

Assisting Concept Assignment using Probabilistic Classification and Cognitive Mapping

Brendan Cleary & Chris Exton

*Department of Computer Science and Information Systems,
University of Limerick,
Ireland.*

Brendan.Cleary@ul.ie, Chris.Exton@ul.ie

Abstract

The problem of concept assignment, that is, the problem of mapping human oriented concepts to elements in the code base of a system under study, and approaches which facilitate concept assignment can be considered as central to assisting software engineers in comprehending the unfamiliar systems they encounter. This paper presents a technique called cognitive assignment that attempts to capture what expert engineers know about the systems they work with and uses that information to generate classifiers that are used to implement a ranked search over a set of software elements.

1. Introduction

When a software engineer encounters an unfamiliar system for the first time, that engineer is tasked with understanding some or all of that system before they are able to make any meaningful contribution to its development or maintenance. While this problem is readily evident in cases of novice engineers joining existing projects [1] it also applies to experienced engineers moving between projects or in cases where a system acquired from one organization needs to be developed or maintained by another.

Tools which assist software comprehension are an integral part of the solution to this “ramp-up” problem, however while software comprehension is widely recognized as one of the pervasive problems of software engineering and while many authors have tried to establish models of how software comprehension occurs [2] [3], few authors have attempted to define what it means for a software engineer to comprehend a software system. Good recognising this deficiency ventures a definition of software comprehension in [4] which

can be considered as being characteristic of other authors attempts [5] [6] in that it establishes comprehension as a process which sees the engineer use information drawn from different sources to form a model of the software which is then used by the engineer in answering questions about the system in the context of performing some task.

Biggerstaff in defining what it is for an engineer to comprehend a software system takes a different perspective, one which de-emphasises models of comprehension and instead looks at what is required for an engineer to be said to comprehend a system [7]. This definition describes software comprehension in terms of an engineer’s ability to communicate intelligently in human oriented terms about a systems implementation. This categorisation of software comprehension rests on two different expressions of “computational intent” and the ability of the software engineer to associate concepts appearing in one description of intent with the concepts in another. Intent is what developers intend when they write software [8]. Different descriptions of intent are separated by constraints on the sets of concepts expressible using the language in which they are described. These constraints constitute a “conceptual gap” [9] between different descriptions or domains of intent. In considering software comprehension we are usually concerned with two descriptions of intent, one described using a human language (problem domain) another using a programming language (solution domain). While a systems implementation may imply the intent that led to its development it is not expressed explicitly rather it is expressed using terms defined by the implementation technologies rather than in terms that appear naturally in the intent [10]. Biggerstaff describes the problem of associating concepts between these different descriptions of intent or domains as the concept assignment problem [11].

Existing approaches which explicitly attempt to assist engineers to bridge this conceptual gap such as tool assisted, lexical, statistical and dynamic concept assignment approaches tend to rely on the parsing of solution domain artefacts (source code) to identify elements of the code base which are then inferred to be related to the implementation of some set of concepts from the problem domain. In this paper we present a complementary concept assignment approach based on the combination of a quantitative text analysis technique called cognitive mapping [12] and probabilistic classification [13].

In section 2 we look at related work from various fields that attempt to alleviate the concept assignment problem. In section 3 we describe our proposed technique the effectiveness of which is tested by an experiment described in section 4 and analysed in section 5. Finally in section 6 we describe limitations to our technique and evaluation and in section 7 we describe our conclusions and future work.

2. Related Work

Given the scope of the concept assignment problem, many techniques and tools from the software visualisation, comprehension, reengineering and even requirements engineering communities could be classified as attempting to tackle the concept assignment problem, here we will briefly examine a small subset of those that explicitly set out to do so.

Dynamic software analysis techniques such as software reconnaissance [14] or formal concept analysis [15], focus on localising concepts that are expressible either through test cases or through navigation of control and data flow. Unfortunately while a systems implementation may imply the intent that led to its development, the intent is not expressed explicitly in that implementation [10]. As such these techniques are only able to localise concepts which are expressible as test cases. While this is a limitation, in cases where there exists no system expert or documentation, they can be of great benefit in assisting engineers understand these dark systems.

