A Theory of Aspects as Latent Topics

Pierre F. Baldi * Cristina V. Lopes * Erik J. Linstead * Sushil K. Bajracharya

Bren School of Information and Computer Sciences
University of California, Irvine
{pfbaldi,lopes,elinstea,sbajrach}@ics.uci.edu

Abstract

After more than 10 years, Aspect-Oriented Programming (AOP) is still a controversial idea. While the concept of aspects appeals to everyone's intuitions, concrete AOP solutions often fail to convince researchers and practitioners alike. This discrepancy results in part from a lack of an adequate theory of aspects, which in turn leads to the development of AOP solutions that are useful in limited situations.

We propose a new theory of aspects that can be summarized as follows: concerns are latent topics that can be automatically extracted using statistical topic modeling techniques adapted to software. Software scattering and tangling can be measured precisely by the entropies of the underlying topic-over-files and files-over-topics distributions. Aspects are latent topics with high scattering entropy.

The theory is validated empirically on both the large scale, with a study of 4,632 Java projects, and the small scale, with a study of 5 individual projects. From these analyses, we identify two dozen topics that emerge as general-purpose aspects across multiple projects, as well as project-specific topics/concerns. The approach is also shown to produce results that are compatible with previous methods for identifying aspects, and also extends them.

Our work provides not only a concrete approach for identifying aspects at several scales in an unsupervised manner but, more importantly, a formulation of AOP grounded in information theory. The understanding of aspects under this new perspective makes additional progress toward the design of models and tools that facilitate software development.

Categories and Subject Descriptors I.2.m [*Computing Methodologies*]: Artificial Intelligence

General Terms Algorithms, Experimentation

* These authors have contributed equally to this work.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

Keywords Aspect-Oriented Programming, Scattering, Tangling, Topic Models

1. Introduction

Since its inception over a decade ago, Aspect-Oriented Programming has attracted substantial attention in both software research and industry. However, its value proposition is often a topic of debate and remains largely unproven, from a scientific perspective. Those who see a paradox in the success of AOP ignore one critical piece of information that has been put forth several times: AOP, and its instantiation in AspectJ, have been an experiment (Kiczales et al. 2001; Lopes 2004) whose empirical validation is to be done over time.

At the most fundamental level, AOP can be associated with three hypotheses: the first two hypotheses pertain to AOP in general, while the third one is specific to the AspectJ implementation. The three hypotheses are:

1 - Complex software must cope with the existence of crosscutting concerns; using traditional procedures–or object-oriented design modularizations, these crosscutting concerns are manifested in the design representations (e.g. UML diagrams or code) as design elements that are *scattered* throughout several modules and *tangled* with other concerns within those modules.

2 - Excessive scattering and tangling are "bad" for the design process, in the sense that they slow the implementation of the artifacts by: (a) forcing the developers to manually map the conceptual integrity of the crosscutting concerns to scattered pieces of design representation; (b) inducing implementation errors that result from shattered conceptual integrity; and (c) violating project management decisions related to divisions-of-labor.

3 - By using the alternative composition mechanisms embodied in a language like AspectJ, the crosscutting concerns become modules of the design; these alternative modularizations are "better" than the traditional ones, in the sense that they speed up the implementation of the artifacts by eliminating the above-stated problems.

One of the main reasons why these hypotheses have been without commonly accepted validation is because the soft-

ware research methods used so far lack the capability to validate or disprove them empirically on a very large scale. The need for better methods can be seen in two articles published by reputable industry watchers whose conclusions are the exact opposites of each other. In 2001, the MIT Technology Review magazine dubbed AOP as one of the "Ten emerging technologies that will change the world" (Tristram 2001). In 2005, industry watcher Forrester Research published a report entitled "Aspect-Oriented Programming Considered Harmful" (Zetie 2005). Given the high stakes at hand, the value of argumentative and small-scale validation of AOP has been exhausted. However, recent advances in open source development, with the production of large amounts of analyzable software data, together with recent advances in data mining techniques, are for the first time creating the conditions necessary for large-scale validation of the AOP hypotheses.

This paper begins to address these questions by developing and applying a novel set of methods for verifying the first and most fundamental AOP hypothesis regarding the existence of scattered and tangled cross-cutting concerns in software. For people with software development experience, their existence seems "obvious." However, intuitions often prove to be wrong; and when they are right, it is important to understand why and how they hold in the real world of software artifacts. For example, one frequently asked question is: besides the few prototypical aspects identified and studied in the research literature, what other aspects are there?, and what are aspects anyway? Very few studies have been conducted in order to systematically investigate this fundamental premise of AOP pertaining to the existence of crosscutting concerns. And yet, if this hypothesis proved to be invalid for software-at-large, the basis of AOP would be severely undermined.

To this end, this paper makes the following contributions:

- An infrastructure, called Sourcerer (`http://sourcerer.ics.uci.edu`), for collecting, pre-processing, analyzing, and searching software projects, in particular open source projects. This infrastructure enables empirical validation of the first AOP hypothesis on a scale three orders of magnitude larger than in most previous studies. Currently, Sourcerer indexes close to 5,000 projects retrieved from Sourceforge, Tigris, and Apache.
- A method for defining and automatically identifying software concerns at multiple scales of software granularity based on unsupervised statistical machine learning and topic modeling methods. Data mining techniques have been applied to software before. These prior applications used techniques which inject a somewhat circular reasoning into the identification of aspects, because they assume too much about what aspects look like in the software representations. An unsupervised probabilistic topic modeling technique is developed and applied here using Latent Dirichlet Allocation (LDA) (Blei et al. 2003) adapted to software data. This produces an unbiased set of latent topics, which assumes very little about what those topics are supposed to be, so that concerns are latent topics.
- A method for defining and precisely measuring scattering and tangling at multiple scales of software granularity based on information theory. Scattering and tangling are measured by the entropies of the topic-over-file and file-over-topics distributions respectively. In particular, aspects or cross-cutting concerns are latent topics with high scattering entropy.
- Large-scale experiments providing sound evidence that crosscutting, with its tangling and scattering effects, exists in software projects to a substantial degree, validating the first AOP hypothesis.

The remainder of the paper is organized as follows. Section 2 provides a description of the Sourcerer infrastructure. Section 3 describes our approach for modeling software concerns with probabilistic topic modeling, and provides the mathematical foundations of the theory. Section 4 presents the empirical validation of the model proposed here. Section 5 compares the proposed approach with previous approaches for defining, identifying, and quantifying aspects. Section 6 looks at some of the implications of the results and Section 7 discusses related work in both software engineering and machine learning and is followed by a brief conclusion.

2. The Sourcerer Infrastructure

Figure 1 shows the architecture of Sourcerer, an infrastructure for collecting, searching, and mining open source projects. (Bajracharya et al. 2006, 2007). The arrows show the main flow of information between the various components. Information on each system component is given below.

- *External Code Repositories:* These are the source code repositories available on the Internet (e.g. Sourceforge).
- *Code Crawlers and Automated Downloads:* We have several kinds of crawlers: some target well-known repositories, such as Sourceforge, others act as web spiders that look for arbitrary code available from web servers. An automated dependency and version management component provides managed downloads of these repositories.
- *Local Code Repository:* The system maintains a local copy of each significant release of the projects, as well as project specific meta-data. The scheme conforms to the project object model as given by the Maven build system.
- *Code Database:* This is the relational database that stores the *features* extracted from the source code, using PostgresSQL 8.0.1.
- *Parser / Feature Extractor:* A specialized parser parses every source file from a project in the local repository

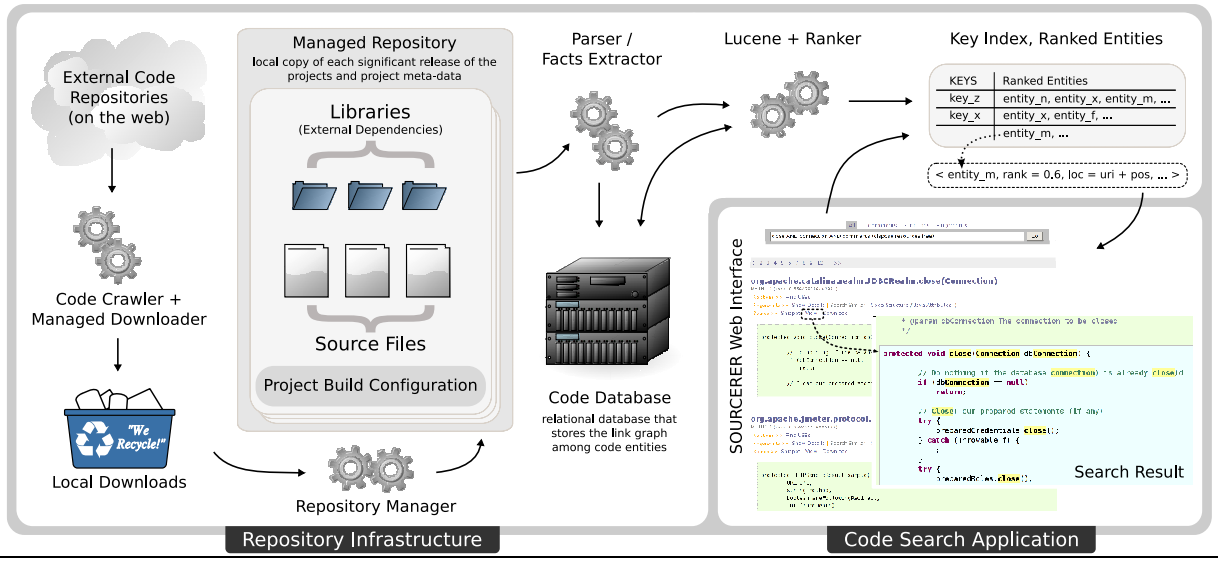

Figure 1. Architecture of the Sourcerer infrastructure

and extracts entities, fingerprints, keywords and relations. These *features* are extracted in multiple passes and stored in the relational database.

- *Text Search Engine (Lucene):* The keywords coming out from the parser, along with information about related entities, are fed into a text search engine powered by Lucene 1.9.1 (`http://lucene.apache.org`).
- *Ranker(s):* The ranker performs additional non-text ranking of entities. The relations table from the code database is used to compute ranks for the entities using several ranking techniques.
- *Search Application(s):* Search engine and applications that use the indexed keys, ranked entities, and local repository provided by the infrastructure to retrieve relevant software.

Sourcerer is in the process of being expanded with a set of APIs so that other researchers can use this infrastructure to conduct validation experiments and develop other experimental tools.

