Can we predict dependencies using domain information?

Amir Aryani*, Fabrizio Perin†, Mircea Lungu†, Abdun Naser Mahmood*, Oscar Nierstrasz†
*RMIT University, Australia
{amir.aryani,abdun.mahmood}@rmit.edu.au
http://www.rmit.edu.au/
†Software Composition Group
University of Bern, Switzerland
{perin,lungu,oscar}@iam.unibe.ch
http://scg.unibe.ch

***Abstract*—Software dependencies play a vital role in program comprehension, change impact analysis and other software maintenance activities. Traditionally, these activities are supported by source code analysis; however, the source code is sometimes inaccessible, and not all stakeholders have adequate knowledge to perform such analysis. For example, non-technical domain experts and consultants raise most maintenance requests; however, they cannot predict the cost and impact of the requested changes without the support of the developers.**

We propose a novel approach to predict software dependencies by exploiting coupling present in domain-level information. Our approach is independent of the software implementation; hence, it can be used to evaluate architectural dependencies without access to the source code or the database.

We evaluate our approach with a case study on a large-scale enterprise system, in which we demonstrate how up to 68% of the source code dependencies and 77% of the database dependencies are predicted solely based on domain information.

I. Introduction

When software maintainers change a software entity, they have to search for other related entities and update them accordingly. This is not a trivial task, and many bugs are introduced by programmers who fail to properly propagate the change [1]. Knowledge of software dependencies is vital to many change impact analysis methods [2], [3], [4], [5].

Source code analysis can be used to trace dependencies [6]. However, it is not easy to apply in many situations, one typical example being projects with heterogeneous source code. In addition, code analysis is not an appropriate option for all stakeholders, *e.g.,* support staff, consultants and non-technical domain experts.

Large majority of enterprise software systems are derived from domains where requirements are uncertain, and are likely to change during the software's lifetime [7]. In these domains, the domain experts are the primary source of information for evaluating requirements [8]. These domain experts drive software evolution by continuously asking for new functionality or requesting changes to existing ones. Unfortunately, domain experts are in a poor position to estimate the impact of the changes they request since they typically do not have inside knowledge of the internal dependencies of the software system.

Enterprise software systems are constructed to model business domains [7]. It is reasonable to expect that real-world dependencies are therefore reflected in the software itself. Consequently, we hypothesize that *software dependencies can be predicted by exploiting domain information*.

In this paper, we propose a novel approach to predicting software dependencies based on the notion of domain-based coupling [9] which is derived from the domain-level relationships between software components. The proposed approach is independent of the software implementation; hence, it can assist software maintainers where source code analysis is not available, or it can be used to evaluate the change propagation prior to changing the source code. In addition, it solely relies on domain information, allowing non-technical domain experts to predict change propagation without the support of programmers.

We evaluate our approach with a case study of a large-scale enterprise system, called ADempiere, where we demonstrate how domain information can be used to identify architectural dependencies in the source code and database layers.

The contributions of this paper are as follows:

- We refine the previously defined domain-based coupling [9], and extend the previous method of selecting the highly coupled components by an automated clustering technique.
- We formally define architectural dependencies and propose a model to trace dependencies among source code, database and user interface components.
- We present an empirical study of one of the biggest open source enterprise systems, demonstrating how domain-based coupling can be used to predict the source code and database dependencies.

The rest of this paper is organised as follows: Section II introduces the system under analysis. Section III describes domain-based coupling analysis. Section IV presents the dependency analysis. Section V demonstrates the evaluation results. Section VI discusses the threats to the validity of our findings. Section VII presents the related work, and finally, Section VIII concludes this paper with a discussion about future areas of investigation.

II. Case Study: ADempiere

We scouted the open-source software landscape for a suitable open-source system to use as a case study for our analysis. After considering several candidates, we eventually settled on ADempiere[1], an Enterprise Resource Planning (ERP) software. The qualities that persuaded us to choose ADempiere as our case study are:

Well defined business domain: An ERP system integrates internal and external management information across an entire organisation, embracing accounting, manufacturing, sales and service, *etc.* Such a system manifests a strict separation between the expertise of the stakeholders and developers. This is the type of software which benefits mostly from domain-based coupling analysis.

Tiered architecture: The system manifests a clear separation between the different architectural tiers.

- The system has a rich set of UI components and four distinct front-ends from which the user can choose including a Java GUI and three web interfaces.
- The system heavily uses relational database management systems (*e.g.,* PostgreSQL and Oracle) for data storage as well as for storing business logic.

Evolving and active system: The ADempiere project traces its evolution back more than a decade. Created in September 2006 as a fork of the Compiere open-source ERP, itself founded in 1999, ADempiere soon reached the top five of the *SourceForge.net* enterprise software rankings. At the time of this publication, it is the first system among that top five. This is a measure of both the size of its developer community and its impact on the ERP software market.

Large-scale and complex design: The system represents cutting edge open-source software technology. It is a multi-language system that aggregates more than 6 million lines of code[2]. The core part is written in Java and contains more than 3,531 classes with more than half a million lines of code[3].

Figure 1 presents a high-level architectural view of the Java core of ADempiere. The view is obtained by aggregating the direct relationships in the system along the package hierarchy [10]. The area of every visible module is proportional to its number of lines of code. Every visible dependency is directed and has its width proportional to the number of abstracted low-level dependencies. Every module is represented as a modified treemap, with the sizes of the contained classes and modules proportional to their size in lines of code.

Active developers and users communities: The system has a very active associated community: often the mailing list has more than 800 messages per month, and the *SourceForge.net* page shows that ADempiere is downloaded more than 15,000 times per month. The system is used by a large number of companies around the world.

