Understanding Phases and Styles of Object-Oriented Systems' Evolution

Zhenchang Xing and Eleni Stroulia
Computing Science Department
University of Alberta
Edmonton AB, T6G 2H1, Canada
{xing, stroulia}@cs.ualberta.ca

Abstract

Understanding the phases and styles of evolution of software systems can provide valuable insight in support of project management. In this paper, we present a method for studying the evolution of object-oriented software at system/subsystem level and analyzing the underlying factors that drive its unfolding over time. This method relies on analyzing the design-level structural changes between two subsequent software versions to identify additions, removals, moves, renamings and signature-changes of classes, interfaces, and their methods and fields, represented as change trees. A sequence of such change trees constitutes the system's evolution profile. Based on discrete system evolution profiles, three types of analyses–phasic analysis, gamma analysis, and optimal matching analysis–are applied, to abstract an overall sequential pattern and structural properties of system evolution phases and to develop the typology of system evolution styles. We report on two case studies evaluating our approach.

1 Introduction

As software systems evolve, their size and complexity increase. To better deal with such size and complexity of the system, software engineers often need reverse engineering methods and tools that provide a way to extract relevant information and produce representations of the system at higher level of abstraction so that they can monitor and control the process and product in systems' evolution.

Consider, for example, a software manager, who is supervising a set of software projects by a corresponding set of development teams. She needs to continuously monitor the development process of all the teams to make sure that they are on the right track, and to intervene in a timely and informative fashion if not. She needs to know what parts of the system are being changed and how, and if there are any "interesting" events or trends, such as, for example, slow or explosive code growth. She needs to identify different evolution phases, such as functionality extensions vs. restructurings, and their interrelationships. She could also study the development history of all the projects, understand their different evolution styles and underlying factors that drive their evolving in a particular way, and use this knowledge to predict the development trajectory of similar projects to ensure they are on the right track.

There have been several research efforts to date aiming at understanding system evolution. For example, Lanza [12] introduced the "evolution matrix", a visualization of size metrics, such as *number of methods and number of instance variables*, of the entire set of system classes, the overall shape of which can be used as an indicator of the evolution of the whole system. However, such visualizations require a substantial interpretation effort on behalf of their users and tend to not scale gracefully to handle large systems.

Another, potentially more precise, source of evolutionary information could be documentation [3,6,13], either at the source-code level or at the change-log level of the version-management system used for development of the software system. Unfortunately, more frequently than not, such documentation is sparse and inconsistent [4,9].

In our work on understanding the evolution of object-oriented software system, we have adopted class models of subsequent system snapshots (which may be released versions or simply snapshots checked-out in regular time intervals) as the primary input of our method. These class models are easily obtainable, given the source code that resides in a versioning system and any of a variety of existing CASE tools [24,25], and they are, by their very nature, fairly accurate representations of the source [14]. The fundamental intuition underlying our method is that by comparing a sequence of snapshots of the system's class model, one can extract a history of the evolution of software system, in terms of the "additions", "removals", "moves", "renamings" and "signature changes" of classes, interfaces, and their fields and methods, between subsequent snapshots as they are reverse engineered from source code.

In addition to its lightweight knowledge assumptions, the other important methodological constraint of our method is that it has to be automated and not to require

much input information or output interpretation from the user.

Given the structural changes reported by the class-model comparison, the descriptive statistics, quantiles and means, are calculated to discretize their continuous values, and consequently five significant classifications of change activities are defined.

A nonparametric sequential analysis (phasic analysis) is then applied to generate a flexible phase map, a series of coherent snapshots following one on another, from the system evolution profile that is composed of the sequence of discrete change activities. Further gamma analysis [15] of phase maps identifies the general order of elements in system evolution profile and abstracts the overall sequential pattern that can be used for qualitatively evaluating the overall system evolution against its underlying development process or clustering a number of software systems according to their evolutionary patterns at higher level of abstraction. Finally, optimal matching analysis [1] has been applied to compute the *Levenshtein* distance index between two system evolution profiles. The resulting distance index is used for clustering analysis to generate typology, such as the definition of evolution styles. These analyses can address several questions:

- What types of development trajectories are there? We have developed a typology of development trajectory of software evolution, in the form of distinct evolution styles. Given this typology, we may be able to identify the most effective evolution style given the particular circumstances of a software project.
- What are the structural properties of a development trajectory? For example, we expect periodic repetitions of refactoring and functionality-extension phases, which are described as “two hats” in [8], if the eXtreme Programming (XP) methodology is being applied. Another interesting phenomenon in the development trajectory may be *breakpoints* where the nature of phase changes suddenly. They are important because they may represent unusual events in the system evolution.
- What factors influence development? Besides the adopted development process and methodology, we expect other factors, such as the adoption of a particular framework and time constraints, for example, to also influence development. Combined with the developer’s subjective view of these factors, we can investigate their influence on the software evolution.

The remainder of the paper is structured as follows. In section 2, we place this work in the context of previous related research. In section 3, we present the overall methodology and rationale of our approach. Two case studies illustrating our approach are discussed in section 4. Finally, we conclude our discussion with a summary of the lessons we have learned to date and our plans for future work.

2 Related works

There already exists a substantial body of literature on the general “software-evolution understanding” topic. In general, this work can be grouped according to the primary source of input data.