3. Software Concerns as Latent Topics

Identifying scattering and tangling in existing programs has been the thrust of a line of research in AOP known as aspect mining. This line of research has produced several techniques, including: query tools of varying sophistication (Griswold et al. 2001; Hannemann and Kiczales 2001), fan-in and fan-out analyses (Marin et al. 2004; Zhang and Jacobsen 2007), clone detection (Bruntink et al. 2005), CVS history analyses (Canfora et al. 2006; Breu and Zimmermann 2006), run-time analyses (Breu 2005; Tonella and Ceccato 2004), natural language processing (NLP) of source code (Shepherd et al. 2005b), and supervised machine learning techniques (Shepherd et al. 2005a). All of these techniques have had some degree of success in small-scale experiments, typically limited to one or a few projects.

These techniques can be seen as attempts to extract higher-level concerns from source code, specifically those concerns whose implementations end up scattered in several modules. As such, these techniques tend to mix two issues together: (1) concern extraction; (2) quantifying scattering. To overcome this confusion, here we first develop a statistical topic modeling technique to identify concerns, and then apply information theory to precisely measure scattering and tangling. Specifically, we have adapted Latent Dirichlet Allocation, which probabilistically models text documents as mixtures of latent topics, where topics correspond to key concepts present in the corpus, and documents are viewed as "bags of words". A by-product of this approach is the full distribution of each topic across the modules, and of each module across the topics. As a result, we propose to measure scattering and tangling by the entropy of the corresponding distributions.

3.1 Latent Dirichlet Allocation

In the LDA model for text, the data consists of a set of documents. The length of each document is known and each document is treated as a bag of words. Let D be the total number of documents, W the total number of distinct words (vocabulary size), and T the total number of topics present in the documents. Here, for simplicity, T is assumed to be fixed beforehand (e.g. $T = 125$), but we experiment with various values of T, in order to find a balance between topics that are overly general and those that are overly project-specific. However, non-parametric Bayesian and other methods exist also to try to infer T automatically from the data.

The model assumes that each topic t is associated with a multinomial distribution $\phi_{\bullet t}$ over words w, and each document d is associated with a multinomial distribution $\theta_{\bullet d}$ over topics. More precisely, the parameters of the model are given by two matrices: a $T \times D$ matrix $\Theta = (\theta_{td})$ of document-topic distributions, and a $W \times T$ matrix $\Phi = (\phi_{wt})$ of topic-word distributions. In generative mode, given a document d containing N_d words, for each word the corresponding $\theta_{\bullet d}$ is sampled to derive a topic t, and subsequently the corresponding $\phi_{\bullet t}$ is sampled to derive a word w. A fully Bayesian probabilistic model is derived by putting symmetric Dirichlet priors with hyperparameters α and β over the distributions $\theta_{\bullet d}$ and $\phi_{\bullet t}$. For instance, the prior on $\theta_{\bullet d}$ is given by

$$D_\alpha(\theta_{\bullet d}) = \frac{\Gamma(T\alpha)}{(\Gamma(\alpha))^T} \prod_{t=1}^{T} \theta_{td}^{\alpha-1}$$

and similarly for $\phi_{\bullet t}$.

The probability of a document can then be obtained in a straightforward manner by integrating the likelihood over parameters ϕ and θ and their Dirichlet distributions. The posterior can be sampled efficiently using Markov Chain Monte Carlo Methods (Gibbs sampling) and the Θ and Φ parameter matrices can be estimated by maximum a posteriori (MAP) or mean posterior estimate (MPE) methods.

LDA has typically been applied to traditional text data, such as journal articles or emails. To apply LDA to software, a particular type of text data, a number of adaptations are required. Source files can be viewed as documents whose content can be represented as bags of words. A key step in the application of LDA to software is to identify the words that will constitute the topic model vocabulary, and ultimately produce the word-document matrix, which represents the occurrence of words in individual source files. To build the word-document matrix, we have developed a comprehensive tokenization tool tuned to the Java programming language that allows us to build the corpus vocabulary from specified code entities. This tokenizer includes language-specific heuristics that follow the commonly practiced naming conventions. For example, the Java class name "DynamicConfigurator" will generate the words "dynamic" and "configurator." In the process of indexing projects, Java keywords and punctuation are ignored. Free text in the source files, i.e. comments, can either be accounted for, or ignored. More generally, using Sourcerer we can selectively choose which structures to account for when harvesting words – class definitions, method definitions, method calls, method bodies, etc. For this study we included all class, interface, method, and field names. Section 4.2 discusses in detail the experiments that drove vocabulary selection.

Once the topics of a project are identified, the quantification of scattering and tangling can be easily derived from the corresponding distributions, for instance in terms of entropy. For example, if the distribution of topic t across modules $m_0 \dots m_n$ is given by $p^t = (p_0^t \dots p_n^t)$ then scattering of topic t can be measured by the entropy

$$H(p^t) = -\sum_{j=0}^{j=n} p_j^t \log p_j^t \quad (1)$$

Likewise, if the distribution of the module m across the topics $t_0 \dots t_r$ is given by $q^m = (q_0^m \dots q_r^m)$ then tangling in module m can be measured by the entropy

$$H(q^m) = -\sum_{j=0}^{j=r} q_j^m \log q_j^m \quad (2)$$

Note that the entropy of a uniform distribution over M classes is given by $\log M$. Thus entropies can be normalized to the [0,1] interval by dividing by $\log M$ to enable comparison of entropies derived over different numbers of classes.

3.2 LDA and Software Concerns: A Theory of Aspects

There is a strong conceptual similarity between latent topics and the concepts of *concerns* and *aspects*. So much so, that we propose to unify the concept of latent topic with the concept of concern in the domain of software:

A concern is a latent topic.

The distribution of a topic across modules indicates whether the topic is more or less scattered. A cross cutting concern is a latent topic that is scattered. This leads to the following proposal for the definition of an aspect:

An aspect is a latent topic with high scattering entropy.

Conversely, a topic with low entropy is not an aspect. Note that entropy is a continuous notion and it may not be necessarily productive to try to define a threshold separating aspects from non-aspects. This definition of aspects seems to be in line with the original idea of aspects as emergent properties of program representations – again, a concept that maps well to the notion of latent topics.

In short, statistical topic modeling provides a potential *theory* for AOP by providing definitions and procedural means for identifying software concerns and aspects, and for precisely measuring their degree of scattering and tangling in actual software data. To test this proposal, several experiments conducted at multiple levels of software granularity are presented and discussed in the following sections.

4. Experimental Results

This section describes a representative subset of results obtained by applying LDA to software data to extract concerns and aspects, as well as their scattering and tangling. While space constraints prevent their inclusion, complete results are available from the supplementary materials page at: `http://sourcerer.ics.uci.edu/oopsla08/results.html`.

Table 1. Data Set.

Projects (with source)	4,632
Files	366,287
Packages	47,640
Classes	426,102
Interfaces	47,664
Methods	2,694,339
Fields	1,320,067
LOC	38,700,000

4.1 Data Set

Using Sourcerer, approximately 12,000 distinct projects were downloaded, primarily from Sourceforge and Apache. Distributions packaged without source code (binaries only) were filtered out. Parsing the resulting multi-project repository yields over 5 million entities organized according to Table 1. The end result is a repository consisting of 4,632 projects, containing 366,287 source files, with 38.7 million lines of code, written by 9,250 developers.

4.2 Empirical Vocabulary Selection

Because Sourcerer separately indexes each code entity (eg. classes, methods, fields) from a given source file, the topic modeling process has significant flexibility when determining which words or tokens should be included in the vocabulary when constructing the document-word matrix. For example, one may choose to represent source file contents using only tokens derived from class names, or one may choose to include words originating from class, method, and field names. While such a decision has no effect on the mechanics of LDA, the choice of vocabulary is ultimately manifested in the resulting topics extracted from the code, both in terms of their specificity and their clarity. To ensure that the topic models capture the existence of crosscutting concerns in source code, several vocabulary options were explored.

As a first attempt in our topic modeling analysis, we used the entire source file texts as bags of words. This resulted in a huge word-document matrix that was computationally heavy to process. Furthermore, the quality of the topics was somewhat mixed. For example, a topic that emerged with very high entropy was copyright notices. While this is definitely an important part of software development, and worth studying, we wanted to focus on code elements, and not on all other activities that find their way into source code files.

As a second attempt, we constructed source files such that their content consisted only of class and interface names. For the full repository this yielded a vocabulary of 49,521 words across approximately 360,000 documents. Results showed that this over-constrained vocabulary produced topics that were both too noisy and too general to be of use in the analysis, essentially providing a topic model of file names, but not of file content. To improve the results we augmented file content to include class, interface, and method names, producing a vocabulary of 89,232 words for the repository. With this change the extracted topics provided some insight into code function, but did not adequately provide insights about implementation facilitated through member variables or method calls to other classes. Thus we decided to drastically expand the modeled vocabulary with this information.

We finally settled on including class and interface names, method and field signatures, and called method names. This proved to be the best compromise between computational overhead and topic quality. This yielded a vocabulary of over 140,000 words and a substantial improvement in the quality of the extracted topics.

When discussing vocabulary selection it is also important to note that extracted topics can be further refined by employing a stop word list to prune common words that contribute little to program understanding. While such stopword lists are common for natural languages in the information retrieval (IR) community, standard lists do not exist for code search and mining, and must be created manually. As a first cut we constructed a stopword list consisting of class names from the Java SDK as well as common English words, believing that this would focus the topic models on what the code was doing, rather than how it was doing it. When examining results, however, it became clear that many crosscutting concerns are facilitated through the standard Java classes. Logging, for example, may be achieved by leveraging the various Java I/O classes. While common, including the names of such classes substantially improved the interpretability of results when mining code on the large scale. Thus, we pruned our stopword list to include only common English words from a standard IR stopword collection, and also excluded all text found in comments. In conjunction with class, interface, method, field, and called method names, SDK class names yield a vocabulary of 141,136 words, and provide the foundation of the scattering and tangling analysis presented below.

4.3 Aspects and Scattering in the Large

Table 2 shows 125 topics identified with the topic modeling method applied to the full repository described in Table 1, and ordered by normalized entropy of their scattering among the files. Due to space constraints we present topics as a list of the 5 most likely words for each concept; in actuality each topic is a probability distribution over all words in the vocabulary, and additional words can be inspected to assist in interpreting each topic. The entropy was calculated using Equation 1 in Section 3. Normalized entropy takes a value between 0 and 1, and represents the uncertainty associated with the random variable representing a given topic's distribution over files. An intuitive interpretation is that topics with high entropy are more pervasive than those with lower entropy, with an entropy of 1 representing a topic with a uniform distribution over all files and an entropy of 0 representing a topic assigned to only one file. These results are in line with results obtained using this technique

Table 2. Scattering Results for Full Repository.

