Identifying Implicit Architectural Dependencies using Measures of Source Code Change Waves

Miroslaw Staron[1)], Wilhelm Meding[2)], Christoffer Höglund[3)], Peter Eriksson[2)], Jimmy Nilsson[2)], and Jörgen Hansson[1)]

[1)]Software Center/Computer Science and Engineering
Chalmers | University of Gothenburg
miroslaw.staron@gu.se
jorgen.hansson@chalmers.se

[2)]Software Center/Ericsson
wilhelm.meding/
peter.r.eriksson/
jimmy.p.nilsson@ericsson.com

[3)]Software Center/Saab Electronic Defense Systems
christoffer.hoglund@saabgroup.com

ABSTRACT
The principles of Agile software development are increasingly used in large software development projects, e.g. using Scrum of Scrums or combining Agile and Lean development methods. When large software products are developed by self-organized, usually feature-oriented teams, there is a risk that architectural dependencies between software components become uncontrolled. In particular there is a risk that the prescriptive architecture models in form of diagrams are outdated and implicit architectural dependencies may become more frequent than the explicit ones. In this paper we present a method for automated discovery of potential dependencies between software components based on analyzing revision history of software repositories. The result of this method is a map of implicit dependencies which is used by architects in decisions on the evolution of the architecture. The software architects can assess the validity of the dependencies and can prevent unwanted component couplings and design erosion hence minimizing the risk of post-release quality problems. Our method was evaluated in a case study at one large product at Saab Electronic Defense Systems (Saab EDS) and one large software product at Ericsson AB.

Categories and Subject Descriptors
D.3.3 [**Software Engineering**]: Software/Program Measurement – visualization techniques.

General Terms
Measurement, Documentation, Design.

Keywords
Change impact analysis, change waves, measure, mining software repositories.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
conference.
Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

1. INTRODUCTION
The introduction of Agile and Lean software development principles has changed the practices in software industry in a number of ways. Agility and the focus on customer led to better products and ability of products to be delivered constantly (so called continuous delivery or continuous deployment). For large software development products these practices introduced new challenges. The principles led to multiple teams working in parallel and developing code for the common code base while working on distinct features. This kind of dynamics led to challenges in monitoring the evolution of the architecture and in particular the dependencies/links between components.

The architecture of the software product under development can erode over time, i.e. the explicit and prescriptive architecture models, assumptions and constraints might change over time. In the case of this research we consider the prescriptive architecture model as *a model which is á priori created by architects to describe how the architecture should be realized*. Our focus is on the fact that this is an explicit model created by architects who á priori "design" the architecture and we contrast this model with a descriptive model of the architecture of the same software product. The descriptive model shows how the architectural design has been realized, is created á posteriori and can be extracted from the existing design in a number of ways (e.g. by extracting component dependencies).

In addition to architecture erosion, the existence of implicit dependencies may lead to quality problems and delays of software delivery if unmonitored, uncontrolled and unmanaged. This paper addresses the problem of monitoring, controlling and explicitly managing the implicit dependencies between components by creating a method for identifying and monitoring of *change waves*. A change wave is a chain of related changes of components in source code during a period of time. Based on analyzing revision histories and identifying related changes we can find components which change together in a large number of cases. By chaining these dependency pairs we could identify waves of changes and predict which components should be developed/tested/monitored together. We consider the pairs as we intend to visualize the dependencies between all components of a change wave, not only the first and the last.

The results of our research are validated at two large industrial products from two different domains (defense and telecom). The validation showed that the change wave analyses were efficient support for architects in identifying dependencies between modules and predicting changes over time.

The remaining of the paper is structured as follows. Section 2 describes the main related work to our research. Section 3 describes the specific challenges in architecture work in Agile and Lean software development which is a context of this study. Section 4 presents the case studied in this paper. Section 5 presents the results and Section 6 presents the conclusions and further work.

2. RELATED WORK

Ball and Nagappan [1] studied the impact or relative code churn measures on software quality at Microsoft. Their work, based on the source code of MS Vista and MS Windows Server showed that these simple measures can predict defect-prone modules with high likelihood. A follow-up similar study was conducted by Bell et al. [2] at AT&T on a product with 18 releases. Bell et al. checked whether there are other metrics which could improve the results of predictions and came to conclusion that the churn measures were indeed the strongest predictors. The metrics to collect were based on the results of the above studies.

Zimmermann et al [3] introduced the methods for mining software repositories in order to guide how software should evolve. Their results were applied on a number of open source projects with good validation based on historical data. In our study we extend their concept of pairwise couplings to change waves and validate the results on a set of ongoing projects, i.e. not on historical analyses. Our initial visualization was based on their visualization.