A substantially analyzed source of data has been the “history” recorded by version-management systems. Lehman et al. [13] proposed laws of software evolution after analyzing change data from the evolution of OS/360 operating system. Eick et al. [6] analyze the change history of the code, which is assumed to reside in a version management system, calculate code-decay indices, and predict the corresponding fault potential and change effort through regression analysis. The objective of this research is mainly to support project management so that code decay is delayed.

Barry et al. [3] use software volatility, in terms of amplitude, periodicity, and dispersion of software changes, to define a set of evolutionary patterns of software systems. Their work shows that systems evolve in different styles; in contrast, the method discussed in this paper focuses on understanding the structural properties of system lifecycle behaviour and the underlying factors that drive the system’s evolving.

Besides the frequent inconsistency of such history data, the major shortcoming of this line of research is that the interpretation of the version-management activities within the lifetime of a software project highly depends on the process model adopted by the developer team. Therefore, when this model is not known, as is frequently the case, inferences based on this data is quite unreliable. Our method removes the dependence on such high-quality change data by basing analyses on the recovered structural changes between snapshots of system’s class model derived from source code.

Another line of research has focused on the visualization of software-process statistics, source code metrics, static dependence graphs, etc. Eick et al. developed tools [7] for visualizing the evolution of software statistics at the source-code line level and change data such as developer, size, effort, etc.

Lanza [12] describes how to use a simple two-dimensional graph to convey the implicit information of software metrics. Based on the visualization of the evolution of class metrics (*number of methods and number of instance variables)*, the evolution matrix can be used as an indicator of the evolution phases of the whole system, similar but more coarse-grained than our taxonomy, thus failing to recognize restructuring activities.

These visualization methods require a substantial interpretation effort on behalf of their users and tend to not scale gracefully to handle large systems. They are limited

by the size of visible area of screen. The method we propose is quite automated with only a little human effort needed. Furthermore, it is relatively more scalable because it focuses on only changing parts of the system instead of all its modules.

Finally, there has been some work at analyzing the changes of software at the design level. Egyed [5] has investigated a suite of rule- and constraint-based and transformational comparative methods for checking the consistency of the evolving UML diagrams of a software system. Selonen et al. [17] have also developed a method for UML transformations, including differencing. However, they cannot surface the specific types of changes as reported by *UMLDiff* and these projects have not explored the product of their analyses in service of evolution understanding.

3 Methodology

In this section, we present our structural modification detection algorithm, *UMLDiff*. We discuss the discretization of recovered structural modifications and the consequent five classifications of change activities. We present three types of sequential analyses given the system-evolution profiles that are composed of the sequences of discrete change activities. As well, we discuss how the recovered knowledge may assist software engineers in their task of understanding software evolution and the influence of the underlying factors that drive the system's evolving.

3.1 *UMLDiff:* Class model change detection

The overall problem of detecting and representing changes to data is important for version and configuration management. It is an active research area on its own in the area of data management. Probably the most well known algorithm for text comparisons, *GNU diff*, was discussed as the string-to-string correction problem using dynamic programming in [19]. Used in the context of code differencing, it reports changes at the code line level.

As more data and documents are stored in XML format, some sophisticated version control systems include XML-aware features to handle XML documents. The general tree-to-tree correction problem has been studied extensively [2], and has been applied to show differences between XML data [23]. However, such general tree-differencing algorithms report changes as "XML element modifications" ignoring the domain-specific semantics of these nodes. Let us consider UML class model represented in XML Metadata Interchange (XMI) format as an example. When an attribute or method was moved from one class to another, a general XML-differencing tool would most likely report two separate activities: the addition of some xml nodes and the deletion of some others, but would not recognize the move of features between objects, since it does not understand the UML class model semantics.

Recognizing changes at this higher level of abstraction and taking into account the UML-specific semantics is exactly the motivation of *UMLDiff*. If we rely on the structural information captured in the class model, we could identify such activities as "moves"; this is because the granularity of changes is larger and correspondences between additions and deletions could be explored to uncover higher-order changes such as "moves". In the context of software evolution, where local transformations, such as refactoring, frequently involve moving features between objects, recognizing such "moves" is essential. A "move" operation often represents the redistribution of information or the reorganization of the class hierarchy, modifications that are usually part of perfective changes that are intended to improve the developer's ability to maintain the software without altering functionality or fixing faults. Thus, it is important that we could recover the movement of objects when analyzing software evolution, in order to recognize "perfective maintenance" phases in the software life span. Figure 1, discussed below, shows an example of such "move" operation.

In general, the problem of detecting the class-model changes between two snapshots of an object-oriented system can be viewed as a graph-difference problem, since class models can be viewed as specific types of directed graphs. This problem is NP-complete which makes an automatic approach impractical. Therefore, we have limited our initial exploration to considering only the inheritance hierarchies of the class model.

The algorithm underlying our evolution analysis work, *UMLDiff,* is essentially a domain-specific structural differencing algorithm, aware of the UML semantics. It takes as input two UML class models, corresponding to two snapshots of the system under analysis, represented in XMI. Such class models can be either produced in the software-design phase by the system developers, or they can be reverse engineered from the source code, using any of the currently available software engineering tools [24,25]. The first step of the algorithm is to parse the input forests of class models into two labeled tree structures, in which the tree nodes are labelled with the type of objects, such as class, interface, field, method, and their corresponding attributes, such as identifier, modifiers, data type, parameter list, etc. The target representation contains the application classes and interfaces, their fields, their methods and their inheritance, implementation, and nested relations.

