Identifying Entities That Change Together

Tudor Gîrba[1], Stéphane Ducasse[2], Radu Marinescu[3] and Daniel Raţiu[4]

[1,2] Software Composition Group
University of Berne, Switzerland
{girba, ducasse}@iam.unibe.ch

[3,4] LOOSE Research Group
University of Timişoara, Romania
{radum, ratiud}@cs.utt.ro

Abstract

[1] *Software system need to change over time to cope with the new requirements. Furthermore, due to design decisions, the new requirements happen to crosscut the system's structure. Understanding how changes appear in the system can reveal hidden dependencies between different parts of the system. We propose to group entities that change together according to a logical expression that specifies the change condition. Furthermore, we can group entities at different levels of abstraction (i.e., method, class, package). Our approach is based on an explicit history meta model that centers around the notion of history and which enables the definition of historical measurements which summarize the changes. We apply our approach on two large case studies and show how we can identify groups of related entities and detect bad smells.*

I. Introduction

Software systems need to change over time to cope with the new requirements [1]. As the requirements happen to crosscut the system's structure, changes will have to be made in multiple places. Understanding how changes appear in the system is important for detecting hidden dependencies between its parts.

In the recent period work has been carried out to detect and interpret groups of software entities that change together [2] [3] [4]. Yet, the detection is based on change information concerning one property, and is mostly based on file level information.

We propose an approach which detects evolutionary groups based on more than one property. We first define a history meta model and based on this meta model we define historical measurements which summarize the evolution of entities. We use these measurements to detect changes between two versions, and build an Evolution Matrix [5] annotated with changes. We use the matrix as an incidence table for the input into a concept analysis machine which returns the groups of entities that changed certain properties in the same versions. Also, for building the matrix of changes, we make use of logical expressions which combine properties with thresholds and which run on two versions of the system to detect interesting entities.

Example. ShotgunSurgery appears when whenever we have to change a class we have to change a number of other classes[6]. We would suspect a group of classes of such a bad smell, when they repeatedly keep their external behavior constant and change the implementation. We can detect this kind of change in a class in the versions in which the number of methods did not change, while the number of statements changed.

We can apply our approach on any kind of entities we have in the meta model. In this paper we show how we detect groups of packages, classes and methods. We applied our approach on two large open source case studies.

Next, we briefly introduce the notion of history as a first-class entity and define two generic historical measurements. We introduce our approach for grouping histories using historical measurements and concept analysis, and then show how we apply it on different levels of abstrac-

[1]Ninth IEEE Workshop on Empirical Studies of Software Maintenance (WESS 2004)

tions (*i.e.*, method, class, package). We discuss the results we obtained when applying our approach on two large case open source studies, and in the end, we conclude and present the future work.

II. History and History Measurements

We define a *history* to be a sequence of versions of the same kind of entity (*e.g.*, class history, system history, etc.). By a version we understand a snapshot of an entity at a certain point in time (*e.g.*, class version, system version, etc.).

Example. In the left side of Figure 1 we use a simplified example of the Evolution Matrix [5] to display a system history with 6 versions. A cell in the matrix is marked by a square and represents a class version. A line in the matrix represents a class history and a column represents a system version. In the figure, class A was present in all the version of the system, class B was removed in the last system version while class E appeared in the system only the third system version.

Addition of a Version Property (*A*). We define a generic measurement, called addition of a version property P (*A*(P, i)), as the addition of that property between version $i-1$ and i of the history H:

$$(i > 1)$$
$$A_i(P,H) = \begin{cases} P_i(H) - P_{i-1}(H), & P_i(H) - P_{i-1}(H) > 0 \\ 0, & P_i(H) - P_{i-1}(H) \leqslant 0 \end{cases} \quad (1)$$

Evolution of a Version Property (*E*). We define a generic measurement, called evolution of a version property P ($E_i(P)$), as being the absolute difference of that property between version $i-1$ and i:

$$(i > 1) \quad E_i(P,H) = |P_i(H) - P_{i-1}(H)| \quad (2)$$

We instantiate the above mentioned measurements by applying them on different version properties of different types of entities:

- Method: *NOS* (number of statements), *CYCLO* (McCabe cyclomatic number [7]).
- Class: *NOM* (number of methods), *WNOC* (number of all subclasses).
- Package: *NOCls* (number of classes), *NOM* (number of methods).

