Published in IET Software
Received on 24th October 2008
Revised on 17th December 2009
doi: 10.1049/iet-sen.2008.0084

ISSN 1751-8806

Component co-evolution and component dependency: speculations and verifications

L. Yu[1] *A. Mishra*[2] *S. Ramaswamy*[3]

[1]*Computer and Information Sciences Department, Indiana University South Bend, 1700 Mishawaka Avenue, P.O. Box 7111, South Bend, IN 46634, USA*
[2]*Department of Computer & Software Engineering, Atilim University, Incek, Ankara, Turkey*
[3]*Computer Science Department, University of Arkansas at Little Rock, Little Rock, AR 72204, USA*
E-mail: ligyu@iusb.edu

Abstract: Software component interaction is essential for realising proper software system functions. Such interactions between software components induce interdependencies between multiple components. One effect of such a dependency is co-evolution, wherein changes made to one component also requires corresponding changes to other component(s). This study presents a mathematical framework for representing component co-evolution. Two types of co-evolution, internal co-evolution and external co-evolution are defined for an evolving software component. The component dependency metrics that are related with component co-evolutions are analysed and the correlations between component dependency and component co-evolution are hypothesised. Further, in a quasi-experiment of nine open-source Java projects, component dependencies are measured and component revision histories are mined to verify the speculated correlations.

1 Introduction

Software components need to interact with each other so that the system can work properly. When there are interactions between software components, there is some degree of dependence between them [1–3]. Too many dependencies between software components can make the resulting software system difficult to maintain. One such effect of component dependency is component co-evolution: changes made to one component that requires corresponding changes to other components [4]. Component co-evolution is an important issue for software maintenance: (i) co-evolution increases the programmers' effort in identifying the necessary co-evolving components; (2) co-evolution increases programmers' effort in testing changes made to software component to ensure no regression faults are introduced in the system; and (iii) insufficient or incorrect co-evolution may introduce errors leading to subsequent system failures.

Owing to the significance of co-evolution, some research has been done to understand, predict and control component co-evolution. Zimmermann *et al.* [5–7] applied data mining techniques to software version history and implemented a tool to detect related component co-evolutions in a maintenance activity. Ying *et al.* [8] applied data mining techniques on the change history data of source code and determined association rules of component co-evolution (sets of files that were changed together). Hassan and Holt [9] proposed several heuristics to predict component co-evolution and presented a framework to measure the performance of the proposed heuristics.

Component co-evolution knowledge is also applied in software quality control. Graves *et al.* [10] used revision history data to predict the distribution of incidences of faults in a software release. They found that the co-evolution patterns identified based on version history can be used to predict possible fault locations for developers who perform maintenance tasks. Williams and Hollingsworth [11] used the source code revision history to detect co-evolution patterns. The studies showed that their bug-finding technique outperforms the same static analysis technique that does not use the revision history data. In other similar research, Lemos [12] analysed software quality based on the interactions and co-evolutions between components.

Most of the previous reported research focused on predicting component co-evolution based on historical data. However, predicting co-evolution based on historical data has its own limitations. One such effect is called concept drift [13], which shows that because of the structure change of a software program, historical data gradually lose its accuracy in predicting co-evolution [14]. This weakness will become apparent when we are dealing with the long lived and significantly changed open-source software systems [14].

The motivation of this study is therefore summarised as follows:

- Software systems are one of the most complex man-made systems and evolutions of software systems are one of the most complex processes of complex systems. Any research method is not expected to uncover the myth that governs complex system evolution; it only provides us one perspective of this complex process. This study intends to look through software evolution from a different perspective and with different methods.

- Existing techniques, such as predicting co-evolution based on historical data is weakened by the concept drift effect as software evolves [14]. This study intends to study software co-evolution based on both component dependency and historical data in order to discover other methods that can overcome the weakness of existing techniques.

This paper discusses the relationship between component co-evolution and component dependency. To the best of our knowledge, the most related work is performed by Yu [4], who studied the revision history of Linux kernel components. The study found that linear correlations exist between co-evolution activities measured in evolutionary coupling, which is based on component revision history, and component interdependency measured in reference coupling, which is based on component logical interactions. This study extends Yu's previous work on correlating software co-evolution and software dependency. The major additions of this work over his previous work are (i) different dependency metrics are utilised and different co-evolution metrics are defined: formal mathematical models are introduced in this paper to make the study align the main stream of empirical software engineering research; (ii) different speculations are formulised in order to study different properties of component co-evolution; and (iii) studies are performed on different software projects: in previous work [4], the speculations are verified on one software system, in this study, the speculations are evaluated on several open-source systems in order to evaluate their applicability.

This study conceptually contains two parts. In Part 1, the component dependency properties that are related with component co-evolutions are identified, analysed and the correlations between component dependency and component co-evolution are speculated as null hypotheses. In Part 2, a quasi-experiment is performed on nine Java projects. For each project, dependencies between Java components are measured and co-evolution histories are mined to verify the speculated correlations.

This study makes the following contributions to the field of software maintenance and evolution:

- a mathematical framework is established for representing component co-evolution, in which, two types of component co-evolution, internal co-evolution and external co-evolution are defined; and

- the correlations between component dependency and component co-evolution are analysed and verified through a quasi-experiment of nine Java projects.

With metrics and knowledge provided in this study about software component co-evolution, our final objective is to (i) help software engineering practitioners incorporate co-evolution in their quality measurement and build corresponding CASE tools to monitor their products evolution; and (ii) help software engineering researchers to better understand software maintenance and evolution in order to design more maintainable and evolvable software systems that can meet the fast growing customer demand and fast changing working environment.

The remainder of the paper is organised as follows. Section 2 describes software component and presents a mathematical framework for representing component co-evolution. Section 3 describes component dependency metrics. Section 4 analyses software dependency and software co-evolution and formulates the speculations of their correlations. Section 5 outlines the verification experiment and describes research data and research method. Section 6 discusses the results of the quasi-experiment. Section 7 discusses threats to the validity of this study. Conclusions and future work are in Section 8.

2 Software component and component co-evolution

There are many definitions of a software component. In this paper, a software component is considered as a composition of software modules (classes, functions, source files and so on) [15–17]. A component could be a single module, such as a class, a function and a source file, or a software package that contains multiple modules, such as classes, functions and files. Fig. 1*a* shows a system that contains nine components (C1–C9), each of which is a composite that contains zero or more subcomponents and/or zero or more modules. For example, C1 contains two modules (m1 and m2) and one subcomponent SC1, which in turn includes two modules (m3 and m4). In component-based software development, each component is considered as a single unit. Therefore in

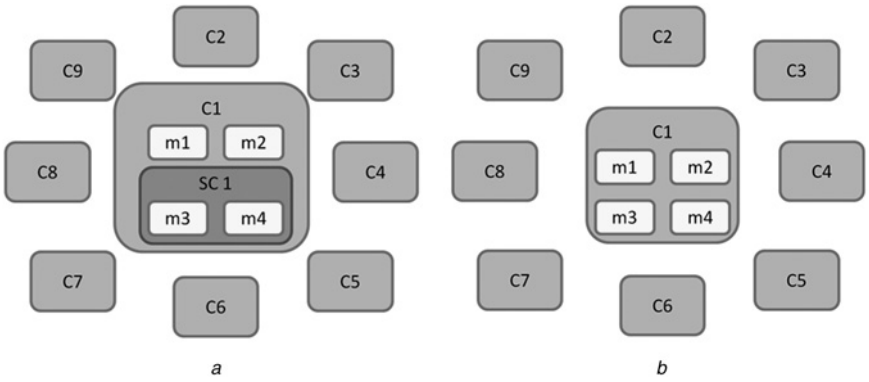

Figure 1 *System that contains nine components*

a Subcomponent representation
b Module-only representation

this paper, we ignore the subcomponent structure of a component and consider it as a composition of modules, which includes the modules directly located in the component and the modules indirectly located in its subcomponents. Accordingly, the structure of component C1 shown in Fig. 1*a* (subcomponent representation) is considered equivalent as the structure of component C1 shown in Fig. 1*b* (module-only representation). In this paper, module-only representation (Fig. 1*b*) is used to illustrate component and component co-evolution because of its simplicity.