The next step of the algorithm is to identify the *after-before* changes between them, in terms of the "additions", "removals", "moves", "renamings" and "signature changes" of classes, interfaces, and their fields and methods. This UML differencing process brings to the surface structural modifications to the software design. The results are represented as *change trees*, i.e., trees of structural modifications, which, if applied to the *before*

snapshot would result in the *after* snapshot. Change trees are represented in an XML-based syntax. Figure 1 diagrammatically depicts an example change tree.

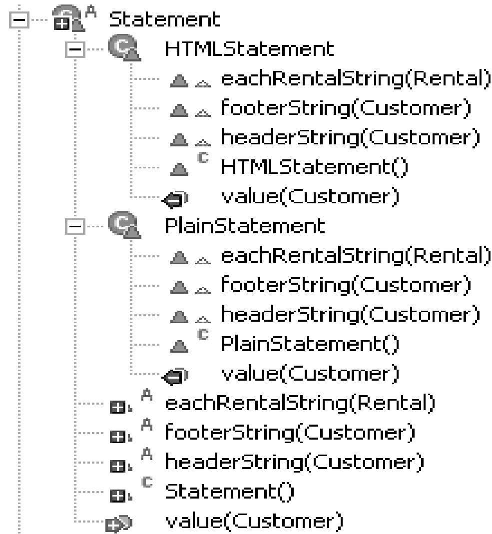

Figure 1 An example of change tree

The different icons to the left of each node represent the different object-oriented entities: *class, interface, field*, and *method*. The top-right adornment shows the attributes of the object, for example, *abstract*, *constructor*, *static*, *final*. The bottom-right adornment represents method overrides (small filled triangle) or implementation (empty triangle). The bottom-left adornment represents the status of a particular object: it can be the *plus* sign for *add*, *minus* sign for *remove*, *01* for *rename/change signature*, *arrow* with a *minus sign* for *move out from source*, *arrow* with a *plus sign* for *move into target*. The change tree presents the detailed structural modifications to the class model as software system evolves from one version to another.

The change tree in Figure 1 corresponds to the differences between version 27 and 28 of the extended refactoring sample from M. Fowler's book [8] as found in [21]. It shows that a new abstract class, *Statement*, was created with three newly created abstract methods, *eachRentalString*, *footerString*, and *headerString*. The *value* methods of its two subclasses, *HTMLStatement* and *PlainStatement*, were pulled up into the new class *Statement*. Their corresponding methods, *footerString*, *eachRentalString*, and *headerString* in version 28 match their counterparts in version 27 (no sign means matching) but they implement now the corresponding newly declared abstract methods in *Statement*. These structural changes reported by *UMLDiff* represent the modifications to the class model after a "Form Template Method" refactoring, which is described as follows: "if you have two methods in subclasses that perform similar steps in the same order, yet the steps are different, get the steps into methods with the same signature, so that the original methods become the same. Then you pull them up [8]"

3.2 Classifying structural modification

Overall, there are four types of structural modifications that can be identified by *UMLDiff*.

- Addition of new classes, interfaces, methods, and/or fields;
- Removal of classes, interfaces, methods, and/or fields;
- Movement of methods and/or fields from one class to another; and
- Modifications of signatures, including renaming, of classes, interfaces, fields and/or methods.

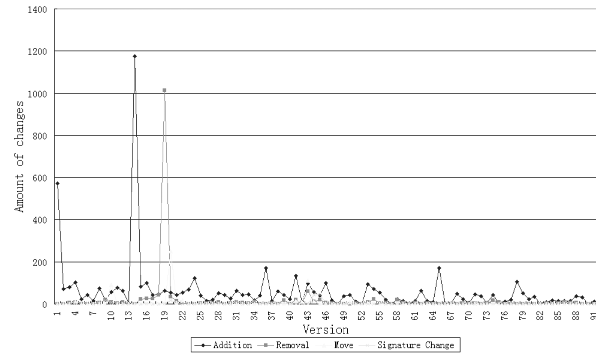

Figure 2 The evolution history of Mathaino

Between any two snapshots of the system evolution, the types and amounts of structural modifications may vary greatly. Figure 2 shows the system evolution history of one of our two case studies, the Mathaino project [11]. Clearly, with this level of detail it is difficult to discern patterns, such as evolution phase and style, for example.

In order to help discern potential patterns of the evolution of software system, we use a discretization technique to reduce the number of values for the given types of structural modifications, by dividing the range of the attribute into intervals. Interval labels can then be used to replace actual data values. To make the discretization more meaningful, we calculate descriptive statistics, quantiles and means, for each type of modification.

Table 1 Classifications of change activities

Type	Combination of Labels	Description
A	(a_H or r_H) & (m_H or o_H)	Active
B	(a_H or r_H) & (m_L and o_L)	Rapid developing
C	!(a_H or r_H) & (m_H or o_H)	Restructuring
D	(a_M or r_M) & (m_L and o_L)	Slow developing
E	(a_L and r_L) & (m_L and o_L)	Steady-going

The quantile is the specific value of a variable that divides the distribution into two parts, those values greater than the quantile value and those values that are less. For example, p percent of the values are less than the pth quantile. We calculate 10% and 90% quantiles for modification of type "Addition" and "Removal". The values are marked "Low" if it is less than the value of 10% quantile, and "High" if greater than the value of 90% quantile. The values fall in between them are marked as "Medium". We also compute the means of "Movement"

and “Signature change” modifications. The values of these two types are marked as “High” if above the mean, or “Low” otherwise. These interval labels are combined to define five significant classifications of change activities when software system evolves (See Table 1).

