Towards a combined static and dynamic analysis approach for feature location

Yasaman Molazadeh
Supervisor: Dr. Juergen Rilling
Department of Electrical and Computer Engineering
Concordia University

Overview

- Feature Location Problem
- Different Approaches and Techniques
- A combined approach based on Impact Analysis
- A case study
- Conclusions
- Future work

Introduction

- **Feature Location Problem:**
  Identifying the relationships between the user's view and the programmer's view [Wilde et al. 1992].

User's view: FEATURES = { f1, f2, . . . , fn}
Programmer's view: UNITS = {u1, u2, . . . , um}.

The feature location problem is to recover the implementation relationships over FEATURES UNITS.

Two types of relationships:
- **Relevant relation** (all units related to a feature)
- **Specific relation** (units related to a feature and not to any other feature's implementation)

Different Approaches:

	Interactive	**Non interactive**
Static	Chen and Rajlich 2000 Griswold et al. 2001	Zhao et al 2006
Dynamic	Mehta and Heineman 2001	Wilde and Scully 1995 Wong et al. 1999 Rohatgi 2007
Combined	Eisenbarth et al. 2003	Poshyvanyk et al. 2007 Antoniol and Gueheneuc 2005

Static Approaches:

- **Call Graphs**
- **BRCG (Branch Reserving Call Graph)**
  Zhao et al. 2006 - SNIAFL
  Qin et al. 2003
- **CDG (Component Dependency Graph)**
  Rohatgi et al. 2007
- **Abstract System Dependency Graph (ASDG)**
  Chen and Rajlich 2000
- ...

Limitation:

- They can rarely identify entities related to a specific execution scenario exactly.
- Tend to be imprecise due to not handling dynamic binding, pointers,….

Dynamic Approaches:

- Based on dynamic execution traces
- Test cases (non Interactive JUNIT, interactive user input)

- Wilde and Scully 1995 - Software Reconnaissance
- Wong et al. 1999
- Mehta and Heineman 2001
- …

Limitation:

- How many traces are needed
- Design of test cases is a difficult task
- They are unable to distinguish between overlapping features

Combined Approaches:

- **Eisenbarth et al. 2003**
  - Concept lattice of each feature using traces
  - CDG

- **Antoniol and Gueheneuc 2005**
  - A model of program architecture in AOL format
  - Collecting trace information based on scenario execution
  - Knowledge-based filtering and probabilistic ranking methods

- **Poshyvanyk et al. 2007**
  - One trace
  - IR
- ...

Interactive Approaches:

- **Biggerstaff et al. 1993**
  - Concept assignment problem.
  - Call graph and the program clustering graph
  - Some regular-expression-based matching tools
- **Chen and Rajlich 2000**
  - Abstract system dependency graph (ASDG)
- **Griswold et al. 2001**
  - Lexical searches
- **Robillard and Murphy 2002**
  - Concern graphs
- …

Different Techniques:

- **Information Retrieval and Latent Semantic indexing(LSI)**
  - Deerwester 1990
  - Antoniol et al. 2000,2002
  - Marcus and Maletic 2003
  - Cubranic and Murphy 2003
  - Zhao et al. 2006 – SNIAFL
- **Scenario Based Probabilistic ranking (SBP)**
  - Poshyvanyk et al. 2006
- **Concept Analysis**
  - Eisenbarth et al. 2001, 2003
  - Poshyvanyk and Marcus 2007
- **Impact Analysis**
  - Rohatgi 2007
- …

