RWTH Aachen University
Software Engineering Group

Feature Location Techniques

Seminar Paper

presented by

Bergerbusch, Timo

1st Examiner: Prof. Dr. B. Rumpe

2nd Examiner: Dipl.-Inform. C. Schulze

Advisor: Dipl.-Inform. C. Schulze

The present work was submitted to the Chair of Software Engineering

Aachen, December 30, 2016

Eidesstattliche Versicherung

______________________________ ______________________________

Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____

____

____

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

______________________________ ______________________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

______________________________ ______________________________

Ort, Datum Unterschrift

Abstract

Locating software artifacts that implement a specific program functionality, whether it's functional or non-functional, are called a feature. Detecting features in a program is the main goal of Feature Location Techniques(FLT). It assists software developers during the maintenance and refactoring of the code. But also the software product line engineering(SPLE), which specifies, designed and implements different products by managing features, uses these techniques to create a product without copying code unstructured but by systematic reuse of the artifacts the FLT's locate [PBvDL05].

Therefore my seminar paper deals with different feature location techniques from very fundamental methods to some of today's newest research fields. In this paper I introduce a real use case example, to show the real utility of the techniques, of the Freemind mind mapping software [www16b].

In this paper we continue to get to know to the basics of FLT's to understand how they are able to define artifacts, the classification of FLT's considering their approach strategy, explaining different techniques of different previously mentioned classes, regarding their strengths and weaknesses, on a realistic use case of a real software segment. At the end will be an outlook to leveraging SPLE architectures and possible improvements of the existing techniques[ZZL+06].

Contents

Chapter 1

Introduction

A feature location technique is aiming at the locating of software artifacts as a realization of a system requirement. It could be *functional*, like the ability of doing a special kind of computation for example counting elements, or it could be *non-functional* like doing a functional requirement in a given time. To be able to understand what a feature location technique in detail should be it is necessary to have a basic knowledge about two aspects of modern software engineering. Without either one of the following two underling definitions it's is not clearly definable what a feature location technique should be capable of and there is also no way to rate if a technique is efficient and correct.

On the one hand there a the features. As defined by the Institute of Electrical and electronics Engineers (IEEE) a feature is defined as 'A distinguishing characteristic of a software item (e.g., performance, portability, or functionality)'.[Wik04a] For us simplified a feature is a software artifact implementing a given requirement. Features are often described by the definition of *Rajlich and Chen*, who describe a feature or concept as a triple of *name*, the name of the feature, *intension*, a short precise description, and *extension*, the artifacts implementing the feature.[KC00]

The the other hand there is the software product line engineering (SPLE). A product line is a variety of products, which in our case are software products, which 'share a common, managed set of features satisfying the specific needs a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.' [www16a]. A good example are the products of SAP like the *Business One*, *Business All-In-One* and *Business ByDesign*, which share a basic set of functionality, build up on each other and often are modified to fit the needs of a customer. The SPLE promotes *systematic* software reuse being base on the knowledge about the set of available features, relationships among the features and the relationship between features and their artifacts. The most essential step for unfolding the complexity of existing implementations to be able to transform it into a SPLE includes the identifying of the implemented features and their corresponding artifacts.

This, the locating and defining of a feature, is the problem a feature location technique should solve, so that developers of software product lines are supported during the maintenance and the aspect-/feature oriented refactoring of software.

Chapter 2

Freemind Example

The example used for this paper is the *automatic save file* feature of Freemind. Freemind is an open source mind-mapping tool. The *automatic save file* feature is a good example, because of it's name. Parts of the name are also mentioned in other features, which makes it slightly more difficult to only locate this specific feature. A representiv callgraph of the important parts is shown in Fig 2.1.

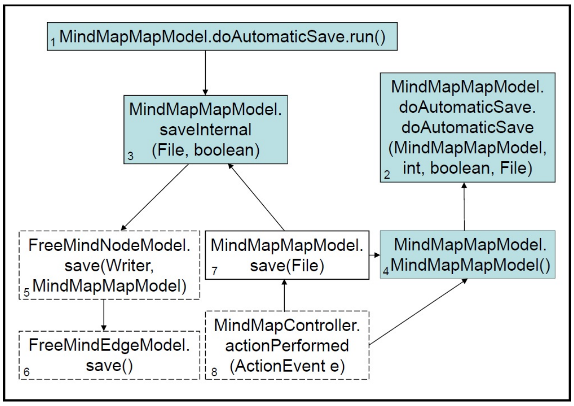

Figure 2.1: The Freemind callgraph [www16b] [RC13]

As you can see here only the relevant constructors and methodsare shown and numbered with indices from 1 to 8. We reference them by using the number sign # and then the corresponding number. Also the feature of the regarded function are highlighted with a blue background color. These are the methods which should be located if the *automatic save file* function is the wanted feature. Note that the all the methods of different classes can in addition call other methods and constructors, which are irrelevant to the feature. So as we can see the feature is mainly implemented by two methods of a subclass of *MindMapMapModel* so called *doAutomaticSave*:

- the constructor, which is #2. This constructor gets a few parameters to configurate the *doAutomaticSave*-function and registers the class in the sheduling queue, so that it gets called.
- the *run()*-function #1. This Methods gets called after the class is registered in the sheduling queue and everytime a special event is occurs. That can be different, like a period of time to shedule an automatic save or a preset number of actions within the main-programm. It calles the *saveInternal*-method to do actual saveoperation.