Other significant areas of related work which attempt to capture and describe the relationship between problem domain concepts and the code base include tool assisted techniques such as the Concern Manipulation Environment (CME) [16] and FEAT [17] which allow engineers to explicitly describe and record associations between software elements and user defined concerns, and artefact recommender systems such as Hipikat [18]

which suggest pertinent artefacts (both code and documentation) to engineers as they engage in an understanding task. While we see these tools as closely related and complimentary to our approach; cognitive assignment differs in that it incorporates problem domain information not present in the code, captured from a system expert or experts, to assist the choices novice¹ users make when mapping problem domain concepts to elements of the code base.

Another significant area of related work are the studies into software engineer work practices carried out by Singer and Lethbridge in the mid to late 90's [19]. Using a set of field research techniques including; interview, shadowing and questionnaires which the authors collectively term software anthropology [20], Singer and Lethbridge performed a series of experiments in which they studied the work practise of software engineers as they engaged in their day to day activities. Their findings across all three studies demonstrate that search was overwhelmingly the dominant activity engaged in by the software engineers they observed. In the longitudinal study of a novice engineer, searching and looking at the source accounted for over 50% of events observed by the authors. In a second study, while editing and debugging grew in importance, searching still accounted for a significant proportion of events observed. Finally a study of tool usage statistics revealed that close to 50% of the calls made by engineers across the company were calls to grep-like search programs.

In this light of the importance of search to software understanding Marcus and Maletic [21] describe the application of an information retrieval technique called Latent Semantic Indexing (LSI) in recovering traceability links between documentation and source code. Marcus et al. expand on this work applying LSI directly to the concept location problem in [22] where they build an index of terms from identifiers and comments in the source code which are then used to localise a user specified query to a set of functions. While our approach coincides with Marcus et al's approach in terms of intent and granularity of localisation, we differ first in that our index is derived not from the source code but from software engineers with expertise in the system under study through cognitive mapping and second in that we use a different classifier to construct the mapping between a user query and the code base.

¹ We use the term novice to indicate software engineers encountering an unfamiliar SUS, these engineers may or may not have experience with other systems.

3. Cognitive Assignment

The cognitive assignment technique consists of 2 phases; a cognitive map derivation phase (performed once per each system-expert pair) and a concept assignment phase (performed each time a novice generates a query). The cognitive map derivation phase first semi-automatically derives a cognitive map from an expert software engineer related to a System Under Study (SUS) by analysing texts related to the SUS authored by the expert, such as design documentation, bug reports, or transcripts of interviews with that expert. The concept assignment phase then, each time the novice specifies a query, generates a probabilistic classifier based on a subset of the concepts and relationships in the expert's cognitive map. This subset is defined in terms of the set of concepts specified in the novices query. The generated classifier is then used to classify elements of the SUS code base according to their probable relation to concepts in cognitive map. These classifications or rankings are then displayed to the novice for their investigation through a search results interface integrated into the Eclipse IDE. The next section describes the theory behind cognitive mapping and cognitive maps.

3.1. Cognitive Mapping

A mental model is the model people have of themselves, others, the environment, and the things with which they interact, formed through experience, training and instruction [23]. Based on the assumption that language and knowledge can be modelled as networks or maps of words and the relations between them [24], texts can be thought of as containing a portion of the author's mental model at the time the text was created [12]. Working under the assumption that the meaning of a text does not result from single words but from the co-occurrence of different words [25], cognitive mapping is a quantitative text analysis technique that systematically extracts and analyses the links between words in a text in order to model the authors mental or cognitive map as networks of words [26] [27]. This map is then hypothesised to approximate a portion of the mental model of the texts author at the time the text was composed [28].

While current general purpose programming languages do not allow for the direct expression of programmer intent [29] [10], software engineers have long used other software artefacts such as requirements, architectural and design documentation and more recently email, bug tracking databases and wikis to express concerns which cannot be expressed directly in the source code. Analysing these texts using cognitive mapping

allows us to extract and make explicit the portion of the software engineer's mental model relative the system under study expressed within as maps of concepts, thus capturing and making explicit some of the original intent of the engineer. These maps can then be bound, using a classifier function, to elements in the code base of the SUS.