[1]http://www.adempiere.com

[2]Based on the public open source software directory at *Ohloh.net*

[3]Measured by ourselves, based on the Java code in the SVN repository at: https://adempiere.svn.sourceforge.net/svnroot/adempiere/tags/trunk_last/

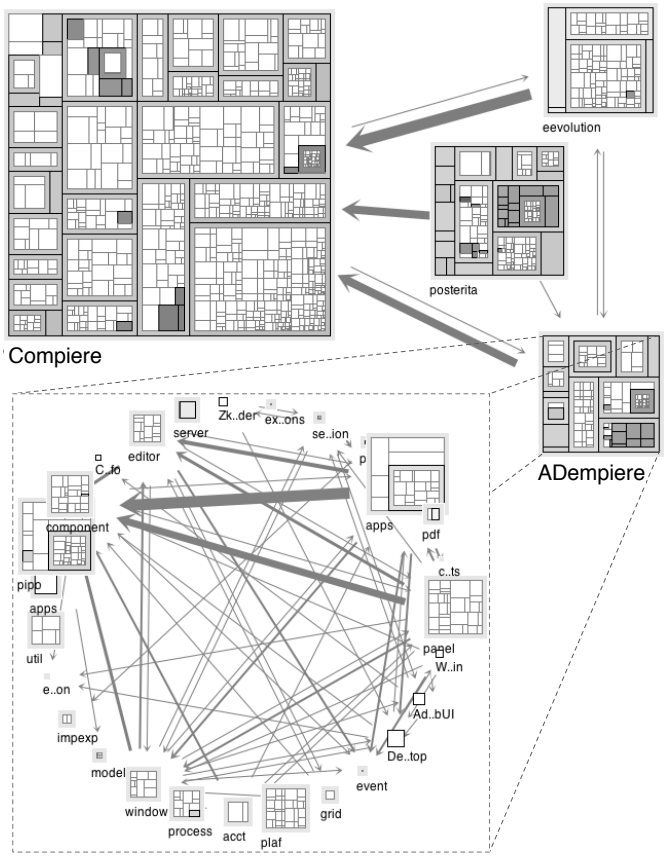


Fig. 1: A high level architectural analysis reveals that ADempiere is a highly complex Java system built and dependent on the older Compiere core.

For all these reasons, we deem ADempiere to be relevant and representative for enterprise systems and for the state of the art in open-source software at the moment of writing this article, and appropriate for our analysis.

III. Domain-Based Coupling Analysis

Domain information can reveal relationships among user interface components (UICs) [11]. In this section, we describe how the domain-based coupling [9] derived from software domain information; can be used to predict dependencies between the user interface components.

We use the following terminology when we talk about the domain model of a system:

- A *domain variable* is a variable unit of data which has a clear identity at the domain level.
- A *domain function* provides proactive or reactive domain-level behaviour of the system which includes at least one domain variable as an input or output.
- A *user interface component (UIC)* is a system component which directly interacts with users, and contains one or more domain functions.

For example, in a business software system, a data entry form is considered a UIC, the entry and editing of business

information are domain functions, and the data fields visible on the form are domain variables.

A. Notations and Definitions

Most of this section quotes our earlier works [9], [11] with the exception of new definitions of the number of common variables (Definition 2), and revised definitions of the domain-based coupling graph (Definition 3).

We adopt the following conventions in this work. For $R, Q \subseteq A \times A$, we denote by $R.Q$ their composition, *i.e.,* $x.R.Q.y$ iff $\exists z : x.R.z \wedge z.Q.y$. We also denote by R^{-1} the inverse of R and by ID the identity relation.

Moreover we abbreviate $x.R = \{y | x.R.y\}$. We visualise relations as graphs, denoting by $G = (V, E, l)$ the graph G with vertices V, edges $E \subseteq V \times V$ and labels $l : E \to L$ for some label set L.

If L is a finite set of relation labels and $l_R \in L$ the name of R for any $R \in X$, then we define $REL(A, X)$ to be the labelled directed graph $REL(A, X) = (V, E, l)$ with $V = A, E = \bigcup_{R \in X} R$ such that: $(v, v') \in E$ *and* $l(v, v') = l_R$ *iff* $v.R.v'$ *for some* $R \in X$.

The three key element types are modelled as follows:

- *Domain variables* are modelled by a finite set V, called *variable symbols*.
- *Domain functions* are modelled by a finite set F, called *function symbols*, and the binary relation $USE \subseteq F \times V$ represents the relation between functions and variables as the input-output of the functions.
- *UICs* are modelled by a finite set C called the *component symbols*, and $HAS \subseteq C \times F$ represents the relation between components and functions.

For the rest of the paper, and without loss of generality, we assume that the system under analysis (SUA) is fixed, that is, V, F and C are fixed and so are their REF, USE and HAS relations.

Definition 1. *The* conceptual connection *relation* $CNC \subseteq C \times C$ *is defined by*

$$CNC = HAS.USE.USE^{-1}HAS^{-1}$$

The domain-based coupling between two components is derived from shared domain variables, based on the following measurements:

Definition 2. *Number of common variables among two UICs is modelled by the function* $\vartheta : C \times C \to \mathbb{R}$ *where*

$$\vartheta(c, c') = |c.HAS.USE \cap c'.HAS.USE|$$

Note that the definition of common domain variables is symmetric, i.e., $\vartheta(c, c') = \vartheta(c', c)$.

Definition 3. *The* domain-based coupling graph *of a SUA is the symmetric weighted graph* $G = (C, CNC \backslash ID, \omega)$ *where coupling weight function* $\omega : C \times C \to [0..1]$ *is*

$$\omega(c, c') = \frac{\vartheta(c, c')}{|c.HAS.USE \cup c'.HAS.USE|}$$

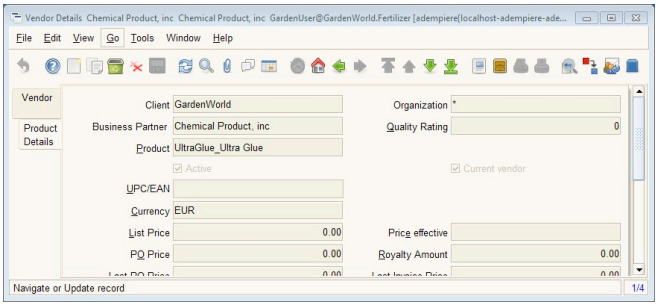


Fig. 2: ADEMPIERE: Vendor Details

It turns out that it is practically useful to weight domain relationships by their level of sharing domain variables. A threshold t can be used to select relevant coupling by their weight $\omega \geq t$.

In the following examples, we demonstrate how to derive domain-based coupling from UICs of ADEMPIERE, and then how to approximate dependencies from that coupling.