Impact Analysis – a combined approach

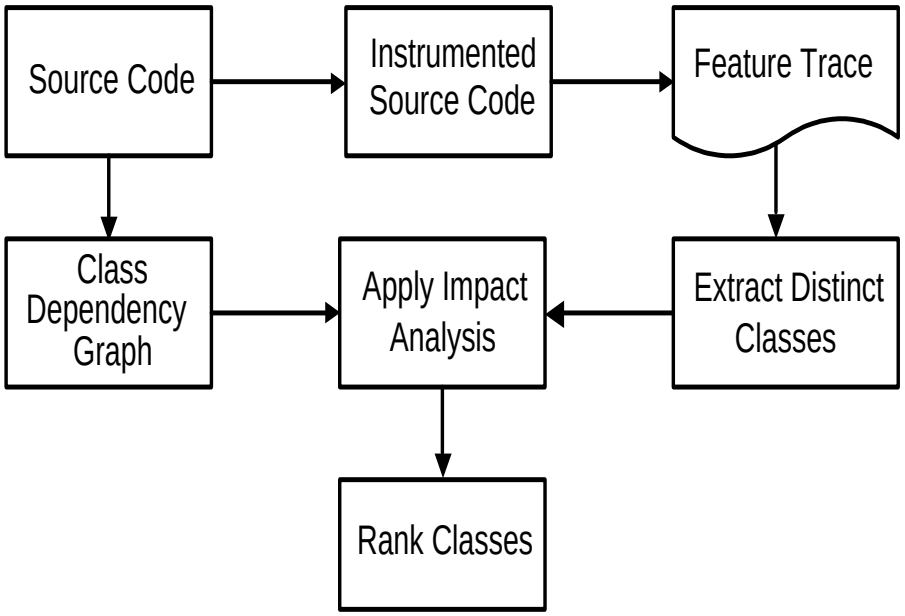
- **Hypothesis:**

  The smaller the impact set related to a component modification, the more likely it is that the component set is specific to a feature.

- **Combination of:**
  - Dynamic Analysis: An execution trace that corresponds to a software feature
  - Static Analysis: A CDG

- **Advantage :**
  - Requires only the creation of **one** trace.
  - Only limited system knowledge is required to perform the analysis (only the input conditions for the specific feature).
  - Can provide a good approximation of the core components specific to a feature.

Approach - Overview



- A trace is generated by exercising the feature of interest and *distinct* classes are extracted from this trace.
- Two impact analysis metrics are applied on the extracted classes.
- These impact measures, classes are ranked based on these impact metrics.

Approach - Basic Definitions

- **Class Afferent Impact CAI(c)**
  The set of classes that are affected directly/indirectly when *c* is modified.

- **Class Efferent Impact CEI(c)** :
  The set of classes that will affect c if they change. These are all classes *c* directly/indirectly depends on.

- **Package Afferent Impact PAI(c):**
  The set of packages affected by a modification of c. Every package in a system is considered as a separate packages.

Approach - Two Way Impact Metric (TWI) and Weighted TWI (WTWI)

Two Way Impact Metric (TWI)

Description: The TWI metric considers both the afferent and efferent impact set at the component level.

Assumption: The modification of a feature specific class has a very low impact on the remaining parts of the system.

Weighted Two Way Impact Metric (WTWI)

Description: Uses additional system information by considering also the number of packages affected by a class modification (PAI).

Assumption: A class affecting five classes from three different packages is more likely to be part of the execution profile of several features than a class affecting five classes all located in one package.

Classes with large CAI correspond to be non-feature specific classes.

Efferent impact with a lower emphasis

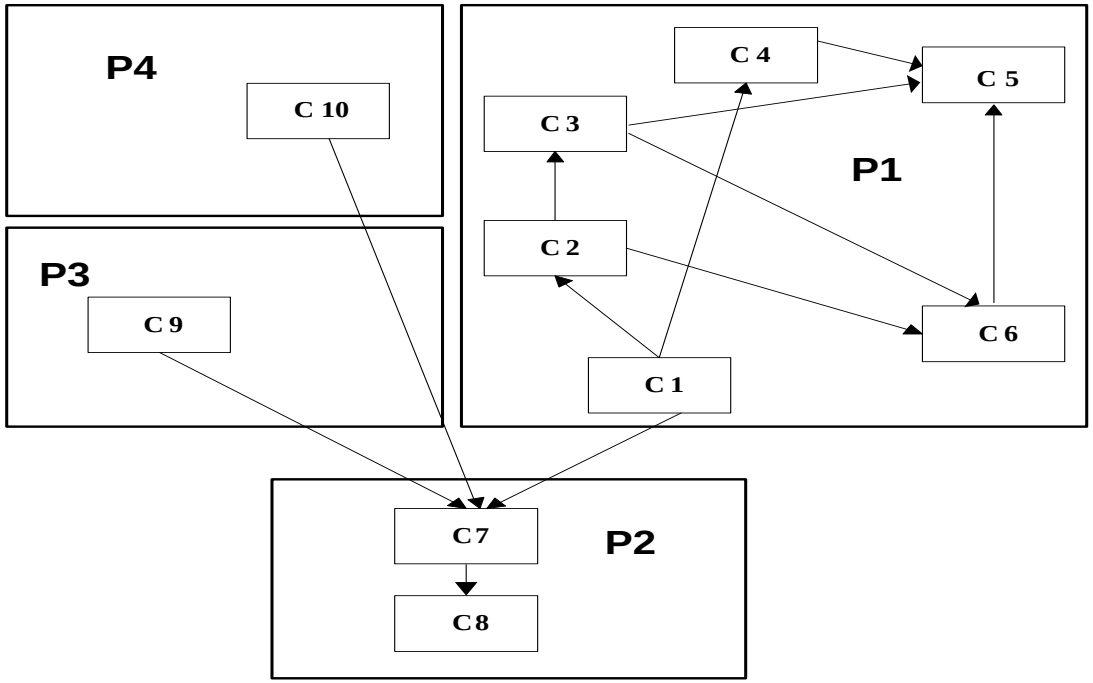
$$TWI\ (c) = \frac{CAI\,(c)}{|\ S\ |} \times \frac{Log(\frac{|\ S\ |}{CEI\,(c)+1})}{Log(|\ S\ |)}$$

