RWTH Aachen University
Software Engineering Group

Feature Location Techniques

Seminar Paper

presented by
Bergerbusch, Timo

1st Examiner: Prof. Dr. B. Rumpe

Advisor: Dipl.-Inform. C. Schulze

The present work was submitted to the Chair of Software Engineering

Aachen, January 20, 2017

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 artefacts 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 artefacts 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. The use case will be the location of the *automaticSaveFile*-function of the Freemind mind mapping software, which is an open source mind map editor. [www16b].

The beginning of this paper deals with the basics of Feature Location Techniques to understand how techniques are able to define artefacts, the classification of FLT's considering their approach strategy, explaining different techniques of different sets of a taxonomy, regarding their strengths and weaknesses, on a realistic use case of a real software segment. At the end will be a recap of the explained techniques and a forecast about the further development.

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 like a Log-In, or it could be *non-functional*, i.e. completing the Log-In within 5 seconds. To be able to understand the approach strategy of a feature location technique and to derive a measurement of the result 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]. 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 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 in most cases are modified to fit the exact 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 identification of the implemented features and their corresponding artifacts.

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, also used in the paper *A Survey of Feature Location Techniques* by *Julia Rubin* and *Marsha Chechik* [RC13]. 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 related callgraph of the important parts is shown in Fig 2.1.

In the graph only the relevant constructors and methods are shown and numbered with indices from 1 to 8. These will be further referenced by using the number sign # and then the corresponding number. Also the feature of the regarded function are highlighted with a blue background colour. These are the methods which should be located if the *automatic save file* function is the wanted feature. Note that all the methods of different classes can in addition call other methods and constructors, which are irrelevant to the feature. So as it is shown within the graph the feature is mainly implemented by two methods of a subclass of *MindMapMapModel* so called *doAutomaticSave*:

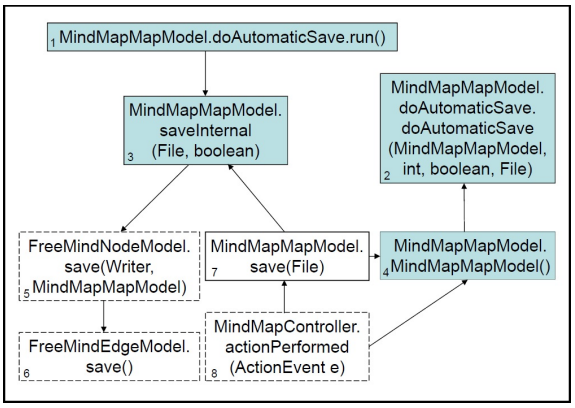

Figure 2.1: The Freemind callgraph [www16b] [RC13]

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

Regarding the previously mentioned definition of a feature by Rajlich and Chen in chapter 1, the regarded feature can be defined as the following:

name	*automatic save file*
intension	saves a file automatically after the occurring of an event
extension	#1, #2, #3 and #4

The methods #5 to #8 are not in the extension of the *automaticSaveFile* feature. Mainly # 5 and #6 are called by methods of the *automaticSaveFile* feature, but are not 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 is not important to the *automaticSaveFile* feature.

While all feature location techniques try to achieve the same goal, which is locating the matching feature extension to a given feature intension, they differ in the underlying base of assumptions they make to be able to get the traceability. 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. The *FCA* regards the binary relation between all objects and attributes and therefore can also provide a model to analyse the 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 Figure 3.1:

1. declaring every word in the objects and methods as w_i to a new i if the word is not 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_ij are checked if σ contains the word w_i