Consider component C1 in Fig. 1*b*. Modules m1–m4 are called internal modules of component C1, and C2–C9 are called external components of C1 and modules in C2–C9 are called external modules of component C1.

Based on the previous description, we present the following definitions about a software component.

Definition 1: Component is a composition of software modules. It is represented as a set of modules, Ci = {m1, m2, . . . , mk}, in which C represents components and m represent modules.

Definition 2: Two components Ci and Cj are considered non-inclusive if Ci is not a subcomponent of Cj and Cj is not a subcomponent of Ci, that is Ci ∩ Cj = Ø.

In Fig. 1, components C1– C9 are non-inclusive, because the two components do not share any common modules. Next we have defined component co-evolutions.

Definition 3: For a system that contains n non-inclusive components (C1, C2, . . . , Ci, . . . , Cn), co-evolution happens if through a single revision, more than one module is changed. It is represented with a set of modules, Ep = {mj, ml, . . . , mk}, in which Ep represents co-evolution in revision p, $|Ep| > 1$, mj, ml, . . . , and mk are modules changed together in this revision.

A single revision is the smallest unit of revision activities that results the change of one or more components (modules) at one time. A single revision usually has one single goal, such as fixing a reported bug. However, multiple goals could be achieved in a single revision. Different software systems have different definitions of single revision. For example, in Linux, each new version release is considered as one revision, whereas in Apache HTTP, each maintenance activity is considered as one revision.

Definition 4: For a system that contains n non-inclusive components (C1, C2, Ci, . . . , Cn), and a co-evolution Ep = {mj, ml, . . . , mk}, if Ep is a subset of component Ci, that is $Ep \subset$ Ci, co-evolution Ep is called an internal co-evolution of component Ci.

Definition 5: For a system that contains n non-inclusive components (C1, C2, Ci, . . . , Cn), and a co-evolution Ep = {mj, ml, . . . , mk}, if Ep contains both internal modules of Ci and external modules of Ci, that is $Ep \cap$ C$i \neq \emptyset$ and $Ep \cup$ C$i \neq$ Ci, co-evolution Ep is called an external co-evolution of component Ci.

In other words, the internal co-evolution of component Ci only involves the evolution of modules within Ci; the external co-evolution of component Ci involves the evolution of modules both within Ci and outside of Ci. For an internal co-evolution, all the evolving modules are within a single component, it is easy to manage and test the changes. For an external co-evolution, changes are related with modules in different components, it requires more effort to identify the co-evolving modules, making the changes and testing the changes. Therefore from a software maintenance perspective, internal co-evolution can be considered easier to manage than external co-evolution.

Definition 6: For a system that contains n non-inclusive components (C1, C2, Ci, . . . , Cn), the internal co-evolution frequency of component Ci is the number of internal co-evolutions that happened to Ci in a specified period.

doi: 10.1049/iet-sen.2008.0084

Definition 7: For a system that contains n non-inclusive components (C1, C2, Ci, ..., Cn), the external co-evolution frequency of component Ci is the number of external co-evolutions that happened to Ci in a specified period.

Definition 8: For a system that contains n non-inclusive components (C1, C2, Ci, ..., Cn) and a co-evolution $Ep = \{mj, ml, \ldots, mk\}$, $Fi = Ci \cap Ep \neq \emptyset$ is a set containing the internal modules of Ci that belong to Ep, and is called co-evolved internal modules of component Ci in co-evolution Ep.

Definition 9: For a system that contains n non-inclusive components (C1, C2, Ci, ..., Cn) and a co-evolution $Ep = \{mj, ml, \ldots, mk\}$, $Fe = Ep - Ci$ is a set containing the external modules of Ci that belong to Ep, and is called co-evolved external modules of component Ci in co-evolution Ep.

Co-evolution frequency measures how frequently a component co-evolves with other components. The number of co-evolved internal modules (|Fi|) and the number of co-evolved external modules (|Fe|) measure the complexity of the co-evolution. For an internal co-evolution of a component, Fe (co-evolved external modules) is an empty set and |Fe| has a zero value.

For a system that contains n non-inclusive components (C1, C2, Ci, ..., Cn) and an internal co-evolution Ep of component Ci, the ratio of |Fi| to |Ci| is the percentage of internal modules of Ci participated in the internal co-evolution Ep. For a system that contains n non-inclusive components (C1; C2; Ci; ..., Cn) and an external co-evolution Ep of component Ci, the ratio of |Fi| to |Ci| is the percentage of internal modules of Ci participated in the external co-evolution Ep.

Definition 10: For a system that contains n non-inclusive components (C1; C2; Ci; ..., Cn) and an external co-evolution Ep of component Ci, the ratio of |Fe| to |Fi| is defined as the external co-evolution strength of component Ci in the external co-evolution Ep.

The percentage of internal modules that participated in an internal co-evolution represents the complexity of the internal co-evolution. The percentage of internal modules participated in an external co-evolution represents the complexity of an external co-evolution. External co-evolution strength represents the external interaction complexity of a component. A larger value of external co-evolution strength indicates the evolution of one internal module might require the co-evolution of many external modules, that is, an internal module interacts with more external modules.

3 Component dependencies

Both software modules and software components need to interact with each other for the system to perform a specific task. The interactions induce software dependencies, which can be divided into two types: dependency within a component, which is also called cohesion [18, 19] and dependency across components, which is also called coupling [3, 20]. Both cohesion [21, 22] and coupling [23, 24] have been widely used in measuring software design qualities. In this paper, component dependencies are measured using the following metrics of a software component [25]:

- *Component size* (N): The number of modules in a component.
- *Afferent couplings* (Ca): The number of external components that depend upon internal modules of a component.
- *Efferent couplings* (Ce): The number of external components that the internal modules in a component depend upon.
- *Component coupling* (Ca + Ce): The summation of afferent coupling and efferent coupling of a component.
- *Coupling density* (D): The ratio of component coupling to component size such that $D = (Ca + Ce)/N$.

Note that these metrics are with reference to a specific software component. A component (module) is considered dependent upon another component (module) if one module (component) utilises the classes, functions/methods, or variables defined in another module (component), that is, the correct operation of one component (module) depends on another component (module). Consider the dependency of component C1 in Fig. 2. An arrow pointing from an external component of C1 to an internal module of C1 indicates that the external component is dependent on the internal module. An arrow pointing from an internal module of C1 to an external component of C1

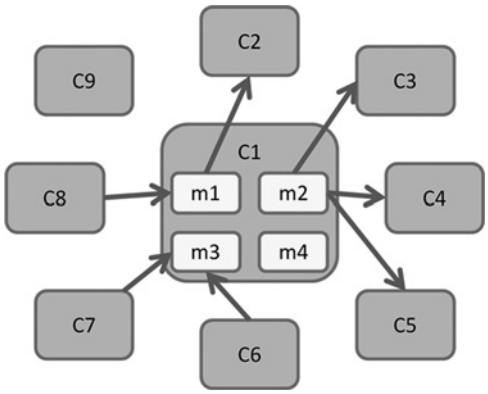

Figure 2 *Dependencies of component C1*

indicates the internal module is dependent on the external component. In Fig. 2, the dependency measures for component C1 are as follows: component size (N) is 4; afferent coupling (Ca) is 3; efferent coupling (Ce) is 4; component coupling (Ca + Ce) is 7; and coupling density (D) is 1.75.