Discovery of architectural dependencies based on runtime analyses was an important input to our work [4]. Arias et al. presented a method for visualizing this kind of dependencies. In our work we were inspired by their approach and complement their work with another way of eliciting dependencies.

An example of a metric of non-conformance of architectural design to the system can be found in [5] where the execution profiles are used to create component dependency maps. Our method complements such an analysis with the analysis of the development of the system. In our future work we plan to use both methods on the same system and compare the results.

Project telemetry using tools like Hackystat [6] usually complement tools used for visual analytics [7] with continuous measurement. The results of this research resulted in a simple tool used at one collaborating company which combines the strengths of both tools – an early warning system. Together with recent studies of Buse and Zimmermann [8] these results provided a solid ground for establishing online measurements in our method. Buse and Zimmermann [8] reported on a survey conducted at Microsoft where information needs were collected from 110 Microsoft designers, project managers and architects. Defect- and code stability related information was among the top information needs – what the managers would like to know. Not only were these aspects important for the historical analyses, they were important for the future insights of the company. The survey from Microsoft shows that the indicators presented in our paper fill an important need in software industry.

IBM has also identified metrics related to technical product development as important for Agile software development [9]. In the category of technical progress, the indicators should show that there is a growth of the product. Our dependency indicators take it one step further and show how “controlled” this growth is in terms of architectural and design dependencies.

Complementary measures to code stability should show the business aspects of software development, e.g. business value, which is one of important measures which should be used by Agile teams and companies [10]. The awareness of how the team contributes to the value is an important driver for the success of Agile projects. What the authors of the cited article postulate is similar to what we intend to achieve – provide key information without introducing manual work overhead. The complementary focus of the cited article is on the customer value, whereas the focus of this article is on quality risk.

Another important measure which is claimed to stimulate agility in software development teams, and thus complement the technical aspects of code stability, is the RTF (Running Tested Features) measure, popular in XP [11]. The metric combines three important concepts – the feature (i.e. a piece of code useful for the end-user, not a small increment that is not visible to the end user), execution (i.e. adding the value to the product through shipping the features to the customer), and the testing process (i.e. the quality of the feature – not only should it be execute, but also be of sufficient quality). This measure needs to be combined with measures on how solid the design is and this is the goal of our indicator.

A set of other metrics useful in the context of continuous deployment can be found in the work of Fritz [12] in the context of market driven software development organization. The metrics presented by Fritz measure such aspects as continuous integration pace or the pace of delivery of features to the customers. These metrics complement the two indicators presented in this paper with a different perspective important for product management.

The delivery strategy which is an extension of the concept of continuous deployment has been found as one of the three key aspects important for Agile software development organizations in a survey of 109 companies by Chow and Cao [13]. The indicator presented in this paper is a means of supporting organizations in their transition towards achieving efficient delivery processes which are in line with the delivery strategy prioritized by practitioners in this survey.

3. ARCHITECTURE IN AGILE DEVELOPMENT

Architecture development in software development is usually conducted by experienced architects and the larger the product, the more experience is required. As each type of system has its specific requirements the architectural design requires attention to specific aspects like real time properties or extensibility. For example in the telecom domain the extensibility and performance are the main aspects whereas in the automotive domain it is the safety and performance that is of the outmost priority. The architecture development efforts are dependent to some extent on the software development process adopted by the company – e.g. the architecture development methods differ in the V-model and in the Agile methodologies. In the V-model the architecture work is mostly prescriptive and centralized around the architects whereas in the Agile methods the work can be more descriptive and distributed into multiple self-organized teams.

As the introduction of Agile software development principles spread in industry, the architecture development evolved. As Agile development teams became self-organized the architecture work became more distributed and harder to control centrally [14]. The difficulties stem from the fact that Agile teams value independence and creativity [15] whereas architecture development requires stability, control, transparency and proactivity [16].

Figure 1 presents an overview on how the functional requiremnets (FR) and non-functional requirements (NFR) are packaged into work packages and developed as features by the teams. Each team delivers their code into the main branch. Each team has the possibility to deliver the code to any component of the product.

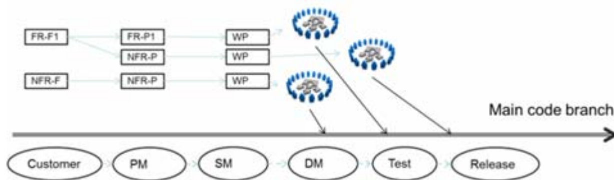

Figure 1. Feature development in Lean/Agile methods.

