JTracker – A Tool for Change Propagation in Java

Steve Gwizdala, Yong Jiang, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202
{sg, jiang, rajlich}@cs.wayne.edu

Abstract

During software evolution, programmers add new functionalities and release new versions of software. This is complicated work, particularly in large applications, and tools are needed to deal with it. In this paper we introduce a tool named JTracker that helps programmers implement change propagation in Java applications. We conducted a case study of a change in open source application JMeter, in which we used JTracker.

1. Introduction

One of the key stages of software lifecycle is software evolution [9]. During software evolution, programmers modify the code of the applications to satisfy user requests and customer feedbacks. The most common kind of modification is incremental change that adds new software functionality.

Incremental change was studied in several earlier papers [3], [8], [10]. The incremental change minicycle consists of the following steps:

- Change request
- Concept location
- Concept actualization
- Incorporation
- Change propagation

A change request is a summary of the desired change, usually expressed in plain English. The programmer extracts the most important concepts of the change request, whether they are domain concepts, programming concepts, or concepts related to the program operation.

During the concept location, the programmer locates in the code the place where the old version of the concepts is implemented. Then, during concept actualization, the programmer implements the new version of the concepts. During incorporation the programmer replaces the old version of the concepts by the new one.

This replacement may break dependencies within the old code, making them inconsistent. In order to remove these inconsistencies, the programmer visits components and their dependencies and makes secondary changes wherever necessary. When a class or component is modified, the relationship between it and its neighboring classes may become inconsistent. To fix this problem, the programmer should check neighboring classes and decide whether or not they need to be modified. The changes in them may trigger additional changes in further classes, which in turn may trigger more changes, and so forth. This process is named change propagation.

Change propagation is difficult, and an error-prone part of incremental change. In order to support it, we developed a tool *JTracker* that detects the classes where secondary changes may need to be made. We validated *JTracker* in a case study of a change in open software called *JMeter*.

Section 2 introduces *JTracker*. Section 3 presents the case study. Section 4 contains relation to other work. Conclusion and future work are in section 5.

2. JTracker

JTracker is an extension to Java environment *JBuilder5* [17] and assists a programmer during change propagation. Whenever programmer changes a class, the relationship between the class and its neighbors may have become inconsistent. The programmer should visit all neighboring classes and see whether they have been affected. *JTracker* marks these neighbors and informs the programmer about them. If changes are not

Proceedings of the Seventh European Conference On Software Maintenance And Reengineering (CSMR'03)
0-7695-1902-4/03 $17.00 © 2003 IEEE
IEEE COMPUTER SOCIETY

necessary, the propagation can be stopped and marks removed. However if changes are needed, the programmer makes modifications using a standard editor. After these modifications, *JTracker* automatically marks additional classes. The change propagation continues until no marked classes exist in the program.

JTracker scans the source code and creates a database of classes and their dependencies. The dependencies are defined broadly; if class A uses class B in any way, either as a data member, local variable, or argument, or if A inherits from B, then A is dependent on B. Classes A and B are neighbors if either A uses B or B uses A. Every time a change is made to a class in the project, the dependency database is updated, and all neighboring classes are marked.

JTracker also supports concept location. *JTracker* searches its database for the classes, which have a method, named "main". One of these classes is the beginning of the search. *JTracker* will mark other classes whose methods have been called in this root class. Several classes may be marked and they all are candidates during the search. The programmers have to comprehend them and make a decision as to which path to take. If class A has been chosen, *JTracker* will find the classes used by A. This process continues until programmers find the accurate location of the concept. If programmers choose a wrong direction, *JTracker* helps them to backtrack to a previous position and repeat the search decision again.

Figure 3 contains the interface of *JTracker* where the programmer views both the dependencies and marks. The interface contains dependency display table, where each row represents one local variable with its class membership, type, and so on. The column "To Modify" represents the mark. The check in that column means that the corresponding class is to be visited, understood, and possibly modified.

JTracker does not make the decision when to finish the search or change propagation. It is the responsibility of the programmer to remove the marks from the classes that do not need change and do not propagate change further.

JTracker was written in Java as an extension of Borland *JBuilder*5 environment. Our decision to use *JBuilder*5 was influenced by the fact that *JBuilder*5 has an API that allows an easy addition of custom tools. The main classes of *JTracker* are listed in Table 1.