4 Speculations of component dependency and component co-evolution

Component co-evolution is directly related with component dependencies. For example, if module m1 (or component C1) is logically dependent on module m2 (or component C2), changes made to m2 (or C2) might require corresponding changes on m1 (or C1).

4.1 Internal co-evolution

Consider an internal co-evolution of component C1 shown in Fig. 3. The frequency of the internal co-evolution and the number of co-evolved internal modules are related with the size component C1. If there are more modules within C1 and there are more interactions among these modules, internal co-evolutions could happen more frequently and there could be larger number of co-evolved internal modules in each co-evolution.

In this research, we do not directly measure the interactions within a software component. Instead, we measure component complexity–component size (number of modules contained in this component. We speculate that component internal co-evolution is correlated with component size. The speculations could be formulated as alternate hypotheses that hypothesise the existence of correlations or null hypotheses that hypothesise no correlations exist. As with conventions, for the purpose of statistical test, null hypotheses are used in this study. Accordingly, we present the following two null hypotheses:

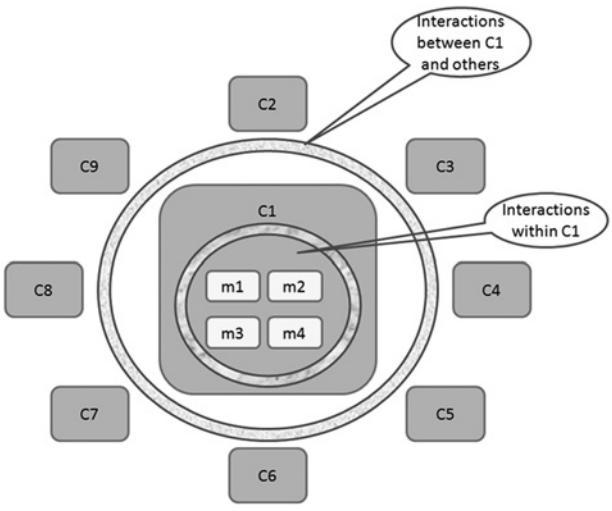

Figure 3 *Interactions within component C1 and between C1 and other components*

- H_{01}: The internal co-evolution frequency of a component is not correlated with the size of the component.

- H_{02}: The number of co-evolved internal modules in an internal co-evolution of a component is not correlated with the size of the component.

4.2 External co-evolution

Consider an external co-evolution of component C1 shown in Fig. 3. The co-evolution of C1 and other components (C2–C9) depends on two factors, the interactions within C1, and the interactions, between C1 and other components. (Note that the interactions between external components of C1 might also affect an external co-evolution of C1. However, these interactions are not directly observable through the target component C1. They are not included in this study.)

The external co-evolution frequency of component C1 is related with the size of C1. If there are more modules inside component C1, there are more chances that modules in C1 need to interact with external components and there will be more frequency of external co-evolution involving C1. Accordingly, we present the following null hypothesis.

- H_{03}: The external co-evolution frequency of a component is not correlated with the size of the component.

In Fig. 4, coupling between component C1 and other external components also determines the chance of an external co-evolution of C1. The coupling between C1 and other external components have two directions: C1 is dependent on Ci ($i \neq 1$), or Ci ($i \neq 1$) is dependent on C1, as shown in Fig. 4. If C1 is dependent on Ci ($i \neq 1$), changes made to Ci ($i \neq 1$) require corresponding changes to be made to C1; If Ci ($i \neq 1$) is dependent on C1, changes to be made to C1 require corresponding changes to be made to Ci ($i \neq 1$). More specifically, efferent couplings of C1 indicate the chance of external co-evolutions of C1 initiated by external components; afferent couplings of C1 indicate the chance of external co-evolutions of C1 initiated by internal modules of C1.

In the mathematical framework, we established in Section 2, we did not differentiate the cause–effect direction of a co-evolution. Therefore we ignore the directions of both component co-evolution and component coupling and present the following null hypothesis.

- H_{04}: The external co-evolution frequency of a component is not correlated with the coupling of the component.

doi: 10.1049/iet-sen.2008.0084

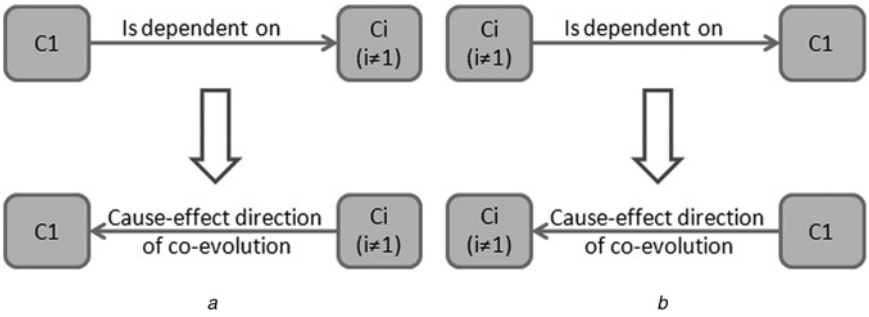

Figure 4 *Cause–effect direction of an external co-evolution of component C1 is related with coupling direction of component C1*

a Efferent coupling
b Afferent coupling

4.3 Coupling density and external co-evolution strength

Coupling density measures the average number of external components an internal module needs to interact with. The corresponding measure in component co-evolution is the external co-evolution strength, which indicates the average number of external modules co-evolved together with one internal module. These two measures are speculated to correlate. Consider component C1 and its internal module m1 in Fig. 5. The more interactions (larger coupling) between a single module m1 and external components, the more external modules inside these external components are likely to participate in the co-evolution with m1. We accordingly present the following null hypothesis.

- H_{05}: The external co-evolution strength of a component is not correlated with the coupling density of the component.

5 Research outlines

So far, we have discussed the speculated relations that might exist between component co-evolution and component dependency. Now, we will empirically examine and verify these relationships in this quasi-experiment (We call this study a quasi-experiment, because the data and processes to generate the data are not user controllable; they are mined and analysed as what they are.) as follows.

5.1 Research data

The verification experiment is performed on nine open-source projects from Apache Software Foundation [26]. They are Ant, Beehive, James, Lenya, Log4j, Mina, Struts, Tomcat and Xerces. The component dependency metrics were measured using open-source tool Jdepend [27]. The description of these nine projects and the specific version used to measure dependency metrics are listed in Table 1.

All the nine projects are written in Java, an object-oriented language. In the remaining of this paper, software modules are referred as classes. A Java source code file .java is considered as one class. A source code package that contains one or more java source code files is referred as a component.

The co-evolution history of each software component that appears in the specific version of these nine projects is obtained through mining their revision histories (CVS archives) [28]. Because the revisions of a single version could not provide enough data for measuring co-evolution, the revisions of all different versions of the same component are used to measure component co-evolution. A software tool written in Perl is used to extract co-evolution information from the CVS web site and the information is saved locally as data files. SPSS 15.0 for windows and Microsoft Excel 2007 are used to analyse data.