Package afferent impact

$$WTWI\ \ (c) = TWI\ (c) \times \frac{PAI\,(c)}{|\ P\ |}$$

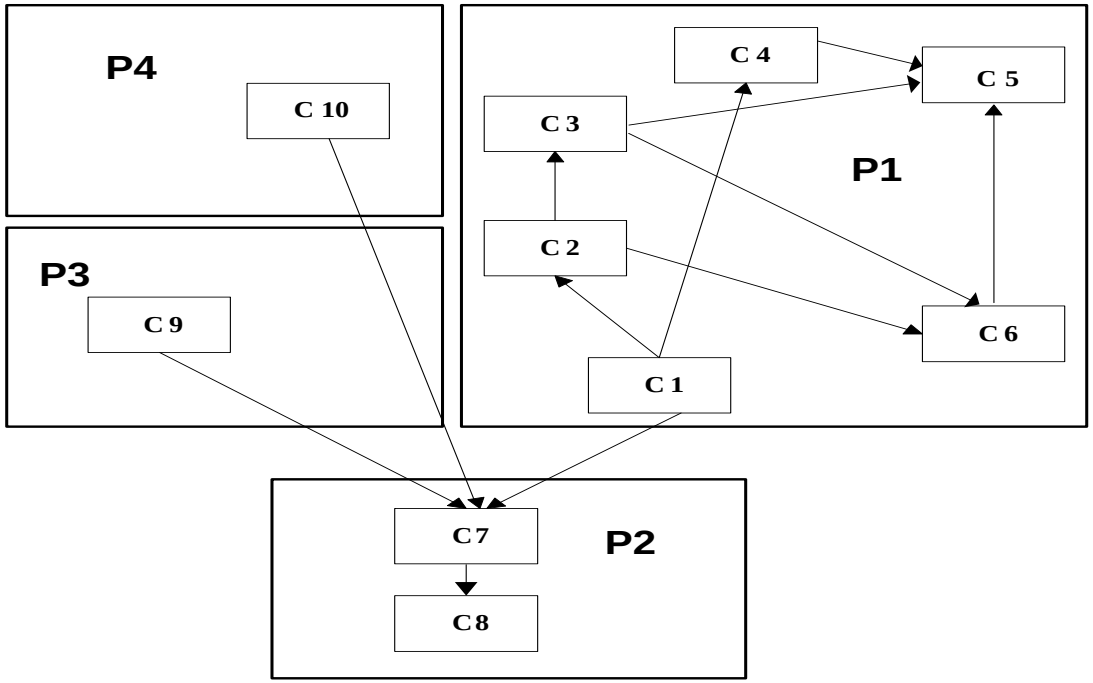
Example - Two Way Impact Metric (TWI) and

Classes	CAI	CEI	TWI
C1	**0**	**7**	**0**
C2	**1**	**3**	**0.04**
C4	**1**	**1**	**0.08**
C3	**2**	**2**	**0.12**
C6	**3**	**1**	**0.25**
C7	3	1	0.25
C5	**5**	**0**	**0.63**
C8	5	0	0.63



Example - Weighted TWI (WTWI)

Classes	PAI	WTWI
C1	**1**	**0**
C2	**1**	**0.01**
C4	**1**	**0.02**
C3	**1**	**0.03**
C5	**1**	**0.16**
C6	**1**	**0.06**
C7	4	0.25
C8	4	0.63



Case Study

- **Checkstyle v3.3**
  - Java Based System.
  - A development tool to help programmers writing Java code that adheres to a coding standard.
  - 17 packages, 210 classes, and 130 KLOC.
  - Well documented systems, allowed an easy manual review.