In [12] Carley and Palmquist present a methodology for extracting, representing and analysing cognitive maps from a corpus of texts consisting of 4 phases;

- A concept set definition phase where the set of concepts which the map is to be constructed from are identified using text pre-processing techniques which eliminate all words from the texts but those which are considered by the researcher to be important in answering the research questions.
- A relationship type definition phase that identifies the relationship types that can exist between concepts in the map, again the relationship types are determined by the researcher.
- A map construction phase where a computer-assisted coding of texts is performed using the identified concepts and relationship types. A set of statements is constructed using a windowing technique from which a map is created based on the union of the set of statements.
- Finally a map analyses phase renders the resultant maps for analysis by the researcher.

Applying cognitive mapping to texts produced by software engineers for the purposes of facilitating concept assignment requires that we customise the method presented above so that it can be applied in a production software development environment. This requires that we automate as much of the process as possible while at the same time attempting to maintain the qualitative nature of the cognitive mapping process. As such we propose to operationalize the cognitive mapping procedure of Carley and Palmquist into one consisting of 2 phases;

- A semi-automated concept set definition phase which identifies a set of concepts from a corpus of text segments using a combined manual content analysis and semi-automatic text pre-processing approach.
- A completely automated map construction phase which uses the set of concepts identified in the concept set definition phase as the basis on which conceptual maps are constructed using a windowing based approach, which creates statements be-

tween concepts in text segments which co-occur within the window.

The next section describes how we construct classifier functions from subsets of concept and relationships captured in a cognitive map and how we use those classifiers to generate rankings for individual software elements.

3.2. Bayesian Classification

Classification is a basic task in data analysis and pattern recognition that requires the construction of a classifier, that is, a function that assigns a class label to instances described by a set of attributes [30]. Applied to text classification, a naïve Bayesian classifier function, given a set of training texts and associated example classifications, determines the probability of a given term (attribute) occurring for each of the given classifications over the set of training texts. This model of conditional term probability can then be used determine the classification of an unseen text based on the product of the probabilities of the set of terms contained in the unseen text. Term or attribute probability is usually calculated based on frequency of occurrence, for example Mitchell divides the frequency of occurrence of a term in the training set by the sum of the total number of distinct word positions in the training data for the classification and the total number of distinct words in the training data [13].

In relation to the concept assignment problem a probabilistic model, based on naïve Bayesian classification, has already been used by Antoniol et al [31] for recovering traceability links between code and documentation. Here the authors used unigram estimation based on term frequency to create links that describe the similarity between elements of the code base (object-orientated classes) and high level system documentation. The authors use a stochastic language model based on identifiers found in the source code elements to calculate the set of conditional probabilities between a given source code element and the set of system documents. Naïve Bayesian classification has also been used to assist in automatically assigning bug reports to engineers with specialist knowledge [32]. Here the authors use an existing database of assigned bugs to learn a naïve Bayesian classifier that can automatically assign or classify unseen bug reports to particular engineers based on previous classifications of bugs that were made.

While being one of the most effective classifiers [30], to make the calculation of the set of conditional prob-

abilities computationally tractable, the naïve Bayesian classifier has to make a strong independence assumption that all attributes are conditionally independent given the value of the class attribute. That is, given attributes A and B and a class C , $\Pr(A|B,C) = \Pr(A|C)$ for all values of A, B and C , whenever $\Pr(C) > 0$. In text classification this independence assumption means that the order or sequence of occurrence of words in a subject text is not taken into consideration in its classification. As such naïve Bayesian text classifiers are sometimes described as treating texts as “bags of words”.

While naïve Bayes classifiers have been shown to be remarkably efficient given their simple structure, the independence assumption on which they are based is clearly not always valid. This observation lead some researchers to relax the independence assumption in an attempt to create better performing classifiers that maintain the desirable computational characteristics of naïve Bayesian while incorporating more information about dependencies between attributes.

In [30] the authors discuss the modification of a naïve Bayes classifier with augmenting edges between attributes that describe the dependencies between those attributes which are then taken into consideration when used as a classifier, thus relaxing the independence assumption of the naïve Bayes. However in order to maintain the naïve Bayes’s computationally tractable performance the authors refrain from developing augmenting edges between each pair of attributes. Instead by applying a maximum spanning tree algorithm [33] over the attribute set they are able to construct the optimal set of augmenting edges in polynomial time.