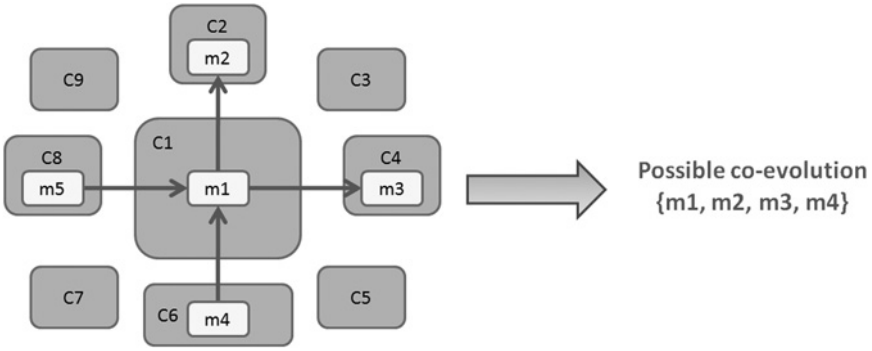

Figure 5 *High coupling density might lead to high external co-evolution strength*

Table 1 Nine Java projects and versions used to measure dependency metrics

Project name	Description	Version for dependency metrics
Ant	a Java-based build tool	1.5.3
Beehive	an application framework for J2EE development	1.0.2
James	Internet mail and news solutions	2.3.1
Lenya	a Java/XML content management system	1.2.5
Log4j	a Java-based logging utility	1.2.15
Mina	a network application framework	1.1.4
Struts	a framework for building Java web applications	2.0.11
Tomcat	an application server for Java Servelet and JSP	5.5
Xerces	an XML manipulation package	2.9.0

5.2 Research method

In Section 4, five null hypotheses were formulated. Each null hypothesis has a corresponding alternate hypothesis that speculates there exists correlation between two variables X (a dependency metric) and Y (a component co-evolution metric). To obtain a measure of how strongly X and Y values are related, we will need to calculate the correlation coefficient between X and Y: if X increases, does Y tend to increase or decrease? There are three commonly used correlation tests: Pearson, Spearman and Kendall tau. Pearson is a parametric test, and Kendall tau and Spearman are non-parametric tests. In more detail, Pearson tests the linear relationship between measured values of variables X and Y whereas Kendall tau and Spearman test the relations of ranked values of two variables X and Y. Therefore if the relationship between the variables X and Y is non-linear, Pearson will return a low coefficient, whereas Spearman and Kendall tau will show a high significance correlation [29].

The difference between Kendall tau test and Spearman test is that Spearman's test can be thought of as a regular Pearson test, except the correlation is computed from ranks instead of real observed values. Kendall tau test, on the other hand, represents the probability that two pairs of data, (X_i, X_j) and (Y_i, Y_j), are in the same order (increasing/decreasing) against the probability that they are in different orders (increasing/decreasing) [30].

Two values can be returned by running a correlation test, correlation coefficient (r) and correlation significance (p). The correlation coefficient, r, is a number between -1 and 1: a correlation of 1 is a perfect direct relationship between two variables X and Y; a correlation of -1 is a perfect inverse relationship between two variables X and Y; a correlation of 0 indicates no relationship between variables X and Y. The correlation significance, p, indicates the confidence of the correlation, that is, the probability that the observed correlation occurred by chance.

In this study, four levels of significance are recorded: the 0.001 level is for p that is less than or equal to 0.001; the 0.01 level is for p that is greater than 0.001 and is less than or equal to 0.01, the 0.05 level is for p that is greater than 0.01 and is less than or equal to 0.05 and the >0.05 level is for p that is greater than 0.05. A test result with significance at the 0.05 level or above (0.01 level, 0.001 level) is considered significant and we must reject the null hypothesis (accept the corresponding alternate hypothesis) and deduce that the correlation between variables X and Y is significant. A test result with significance at the >0.05 level is considered insignificant and we cannot reject the null hypothesis (must reject the corresponding alternate hypothesis) and deduce that the correlation between variables X and Y is not significant.

6 Quasi-experiment

6.1 General results

Tables 2 and 3 show the data of component size and component coupling of nine Java projects, respectively. The smallest component in each project contains one class; the largest component is found in Project Ant and it contains 130 classes. The smallest coupling is one and the components are found in six out of the nine projects; the largest coupling is 53 and is found in Project Ant. We

Table 2 Descriptive statistic: component size

Project name	Number of components	Number of classes			
		Min	Max	Mean	Std. deviation
Ant	46	1	130	12	20
Beehive	123	1	116	10	16
James	31	1	64	11	13
Lenya	47	1	58	10	12
Log4j	11	1	36	9	9
Mina	38	2	22	6	4
Struts	68	1	63	9	13
Tomcat	98	1	129	12	15
Xerces	33	1	85	21	18

doi: 10.1049/iet-sen.2008.0084

Table 3 Descriptive statistic: component coupling (Ca + Ce)

Project name	Number of components	Number of couplings			
		Min	Max	Mean	Std. deviation
Ant	46	1	53	10	9
Beehive	123	1	49	8	6
James	31	3	36	14	8
Lenya	47	1	32	8	6
Log4j	11	1	4	2	1
Mina	38	3	23	9	5
Struts	68	1	6	2	1
Tomcat	98	1	16	5	2
Xerces	33	4	40	16	10

remark here that these data are retrieved from one specific version of each project as shown in Table 1.

Table 4 listed the total number of revisions (until 21 September 2007) and the number of related revisions of each project. It is worth noting that not all the revisions are used for measure co-evolutions. Only the revisions that contain the components in these specific versions as shown in Table 1 are utilised. These revisions are called related revisions. Other revisions of old components that have been removed from these specific versions or new components that are introduced later than these specific versions are not utilised.

Tables 5 and 6 show the frequency of internal co-evolutions and external co-evolutions of Java components in these nine projects, respectively. The internal co-evolution frequency varies from component to component, even within the same project. For example in Project Tomcat, for some components, internal co-evolution never happened (internal co-evolution frequency is 0); for some components, internal co-evolution happened up to 4522 times. The external co-evolution frequency also varies from component to component. For example in Project Log4j, external co-evolution never happened to some components (external co-evolution frequency is 0), whereas some other components have up to 192 external co-evolutions.

Table 4 Number of revisions

Project name	Number of components	Total number of revisions	Number of related revisions
Ant	46	8343	6017
Beehive	123	661	422
James	31	2888	1998
Lenya	47	2047	271
Log4j	11	2181	2095
Mina	38	484	351
Struts	68	4358	774
Tomcat	98	13 387	9619
Xerces	33	4832	3259

Table 5 Descriptive statistic: frequency of internal co-evolutions

Project name	Number of components	Number of co-evolutions			
		Min	Max	Mean	Std. deviation
Ant	46	1	4111	172	638
Beehive	123	0	61	2	7
James	31	0	520	63	116
Lenya	47	0	85	5	13
Log4j	11	0	1988	223	586
Mina	38	0	116	8	21
Struts	68	0	765	21	94
Tomcat	98	0	4522	163	547
Xerces	33	0	1560	123	285

6.2 Internal co-evolution

Pearson, Spearman and Kendall tau tests are performed to test hypothesis H_{01}, that is, to study the correlations

Table 6 Descriptive statistic: frequency of external co-evolutions

Project name	Number of components	Number of co-evolutions			
		Min	Max	Mean	Std. deviation
Ant	46	17	947	113	165
Beehive	123	0	85	7	11
James	31	3	257	72	73
Lenya	47	0	78	10	17
Log4j	11	0	192	95	73
Mina	38	1	52	15	12
Struts	68	0	142	19	29
Tomcat	98	0	489	70	88
Xerces	33	2	291	59	78

Table 7 Correlations between component size and component internal co-evolution frequency

Project name	Pearson's correlation		Kendall's correlation		Spearman's correlation	
	Coefficient	Significance	Coefficient	Significance	Coefficient	Significance
Ant	0.919	0.001	0.522	0.001	0.687	0.001
Beehive	0.517	0.001	0.270	0.001	0.325	0.001
James	0.536	0.001	0.456	0.001	0.579	0.001
Lenya	0.270	**>0.05**	0.277	0.05	0.338	0.05
Log4j	0.919	0.001	0.623	0.01	0.732	0.01
Mina	0.538	0.001	0.294	0.05	0.385	0.05
Structs	0.130	**>0.05**	0.378	0.001	0.500	0.001
Tomcat	0.143	**>0.05**	0.449	0.001	0.593	0.001
Xerces	0.514	0.001	0.435	0.001	0.620	0.001

between component size and internal co-evolution frequency. The results are shown in Table 7. The significance values (p) that are greater than 0.05 are in bold letters. In all nine Java projects, Kendal tau test and Spearman test return significant correlations. Pearson test shows that six out of nine correlations are linear and three are non-linear. Fig. 6 shows the scatter plots of component size and internal evolution frequency of components in the nine projects. Although scatter plots could not directly tell the correlation coefficient and significance, they visualise the correlations. For example, in Fig. 6, Ant clearly has better linear correlation of component size and component internal co-evolution frequency than Lenya, Structs and Tomcat.