As *JBuilder* parses the Java code, it stores the resulting data in its Java Object Toolkit (JOT) classes. The *JTracker* uses its *ParsedCode* class to access the appropriate JOT classes and extract information on variables and their types. When a variable is found, *JTracker* stores its name, class name, type, field type, and method name.

JTracker	the main class that loads the tool into the *Jbuilder* environment
ParsedCode	accesses the *JBuilder* internal data structure to fetch dependency information
DisplayDependencies	displays the dependency database in table format
Dependency	stores dependencies for the whole program
DependencyList	stores dependencies of a single class

Table 1. The main classes of *JTracker*

To store the dependency data we created two classes: *Dependency*, and *DependencyList*. *DependencyList* stores data for a single class. It is used in the *Dependency* class as an element stored in a vector.

3. Case Study

To examine the performance of *JTracker*, we conducted a case study of an incremental change in *JMeter* [18].

The Apache *JMeter* is a pure Java application that has more than 400 classes. *JMeter* was originally designed to test the performance of web sites. It tests functional behavior and measures performance of Java applications. It simulates a heavy load on a server or a network and analyzes overall performance under different load types. It also tests server/script/object behavior under concurrent load and displays the results graphically. It may be used to test performance both on static and dynamic resources like files, Servlets, Perl scripts, Java objects, Databases and queries, and FTP Servers.

3.1 Change Request

JMeter version 1.7 does not provide notification that the testing is completed. The users do not know when to check the results because they are not sure whether the testing has finished. Without any executing information, users either wait a long time, or stop a running process and get no answer about results. In the latter case, they have to redo the testing. This is a problem that authors of *JMeter* listed among the bugs to be fixed.

IEEE COMPUTER SOCIETY

The change request is to implement "process control" that will resolve this problem. When a user sends the request to test a web site, the process control will monitor dynamical execution and inform the user whether the process is still running or has finished. A pop up window will indicate status of the testing.

During the change, we followed the methodology explained in [2], [3], [8], [10] and used the tool *JTracker* for change propagation.

3.2 Concept Location

The first step was to locate the concept "process control". We used the methodology of search through the static code of [2], [8]. The search progressed top-down through control flow dependencies. Figure 1 shows the UML class diagram of selected classes that play an important role in our case study.

At first, we needed to know where the *JMeter* interface is implemented because the new functionality will add new control buttons to *JMeter* menu bar. With the help of *JTracker* we found the method "main" in a class named *NewDriver*. The concept location began from this class because the functionality of the whole program is summarized there. *JTracker* informed us about the classes used in class *NewDriver*. After viewing the code of these classes, we concluded that class *MainFrame* is responsible for the interface of *JMeter*.

Class *MainFrame* is responsible for the whole *JMeter* interface and objects of class *JMeterMenuBar* implement individual buttons of the interface. This is where the interface part of the new concept was located.

Then we searched for the location where the "process control" executes. When users want to execute *JMeter*, they click a "start" button in the *JMeter* menu bar. This action triggers an event in *JMeterMenuBar*. Class *Start* starts various processes in *JMeter*, but it does not contain detailed functionality. It sends commands to run and stop a process. An object of type *StandardJMeterEngine* is declared in class *Start*. It gets process information from class *JMeterThread*. We found that *JMeterThread* executes the processes, so the concept is located in class *JMeterThread*.

3.3 Actualization

In this phase, we implemented a new class named *TestEnd* that traces process execution.

The threads that test web performance run concurrently and they may finish in an arbitrary order. When a thread starts to run, it calls *run()* method in class *JMeterThread*. Each thread runs separately and does not know whether other threads are still running or have finished. We declared a variable named *threadflag* in the class *TestEnd* with initial value equal to 0. The member function *addflag()* increases the value of *threadflag* by 1. The other member function *checkflag()* compares the value of *threadflag* with the total number of threads that are finished. If they are equal, *TestEnd* knows that the testing process is finished and shows a *test end message* to the user.