Topic	Entropy	Topic	Entropy
'name names folder full qualified'	0.830006642	'current time task millis system'	0.752075685
'object lisp objects unwrap coerce'	0.81110364	'vector element size add remove'	0.750874803
'add param controller section params'	0.808862952	'model selection cell object editor'	0.75081679
'string equals blank virtual slashes'	0.805664392	'end start line offset begin'	0.750695144
'value boolean integer warn poinfo'	0.802285953	'reference object string space home'	0.749890641
'string case length with substring'	0.801405454	'element document attribute schema child'	0.749155948
'string display initialize refresh mask'	0.798702917	'char length character string chars'	0.748791035
'exception illegal argument runtime pointer'	0.796180398	'action event performed menu add'	0.746993332
'string concat jam outdent gethandle'	0.795911551	'selected button panel enabled box'	0.74666193
'string callback annotation dao native'	0.795584257	'byte read write bytes short'	0.746638563
'print println stream main dump'	0.794642995	'user group role application permission'	0.744858387
'create helper console factory creator'	0.79379934	'server socket send address client'	0.743774049
'throwable trace stack print message'	0.79127469	'string report definition def resolve'	0.742770604
'type types java primitive fragment'	0.790930013	'status transaction cache open commit'	0.74106993
'buffer string append length replace'	0.790911771	'make not opt condition empty'	0.740774325
'error log debug string throwable'	0.790188551	'connection query execute close driver'	0.740733329
'code equals object hash blog'	0.789997428	'state pos desc initial transition'	0.740178479
'iterator next has collection abstract'	0.787162973	'handle string script app gtk'	0.739446967
'list add size linked remove'	0.787161183	'test suite down concept main'	0.739314191
'class loader name instance classes'	0.786804788	'block rule option options symbol'	0.736177097
'size clear reset mark use'	0.786009268	'generate engine spec provider extension'	0.736034727
'stream input output read write'	0.785941016	'event mouse component focus cursor'	0.733843031
'index count compare sort comparator'	0.784745691	'handler entity string prefix identifier'	0.733316293
'system string exit main runtime'	0.784522048	'table column row count rows'	0.730981259
'description string factory record name'	0.783416443	'view point edit active figure'	0.730484746
'parse int command string parser'	0.783176526	'filter access random channel sample'	0.729546548
'array list add size abstract'	0.78180484	'impl ref operation object unsupported'	0.728713961
'new for return member instance'	0.781180287	'abstract register convert proxy builder'	0.72374688
'default bean values widget history'	0.779643822	'request servlet http response session'	0.723093293
'instance process post device activity'	0.775346428	'string project template link cms'	0.722859294
'check find all store and'	0.774424449	'integer big decimal string value'	0.722771094
'string url header uri encode'	0.771714455	'date format time calendar day'	0.720240969
'info parameter parameters attr doc'	0.768179882	'image graphics width draw height'	0.71852396
'thread run start stop wait'	0.767481896	'mode zip unit units calc'	0.718489486
'string config password login email'	0.76717566	'expression variable function evaluate expr'	0.717246376
'version position uid serial render'	0.767144361	'string copy language modified region'	0.714682308
'map hash put contains generic'	0.766980763	'font label style color border'	0.712924194
'file path directory exists dir'	0.766787047	'graph left right top edge'	0.709464393
'last first word after before'	0.765637113	'target internal lookup drop drag'	0.709066174
'text area caret length wrap'	0.765305668	'logger level logging log settings'	0.708989244
'form string mapping local forward'	0.764609549	'key primary single enabled find'	0.70555609
'listener change remove add fire'	0.764262932	'descriptor feature string wrapper seq'	0.704315125
'writer reader write buffered read'	0.763701713	'method class reflect object call'	0.703677432
'item resource locale bundle items'	0.763171752	'string attribute attributes from element'	0.70292847
'clone base setup interface default'	0.762996228	'visit simple visitor accept plugin'	0.7006347
'context manager results execution factory'	0.762984164	'assert equals test true null'	0.699647577
'data meta converter idata dbobject'	0.762827588	'player game move board score'	0.689682317
'string source load flag trans'	0.761916518	'string sub str val rel'	0.689211564
'string content xml title track'	0.761577087	'search string database order product'	0.688891143
'message session msg send messages'	0.760400967	'tag page out body start'	0.684469105
'property properties string prop load'	0.760035593	'statement result prepared prepare close'	0.683121859
'configuration validate string flow obj'	0.760015988	'color module background world red'	0.681868614
'entry service string valid complete'	0.758106581	'geom transform shape fill stroke'	0.679129364
'string match pattern sequence regex'	0.757579105	'long move literal getn read'	0.678205098
'string token tokenizer next tokens'	0.756539834	'sql object fields persistence jdbc'	0.676782638
'component container layout size border'	0.756329833	'instruction constant stack push pop'	0.658630273
'icon location control tool bar'	0.75542027	'mob environmental can stats room'	0.649525821
'update select delete insert build'	0.755399561	'node scope token nodes scan'	0.645260254
'double max math min num'	0.755215107	'any logic context standard html'	0.643117635
'number from step back activation'	0.753931021	'category range domain axis paint'	0.641137336
'hashtable elements enumeration has next'	0.753734287	'field string security underlying leg'	0.496411488
'window show frame dialog component'	0.753499534	'long address gsl short matrix'	0.46562778
'tree path parent child root'	0.753069533		

in natural language texts. Most of these latent topics can be easily recognized and tagged, while a few may seem obscure. The topics include include:

- manipulating strings: 'string equals blank virtual slashes', 'string case length with substring', 'buffer string append length replace'
- exception handling: 'exception illegal argument runtime pointer', 'throwable trace stack print message'
- printing on the screen: 'print println stream main dump'
- logging: 'logger level logging log settings'
- configuration: 'property properties string prop load', 'configuration validate string flow obj'
- iterating through collections: 'iterator next has collection abstract'
- manipulating lists: 'list add size linked remove'
- reading/writing from/to streams: 'stream input output read write', 'writer reader write buffered read'
- concurrency: 'thread run start stop wait'
- login: 'string config password login email'
- authentication: 'user group role application permission'
- interacting with the file system: 'file path directory exists dir'
- event handling: 'listener change remove add fire'
- GUIs and GUI events: 'action event performed menu add', 'event mouse component focus cursor', 'text area caret length wrap', 'font label style color border', 'selected button panel enabled box', 'icon location control tool bar', 'component container layout size border', 'window show frame dialog component'
- input parsing: 'string match pattern sequence regex', 'string token tokenizer next tokens'
- traversing trees and graphs: 'tree path parent child root', 'graph left right top edge'
- timing and date actions: 'current time task millis system', 'date format time calendar day'
- xml data representations: 'element document attribute schema child'
- networking: 'server socket send address client'
- data persistentcy: 'connection query execute close driver', 'table column row count rows', 'statement result prepared prepare close', 'sql object fields persistence jdbc', 'update select delete insert build', 'status transaction cache open commit'
- testing/assertions: 'test suite down concept main', 'assert equals test true null'
- web interfacing: 'request servlet http response session'
- the factory pattern: 'create helper console factory creator'
- the visitor pattern: 'visit simple visitor accept plugin'

The first striking observation is that some of these topics correspond to the aspects that have been used as prototypical examples for AOP, namely: exception handling, logging, concurrency, persistency, authentication, GUIs, and even a couple of well-known design patterns (eg. the visitor pattern). In order to better understand this result, we need to analyze the meaning of these topics in the context of the entire repository.

The topics are given in latent manner by lists of words that occur frequently together, in probabilistic terms. The fact that their entropy is so high tells us that these topics are pervasive in the large collection of software that was analyzed. In other words, concerns such as string manipulation, exception handling, and so on, recur quite frequently in the 366,287 source files of the repository. This result is not surprising, and it attests to the validity of this method: the list above is a comprehensive list of topics/knowledge that Java software developers have to master. Some of them are basic interactions with widely-used classes of the JDK, such as string, collection, list, and file manipulations, as well as exception handling – the basic building blocks of Java programs. Others such as logging, concurrency, and authentication are not so basic, but have an equally strong scattered presence.

An essential property of the statistical topic modeling approach is that it is an *unsupervised* method, i.e. topics emerge automatically from the data, without the need for human supervision or annotation, or for a pre-existing and possibly biased definition of concerns and aspects. There is no need for an annotated training set or for using structural properties such as fan-in/fan-out or any other heuristics. As the results show, without using any assumptions about aspects, crosscutting exists on both large and small scales, in the form of latent topics that interact with each other in the source files that form the programs.

4.4 Aspects and Scattering in Individual Projects

While validation of the topic modeling approach is essential on the large scale, to complete the validation it is also important to analyze individual projects in detail. To this end, here the topic modeling approach is applied also to 5 individual open source projects: JHotDraw (`www.jhotdraw.org`), Jikes (`jikes.sourceforge.net`) , PDFBox (`www.pdfbox.org`), JNode (`www.jnode.org`), and CoffeeMud (`coffeemud.zimmers.net`). Together these projects represent a collection of well-known, non-trivial software products of varying size and complexity, spanning a diverse set of domains from technical drawing to gaming. In this section we briefly describe each of the projects considered, and provide selected results of our aspect analysis.

Tables 3 through 7 present selected topics and their scattering for each of these 5 projects. For each project, a rep-

Table 3. Example JHotDraw Topics.

Topic	Entropy
'instance test tear down vault'	0.813075061
'create factory collections map from'	0.722463637
'point move box index start'	0.71436202
'storable read write input output'	0.650160953
'list next has iterator add'	0.638290561
'polygon point internal chop count'	0.46080295
'size selected frame frames dimension'	0.43364049
'shape geom rectangular rectangle2 hashtable'	0.353301264
'drag drop target source listener'	0.352124151
'event component size transform mouse'	0.338653373

Table 4. Example Jikes Topics.

Topic	Entropy
'next has element enumeration elements'	0.699351996
'buffer check empty char insert'	0.661522459
'print stream println writer total'	0.636898546
'hash map iterator next add'	0.636035451
'type array reference code resolved'	0.635043332
'cycles end time right begin'	0.486326254
'field type reflect value unchecked'	0.4684958
'short switch reference type read'	0.447104842
'sys lock unlock write socket'	0.428127362
'offset mask fits forward code'	0.346995542
'emit assembler gen reference laddr'	0.266546555

Table 5. Example PDFBox Topics.

Topic	Entropy
'file stream print close pddocument'	0.762635663
'string int date embedded calendar'	0.759588402
'list size add array cosarray'	0.756535565
'page box pdpage find node'	0.674842006
'byte class width code line'	0.639055832
'font name width kern character'	0.629830087

resentative sample of high and low entropy topics are listed, along with their normalized entropy value.

JHotDraw is a well-known open source GUI framework for drawing technical and structured graphics. Originally conceived by Erich Gamma and Thomas Eggenschwiler, the current version of the software (6.0 beta 1) indexed by Sourcerer consists of 485 files representing 650 classes, 4,712 methods, and 845 fields across 28,335 lines of code. Selected topics for JHotDraw are shown in Table 3.