3.3. Cognitive Assignment

Our cognitive assignment procedure uses a probabilistic model, based a tree augmented Bayesian classifier formed from a subset of an experts cognitive map, to classify elements of a SUS code base in terms of how related they are to a concept set (classification) defined by the novice engineer.

Given a cognitive map M defined by an expert for a system under study S , the procedure for constructing the classifier and applying it to classification of a set of elements is as follows;

1. The novice engineer, engaged in assigning a concept C , to a set of source code elements E in S , defines a subset of the experts cognitive map

m , consisting of a set of concepts from M which the novice considers related to the concept or class C which she is attempting to localise. We call the subset, m , a concern map.

2. Given the concern map m we construct a tree augmented classifier X_m by computing a mutual information function over the set of pairs of concepts in m based on their individual and co-occurrence frequencies derived from the original texts and the cognitive map, respectively. Then using this score we annotate the edges between the pairs of concepts and derive a maximum spanning tree over the set of concepts in m .
3. We then transform the resulting undirected tree into a directed one by picking a root concept and setting the direction of all edges to be outward from it.
4. This classifier, X_m , is then used to classify the set of source code elements E according to how related those elements are to the concept C as defined by the novice engineer in m . This relationship is established based on the occurrence of concepts from m in the text of the source code elements, which includes both executable and non-executable statements.

This process is repeated each time the novice engineer generates a query set of concepts using the tool support provided in the cognitive assignment Eclipse plug-in. The cognitive assignment plug-in [34] is an Eclipse plug-in that implements the second phase of the cognitive assignment technique described above to allow an engineer encountering an unfamiliar system to construct and record a set of associations between problem domain concepts captured by a system expert in a cognitive map and the elements of the SUS code base that comprise that system. The next section describes an experiment in which we assess the performance of the cognitive assignment Eclipse plug-in in generating correctly ranked element sets.

4. Evaluation

To evaluate our proposed technique, we conducted a small lab based experiment with 4 participants to quantitatively assess the performance of our cognitive assignment Eclipse plug-in over 4 tasks in terms of precision and recall versus sets of elements defined by a system expert. A cognitive map was also defined for the SUS in the experiment using the procedure described in section 3.1. Both the expert element set and

the cognitive map we defined prior to the experiment by the primary author.

4.1. Case Study System

The experiment was performed over the CHVIE software visualisation tools framework [35]. The CHIVE has been employed in the implementation of several software understanding tools [36] and has been in development for over 3 years. The CHIVE core, the framework itself, consists of 7 packages, 25 classes and over 15 KLOC of Java. Finally between the client applications and the framework there is over 40,000 words of academic and technical text documenting CHIVE and its client applications. We chose the CHIVE framework as the basis of this case study because it constitutes a non trivial system with which the authors of this paper were intimately familiar but which the participants of the study were not and finally because the source code of CHIVE is also open source.

4.2. Participant Profile

The 4 participants selected for this study were post-graduate students, with on average 6 months of academic Java development experience and 3 years of academic development experience with other object oriented languages. The participants also had on average 3 months commercial Java development experience and over 8 months commercial development experience with other object orientated languages.

4.3. Experiment Procedure

Prior to the experiment each participant was briefed on the experiments objectives and protocol. Next the participant received training in the use of the plug-in and an introduction to using Eclipse. The participants then received a 10 minute introduction to the system against which the experiment was run. Next the participants were presented with the tasks which they were to perform in series during the experiment. For each task the participants were given 5 minutes to read the description and ask the experiment supervisor questions on the description. They were then be asked to (using the cognitive assignment plug-in) identify elements² of the source of the system under study which they thought were important to the concept/task under investigation. When the participant had completed all tasks they were thanked for their contribution, debriefed and given the opportunity to review the data collected.

² For this study we limit the element of localisation (source code unit) to the Java method; however our technique is applicable to any unit of decomposition.