Regarding the previously mentioned definition of a feature by Rajlich and Chen 1, we can now define the regarded feature as the following:

name: *automatic save file*
intesion: saves a file automaticly afer the occuring of an event
extension: #1, #2, #3 and #4

The methods #5 to #8 aren't in the extension of the *automaticSaveFile* feature. Mainly # 5 and #6 are called by methods of the *automaticSaveFile* feature, but aren't relevant to the specifies of this function. #7 and #8 in fact call #3 and #4, but they handle a user triggered save-event, which obviously isn't important to the *automaticSaveFile* feature.

While all feature location techniques try to achive the same goal, which is the locating the feature extension to a given feature intension, they differate in the underlying base of assumptions they make to be able to get the tracebility. It will be declared more specific in chapter 4.

Chapter 3

Basic Underlying Techniques

To understand how feature location techniques work it is important to understand a few basic techniques that are commonly used to create or improve feature location. All the basic techniques will be exemplary executed on the previously introduced Freemind-example in chapter 2.

3.1 Formal Concept Analysis (FCA)

Formal Concept Analysis (short: *FCA*) is a predominantly mathematical approach to identify groups of classes and methods compared by the sharing of attributes. Therefor the *FCA* regards the binary relation between all objects and attributes and therefor can also provide a model to analyze hierarchy, because hierarchy structures often have similar relations.

The *FCA*'s goal is to define so called *concepts*. A *concept* is a tuple of extension, the objects that belong to a concept, and intension, all the attributes that every object of the extension has. In order to be able to derive such a *concept* the *FCA* creates an incidence table. The table can be derived in 3 steps as seen in Fig. 3.1:

1. declaring every word in the objects and methods as w_i to a new i if the word isn't already defined
2. decapitalizing every w_i
3. creating the table with every decapitalize word as a row and every σ as a column. The cells $c_i j$ are checked if σ contains the word w_i

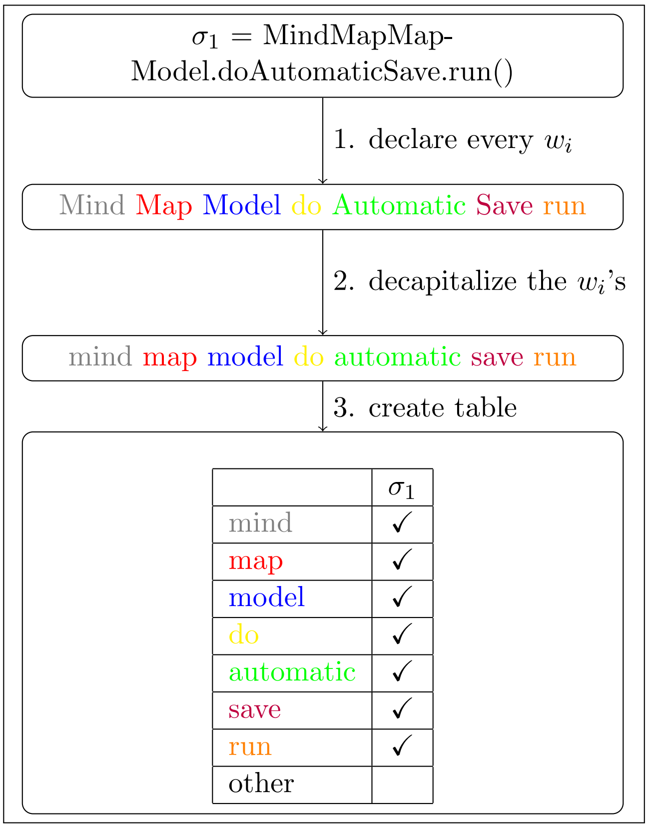

Figure 3.1: #1 of the Freemind Example as example

objects ↓	σ_1 ↓	σ_2 ↓	σ_3 ↓	σ_4 ↓	σ_5 ↓	σ_6 ↓	σ_7 ↓	σ_8 ↓
action								✓
automatic	✓	✓						
controller								✓
do	✓	✓						
file								
free					✓	✓		
internal			✓					
map	✓	✓	✓	✓			✓	✓
mind	✓	✓	✓	✓	✓	✓	✓	✓
model	✓	✓	✓	✓	✓	✓	✓	✓
node						✓	✓	
performed								✓
run	✓							
save	✓	✓	✓		✓	✓	✓	

Figure 3.2: The complete incidence table of the Freemind Example

Keeping the methods numbers as we did we get Figure 3.2 as a result. Mathematically it leads us to defining O as a set of objects, A as a set of attributes and R as the set of relations $r = (o, a) \quad o \in O, a \in A$ as derivable of the table. Also we define that
$\sigma(O) = \{a \in A | (o, a) \in R, \forall o \in O\}$ "all attributes that every $o \in O$ has"
$\rho(A) = \{o \in O | (o, a) \in R, \forall a \in A\}$ "all objects that every $a \in A$ has"
So a concept can be declared as a tuple $c = (O, A)$ so that $A = \rho(0)$ and $O = \sigma(A)$. So O is the extension and A is the intension.

