

[30] Andrian Marcus and J.I. Maletic. Identification of high-level concept clones in source code. pages 107– 114, 12 2001.

[31] Jean Mayrand, Claude Leblanc, and Ettore M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. pages 244–, 01 1996.

[32] Gustavo Ansaldi Oliva and Marco Aurelio Gerosa. On the interplay between structural and logical dependencies in open-source software. In *Proceedings of the 2011 25th Brazilian Symposium on Software Engineering*, SBES '11, pages 144–153, Washington, DC, USA, 2011. IEEE Computer Society.

[33] Gustavo Ansaldi Oliva and Marco Aurélio Gerosa. Experience report: How do structural dependencies influence change propagation? an empirical study. In *26th IEEE International Symposium on Software Reliability Engineering, ISSRE 2015, Gaithersbury, MD, USA, November 2-5, 2015*, pages 250–260, 2015.

[34] Denys Poshyvanyk, Andrian Marcus, Rudolf Ferenc, and Tibor Gyimóthy. Using information retrieval based coupling measures for impact analysis. *Empirical Software Engineering*, 14(1):5–32, Feb 2009.

[35] Vaclav Rajlich. Modeling software evolution by evolving interoperation graphs. *Ann. Software Eng.*, 9:235–248, 05 2000.

[36] Xiaoxia Ren, B. G. Ryder, M. Stoerzer, and F. Tip. Chianti: a change impact analysis tool for java programs. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pages 664–665, May 2005.

[37] Spencer Rugaber. Program comprehension. 08 1997.

[38] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems,*

Languages, and Applications, OOPSLA '05, pages 167–176, New York, NY, USA, 2005. ACM.

[39] Kamran Sartipi. Software architecture recovery based on pattern matching. 09 2003.

[40] Mark Shtern and Vassilios Tzerpos. Clustering methodologies for software engineering. *Adv. Soft. Eng.*, 2012:1:1–1:1, January 2012.

[41] Dag Sjberg, Tore Dyb, and Magne Jorgensen. The future of empirical methods in software engineering research. pages 358–378, 06 2007.

[42] Adelina Diana Stana. and Ioana ora. Identifying logical dependencies from co-changing classes. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE,*, pages 486–493. INSTICC, SciTePress, 2019.

[43] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. 11 2002.

[44] Igor Scaliante Wiese, Rodrigo Takashi Kuroda, Reginaldo Re, Gustavo Ansaldi Oliva, and Marco Aurélio Gerosa. An empirical study of the relation between strong change coupling and defects using history and social metrics in the apache aries project. In Ernesto Damiani, Fulvio Frati, Dirk Riehle, and Anthony I. Wasserman, editors, *Open Source Systems: Adoption and Impact*, pages 3–12, Cham, 2015. Springer International Publishing.

[45] Norman Wilde. Understanding program dependencies, 1990.

[46] Hongji Yang and Martin Ward. *Successful Evolution of Software Systems*. Artech House, Inc., Norwood, MA, USA, 2003.

[47] Hongji Yang and Martin Ward. Successful evolution of software systems. 01 2003.

[48] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference, 1978. COMPSAC '78.*, pages 60–65, Nov 1978.

[49] Liguo Yu. Understanding component co-evolution with a study on linux. *Empirical Softw. Engg.*, 12(2):123–141, April 2007.

[50] Thomas Zimmermann, Peter Weisgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 563–572, Washington, DC, USA, 2004. IEEE Computer Society.