B. Example 1

In ADEMPIERE, *Vendor Details* (Figure 2) and *Import Product* are the UICs which we use in this example. *Vendor Details* (c_1) has 2 domain functions, and in total 25 domain variables, as follows:

$c_1.HAS =$ { Edit Vendor, Edit ProductDetails }.

$c_1.HAS.USE =$ { DeliveryTime, BusinessPartner, CostPerOrder, Currency, Vendor, Manufacturer, ListPrice,... }.

Import Product (c_2) contains one domain function and 42 domain variables as follows:

$c_2.HAS =$ { Import Products }.

$c_2.HAS.USE =$ { CostPerOrder, PriceEffective, Weight, BusinessPartner, SKU, UOM, Processed, Royalty,... }.

There are 18 common domain variables between these UICs as follows:

$c_1.HAS.USE \cap c_2.HAS.USE =$ { BusinessPartner, CostPerOrder, Currency, Discontinued, DiscontinuedAt, ListPrice, Manufacturer, MinOrderQty, OrderPackQty, PartnerCategory, PartnerProductKey, POPrice, PriceEffective, Product, PromisedDeliveryTime, Royalty, UOM, UPC/EAN }.

and in total 49 (42+25-18) variables used by either of these UICs; thus:

$$\vartheta(c_1, c_2) = 18$$
$$\omega(c_1, c_2) = 18/49 = 0.37$$

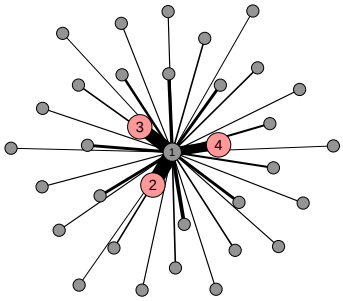
The next section demonstrates how to create a weighted graph from all CNC relations of *Vendor Details*.

C. Example 2

Now that we have explained the domain definitions, let's demonstrate how to use them for predicting dependencies. Imagine if a domain expert considers asking for an enhancement to *Vendor Details* (c_1), then given the domain information of ADEMPIERE, she can derive common domain variables (ϑ)

among c_1 and other UICs similar to what was described in the previous example.

Figure 3 shows there are 33 UICs for which the coupling weight with c_1 is greater than a given threshold $\omega \geq 0.5$. The selected threshold is applied to avoid weak results which do not likely lead to any architectural dependencies. This also reduces the density of the resulting domain-based coupling graph and makes it more readable. The results are illustrated (Figure 3) as a weighted graph where the edge width is proportional to ω, and edge length is proportional to $1/\omega$, *i.e.,* the stronger the coupling weight, the thicker is the edge and the closer the node to the center (c_1).



Legend: Nodes represent UICs and edges represent domain-based coupling. The tagged nodes are (1) Vendor Details, (2) Import Products, (3) Spare Parts and (4) Product Planning. Node size has no implication, but edge width is proportional to ω and edge length is proportional to $1/\omega$. For readability, the graph only contains $c_1.CNC$, excluding edges between other nodes.

Fig. 3: Vendor Details — domain-based coupling graph

The top 3 closest UICs are: *Import Products* (c_2), *Spare parts,* (c_3) and *Product Planning* (c_4), where the coupling weight values are 0.37, 0.32 and 0.25 respectively. Investigating the source code shows that all three UICs are connected to *Vendor Details* by source code dependencies.

D. Expectation Maximisation Clustering

In the previous section, we discussed using a threshold value for domain-based coupling to identify highly coupled components. Previously, the threshold value has been selected manually based on the system characteristics like distribution of the coupling values, or by graph visualisation [11]. However, the manual approach is subject to human errors and not scalable for large datasets. In order to address this limitation, in this work, we use a clustering technique to identify highly coupled components automatically.

The aim of clustering is to group a given set of objects so that similar objects are grouped together and dissimilar objects are kept apart. There are many different multi-dimensional clustering techniques [12]. In this paper, we have used a statistical clustering technique called Expectation Maximization (EM) since it can automatically find the optimum number of clusters [13].

The main idea behind EM is fitting the parameters of a distribution model by using training data. The EM algorithm assigns a probability distribution to each instance of the *number of common variables* (ϑ), which indicates the probability of the instance belonging to each of the generated clusters.

In Section V, we discuss how EM clustering improves the precision of identifying dependencies.

IV. Dependency analysis

ADempiere has been designed in such a way that a developer can extend the system by touching as little code as possible. Whenever a new table is added to the database, the required Java code is automatically generated.

Most domain-level relations are managed at the data layer. As a consequence, traditional coupling metrics fail to capture the domain-level relationships between these classes. Moreover, the database contains important information about the architectural dependencies in the system. We therefore need to develop a model which is capable of expressing dependencies both at the source code and at the database layers.

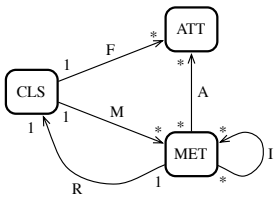
In this section, we present two general models for representing a system and its architectural relationships based on the analysis of the source code and the database. We also explain how we populated our model in the particular case of ADempiere.

A. Source Code Dependencies

The main wellspring of architectural relations is the source code. At the source code level our analysis models three key entities and their associated relations. These entities are independent of the programming language, as long as it is object-oriented:

- *Classes* are represented by a finite set CLS.
- *Attributes* are represented by a finite set ATT. The binary relation $F \subseteq CLS \times ATT$ maps attributes to the containing classes.
- *Methods* are represented by the finite set MET. The binary relation $M \subseteq CLS \times MET$ maps methods to the classes that contain them.

In addition, the relation $R \subseteq MET \times CLS$ expresses the return types of methods (NB: we allow $Void \in CLS$ to model methods that return void), $I \subseteq MET \times MET$ represents method invocations, and $A \subseteq MET \times ATT$ represents the accesses of methods to attributes. These relationships are illustrated in Figure 4.



CLS: classes, ATT: attributes, MET: methods

Fig. 4: Source code elements and relations among them.

Two classes $cls, cls' \in CLS$ can have following relationships:

$$cls'.M.R.cls \tag{1}$$

$$cls.M.I.M^{-1}.cls' \tag{2}$$

$$cls.M.A.F^{-1}.cls' \tag{3}$$

Where Equation 1 shows cls is the return type of cls', Equation 2 shows a method of cls invokes a method of cls', and Equation 3 shows a method of cls accesses an attribute of cls'.