From there it is very easy to see, that the set of all concepts C is a partial order (*superconcept* - *subconcept*) defined as:

$$(O_1, A_1) \leq (O_2, A_2) \quad \Leftrightarrow \quad O_1 \subset O_2 \; or \; A_1 \subset A_2.$$

Which leads to the definition that $C, \leq$ form a concept lattice and in our example it's a taxonomy of name tokens.

3.2 Latent Semantic Indexing (LSI)

$$
\begin{array}{l}
\text{Documents/} \\
\text{Terms} \\
\downarrow \\
\texttt{action} \\
\texttt{automatic} \\
\texttt{controller} \\
\texttt{do} \\
\texttt{file} \\
\texttt{free} \\
\texttt{internal} \\
\texttt{map} \\
\texttt{mind} \\
\texttt{model} \\
\texttt{node} \\
\texttt{performed} \\
\texttt{run} \\
\texttt{save}
\end{array}
\quad A = \begin{pmatrix}
d_1 & d_2 & d_3 & d_4 & d_5 & d_6 & d_7 & d_8 \\
\downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 4 & 0 & 0 & 2 & 1 \\
1 & 1 & 1 & 2 & 1 & 1 & 1 & 1 \\
1 & 1 & 1 & 2 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 2 & 1 & 0 & 1 & 1 & 1 & 0
\end{pmatrix}
\quad q = \begin{pmatrix}
q \\ \downarrow \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1
\end{pmatrix}
$$

Figure 3.3: The term-document matrix

The *Latent Semantic Indexing*(short: *LSI*) is an automatic statistical technique. It derives to a given document a vector representation of the query and the corpus by creating a term-document matrix of co-occurring terms. A term t_i is a word, as a tokenized and decapitalized word of the methods ordered alphabetically and is represented in a row of the matrix. A document d_j, which are in our example the different method- and class names, are represented as the columns of the matrix. So the matrix, shown in Figure **??**, looks very similar to the table of FCA (Fig. 3.2) with the difference of an unsigned integer value v_ij, representing how often a document $d_j = MindMapMapModel.doAutomaticSave.run()$ contains token t_i, i.e. d_1 contains the token $t_7 = map$ twice, but the token $t_2 = automatic$ only once and doesn't contain $t_1 = action$ at all. Also a query q is given, which has a *1* at the terms *automatic*, *save* and *file* representing the feature that should be analyzed.

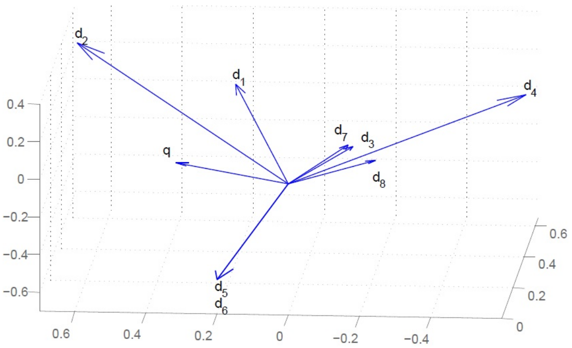

Figure 3.4: The vector representation of the documents d_j and the query q from the Freemind Example 2

By normalizing and decomposing using a singular value decomposition the documents can be put into vector representation so that every document has a vector representing their equality to the query q. Taking the $cosine()$ of the query q and a document d_j it is possible to measure the similarity. If a document and a query are equal the spreading angle would be 0 and therefor the best possible similarity is given by $cosine(0) = 1$. Also the worst possible angle is 180, which is equal to document "pointing in the opposite direction", and therefor the worst similarity is given by $cosine(180) = -1$. The common interpretation of the values, regarding that D is the set of all documents, is, that the set $\{d_i \in D | cosine(d_i, q) \geq 0\} \subseteq D$ are considered to be a related to the query of interest, hence every other document is not. It's simple to see that a document is more similar if it points in the same general direction as the query, because of the shared terms. In the Freemind Example the document $d_2 = MindMapMapModel.doAutomaticSave.doAutomaticSave$ is the most similar to the query $q = automaticSaveFile$, while $d_8 = MindMapController.actionPerformed$ is the least similar.

d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
0.6319	0.8897	-0.2034	-0.5491	0.2099	0.2099	-0.1739	-0.6852

Like previously mentioned d_2 is the most similar to q, because of the "pointing in the same general direction", which it now proven by having the highest value $cosine(d_2, q) = 0.8897 = max\{cosine(d_i, q) | d_i \in D\}$ and also d_8 is the least similar with a value $cosine(d_8, q) = -0.6852 = min\{cosine(d_i, q) | d_i \in D\}$.