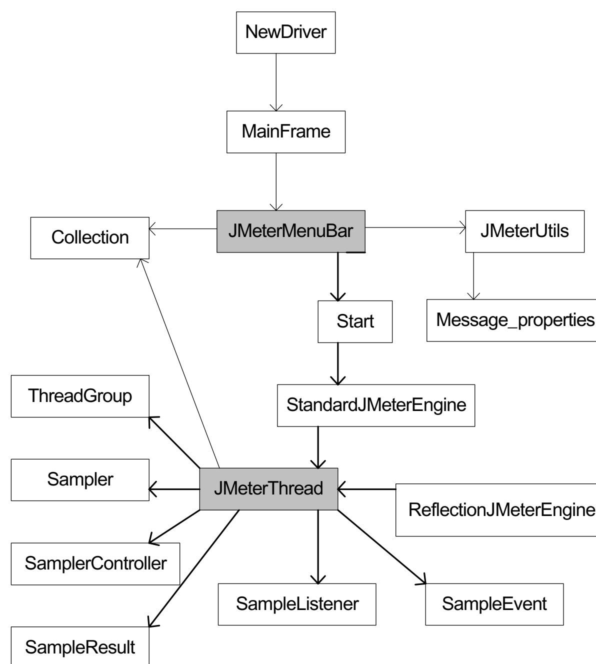

Figure 1. Relevant classes of *JMeter*

3.4 Incorporation

Incorporation connects new code with the old, in our case by creation of instances of the new class *TestEnd* in classes *JMeterThread* and *JMeterMenuBar.* In class *JMeterThread*, we modified methods *run(), stop()* and *rampUpDelay().* Method *run()* calls *TestEnd*'s methods *addFlag()* and *checkFlag()* and it also calls *showMessage(String)* to inform the user that the testing ended. In class *JMeterMenuBar*, method *reset()* obtains a completed message from class *TestEnd* and calls other member functions in *JMeterMenuBar* to reset the status of buttons of run menu bar in *JMeter* interface. Figure 2 shows the new class *TestEnd* and modified classes *JMeterThread* and *JmeterMenuBar*.

IEEE COMPUTER SOCIETY

3.5 Change Propagation

After modifying the classes *JMeterThread* and *JMeterMenuBar*, the source code was no longer consistent and we needed to fix the code inconsistencies. We used the tool *JTracker* to find the dependencies of modified classes and propagate the change.

JTracker automatically detects modifications that have been made in the code and marks neighbors of the modified classes. After we modified the classes *JMeterThread* and *JMeterMenuBar, JTracker* marked their neighbors *ReflectionJMeterEngine, StandardJMeterEngine, ThreadGroup, Sampler, SamplerController, SampleResult, SampleListener, SampleEvent, MainFrame, Collection, Start* and *JMeterUtil.*

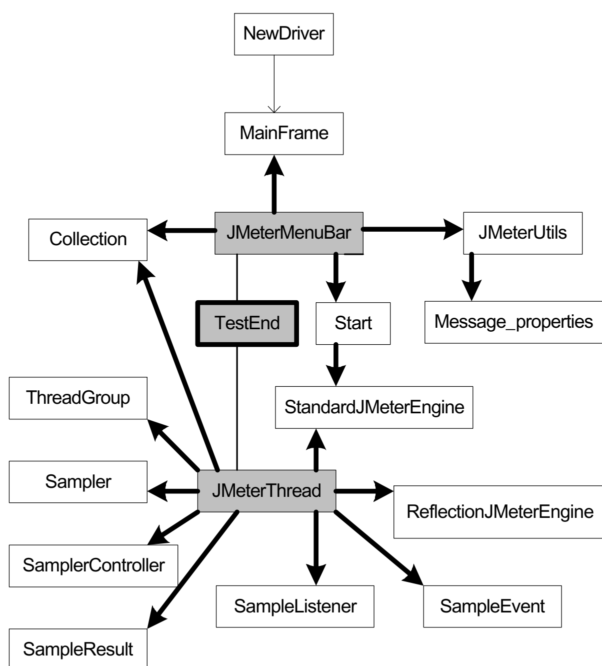

Figure 2. UML class diagram of affected classes

Figure 3 shows part of *Jtracker*'s dependency display table with the marked classes *ReflectionJMeterEngine* and *StandardJMeterEngine*. Please note that the entire table could not be shown because of size. Also note that statuses of marked classes in Figure 3 are "Unvisited". That means these classes have been marked but not checked. Checking these classes is in the next step.

In the next step, we separately visited each of these classes. For example, we visited class *ReflectionJMeterEngine*. This class did not need any modification. Therefore we cleared its mark. Figure 4 shows that *JTracker* changed the status of the class to "Visited" to inform the programmer this class has been checked. Then we visited other marked classes. If they needed to be modified and were changed, their status was changed to "Visited" too and their neighboring classes were marked. The complete change propagation is summarized in Figure 2 where the fat arrows represent all marks that *JTracker* provided at some point in the process.