Definition 4. *A direct relation between two classes is defined as* $D = \{M.R, M.I.M^{-1}, M.A.F^{-1}\}$. *For two classes* $cls, cls' \in CLS$, *we denote* cls *is directly dependent on* cls' *if and only if* $cls.D.cls'$

Definition 5. *For two classes* $cls, cls' \in CLS$, *we denote* cls *is indirectly dependent on* cls' *if and only if* $cls.D.D^{-1}cls'$

B. Database Relationships

A significant part of a system's business logic is incorporated in the database relationships, and these relationships complement the ones which are visible at the source code level.

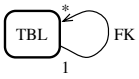


Fig. 5: Database table with the foreign key relation

The main type of entity that we model at the database level is the table, and we denote the set of all the tables with TBL. The binary relation $FK \subseteq TBL \times TBL$ maps tables on tables based on the foreign keys. Figure 5 illustrates this relationship.

As in the case of source code, we define both direct and indirect relationships in the database:

Definition 6. *Given two tables* $t, t' \in TBL$, *we say that* t *has a direct relation to* t' *if and only if* $t.FK.t'$.

Definition 7. *Given two tables* $t, t' \in TBL$, *we say that* t *has indirect relation to* t' *if and only if* $t.FK.FK^{-1}.t'$

While foreign key relations among tables are there to model a specific aspect of the domain, indirect relations between tables should suggest how different concepts are bound together.

C. Architectural dependencies

Two components are considered to be architecturally dependent either by direct or indirect dependencies between the classes behind them, or by direct or indirect relationships between the tables accessed by these classes.

Figure 6 shows the relations between the Components (C), Classes (CLS) and Tables (TBL) of ADEMPIERE. These elements are related by $DEP \subseteq C \times CLS$ which represents classes that a UIC depends on, and $REF \subseteq CLS \times TBL$ which represents tables that a class reads or writes to.

C: components, CLS: classes, TBL:tables

Fig. 6: Relationships between software elements

Definition 8. *For two components* $c, c' \in C$, *we denote* c *has an architectural dependency to* c' *if and only if they have one or more of the following relationships:*

$$c.DEP.DEP^{-1}.c' \quad (4)$$

$$c.DEP.D.DEP^{-1}.c' \quad (5)$$

$$c.DEP.D.D^{-1}.DEP^{-1}.c' \quad (6)$$

$$c.DEP.REF.REF^{-1}.DEP^{-1}.c' \quad (7)$$

$$c.DEP.REF.FK.REF^{-1}.DEP^{-1}.c' \quad (8)$$

$$c.DEP.REF.FK.FK^{-1}.REF^{-1}.DEP^{-1}.c' \quad (9)$$

This definition describes all direct and indirect dependencies through classes or tables. Equation 4 defines a connection between two components based on shared classes. Equation 5 connects two components considering direct dependencies between their shared classes. Equation 6 considers indirect dependencies between classes to connect two components. Equation 7 defines a connection between two components based on their shared database tables. Equation 8 and Equation 9 consider direct and indirect dependencies between database tables which connect two components.

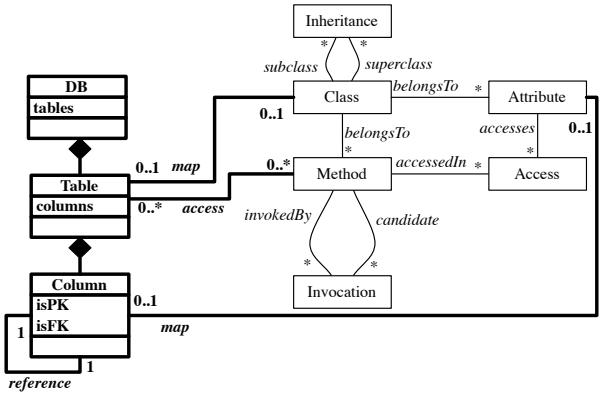


Fig. 7: Extended version of the FAMIX Meta-Model including a meta-model for relational databases

D. Dependency Analysis in ADEMPIERE

To perform the analysis on ADEMPIERE we first needed to extend the *FAMIX* [14] meta-model, which describes the static structure of object-oriented software systems, with information about database dependencies. Figure 7 shows the subset of the extended meta-model where the entities modeling relational databases are represented with bold. A class that *maps* a table is a class that represents a table at the source code level, *e.g.,* Enterprise Entity beans. The same happens to the class attributes that *map* table columns. The architecture of ADEMPIERE only contains one to one mappings, hence, we omit alternative mappings such as table-per-hierarchy.

The relation *access* represents class methods accessing database tables. The access can be made directly or through frameworks like Hibernate. The relation *reference* represents connections among table columns achieved using a foreign key constraint. These modifications to the FAMIX meta-model

have been implemented in MooseJEE [15], an extension of the Moose [16] software analysis platform.

Once the two models were populated, we needed to extract the mapping between UI components and classes and between classes and tables. This step is dependent on the technology used in the analyzed system. In the particular case of ADEMPIERE, these mappings can be found in the *Application Dictionary*, a database data structure that keeps track of all these dependencies. Once we had these mappings, we were able to compute the dependencies between the components based on architectural relationships.

V. EVALUATION

In this section, we provide empirical evidence on the usefulness of domain-based coupling in approximating architectural dependencies. We examine the following scenarios:

- **Searching for source code dependencies:** Suppose a software maintainer has no access to source code analysis tools. Using software domain information, how accurately can she predict existence of source code dependencies between UICs?
- **Searching for database relationships:** Some business constraints and relationships are defined and managed at the data layer. These relationships may or may not be visible at the source code level [3], [2], or can be difficult to analyse such as legacy databases. How accurately can a domain expert predict such relationships without analysing the database?
- **Searching for architectural dependencies:** When a domain expert requires to estimate the impact of a change to a UIC, how accurately can she identify other connected UICs by architectural dependencies?

A. Evaluation Setup

For a given UIC, $c \in C$, we test the query $AN = q(c, E)$ where the expected outcome $E \subseteq C$ is the set of UICs which have architectural dependencies to c, and the returned answer

$$AN = \{c_i | c_i \in C, \vartheta(c, c_i) > 0\}$$

is the set of UICs which are coupled with c at the domain level. We describe the outcome of such a query as follows:

TP =$|E \cap AN|$ shows the number of correctly identified dependent components.