3.3 Term Frequency - Inverse Document Frequency (tf-idf)

The *term frequency - inverse document frequency* technique is a statistical technique to derive a feature to a given intension. It measures the importance of a term or multiple terms to documents by its frequency of appearing. The terms are terms of the intension of the feature that is wanted to be analyzed. In a simple way it can be described as: "the more frequent a term occurs in the document, the more relevant the document is to the term".
This is mathematically described as the $documentfrequencytf = (t, d)$, counting how often the term t is contained in the document d. In our example the term $t_2 = save$ appears in d_3 once and the term $t_1 = automatic$ doesn't appear at all so: $tf(t_2, d_3) = 1$ and $tf(t_1, d_3) = 0$.
Doing that for the terms $t_1 = automatic, t_2 = save$ and $t_3 = file$ and the documents d_1 to d_8 we get the matrix shown in Fig. **??**. The main problem of this technique is, that uninformative terms appearing within a document-set, often referred as *corpus* and shortened by D, maybe even multiple times can distract from terms, which are mentioned

less frequent but are more relevant. To compensate that, the technique relativizes by calculating how many documents contain the term and normalizing it. If it's a commonly used term shared by many documents this term can't be taken as a measurement to differentiate between documents. Or colloquially "the more documents include a term, the less this term discriminates between documents".
So the so-called *inverse document frequency (idf(t))* is calculated as

$$idf(t) = log((|D|)/|\{d \in D|t \in d\}|)$$

with D still being the set of documents. And the final *term frequency - inverse document frequency* is the multiplication of both scores, so:

$$tf\text{-}idf(t,d) = tf(t,d) * idf(t)$$

Regarding our example we can compute the *idf* of our terms:

$$\begin{aligned}&t_1 = log(automatic/idf(t_1)) = log(8/2)\\&t_2 = log(save/idf(t_2)) = log(8/6)\\&t_3 = idf(t_3) = 0\end{aligned}$$

Like in the example if the focus isn't on one term but on a set of terms the *tf-idf(t,d)* values to a document d are added up. So finally the matrix can be derived as it is shown in Table **??**.

	d_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8
$t_1 = automatic$	0.6021	1.2041	0	0	0	0	0	0
$t_2 = save$	0.1249	0.2499	0.1249	0	0.1249	0.1249	0.1249	0
$t_3 = file$	0	0	0	0	0	0	0	0
$\sum_{i=1}^{3} tf\text{-}idf(t_i, d_j)$	0.727	1.454	0.1249	0	0.1249	0.1249	0.1249	0

Table 3.1: Term Frequency - Inverse Document Frequency

3.4 Hyper Link Induced Topic Search (HITS)

The *Hyper Link Induced Topic Search* (short: *HITS*) is a page ranking algorithm for web mining[1], which is the counterpart of the famous *Google Page Rank*-algorithm and is currently used by the *Ask Search Engine* [Wik04b]. Its basically used to get websites that correspond best to a given input, like every search engine. The *HITS*-algorithm distinguishes between two forms of web pages, which aren't necessarily disjoint:

1. hub
   A hub is a web page pointing towards other web pages , which can be a hub, an authority or even both. A pragmatism is to say: "a good hub points to many authorities."

[1] web mining is the analysis step of the knowledge discovery in databases process within the World Wide Web CITE

2. authority
   An authority is a web page, that other pages point to in order to cite or prove. The rule of thumb is: "a good authority is pointed by many good hubs."

Regarding the definition of hubs and authorities it seems quite natural to define a directed graph $G = (V, E)$ with vertices $V =$ web pages and edges $E = \{(v, w)|v \text{ refers to } w\}$ (also called *links*) . A hubscore is the number of authorities the hub refers to. An authorityscore is a number of good links that refer towards this authority. Both are initialized with 1. Keeping the graph G in mind the hub- and authority scores can be defined as the following.

$$\begin{array}{rl} \text{authority score of page } p & A_p = \{\sum_{\{q|(q,p)\in E\}} H_q\} \\ \text{hub score of page } p & H_p = \{\sum_{\{q|(p,q)\in E\}} A_q\} \end{array}$$

Given the two values the graph can be rewritten as $G' = (V', E')$ with $V' = \{(p, H_p, A_p)|\forall p \in V\}$ and $E' = E$. By iterating over the graph the values of H_p and A_p are calculated for every page p. In order to don't just count up to infinity the values have to be normalized like the following:

$$\begin{array}{rl} \text{normalizing the authority score of page } p & A_p = A_p/\sqrt{\sum_{(q,H_q,A_q)\in V'} A_q^2} \\ \text{normalizing the hub score of page } p & H_p = H_p/\sqrt{\sum_{\{(q,H_q,A_q)\}\in V'} H_q^2} \end{array}$$

The normalized values satisfy the condition, that $\sum_{\{(p,H_p,A_p)\}\in V'} H_p^2 = \sum_{\{(p,H_p,A_p)\}\in V'} A_p^2 = 1$.