The character a, r, m, and o represent Addition, Removal, Movement, and Modification of signature respectively. The subscript H, M, and L represent High, Medium, and Low respectively when discretizing continuous values. Type A represents the period of system evolution, where the number of newly created and/or removed entities is High and the number of moved methods/fields and/or signature changes is also High. It represents the most Active portion of system evolution. The high number of move and signature changes probably implies a lot of perfective maintenance activities happened during this period, such as move features between objects, make the method calls simpler, deal with generalization, etc. The high number of newly created and/or removed entities could be the result of functionality extension, defects fixing, etc.

The difference between type B and A is that type B does not include many perfective maintenance activities, while the number of newly created and/or removed entities still remains High. The software system develops rapidly during this period of time. In contrast, type C represents the inverse situation to type B. It contains many maintenance activities that result in the High number of move and signature change, while the number of newly created and/or removed entities remains Medium or Low. So it is called restructuring.

Types D and E represent the slow developing and steady-going periods of system evolution where the number of maintenance activities remains Low and the number of other activities is Medium or Low.

3.3 Analyzing evolution phases and styles

UMLDiff reports the changes between two snapshots of a system’s class model. There are *N* such models in an evolving software system with *N* successive snapshots, and consequently *UMLDiff* can be applied *N-1* times to generate the differences between the $(I+1)^{th}$ and I^{th} snapshots, where $1 \leq I < N$. Thus, *N-1* change trees can be obtained that record the structural modifications when software system mevolves from one snapshot to another.

We think of a change tree, plus the first snapshot, as a transaction that records the system classes that have been modified (including creation and removal) in the corresponding snapshot. For a software system with *N* snapshots, its corresponding system-evolution profile of length *N* is generated, which is composed of a sequence of *N* discrete change activities.

This section discusses three techniques for the sequential analyses on system-evolution profiles. The applications of these analyses will be discussed in next section.

Phasic analysis is one of two primary forms of sequential analysis, which has been widely applied in longitudinal analysis of temporal processes of social phenomena. It works with nominal and categorical data and has been successfully applied in the study of information system development process [16].

Overall, sequential analysis consists of the identification and testing of a model or explanation that accounts for the characteristics of long series of time-ordered observations. It assumes that social behaviors can be described in units larger than individual acts, which can cohere into phases or patterns in the developmental path of a social event. These patterns are the result of dynamics that drive the changes over time [10].

We think of the evolution of software system as a social event that unfolds and changes over time. They develop according to some form of underlying process. They evolve to fix defects, meet customer-driven functionality enhancements, adapt to changes in the deployment environment, and so forth. Thus, we believe that the phasic analysis is a promising way to understand the general evolution of software systems and discover how they are born, develop, and terminate, and the processes that drive their unfolding.

We use *WinPhaser* to make sequential phasic analysis on the discrete system evolution profiles. *WinPhaser* is a set of sequence description and analysis tools [10] for the study of human interaction in small groups, but the tools can be applied to any sequential data consisting of a time-ordered set of discrete elements.

WinPhaser generates a flexible phase map from a data sequence consisting of discrete data elements. A phase map is composed of a series of coherent units. *WinPhase*’s flexible phase mapping module allows it to parse the data sequence into phases of different length based on shifts in the date sequence. It labels the phase with the type of predominant elements in that phase and identifies noisy periods with no predominant elements as *pending* phases. Therefore, in our use of *WinPhaser*, three are six different types of phases that could be identified in the system evolution profiles. They are: *Active, Rapid developing, Restructuring, Slow developing, Steady-going* and *Pending*.

In addition to phasic analysis, *WinPhaser* also provides two more tools for sequence analyses[1].

Gamma analysis [15] provides a measure of the general order of elements in a sequence and a measure of the distinctness or overlap of element types. It is a nonparametric statistic, based on the Goodman-Kruskal gamma, that is a measure of the proportion of A events that precede or follow B events in a sequence. A pairwise gamma is given by P-Q/P+Q, where P is the count of A events preceding B events and Q is the count of B events

[1] The description of gamma analysis and optimal matching analysis is excerpted from *WinPhaser* help.

preceding A events. Gamma analysis of a sequence yields a table (see Table 2) consisting of pairwise gamma scores for each possible pair of element types.

Based on the pairwise gamma-analysis data, the precedence and separation score are calculated for an element type. The precedence score is given by the mean of its pairwise gamma scores. The precedence score indicates the location of the element in the overall ordering of element types and can range from -1 to 1. A phase score of -1 occurs in the beginning of a sequence. A phase score of 1 occurs at the end of the sequence. The separation score for an element type is given by the mean of the absolute value of its pairwise gamma scores. It is a measure of the relative distinctiveness of the element type and can range from 0 to 1. Separation approaches 1 as a larger proportion of the units of a given element type occurs contiguously. An element will obtain a lower separation score if it occurs at several widely separated points in a sequence.