TN =$|C \backslash \{AN \cup E\}|$ shows the number of correctly identified independent components.

FP =$|AN \backslash E|$ shows the number of incorrectly predicted dependent components.

FN =$|E \backslash AN|$, shows the number of incorrectly predicted independent components.

We use the well-known definitions of *precision* (P_q) and *recall* (R_q) to evaluate the outcomes of a given query:

$$P_q = \frac{TP_q}{TP_q + FP_q} \qquad R_q = \frac{TP_q}{TP_q + FN_q}$$

Precision and *recall* only evaluate TP. In order to describe both TP and TN, we measure *accuracy* (A_q) which is the degree of closeness of results to the preferable values where all dependent and independent components are correctly identified. *Accuracy* [17] is defined as follows:

$$A_q = \frac{TP + TN}{TP + FP + FN + TN}$$

The higher the *accuracy*, the closer the prediction outcomes to the perfect results where both FP and FN are equal to zero.

B. Macro Evaluation

In order to evaluate the results for all UICs in ADEMPIERE, we take the mean value of measurements of all queries as

$$f_M = \frac{1}{n} \sum_{i=1}^{n} f_{q_i}$$

where f is one of these measurement functions: TP, TN, FP, FN, R, P or A.

C. Likelihood

One application of domain-based coupling might be notifying software maintainers of possible dependent components when they browse a list of UICs. To assess the usefulness of such notifications, we measure the *likelihood* (L) whether at least one of the top three, five or ten returned results have architectural dependencies. More formally if $AN_{c,n}$ shows the top n results for a component c, then

$$L_n = \frac{|\{c | c \in C, AN_{c,n} \cap E_c \neq \emptyset\}|}{|\{c | c \in C, E_c \neq \emptyset\}|}$$

The *likelihood* function enables us to distinguish between the topmost results and the entire returned result set.

D. Results: Searching For Source Code Dependencies

ADEMPIERE contains 347 UICs. The source code analysis revealed $14,898$ indirect dependencies and no direct dependencies among classes behind these UICs. We compared these dependencies with the domain-based coupling graph to evaluate how accurately source code dependencies can be derived from domain information.

The results are presented in Table I. On average for a given UIC, 28 connected UICs by source code dependencies identified correctly whilst 15 UICs with source code dependencies are incorrectly described as independent components, and 78 independent UICs are falsely called to have source code dependencies. These results lead to average *recall* equal to 0.68 and average *precision* equal to 0.27.

Also the *accuracy* of the dependency prediction is equal to 0.73, implying that for more than 7 out of 10 UICs, our prediction method correctly identified if two UICs are dependent or independent at the source code level.

The likelihood of discovering source code dependencies in the top three coupled UICs is 69%, and it will increases to 78% for the top ten UICs.

Summary: *On average* 68% *of UICs connected by source code dependencies are discovered correctly, while for* 78% *of queries the top ten results contains one or more source code dependencies.*

	#Dependencies	TP_M	FN_M	TN_M	FP_M	R_M	P_M	A_M	L_3	L_5	L_{10}
Source Code Dependencies	14,898	28	15	226	78	0.68	0.27	0.73	0.69	0.74	0.78
Direct Database Relationships	8,132	19	4	237	87	0.77	0.20	0.74	0.59	0.66	0.75
Indirect Database Relationships	12,178	22	13	227	85	0.71	0.23	0.72	0.51	0.56	0.62
Architectural Dependencies	16,968	31	18	223	76	0.64	0.30	0.73	0.72	0.78	0.84

TABLE I: Prediction Results

E. Results: Searching For Database Relationships

The database analysis of ADEMPIERE showed that there are 8,132 direct and 12,178 indirect relationships among data tables behind UICs.

We queried these relationships using the domain-based coupling, and the results are presented in Table I. On average for a given UIC, 19 directly related UICs and 22 indirectly related UICs are identified correctly. The results show only 4 false negatives for direct relationships which is more than three times lower than 13 false negatives for indirect relationships. However, the number of false positives are similar: 87 and 85 for direct and indirect relationships respectively.

Comparing the results between direct and indirect relationships shows that for direct relationships the *recall* is slightly higher (0.77 vs 0.71) whilst the *precision* is slightly lower (0.2 vs 0.23). The *accuracy* values for both relationship types are more than 0.7, suggesting that for 7 in 10 UIC pairs, their relationship state is identified correctly.

In addition, validating the topmost results shows that the likelihood of database relationships in the top three results is 51% for direct and 59% for indirect relationships. Also the likelihood of indirect relationships increases to 75% for the top ten results.

Summary: *On average up to 77% of database relationships can be derived from domain information, and for 75% of queries, the top ten results contain at least one database relationship.*

F. Results: Searching For Architectural Dependencies

The analysis of the source code and the database of ADEMPIERE shows 16,968 architectural dependencies (Definition 8).

We evaluated how accurately a domain expert can predict if there is at least one architectural dependency between any given pair of UICs. The results are presented in Table I. On average for a given UIC, 31 dependent UICs, and 223 independent UICs are identified correctly using domain information. However, 18 dependent and 76 independent UICs are incorrectly placed in the opposite dependency state. These results lead to an average *recall* of 0.64 and *precision* of 0.30. The mean *accuracy* of the predictions is 0.73, suggesting that for 7 in 10 UIC pairs, their dependency state is identified correctly.

In addition, the likelihood of discovering an architecturally dependent UIC pair in the top three results is 72%. This likelihood will increase to 84% for the top ten results.

Summary: *On average 64% of architecturally dependent UICs are discovered using domain information, and the likelihood of discovering a correct architectural dependency in the top ten predictions is 84%.*

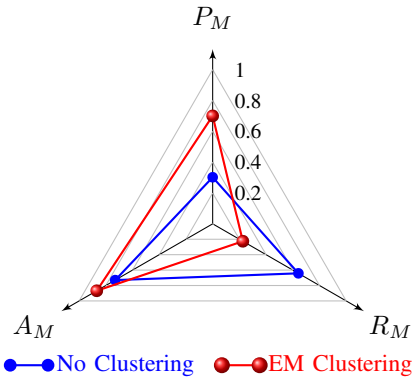


Fig. 8: Improving precision and changes in recall and accuracy

G. Improving Precision

The prediction results for architectural dependencies (Table I) show that the average precision is 0.30. In order to improve the *precision*, we utilised the expectation maximisation technique (Section III-D) to filter out weakly coupled pairs, with the assumption that UICs with strong domain-based coupling are more likely to have architectural dependencies.