The requirements come from the customers and are prioritized and packaged into features by product management (PM) who communicates with the system management (SM) on the technical aspects of how the features affect the architecture of the product. The system management communicates with the teams (DM, Test) who design, implement and test (functional testing) the feature before delivering to the main branch. The code in the main branch in tested thoroughly by dedicated test units before being able to release [17].

The method proposed and evaluated in this paper is based on mining software repositories to find situations where groups of components are updated within an arbitrary number of days. The working assumption is that the components which are often updated together (in this case within the same week) are usually dependent upon each other. The method uses basic statistics combined with simple visualizations to present the results to architects who can verify the results of the statistics.

Examples of dependencies in the studies products are:

- Dependencies by-design – explicit in the architectural design
- Dependencies by-implicit – e.g. dependencies by-protocol – when two components implement protocols that are somehow dependent, but the components are not explicitly connected in the diagrams

The implicit dependencies are naturally more interesting than the explicit ones since they constitute risks for the overall internal and external quality of the product. The implicit dependencies have the tendency to become tacit knowledge over time and hard to maintain. They could lead to "forgetting" to update dependent components and thus defects detected late in the integration phases or system test phases. Therefore it is important to use automated measurement systems to identify, monitor and alert about these dependencies. The alerts give the teams the possibility to react and to prevent architecture erosion (through refactoring) or quality deterioration (through smarter testing).

4. CASE STUDY DESIGN

This case study was designed based on mixed flexible-fixed research design [18]. The design of the pilot study and the validation of the results were fixed, although we intended to adjust the method after the pilot study – thus making it mixed design. The sampling of the companies was done based on the size of their products and development methods used. Since the study was designed to be quantitative there was a need for large quantities of data, which dictated working with large companies developing large products. In this study we had the unique opportunity to work with 2 large companies – Saab EDS (development of software for defense systems) and Ericsson AB (development of telecom network equipment). The criteria for choosing the projects in these companies were:

- Use of source code for product development – although almost all companies execute projects in model-driven manner, we chose the projects where source code was the main artifacts, i.e. designers used programming languages like Java, C, C++ or Erlang for development.
- Initiated changes towards continuous deployment – the projects started changing their ways-of-working towards continuously deploying functionality to their customers.
- Size of the product – the products developed should be of significant size (more than 100.000 LOC) and should be developed during a period of time longer than 1 year (with multiple releases since the beginning of the product lifecycle).

Saab EDS developed embedded software and graphical user interfaces for ground based radar systems. The specific product we worked on was part of a larger product developed by several hundred developers, designers, testers, analysts etc. The historic project developing the product was driven in increments and did not utilize cross functional teams. The project management did some manual metrics on trouble reports.

The organization has since this project evolved into using more agile processes and cross functional teams. A lot of improvements and optimizations have also been done regarding software build and delivery times. Also to improve customer value, market competitiveness and profit, Saab AB Electronic Defense Systems in Gothenburg is going through a Lean transformation.

Ericsson AB developed large products for the mobile telephony network. The size of the organization was several hundred engineers and the size of the projects was up to a few hundreds[1]. Projects were increasingly often executed according to the principles of Agile software development and Lean production system referred to as Streamline development (SD) within Ericsson [19]. In this environment various disciplines were responsible for larger parts of the process compared to traditional processes: design teams (cross-functional teams responsible for complete analysis, design, implementation, and testing of particular features of the product), network verification and integration testing, etc.

The organization used a number of measurement systems for controlling the software development project (per project) described above, a number of measurement systems to control the quality of products in field (per product) and a measurement system for monitoring the status of the organization at the top level. All measurement systems were developed using the in-

[1] The exact size of the unit cannot be provided due to confidentiality reasons.

house methods described in [20, 21], with the particular emphasis on models for design and deployment of measurement systems presented in [22, 23].

The needs of the organization had evolved from metric calculations and presentations (ca. 7 years before the writing of this paper) to using predictions, simulations, early warning systems and handling of vast quantities of data to steer organizations at different levels and providing information from project and line. These needs have been addressed by the action research projects conducted in the organization, since the 2006.

4.1 Metrics used in the study

The base for calculating the strength of potential dependency between two components was the measure of *number of common change burst (NoCB)*, which was defined as the number of bursts which contain both components. The measure is non-transitive and non-reflective.