The Jikes project provides an open-source implementation of a Java virtual machine, allowing researchers to easily plug in and explore new algorithms for garbage collection, threading, and optimization. The JikesRVM 2.4.4 code base considered here consists of 940 files corresponding to 1,149 classes, 9,045 methods, and 4,572 fields with 170,066 lines of code. Selected topics for Jikes are shown in Table 4.

Table 6. Example JNode Topics

Topic	Entropy
'string length append substring tokenizer'	0.76224123
'map hash equals object value'	0.726874809
'byte array bytes arraycopy system'	0.723141514
'stream write output writer array'	0.723069203
'input read stream reader buffered'	0.718017023
'graphics color paint icon rectangle'	0.567084036
'image raster buffered create writable'	0.548839911
'time date calendar zone simple'	0.525970475
'zip entry jar plugin deflater'	0.515858882
'focus event window component listener'	0.502999404

Table 7. Example CoffeeMud Topics.

Topic	Entropy
'environmental mob msg location send'	0.861222835
'environmental name text vector string'	0.823602707
'vector element size add remove'	0.795135882
'mob hash environmental iterator next'	0.77667159
'string mob currency environmental shop'	0.600152681
'string channel imc send mud'	0.591218453
'string vector from xml buffer'	0.586218656
'string mob gen scr tell'	0.390775366

PDFBox provides a substantial Java library for creating, manipulating, and converting portable document format (PDF) files, and represents an open source project with a substantial following. In terms of documents, PDFBox 0.7.2 represents the smallest project in the collection with a total of 370 indexed files. The project is comprised of 384 classes, 2,955 methods, and 1,255 fields with 38,241 lines of code. Selected topics for PDFBox are shown in Table 5.

JNode provides an open source Java implementation of an operating system, the goal of which is to be able to install and run any Java application in an efficient and secure environment. The largest of the 5 projects, JNode represents a substantial software product with approximately 6,200 files consisting of 6,599 classes, 45,792 methods, 20,264 fields and a substantial 610,000 lines of code. Selected topics for JNode are shown in Table 6.

Finally, CoffeeMud represents a full Java implementation of a game engine for text based role-playing or adventure games, including facilities for online play. A substantial project with over 2,900 files, CoffeeMud is composed of over 2,989 classes containing 29,111 methods and 5,081 fields with 379,710 lines of code. Selected topics for CoffeeMud are shown in Table 7.

Taken together, these tables provide further support for the idea that meaningful software concerns, and their scattering, can be identified automatically and quantified through the unsupervised topic modeling approach and the resulting distributions and associated entropies. The first observation is that by zooming in on individual projects, we start see-

ing project-specific topics, in addition to the general-purpose topics seen for the entire repository. For example, in JHotDraw we see drawing concerns given by 'point move box index start', 'polygon point internal chop count', 'shape geom rectangular rectangle2 hashtable', as well as a strong presence of GUI given by 'drag drop target source listener', 'event component size transform mouse'. In Jikes, we see programming language concerns such as 'field type reflect value unchecked', 'offset mask fits forward code' and 'emit assembler gen reference laddr'. In PDFBox, we see PDF-related concerns in 'page box pdpage find node' and 'font name width kern character'. In JNode, we see program image concerns in 'image raster buffered create writable', as well as Java packaging concerns in 'zip entry jar plugin deflater'.

At first sight, CoffeMud does not seem to support the argument. We include it here in order to illustrate an important point about this approach. The identification of high-level concepts based on these sets of keywords often requires domain and even project expertise. In the case of CoffeeMud, the latent topics identified did not seem to make much sense. In particular, the word 'mob', which appears repeatedly, is quite foreign. Further investigations about this project (e.g. in Wikipedia) indicate that 'mob' is actually a central concept of this system and of multi-user domains (MUDs) in general. Mob stands for Mobile Object, which corresponds to [artificial] players that can move around in the gaming environment. Given this additional piece of information, we can then see several project-specific concerns given by 'environmental mob msg location send' (mob communication), 'string mob gen scr tell', as well as a concern that seems to be related to Instant Messaging, 'string channel imc send mud'.

A second observation is that the general-purpose concerns tend to have higher entropy than the project-specific concerns, i.e. they are more scattered. This is, in many ways, an expected result, but it has important implications for AOP, discussed in the next section. All projects, except CoffeeMud, show this property. CoffeeMud, again, seems to be different, in that the topic with highest entropy is a project-specific topic. In inspecting the code, one can see that the first topic is related to the basic communication mechanisms provided by the engine. While we have not explored this further, it may be that CoffeeMud's communication facilities, which in a normal project would appear with words such as 'socket, send, msg', for example, wrap around the underlying general-purpose communication facility java.net. And this may drive the entropy up, given that communication is something that needs to happen at several points of the code.

Figure 2 provides a visual summary of scattering behavior for the 5 projects chosen for this analysis. Scattering curves are produced by sorting the topic entropies of a project in descending order, and plotting them versus the number of topics extracted from the project. In addition to providing a simple means for examining topic scattering within a single project, one can use such a visualization to directly compare scattering across projects. For example, from the figure one can see that scattering values for CoffeeMud are noticeably higher than the remaining projects. One can also compare the variance of scattering within projects, noting that the range of values for JNode, for example, is substantially more concentrated than JHotDraw. Ultimately these curves provide a graphical representation of scattering that can be used to drive additional investigation at the project or multi-project management level.

4.5 Aspects and Tangling in the Large

Using our model it is also mathematically straightforward to quantify the tangling of concerns in the files by computing the entropy of the distribution of files over topics (Equation 2 in Section 3). Normalized tangling entropy is again a number between 0 and 1. Intuitively, a file with a higher entropy contains code corresponding to a wider variety of topics than a file with a lower entropy. A maximum normalized entropy of 1 is assigned to a file whose topic assignment is uniformly distributed. A minimum entropy of 0 is assigned to a file with only 1 topic assignment.

Table 8 provides a very small sample of tangling results for the topic model of the full repository, highlighting 5 files with high entropy and five files with low entropy.[1] Three of the five files with high entropy correspond to report generation. In looking at these files, one notices that they include string manipulation, database access, exception handling, interface design, and a wide variety of other concepts. The same can be said for the remaining high entropy files, which constitute an IMAP server (.788) and a music organizer (.7664), both of which are noticeably complicated in implementation. At the other end of the spectrum, the low entropy files correspond to very specific functional modules, such as handling nested class declaration in an abstract syntax tree (.3379) and column manipulation database code (.2275). The table also gives examples of 2 files with a tangling of 0.0. Door.java originates from the CoffeeMud game engine, and is assigned only to the topic 'mob environmental can stats room.' Similarly, FocusLostEvent.java, a specific event listener for capturing mouse focus, is assigned only to the topic 'event mouse component focus cursor.' In inspecting these files, we confirmed that they are quite simple.

4.6 Aspects and Tangling in Individual Projects

As with scattering, the entropy method for quantifying tangling is applicable also at the granularity of a single project. Table 9 and Table 10 provide sample results for JHotDraw and Jikes, respectively (we omit the other three projects, because the results are similar). For each, a select number of files exhibiting both high and low levels of tangling are given, together with the corresponding entropy score. For

[1] The entire tangling matrix has 366,287 rows, one for each file.

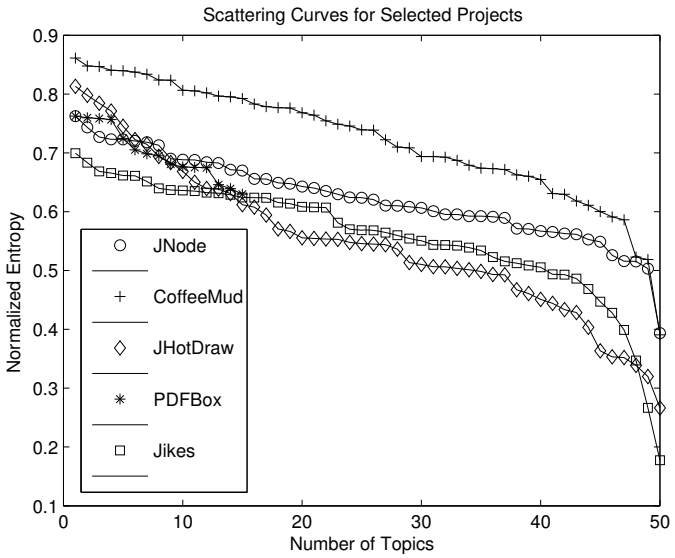

Figure 2. Scattering Curves for Selected Projects

Table 8. Example Tangling Results for Full Repository.

File	Entropy
org/openharmonise/rm/commands/CmdGenerateReport.java	0.8258
it/businesslogic/ireport/gui/ReportQueryDialog.java	0.7885
mail/core/org/columba/mail/imap/IMAPServer.java	0.7881
jRivetFramework/webBoltOns/ReportWriter.java	0.7869
org/lnicholls/galleon/apps/musicOrganizer/MusicOrganizer.java	0.7664
doctorj-5.0.0/org/incava/java/ASTNestedClassDeclaration.java	0.3379
nfop/fo/properties/FontSelectionStrategy.java	0.2275
net/sf/farrago/namespace/jdbc/MedJdbcColumnSet.java	0.2275
com/planet_ink/coffee_mud/Exits/Door.java	0.0
buoy/event/FocusLostEvent.java	0.0

JHotDraw, two of the files presented, BouncingDrawing and URLTool, correspond to sample applications bundled with the software. Because these sample applications exercise multiple JHotDraw capabilities, it is not surprising that they are associated with multiple topics, and are thus assigned a high entropy score. The same is true of SingleFigureEnumeratorTest, which represents unit tests to exercise and validate several project features. Like the results for the full repository, files with low tangling in JHotDraw correspond to very specific units of functionality, such as exceptions and handlers for events and GUI interaction. Similar observations can be made for Jikes, with files implementing complex functionalities such as debugging and process management coming in with high entropy scores in the range of .6736-.6932, while files implementing small or specific functionalities such as constant definition or timeout exceptions are measured to have normalized entropy in the range of 0.0-.0693. Indeed, closer inspection of VM_Contstants.java reveals it is assigned to only 1 topic: 'bytes default fail constants option'. Full results for all projects are available from the supplementary materials page, and can be interpreted in the same manner as the examples presented here.

Table 9. Example Tangling Results for JHotDraw.