The *E* measurement shows a change of a certain property, while the *A* measurement shows the additions of a certain version property. Thus, these measurements summarize the evolution of properties.

III. Grouping Mechanism

In the left side of Figure 1 we display an Evolution Matrix in which each square represents a class version and the number inside a square represents the number of methods in that particular class version. A grayed square shows a change in the number of methods of a class version as compared with the previous version ($E_i(NOM) > 0$).

Based on such a matrix we can build a concept lattice by considering the histories as entities and the properties are given by "grayed in version x". In the right side of Figure 1, we show the concept lattice obtained from the Evolution Matrix on the left.

Each concept in the lattice represents all the class histories which changed certain properties together in those particular versions. In the given example, class history A and D changed their number of methods in version 2 and version 6.

We not only want to detect entities that change one certain property in the same time, we want to detect entities that change more properties, and/or do not change other properties. For example, to detect parallel inheritances it is enough to just look at the number of children of classes; but, when we want to look for classes which need to change the internals of the methods in the same time without adding any new functionality, we need to look for classes which change their size, but not the number of methods.

We encode this information in a expressions which consist of logical combination of historical measurements. These expressions are applied at every version on the last two versions. In the example from Figure 1, the expression used was $E_i(NOM) > 0$ which we applied on class histories.

In the followings we will introduce several expressions applicable on packages, classes and respectively methods.

A. Method Histories Grouping Expressions.

Parallel Complexity. A set of methods are effected by *Parallel Complexity* when a change in the complexity in one method involves changes in the complexity of other methods. As a measure of complexity we used the McCabe cyclomatic number. Classes with parallel complexity could reveal parallel conditionals.

$$ParallelComplexity : (A_i(CYCLO) > 0) \quad (3)$$

Parallel Bugs. We name a change a bug fix, when no complexity is added to the method, but the implementation changes. When we detect repetitive such bug fixes in more methods in the same versions, we group those methods in a