3.6 Conclusions of the case study

In this case study, we used *JTracker* to keep track of change propagation and inspected classes that needed to be checked. We checked all neighbors of modified classes to make sure we solved the inconsistencies caused by changes. Relevant data in this case study are shown in Table 2.

Classes added	1
Classes visited	15
Classes modified	5
LOC modified	53

Table 2. Numerical data of case study

JTracker proved to be a useful tool in change propagation. It collected dependency information we needed to propagate the changes. When a modification occurred, *JTracker* informed us of the classes that may be affected by the modification. We followed this information to do additional checking of these classes. We propagated changes in the source code until no class was marked.

JTracker is able to find all neighboring classes that may be affected by a changed class and frees programmers from this tedious work. At the same time, it allows programmers to make important decisions. Hence it contributes to the quality and productivity of programmer's work.

IEEE COMPUTER SOCIETY

Dependency Display

Close | Refresh | Clear

Class Name	Variable Name	Variable Type	Field Type	Method Name	To Modify	Status	Date
ClientJMeterEngine	rsl	RemoteSampleLi...	method field	makeListenersRe...	☐	Unvisited	
IncrementalJMeterEngine	catClass	Category	class field	NA	☐	Unvisited	
IncrementalJMeterEngine	threads	JMeterThread[]	method field	runTest	☐	Unvisited	
IncrementalJMeterEngine	threadList	Thread[]	method field	runTest	☐	Unvisited	
IncrementalJMeterEngine	iter	Iterator	method field	runTest	☐	Unvisited	
IncrementalJMeterEngine	group	ThreadGroup	method field	runTest	☐	Unvisited	
JMeterEngine	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
ReflectionJMeterEngine	catClass	Category	class field	NA	☐	Unvisited	
ReflectionJMeterEngine	jMeterThread	JMeterThread	method field	runTest	☑	Unvisited	
ReflectionJMeterEngine	iter	Iterator	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	thisThreadGroup	ThreadGroup	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	control	GenericController	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	subControllerList	List	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	configElementList	List	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	subControllersIter...	Iterator	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	samplerController	SamplerController	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	lc	LoopController	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	newThread	Thread	method field	runTest	☐	Unvisited	
RemoteJMeterEngine	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
RemoteJMeterEngineImpl	backingEngine	JMeterEngine	class field	NA	☐	Unvisited	
RemoteJMeterEngineImpl	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
RemoteJMeterEngineImpl	engine	RemoteJMeterEng...	method field	main	☐	Unvisited	
StandardJMeterEngine	threadGroups	Collection	class field	NA	☐	Unvisited	
StandardJMeterEngine	allThreads	Collection	class field	NA	☐	Unvisited	
StandardJMeterEngine	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
StandardJMeterEngine	threads	JMeterThread[]	method field	runTest	☐	Unvisited	
StandardJMeterEngine	iter	Iterator	method field	runTest	☐	Unvisited	
StandardJMeterEngine	group	ThreadGroup	method field	runTest	☐	Unvisited	
StandardJMeterEngine	newThread	Thread	method field	runTest	☐	Unvisited	
StandardJMeterEngine	iter	Iterator	method field	stopTest	☐	Unvisited	
StandardJMeterEngine	item	JMeterThread	method field	stopTest	☑	Unvisited	
ButtonPanel	add	JButton	class field	NA	☐	Unvisited	
ButtonPanel	delete	JButton	class field	NA	☐	Unvisited	
ButtonPanel	edit	JButton	class field	NA	☐	Unvisited	
ButtonPanel	load	JButton	class field	NA	☐	Unvisited	
ButtonPanel	save	JButton	class field	NA	☐	Unvisited	
ButtonPanel	d	Dimension	method field	init	☐	Unvisited	
ButtonPanel	g	GridBagLayout	method field	init	☐	Unvisited	
ButtonPanel	c	GridBagConstraints	method field	init	☐	Unvisited	
ButtonPanel	listener	ActionListener	formal parameter	addButtonListener	☐	Unvisited	
FileDialoger	promptToOpenFile	JFileChooser	method return	promptToOpenFile	☐	Unvisited	

Start | Jtracker10[1].08.doc - Mi... | JBuilder - D:/JTracker1/u... | Dependency Display | 1:13 AM

Figure 3. *JTracker* window

4. Relation to Other Work

Biggerstaff et al. [1] defined and investigated concept location, which he called “concept assignment”. He indicated that concepts and program are not at the same level of abstraction and human involvement is necessary. He also recommended computer assistance and appropriate tools that enhance human performance.