File	Entropy
BouncingDrawing.java	0.6650
SingleFigureEnumeratorTest.java	0.6538
URLTool.java	0.6449
UndoRedoActivity.java	0.1000
CommandCheckBoxMenuItem.java	0.0892
JHotDrawException.java	0.0831

In addition to analyzing tangling of software via the inspection of individual files, it is also possible to succinctly summarize tangling behavior visually. Figures 3 and 4 contain the tangling curves for the the individual projects; Figure 5 shows the tangling for the entire repository. The curves are produced by sorting the tangling entropy values for each file in descending order, and then plotting the values directly.

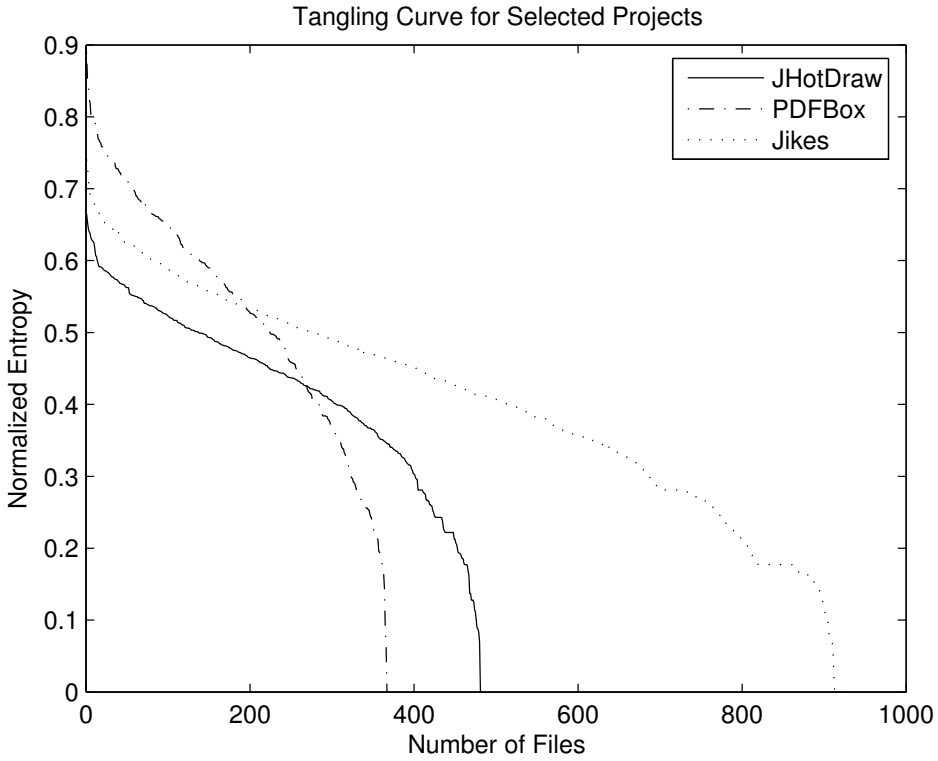

Figure 3. Tangling Curves for the 3 Smaller Projects

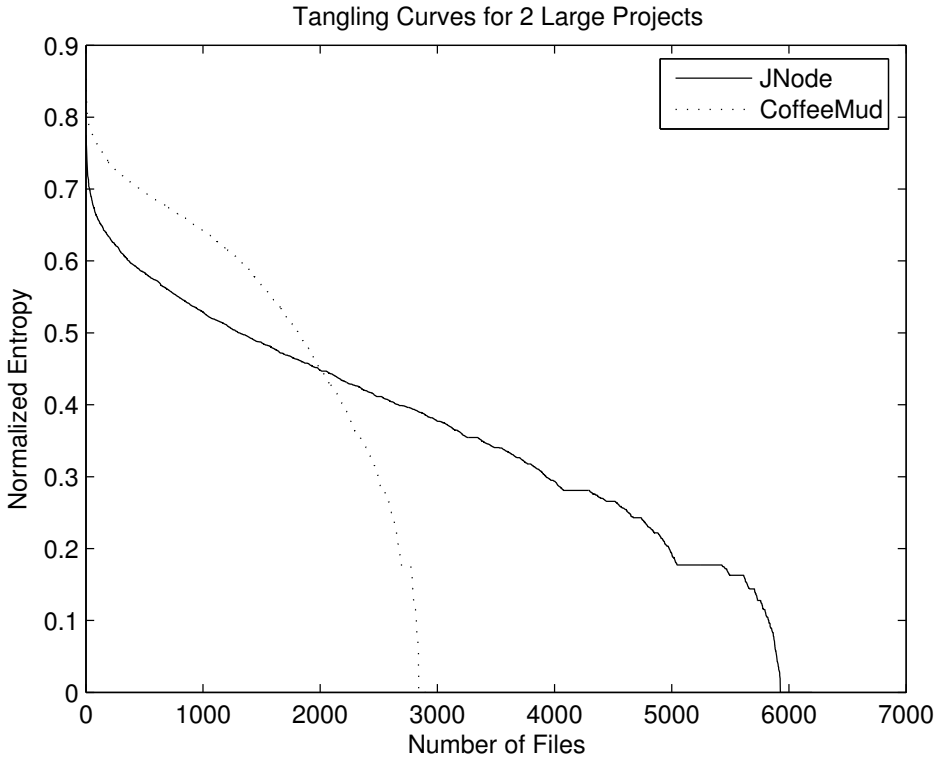

Figure 4. Tangling Curves for the 2 Larger Projects

Table 10. Example Tangling Results for Jikes.

File	Entropy
DebugerThread.java	0.6932
TraceBuffer.java	0.6845
VM_Process.java	0.6736
VM_Listener.java	0.0693
PPC_Disassembler.java	0.0554
VM_Contstants.java	0.0

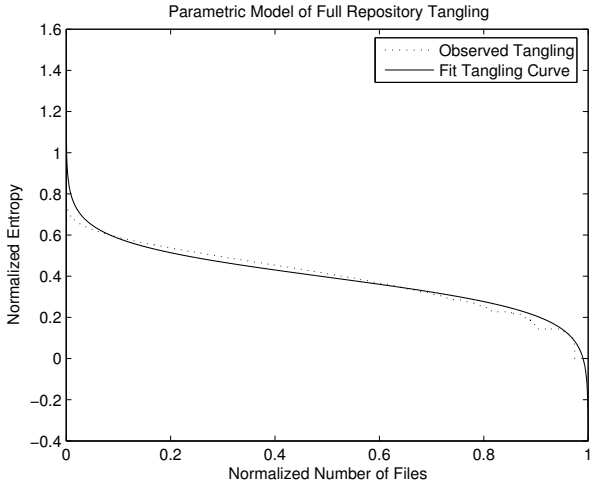

Figure 5. Fit of Parameterized Model to Full Repository Tangling Curve

In addition to understanding the high-level "picture" of tangling behavior, such curves are also useful for identifying areas of interest within tangling, allowing one to focus on groups of files with particularly high or low tangling.

From the figures one also notices that tangling curves follow an inverted "S" shape, where the S shape is commonly associated with the sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

We discuss this observation next, where we leverage the functional form of the curves to build a parametric model for tangling.

4.7 A Parameterized Model of Tangling

When examining Figures 3, 4, and 5 one is immediately struck by the fact that the tangling curves, without exception, follow inverse sigmoidal behavior. Such knowledge can be used to construct a parameterized model of project tangling, and if the model is accurate, allows the tangling behavior of software to be characterized using only a few parameters. To test this hypothesis we constructed a simple two parameter model of tangling of the form

$$f(x) = \alpha * \ln((1/x) - 1) + \beta$$

Table 11. Estimated Parameter Values for Software Tangling

Project	α	β	R-Square
JHotDraw	.06703	.4201	.9485
Jikes	.08395	.4015	.9556
PdfBox	.09231	.5248	.9557
JNode	.08843	.3599	.9630
CoffeeMud	.1024	.5176	.8985
Full Repository	.08564	.3956	.9624
Mean	.08662	.43808	.94728
Standard Dev	.01163	.06759	.024481

where α and β are the function parameters that are to be fit to the model.

Using nonlinear optimization with least squares we estimated the tangling parameters for 5 individual projects, as well as our full repository. Table 11 presents the estimated parameters, as well as the R-Square values for the fit curves. From the table we see that the values of α and β are reasonably concentrated across projects. More telling are the R-Square values, which indicate that, on average, 94.7% of the variation in software tangling can be explained by the model. This is demonstrated visually in Figure 5, where the observed tangling curve of the full repository is compared to the tangling curve produced by inverted sigmoid model.

5. Comparison to Other Aspect Mining Techniques

To further validate the approach of identifying aspects through LDA-based topic modeling, the results obtained for the JHotDraw project are compared with previous results for the same project presented in the literature. JHotDraw is a good choice because it is widely used in the aspect mining literature and includes robust and verifiable implementations of various features that makes it suitable for this kind of analysis (Kellens et al. 2007; Robillard and Murphy 2007; Ceccato et al. 2005; Zhang and Jacobsen 2007; Canfora and Cerulo 2005).

Since existing techniques have various underlying assumptions about what aspects or concerns are and how to mine them, there is not a consistent benchmark of aspects to be compared, even for the same project. Results differ from one approach to another. Thus, we collect a handful of candidate aspects that have been either manually mapped to the JHotDraw implementation or automatically found by various aspect mining techniques and compare them in three ways: (i) whether we can find similar topics in our results, (ii) whether we observe similar degrees of scattering; and (iii) whether we observe similar degrees of tangling. For the last two, we look into scattering and tangling behavior that has been identified before in JHotDraw.