This measure can be illustrated based on change patterns in Figure 2 and Figure 3, where the dots with different fill show changes in different components. The dot with the solid black fill shows the change in component A, which is chosen as the starting point for the first burst (the upper timeline), the skew-lined fill of the dots indicate the change event in another component (component B) which is included in the change burst of component A, but also can be seen as a starting point for the next change burst – as illustrated in the lower timeline.

The time interval for the change burst is set arbitrary to one week in this example and could be adjusted. Choosing the interval of one week allows capturing check-in patterns of daily check-ins of some designers and once per week by others and anything in-between.

Figure 2 shows two change bursts originating in component A of a length of one week each.

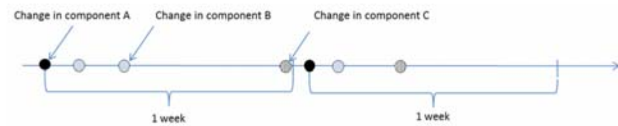

Figure 2. Component change patterns with bursts originating in Component A, based on [1]

The NoCB (Number of Common Change Bursts) measure for pairs originating in Component A are:

- $NoCB_{A\text{-}B}$ = 2: Component B changes in both change bursts originating at component A.
- $NoCB_{A\text{-}C}$ = 2: Component C changes in both bursts originating at component A.

These change bursts need to be complemented with the change bursts originating at component B, which is illustrated in Figure 3. In the figure there are three bursts of size of one week which originate in component B.

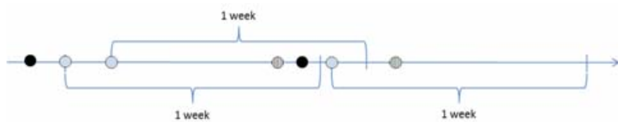

Figure 3. Component change patterns with bursts originating in Component B, based on [1]

The common change burst measures for the example in Figure 3 are:

- $NoCB_{B\text{-}A}$ = 2: Component A changes in two bursts originating at component B.
- $NoCB_{B\text{-}C}$ = 2: Component C changes in two bursts originating at component B.

The numbers above show that the measure of common burst provide only a basis for calculating the *strength of dependency (SoD)* which has to take into the account also the total number of bursts for the originating component. In order to calculate that strength of dependency we defined the *total number of bursts (NoB)*. The definition of the strength of dependency is defined as:

$$SoD = \frac{NoCB}{NoB} * 100\%$$

In the example the formula provides the following results for the dependency between component A and B:

$$SoD_{A-B} = \frac{NoCB_{A-B}}{NoB_A} * 100\% = \frac{2}{2} * 100\% = 100\%$$

The results for the entire example are:

- $SoD_{A\text{-}B}$: 100%
- $SoD_{A\text{-}C}$: 100%
- $SoD_{B\text{-}A}$: 67%
- $SoD_{B\text{-}C}$: 100%
- $SoD_{C\text{-}A}$: 50%
- $SoD_{C\text{-}B}$: 50%

The data shows that changes in component A can potentially initiate changes in components B and C, while changes in component C do not cause changes in components A and B equally often. It could be visualized in a table to provide an overview – Table 1, the colors indicate the strength of dependency for attracting the attention of the stakeholders to pairs of components which should be considered first (the most intensive colors) as prescribed by [24].

Table 1. Strength of dependency visualized in a table

	A	B	C
A		100%	100%
B	67%		100%
C	50%	50%	

In general, some of the dependencies which are found in this method could be explicit, i.e. exist in the architecture diagrams, whereas some were implicit, i.e. not present in the diagrams. The latter are naturally more interesting for the architects and in the case study at Saab EDS and Ericsson we found that many of these dependencies were not explicit, which showed the value of the presented method.

This tabular visualization can show interesting patterns of component dependencies as analyzed by Zimmermann et al. [3], but it does not show the real change wave, i.e. the pattern how changes in the components spread over the system. For this we used a simple visualization of how the change flows presented in Figure 4.

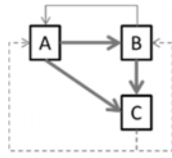

Figure 4. Visualization of change waves as a flow

The bold lines in the diagram show the strongest dependencies while the dotted ones show the weakest ones. The dependencies correspond to the ones in Table 1. Focusing only on the strongest dependencies the diagram shows that component A usually is the component where changes originate and that they propagate to the other two components. Changes "back" to component A are not that often, which indicate lower dependency.

In this study we used flows to identify change waves and when discussing them with the architects.