Gamma mapping is the final step in gamma analysis. Precedence and separation scores are used to construct gamma maps. Element types are ordered sequentially on the basis of precedence scores (from smallest to largest). Phases with separation scores below .50 are not clearly separated from other phases.

Gamma analysis is able to abstract an overall sequential pattern from a phase map. The resulting gamma map is simple enough, as shown in section 4, to be used for qualitatively evaluating the overall system evolution against its underlying development process or clustering a number of software system according to their evolution patterns at this higher-level of abstraction.

Optimal matching analysis provides a generic tool for sequence comparison when each sequence is represented by well-defined elements drawn from an alphabet of a relatively small set of (repeating) event types.

Sequence comparison methods are applied in areas as diverse as computer programming, molecular genetics and birdsong analysis. For example, a set of discrete change activity sequences of system evolution may be compared in order to determine the relationships between them.

It is possible to qualitatively clustering a small collection of short sequences, such as those abstracted by gamma analysis, into sets, but such qualitative clustering is likely to miss important distinctions among larger collections of complex sequences. Large data sets of long sequences would be better compared if we could quantify the similarity or dissimilarity of any two sequences in a numerical distance index. However, in cases like change-activity sequences where elements may repeat, the traditional methods of enumeration, permutation statistics, or scaling are not appropriate [1].

Optimal matching analysis is designed to handle such situations. It produces an index of the relative "distance" between any two sequences. This index, called the *Levenshtein distance*, is the smallest possible cost of operations of insertion, substitution and deletion of sequence units required to align two sequences, that is, to transform one sequence into the other. The more similar the sequences being compared, the fewer operations required to align them and the smaller the index of distance or dissimilarity.

We illustrate the concept of optimal matching using a simplified example. For the purposes of this discussion we use a simple alphabet of five possible sequence elements: A, B, C, D and E, and three short sequences composed of these elements:

- Sequence 1: E D C D E
- Sequence 2: E D C E D E
- Sequence 3: E D B C E B E

An intuitive judgment of these three sequences based on the alignment of shared elements is that sequence one and sequence three are most dissimilar, with sequence two resembling one and three more than they resemble each other.

For the purposes of this example, the costs of insertions, deletions, and substitutions will be set at 1.0. Now let's see how these sequences align. For sequences one and two, there are five elements already aligned. The insertion of E in sequence one or the deletion of E from sequence two will align the two sequences completely. The cost to align them is 1.0. In the case of sequences two and three, there are five aligned elements and one substitution of D with B and one insertion of B required to complete the alignment. The cost is 2.0. The final pair, sequences one and three, displays only four aligned elements. One substitution and two insertions are required, so the cost is 3.0.

The above calculation echoes our earlier judgment of the relative similarity of the three sequences. The sequence one is most similar to the sequence two and is most dissimilar to the sequence three.

Of course, sequence alignment is seldom as simple as in this example. The three sequences discussed here are actually the phase maps of three students' software projects that will be discussed in section 4.2. For the purpose of intuitive illustration, the cost for different types of operations is here equally set at 1.0. However, more realistically and more generally, each operation should be assigned a cost that represents the difficulty of making that change [2]. The cost can be thought of as the perceived unlikelihood of the change having arisen at random in whatever process produced the changed unit. For example, as we discussed in section 3.2, element E represents the steady-going phase of system evolution where few changes have been made, while B represents rapid growing phase and C represents restructuring. Therefore, in the cost model for the alignment of two phase maps of system evolution, the cost to substitute C with B should be much higher than that to insert a E, since B and C are very

different type of phases, but phase type E is almost equal to nothing.

Furthermore, there are many possible sets of operations to align any two sequences. Consider sequences one and three again. If we align the 5th E of the two sequences, then we can substitute the third and fourth elements and insert BE at the end. It then takes four operations to align them. The dynamic programming algorithm is often applied to always reveals the least possible cost for the alignment.

A single distance index provides only a numerical value for the dissimilarity between two sequences. In and of itself, this index may not be especially interesting. But typically a larger set of sequences are compared and the results are used in one of two ways:

- If we have an ideal sequence, possibly by hypothesizing a particular software-development process, we can then compare a set of empirical data sequences to the expected sequence in order to gauge their relative fit to the model. For example, as discussed in the second case study (section 4.2), we may be able to produce an ideal developmental model as we accumulate enough data that can be used for distinguishing good teams from bad teams.
- Given enough data sequences, we can compare all the sequences in a data set to each other. The resulting distance matrix can then be subjected to cluster analysis in order to generate a sequence typology, such as different evolution styles discussed in next section.

4 Case studies

The objective of our evolution analysis work is to support software engineers to understand software evolution at the higher level of abstraction by identifying and analyzing the evolution phases and styles of software system. In this section, we discuss two case studies that we conducted to evaluate the effectiveness of our method.

4.1 Longitudinal evolution analysis

Mathaino [11] is a research prototype tool that can be used to migrate text-based legacy interfaces to modern web-based platforms. It was developed by following a strict refactoring-based development process, inspired by XP methodology. It underwent 91 builds from July 2000 till February 2001. The first version has 29 classes, 284 methods, and 256 attributes. The last version has 144 classes and about 1800 methods and 1800 fields.