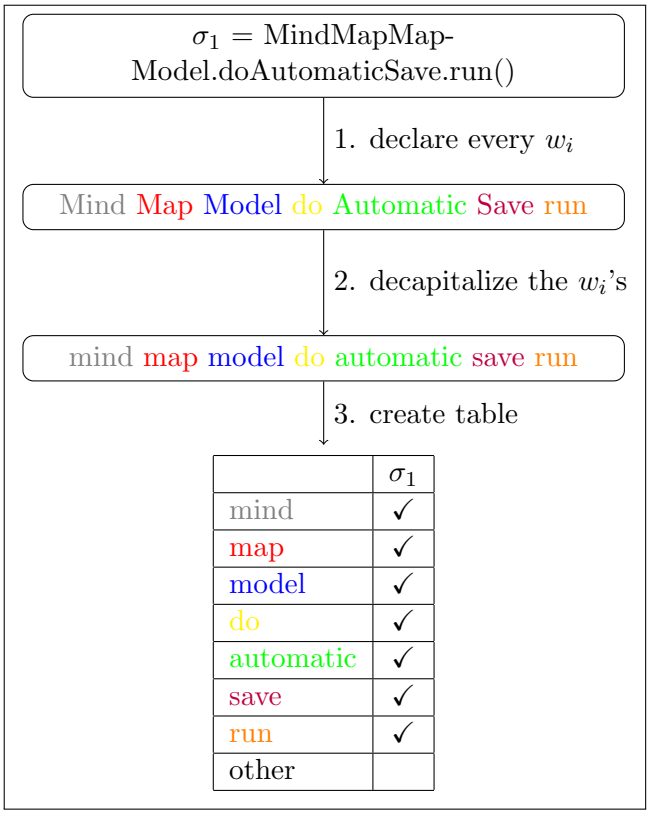

Figure 3.1: #1 of the Freemind Example as example

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 method numbers as defined in chapter 2 Figure 3.2 is the result. Mathematically it leads to defining O as a set of objects, A as a set of attributes and R as the set of relations $r = (o, a)\ o \in O, a \in A$ as derivable of the table. Also defining
$\sigma(O) = \{a \in A | (o, a) \in R, \forall o \in O\}$ "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 simple to see, that the set of all concepts C is a partial order defined as:

$$(O_1, A_1) \leq (O_2, A_2) \quad \Leftrightarrow \quad O_1 \subset O_2 \text{ or } A_1 \subset A_2.$$

(O_1, A_1) is called the *subconcept* of his corresponding *superconcept* (O_2, A_2).
Which leads to the definition that $C, \leq$ form a concept lattice and in the *Freemind-Example* (chapter 2) it's a taxonomy of name tokens.

3.2 Latent Semantic Indexing (LSI)

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

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 de-capitalized word of the methods ordered alphabetically and is represented in a row of the matrix. A document d_j, which are in the *Freemind-Example* (chapter 2) the different method- and class names, are represented as the

columns of the matrix. So the matrix, shown in Figure 3.3, looks very similar to the table of FCA (Figure 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 does not 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 analysed.

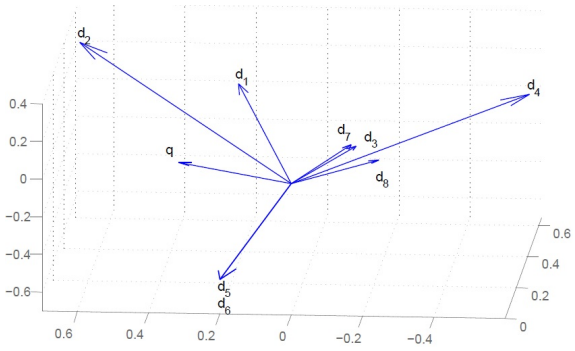

Figure 3.4: The vector representation of the documents d_j and the query q from the Freemind Example 2

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 therefore 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 therefore 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 *documentfrequency* $tf = (t, d)$, counting how often the term t is contained in the document d. In the *Freemind-Example* (chapter 2) the term $t_2 = save$ appears in d_3 once and the term $t_1 = automatic$ does not 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 the matrix shown in Figure 3.3 can be derived. 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 ca not 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 the *Freemind-Example* (chapter 2) the *idf* terms can be computed as:

$$t_1 = log(automatic/idf(t_1)) = log(8/2)$$
$$t_2 = log(save/idf(t_2)) = log(8/6)$$
$$t_3 = idf(t_3) = 0$$

Like in the example if the focus is not 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 3.1.