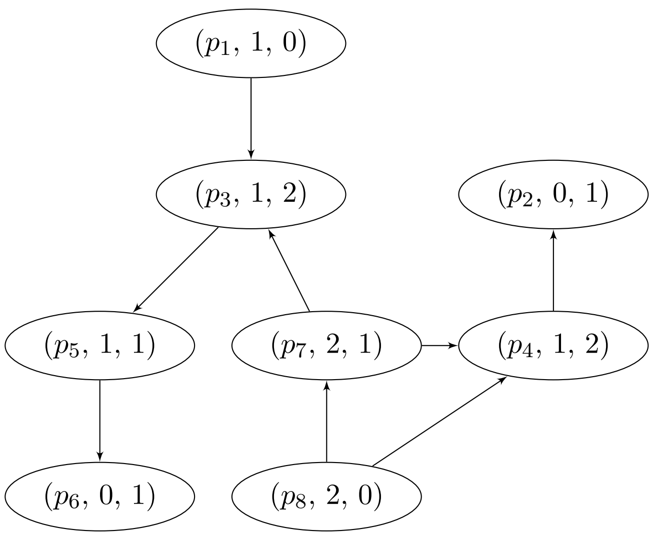

Figure 3.5: The graph G

Applying the *HITS*-algorithm to program code hubs can be colloquially described as methods, that call many other methods, and authority's can be described as methods, that implement a function.
In the Freemind Example the first graph will look very similar to the class diagram, as it is shown in Fig 3.5. The class #i will refer to page p_i. After transferring it into the graph of the form of G' and after the first iteration the graph looks like Fig. 3.5.
Including the normalization the graph G' looks like Fig. 3.6. The normalization was done by calculating for every H_p and A_p of a page p as:

$$\begin{array}{rcl} H_p & = & H_p/\sqrt{1^2+0^2+1^2+1^2+1^2+2^2+0^2+2^2} = H_p/\sqrt{12} \text{ and} \\ A_p & = & A_p/\sqrt{0^2+1^2+2^2+2^2+1^2+1^2+1^2+0^2} = A_p/\sqrt{12}. \end{array}$$

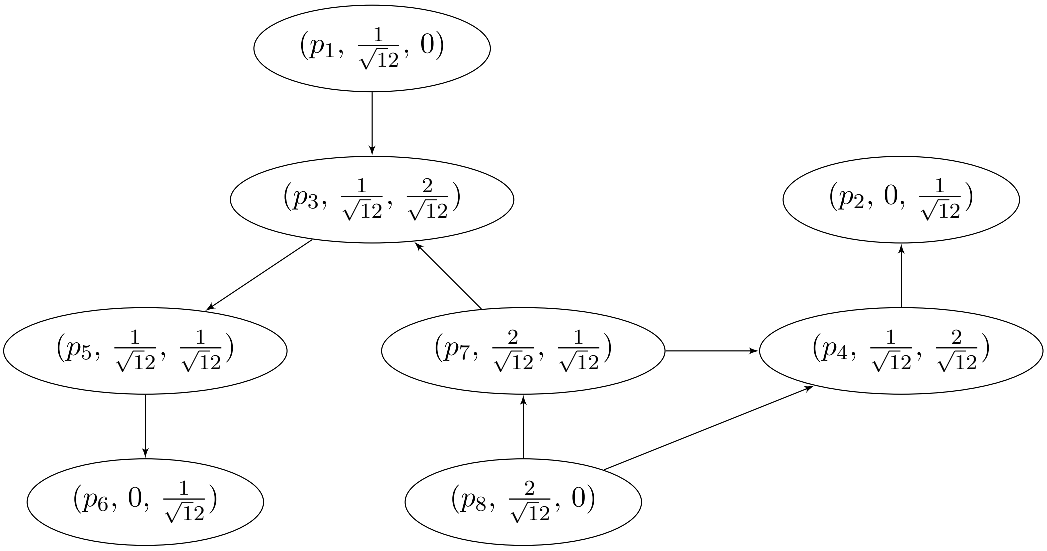

Figure 3.6: G' after the first iteration with normalizing the scores

Chapter 4

Classification and Methodology

The classification of feature location techniques is very important, because of the different special demands of some classes of techniques and their assumptions they have towards special parts of the system or code. The first big distinction is the difference of dynamic and static techniques.

dynamic:
Dynamic approaches collect information about the program at runtime. They do so by using program dependency analysis, information retrieval, latent semantic indexing (3.2 or the term frequency - inverse document frequency **??**, which only consider the methods and classes, which are involved during the current execution of the program. This is a big advantage, because by knowing that looking for a feature knowing roughly the part of the program where it could be used the user is able to steer the program into the direction. In our example the *automaticSaveFile*-function wouldn't be in the main-menu or the settings, but is more likely to be involved if the user creates a mind map and waits till the *automaticSaveFile*-function is triggered. But that advantage has also it's flipside. By only analyzing the involved parts of the program the whole information retrieval is based on the input the program gets and has to generalize from that, which may not be the right thing to do. Also collecting information on test-cases can only derive *functional* requirements, but isn't able to derive non-functional requirements. In general the dynamic approaches under approximate.
static:
Static approaches don't need the program to be executed. They collect information directly out of the source, which has one big disadvantage. A static approach would look at every single part of the code to derive information about the feature, the user want's to locate, which can be very costly. Imagine a program which is very very complex and big and the user wants to locate a very small special feature which is contained in very little of the code for example in only 0.01% . The static approach will look through the whole 100% of code, of which 99.99% are not related to the feature. The big advantage of the static approach is that the information it reveals are safe, which means it doesn't has to generalize out of a case but can validate on the whole information. This results in the ability to derive functional and non-functional requirements. This whole information can on the other side lead again to problems. Knowing every little detail can lead to situations in which the information's are undecidable in the matter of affiliation to the feature. So the technique has to approximate a solution, which may be to imprecise. In general the static approaches over approximate.