91 class models have been reverse-engineered from their corresponding code versions and then *UMLDiff* has been applied to surface the structural modifications between subsequent versions. The values of structural modifications reported by *UMLDiff* were discretized and each version was therefore classified into one of five possible change activity categories, *Active, Rapid developing, Restructuring, Slow developing, and Steady-going*. Finally, the system evolution profile of 91 discrete change activities was generated, which has been analyzed by *WinPhaser* to generate the phase map of the developmental path of Mathaino project. The minimum phase length was set at 2. Table 2 lists the resulting phase map in detailed format with the identified phase names, their staring and ending versions, and their length.

19 phases have been identified, greatly reducing the length (91) of system evolution profile, which enhances our chance to get a better overall comprehension of the system under investigation. Most phases of *active, rapid developing,* and *restructuring* appear before the 9th phase, while almost all phases of *slow developing* and *steady-going* appear after 9th phase. The phase map reveals clearly that the Mathaino project was much more active in its first half development process than in its second half. Mathaino started with a *rapid developing* phase of length 3 that indicates its rapid growth in its very beginning. A *restructuring* phase of length 3 follows the starting *rapid developing* phase, which suggests that the developer of Mathaino might have realized he needed to restructure the code a little bit before any further development. This restructuring phase has not been documented in the developer's report [18].

Table 2 The phase map of Mathaino's evolution

#	Phase	Starting	Ending	Length
1	B	01	03	3
2	C	04	06	3
3	A	07	11	5
4	B	12	19	8
5	A	20	21	2
6	B	22	24	3
7	D	25	40	16
8	A	41	46	6
9	P	47	50	4
10	E	51	52	2
11	A	53	54	2
12	P	55	60	6
13	D	61	62	2
14	P	63	67	5
15	D	68	73	6
16	B	74	75	2
17	P	76	78	3
18	D	79	89	11
19	E	90	91	2

A cycle of *active* and *rapid developing* phase appears from the 3rd to the 6th phase, which is followed by a long *slow developing* phase of length 16 (from version 25 to 40), which is over at version 41 where another *active* phase of length 6 appears. In his own report [18], the developer described the major refactoring made during this period of time. The generic navigator interfaces were extracted to improve the compatibility of its platform independent GUI components.

Since the 9th phase (from version 47), the system went into a relative stable status represented by a large portion of *slow developing* and *steady-going* phase in the second half of development of Mathaino system. There exist four pending phases (represented as P) that generally mix up the change activities of different types, such as, for example, the portion of system evolution profile of phase 12 is CDEBCE.

Table 3 Pairwise gamma scores

	A	B	C	D	E	P
A	.000	.267	1.000	-.543	-.875	-.944
B	-.267	.000	.600	-.817	-.867	-.778
C	-1.000	-.600	.000	-1.000	-1.000	-1.000
D	.543	.817	1.000	.000	-.457	-.222
E	.875	.867	1.000	.457	.000	.222
P	.944	.778	1.000	.222	-.222	.000

Table 4 Precedence scores

A	B	C	D	E	P
-.219	-.426	-.920	.376	.684	.504

Table 5 Separation scores

A	B	C	D	E	P
.726	.666	.920	.568	.684	.593

Whether from the perspective of phase map, such as from phase 1 to 6, or at the detailed level of system evolution profile, such as the portion defined by phase 12, an intuitive impression is that *active, rapid developing,* and *restructuring* phases often follow one another. There exist phase cycles or repetitions among them. Since *active* and *restructuring* phase represent the existence of a lot of restructuring or refactoring activities and *rapid developing* phase represents the major functionality extensions, does that mean there exists some sort of repetitive relationship between functionality extensions and refactorings? To answer this question, the general order of change activities in Mathaino's system evolution profile has been analyzed through the gamma analysis.

The pairwise gamma scores are computed for each possible pair of phase types as shown in Table 3. Table 4 and Table 5 shows the corresponding precedence and separation scores for six phase types. Finally, the gamma map is constructed as follows:

(((C))) ((B)) ((A)) ((D)) ((P)) ((E))

It displays the phase types in their precedence order and indicates, with ( ), the overall separation score for each type. The separation scores of all six phase types are greater than 0.50, which means these phase types are distinct enough from each other. This also increases our confidence in the correctness of classifying the change activities into these types. Gamma map abstracts an overall sequential pattern from the phase map. It indicates the general order of phase types in system evolution profile, which echoes our intuitive judgment of the relationship of *active, rapid developing,* and *restructuring* phases, which is defined as "two hats" in [8]. The refactoring-based software development involves two distinct activities: adding function and refactoring. Developers adopting this style swap hats regularly.

Mathaino adopted a strict refactoring-based development process. Thus, it is expected that we could identify such phase repetitions in its evolution. Its developer started by trying to add new functions, and he realized he should restructure the code somehow so that it would be easier for him to add such new functions. So he swapped hats and refactored (*restructuring* phase). Once the code was better structured, he swapped hats and added the new function (*rapid developing* phase). Once he got the new function working, he realized it was developed in a way that's awkward to extend, so he swapped hats again and refactored (*active* phase). The pending phase, where no predominant change activity types exist, indicates he probably swapped hats frequently. Mathaino then went into *slow developing* phase and was gradually stable (*steady-going* phase).

Software managers can use the overall sequential pattern abstracted by the gamma map to check if or not the development of a particular software system follows the adopted methodology. Furthermore, since the gamma map is a very simple representation of system evolution, they can also use it to qualitatively cluster a number of software systems according to their general sequential evolutionary patterns, which could be further studied, together with other factors such as budget, time constraints, etc., to address such question as "what is the most effective evolution pattern given the particular circumstances of a software project?"