	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 Hyperlink 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

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

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 are not 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."

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.

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

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. Without any kind of normalization the scores of the adjacent nodes would add up in every iteration leading towards resulting scores that are great but not meaningful numbers. Therefore the normalized score can be computed as the following:

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

The normalized values satisfy the condition, that

$$\sum_{\{(p,H_p,A_p)\}\in V'} H_p^2 = \sum_{\{(p,H_p,A_p)\}\in V'} A_p^2 = 1.$$

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* (chapter 2) the first graph will look very similar to the class diagram, as it is shown in Figure 3.5. Keeping the labelling of classes in mind, which was a mapping to the numbers #1 − #8 as it is shown in Figure 2.1, the mapping of the *HITS*-Algorithm is referring to the class #i as page p_i. After transferring it into the graph of the form of G' and after the first iteration the graph looks like Figure 3.5.

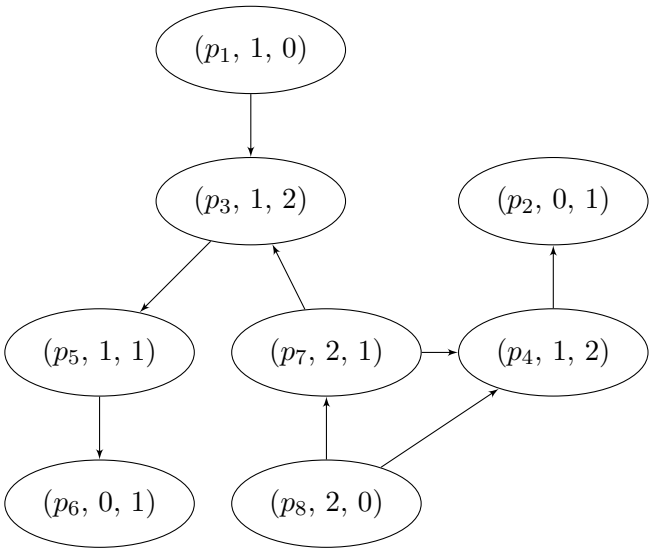

Figure 3.5: The graph G after the first iteration without normalizing

Including the normalization the graph G' looks like Figure 3.6 The normalization was done by calculating for every H_p and A_p of a page p as:

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

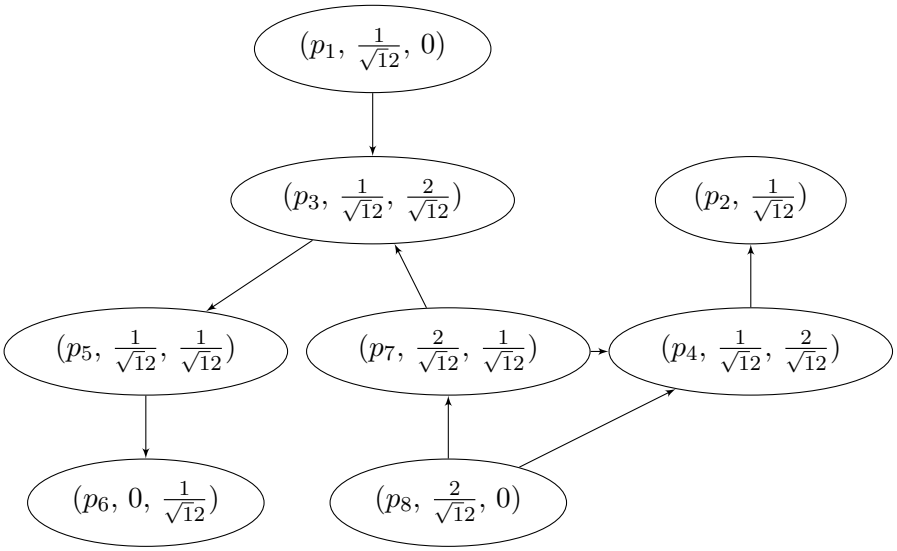

Figure 3.6: The graph G' after normalizing the hub- and authorityscores

The final HITS-Graph can be derived as the HITS-graph G from Figure 3.5 iterating for 12 times. In this example after the 12th iteration the values of the pages will not change. The resulting graph is shown in Figure 3.7. The threshold to consider if a page is relevant or not is in most cases a weighted sum of the hub- and authority score. The common way to interpret the scores is by saying, that pages with high authorityscores implement feature related functions and high hubscores coordinate feature implementing functions.

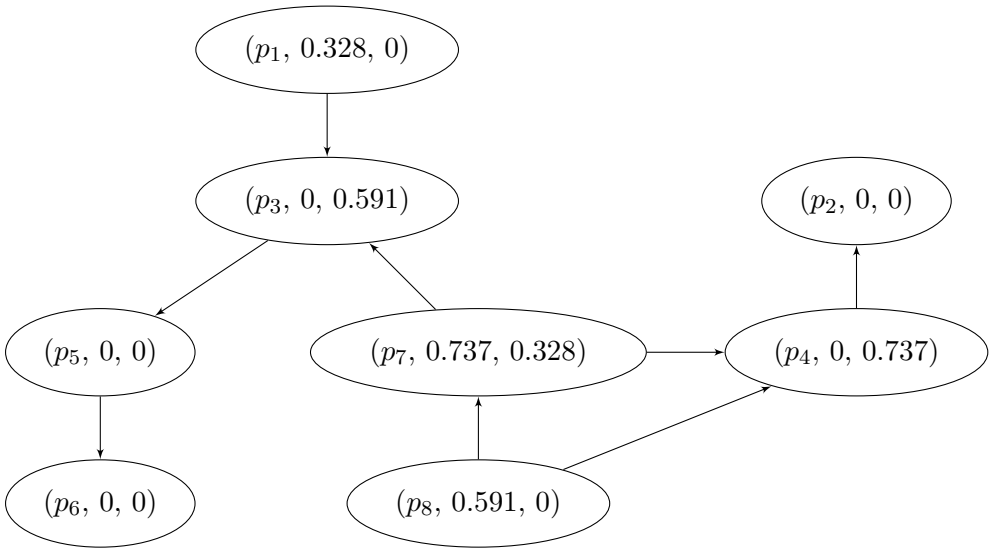

Figure 3.7: G' after 12 iteration steps and only in the last step rounded values

The resulting graph in Figure 3.7, can be interpreted by a function:

$$f(v) = \begin{cases} 1 & ,if\ v_h > 0.7 \\ 1 & ,if\ v_a > 0.4 \\ 0 & else \end{cases}$$

interpreted as $f(v) = 1$ is considered relevant, with $v \in V$ and v_h, v_a being the corresponding hub-/authorityscores. Therefore the method will consider p_3 and p_4 as relevant, which is only 50% of the pages that should be relevant as stated in chapter 2. The other two relevant pages p_1 and p_2 are not considered, so called *false-negatives*, because of their appearance within the callgraph. p_1 has a very low hubscore and no authority value and p_2 no value at all. So the *HITS-Algorithm* does not work quite well on this example because of the callgraph structure, but combining it with other techniques leads to the fact that the *HITS-Algorithm* is a frequently used technique.

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 (section 3.2) or the term frequency - inverse document frequency (section 3.3), only considering the methods and classes, which are involved during the current execution of the program. This is a big advantage, because by 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 would not 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 downside. By only analysing 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 contains the possibility of leaving out important parts not considered during the scenarios. Also collecting information on test-cases can only derive *functional* requirements, but is not able to derive non-functional requirements. In general the dynamic approaches under approximate, because of the lack of possibility to derive every requirement the and the generalizing without in depth detailed information.
static:
Static approaches do not 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 wants to locate, which can be very costly. Imagine a program which is very complex and contains thousands of lines code within hundreds of classes 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 does not 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 artefacts, 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 artefacts in a special arrangement to build an interpretation, like ordering the artefacts based on the relevance it is considered to have. Also often a so called *Program Dependency Graph* is given to not only show relevant artefacts, but also give a dependency of these artefacts. 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 *Latent Semantic Indexing*, which is explained in section 3.2, 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 does not fulfil the assumptions [RC13] [DRGP13].