4.4. Task Types

The participants were asked to complete 4 tasks, 2 concept localisation tasks, a feature request task and a bug location exercise. The tasks were each described in a paragraph of text similar to that which would be entered in a use case description, feature request or bug report. The concept location tasks required the participant to identify the elements of the system which they thought were important to the implementation of the concept as described in the given use case description. The feature request task asked the participants to identify elements that they thought either would be impacted by the proposed feature request or which could be reused in the features implementation. However for this task the users were not asked to implement the feature request. Finally the participants were asked to locate the single element that was the cause of a bug described in a bug report and demonstrated to the participant by the experiment supervisor.

5. Results & Analysis

In order to assess the performance of our technique we specified, prior to the experiment, a set of “correct” elements for each of the tasks the experiment participants would perform. These expert sets allow us to assess the performance of the cognitive assignment plug-in in generating the correct result sets. Also here we present an analysis of the lowest ranked elements investigated by the participants, this analysis helps us to empirically establish limits for the calculation of the performance of our technique and also inform future research on ranked element search in software understanding tools.

5.1. Tool Precision and Recall

Our first analysis assesses how well the cognitive assignment plug-in or more specifically, the tree augmented classifier, performed. To do this for each task we captured the final concern maps that were generated by the participants performing the experiment using the cognitive assignment plug-in. We then re-generated the set of classification probabilities produced by these concern maps. This then gave us for each task 4 sets of elements ordered by the classification function. To assess the performance of the tool we then compared these sets against the expert element set for each task.

	Task 1	Task 2	Task 3	Task 4	Average
Relevant Elements	15	9	17	1	10.5
Top 10 Total	4.75	5.75	6.5	0.5	4.375
Top 10 Recall	0.3167	0.6389	0.3824	0.5	0.45948
Top 10 Percision	0.475	0.575	0.65	0.05	0.4375
Top 20 Total	5.75	6	6.5	0.5	4.6875
Top 20 Recall	0.3833	0.6667	0.3824	0.5	0.48309
Top 20 Percision	0.575	0.6	0.65	0.05	0.46875

Table 1 - Technique Precision & Recall

We use element recall (Equation 1) to measure the number of elements correctly retrieved from the set of elements against the total number of correct elements as defined by the expert. Element precision (Equation 2) then measures the number of relevant elements retrieved against the total number of elements retrieved.

$$\text{Element Recall} = \frac{\text{Number of relevant elements reterived}}{\text{Total number of relevant elements in collection}}$$

Equation 1 Element Recall

$$\text{Element Precision} = \frac{\text{Number of relevant elements retrieved}}{\text{Total number of elements retrieved}}$$

Equation 2 Element Precision

Table 1 shows the element precision and recall achieved by the cognitive assignment plug-in, using the concern maps generated by the participants, against the expert defined element sets for the top 10 and 20 elements positions of each of the 4 tasks and the average. Here we show that our technique was able to achieve on average 45 and 43 percent recall and precision respectively when we consider the top 10 positions in the results. This rises slightly to 48 and 46 percent when we consider the top 20 positions. The cognitive assignment classifier function best performed in task 2 where we achieved precision and recall of over 60% in the top 20. While the recall in the top 20 on average was not as high as we anticipated we were satisfied with the precision rates across the first 3 tasks (task 4 had only a single correct element and so precision tends not to record the classifiers performance on this task very well).

5.2. Lowest Ranked Element Investigated

One of the risks identified by the authors prior to the experiment was the potential for participants using the cognitive assignment tool to fail to investigate all relevant classification results because of the rankings allocated.

Table 2 describes the lowest ranked element investigated by participants performing the experiment using the cognitive assignment plug-in.

Participant	Task 1	Task 2	Task 3	Task 4	Average
P1	23	38	11	9	20.25
P2	3	7	9	3	5.5
P3	14	4	9	13	10
P4	2	4	11	14	7.75
Average	10.5	13.25	10	9.75	10.875

Table 2 - Lowest Ranked Element Investigated

This analysis shows that the participants tended to only investigate those elements which were returned high in the classification results. On average the participants stayed within the top 10 results. This is an especially stark finding when we consider that 269 elements (the number of methods in the SUS) were classified and returned to the participants for each task. While these results are only preliminary we consider this a potential risk factor to the use and adoption of ranked search results to assist in concept assignment, in that if the classification function used to generate the rankings does not return the “correct” elements within the top few positions the user is likely not to investigate further down the rankings and so is likely to, initially at least, miss potentially significant elements.