- Software Feature: **Check Code -** Analyzes Java code for coding problems.
  - Check code feature trace contains 68 distinct classes. Manual analysis showed that 32 of these classes can be considered feature specific.
  - Coding package contains most important classes required for the implementation of check code feature.

Feature Location for the CheckCode Feature Using TWI– Part 1

Class	CAI	CEI	TWI*1000
coding.ExplicitInitializationCheck	1	22	0.59
coding.MagicNumberCheck	1	22	0.59
coding.IllegalTokenTextCheck	1	21	0.6
coding.IllegalTypeCheck	1	21	0.6
coding.NestedIfDepthCheck	1	21	0.6
coding.RedundantThrowsCheck	1	21	0.6
coding.SuperCloneCheck	1	21	0.6
coding.SuperFinalizeCheck	1	21	0.6
coding.DeclarationOrderCheck	1	20	0.62
coding.HiddenFieldCheck	1	20	0.62
coding.IllegalCatchCheck	1	20	0.62
coding.JUnitTestCaseCheck	1	20	0.62
coding.MissingSwitchDefaultCheck	1	20	0.62
coding.CovariantEqualsCheck	1	19	0.63
coding.IllegalInstantiationCheck	1	19	0.63
coding.IllegalTokenCheck	1	19	0.63
coding.NestedTryDepthCheck	1	19	0.63
coding.ArrayTrailingCommaCheck	1	18	0.64
coding.AvoidInlineCondCheck	1	18	0.64
coding.DoubleCheckedLockCheck	1	18	0.64
coding.EmptyStatementCheck	1	18	0.64
coding.EqualsHashCodeCheck	1	18	0.64
coding.FinalLocalVariableCheck	1	18	0.64
coding.InnerAssignmentCheck	1	18	0.64
coding.PackageDeclarationCheck	1	18	0.64
coding.ParameterAssignmentCheck	1	18	0.64
coding.ReturnCountCheck	1	18	0.64
coding.SimplifyBooleanExpCheck	1	18	0.64
coding.SimplifyBooleanReturnCheck	1	18	0.64
coding.StringLiteralEqualityCheck	1	18	0.64
checkstyle.DefaultConfiguration	*1*	*2*	*1.14*
checks.DescendantTokenCheck	*2*	*19*	*1.26*
checks.GenericIllegalRegexpCheck	*2*	*19*	*1.26*
checkstyle.TreeWalker	*3*	*32*	*1.49*

Feature Location for the CheckCode Feature Using TWI– Part 2

checkstyle.Checker	*3*	*21*	*1.81*
checks.AbstractTypeAwareCheck	*3*	*20*	*1.85*
coding.AbstractSuperCheck	3	20	1.85
coding.AbstractNestedDepthCheck	3	18	1.93
checkstyle.DefaultLogger	*3*	*11*	*2.3*
checkstyle.ConfigurationLoader	*3*	*3*	*3.19*
checkstyle.PropertiesExpander	*3*	*2*	*3.42*
checkstyle.PackageNamesLoader	*4*	*4*	*4.01*
grammars.GeneratedJava14Lexer	*4*	*3*	*4.25*
checkstyle.PropertyCacheFile	*4*	*2*	*4.56*
grammars.GeneratedJava14Recognizer	*4*	*2*	*4.56*
checkstyle.StringArrayReader	*4*	*1*	*4.99*
checkstyle.PackageObjectFactory	*5*	*2*	*5.69*
apis.FilterSet	6	5	5.72
apis.AbstractFileSetCheck	8	13	5.81
checkstyle.DefaultContext	*7*	*2*	*7.97*
checks.CheckUtils	*8*	*3*	*8.49*
checkstyle.AbstractLoader	*7*	*1*	*8.73*
checks.AbstractFormatCheck	*17*	*18*	*10.95*
apis.TokenTypes	9	1	11.23
apis.AuditEvent	13	3	13.8
apis.ScopeUtils	19	3	20.17
apis.FullIdent	21	2	23.92
apis.Scope	20	1	24.95
apis.Check	126	17	82.99
apis.AbstractViolationReporter	132	8	111.47
apis.FileContents	127	4	127.26
apis.AutomaticBean	140	6	127.65
apis.LocalizedMessages	132	3	140.16
apis.DetailAST	131	1	163.44
apis.LocalizedMessage	148	2	168.57
apis.Utils	136	1	169.68
apis.StrArrayConverter	141	1	175.92
apis.SeverityLevel	150	1	187.15