Pearson, Spearman and Kendall tau tests are performed to test hypothesis H_{02}, that is, to study the correlations between component size and the number of co-evolved internal classes. The results are shown in Table 8. The significance values (p) that are greater than 0.05 are in bold. In all nine Java projects, Kendal tau test and Spearman test return significant correlations. Pearson test shows that six out of nine correlations are linear and three are non-linear. Fig. 7 shows the scatter plots of component size and the number of co-evolved internal classes of components in nine projects. Similar to Fig. 6, in Fig. 7, Ant clearly has better linear correlation of component size and the number of co-evolved internal classes than Lenya, Structs and Tomcat.

Based on the above analysis, we reject the null hypotheses H_{01} and H_{02} and conclude that, for internal co-evolution, internal co-evolution frequency is correlated with component size (number of classes) and the number of

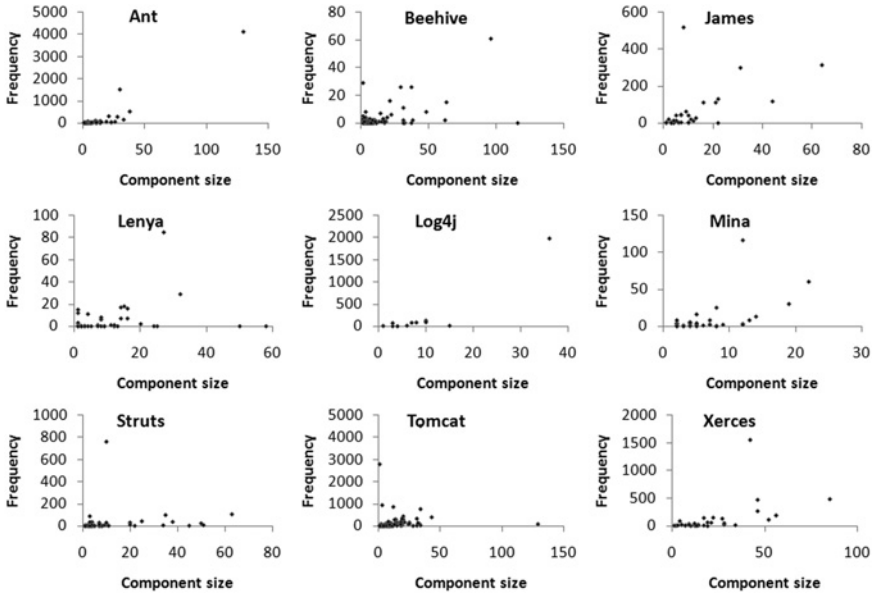

Figure 6 *Scatter plots of component size (number of classes) and component internal co-evolution frequency of components in nine Java projects*

IET Softw., 2010, Vol. 4, Iss. 4, pp. 252–267
doi: 10.1049/iet-sen.2008.0084

Table 8 Correlations between component size and the number of co-evolved internal classes

Project name	Pearson's correlation		Kendall's correlation		Spearman's correlation	
	Coefficient	Significance	Coefficient	Significance	Coefficient	Significance
Ant	0.915	0.001	0.616	0.001	0.772	0.001
Beehive	0.234	0.01	0.261	0.001	0.323	0.001
James	0.496	0.005	0.446	0.001	0.573	0.001
Lenya	0.267	**>0.05**	0.281	0.05	0.338	0.05
Log4j	0.915	0.001	0.523	0.05	0.675	0.05
Mina	0.575	0.001	0.341	0.01	0.440	0.01
Structs	0.041	**>0.05**	0.393	0.001	0.525	0.001
Tomcat	0.132	**>0.05**	0.484	0.001	0.634	0.001
Xerces	0.465	0.01	0.521	0.001	0.697	0.001

co-evolved internal modules (classes) are correlated with component size (number of classes).

6.3 External co-evolution

Pearson, Spearman and Kendall tau tests are performed to test hypothesis H_{03} and H_{04}, that is, to study the correlations between component size (H_{03})/component coupling (H_{04}) and external co-evolution frequency. The test results of hypothesis H_{03} are shown in Table 9. The significance values (p) that are greater than 0.05 are in bold letters. In eight out of the nine Java projects, Spearman test, Kendal tau test and Spearman test all return significant correlations. In one project (Log4j), none of Pearson test, Kendal tau test or Spearman test returns significant correlations. Fig. 8 shows the scatter plots of component size and external co-evolution frequency of components in nine projects, in which Ant has a representative linear correlations of component size and component external co-evolution frequency, and Log 4j is an exception and much different from others.

Table 10 contains the results of testing hypothesis H_{04}, that is, the correlations between component coupling (Ca + Ce) and external co-evolution frequency. The significance values (p) that are greater than 0.05 are in bold letters. In all nine Java projects, Kendal tau test and Spearman test return significant correlations. Pearson test shows that eight out of nine correlations are linear whereas one is non-linear. Fig. 9 shows the scatter plots of component coupling and external co-evolution frequency of components in nine projects. Again, in these plots, Ant has

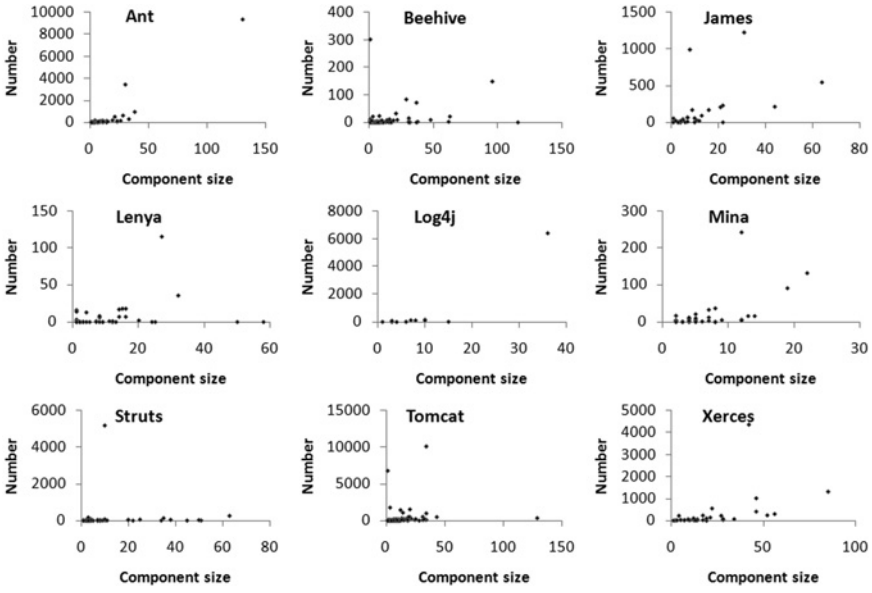

Figure 7 *Scatter plots of component size (number of classes) and the number of co-evolved internal classes in nine Java projects*

Table 9 Correlations between component size and component external co-evolution frequency

Project name	Pearson's correlation		Kendall's correlation		Spearman's correlation	
	Coefficient	Significance	Coefficient	Significance	Coefficient	Significance
Ant	0.894	0.001	0.524	0.001	0.691	0.001
Beehive	0.551	0.001	0.262	0.001	0.335	0.001
James	0.547	0.001	0.487	0.001	0.624	0.001
Lenya	0.313	0.05	0.286	0.05	0.349	0.05
Log4j	0.215	**>0.05**	0.374	**>0.05**	0.542	**>0.05**
Mina	0.658	0.001	0.371	0.01	0.491	0.01
Struts	0.611	0.001	0.396	0.001	0.520	0.001
Tomcat	0.243	0.05	0.423	0.001	0.561	0.001
Xerces	0.718	0.001	0.431	0.001	0.575	0.001

a representative linear correlations of component coupling and component external co-evolution frequency, and Log 4j is an exception and much different than others.