4.2 Data collection and analysis

The process of data collection was as follows:

1. Pilot at Saab EDS: Initially we evaluated the measures in a pilot study at Saab EDS where we calibrated the way in which the NoCB metric is collected, we defined the information model for this measure according to ISO 15939 [25] and measuring the dependency for one large product. The results showed that the method identified a number of implicit dependencies.
2. Study at Ericsson: Based on the pilot study we decided to collect the dependencies from another large product from a different company – Ericsson. We also decided to use a different visualization technique to show the dependencies and we had the possibility to validate whether the dependencies are implicit or by-design with two main architects for the product. The results showed that there is a set of implicit dependencies and a pattern of change waves.
3. Study at Saab EDS: Finally we used the method to make a map of dependencies for another product from the same product line at Saab EDS. The patterns of change waves were different, i.e. longer, than the change waves at Ericsson.

We collected the data using scripts in Ruby and Perl and visualized the data using MS Powerpoint and MS Excel. The analysis of data was done through interviews, i.e. discussions with architects.

5. RESULTS AND IMPACT

The results from the pilot study at Saab EDS significantly influenced the method in terms of how the measures are calculated. In particular we experimented with different ways of calculating NoCB measure.

5.1 Pilot study at Saab EDS

Before the case study was executed at both companies we conducted a pilot study at Saab EDS where we validated the approach and the empirical validity of the measures used in the study. In particular we validated the NoCB measure by investigating change wave from one component and interviewing the architect of the product.

The setup of the pilot study was to investigate a number of components (identified á priori by architects) and their dependencies to other components. The data was visualized using a bar chart as presented in Figure 5 where the size of the bar represents the strength of dependency (SoD) of component on the x-axis to the arbitrary architect-chosen component (let us refer to it as Component 0). For confidentiality reasons the scales and names of components have been removed.

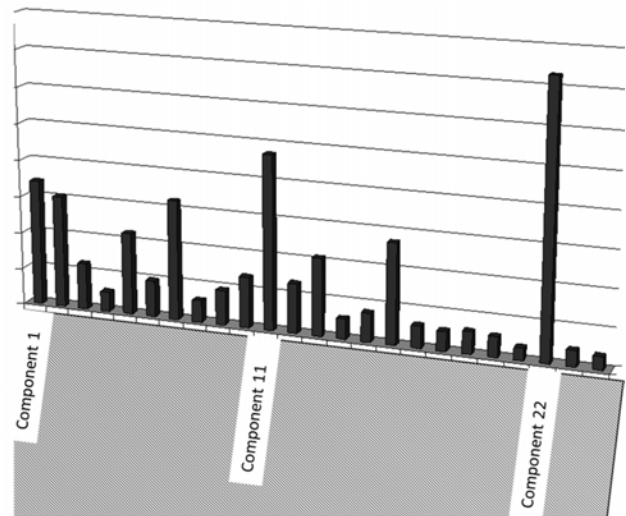

Figure 5. Bar chart illustrating the strength of potential dependency of other components on Component 0

Using the bar chart in Figure 5 we managed to attract the attention of the architects to Component 0 (not in the diagram) and Component 22 in the diagram. The largest bar represented a dependency which was not defined á priori, but appeared as a result of the dependency of these two components on a common protocol. This dependency showed that in practice the design of those two dependent components needed to be synchronized otherwise a risk of integration defects (hard and costly to find) can be significant.

The next step in our analysis was to investigate dependencies originating from Component 22, which are depicted in Figure 6.

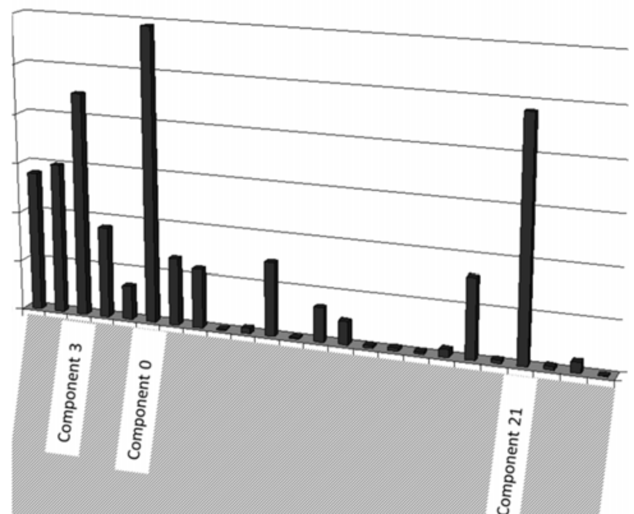

Figure 6. Bar chart illustrating the strength of potential dependency of other components on Component 22

The chart shows that it is Component 0 which is the most dependent one on Component 22. That dependency was not explicit in the architecture diagram, but was confirmed by the architect – the architect was able to explain why these two components changed "together" and that there was indeed an implicit relationship between these component via a communication protocol.