4.2 Collaborative software development of small undergraduate teams

In this subsection, we discuss a case study of the term projects of five undergraduate teams that took place during a single-term (about four months) software engineering course. This course was organized in a three-phase lifecycle, and the deliverables of three phases were software design, user-interface prototype implementation and complete implementation. Currently, these three deliverables were the points where the instructor can identify design problems. Since the student-provided documentation is not consistent throughout the various documents, comparative analysis is difficult and problems may go undetected. We chose to evaluate our evolution understanding method on this data, in order to explore its potential impact on monitoring and controlling project development at this high-level of abstraction.

We took weekly snapshots of the projects from the CVS repository, from January 20th, 2003 through April 14th, 2003, resulting in 13 versions for each project. Although the "official" project length was 13 weeks, only 5 to 7 different snapshots were actually obtained for each project. Therefore, the minimum phase length was set at 1.

The phasic analysis has then been applied on this data set. Table 6 shows the phase maps for software projects of five student teams in Run-Length Encoded (RLE) format that identifies the length and name of each phase in the map.

Table 6 Phase maps of five student teams

Team	Phase Map
(a)	5E 2D 2C 1E 1D 2E
(b)	6E 2D 1B 1E 1D 2E
(c)	7E 1D 1B 1C 1E 1B 1E
(d)	4E 1D 1B 1D 2C 1E 1C 2E
(e)	7E 3D 1C 1D 1E

RLE sequences have been parsed by *WinPhaser* optimal matching analysis module, their *Levenshtein* distance indices revealed that the development trajectories of teams (a) and (e) are similar, those of team (b) and (c) are similar, while team (d) has the most distinctive developmental path. We therefore define the typology of software systems' evolution according to the main characteristics of their developmental paths in the context of undergraduate software engineering course: *continuous small modifications* and *occasionally large modifications.*

The main characteristic of the developmental paths of teams (a) and (e) is that they proceeded to develop one step at a time. Their change activities involved *continuous small modifications*. They didn't exhibit the aggressive growth spurts, such as those of team (b) and (c), which means they did not try to implement their project at the last minute before the deadline like teams (b) and (c).

The evolution processes of teams (b) and (c) contain phases with aggressive growth spurts exhibited as the *breakpoints* (*rapid developing* phases) where unusual events happened. Their projects started with a few classes and did not change a lot until week 8. However, there is a sharp increase in the size of their projects at week 9, which is followed by small changes. These *occasional large modifications* coincide with the deadlines for project part 2. This means that most features and/or functionalities of their projects were implemented or checked into the CVS repository just before the deadline−a bad but not untypical practice. Teams (b) and (c) exhibited similar evolutionary development styles. However, since team (c) adopted the Model-View-Controller (MVC) model as the application architecture, their work is more organized than that of team (b), such as, for example, there are some maintenance activities (exhibited as a *restructuring* phase) in the middle of their project development, and their project quality as evaluated by the course TA (for each deliverable, all team projects were marked by the same TA) was better. This result validates our intuition that good architecture enables software quality.

Team (d) exhibited a very interesting evolution style. They started their project development much earlier than other four teams. The most changes were made within the first two consecutive phases when they started the development of their project, in weeks 5 and 6. Their change activities are just the opposite to those of most other teams. Most other teams added new functions (exhibited as *rapid* or *slow developing* phase) when the project deadline was approaching in week 9 and 11. However, the most remarkable thing for team (d) is that they refactored and performed some perfective maintenance activities (exhibited as *restructuring* phases) in weeks 9 and 11, which means they were trying to improve the code structure, when most other teams were struggling to meet their requirements. Team (d) may have a very good requirement analysis and high-level system design in the first place. Therefore, they seem to know what architecture should be adopted, what functionalities should be supported, and further how to implement them. In that way, they were able to put almost everything in place when they started implementation at the very early stage. Actually, they obtained consistently the highest mark for all their project deliverables.

Although this case study was done for the relative simple undergraduate course projects, we believe the results are not limited in the context of undergraduate software engineering education. The sequential pattern analyses based on *UMLDiff* enable a better understanding of the nature of each individual project evolution. It also validates the general assumption that there exist many factors, such as time constraints, the adoption of good architecture, and the upfront design, which influence the developmental path of software system. A good team like team (d), who met all these criteria, obtained the highest mark among these five teams. They adopted the MVC model, they obtained the best mark for the first deliverable which is essentially a requirements-and-design document, and started their implement the earliest. In contrast, team (b) obtained the poorest mark. Their development profile seems fairly ad-hoc, their architecture is poor, and they had to do a substantial last-minute development to meet the deadline. By accumulating more data, software managers may be able to construct an ideal developmental model that can be used for distinguishing good development practices from bad ones, which could assist them in better monitoring and assessing software system.

5 Conclusion and future work

In this paper, we discussed our method on understanding evolution phases and styles of object-oriented software system and their relationships with various underlying factors, such as the adopted development methodology, which drive the system's evolving. This method relies on readily available data, as opposed to consistently documented software project change logs and is scalable since it focuses only on the changing part of the system. It takes as input a sequence of class models of the system, reverse engineered from a corresponding set of code versions. These models are compared using the design-level structural differencing

algorithm, *UMLDiff*, to surface various types of changes made to the system's class model over time.