Wilde et al. [13], [14] developed a concept location technology called Software Reconnaissance. The technology is dynamic and is based on the analysis of test cases.

Chen and Rajlich [3] used dependence graph for change in C programs. In [2], they presented concept location methodology we used in our case study.

JTracker belongs to the category of tools called software browsers [11] that extract and represent program dependencies.

Refactoring is a kind of change that modifies the software’s structure and makes it more suitable for future evolution [6]. Tokuda and Batory [12] described three modes of evolution for Object-Oriented software architectures. They are schema transformations, design pattern microarchitectures and the hot-spot-driven-approach. They claim that refactoring can support many evolutionary changes through reduction of unnecessary complexity and inefficiency of the existing code.

Dependency Display

Close | Refresh | Clear

Class Name	Variable Name	Variable Type	Field Type	Method Name	To Modify	Status	Date
ClientJMeterEngine	rsl	RemoteSampleLi...	method field	makeListenersRe...	☐	Unvisited	
IncrementalJMeterEngine	catClass	Category	class field	NA	☐	Unvisited	
IncrementalJMeterEngine	threads	JMeterThread[]	method field	runTest	☐	Unvisited	
IncrementalJMeterEngine	threadList	Thread[]	method field	runTest	☐	Unvisited	
IncrementalJMeterEngine	iter	Iterator	method field	runTest	☐	Unvisited	
IncrementalJMeterEngine	group	ThreadGroup	method field	runTest	☐	Unvisited	
JMeterEngine	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
ReflectionJMeterEngine	catClass	Category	class field	NA	☐	Unvisited	
ReflectionJMeterEngine	jMeterThread	JMeterThread	method field	runTest	☐	Visited	
ReflectionJMeterEngine	iter	Iterator	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	thisThreadGroup	ThreadGroup	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	control	GenericController	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	subControllerList	List	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	configElementList	List	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	subControllersIter...	Iterator	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	samplerController	SamplerController	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	lc	LoopController	method field	runTest	☐	Unvisited	
ReflectionJMeterEngine	newThread	Thread	method field	runTest	☐	Unvisited	
RemoteJMeterEngine	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
RemoteJMeterEngineImpl	backingEngine	JMeterEngine	class field	NA	☐	Unvisited	
RemoteJMeterEngineImpl	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
RemoteJMeterEngineImpl	engine	RemoteJMeterEng...	method field	main	☐	Unvisited	
StandardJMeterEngine	threadGroups	Collection	class field	NA	☐	Unvisited	
StandardJMeterEngine	allThreads	Collection	class field	NA	☐	Unvisited	
StandardJMeterEngine	tGroup	ThreadGroup	formal parameter	addThreadGroup	☐	Unvisited	
StandardJMeterEngine	threads	JMeterThread[]	method field	runTest	☐	Unvisited	
StandardJMeterEngine	iter	Iterator	method field	runTest	☐	Unvisited	
StandardJMeterEngine	group	ThreadGroup	method field	runTest	☐	Unvisited	
StandardJMeterEngine	newThread	Thread	method field	runTest	☐	Unvisited	
StandardJMeterEngine	iter	Iterator	method field	stopTest	☐	Unvisited	
StandardJMeterEngine	item	JMeterThread	method field	stopTest	☑	Unvisited	
ButtonPanel	add	JButton	class field	NA	☐	Unvisited	
ButtonPanel	delete	JButton	class field	NA	☐	Unvisited	
ButtonPanel	edit	JButton	class field	NA	☐	Unvisited	
ButtonPanel	load	JButton	class field	NA	☐	Unvisited	
ButtonPanel	save	JButton	class field	NA	☐	Unvisited	
ButtonPanel	d	Dimension	method field	init	☐	Unvisited	
ButtonPanel	g	GridBagLayout	method field	init	☐	Unvisited	
ButtonPanel	c	GridBagConstraints	method field	init	☐	Unvisited	
ButtonPanel	listener	ActionListener	formal parameter	addButtonListener	☐	Unvisited	
FileDialoger	promptToOpenFile	JFileChooser	method return	promptToOpenFile	☐	Unvisited	

Start | Jtracker10[1].08.doc - Mi... | JBuilder - D:/JTracker1/u... | Dependency Display | 1:56 AM

Figure 4. *JTracker* window after the mark of class *ReflectionJMeterEngine* was cleared

Change propagation has been investigated in [15], [5], [7]. In [7], the author defines evolving interoperation graphs that are used as a theoretical model of change propagation. Interesting properties of change propagation were investigated in [16]. For some changes, change propagation can be substantial and involve many classes. Case study of [4] presented a change that involved more than 30 classes.