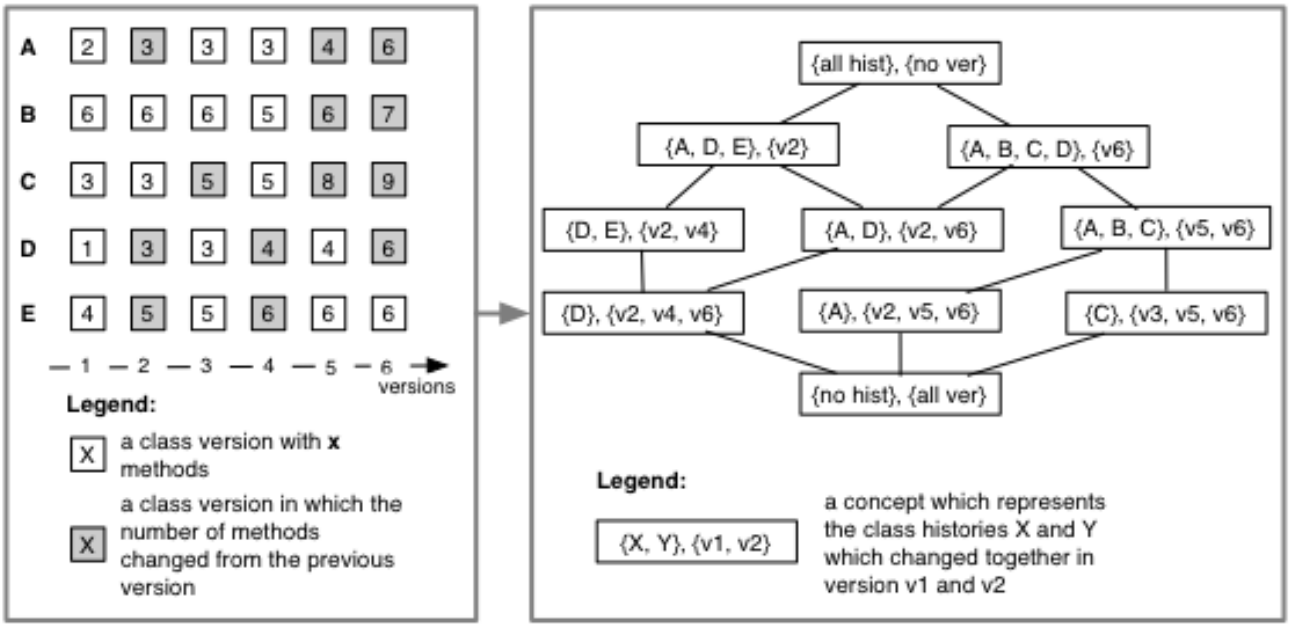

Fig. 1. Example of applying concept analysis to group class histories based on the changes in number of methods. The Evolution Matrix on the left is the incident table used as the input for the concept analysis machine which outputs the concept lattice on the right.

Parallel Bugs group. Such a group, might give indications of similar implementation which could be factor out. As an implementation measure we used number of statements.

$$ParallelBugs : (E_i(NOS) > 0) \wedge E_i(CYCLO) = 0) \quad (4)$$

B. Class Histories Grouping Expressions

Shotgun Surgery. The *Shotgun Surgery* bad-smell is encountered every time when a change operated in a class involves a lot of small changes to a lot of different classes [6]. We detect this bad smell, by looking at the classes which do not change their interface, but change their implementation together.

$$ShotgunSurgery = (E_i(NOM) = 0 \wedge E_i(NOS) > 0) \quad (5)$$

Parallel Inheritance. *Parallel Inheritance* is detected in the classes which change their number of children together [6]. Such a characteristic is not necessary a bad smell, but gives indications of a hidden link between two hierarchies. For example, if we detect a main hierarchy and a test hierarchy as being parallel, gives us indication that the tests were developed in parallel with the code.

$$ParallelInheritance = (A_i(WNOC) > 0) \quad (6)$$

Parralel Semantics. Methods specify the semantics of a class. With *Parallel Semantics* we detect classes which add methods in parallel. Such a characteristic could reveal hidden dependencies between classes.

$$ParallelSemantics = (A_i(NOM) > 0) \quad (7)$$

C. Package Histories Grouping Expression

Package Parallel Semantics. If a group of classes is detected, as having parallel semantics, we would want to relate the containing packages as well. *Package Parallel Semantics* detects packages where some methods have been added, but no class have been added or removed.

$$PackageParallelSemantics = (E_i(NOCls) = 0) \wedge (A_i(NOM) > 0) \quad (8)$$

IV. Experiments: JBoss

For our experiments we chose 41 versions of JBoss[2]. JBoss is an open source J2EE application server written in Java. The versions we selected for the experiments are at two weeks distance from one another starting from the beginning of 2001 until the end of 2002. Table I shows the characteristics of the case study. The first version has 632 classes, the last one has 4276 classes (we took into consideration all test classes, interfaces and inner classes).

System	Language	Versions	First Version (Size)	Last Version (Size)
JBoss	Java	41	40 kLOC 632 classes	281 kLOC 4276 classes

TABLE I. Characteristics of the JBoss case study.

Due to space limitation, we will only discuss the *ParallelInheritance* results we obtained on JBoss.

After applying the mechanism described above, we obtained 68 groups of class histories which added subclasses in the same time. Manual inspection showed there were a lot of repetitions (due to the way the concept lattice is built), and just a limited number of groups were useful. Furthermore, inside a group not all classes were relevant for that particular group.