Misplaced classes

apis
a Checkstyle
utility package

Feature location with *WTWI* applied to the CheckCode feature – Part 1

Class	PAI	WTWI*1000
coding.ExplicitInitializationCheck	1	0.03
coding.MagicNumberCheck	1	0.03
coding.IllegalTokenTextCheck	1	0.04
coding.IllegalTypeCheck	1	0.04
coding.NestedIfDepthCheck	1	0.04
coding.RedundantThrowsCheck	1	0.04
coding.SuperCloneCheck	1	0.04
coding.SuperFinalizeCheck	1	0.04
coding.DeclarationOrderCheck	1	0.04
coding.HiddenFieldCheck	1	0.04
coding.IllegalCatchCheck	1	0.04
coding.JUnitTestCaseCheck	1	0.04
coding.MissingSwitchDefaultCheck	1	0.04
coding.CovariantEqualsCheck	1	0.04
coding.IllegalInstantiationCheck	1	0.04
coding.IllegalTokenCheck	1	0.04
coding.NestedTryDepthCheck	1	0.04
coding.ArrayTrailingCommaCheck	1	0.04
coding.AvoidInlineConditionalsCheck	1	0.04
coding.DoubleCheckedLockCheck	1	0.04
coding.EmptyStatementCheck	1	0.04
coding.EqualsHashCodeCheck	1	0.04
coding.FinalLocalVariableCheck	1	0.04
coding.InnerAssignmentCheck	1	0.04
coding.PackageDeclarationCheck	1	0.04
coding.ParameterAssignmentCheck	1	0.04
coding.ReturnCountCheck	1	0.04
coding.SimplifyBooleanExpCheck	1	0.04
coding.SimplifyBooleanReturnCheck	1	0.04
coding.StringLiteralEqualityCheck	1	0.04
checkstyle.DefaultConfiguration	*1*	*0.07*
checkstyle.Checker	*1*	*0.11*
coding.AbstractSuperCheck	1	0.11
coding.AbstractNestedDepthCheck	1	0.11

Using WTWI –an improved clustering has been achieved

Feature location with *WTWI* applied to the CheckCode feature – Part 2

checkstyle.DefaultLogger	*1*	*0.14*
checks.DescendantTokenCheck	*2*	*0.15*
checks.GenericIllegalRegexpCheck	*2*	*0.15*
checkstyle.TreeWalker	*2*	*0.18*
checkstyle.ConfigurationLoader	*1*	*0.19*
checkstyle.PropertiesExpander	*1*	*0.2*
checkstyle.PackageNamesLoader	*1*	*0.24*
checks.AbstractTypeAwareCheck	*3*	*0.33*
checkstyle.PackageObjectFactory	*1*	*0.33*
checkstyle.PropertyCacheFile	*2*	*0.54*
checkstyle.StringArrayReader	*2*	*0.59*
grammars.GeneratedJava14Lexer	*3*	*0.75*
grammars.GeneratedJava14Recognizer	*3*	*0.8*
checkstyle.DefaultContext	*2*	*0.94*
apis.FilterSet	3	1.01
checkstyle.AbstractLoader	*2*	*1.03*
checks.CheckUtils	*3*	*1.5*
apis.AbstractFileSetCheck	6	2.05
apis.AuditEvent	3	2.44
checks.AbstractFormatCheck	*4*	*2.58*
apis.TokenTypes	5	3.3
apis.ScopeUtils	7	8.31
apis.Scope	7	10.27
apis.FullIdent	8	11.26
apis.Check	14	68.34
apis.AbstractViolationReporter	15	98.35
apis.FileContents	14	104.8
apis.AutomaticBean	16	120.14
apis.LocalizedMessages	15	123.67
apis.DetailAST	14	134.6
apis.StrArrayConverter	15	155.22
apis.LocalizedMessage	16	158.65
apis.Utils	16	159.7
apis.SeverityLevel	16	176.14