The results from the pilot study showed that this type of analysis has a potential to find dependencies that were not explicit for the architects. This analysis was named as "change wave analysis" since it showed dependency between components based on the propagation of changes. It was also decided that we should extend this analysis to visualize dependencies between all components in one diagram to avoid the need for the first manual step – arbitrary choice of the initial component (Component 0).

5.2 Results from Ericsson and Saab EDS

As defined in our research process the change wave analysis method was applied at two large products at two different companies. The results show that these two products have different architectural dependencies and that some of these dependencies were not explicitly known to the architects.

Figure 7 presents the change waves identified in the study with different types of lines encoding strengths of the dependency – bold (50-100%), normal (30-50%) and dotted (10-30%).

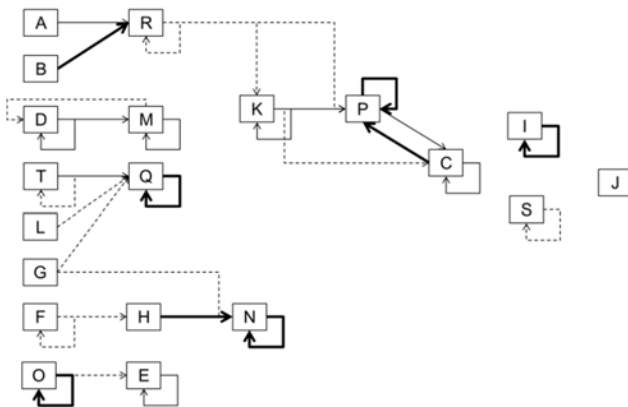

Figure 7. Change waves for the product at Saab

The waves starting from components A and B in Figure 7 are rather long and complex. For example, if component B changes, there is a significant chance that component R will change and a chance that components K and P might change.

Disregarding the weakest dependencies, i.e. the dotted lines, the figure shows that there are still dependencies between components, for example, A-R, B-R and K-P-C. These dependencies should be used to plan testing of the system.

In Figure 7 one component is different from the others – component J – as it is not dependent on other components. The component was developed separately from others and no change waves originate from this component of lead to this component.

Figure 8 presents the results of applying the change wave analysis for the product at Ericsson where we use different names for the components (C1-C26), emphasize the change waves and disregard the weakest dependencies (below 30%) since with these dependencies nearly all components were inter-connected.

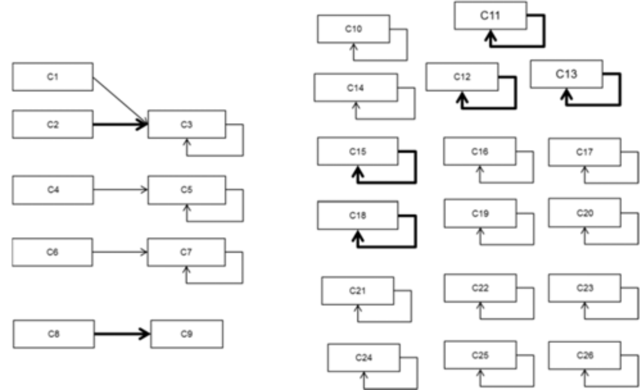

Figure 8. Change waves for the product at Ericsson

The change waves presented in Figure 8 are shorter than in case of the product in Saab, but there are more "intra-component" waves – components C10 – C26 in the right-hand side of the figure. This means that changes are usually contained within a single component, which might lead to a number of conclusions about the quality of the architecture and the ways-of-working at the company. One of the conclusions was that the architectural components are rather independent from each other, which is caused by the fact that they are developed by geographically distributed teams.

5.3 Evaluation

We identified a number of parameters which are worthy evaluation and discussion with the architects:

- *Length of the burst*, for example one day, one week, one release.
- *Branch filtering*, for example ignoring branch name, strict branch name (only changes in the same branch are calculated) or similarity of branches (using Levenshtein distance of one-5 characters).
- *Time period for collecting the data*, for example complete product revision history, one release, one month.

The first evaluation of the method was done during the pilot study with a focus group of two architects at Saab EDS. The evaluation was positive and the research team decided to continue to develop the method completely and apply it to other products. A number of implicit dependencies were found and discussed with the architects.