Another 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 artefacts with almost only the name of the wanted feature, others need very much interaction to derive these artefacts 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 deals with five different feature location techniques in detail. They are defined as two static and two dynamic techniques with each one technique giving plain, one giving guided output and the static plain technique *SNAIFL* as a special case. The techniques presented in the following can be classified by the characteristics of chapter 4:

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

Table 5.1: The techniques discussed further on in this paper

In order to define if the technique is suitable for our Freemind Example(chapter 2) we regard the number of *false-negatives* and *false-positives*. A *false-negative* method is a method stated relevant by the example but is not a relevant feature derive from the technique. A *false-positive* is the opposite.

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 [SFH+07]. 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 using LSI (section 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 "++".

Regarding the Freemind Example of chapter 2 the technique can[1] result in the following. The input query, which has to be a *verb-DO* pair, equals *(doAutomaticSave, MindMapMapModel)*. Because of the mentioned adding of words, because of similarity or collocating the terms *save* and *saveInternal* may be added to the query. After performing *LSI* (section 3.2) the result is the callgraph looks like Figure 5.1.

The result fits the real relevant features, as mentioned in chapter 2, quite well by stating every relevant method as relevant and only stating #7 as a*false-positive* . The technique

[1] because of the different ways of extending the query the result may differ from others with the same query

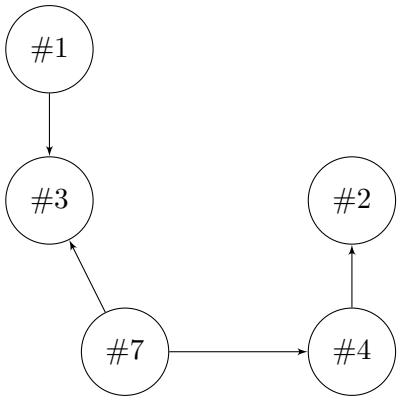

Figure 5.1: The result graph of the *Find-Concept*

fits the example quite well because of the applicable assumptions towards the underlying code, like the object-oriented structure or the verb-object writing distinction.

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*)[2][HPVS07]. 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 method n calls method 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. 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 should not be considered, which leads to the following formulae:

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

[2]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*"

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:

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

In the case of 1 and 2 Dora traverses to the neighbourhood of the node, if it does not harm 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

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

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 Figure 2.1 is user made, the scores are equal to their score in Table 3.1.
The threshold are chosen like the following:

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

So the final graph Dora computes looks like the Figure 5.2 with #1 being the *seed method*. 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 is not explored due to the missing path from #1 to #2 (with a maximum length of 3). So *Dora* is not a very good technique for this example because of #2,#3 and #4 being *false-negatives* and therefore only deriving #1 as relevant.

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 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 "+".

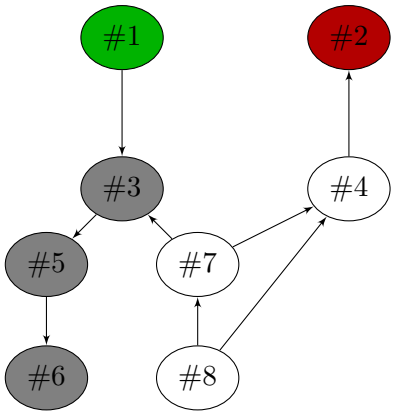

Figure 5.2: The result graph of *Dora*, with #1 as *seed method*, *depth*=3, rt=0.5 and et=0.1

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* [WS95]. *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 do not $\overline{S_f}$.
Regarding the execution traces *Software Reconnaissance* categorizes methods/lines of code M[3] into four groups:

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)

The result are the first three sets for every feature f that is set in the 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 is just a not user triggered *internalSave*. Keeping in mind the call graph(Figure 2.1) methods #3, #5 and #6 will be considered as *common* components, because of the fact that the *automatic save* feature relies on the *manual save* feature. Method #1 will be considered *uniquely involved* to the feature f_1. Methods #2 and #4 are in the *potentially*

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

involved set, because not every *automatic save* run uses these components. So we get no *false-positive* and the if #2 and #4 are *false-negatives* is a matter of defining these sets. Which shows that *Software Reconnaissance* is a suitable technique for the Freemind Example.

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:

$$Revelle[\text{RDP10}] \rightarrow Liu[\text{LMPR07}] \rightarrow Poshyvanyk[\text{PGM}^{+}07] \rightarrow Marcus[\text{Mar04}]$$

The very base technique by Marcus is to take a given 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* [4]. To analyse a program it has to be given as an input in an executable form, to determine which methods are called on a scenario, and a set of documents, which can be defined out of a query with *Marcus*. Also the technique needs two sets of scenarios: one that invoke the feature of interest and one that does not. 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 *Liu 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:

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

- low-ranked methods
  - typically used on *HITS authority score*
  - methods that are not called often are not extremely important
- high-ranked methods
  - typically used on *HITS hub score*
  - methods that call very many other methods are not 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 have to be given as an input and therefore rated with "+" in Table 5.1. [RDP10]

In the Freemind Example (chapter 2) and given a scenario, where the *automaticSaveFile*-feature is executed the methods #1,#3, #5 and #6 are invoked. Scoring these using the *HITS-Algorithm* of section 3.4 the elements receive their score in Figure 3.7. Assuming the *HITS-Algorithm* has a significant threshold #1 will be filtered, because it's authority score of 0.
The remaining methods #3, #5 and #6 are ranked by their *LSI*(section 3.2) score, which are presented in Figure 3.2 with respect to the query *automaticSaveFile*.
The result has regarding our example two *false-positives* (#5 and #6) and three *false-negatives* (#1, #2 and #4), which makes it a technique that should not be used for the Freemind Example.

5.5 Future Technique Approaches

All of the presented techniques are still under research to improve the results accuracy, the runtime and the amount of user interaction. Even if the last point is not that important in the beginning is can be the most essential due to the possibility of high serialisation. Also the generally preferred technique group is the static one, because of the high amount of pre-computing used to derive scenarios and checking if they invoke the feature and the execution time used to run these.
W. Zhao, L. Zhang, J. Sun and F. Yang from the University of Peking in cooperation with *L. Yin* of the Rensselaer Polytechnic Institute are working on an approach of a *static non-interactive* technique by using *Program Dependency Analysis* and *Information Retrieval* technologies [ZZL+06].
They define two types of functions of a feature:

1. *specific functions*: functions only used to implement the feature and not used by any other feature.
2. *relevant functions*: functions involved in the implementation of the feature

The set of *specific features* is indisputable a subset of the set of *relevant features*. The presentation of the program within the technique is realized with a so called *Branch-Reserving Call Graph (BRCG)*, which is a normal call graph expanded with branching and sequential information. These informations can be used to construct pseudo execution

traces for a feature. The *BRCG* can be written as $G = (V, E)$ with the nodes V as a function, a branch or a return statement. Loops are defined as two branch statements: one going through the loop body and one exiting immediately.

```
1  void func(){
2    f1();
3    if(condition){
4      f2();
5      return;
6    } else {
7      while(condition){
8        f3();
9        f4();
10     }
11     f5();
12   }
13   f6();
14 }
```