File Names (Rows) \ Tagged Topics (Columns)	COMMAND, menu	CONTENT	DISPLAY	DRAWING Connectors	DRAWING HANDLE, Invocation	DRAWING Lines	DRAWING Poly	DRAWING Rect	DRAWING, Figure Attributes	DRAWING, Figure Selection	DRAWING, Figures, Decorator	DRAWING, Lines, Decorator	DRAWING, Observer	DRAWING, Rendering	DRAWING, Shapes, Geometry	DRAWING. Color	DRAWING. Point, moving	Events DnD UI	Events, DRAWING Figure UI	Events, Mouse, Tool	Events, UI Mouse changes	Events, UI Observable	Events, UI Observer	EXCEPTION HANDLING	FACTORY	HASTABLE, Math	IMAGE, Load Register	ITERATION	LAYOUT	MAP	PERSISTENCE, Read-write	RESOURCE Mgmt, Strategy	ROTATION	STORAGE FORMAT	TEXT, Fonts	TEXT, Typing	TOOL, APPLET, url	TOOL, Editor, Animation	TOOL, Menu	TOOL, Palette	TOOL, TEST	UI, Decorator	UI, Dimensions	UNDO	VERSION CHECK	VISITOR	*ENTROPY*
/util/UndoRedoActivity.java	0.01																	0.01																											0.92	0.01	0.10
/util/Undoable.java											0.05		0.11																															0.84		0.14
/standard/CutCommand.java	0.01			0.01						0.05	0.01		0.01															0.12															0.67	0.01	0.33
/standard/PasteCommand.java	0.03		0.03			0.03				0.01		0.02	0.06	0.01														0.01															0.66	0.03	0.36
/standard/DeleteCommand.java	0.09		0.02		0.01	0.01							0.05					0.01	0.02			0.02					0.07	0.01															0.66		0.35
/figures/UngroupCommand.java	0.03								0.14	0.03	0.03		0.10																															0.60	0.01	0.37
/util/UndoableAdapter.java									0.01	0.03	0.02		0.06			0.01												0.20															0.58		0.37
/standard/DuplicateCommand.java	0.12										0.02	0.02	0.09																															0.58		0.39
/figures/GroupCommand.java	0.02	0.01	0.01						0.05	0.01	0.02		0.09			0.01		0.01	0.02						0.01			0.12															0.54		0.47
/standard/SendToBackCommand.java	0.02								0.01	0.01	0.22		0.11																															0.53		0.38
/standard/SelectAllCommand.java	0.07								0.10				0.14																															0.52	0.01	0.43
/standard/ChangeAttributeCommand.java	0.01	0.01							0.34	0.01			0.02		0.01				0.03						0.01			0.01															0.51		0.35
/standard/FigureTransferCommand.java									0.05	0.08	0.11								0.05			0.03						0.01															0.60		0.46
/util/UndoableCommand.java	0.50								0.50																0.50			0.50															0.50		0.16
/figures/BorderTool.java				0.01	0.01		0.01		0.02	0.01	0.01		0.08							0.07		0.01	0.01		0.03			0.03															0.49		0.50
/standard/BringToFrontCommand.java	0.11			0.02			0.01		0.02	0.05	0.02	0.01	0.08									0.04	0.01	0.01	0.02			0.02															0.47		0.45
/standard/DragTracker.java									0.01	0.03	0.09		0.16										0.01					0.01															0.45		0.35
/figures/InsetImageCommand.java	0.04										0.09		0.09				0.17		0.01	0.30						0.32															0.38		0.42
/standard/AlignCommand.java	0.01		0.43				0.01	0.01			0.01		0.22			0.03	0.01	0.02		0.01	0.01	0.01																			0.38		0.39
/contrib/dnd/JHDDragSourceListener.java		0.01	0.01			0.00												0.42	0.02					0.01	0.01			0.01						0.00	0.01	0.04							0.01	0.47		0.42
/util/RedoCommand.java	0.18										0.02		0.03												0.00			0.00															0.36		0.45
/util/UndoCommand.java	0.12					0.03				0.06			0.24																															0.35		0.45
/util/collections/jdk11/IteratorWrapper.java											0.07		0.21																															0.32		0.52
/util/UndoManager.java					0.01								0.03																															0.29		0.47
/contrib/PolygonScaleHandle.java			0.02	0.01	0.11		0.30						0.01	0.01	0.08													0.62															0.28		0.51
/contrib/dnd/JHDDropTargetListener.java		0.01								0.00	0.03		0.03												0.00																			0.24	0.02	0.34
/contrib/TriangleRotationHandle.java				0.01	0.12		0.01			0.03			0.03															0.01															0.22	0.01	0.54
/figures/RadiusHandle.java			0.06			0.35	0.01						0.07	0.06		0.01	0.05																											0.22		0.47
/figures/ScribbleTool.java		0.01							0.04	0.10			0.02				0.04			0.54	0.01																							0.21	0.01	0.37
/figures/ConnectedTextTool.java				0.01							0.18		0.12									0.02																									0.41

Figure 6. Concern Mapping for JHotDraw Using our LDA-Based Approach.

To perform these comparisons we inspect the document-topic matrix for JHotDraw that provides us with the probability distribution of topics over files. For each latent topic identified by a set of 5 words, we assign a meaningful concern name. In some cases the terms in the sets of words are good enough to identify the concerns. For others, we consult the documentation of JHotDraw and association of the topics with the files to determine a meaningful name. While naming these concerns we followed a convention of appending each one with a more general term, if possible.

Figure 6 shows a portion of the document-topic matrix. It includes all the latent topics and corresponding concern names for JHotDraw, except those that were identified as belonging to testing features. The first row shows the sets of words (each topic in a column), and the second row shows the concerns that were identified from the topics. The first column from the third row onwards is the list of all the files, with only a portion shown here to save space. The rest of the cells in the matrix contain numbers that indicate the probability of a topic (from a column) belonging to an individual file (in a row). Empty cells denote zero probability. The last column lists the entropy for each document that measures the degree of tangling. It is easy to see that with this representation one can get an intuition about scattering and tangling.

To observe the degree of scattering one can inspect all the documents that would be assigned to a particular concern. A reasonable threshold can be adopted to exclude documents with lower probabilities. To observe tangling, one can look across the row to see what other topics also belong to the same file. Based on probabilities one can make an educated guess on whether the multiple concerns might be interacting with each other. We now discuss some of the existing concerns that have been noted in the literature that were also identified in our results.

- *Finding Aspects:* Aspects mined from JHotDraw, for the most part, fall into two categories: project-specific features such as those that deal with manipulating figures, and general-purpose aspects such as design patterns. In Table 12 we list most of those that were listed by other authors (Robillard and Murphy 2007; Ceccato et al. 2005; Zhang and Jacobsen 2007; Canfora and Cerulo 2005). The first column gives the aspects that were mined. The second column lists the techniques that have been applied to mine the aspect given in the first column. The following abbreviations are used for the techniques: IA = Identifier-analysis, FA = Fan-In Analysis, D = Dynamic Analysis, M = Manual, and R = Mining Code Revisions. The third column provides a similar concern that we identified with our technique (also shown in Figure 6).

  This result shows that the automated topic modeling based approach is equally good in mining identified aspects from a well-known and studied sample project. Among all the aspects that we could find in the studies we looked at, only the following were not explicitly identified by our approach: consistent behavior, contract enforcement, and composite design pattern. The first two are more of a specification for a concern rather than a concern itself and could only be detected properly with a technique that had an earlier assumption about the program structure. Our technique does not look for any prespecified structure in the source code, so as not to inject bias toward what aspects are. Unless design patterns manifest themselves with meaningful names, any text based mining, like ours, will dismiss them.

- *Degree of scattering among some common concerns:* The technique in (Zhang and Jacobsen 2007) ranked three common aspect-candidates in descending order of measured crosscutting: Persistence $>$ Undo $>$ Figure Selection. When we rank the corresponding concerns from our results, one obtains exactly the same ordering: PERSISTENCE (0.65) $>$ UNDO (0.63) $>$ DRAWING (Figure Selection) (0.46), with the values inside the parenthesis denoting entropies. While this is just a single result, the similarity of the results obtained with these two completely different approaches is striking and attests to the potential of the topic modeling approach.

- *Traces of Tangling:* Some notable instances of tangling that have been discussed in JHotDraw are: tangling of Undo and Command concerns; tangling of UI (user interface); Storage Management and Writing; tangling of change notification; and subject registration in Observer concerns (Robillard and Murphy 2007; Canfora and Cerulo 2005). We noted similar tangling of concerns in our results. Figure 6 shows that while looking at the UNDO concern, tangling of COMMAND, DRAWING, ITERATION, and VISITOR concerns are likely. Looking at the complete concern map (not shown), concerns like STORAGE FORMAT, DRAWING and CONTENT appear likely to be tangled with the PERSISTENCE concern.

The observations above provide confidence that the automated topic modelling approach for identifying aspects and quantifying scattering and tangling is consistent with the results obtained by prior approaches to aspect identification. However the topic modelling approach extends the previous approaches and overcomes some of their conceptual deficiencies. Most previous approaches introduce a somewhat circular definition of aspects by focusing on specific program or execution structures (Dynamic Analysis, Fan-In/Fan-Out, Code Revisions). By saying, for example, that "an aspect is a program/design element with high fan-in," fan-in analysis immediately introduces a circularity related to the representation of the program/design, and the specific programming/design technologies used. In our view, software concerns exist at a higher-level than the program representation, so a good theoretical framework for them must

Table 12. Comparative Analysis of Aspects in JHotDraw. IA = Identifier Analysis, FA = Fan-In Analysis, DA = Dynamic Analysis, M = Manual, R = Mining Code Revisions

Aspects/Concerns from other techniques	Technique used	# Concerns identified with our technique
Loading Registers	IA	IMAGE, Load Register
Manipulating Figures (drawing, moving, connecting)	IA, DA	DRAWING
Managing Views	DA	LAYOUT, UI Decorator, UI Dimensions
Adding Text	DA	TEXT Typing
Add URL to figure	DA	TOOL Applet URL
Persistence	IA, FA, DA, R, M	PERSISTENCE Read-Write
Storage Management	M	STORAGE FORMAT Getting/Setting attributes
Getting and Setting attributes	DA	DRAWING Figure Attributes
Command Execution	FA, M	COMMAND menu
Undo operation	IA, FA, DA, R	UNDO
Iterating over collection	IA,	ITERATION
Manage Handles	DA	DRAWING Handle Invocation
Observer	IA, FA, R	Events, UI Observer
Decorator	FA	Drawing Figure Decorator, Drawing Line Decorator
Visitor	IA	VISITOR
Adapter	FA	Test Adapter
User Interface	M	Display, UI

exist above concrete programming/design technologies and tools.

Identifier analysis (IA), the closest to our approach, does not make such assumptions. However, being based on formal concept analysis, it has several limitations. Because the LDA-based approach models program entities as mixtures of concepts (topics) which are themselves mixtures of words, it can capture more subtle statistical relationships among topics, words, and entities, with demonstrated benefits in concern identification.

6. Implications of the Theory

This study is mainly descriptive; it describes empirically verified properties of real-world software programs, at both large and small granularity scales. Software research is usually prescriptive, aiming to provide solutions for perceived problems. As such, we make a first attempt at laying out the implications of our observations for the development of programming models and the design of software tools. These comments are intentionally generic, as the goal is not to advocate any particular solution, but to point out the large research and design space that is still open related to software concerns. We note that, at this point, the issue of whether excessive scattering and tangling are "bad" (hypothesis 2 in the Introduction) is still an open question, although there is some evidence that it is so (Eaddy et al. 2008).

At first glance, based on the empirical validation of our machine learning techniques, it would appear that one can only identify the high-level software concerns, and therefore the aspects, *after* the programs are written. This is not necessarily the case, as explained next.

As shown in Section 4.3, the study of a very large repository of Java programs shows that there are about two dozen high-level, highly crosscutting software concerns that are likely to affect any new Java project. Without any special new tools, developers can use this knowledge to keep those crosscutting concerns under a tight control by being careful with naming and coding conventions. Our results at this very large scale explain the strong intuitions driving AOP from the beginning towards aspects such as exception handling, authentication, persistency, and so forth.