Based on the system evolution profiles that are composed of the sequences of discrete change activities, three types of sequential analyses are then applied. We have discussed potential applications of knowledge revealed by these analyses about system evolution phases and styles in assisting software engineers in qualitatively evaluating the overall system evolution against its underlying development process, clustering a number of software system according to their evolutionary patterns, and abstracting the ideal developmental model, etc.

The approach of understanding system evolution presented in this paper has been implemented as a part of one of two analysis plugins in Eclipse [22], in the context of the JRefleX project [20]. In addition to the *UMLDiff* algorithm and three types of sequential analyses we have already discussed, the plugin has also implemented several visualization instruments, such as change tree (see Figure 1) and system evolution matrix, etc., to present analysis results.

We are now working on two extensions of this work. First, we are developing a fact extractor plugin in Eclipse to remove the dependence on external tools [24,25], which enable us to handle more types of relations other than class hierarchy, such as method calls, field accesses, etc. This will enrich the accuracy and types of output of *UMLDiff* algorithm. Second, we are conducting a similar case study on a much more complex industrial-wise software system, Eclipse, an open source Java IDE, which is built on an extensible plugin framework. The core of Eclipse has more than 60 plugins, most of which have several dozens of revisions. Its core plugins have been divided into several subgroups, such as compare support, team support, search, user interface, etc., which have been developed in separate IBM research branches. We believe Eclipse provides a good test bed to further evaluate our method, through which we are expecting to be able to refine the classifications of evolution phases, enrich the typology of evolution styles, and study further the influence of various factors on software evolution.

References

1. A. Abbott and A. Hyrcak, "Measuring resemblance in sequence data: An optimal matching analysis of musicians' careers", *American Journal of Sociology*, 1990(96):144-185.
2. D. Barnard, G. Clarke and N. Duncan, "Tree-to-tree Correction for Document Trees", Technical Report 95-375, Queen's University, January 1995.
3. E. J. Barry, C.F. Kemerer, and S.A. Slaughter, "On the Uniformity of Software Evolution Patterns", *Proceedings of the 25th International Conference on Software Engineering,* 2003, pp. 106-113.
4. E.J. Chikofsky and J.H. Cross, "Reverse engineering and design recovery: A taxonomy", *IEEE Software*, January 1990, pp 13-17.
5. A. Egyed, "Scalable Consistency Checking between Diagrams - The VIEWINTEGRA Approach," *Proceedings of the 16th IEEE International Conference on Automated Software Engineering,* San Diego, USA, 2001.
6. S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data", *IEEE Transactions on Software Engineering*, 2001, 27(1):1–12.
7. S. G. Eick, T.L. Graves, A.F. Karr, A. Mockus, and P. Schuster, "Visualizing Software Changes", *Software Engineering,* 2002, 28(4):396-412.
8. M. Fowler, "*Refactoring: Improving the Design of Existing Code*", Addison-Wesley, 1999.
9. M. W. Godfrey and Q. Tu, "Evolution of Open Source Software: A Case Study", *Proceedings of ICSM,* 2000.
10. M.E. Holmes and M.S. Poole, "Longitudinal analysis", *In S. Duck & B. Montgomery (Eds.), Studying interpersonal interaction,* pp 286-302, 1991.
11. R. Kapoor and E. Stroulia, "Mathaino: simultaneous legacy interface migration to multiple platforms", *Proceedings of 9th International Conference on Human Computer Interaction,* 2001.
12. M. Lanza, "The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques", *Proceedings of International Workshop on Principles of Software Evolution,* 2001.
13. M. M. Lehman and L. A. Belady, "Program Evolution-Processes of Software Change", Academic Press, London, 1985.
14. G. Murphy, D. Notkin, and K. Sullivan, "Software Reflexion Models: Bridging the Gap between Source and High-Level Models", *Proceedings. ACM SIGSOFT Symposium Foundations of Software Engineering*, 1995
15. D.C. Pelz, "Innovation complexity and the sequence of innovating stages", *Knowledge: Creation, Diffusion, Utilization*, 1985(6):261-291.
16. R. Sabherwhal and D. Robey, "An empirical taxonomy of implementation processes based on sequences of events in information system development", *Organization Science*, 1993(4):548-576.
17. P. Selonen, K. Koskimies, M. Sakkinen, "Transformations between UML Diagrams", *Journal of Database Management*, Vol. 14, No. 3, 2003.
18. E. Stroulia and R. Kapoor, "Metrics of Refactoring-based Development: An Experience Report", *Proceedings of the 7th International Conference on Object-Oriented Information Systems*, Calgary, AB, Canada, 27-29 August 2001, pp. 113-122.
19. R. A. Wagner and M.J. Fischer, "The string-to-string correction problem", *Journal of the ACM,* January 1974, 21(1):168-173.
20. K. Wong, W. Blanchet, Y. Liu, C. Schofield, E. Stroulia, and Z. Xing, "JRefleX: Towards Supporting Small Student Software Teams", IBM Eclipse Workshop at OOPSLA 2003.
21. http://www.cs.unc.edu/~stotts/COMP204/refactor.
22. Eclipse, http://www.eclipse.org.
23. Mosell EDM Ltd, http://www.deltaxml.com.
24. Rational Rose, http://www.rational.com.
25. Together, http://www.togethersoft.com.