6. Technique and Evaluation Limitations

Current limitations of our technique include the cognitive mapping procedure itself and the types of systems to which the cognitive assignment plug-in can be applied. The cognitive mapping procedure, originally designed as a social science research tool, can be manually intensive to implement. For this reason we are currently investigating more automatic mechanisms, which while maintaining a human in the loop, could be used for constructing simple cognitive maps in the cases where access to expert software engineers is limited. Another significant limitation of the technique is that it requires that there be a considerable amount of problem domain concepts embedded in identifiers and comments in the code. In cases where it does not hold we are investigating the use of abbreviation generator algorithms such as is presented in [37] to construct sets of candidate concepts which can be accepted in place of the problem domain concept being searched for.

Our evaluation presented here is also limited in that the size of the system under study was relatively small, 15KLOC compared to large industrial systems, as such

the performance of the cognitive mapping procedure and the cognitive assignment classification function could be a limitation when exercised over larger systems. Also the number of participants, while larger than that usually available in industrial studies, is small and so limits the generality of our results.

7. Conclusions and Future Work

We have presented here a technique for assisting concept assignment for the purposes of software understanding where engineers encounter unfamiliar systems. The cognitive assignment technique applies cognitive mapping, a quantitative text analysis technique, to texts authored by engineers familiar with existing systems. We extract from those texts the cognitive maps of the engineers related to those systems which can then be used to establish mappings between the human orientated concepts captured in the cognitive maps and elements in the system under study’s code base using a probabilistic classification function. These mappings, presented in the form of ranked search results, can then be used by engineers attempting to understand those unfamiliar systems to facilitate software comprehension.

We have implemented the cognitive assignment technique in an Eclipse plug-in and have here also presented the results of an experiment involving 4 participants where we compare the cognitive assignment plug-in’s success in generating sets of element rankings against an expert defined set. While the results of the experiment in terms of precision and recall (less than 50% on average) are not as good as we had anticipated, our immediate goal in the light for this experiment is to attempt to generalise our findings by extending this experiment to use other classifier functions and techniques such as Latent Semantic Indexing (LSI) which may demonstrate better precision and recall versus the classifier implemented here. We also wish to extend our evaluation to investigate the impact that ranked search results have on the decisions that novice software engineers make when engaged in concept assignment.

8. References

- [1] D. Cubranic and G. C. Murphy, "The ramp-up challenge in open-source software projects," presented at Workshop on Open-Source Software, held as part of the IEEE/ACM International Conference on Software Engineering (ICSE'01), Totonto, Ontario, Canada, 2001.
- [2] R. Brooks, "Towards a Theory of the Comprehension of Computer Programs," International Journal of Man-Machine Studies, vol. 18, pp. 543-554, 1983.