	R_M	P_M	A_M
Source Code Dependencies	0.29	0.68	0.88
Direct Database Relationships	0.40	0.57	0.89
Indirect Database Relationships	0.27	0.61	0.93
Architectural Dependencies	0.23	0.70	0.87

TABLE II: Prediction Results Using EM Clustering

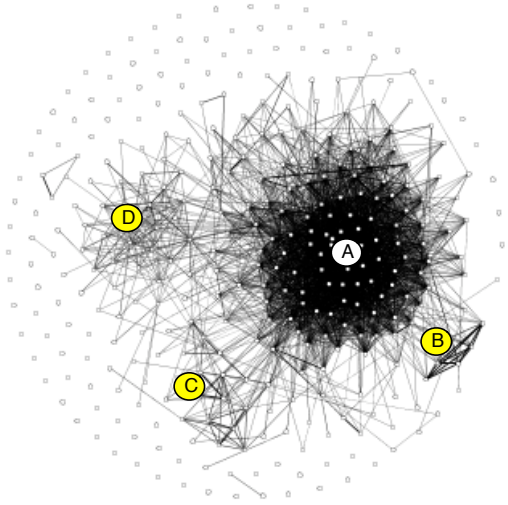
Table II shows the improved results. The mean *precision* for architectural dependencies is increase from 0.30 to 0.7, and the mean *accuracy* is increased from 0.73 to 0.87.

However, these improvements are achieved at the expense of the reduction in *recall*. As illustrated in Figure 8, while the value of *precision* is more than doubled, the value of *recall* decreased almost three times (from 0.64 to 0.23). This implies that there are a number of architectural dependencies between UICs which have no strong coupling at the domain level.

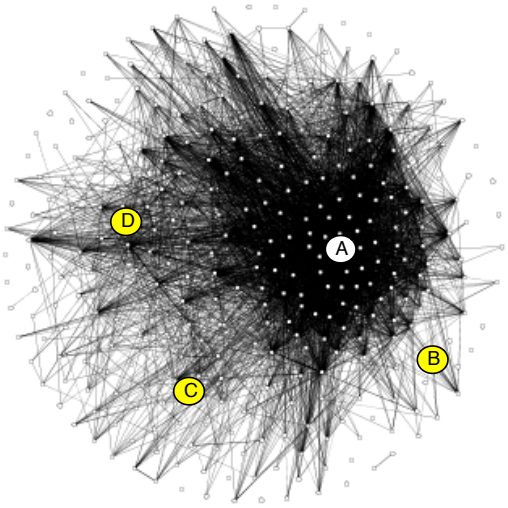
Summary: *By using expectation maximisation technique, precision can be improved up to 0.7. However, it is a trade-off between precision and recall.*

H. Visual Comparison

The domain-based coupling graph (Figure 9a) is visualised using Fruchterman and Reingold's [18] force-based graph



(a) Domain-based coupling graph - EM applied



(b) Architectural dependency graph

Legend: Nodes are the UICs of ADEMPIERE in both graphs. Left: Edges are domain-based coupling (Definition 3) which are selected by Expectation Maximisation (Section III-D). Right: Edges are architectural dependencies (Definition 8). Tags (A, B, C and D) are concentration areas.

Fig. 9: Domain-based coupling vs architectural dependencies

layout in three steps: first, the graph is created based on Definition 3; second, the exception maximisation (EM) technique (Section III-D) is applied; third, the derived graph is visualised by the force-based layout algorithm.

In order to compare the domain-based coupling graph with the architectural dependencies, the edges from Figure 9a are replaced with the architectural dependencies without changing the location of nodes. The resulting graph (Figure 9b) illustrates the distribution of the architectural dependencies in compare to the domain-based coupling.

The comparison between Figure 9a and Figure 9b shows that the most populated cluster (tagged by A) in the domain-based coupling graph has the biggest number of architectural dependencies. However, the number of architectural dependencies decreases in the clusters with poor domain-based coupling (B, C and D). In addition, there are a number of architectural dependencies where there is no domain-based coupling, illustrating that not all dependencies can be derived from the domain-based coupling graph.

I. Discussion

In this evaluation, we reported that on average 64% of architectural dependencies could be derived from domain-based coupling graph. The accuracy of the prediction is on average 0.73 while the precision is 0.30. The precision can be increased up to 0.7 using expectation maximisation technique. Trading off precision for recall would be a good approach if one would build a tool that would be used by maintainers: having too many false positives might deter the users of such a tool.

In addition, we demonstrated how domain-based coupling could be used to inform software maintainers while they browse software UICs. The results show the likelihood of discovering architectural dependencies among the top ten coupled UICs is 84%. Given that these results are obtained without looking at the source code or the database, they are quite promising. On the other hand in the current form, domain-based coupling analysis cannot completely replace the source code analysis.

VI. Threats to Validity

In this section, we discuss the threats to validity of our findings, and how we addressed them.

Threats to *external validity* are concerned with generalisation of our findings. Although we performed our evaluation on a large-scale enterprise system which is representative of the state of the art enterprise systems developed in Java, we are aware that more studies are required to be able to generalize our findings.

Threats to *construct validity* are concerned with the quality of the data we analysed, and the degree of manual analysis that was involved. The domain information typically is provided by the domain experts using a manual data collection process. To minimise the risk of human error, we extracted the relationship between domain variables and UICs from user manuals and help documents. In ADEMPIERE, this information is stored in the database. We only used manual inputs from domain experts to confirm this information and kept the manual additions and alterations to a minimum.

One other factor that could affect the validity of the results is the granularity used to look at the selected UICs. We chose windows as UICs. Each window contains multiple tabs and each tab provides one or more functions. Different results could be achieved if the evaluation was performed at the tab level, or module level.