The second evaluation was done through focus group interviews with two architects at Ericsson where the method has shown itself useful when:

1. An explicit dependency on a common library was found. The dependency shown a pattern of all components which were affected by a library update – it was confirmed by the architects that this was indeed the case. This explicit dependency was found when running the method on a period of time of one release.
2. A number of implicit dependencies of components were found when analyzing the flow diagrams. The diagrams were plotted based on the dependency data collected from the whole product lifecycle. An example of an implicit dependency was dependencies between state machines implementing similar/related protocols.

3. By creating the dependency chart for each release (i.e. using source code revisions only for the period of the release) we found how the development of product features affected the architecture of the product – the method pinpointed which components were changed as a result of implementing a number of features in the release.

Capturing the explicit dependencies as in (1) showed that the method presented in this paper indeed identifies dependencies which exist in the product. Their analysis á priori indicated that the change in the library would spread throughout the system and affect numerous components.

Discussing the implicit dependencies as in (2) showed that the method is a good support for the architects when evaluating their design decisions and understanding the structure of the system from a new perspective.

Identifying dependency between component and features as in (3) showed that the method is an effective tool for the architects to evaluate the risks when implementing new features in the product and supporting the test planning. Understanding how the features affect components is a crucial element in managing the evolution of architectures and prevents design erosion. Since the method presented in this paper is automated, these analyses require minor effort for data collection and presentation, but require the attention of architects for analyzing the results and acting upon them.

5.3.1 Recommendations for other companies

Based on the experiences from using this method at two companies we identified a number of recommendations for companies willing to adopt this approach:

a) Implicit dependencies identified using this method should be used as input for test planning and execution at the feature level and at the system level (at least). This input can result in smarter testing and thus identifying defects early.
b) Change wave measures and analyses should be used by architects and designers to monitor the dependencies between components in the system. The dependencies can be formalized/documented in the diagrams in order to assure future maintainability of the system.
c) The analysis of dependencies between components should be complemented with the analysis of dependencies between modules/files in the components. The inter-module dependencies are more useful for the designers who need to be alerted about which components should be updated based on the change wave.
d) The analysis should be done on historical releases, but it should be used to predict how changes in components might spread in the new releases. Identifying the origin of the change wave should be communicated to designers who should take active decision whether the next component in the change wave should be updated – this decreases the risk of omitting important code updates that could result in defects later in the development process.

In addition to the recommendations for the use of the method, together with the architects we found the following calibration parameters to be useful for a number of analyses:

a) Length of change burst (one week in the example in section 4.1) is the same as the length of the release – the analysis shows how features spread over components in reality, which might be different from the design.
b) Length of change burst is 1 day – only the dependencies which are identified by small number of designer and can support efficient set-up of the cross-functional self-organized team.
c) Length of change burst > iteration cycle (including testing) – the results will include dependencies which are not known by the designers and are found during testing (when a test case failed because a component is not changed or changed incorrectly).
d) Filtering by branch name – if the bursts are filtered per branch as presented in Figure 2 in section 4.1 then the method identifies implicit although direct dependencies (e.g. correcting one defect or a single release). If the filtering is not done, then the method identifies more false-positives like dependencies between features developed for two distinct products in a product line.
e) Using sliding time intervals – using the analysis month-by-month or release-by-release provides the architects with the possibility of monitoring the evolution of architecture and identifying new implicit dependencies as they appear. If the analysis is done on the whole revision history, then the "new" dependencies are usually less visible as the "old" dependencies were present in the system for a longer time and are more strongly visible in the statistics.

In our further work we plan to extend the set of recommendations useful for other companies based on the experiences which we collect over time.

6. CONCLUSIONS

In this paper we presented a method for identifying implicit architectural dependencies using revision history of source code change waves. The results from the evaluation of this method at two companies – Saab Electronic Defence Systems and Ericsson AB – showed that the method identifies both the explicit and implicit dependencies. The results showed that manipulating with three parameters of the method (time period, length of change bursts and branch filtering) results in identifying distinct types of dependencies like feature-component dependencies or dependencies of components through common libraries or protocols.

The method is based on calculations which are relatively simple to replicate, but provide support for taking preventive measures from design corrosion or quality problems, not uncommon in large and long-live software products. The recommendations for other companies, which are based on the observations of how architects used the method, provide a starting point for using the method and initial guidelines on how to analyze the results.

In our future work we plan to identify more analysis patterns and expand the scope of data mining to static code analysis methods, to filter out explicit dependencies and only include the implicit ones.

ACKNOWLEDGMENTS

This research has been carried out in the Software Centre, Chalmers, Göteborgs Universitet and Ericsson AB, Saab Aktiebolag.