The techniques can also be splitted within the *static/dynamic*-groups due to the form of output the methods give.

plain:
The plain-output techniques present an unsorted list of artifacts, which are considered by the technique to be relevant to the feature. They leave the interpreting of the output to the user.
guided:
The guided-output techniques present the collected artifacts in a special arrangement to build an interpretation, like ordering the artifacts based on the relevance it is considered to have. Also often a so called *Program Dependency Graph* is given to not only show relevant artifacts, but also give a dependency of these artifacts. This topic is further explained in "Case Study of Feature Location Using Dependence Graph" by K. Chen and V. Rajlich [CR00].

Also the different techniques make assumptions. For example the in chapter 3.2 mentioned *Latent Semantic Indexing* does the assumption that the classes and methods of the code are named like the function they implement. The same technique can be useful on one code fragment, which fits the assumptions, but completely useless on an other one, which doesn't fulfill the assumptions. [RC13] [DRGP13]

An other file in which the different methods can be distinguished is the amount of user interaction within the process of locating a feature. While some methods can derive features and corresponding artifacts with almost only the name of the wanted feature, others need very much interaction to derive these artifacts. The result depends on the underlying code, the feature and also on the assumptions they make towards the code.

Chapter 5

Feature Location Techniques

In this chapter we want to look at four different feature location techniques in detail. We choose two static and two dynamic techniques with each one technique giving plain and one giving guided output. The techniques presented in the following can be classified by the characteristics of chapter4:

	technique	output	underlying technology	input	result	user
static	Find-concept	plain	PDA, NLP	query query	AOIG documents	++
static	SNIAFL	plain	tf-idf, LSI, PDA	set of query's	BRCG	-/+
static	Dora	guided	PDA, tf-idf	method, depth query	call graph documents	+
dynamic	Software Reconnaissance	plain	FCA, PDA	set of scenarios, query	executable	+++
dynamic	Revelle	guided	trace analysis LSI, HITS	scenario and query	executable, documents	+

Table 5.1: The techniques discussed further on in this paper

5.1 Static - Plain

As an example of a static technique with plain output the *Find-concept (short FC)* of David Shepherd, Emily Hill, K. Vijay-Shanker and Lori Pollock of the University of Delaware and also Martin P. Robillard of the McGill University in Canada is a reasonable choice The technique makes, as previously mentioned in Chapter 4, some assumptions to the underlying code. To apply *FC* the code has to be object-oriented, the comments and identifiers, which are objects and methods, have to be named in a way so that the technique can retrieve domain knowledge. Also it makes the premise that verbs correspond to methods and nouns refer to objects. Also FC defines so called *direct objects*, which are objects corresponding to a verb. In our example the verb *save* corresponds to $MindMapMapModel$, $MindMapNodeModel$ and $MindMapEdgeModel$, which are therefore the direct objects of *save*.

The input to the FC is given by the user as a query of description phrases of the feature of interest and after that decomposed into a set of *verb-DO* pairs. In order to improve the result the technique collects related words,like synonyms or verbs in different time forms, and also regards words, which are often mentioned in the context of words from the query. These collected words then get ranked by their similarity to the query words with for example LSI 3.2, calculating with a variable weight for the synonyms, and the ten most analogous are presented to the user to augment the query with these terms and program methods already matching to the current query.

The important aspect the user wants to retrieve are the *verb-DO* pairs matching the query. To be able to derive the matching pairs the FC builds an *action-oriented identifier graph model (AOIG)*. The *AOIG* contains four kinds of nodes and 2 types of edges:

verb nodes:	a node for each specific verb/action
direct object (DO) nodes:	a node for each direct object
verb-DO nodes:	a node for every *verb-Do* pair. (A *DO* can be in multiple *verb-DO* nodes)
use nodes:	a node for each incidence of a *verb-DO* pair in comments or the source code
pairing edges:	connecting every verb and DO to the *verb-DO nodes* containing them
use edges:	connecting each *verb-DO node* to every corresponding *use node*.

After several steps of improving the query the final query traverses through the *AOIG* and filters every *verb-DO* pair containing words of the query, extracting all methods using the filtered pairs and apply *Program Dependency Analysis (PDA)* on it to reveal call relations within the extraction.

Finally the *FC* is able to generate the result graph with methods matching the query as nodes and structural relations between the methods computed by the *PDA*. [SFH+07]
Due to the overhead of computing the *verb-DO* pairs out of the query and the step by step improvement of the input the user interaction in Table 5.1 is rated with "++".

5.2 Static - Guided

The technique presented by *Emily Hill, Lori Pollock* and *K. Vijay-Shanker*, professors of the *University of Delaware* in *Computer and Information Science*, is named *Dora the Program Explorer* (short: *Dora*)[1]. Dora also uses a call graph $G = (V, E)$ to derive dependency, like the *Find-concept* in section 5.1, but combines it with the *tf-idf* ranking method explained in section 3.3 with the methods as nodes $n \in V$, it's body as the documents $d(n)$ and edges $e = (n, m) \in E$ if n calls m.
As an input the user has to yield an initial query, a so called *seed method*$n_0 \in V$ the examination should start from, and a depth defining a graph-neighbourhood, which should be included in the search(i.e. a maximal distance).
Given the input Dora proceeds by traversing through the call graph G calculating how suitable the document $d(n)$ of the current node n is by combining the succeeding three values:

[1]Dora comes from exploradora, the Spanish word for a female explorer[HPVS07]. Also the name chosen in account of the children's series "*Dora the Explorer*"