For example, in 19 versions a class was added in the `JBossTestCase` hierarchy (`JBossTestCase` is the root of the JBoss test cases). Another example is `ServiceMBeanSupport` which is the root of the largest hierarchy of JBoss. In this hierarchy, classes were added in 18 versions. That means that both `JBossTestCase` and `ServiceMBeanSupport` were present in a large number of groups, but was not necessary related to the other classes in these groups.

These results showed that just applying concept analysis produced too many false positives. That is why we added a filtering step. The filtering step consists in identifying and removing from the groups the entities that changed their relevant properties (*i.e.,* according to the expression) more times than the number of properties detected in a group:

$$FilteringRule = \frac{groupVersions}{totalChangedVersions} > threshold \quad (9)$$

In our experiments, we chose the threshold to be 3/4. For example, if `JBossTestCase` was part of a group of classes which changed their number of subclasses in 10 versions, we would rule the class out of the group. We chose an aggressive threshold to reduce the number of false positives as much as possible, in the detriment of having true negatives.

[2]See http://www.jboss.org for more information.

Class Histories	Versions
org::jboss::system::ServiceMBeanSupport org::jboss::test::JBossTestCase	24 27 28 29 30 32 33 34 37 38 39 40 41 19 20
javax::ejb::EJBLocalHome javax::ejb::EJBLocalObject	24 41 28 30 32 36 37 38 23

TABLE II. Parallel Inheritance in JBoss

After the filtering step, we obtained just two groups. In Table II we show the class histories and the versions in which they changed the number of children.

In the first group we have two classes which change their number of children 15 times: `ServiceMBeanSupport` and `JBossTestCase`. The interpretation of this group is that the largest hierarchy in JBoss is highly tested.

The second group detects a relationship between the EJB interfaces: `EJBLocalHome` and `EJBLocalObject`. This is due to the architecture of EJB which requires that a bean has to have a `Home` and an `Object` component.

V. Implementation: Moose, Van and ConAn

We carried the experiments using a combination of tools:

- Van is our version analysis tool. It implements the history meta model (Hismo) and is built on top of Moose[8].
- ConAn is a concept analysis developed on top of Moose.

VI. Related Work

The first work to study the entities that change together was performed by Gall *et al.* [2]. The authors use the change information to define a proximity measurements which they use to cluster related entities. The work has been followed up by the same authors [9] and by Itko *et al.* [3].

Shirabad *et al.* employ machine learning techniques to detect files which are likely to need to be changed when a particular file is changed [10].

As opposite with the previous approaches, Zimmerman *et al.* placed their analysis at the level of entities in the meta model [4]. Their focus was to provide a mechanism to warn developers that: “Programmers who changed these

functions also changed. . . ”. Their approach differs from ours because they only look at syntactic changes, while we identify changes based on the semantics of the changes. Furthermore, our approach takes into consideration different changes in the same time.

Davey and Burd proposed the usage of concept analysis to detect evolutionary concepts, but there was no implementation evidence [11].

Detection of problems in the source code structure has long been a main issue in the quality assurance community. Marinescu [12] detects design flaws by defining detection strategies. Ciupke employed queries usually implemented in Prolog to detect “critical design fragments” [13]. Tourwe *et al.* also explored the use of logic programming to detect design flaws [14]. vanEmden and Moonen detected bad smells by looking at code patterns [15]. These approaches differs from ours because they use only the last version of the code, while we take into account historical information. Furthermore, vanEmden and Moonen proposed as future research the usage of historical information to detect Shotgun Surgery or Parallel Inheritance.

We developed a previous approach to using the history of entities to detect design flaws, but in that case we extended the concept of detection strategies proposed by Marinescu to take into account the time information and showed how we improved the detection [16] [17].

Two of the authors already used historical information to characterize how changes appear during the history of systems[18].

VII. Conclusions and Future Work

Understanding how a system changes can reveal hidden dependencies between different parts of the system. Moreover, such dependencies might reveal bad smells in the design.