VII. Related Work

This work is motivated by the application of dependency analysis in software maintenance and change impact analysis. In the literature, several formal models of change propagation have been introduced. Luqi [19] presented a graph model for software evolution, based on indirect relationship between components. Rajlich [4] introduced a model for change propagation

based on graph rewriting which requires an understanding of the dependencies between software elements. Arnold and Bohner [20] model change impact analysis as a cycle of revisions derived from relationships between software elements. Mirarab *et al.* [21] introduced a hybrid impact analysis method based on dependency information and co-change history. The knowledge of software dependencies is the prerequisite for these impact analysis models. The other key applications of dependency analysis are program comprehension, concept location and reverse engineering [22], [23], [24], [25].

Source code analysis [6] is an established approach for tracing software dependencies [26], [27] or evaluating the evolution of code and design [28]. One of the most well-known code analysis methods is *program slicing*, which has been exhaustively explored by many researchers, and extended to many programming paradigms [29], [30], [31], [32]. Source code analysis is further enhanced using dynamic analysis [33], [34] to capture dependencies which might not be traceable from static relationships between software elements.

Structural coupling metrics have received a lot of attention in the past years resulting in many different approaches ranging from dynamic coupling [35], [36] to evolutionary and logical coupling [37], [38]. In particular, we are interested in structural coupling metrics, as are comprehensively described by Briand *et al.* [39]. Metrics like the Coupling Between Objects (CBO) or the CBO' [40] consider the inheritance between classes to measure the coupling among software elements. Other metrics like the Response For Class (RFC) [41] and the RFC_∞ [40] consider indirect relations among classes based on a level of indirection in the invocation chain of the class methods. In our case, we measure the coupling among classes without relying on inheritance. Moreover, our definition of indirect relation is based not only on invocations between methods but also factors like the method's return type and attributes accessed by a method.

More recent research effort has concentrated on defining coupling metrics based on the concepts specific to software systems [42], [43], [44], [45]. These approaches attempt to identify and measure the relation among software entities in object-oriented software by considering latent topics from the source code. The domain-based coupling approach presented in this paper is source code independent so different from the conceptual coupling metrics.

An alternative approach to source code analysis is mining dependencies from the software repositories [46], [47], [48], [49], [24]. It can be argued that these approaches are less expensive, and require less technical expertise. However, they are not applicable where maintenance history is not accessible.

VIII. Conclusion and Future Work

In this paper, we demonstrated how domain information could be used to predict architectural dependencies, and assist software maintainers in searching for connected components at the source code or the database layers. Our proposed approach for predicting dependencies promises independence from software implementation and simplicity and usability for non-technical domain experts. Hence, it can assist managers and consultants to take decisions about software changes without the support of the developers.

The proposed dependency analysis method is based on relationships between software domain information and user interface components (UIC), modelled as a weighted graph. We demonstrated how such a model could assist predicting dependencies with a case study on a large-scale enterprise system, called ADempiere. We derived architectural dependencies as a set of source code and database dependencies, and compared them with the domain-based coupling between UICs. The results show that on average 68% of the source code and up to 77% of the database dependencies could be derived from the domain-based coupling. The accuracy of such predictions is on average more than 70%, implying that for 7 out of 10 component pairs their dependency state is identified correctly.

The results promise that domain information might be used to predict the existence of architectural dependencies, and the accuracy of these predictions could support maintenance activities such as change impact analysis. However, at the current stage, this approach cannot replace the source code analysis or the database analysis.

In this work, we have only examined the dependencies between application windows. We plan studies of finer-grained UICs (*e.g.,* Tabs) as future work. Some of the research questions to be answered are: What is the efficient granularity level? What properties of UICs affect the results (*e.g.,* size and complexity)?

The other area of future investigation is the impact of different domains on the results. ADempiere contains various modules which provide functions of different domains like ERP, CRM and Asset Management. Distinguishing between these domains and their domain-based coupling graphs might lead to better understanding of the relationships between domain-based coupling and architectural dependencies.

Acknowledgments - We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 – Sept. 2012). Part of Amir Aryani's contribution to this work has been funded by an Australian Postgraduate Award (APA). We would also want to thank Dr. Margaret Hamilton, Nicholas May and Jorge Ressia for their comments on this paper and their support on this project.

References

[1] A. Hassan and R. C. Holt, "Replaying development history to assess the effectiveness of change propagation tools," *Empirical Software Engineering*, vol. 11, pp. 335–367, 2006.

[2] Z. Yu and V. Rajlich, "Hidden dependencies in program comprehension and change propagation," in *Ninth International Workshop on Program Comprehension (IWPC'01)*, Canada, 2001, pp. 293–299.

[3] R. Vanciu and V. Rajlich, "Hidden dependencies in software systems," in *International Conference on Software Maintenance (ICSM)*, 2010.

[4] V. Rajlich, "A model for change propagation based on graph rewriting," in *IEEE International Conference on Software Maintenance (ICSM)*, 1997, pp. 84–91.

[5] A. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *International Conference on Software Maintenance*, 2004, pp. 284–293.

[6] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering (FOSE)*, may 2007, pp. 104–119.

[7] M. Lehman, "Programs life cycles and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, sept. 1980.

[8] S. Cook, R. Harrison, M. M. Lehman, and P. Wernick, "Evolution in software systems: foundations of the spe classification scheme," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, no. 1, pp. 1–35, 2006.

[9] A. Aryani, I. D. Peake, and M. Hamilton, "Domain-based change propagation analysis: An enterprise system case study," in *IEEE International Conference on Software Maintenance (ICSM)*. Romania: IEEE, September 2010.

[10] M. Lungu and M. Lanza, "Softwarenaut: Exploring hierarchical system decompositions," in *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*. Los Alamitos CA: IEEE Computer Society Press, 2006, pp. 351–354.

[11] A. Aryani, I. D. Peake, M. Hamilton, H. Schmidt, and M. Winikoff, "Change propagation analysis using domain information," in *20th Australian Software Engineering Conference (ASWEC)*. Australia: IEEE, April 2009, pp. 34–43.

[12] A. Mahmood, C. Leckie, and P. Udaya, "An efficient clustering scheme to exploit hierarchical data in network traffic analysis," *IEEE Transactions on Knowledge and Data Engineering*, vol. 20, pp. 752–767, 2008.