1. the *tf-idf* score of the identifiers within the method name (n)
2. the *tf-idf* score of the identifiers within the method body ($d(n)$)
3. a binary value to indicate if the method belongs to a library or is part of the user-made code

Dora can be parametrized by the weight of these three components, for example the method name(1) should be more important than the method body(2) and if the method is out of a library it shouldn't be considered, which leads to the following formulae:

$$s(n) = (1 - b) * [\tfrac{2}{3} \mathit{tf\text{-}idf}(n) + \tfrac{1}{3} \mathit{tf\text{-}idf}(d(n))]$$

where b defines if n belongs to a library($b = 1$) or n is user-made($b = 0$). There are two more adjustable values: the relevance threshold(rt) and exploration threshold(et). The relevance threshold determine whether a node is relevant or not can be parametrized by giving a value $rt, et \in [0, 1]$ and typically $et < rt$, that given a node n:

$$\begin{array}{rcl} rt <= s(n) & \rightarrow & \text{the node is relevant} \\ et <= s(n) < rt & \rightarrow & \text{the node isn't relevant, but maybe it's neighbours} \\ s(n) < et & \rightarrow \text{the node can be neglected} & \end{array}$$

In the case of 1 and 2 Dora traverses to the neighbourhood of the node, if it doesn't harm the the initial depth, and otherwise discards the node. So in finite steps of traversing through the call graph Dora has reached a point, where no additional elements need to be explored.
The result Dora computes is a subgraph $G' = (V', E')$ of the call graph, where $V' = \{n \in V | et <= s(n)\}$, $E' = \{(n, m) \in E | n, m \in V'\}$ and a function

$$f : n \in V' \rightarrow \{0, 1\}, n \rightarrow \left\{ \begin{array}{ll} 1, & s(n) >= rt \\ 0, & else \end{array} \right. .$$

This function can be described as a colouring of every *relevant* node. The final output is the coloured sub-call-graph G'.

In the *Freemind Example* of chapter 2 the result can look different, by changing the parameters like the *seed method*, the *depth* or the *threshold values*. Simplifying the method in the fact of disregarding the method body's and by knowing that every method called in the diagram **??** is user made, the scores are equal to their score in chapter **??**.
The threshold are choosen like the following: So the final graph Dora computes looks

$rt = 0.5$	methods with a score of 0.5 or higher are considered relevant
$et = 0.1$	methods with a score of 0.3 or higher should be explored further

like the graphfigure 5.1. The green nodes are relevant to the feature, the grey nodes are explored but not relevant. The red node(#2) is highly relevant to the feature with a *tf-idf score* of 1.454, but isn't explored due to the *depth* of 3. In modern cases of application the *threshold*-values are chosen by a heuristic of other cases and general knowledge of the underlying program. Including the *methods body* (2) and the binary value of the formulae

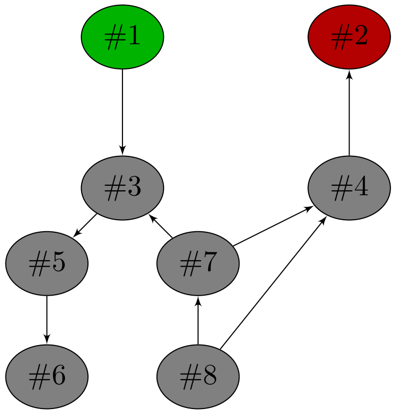

Figure 5.1: The result graph of *Dora*, with #1 as *seed method*, *depth*=3, rt=0.5 and et=0.1

the result can be refined by slightly changing the query or the *threshold's*.
Dora only needs a query and a *depth* to compute a result, which takes to further interaction, which is marked within the Table 5.1 with only one "+".

5.3 Dynamic - Plain

One of the most important dynamic plain approaches is the very first one of Norman Wilde and Michael Scully known under the term *Software Reconnaissance*. *Software Reconnaissance* tries to define a feature f by getting two sets of scenarios S_f and $\overline{S_f}$ as an input and distinguishing between scenarios that invoke the feature of interest S and scenarios that don't $\overline{S_f}$.
Regarding the execution traces *Software Reconnaissance* categorizes methods/lines of code M[2] into three groups: The result are the first 3 lists for every feature f that is set in the

1. potentially involved
   $I_1 = \{m \in M | \exists s \in S_f \text{ s.t. } s \text{ executes } m\}$
   get executed by at least one scenario of S_f
2. indispensably involved
   $I_2 = \{m \in M | \forall s \in S_f : s \text{ executes } m\}$
   get executed by every scenario of S_f
3. uniquely involved
   $I_3 = \{m \in M | m \in I_1 \text{ and } \forall s \in \overline{S_f} : s \text{ does not execute } m\}$
   executed by at least one scenario of S_f and by no scenario of any other feature
4. common components
   $C = \{m \in M | \forall s \in S_f \cup \overline{S_f} : s \text{ executes } m\}$
   used by every scenario (for example a *main*-method)

query and once the list of all *common* components. Different versions of this technique state, that $I_2 \cap C = \emptyset$.[WS95]

In our example regarding the two features of $f_1 = automaticSaveFile$ and $f_2 = manualSaveFile$ the execution traces are quite similar owed to the fact, that the *automaticSaveFile*-feature

[2] the degree of fineness is chosen by the user

is just a not user triggered *internalSave*. Keeping in mind the callgraph(Figure **??**) methods #3, #5 and #6 will be considered as *common* components. Method #1 will be considered *uniquely involved* to the feature f_1.