Listing 5.1: An example code for BRCG

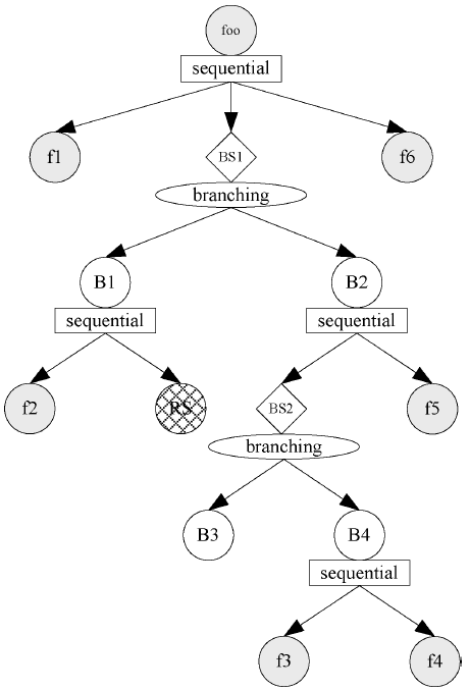

Figure 5.3: The BRCG of the example code in Listing 5.1 [ZZL+06]

In Figure 5.3 is shown an example graph for the example code of Listing 5.1, with $f1$ to $f6$ being function. The nodes $func, B1, B2$ and $B4$ are *sequential* of code, which indicate a sequence of other nodes following, while $BS1$ and $BS2$ are *branching* nodes indicating a conditional-based decision, like an if-statement or a *while*-loop. RS is a *return*-statement and $B3$ is the automatic *exit*-statement previously mentioned.
As an input the technique takes textual paragraphs for every feature, which can be derived by the requirements documentation. These paragraphs are converted into documents using the code style like in section 5.3 and normalizing[5] the terms. Also *FCA* (section 3.1) will be used to convert the methods into queries.
After deriving the documents and queries the technique uses a special *LSI* variation to convert these two sets to a vector space and measure the similarity using the *cosine*. The speciality of this variation is, that the weights of the terms, defining which terms are more important than others, are not given by the user or chosen equally balanced, but are also computed using *tf-idf* (section 3.3).
After these steps the technique has a ranked list L_d of queries(functions) for every document d (feature) ranked by their similarity. Within every list is a pair of queries p with the largest difference so $p^d = max\{(q_i^d, q_{(i+1)}^d) \in L_d | p_i^d - p_{(i+1)}^d\}$, called *division point*. Every function before the *division point* are considered *initial specific functions* to the feature.
The next step is to traverse the *BRCG* and cut off every branch that does not contain any of the *initial specific functions*, considered to be not relevant to the feature. Vice versa

[5] in general converting to lower case alphabetical letters

every other function is marked as relevant. So a pseudo-execution can be given to the user as a traversing through the trimmed *BRCG*.

Overall the technique is based on a high amount of computation but works without a single input. **But** in reality the requirements documentation and comments are not enough to generate that much knowledge about the features, that they can be derived satisfyingly. So the technique works on programs that are written in a certain way, which is not the final solution but is an approach leading the way.

Chapter 6

Conclusion

In these chapters 4 basic underlying techniques (chapter 3) and a technique for every major category (chapter 5) are presented, which are only the peak of the iceberg. Given an example of the Freemind Mind mapping software (chapter 2) and explaining the techniques based on it the general function of feature location techniques are shown. Overall the conclusion of the techniques is an obvious, but not irrelevant statement:

No *Feature Location Technique* can be perfect in every kind of scenario, but by knowing the exact fitting assumptions the best resulting technique can be chosen.

Feature Location Techniques are an essential part of software development and in preparing the steps towards Software Product Line Engineering. Reality shows that even with the best architecture and planning software project tend to be a complex and highly branched construct, which are almost impossible to be analysed without any help of techniques.
The field of Feature Location is not a new upcoming issue, but never the less it is still up-to-date. The growth of software projects keeps increasing and therefore the amount of issues that comes along. Common techniques are still refined and new techniques are developed in a constant manner. New heuristics of weighting are neglecting aspects can change a technique from an average resulting technique to one that has the ability to locate features like no other. But the even best technique is only reliable on a decent fit to its assumptions.

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.