- [3] N. Pennington, "Comprehension strategies in programming," presented at Empirical Studies of Programmers: Second Workshop, New Jersey, 1987.
- [4] J. Good, "Programming Paradigms, Information Types and Graphical Representations: Empirical Investigations of Novice Program Comprehension," University of Edinburgh, 1999.
- [5] A. v. Mayrhauser and A. M. Vans, "From program comprehension to tool requirements for an industrial environment," Proceedings of the Second IEEE Workshop on Program Comprehension, Capri, Italy., pp. 78-86, 1993.
- [6] J. I. Maletic and A. Marcus, "Supporting program comprehension using semantic and structural information," presented at Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on, 2001.
- [7] T. J. Biggerstaff, B. G. Mitbender, and D. E. Webster, "Program understanding and the concept assignment problem," *Commun. ACM*, vol. 37, pp. 72-82, 1994.
- [8] C. Simonyi, "Intentional Programming," The International Software Corporation, 2005.
- [9] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," presented at Program Comprehension, 2002. Proceedings. 10th International Workshop on, 2002.
- [10] J. Greenfield and K. Short, *Software Factories: Assembling Applications with Patterns, Frameworks, Models & Tools*: John Wiley & Sons, 2004.
- [11] T. J. Biggerstaff, "Design recovery for maintenance and reuse," *Computer*, vol. 22, pp. 36-49, 1989.
- [12] K. Carley and M. Palmquist, "Extracting, Representing and Analyzing Mental Models," *Social Forces*, vol. 3, pp. 601-636, 1992.
- [13] T. M. Mitchell, *Machine Learning*. New York: McGraw Hill, 1997.
- [14] N. Wilde and M. C. Scully, "Software reconnaissance: Mapping program features to code," *Journal of Software Maintenance: Research and Practice*, vol. 7, pp. 49-62, 1995.
- [15] T. Eisenbarth, R. Koschke, and D. Simon, "Locating features in source code," *Software Engineering, IEEE Transactions on*, vol. 29, pp. 210-224, 2003.
- [16] W. Chung, W. Harrison, V. Kruskal, H. Ossher, J. Stanley M. Sutton, and a. P. Tarr, "Working with Implicit Concerns in the Concern Manipulation Environment," presented at Linking Aspect Technology and Evolution Co hosted with Aspect Orientated Software Development (ASOD 05), Chicago, USA, 2005.
- [17] M. P. Robillard, "Representing Concerns in Source Code," The University of British Columbia, 2003.
- [18] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: a project memory for software development," *Software Engineering, IEEE Transactions on*, vol. 31, pp. 446-465, 2005.
- [19] J. Singer and T. Lethbridge, "Studying work practices to assist tool design in software engineering," presented at Proceedings 6th International Workshop on Program Comprehension IWPC98, Ischia Italy, 1998.
- [20] T. C. Lethbridge, S. E. Sim, and J. Singer, "Software Anthropology: Performing Field Studies in Software Companies," 2004.
- [21] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," presented at Software Engineering, 2003. Proceedings. 25th International Conference on, 2003.
- [22] A. Marcus, A. Sergeev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," presented at Reverse Engineering, 2004. Proceedings. 11th Working Conference on, 2004.
- [23] D. Norman, *Things that make us smart : defending human attributes in the age of the machine*: Reading, Mass : Addison-Wesley Pub. Co, 1993.
- [24] J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*: Addison-Wesley., 1984.
- [25] Z. Cornelia and L. Juliane, "Computer-assisted Content Analysis without Dictionary," presented at Sixth International Conference on Logic and Methodology - Recent Developments and Applications in Social Research Methodology, Amsterdam, The Netherlands, 2004.
- [26] J. Diesner and K. Carley, "Using Network Text Analysis to Detect the Organizational Structure of Covert Networks," presented at NAACSOS, 2004.
- [27] R. Popping, *Computer-assisted Text Analysis*. London: Thousand Oaks: Sage Publications, 2000.
- [28] K. M. Carley, "Extracting team mental models through textual analysis.," *Journal of Organizational Behavior*, vol. 18, pp. 533-558, 1997.
- [29] K. Czarnecki and U. Eisenecker, *Generative Programming - Methods, Tools and Applications*: Addison-Wesley, 2000.
- [30] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classifiers," *Machine Learning*, vol. 29, pp. 131-163, 1997.
- [31] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *Software Engineering, IEEE Transactions on*, vol. 28, pp. 970-983, 2002.
- [32] D. Cubranic and G. C. Murphy, "Automatic bug triage using text categorization," presented at Conference on Software Engineering and Knowledge Engineering (SEKE'04), 2004.
- [33] R. C. Prim, "Shortest connection networks and some generalisations," *Bell System Technical Journal*, pp. 1389-1401, 1957.
- [34] B. Cleary and C. Exton, "The Cognitive Assignment Eclipse Plug-in (ICPC 06)," presented at International Conference on Program Comprehension, Athens, Greece, 2006.
- [35] B. Cleary and C. Exton, "CHIVE - a program source visualisation framework," presented at 12th IEEE International Workshop on Program Comprehension, Bari, Italy, 2004.
- [36] A. LeGear, B. Cleary, J. Buckley, J. J. Collins, and C. Exton, "Making a Reuse Aspectual View Explicit in Existing Software," presented at Linking Aspect Technology and Evolution Co hosted with Aspect Orientated Software Development (ASOD 05), Chicago, USA, 2005.
- [37] N. Anquetil and T. C. Lethbridge, "Recovering software architecture from the names of source files," *Journal of Software Maintenance*, vol. 11, pp. 201-221, 1999.