5. Conclusions and Future Work

In this paper, we introduced a tool called *JTracker* to help the programmer during change propagation. *JTracker* is a prototype tool that detects dependency relationships in Java code. The case study of *JMeter* showed that *JTracker* is able to work on practical systems. *JTracker* saves programmer's time and helps to avoid omissions during change propagation.

Future work includes improvements in the precision of the analysis part of *JTracker* and avoidance of the situation where *JTracker* marks classes that do not need change. More precise algorithms of the program analysis than the ones used so far will accomplish this. Another direction for the analysis is to improve recall and in particular to include hidden dependencies of [16] into the database of dependencies of *JTracker.*

Based on our case study, we believe that *JTracker* can significantly help the programmers during the incremental change and hence improve the productivity and quality of their work.

References

[1] T.Biggerstaff, B.Mitbander, and D.Webster, “Program Understanding and the Concept Assignment Problem”, Communications of the ACM 37, No.5, May 1994, pp. 72-83

[2] K. Chen, V. Rajlich, “Case Study of Feature Location Using Dependence Graph”, Proceeding of IWPC’00, IEEE Computer Society Press, Los Alamitos, CA, pp.241-249

[3] K. Chen, V. Rajlich, “RIPPLES: Tool for Change in Legacy Software”, Proceedings of IEEE International Conference on Software Maintenance, 2001, IEEE Computer Society Press, pp.230-239

[4] M. Lindvall, K. Sandahl, “Traceability Aspects of Impact Analysis in Object-Oriented Systems”, J. Soft. Maint: Res. Pract, 10, 1998, pp.37-57

[5] Luqi, “A Graph Model for Software Evolution”, IEEE Trans. On Software Engineering, 1990, pp.917-927

[6] R. Fanta, V.Rajlich, “Reengineering an Object Oriented Code”, In Proceedings of IEEE International Conference on Software Maintenance, 1998, IEEE Computer Society Press, pp.238-246

[7] V.Rajlich, “Modeling Software Evolution by Evolving Interoperation Graphs”, In Annals of Software Engineering, Vol. 9, 2000, pp.235-248

[8] V.Rajlich, P. Gosavi, “A Case Study of Unanticipated Incremental Change”, Proceeding IEEE Int. Conference on Software Maintenance, 2002, pp.442-451

[9] V. Rajlich, K.H. Bennett, “The Staged Model of The Software Lifecycle”, IEEE Computer, July 2000, pp.66-71

[10] V. Rajlich, “A Methodology for Incremental Change”, In M. Marchesi, G. Succi, D. Wells, L. Williams, Extreme Programming Perspectives, Addison Wesley, 2002, pp.201-214

[11] V. Rajlich, N. Damaskinos, P. Linos, W. Khorshid, “VIFOR: A Tool for Software Maintenance”, Software Practice and Experience, Vol. 20(1), January 1990, pp.67-77

[12] Lance Tokuda, Don Batory, “Automating Three Modes of Evolution for Object-Oriented Software Architectures”, In proceedings of 5th Conference on Object-Oriented Technologies, (COOTS’99), May 1999

[13] N. Wilde, Michael Scully, “Software Reconnaissance: Mapping Program Features to Code”, Software Maintenance: Research and Practice, 1995, Vol.7, pp.49-62

[14] N. Wilde, T. Gust, “Locating User Functionality in Old Code”, Proceedings of Conference on Software Maintenance 1992, Orlando, Florida, 1992, pp.200-205

[15] S.S. Yau, R.A. Nicholl, J.J. Tsai, S. Liu, “An Integrated Life-Cycle Model for Software Maintenance”, IEEE Trans. Software Engineering, 15(7), 1988, pp.58-95

[16] Z. Yu, V. Rajlich, “Hidden Dependencies in Program Comprehension and Change Propagation”, In Proceedings of 9th International Workshop on Program Comprehension, 2001, IEEE Computer Society, pp.293-299

[17] http://www.borland.com/jbuilder

[18] http://jakarta.apache.org/jmeter

0-7695-1902-4/03 $17.00 © 2003 IEEE