This technique already requires voluminous overhead, because of the two sets of scenarios S_f and $\overline{S_f}$ for every feature.

5.4 Dynamic - Guided

The dynamic guided technique by Meghan Revelle, Bogdan Dit and Denys Poshyvanyk, which are professors at the College of William and Mary in Virginia, is based on a chain of other techniques here and further named as the main author:

$$Revelle \rightarrow Liu \rightarrow Poshyvanyk \rightarrow Marcus$$

The very base technique by Marcus is to take given an input query and convert it into a document in vector space using the in section 3.2 mentioned *LSI*. The technique then separates different software elements, for example methods, and creates separate documents using the identifiers and also converting them into vector space. The identifiers are often separated using typical code style, like the connecting of two words using "_" or changing from lower to upper case letters. In order to filter the result the search space in partitioned by refining the documents similarity values, so that in step $i + 1$ are only the documents of step i, which are higher than a given threshold. After that the user decides which documents are relevant to the feature. Once the user decides that no further document is relevant to the feature the the algorithm terminates. [Mar04]
Poshyvanyk uses a combination of *Marcus LSI* method and *execution-trace analysis* [3]. To analyse a program is has to be given as an input in an executable form, to determine if which methods are called on a scenario, and a set of documents, which can be defines out of a query with *Marcus*. Also the technique needs two sets of scenarios: one that invoke the feature of interest and one that doesn't. First the technique ranks the documents like within *Marcus*. After that the scenario sets are executed and execution profiles are derived. By that the methods can be ranked by the appearance within the traces of the scenarios that execute the feature versus the appearance within the other scenarios. The final result of a method is a weighted sum of the *LSI*-rank and the *trace*-rank. So the final output is the again a ranked list of methods. [PGM+07]
The technique *Liu* is quite similar to *Poshyvanyk*, with the difference, that instead of using two sets of scenarios *Liu* only works with a single scenario executing the feature of interest. This reduces the overhead of input and also accelerates the process, accepting the fact that the result may not be as accurate as it would be with *Poshyvanyk*. [LMPR07]
The technique *Revelle* combines *Information Retrieval*, *dynamic* and *web-mining* analysis in order to improve the results of the previous methods. Like *Riu Revelle* gets a single scenario that exercises the feature of interest and a query as input. While running the scenario the call graph from the execution trace is constructed, which nodes are methods that are actually executed. Every node gets a score using a web-mining algorithm like the HITS-algorithm mentioned in section 3.4. After assigning the values *Revelle* filters one of the following two out:

[3] further information on that topic within the *IEEE*-paper [AG06]

- low-ranked methods
  - typically used on *HITS authority score*
  - methods that aren't called often aren't extremely important
- high-ranked methods
  - typically used on *HITS hub score*
  - methods that call very many other methods aren't meaningful

The remaining set of methods get ranked by using *Liu* and the final ranked list is returned to the user. The overall user interaction is quite sparely, because of one scenario and a query about the feature of interest and therefore rated with "+" in Table 5.1. [RDP10]

Bibliography

[AG06] Giuliano Antoniol and Y-G Guéhéneuc. Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641, 2006.

[CR00] Kunrong Chen and Václav Rajlich. Case study of feature location using dependence graph. In *IWPC*, pages 241–247. Citeseer, 2000.

[DRGP13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013.

[HPVS07] Emily Hill, Lori Pollock, and K Vijay-Shanker. Exploring the neighborhood with dora to expedite software maintenance. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 14–23. ACM, 2007.

[KC00] Václav Rajlich Kunrong Chen. *Case Study of Feature Location Using Dependence Graph*. IWPC, 2000.

[LMPR07] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 234–243. ACM, 2007.

[Mar04] Andrian Marcus. Semantic driven program analysis. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 469–473. IEEE, 2004.

[PBvDL05] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[PGM+07] Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[RC13] Julia Rubin and Marsha Chechik. A survey of feature location techniques. In *Domain Engineering*, pages 29–58. Springer, 2013.

[RDP10] Meghan Revelle, Bogdan Dit, and Denys Poshyvanyk. Using data fusion and web mining to support feature location in software. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 14–23. IEEE, 2010.

[SFH+07] David Shepherd, Zachary P Fry, Emily Hill, Lori Pollock, and K Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224. ACM, 2007.

[Wik04a] Wikipedia. Plagiarism — Wikipedia, the free encyclopedia, 2004. [Online; accessed 22-July-2004].

[Wik04b] Wikipedia. Plagiarism — Wikipedia, the free encyclopedia, 2004. [Online; accessed 13-December-2016].

[WS95] Norman Wilde and Michael C Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[www16a] Software Product Lines `http://www.sei.cmu.edu/productlines/`, november 2016.

[www16b] Freemind website `http://freemind.sourceforge.net/`, october 2016.

[ZZL+06] Wei Zhao, Lu Zhang, Yin Liu, Jiasu Sun, and Fuqing Yang. Sniafl: Towards a static noninteractive approach to feature location. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 15(2):195–226, 2006.