REFERENCES

[1] T. Ball and N. Nagappan, "Use of relative code churn measures to predict system defect density," in *27th*

International Conference on Software Engineering, St. Louis, MO, USA, 2005, pp. 284-292.
[2] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Does measuring code change improve fault prediction?," presented at the Proceedings of the 7th International Conference on Predictive Models in Software Engineering, Banff, Alberta, Canada, 2011.
[3] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *Software Engineering, IEEE Transactions on,* vol. 31, pp. 429-445, 2005.
[4] T. Arias, P. Avgeriou, and P. America, "Analyzing the Actual Execution of a Large Software-Intensive System for Determining Dependencies," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, 2008, pp. 49-58.
[5] S. Sarkar, G. M. Rama, and R. Shubha, "A Method for Detecting and Measuring Architectural Layering Violations in Source Code," in *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, 2006, pp. 165-172.
[6] P. M. Johnson, "Requirement and Design Trade-offs in Hackystat: An In-Process Software Engineering Measurement and Analysis System," presented at the Proceedings of the First International Symposium on Empirical Software Engineering and Measurement, 2007.
[7] R. P. L. Buse and T. Zimmermann, "Information Needs for Software Development Analytic," presented at the ICSE, International Conference on Software Engineering, Zurich, Switzeland, 2012.
[8] R. P. L. Buse and T. Zimmermann, "Information Needs for Software Development Analytics," presented at the 34th International Conference on Software Engineering (ICSE 2012 SEIP Track), Zurich, Switzerland, 2012.
[9] N. Ward-Dutton. (2011, Software Econometrics: Challenging assumptions about software delivery. *IBM.com podcast companion report*.
[10] D. Hartmann and R. Dymond, "Appropriate agile measurement: using metrics and diagnostics to deliver business value," in *Agile Conference, 2006*, 2006, pp. 6 pp.-134.
[11] R. Jeffries. (2004). *A Metric Leading to Agility*. Available: http://xprogramming.com/xpmag/jatRtsMetric
[12] T. Fitz. (2009). *Continuous Deployment at IMVU: Doing the impossible fifty times a day*. Available: http://timothyfitz.wordpress.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/
[13] T. Chow and D.-B. Cao, "A survey study of critical success factors in agile software projects," *Journal of Systems and Software,* vol. 81, pp. 961-971, 2008.
[14] E. Richardson, "What an Agile Architect Can Learn from a Hurricane Meteorologist," *Software, IEEE,* vol. 28, pp. 9-12, 2011.
[15] H. Sharp, N. Baddoo, S. Beecham, T. Hall, and H. Robinson, "Models of motivation in software engineering," *Information and Software Technology,* vol. 51, pp. 219-233, 2009.
[16] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes,* vol. 17, pp. 40-52, 1992.
[17] M. Staron and W. Meding, "Monitoring Bottlenecks in Agile and Lean Software Development Projects – A Method and Its Industrial Use," in *Product-Focused Software Process Improvement*, Tore Cane, Italy, 2011, pp. 3-16.
[18] C. Robson, *Real World Research*, 2 ed. Oxford: Blackwell Publishing, 2002.
[19] P. Tomaszewski, P. Berander, and L.-O. Damm, "From Traditional to Streamline Development - Opportunities and Challenges," *Software Process Improvement and Practice,* vol. 2007, pp. 1-20, 2007.
[20] M. Staron, W. Meding, G. Karlsson, and C. Nilsson, "Developing measurement systems: an industrial case study," *Journal of Software Maintenance and Evolution: Research and Practice,* pp. n/a-n/a, 2010.
[21] M. Staron, W. Meding, and C. Nilsson, "A Framework for Developing Measurement Systems and Its Industrial Evaluation," *Information and Software Technology,* vol. 51, pp. 721-737, 2008.
[22] M. Staron and W. Meding, "Using Models to Develop Measurement Systems: A Method and Its Industrial Use," presented at the Software Process and Product Measurement, Amsterdam, NL, 2009.
[23] W. Meding and M. Staron, "The Role of Design and Implementation Models in Establishing Mature Measurement Programs," presented at the Nordic Workshop on Model Driven Engineering, Tampere, Finland, 2009.
[24] X. Lai and J. K. Gershenson, "Representation of similarity and dependency for assembly modularity," *The International Journal of Advanced Manufacturing Technology,* vol. 37, pp. 803-827, 2008/06/01 2008.
[25] International Standard Organization and International Electrotechnical Commission, "ISO/IEC 15939 Software engineering – Software measurement process," International Standard Organization / International Electrotechnical Commission,, Geneva2007.