Conclusions

- Identified and ranked classes that are most specific to a feature.
- Straight forward approach (re-apply existing techniques and tools).
- WTWI (Weighted Two Way Impact) shows improved results over TWI by considering system architectural information =>
  - Improved grouping and
  - Closer ranking of feature specific classes.

Limitations:

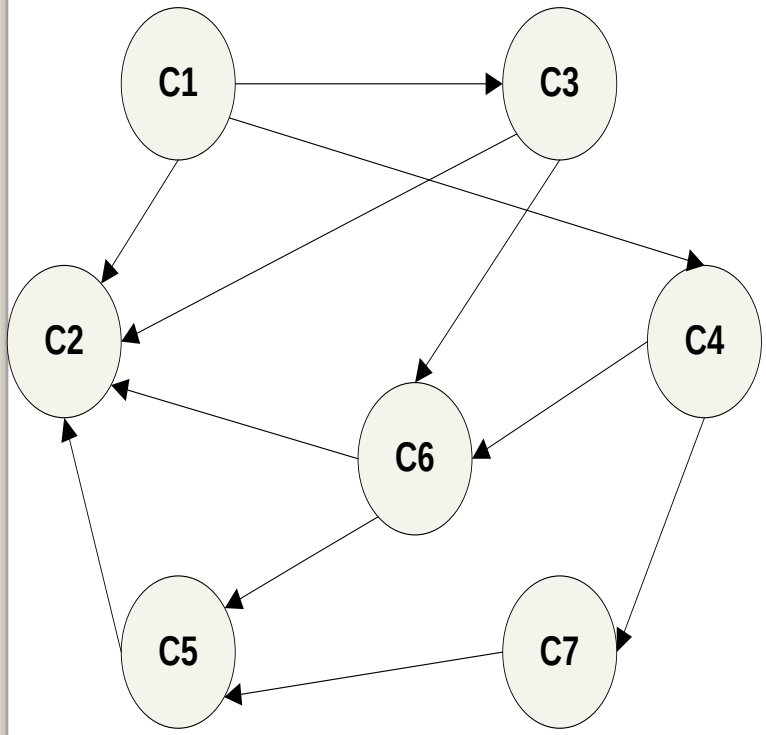
- Quality of feature trace
- Structure of the source code (high coupled and low cohesive)
- Interpretation of clustering

Future Work

- Use of thresholds to support automatic grouping.
- Investigate the use of other metrics and further refine current ones.
- Evaluation of the approach
  - based on *precision* and *recall* and
  - direct comparison with other feature location approaches.
- Combine impact analysis approach with other techniques like IR
- Further evaluations are needed for larger, less well designed systems.

Thank You

Approach *continued* -Impact Analysis



Component Dependency Graph

- The impact set for a component modification involving a class *C is defined by all classes that are directly or indirectly* affected by the modification of class *C.*
- Impact (C5) = (C6 (C3, C1, C4), C7 (C4, C1), C4 (C1))
- The classes *C6, C7 and C4 are those that are directly* concerned with the modification of class *C5.*
- *The classes* inside the inner brackets *C3, C1, C4 are these classes that* are indirectly affected by modification of class *C5.*
- *Note : For the* purpose of this research only the set of distinct classes in the impact set are considered.

REFERENCES

ANTONIOL, G., CANFORA, G., CASAZZA, G., AND DELUCIA, A. 2000. Information retrieval models for recovering traceability links between code and documentation. In Proceedings of the 16th International Conference on Software Maintenance. (San Jose, Calif. Oct.). IEEE Computer Society, Washington, D.C. 40–49.

ANTONIOL, G., CANFORA, G., CASAZZA, G., DELUCIA, A., AND MERLO, E. 2002. Recovering traceability links between code and documentation. IEEE Trans. Soft. Eng. 28, 10, 970–983.