As for project-specific concerns, there is no absolute reference about what is and what is not crosscutting, as there are no a-priori aspects; there are only project-specific concerns. According to the theory presented here, crosscutting is not a binary quantity, but instead a continuous value of entropy between 0 an 1. Depending on how the developers specify and design their software, these concerns may become more or less scattered and tangled with each other and with the general-purpose concerns in the program representations.[2] So, for these, crosscutting can only be measured after the fact as an emergent property of the representations and the development process itself. If a concern becomes scattered, developers can rethink its representation (at whatever level they are working) and refactor it.

Our results show that project-specific concerns tend to be less scattered than the general-purpose ones, possibly because developers do a good job at modularizing them in their designs. But, as seen in CoffeeMud and others, it is also quite possible to scatter them widely.

The technique described in this paper can be integrated into a wide variety of development tools, such as an IDE, in order to both track the vocabulary used by the develop-

[2] By "program representation" we mean any representation, from formal specifications to code.

ers, and measure scattering and tangling as the project progresses. A project whose latent topics look mostly like noise is likely to have serious problems in terms of vocabulary choices among developers, which in turn is likely to cause misunderstandings and, consequently, bugs. A project with well-identified project-specific topics, but with high levels of scattering of those topics, is likely to require refactoring, or at least rethinking.

The results for the entire repository raise some questions for existing AOP solutions. If all of those software topics are scattered, AOP solutions should support them all in equal manner; some of them do. For example, AspectJ, being so general-purpose, provides support for designing string and list manipulation as separate modules. That, however, produces odd designs, with most of the inner core of objects pulled out from them and programmed in reverse-style; that is rarely done. But if that would be odd, the question is raised about the reason for doing that for concerns such as logging and concurrency. At the very least, that practice, advocated by AOP from early on, requires a better justification.

More importantly, though, if concerns are topics that have to be woven/coordinated together in the software representations, somehow, we can use this knowledge to forge improved development methodologies and tools to support this inevitable process that underlies software construction. As such, the work presented here provides a theoretical foundation and a mean for modeling aspects early in the software development life cycle, and supports work (Clarke and Baniassad 2005; Baniassad et al. 2006) that is fundamentally grounded on the assumption that crosscutting concerns exist above the representations, and thus there should be a support for their early modeling.

To validate our methodology, we focused only on Java programs, because the vast majority of practical AOP work has been done in Java. While an argument can be made that the same theory applies to software written in other programming languages and other representations (e.g. UML), strictly speaking, the results should not be extrapolated at this point. Further empirical analysis of projects using other representations must be done in order to find out how well the proposed definitions of concerns and aspects applies to them.

7. Related Work

This work builds on a large body of literature and research in software engineering and data mining.

The concept of **software concern** has been around for a long time, taking different names and slightly different flavors over the years, and leading to the concept of *Aspects*, as crosscutting concerns. An early analysis of concerns was conducted by Biggerstaff (Biggerstaff 1989; Biggerstaff et al. 1993), who proposed the idea of *concept assignment* in the context of program understanding for reuse. This work identified the gap between the source code and the human-level concepts, suggesting the need for tools that help bridge this gap. The identification of this problem gap was an important contribution, but the the specific solution proposed was limited, and focused on C programs: it consisted of an early call for model-driven development which, the focus being C, was also an early call for OOP. Those papers did, however, stress the importance of human-defined identifiers ("natural language tokens") and other informal information such that of comments. As such, this early work continues to be used by more recent work, and is still a major reference for the theory of aspects.

Most research reports on AOP assessment tend to focus on the third hypothesis (i.e. AspectJ). These studies have used analytical argumentation (Kiczales et al. 1997; Hannemann and Kiczales 2002), case-study methodologies (Lippert and Lopes 2000; Kienzle and Guerraoui 2002; Lopes and Bajracharya 2006; Kulesza et al. 2006), comparative analyses of very small collections of existing systems(Garcia et al. 2005; Cacho et al. 2006; Filho et al. 2006), and small user studies (Murphy et al. 1999; Walker et al. 1999). The small scale and controlled conditions of these studies make them prone to subjective interpretations and overfitting. In any case, the focus of this paper is not AspectJ, or any other AOP tool in particular, but the concept of *aspect* itself. Thus the work most closely related to this paper is the recent work on software and aspect mining discussed below.

7.1 Topic Modeling of Source Code

Formal concept analysis (FCA) is an unsupervised clustering algorithm used to group together objects with a shared set of attributes, and similarly to identify the set of attributes shared by objects within a cluster (Ganter and Wille 1999). FCA has been used to identify concepts in software. Attributes corresponding to code features of interest are manually defined by the user beforehand. Unlike LDA, FCA does not operate within a probabilistic framework, but rather uses lattice theory to construct groups of conceptually similar software artifacts. The presence or absence of attributes alone drives the concept clustering, with no formal models of uncertainty, likelihood, or prior knowledge being leveraged. The result is that an entity belongs to a concept cluster or it doesn't, as opposed to LDA, where an entity belongs to all concepts, but with varying degrees of belief. Identifier analysis (IA) can be used in conjunction with FCA to attempt to cluster program entities. In this case IA is used to represent a program entity as a collection of tokens derived from entity name or text, and it is these tokens that serve as the attributes upon which the formal concept analysis is based. Ultimately the concept clusters produced are based on the sharing of identifier tokens among program entities. However, unlike the approach presented here, FCA/IA is not a mixture model. Because the LDA-based approach models program entities as mixtures of concepts (topics) which are themselves mixtures of words, we can capture more subtle statistical relationships among topics, words, and entities.

Moreover, because this mixture model is probabilistic, we can apply information theory directly to the problem of scattering and tangling quantification.

The idea of modeling source code with topics has been proposed before. For example, code topic classification has been explored using support vector machines (Ugurel et al. 2002), but the technique is significantly different from ours. Firstly, a training set must be manually partitioned into categories based on project metadata. Topics, consisting of commonly occurring keywords, are extracted for each category and used to form features on which to train the model. The training and testing set comprise only 100 and 30 projects respectively. Marcus et al., in the line of Biggerstaff's work, use latent semantic analysis (LSA) to locate concepts in code (Marcus et al. 2004). The goal is to enhance software maintainability by easily identifying related pieces of code in a software product that work together to provide a specific functionality (concept), but may be described with different keywords (synonyms, etc). In this sense the work shares some of our goals, but does not consider the problem of automatically extracting topic distributions from arbitrary amounts of source code. Progress was made in this area through the application of LSA to software repositories in order to cluster related software artifacts (Kuhn et al. 2006). However, new approaches for defining topic scattering and document tangling were not considered. Latent Dirichlet Allocation has been previously applied to log traces of program execution, providing a framework for statistical debugging (Andrzejewski et al. 2007).

Closely related to our own work is the MUDABlue system, which explicitly considers the need for unsupervised categorization techniques of source code, and develops such a technique as a basis for software clustering with the aim of information sharing (Kawaguchi et al. 2004). Unlike our work, however, MUDABlue utilizes LSA rather than a probabilistic framework. Furthermore, their assessment was also done at a much smaller scale than ours and considered only 41 projects.

Minto and Murphy have proposed a technique for mining developer expertise to assist in bug fixes (Minto and Murphy 2007). While expertise is related to the general idea of topics, the approach is substantially different, relying on author and file update metadata of the configuration management system rather than source code directly. Additionally, the approach is validated on only 3 software projects rather than the thousands considered in this paper.

Recently, we have applied author-topic models to source code to extract developer contributions from a subset of files of the Eclipse 3.0 codebase (Linstead et al. 2007), but this work did not consider the analysis of crosscutting concerns as addressed in this paper. We later expanded the scope of topic modeling techniques to multi-project repositories, as well as presented a preliminary statistical analysis of a large software repository (Linstead et al. 2008).

7.2 Aspect Mining and Modeling

As mentioned before several techniques and methodologies exist for aspect mining. In general aspect mining consists of identification, mapping and use of metrics for aspects (Eaddy et al. 2007). Identification deals with discovering aspects in existing software. This is done either manually (Robillard and Murphy 2002; Griswold et al. 2001; Hannemann and Kiczales 2001) or (semi) automatically (Marin et al. 2004; Zhang and Jacobsen 2007; Kellens et al. 2007). Mapping deals with associating identified aspects or concerns with the modules in implementation, usually elements of source code such as classes or methods (Robillard and Murphy 2002; Eaddy et al. 2008). The task of identification and mapping of aspects is explicit when the underlying aspect-mining method clearly differentiates a concern model from the implementation. These two tasks seem to get intermixed when there is no such explicit distinction between the concern model and the implementation. Metrics used in aspect mining usually deal with measures of crosscutting, scattering, and tangling. These metrics are used in two different ways: (1) to measure the precision and accuracy of the aspect mining technique in capturing crosscutting concerns (Cojocar and Şerban 2007); and (2) to derive various conclusions regarding the effect of crosscutting, by correlating them with selected quality attributes for software (Eaddy et al. 2008).

Kellens et al. offer a detailed survey of seven different code-based aspect mining techniques (Kellens et al. 2007). Since all techniques surveyed are code-centric they do not bring out the distinction between identification and mapping tasks during aspect mining. Their comparison framework implicitly assumes that there exists such a mapping of aspects to source code elements. Our work presented in this paper has focused on automatic discovery of topics in code with measures of scattering and tangling. Thus, these topics are possibly aspects manifested in the implementation. We can use Kellen et al.'s framework to highlight the features of our approach to compare it against existing approaches. Our approach can be characterized with the following list of attributes taken from their framework.

1. **Static vs. Dynamic Data:** Our approach is based on static data, no dynamic analysis is required.
2. **Token-Based vs. Structural/Behavioral Analysis:** The information used in topic modeling comes from a deep structural analysis of code, as well as lexical analysis of the names of entities extracted from the code elements.
3. **Granularity:** Our approach yields a probabilistic assignment of discovered topics to individual source code files.
4. **Tangling and Scattering:** Our technique yields precise measures for both tangling and scattering.
5. **User Involvement:** No manual input is required from the user in identification of the topics. However, human

judgement is needed to make sensible interpretation of the set of words that emerge out as latent topics.

6. **Largest System:** Our technique has been validated in systems of varying sizes, and at several scales. Our validation considered 4,632 Java projects, as well as over individual projects of varying sizes.
7. **Empirical Validation:** This is possibly the strongest component of our approach. We have conducted validation experiments at multiple granularity scales, from single projects to Internet-scale repositories, several order of magnitude beyond any previous experiments.
8. **Preconditions:** Our approach makes very little assumptions about what an aspect is; it simply relies on the assumption that developers choose reasonable names for their software elements. This is true for other approaches such as 'Identifier Analysis', 'Method Clustering','Language Clues' and 'Token-based clone detection'. While working on vocabulary selection we gained experience in improving topic quality by carefully expanding the vocabulary to include tokens from the JDK and method calls.