[13] A. Dempster, N. Laird, and D. Rubin, "Maximum likelihood from incomplete data via the em algorithm," *Journal of the Royal Statistical Society, Series B*, vol. 39, no. 1, pp. 1–38, 1977.

[14] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz, "A meta-model for language-independent refactoring," in *Proceedings of International Symposium on Principles of Software Evolution (ISPSE '00)*. IEEE Computer Society Press, 2000, pp. 157–167.

[15] F. Perin, "MooseJEE: A Moose extension to enable the assessment of JEAs," in *Proceedings of the 26th International Conference on Software Maintenance (ICSM 2010) (Tool Demonstration)*, Sep. 2010.

[16] O. Nierstrasz, S. Ducasse, and T. Gîrba, "The story of Moose: an agile reengineering environment," in *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*. New York NY: ACM Press, 2005, pp. 1–10, invited paper.

[17] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[18] T. M. J. Fruchterman and E. M. Reingold, "Graph drawing by force-directed placement," *Software: Practice and Experience*, vol. 21, no. 11, pp. 1129–1164, Nov 1991.

[19] Luqi, "A graph model for software evolution," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 917–927, Aug. 1990.

[20] S. A. Bohner and R. S. Arnold, "An introduction to software change impact analysis," in *Software Change Impact Analysis*, J. Butler and L. O'Conner, Eds. IEEE Computer Society Press, 1996, pp. 1–26.

[21] S. Mirarab, A. Hassouna, and L. Tahvildari, "Using bayesian belief networks to predict change propagation in software systems," in *IEEE International Conference on Program Comprehension (ICPC)*, 26-29 2007, pp. 177–188.

[22] B. Cleary and C. Exton, "Assisting concept location in software comprehension," in *Proceedings of 19th Annual Psychology of Programming Workshop (PPIG 07)*, Joensuu, Finland, Jul. 2007.

[23] V. Tzerpos and R. Holt, "Accd: an algorithm for comprehension-driven clustering," in *Reverse Engineering, 2000. Proceedings. Seventh Working Conference on*, 2000, pp. 258 –267.

[24] R. J. Walker, R. Holmes, I. Hedgeland, P. Kapur, and A. Smith, "A lightweight approach to technical risk estimation via probabilistic impact analysis," in *Proceedings of the international workshop on Mining software repositories*. USA: ACM, 2006, pp. 98–104.

[25] C. Marinescu, "Discovering the objectual meaning of foreign key constraints in enterprise applications," in *Working Conference on Reverse Engineering*, oct. 2007, pp. 100 –109.

[26] M. Harman, D. Binkley, K. Gallagher, N. Gold, and J. Krinke, "Dependence clusters in source code," *ACM Trans. Program. Lang. Syst.*, vol. 32, pp. 1:1–1:33, Nov. 2009.

[27] A. Cleve, J. Henrard, and J.-L. Hainaut, "Data reverse engineering using system dependency graphs," in *WCRE 06: Proceedings of the 13th Working Conference on Reverse Engineering*, Oct. 2006, pp. 157 –166.

[28] M. Hammad, M. Collard, and J. Maletic, "Automatically identifying changes that impact code-to-design traceability," in *Program Comprehension, 2009. ICPC '09. IEEE 17th International Conference on*, May 2009, pp. 20 –29.

[29] D. Binkley and M. Harman, "A survey of empirical results on program slicing," ser. Advances in Computers. Elsevier, 2004, vol. 62, pp. 105 – 178.

[30] D. Willmor, S. Embury, and J. Shao, "Program slicing in the presence of database state," in *International Conference on Software Maintenance (ICSM 2004)*, sept. 2004, pp. 448–452.

[31] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen, "A brief survey of program slicing," *SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 1–36, March 2005.

[32] J. Silva, "A vocabulary of program-slicing based techniques," *ACM Computing Surveys (To appear)*, 2011.

[33] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on*, Mar. 2005, pp. 124 – 133.

[34] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 684–702, 2009.

[35] E. Arisholm, L. Briand, and A. Foyen, "Dynamic coupling measurement for object-oriented software," *Software Engineering, IEEE Transactions on*, vol. 30, no. 8, pp. 491–506, Aug. 2004.

[36] Y. Hassoun, R. Johnson, and S. Counsell, "A dynamic runtime coupling metric for meta-level architectures," *Software Maintenance and Reengineering, European Conference on*, vol. 0, p. 339, 2004.

[37] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, "Mining version histories to guide software changes," in *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, May 2004, pp. 563–572.

[38] H. Gall, M. Jazayeri, and J. Krajewski, "CVS release history data for detecting logical couplings," in *International Workshop on Principles of Software Evolution (IWPSE 2003)*. Los Alamitos CA: IEEE Computer Society Press, 2003, pp. 13–23.

[39] L. C. Briand, J. W. Daly, and J. K. Wüst, "A Unified Framework for Coupling Measurement in Object-Oriented Systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1, pp. 91–121, 1999.

[40] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[41] ——, "Towards a metrics suite for object oriented design," in *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, vol. 26, Nov. 1991, pp. 197–211.

[42] D. Poshyvanyk, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *Proceedings of the 2nd international conference on program comprehension (ICPC)*. ACM Press, 2006.

[43] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 469–478.

[44] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, Feb. 2009.

[45] M. Gethers and D. Poshyvanyk, "Using relational topic models to capture coupling among classes in object-oriented software systems," in *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ser. ICSM '10. Washington, DC, USA: IEEE Computer Society, Sep. 2010, pp. 1–10.

[46] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, pp. 574–586, sept. 2004.

[47] D. G. A. Hindle and N. Jordan, "Visualizing the evolution of software using softchange," in *Proceedings of the 16th International Conference on Software Engineering & Knowledge Engineering (SEKE 2004)*. New York NY: ACM Press, 2004, pp. 336–341.

[48] M. D'Ambros, M. Lanza, and R. Robbes, "On the relationship between change coupling and software defects," in *Proceedings of WCRE 2009 (16th IEEE Working Conference on Reverse Engineering)*. IEEE CS Press, 2009, pp. 135–144.

[49] H. Kagdi, J. Maletic, and B. Sharif, "Mining software repositories for traceability links," in *Program Comprehension, 2007. ICPC '07. 15th IEEE International Conference on*, Jun. 2007, pp. 145 –154.