BIGGERSTAFF, T., MITBANDER, B., AND WEBSTER, D. 1993. The concept assignment problem in program understanding. In Proceedings of the 15th International Conference on Software Engineering (Baltimore, Md., May.). IEEE Computer Society, Los Alamitos, Calif. 482–498.

CHEN, K. AND RAJLICH, V. 2000. Case study of feature location using dependence graph. In Proceedings of the 8th International Workshop on Program Comprehension (Limerick, Ireland, June). IEEE Computer Society, Washington, D.C. 241–249.

CUBRANIC, D. AND MURPHY, G. C. 2003. Hipikat: Recommending pertinent software development artifacts. In Proceedings of the 25th International Conference on Software Engineering (Portland, Oreg., May). IEEE Computer Society, Washington, D.C. 408–418.

DEERWESTER, S., DUMAIS, S. T., FURNAS, G. W., LANDAUER, T. K., AND HARSHMAN, R. 1990. Indexing by latent semantic analysis. J. Am. Soc. Inf. Sci. 41, 391–407.

EISENBARTH, T., KOSCHKE, R., AND SIMON, D. 2003. Locating features in source code. IEEE Trans. Softw. Eng. 29, 3, 210–224.

GRISWOLD,W. G., YUAN, J. J., AND KATO, Y. 2001. Exploiting the map metaphor in a tool for software evolution. In Proceedings of the 23rd International Conference on Software Engineering (Toronto, Ont. May). IEEE Computer Society, Washington, D.C. 265–274.

MARCUS, A. AND MALETIC, J. I. 2003. Recovering documentation-to-source-code traceability links using latent semantic indexing. In Proceedings of the 25th International Conference on Software Engineering (Portland, Oreg., May). IEEE Computer Society, Washington, D.C. 125–135.

MEHTA, A., AND HEINEMAN, G., T., 2001. Evolving legacy systems features using regression test cases and components. In Proceedings of the 4th International Workshop on Principles of Software Evolution(Vienna, Austria). Session 7B: Verification, validation and testing,190–193.

POSHYVANYK, D., Gueheneuc, Y., G., Marcus, A., Antoniol, G. AND Rajlich, V., 2007. Feature Location using Probabilistic Ranking of Methods based on Execution Scenarios and Information Retrieval, IEEE Transactions on Software Engineering, 33(6), pp. 420-432.

QIN, T., ZHANG, L., ZHOU, Z., HAO, D., AND SUN, J. 2003. Discovering use cases from source code using the branch-reserving call graph. In Proceedings of the 10th Asia-Pacific Software Engineering Conference (Chiang Mai, Thailand, Dec.). IEEE Computer Society, Washington, D.C. 60–67.

ROBILLARD, M. P. AND MURPHY, G. C. 2002. Concern graphs: Finding and describing concerns using structural program dependencies. In Proceedings of the 24th International Conference on Software Engineering (Orlando, Fla., May). IEEE Computer Society, Washington, D.C. 406–416.

ROHATGI, A., HAMOU-LHADJ, A., AND Rilling, J., 2007. "Feature Location Based on Impact Analysis", In Proc. of 11th IASTED Int. Conf. on Software Engineering and Applications.

WILDE, N., GOMEZ, J. A., GUST, T., AND STRASBURG, D. 1992. Locating user functionality in old code. In Proceedings of the 8th International Conference on Software Maintenance (Orlando, Fla., Nov.). IEEE Computer Society, Washington, D.C. 200–205.

WILDE, N. AND SCULLY, M. C. 1995. Software reconnaissance: Mapping program features to code. J. Softw. Maintenance: Res. Pract. 7, 1, 49–62.

WONG, W. E., GOKHALE, S. S., HORGAN, J. R., AND TRIVEDI, K. S. 1999. Locating program features using execution slices. In Proceedings of the 2nd Symposium on Application-Specific Systems and Software Engineering Technology (Richardson, Tex., Mar.). IEEE Computer Society, Washington,D.C. 194–203.

ZHAO, W., ZHANG, L., LIU, Y., SUN, J., AND YANG, F. 2006. SNIAFL: Towards a static non-interactive approach to feature location. ACM Transactions on Software Engineering and Methodology, 15(2):195-226, April 2006. CODEN ATSMER. ISSN 1049-331X.