Manual approaches of mining aspects have the advantage of being controlled, and thus produce a highly accurate identification and mapping of concerns to the underlying implementation. This, however, can also be a drawback; the results and the methodology are prone to subjective interpretation and they are difficult to replicate, especially at large scale. Manual techniques are highly labor intensive, requiring hundreds of hours to analyze even a single project, which renders such methods inappropriate for general application.

Recent work in early aspects has employed automated techniques to mine aspects in non-code artifacts such as requirements (Duan and Cleland-Huang 2007; Kit et al. 2006). These techniques are similar in nature that they employ mostly text-mining techniques and bear some resemblance with the text-based aspect mining mentioned before.

8. Conclusion

We have presented an operational theory of aspects based on unsupervised, probabilistic topic modeling and on information theory. Our framework aims at providing a solid foundation for understanding what aspects are and where they come from. It can be summarized in the following sentence: aspects are latent topics with high scattering entropy. To identify latent topics, we use a technique known as Latent Dirichlet Allocation, a relatively recent statistical data mining technique that has been used very successfully in topic modeling for natural language texts. To be able to apply it to software in a meaningful way, we pre-process the words according to widely-used naming conventions, and choose to use only certain words of the source files that we know, by domain experience, directly contribute to the effective functional information of software. The result is a model of software concerns that directly maps to the concept of latent topics.

By using this probabilistic technique, we can then easily measure scattering and tangling by the entropies of the probability distributions of topics over files in the case of scattering, and files over topics in the case of tangling.

We have validated our model empirically at two scales: 1) on a very large data set consisting of 4,632 Java projects; and 2) on 5 Java projects of varying sizes. The results confirm the main AOP intuitions about the existence crosscutting concerns, and the possibility of measuring scattering and tangling. The crosscutting concerns identified at the very large scale include most of the aspects used as prototypical examples of crosscutting in the literature. Furthermore, we compared our results with several other aspect mining techniques, and found significant agreement, confirming that our mathematical model of aspects matches and extends the model that the community has been working with.

Acknowledgments

Work in part supported by National Science Foundation MRI grant EIA-0321390 and a Microsoft Faculty Research Award to PB, as well as National Science Foundation grant CCF-0347902 to CL and CCF-0725370 to CL and PB.

References

David Andrzejewski, Anne Mulhern, Ben Liblit, and Xiaojin Zhu. Statistical debugging using latent topic models. In Stan Matwin and Dunja Mladenic, editors, *18th European Conference on Machine Learning*, Warsaw, Poland, September 17–21 2007.

Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a search engine for open source code supporting structure-based search. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 681–682, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-491-X. doi: http://doi.acm.org/10.1145/1176617.1176671.

Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. A study of ranking schemes in Internet-scale code search. Technical report, UCI Institute for Software Research, 2007.

Elisa L. A. Baniassad, Paul C. Clements, João Araújo, Ana Moreira, Awais Rashid, and Bedir Tekinerdogan. Discovering early aspects. *IEEE Software*, 23(1):61–70, 2006. URL `http://doi.ieeecomputersociety.org/10.1109/MS.2006.8`.

Ted J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22(7):36–49, 1989.

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *ICSE '93: Proceedings of the 15th International Conference on Software Engineering*, pages 482–498, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. ISBN 0-89791-588-7.

David M. Blei, Andrew Y. Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*,

3:993–1022, January 2003. URL http://jmlr.csail.mit.edu/papers/v3/blei03a.html.

Silvia Breu. Extending dynamic aspect mining with static information. In *SCAM '05: Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM'05)*, pages 57–65, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2292-0. doi: http://dx.doi.org/10.1109/SCAM.2005.9.

Silvia Breu and Thomas Zimmermann. Mining aspects from version history. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 221–230, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2579-2.

M. Bruntink, A. van Deursen, R. van Engelen, T. Tourwe. On the use of clone detection for identifying crosscutting concern code. *IEEE Trans. Softw. Eng.*, 31(10):804–818, 2005. ISSN 0098-5589.

Nelio Cacho, Claudio Sant'Anna, Eduardo Figueiredo, Alessandro Garcia, Thais Batista, and Carlos Lucena. Composing design patterns: a scalability study of aspect-oriented programming. In *AOSD '06: Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, pages 109–121, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-300-X.

Gerardo Canfora and Luigi Cerulo. How crosscutting concerns evolve in jhotdraw. In *STEP '05: Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 65–73, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2639-X. doi: http://dx.doi.org/10.1109/STEP.2005.13.

Gerardo Canfora, Luigi Cerulo, and Massimiliano Di Penta. On the use of line co-change for identifying crosscutting concern code. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 213–222, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2354-4.

M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. A qualitative comparison of three aspect mining techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2254-8. doi: http://dx.doi.org/10.1109/WPC.2005.2.

Siobhàn Clarke and Elisa Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005. ISBN 0321246748.

Grigoreta Sofia Cojocar and Gabriela Şerban. On some criteria for comparing aspect mining techniques. In *LATE '07: Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution*, page 7, New York, NY, USA, 2007. ACM. ISBN 1-59593-655-4. doi: http://doi.acm.org/10.1145/1275672.1275679.

Chuan Duan and Jane Cleland-Huang. A clustering technique for early detection of dominant and recessive cross-cutting concerns. In *EARLYASPECTS '07: Proceedings of the Early Aspects at ICSE*, page 1, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2957-7. doi: http://dx.doi.org/10.1109/EARLYASPECTS.2007.1.

Marc Eaddy, Alfred Aho, and Gail C. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *ACoM '07: Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*, page 2, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2967-4. doi: http://dx.doi.org/10.1109/ACOM.2007.4.

Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred Aho. Do crosscutting concerns cause defects. IEEE Transactions on Software Engineering 2008.

Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhao, Alessandro Garcia, and Cecilia Mary F. Rubira. Exceptions and aspects: the devil is in the details. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 152–162, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-468-5.

B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 3–14, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-042-6.

William G. Griswold, Jimmy J. Yuan, and Yoshikiyo Kato. Exploiting the map metaphor in a tool for software evolution. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 265–274, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1050-7.

J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition of legacy code. In *Workshop Advanced Separation of Concerns, ICSE'01*, 2001.

Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-471-1.

Shinji Kawaguchi, Pankaj K. Garg, Makoto Matsushita, and Katsuro Inoue. Mudablue: An automatic categorization system for open source repositories. In *APSEC '04: Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, pages 184–193, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2245-9. doi: http://dx.doi.org/10.1109/APSEC.2004.69.

Andy Kellens, Kim Mens, and Paolo Tonella. A survey of automated code-level aspect mining techniques. In *Transactions on Aspect-Oriented Software Development IV*. Springer Berlin / Heidelberg, 2007. doi: 10.1007/978-3-540-77042-8_6.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In

ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming, pages 327–353, London, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4.

Jorg Kienzle and Rachid Guerraoui. AOP: Does it make sense? the case of concurrency and failures. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 37–61, London, UK, 2002. Springer-Verlag. ISBN 3-540-43759-2.

Lo Kwun Kit, Chan Kwun Man, and Elisa Baniassad. Isolating and relating concerns in requirements using latent semantic analysis. *SIGPLAN Not.*, 41(10):383–396, 2006. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1167515.1167506.

Adrian Kuhn, Stephane Ducasse, and Tudor Girba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 2006.

Uira Kulesza, Claudio Sant'Anna, Alessandro Garcia, Roberta Coelho, Arndt von Staa, and Carlos Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 223–233, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2354-4.

Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining eclipse developer contributions via author-topic models. *MSR 2007: Proceedings of the Fourth International Workshop on Mining Software Repositories*, 0:30, 2007a. doi: http://doi.ieeecomputersociety.org/10.1109/MSR.2007.20.

Erik Linstead, Paul Rigor, Sushil Bajracharya, Cristina Lopes, and Pierre Baldi. Mining internet-scale software repositories. *NIPS 2007: Advances in Neural Information Processing Systems 20*, 0, 2008.

Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *International Conference Software Engineering*. ACM Press, 2000.

Cristina Videira Lopes. AOP: A historical perspective (what's in a name?). In Robert Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, chapter 5, pages 97–122. Addison Wesley, 2004.

Cristina Videira Lopes and Sushil Krishna Bajracharya. Assessing aspect modularizations using design structure matrix and net option value. *Transactions on Aspect-Oriented Software Development*, 1:1–35, 2006.

Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 214–223, November 2004.

Marius Marin, Arie van Deursen, and Leon Moonen. Identifying aspects using fan-in analysis. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 132–141, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2243-2.

Shawn Minto and Gail C. Murphy. Recommending emergent teams. In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 5, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.27.

Gail Murphy, Robert Walker, and Elisa Baniassad. Evaluating emerging software development technologies: Lessons learned from assessing aspect-oriented programming. *IEEE Transactions on Software Engineering*, 25(4):435–455, 1999.

Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol.*, 16(1): 3, 2007. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/1189748.1189751.

M.R. Robillard and G.C. Murphy. Concern graphs: finding and describing concerns using structural program dependencies. In *ICSE 2002. Proceedings of the 24th International Conference on Software Engineering*, pages 406–416, 2002.

David Shepherd, Jeffrey Palm, Lori Pollock, and Mark Chu-Carroll. Timna: a framework for automatically combining aspect mining analyses. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 184–193, New York, NY, USA, 2005a. ACM Press. ISBN 1-59593-993-4.

David Shepherd, Lori Pollock, and Tom Tourwé. Using language clues to discover crosscutting concerns. In *MACS '05: Proceedings of the 2005 Workshop on Modeling and Analysis of Concerns in Software*, pages 1–6, New York, NY, USA, 2005b. ACM Press. ISBN 1-59593-119-8.

Paolo Tonella and Mariano Ceccato. Aspect mining through the formal concept analysis of execution traces. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 112–121, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2243-2.

Claire Tristram. Untangling code. MIT Technology Review: Ten Emerging technologies that will change the world, February 2001.

S. Ugurel, R. Krovetz, and C. L. Giles. What's the code?: automatic classification of source code archives. In *KDD '02: Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 632–638, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-567-X. doi: http://doi.acm.org/10.1145/775047.775141.

Robert Walker, Elisa Baniassad, and Gail Murphy. An initial assessment of aspect-oriented programming. In *International Conference Software Engineering*. IEEE Computer Society Press, 1999.

Carl Zetie. Aspect-oriented programming considered harmful. Forrester Research, April 2005.

Charles Zhang and Hans-Arno Jacobsen. Efficiently mining crosscutting concerns through random walks. In *Aspect-Oriented Software Development (AOSD'07)*, March 2007.