Based on the above analysis, we reject the null hypotheses H_{03} and H_{04} and conclude that, the external co-evolution frequency of a component is correlated with component size and component coupling.

6.4 Co-evolution complexity

Table 11 shows the average number of co-evolved classes in internal co-evolution and external co-evolution. It is significant to note that there are no external classes involved in an internal co-evolution. It can be seen, on an average, less than three classes are involved in each internal co-evolution; three to seven internal classes are involved in each external co-evolution; more external classes (29–243) are involved in each external co-evolution. Therefore external co-evolution is more complicated than internal co-evolution, not only because it involves external classes, also because more internal classes and external classes are involved.

As discussed in Section 2, the percentage of modules participated in an internal co-evolution represents the complexity of the internal co-evolution. Fig. 10 shows the boxplot of the percentage of classes participated in an internal co-evolution for each Java project. In the figure, the box contains the middle 50% of data; the line in the

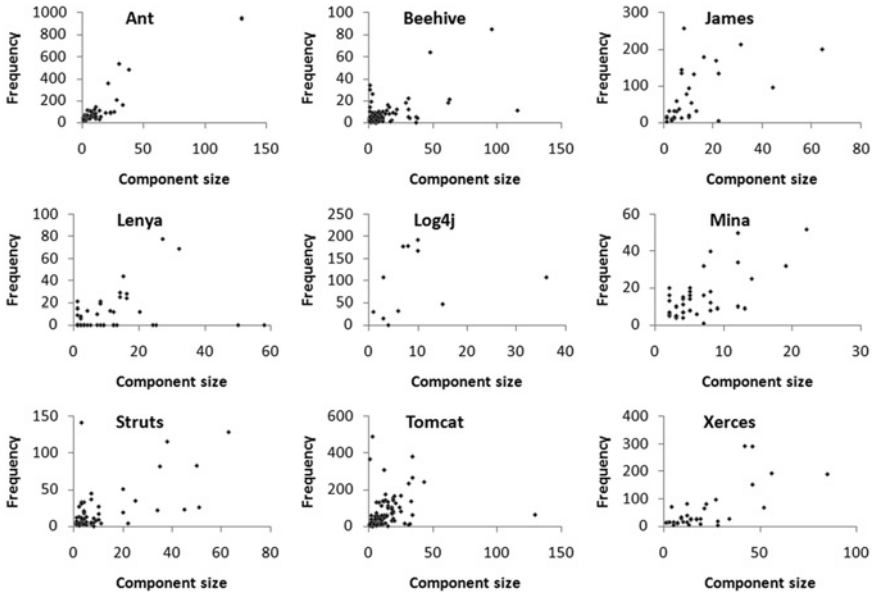

Figure 8 *Scatter plots of component size (number of classes) and component external co-evolution frequency in nine Java projects*

IET Softw., 2010, Vol. 4, Iss. 4, pp. 252–267
doi: 10.1049/iet-sen.2008.0084

Table 10 Correlations between component coupling (Ca + Ce) and component external co-evolution frequency

Project name	Pearson's correlation		Kendall's correlation		Spearman's correlation	
	Coefficient	Significance	Coefficient	Significance	Coefficient	Significance
Ant	0.909	0.001	0.576	0.001	0.730	0.001
Beehive	0.384	0.001	0.344	0.001	0.458	0.001
James	0.755	0.001	0.703	0.001	0.860	0.001
Lenya	0.331	0.05	0.292	0.01	0.344	0.05
Log4j	0.509	**>0.05**	0.583	0.05	0.733	0.01
Mina	0.781	0.001	0.521	0.001	0.665	0.001
Struts	0.333	0.01	0.319	0.001	0.402	0.001
Tomcat	0.294	0.01	0.329	0.001	0.442	0.001
Xerces	0.730	0.001	0.476	0.001	0.632	0.001

box indicates the median value of the data; the interquartile range (the upper edge and lower edge) indicates 75th and 25th percentile of the data set; circles are outliers that are data 1.5 times of interquartile range lower than the 25th percentile or 1.5 times of interquartile range higher than the 75th percentile.

Fig. 10 shows that the percentage of classes participating in internal co-evolution are different from project to project. On average, Project Lenya has the smallest percentage and Project Struts has the largest percentage. Some internal co-evolutions involve all (100%) of the internal classes. From the plot, we can deduce that the internal co-evolution of Project Lenya, which involves small percentage classes, is less complex than the internal co-evolution of Project Struts, which involves large percentage classes.

The percentage of classes participating in an external co-evolution represents the complexity of the external co-evolution. Fig. 11 shows the boxplot of the percentage of internal classes participated in an external co-evolution for each Java project. On average, Project Xerces has the smallest percentage and Project Struts has the largest percentage, which means the external co-evolution of Project Xerces is less complex than the external co-evolution of Project Struts.

It can also be seen from Figs. 10 and 11 that higher percentage of classes is involved in an external co-evolution than an internal co-evolution. The percentages of internal classes involved in an external co-evolution are different from project to project, in which, Project Xerces and Project Log4j have the smallest average value and Project Struts has the largest average value.

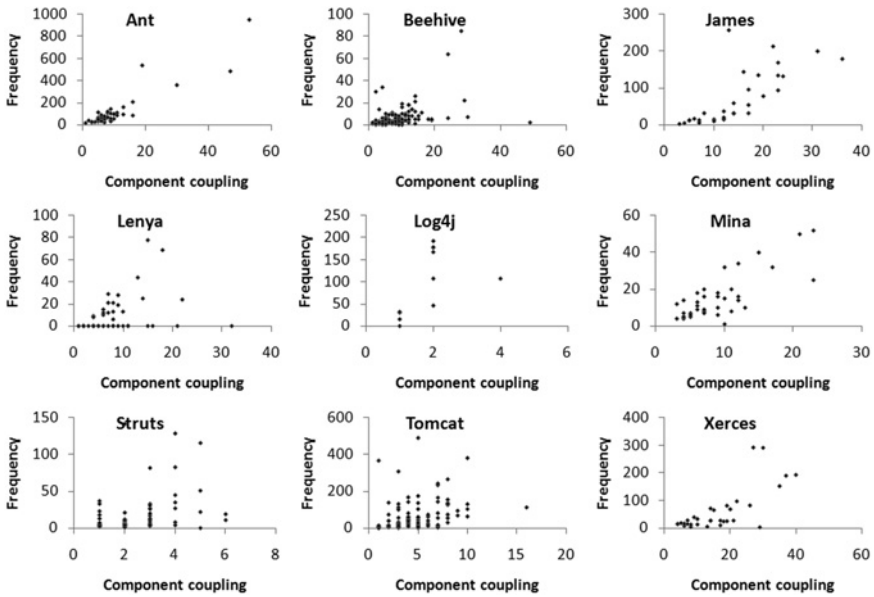

Figure 9 *Scatter plots of component coupling (Ca + Ce) and component external co-evolution frequency in nine Java projects*