We proposed the usage of historical information to detect parts of the system that change in the same time, according to different rules. For that, we defined a history meta model and history measurements. Based on this model we built queries which detect different types of changes. By applying these queries on every version we obtained an Evolution Matrix annotated with the change information which we then used as input for a concept analysis machine. The result obtained by the machine were groups of entities that change together and the versions in which they changed. We applied our approach on one large open source case study and discussed some of the results we obtained.

According to our algorithm the effectiveness of the approach is highly affected by the value of the threshold. When the threshold is high (*i.e.,* close to 1) we want to remove the false positives but we risk missing true negatives. Further work is required to better understand the nature of this threshold and its interpretation.

In the future we would also like to apply our approach on more case studies and analyze in depth the results we obtain at different levels of abstractions.

Acknowledgments

Ducasse and Gîrba gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects “Tools and Techniques for Decomposing and Composing Software” (SNF Project No. 2000-067855.02, Oct. 2002 - Sept. 2004) and “RECAST: Evolution of Object-Oriented Applications” (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

Gîrba would like to thank European Science Foundation for the financial support.

References

[1] M. M. Lehman and L. Belady, *Program Evolution — Processes of Software Change*. London Academic Press, 1985.

[2] H. Gall, K. Hajek, and M. Jazayeri, “Detection of logical coupling based on product release history,” in *Proceedings of the International Conference on Software Maintenance 1998 (ICSM ’98)*, 1998, pp. 190–198.

[3] J. Itkonen, M. Hillebrand, and V. Lappalainen, “Application of relation analysis to a small java software,” in *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, 2004, pp. 233–239.

[4] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” in *26th International Conference on Software Engineering (ICSE 2004)*, 2004.

[5] M. Lanza and S. Ducasse, “Understanding software evolution using a combination of software visualization and software metrics,” in *Proceedings of LMO 2002 (Langages et Modèles à Objets*, 2002, pp. 135–149.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[7] T. McCabe, “A measure of complexity,” *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, Dec. 1976.

[8] S. Ducasse, M. Lanza, and S. Tichelaar, “Moose: an extensible language-independent environment for reengineering object-oriented systems,” in *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[9] H. Gall, M. Jazayeri, and J. Krajewski, “Cvs release history data for detecting logical couplings,” in *International Workshop on Principles of Software Evolution (IWPSE 2003)*, 2003, pp. 13–23.

[10] J. S. Shirabad, T. C. Lethbridge, and S. Matwin, “Mining the maintenance history of a legacy software system,” in *International Conference on Software Maintenance (ICSM 2003)*, 2003, pp. 95–104.

[11] J. Davey and E. Burd, “Clustering and concept analysis for software evolution,” in *Proceedings of the 4th international Workshop on Principles of Software Evolution (IWPSE 2001)*, Vienna, Austria, 2001, pp. 146–149.

[12] R. Marinescu, “Measurement and quality in object-oriented design,” Ph.D. Thesis, Department of Computer Science, ”Politehnica” University of Timişoara, 2002.

[13] O. Ciupke, “Automatic detection of design problems in object-oriented reengineering,” in *Proceedings of TOOLS 30 (USA)*, 1999, pp. 18–32.

[14] K. Mens, T. Mens, and M. Wermelinger, “Maintaining software through intentional source-code views,” in *Proceedings of SEKE 2002*. ACM Press, 2002, pp. 289–296.

[15] E. van Emden and L. Moonen, “Java quality assurance by detecting code smells,” in *Proc. 9th Working Conf. Reverse Engineering*. IEEE Computer Society Press, Oct. 2002, pp. 97–107.

[16] D. Ratiu, “Time-based detection strategies,” Master’s thesis, Faculty of Automatics and Computer Science, ”Politehnica” University of Timişoara, Sept. 2003.

[17] D. Ratiu, S. Ducasse, T. Gîrba, and R. Marinescu, “Using history information to improve design flaws detection,” in *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*, 2004, pp. 233–232.

[18] T. Gîrba, S. Ducasse, and M. Lanza, “Yesterday’s weather: Guiding early reverse engineering efforts by summarizing the evolution of changes,” in *20th International Conference on Software Maintenance (ICSM 2004)*, 2004.