Table 11 Average number of co-evolved internal classes and external classes in each co-evolution

Project name	Internal co-evolution	External co-evolution	
	Internal classes	Internal classes	External classes
Ant	2.11	4.66	243.00
Beehive	2.05	3.85	571.27
James	2.34	4.19	113.47
Lenya	1.06	5.25	174.41
Log4j	1.56	4.65	32.67
Mina	2.02	4.58	169.42
Struts	2.07	6.62	206.92
Tomcat	1.77	4.39	168.58
Xerces	2.45	3.63	29.21

Pearson, Kendall tau and Spearman test are used to test the correlation between coupling density and evolution strength (hypothesis H_{05}). Table 12 contains the result. It can be observed that seven out of the nine projects show significant correlations between coupling density and evolution strength (through Kendall tau test and Spearman test), four of which show linear correlations between coupling density and evolution strength (through Pearson test). Fig. 12 shows the scatter plot of coupling density and evolution strength of components in nine projects. In these plots, Xerces has a representative linear correlation of component coupling density and component evolution strength, and Log 4j and Mina are representatives of non-linear correlations of component coupling density and component evolution strength.

Based on the above analysis, we reject the null hypotheses H_{05} and conclude that the evolution strength of a software component is correlated with the coupling density of this component.

7 Threats to validity of the research

As with any other empirical research, there are certain threats to the validity of this experiment. The internal threat to validity is the significance of the results. Our quasi-experiment shows that most correlations are significant at the 0.01 level or above, but some correlations are at the 0.05 level, which is considered insignificant. To reduce this threat, more data should be gathered and additional studies performed to further verify these speculations. The external threat to validity is that this quasi-experiment is performed on open-source Java projects. The results might not be applicable to closed-source projects or projects written in different languages.

One construct threat to validity is that in this study, we use package to represent component and use class to represent module. These representations could be accurate for object-oriented software systems. However, the results might not be applicable to traditional structured software system. To reduce this threat, similar studies should be performed on traditional structured software system. Another construct threat comes from the selection of the version of different projects for dependency measurement. In this study, a specific version of each project is used for dependency measurement, whereas co-evolution analysis is performed on all versions of that project. As described in Section 5, the revisions of a single version could not provide enough data for co-evolution analysis. Therefore we decided to choose all revisions of the same component to measure its co-evolution. Because one component is

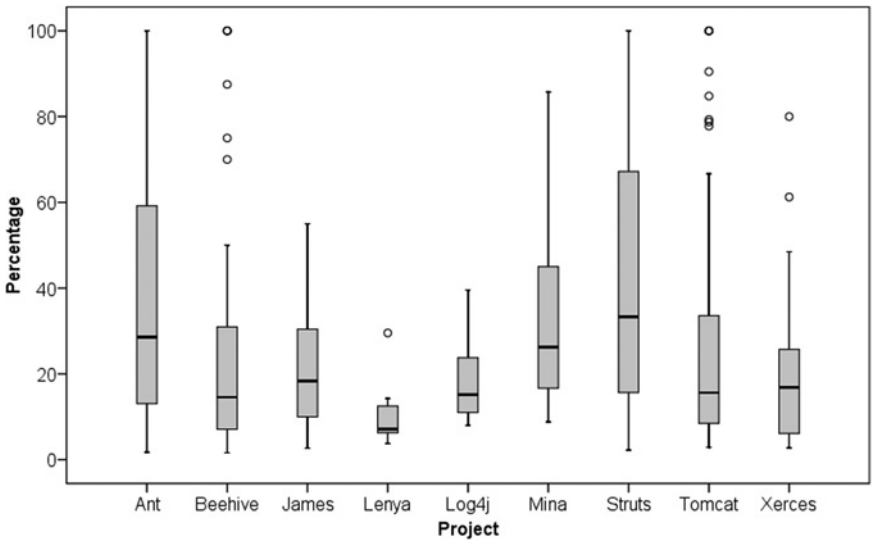

Figure 10 *Percentage of classes participated in an internal co-evolution of components in nine projects*

doi: 10.1049/iet-sen.2008.0084

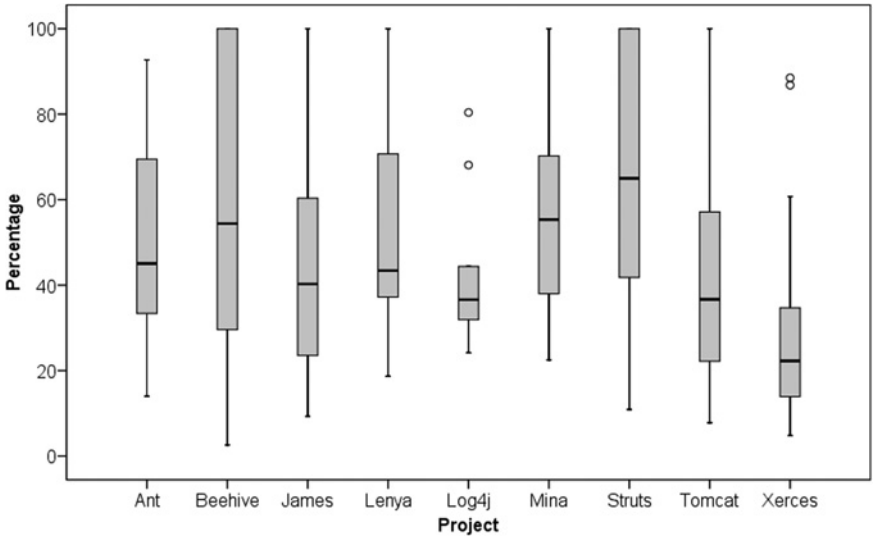

Figure 11 *Percentage of internal classes involved in an external co-evolution of components in nine projects*

Table 12 Correlations between component coupling density and component evolution strength

Project name	Pearson's correlation		Kendall's correlation		Spearman's correlation	
	Coefficient	Significance	Coefficient	Significance	Coefficient	Significance
Ant	0.419	0.01	0.421	0.001	0.601	0.001
Beehive	0.092	**>0.05**	0.208	0.01	0.292	0.01
James	0.260	**>0.05**	0.312	0.05	0.469	0.01
Lenya	0.447	**>0.05**	0.512	0.01	0.702	0.001
Log4j	0.322	**>0.05**	0.270	**>0.05**	0.117	**>0.05**
Mina	0.117	**>0.05**	0.009	**>0.05**	0.012	**>0.05**
Struts	0.305	0.05	0.291	0.001	0.421	0.001
Tomcat	0.353	0.001	0.285	0.001	0.407	0.001
Xerces	0.546	0.001	0.325	0.01	0.426	0.05

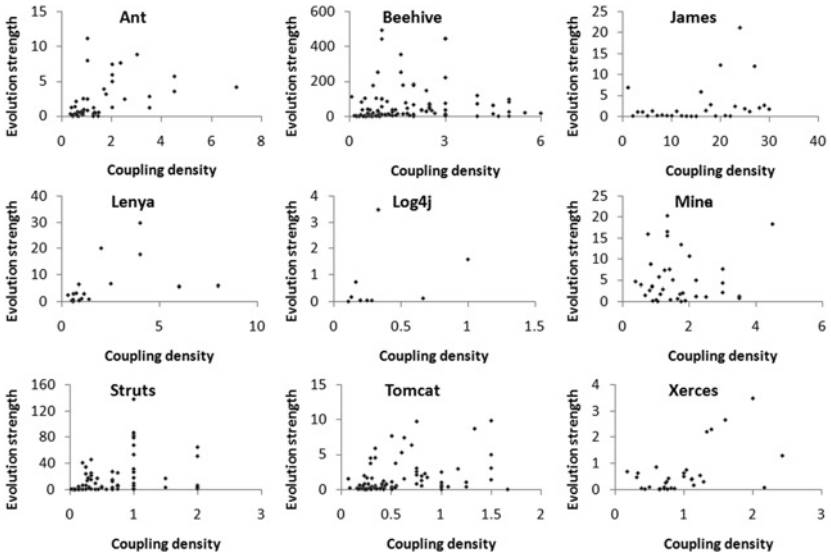

Figure 12 *Scatter plot of component coupling density and component evolution strength in nine projects*

under continuous revision, its dependency metrics might change from version to version. Therefore using a single version to measure dependency metrics might be not accurate. To reduce this threat, more versions of the same component should be used for dependency measurement.

8 Conclusions and future research

In this paper, we have presented a mathematical framework to represent component co-evolution. We have analysed and speculated on the correlations between component dependency and component co-evolution. Verification studies were performed on Java components of nine open-source projects, in which we measured the component dependency and mined component co-evolution history. The speculations of the relations between component dependency and component co-evolution were validated through testing the correlations of dependency measures and co-evolution measures.

Component co-evolution is an important issue in software maintenance. Understanding its relationship with code properties, such as coupling and cohesion, can help both researchers and practitioners design high maintainable and evolvable software systems. Our future work in this direction will focus on the design of new coordination mechanisms between software components in order to reduce the side effect of co-evolution. More specific future work is listed below:

1. *Apply cohesion metrics:* The metrics suite used in this research does not include a metric for measuring cohesions of a component. In our future work, we will utilise other metrics suite such as Chidamber and Kemerer's CBO [24] metric to measure component cohesion and study its relationship with component co-evolution.

2. *Study aspect-oriented system:* Aspect-oriented technologies [31] are aimed at providing mechanisms to reduce the coupling between components. One such research is to compare aspect-oriented program with object-oriented program and investigate whether the feature metrics of objected-oriented program, crosscutting concerns, have lower impact on component co-evolution. This is some expectation that it is assumed in aspect-oriented program but has not been appropriately evaluated throughout practical experiments.

9 References

[1] STEVENS W.P., MYERS G.Z., CONSTANTINE L.L.: 'Structured design', *IBM Syst. J.*, 1974, **13**, (2), pp. 115–139

[2] OFFUTT J., HARROLD M.J., KOLTE P.: 'A software metric system for module coupling', *J. Syst. Softw.*, 1993, **20**, (3), pp. 295–308

[3] PRESSMAN R.S.: 'Software engineering: a practitioner's approach' (McGraw-Hill, 2005)

[4] YU L.: 'Understanding component co-evolution with a study on Linux', *Empir. Softw. Eng.*, 2007, **12**, (2), pp. 123–141

[5] ZIMMERMANN T., DIEHL S., ZELLER A.: 'How history justifies system architecture (or not)'. Proc. 6th Int. Workshop on Principles of Software Evolution, Helsinki, Finland, September 2003, pp. 73–83

[6] ZIMMERMANN T., WEISSGERBER P., DIEHL S., ZELLERS A.: 'Mining version histories to guide software changes'. Proc. 26th Int. Conf. on Software Engineering, Scotland, UK, May 2004, pp. 563–572

[7] ZIMMERMANN T., WEISSGERBER P., DIEHL S., ZELLERS A.: 'Mining version histories to guide software changes', *IEEE Trans. Softw. Eng.*, 2005, **31**, (6), pp. 429–445

[8] YING A.T.T., NG R., CHU-CARROLL M.C., MURPHY G.C.: 'Predicting source code changes by mining change history', *IEEE Trans. Softw. Eng.*, 2004, **30**, (9), pp. 574–586

[9] HASSAN A.E., HOLT R.C.: 'Predicting change propagation in software systems'. Proc. 20th Int. Conf. on Software Maintenance, Chicago Illinois, USA, September 2004, pp. 284–293

[10] GRAVES T.L., KARR A.F., MARRON J.S., SIY H.: 'Predicting fault incidence using software change history', *IEEE Trans. Softw. Eng.*, 2000, **26**, (7), pp. 653–661

[11] WILLIAMS C.C., HOLLINGSWORTH J.K.: 'Automatic mining of source code repositories to improve bug finding techniques', *IEEE Trans. Softw. Eng.*, 2005, **31**, (6), pp. 466–480

[12] DE LEMOS R.: 'Analysing failure behaviours in component interaction', *J. Syst. Softw.*, 2004, **71**, (1–2), pp. 97–115

[13] EKANAYAKE J., TAPPOLET J., GALL H.C., BERNSTEIN A.: 'Tracking concept drift of software projects using defect prediction quality'. Sixth IEEE Int. Working Conf. on Mining Software Repositories, Vancouver, BC, Canada, May 2009, pp. 51–60

[14] YU L., SCHACH S.R.: 'Applying association mining to change propagation', *Int. J. Softw. Eng. Knowl. Eng.*, 2008, **18**, (8), pp. 1043–1061

[15] MEI H., ZHANG L., YANG F.: 'A software configuration management model for supporting component-based software development', *ACM SIGSOFT*, 2001, **26**, (2), pp. 53–58

[16] DE JONGE M.: 'Package-based software development'. Proc. 29th Euromicro Conf., Antalya, Turkey, September 2003, pp. 76–85

doi: 10.1049/iet-sen.2008.0084

[17] DE JONGE M.: 'Multi-level component composition'. Proc. Second Groningen Workshop on Software Variability Modeling, Groningen, Netherland, December 2004

[18] BIEMAN J.M., KANG B.: 'Measuring design-level cohesion', *IEEE Trans. Softw. Eng.*, 1998, **24**, (2), pp. 111–124

[19] BIEMAN J.M., OTT L.M.: 'Measuring functional cohesion', *IEEE Trans. Softw. Eng.*, 1994, **20**, (8), pp. 644–657

[20] SCHACH S.R.: 'Object-oriented and classical software engineering' (McGraw-Hill, New York, 2007, 7th edn.)

[21] GUI G., SCOTT P.D.: 'Coupling and cohesion measures for evaluation of component reusability'. Proc. Third Int. Workshop on Mining Software Repositories, Shanghai, China, May 2006, pp. 18–21

[22] GUI G., SCOTT P.D.: 'Ranking reusability of software components using coupling metrics', *J. Syst. Softw.*, 2007, **80**, (9), pp. 1450–1459

[23] BRIAND L.C., DALY J.W., WÜST J.K.: 'A unified framework for coupling measurement in object-oriented systems', *IEEE Trans. Softw. Eng.*, 1999, **25**, (1), pp. 91–121

[24] CHIDAMBER S., KEMERER C.: 'A metric suite for object oriented design', *IEEE Trans. Softw. Eng.*, 1994, **30**, (6), pp. 476–493

[25] MARTIN R.: 'OO design quality metrics: an analysis of dependencies', 1994, available at http://www.objectmentor.com/resources/articles/oodmetrc.pdf

[26] Apache Project: http://www.apache.org/

[27] Jdepend: http://clarkware.com/software/JDepend.html

[28] Apache CVS: http://svn.apache.org/viewvc/?limit_changes=0

[29] HILL T., LEWICHI P.: 'Statistics: methods and applications: a comprehensive reference for science' (StatSoft, Inc, 2006)

[30] ABDI H.: 'Kendall rank correlation' in SALKIND N.J. (ED.): 'Encyclopedia of measurement and statistics', (Thousand Oaks, 2007)

[31] FUENTES L., SÁNCHEZ P.: 'Aspect-oriented coordination', *Electron. Notes Theor. Comput. Sci.*, 2007, **189**, pp. 87–103