What is Middleware Made Of?

Exploring abstractions, concepts, and class names in modern middleware

François Taiani[1], Jackie Rice[2], Paul Rayson[1]
[1]Lancaster University (UK), [2]University of Lethbridge (Canada)
{f.taiani,p.rayson}@lancaster.ac.uk, j.rice@uleth.ca

ABSTRACT

Developing appropriate abstractions for distributed programming is one of the core aim of middleware research. Yet, analysing the impact, diffusion, and success of these abstractions in concrete middleware code is difficult and time consuming. In this paper we propose to use the constituting words found in program identifiers to explore the concepts used in popular middleware platforms. We study and compare four middleware products (JBoss, Hadoop, Axi2, and ActiveMQ), and show the existence of a substantial core of shared concepts that we think capture some of the key tenets of modern middleware engineering.

1. INTRODUCTION & MOTIVATION

One of the aims of middleware research is to find programming abstractions that make it easier to construct rich and complex distributed systems. Evaluating the success of these abstractions is however a complex and multi-faceted task. What are good middleware abstractions? Have abstractions proposed in the past worked? What has been their concrete impact on middleware products used in the field? Many researchers can provide intuitive answers to these questions, but we generally lack clear approaches to answer these questions in principled, systematic and reproducible ways.

Although questions on the impact of middleware abstractions are difficult to answer, they are critical to the field of middleware research: We need to know whether and how middleware abstractions are successful to inform and motivate future research, identify potential research gaps, and train future generations of researchers and practitioners. The need for principled approaches to evaluate the impact of middleware abstractions has led to early experimental works in that directions based on historical analysis [6], corpus linguistics [11], and cognitive analysis [9].

Although these methods have each been able to provide unique insights on middleware design and research, they are typically time-consuming and costly. They thus make it dif-

ficult to rapidly compare and contrast a broad range of different middleware systems. In this paper, we therefore propose a different strategy to investigate abstraction impact that is based on an analysis of program identifiers. Our hypothesis is that the concerns, mechanisms, and technologies used in a piece of middleware are (in part) reflected in the names given to its constituing elements (packages, classes, methods). Identifiers are however often highly specific (e.g. CMPClusteredInMemoryPersistenceManager) and difficult to generalise from. We therefore follow a growing trend in experimental software engineering [5, 3, 7, 8] and focus instead of the constituting words of identifiers. These words usually denote *concepts* that are central to the design or organisation of a piece middleware, and are generic enough to allow comparison across multiple middleware products.

We consider four popular open-source middleware products (JBoss, Hadoop, Axis2 and ActiveMQ) and conduct an explorative study of the concepts contained in their class and interface names. We investigate in particular to which extent this projects tend to share concepts, and what relationships can be found between these concepts. Finally we propose a rapid analysis of the Hadoop projects based on our findings.

2. METHODOLOGY

In this preliminary work we focus solely on the names of public Java classes and interfaces, but the same approach would be applicable to other languages and program elements. For brevity's sake, we will often use 'class' in the rest of the paper to mean both public Java classes and interfaces.

2.1 Concept extraction

The first step consists in extracting the elements of a compound name. We rely here on a set of simple regular expressions capturing the Camel case commonly used in Java projects. To handle specific terms such as *J2EE*—which would be wrongly parsed as *J*, *2*, *EE*—we also use a greedy dictionary approach [4] based on a small ad hoc dictionary of problematic terms (e.g. *JBoss* because it does not follow camel case convention, or *J2EE* because it combines capitals and numbers). For each class name c we obtain a sequence of words which we note concepts(c), e.g. concepts(c) = [*CMP, Clustered, In, Memory, Persistence, Manager*]. We consider each word refers to a concept, and use the two terms (*word* and *concept*) interchangeably in the rest of the paper.

2.2 Metrics

In order to rank and compare the concepts extracted from a corpus of class names $\mathcal{C}$ we use two metrics. The first one simply counts how many class names contain a concept w:

$$\mathsf{class\_count}(w) = |\{c \in \mathcal{C} : w \in \mathsf{concepts}(c)\}|$$

Our second metrics refines the first by taking into account the size of each class' code, but giving more weight to concepts appearing in larger classes:

$$\mathsf{locs\_count}(w) = \sum_{c \in \mathcal{C} \wedge w \in \mathsf{concepts}(c)} \mathsf{locs\_size}(c)$$

2.3 Inter-concept analysis

To analyse how concepts tend to appear together, we use two tools: n-grams, a common tool taken from linguistics, and *conceptual graphs*, which we describe just below. n-grams [2] capture collocations, i.e. words repeatedly appearing together. An n-gram model essentially counts how often particular tuples of words (pairs, triplets, etc.) appear together. Frequent tuples often represent a specific meaning, beyond the meaning of the individual words. For instance the 3-gram *blue sky research* appears much more often in English texts than *brown sky research* because it denotes a well delineated meaning, which the other does not.

To visualise inter-concepts relationships, we use a simple graph representation that builds on the sequences of concepts formed by each class names. A sequence s, e.g. $s = [Socket, Impl]$, is a simple graph $G(s)$ with two nodes and one directed edge:
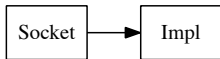
$$\boxed{Socket} \longrightarrow \boxed{Impl}$$

$$G(s) = \Big(V(s) = \{Socket, Impl\}, E(s) = \big\{(Socket, Impl)\big\} \Big)$$

The *conceptual graph* of a corpus of class names $\mathcal{C}$ is simply defined of the union of the graphs representing its concept sequences:

$$G(\mathcal{C}) = \Big(\bigcup_{c \in \mathcal{C}} E\big(\mathsf{concepts}(c)\big), \bigcup_{c \in \mathcal{C}} V\big(\mathsf{concepts}(c)\big) \Big)$$

3. RESULTS

In this study we consider four open-source middleware projects that are representative of the wide-range of available Java-based middleware. These are the J2EE application server JBoss from RedHat, the distributed map-reduce engine Hadoop, the Web Service engine Axis2/Java, and the message-oriented middleware ActiveMQ, with all last three projects hosted by the Apache Foundation. The four projects all aim to support the development of distributed systems, but differ in the services and technologies they provide. They are used in production, and generally follow good and clearly documented software development practices. They remain diverse, however, and although three (Hadoop, Axis2, and ActiveMQ) belong to the Apache Foundation, they are managed and developed by distinct teams, in independent projects, under their own architecture and coding conventions.

Each project is substantial, containing between 247k and 345k lines of Java code (LoCs), and between 1514 and 2990 public Java classes and interfaces (ignoring comments, test and sample code, Table 1).

Table 1: The four middleware projects considered

middleware	LoCs	# classes
JBoss AS (6.0.0)	345,063	2,290
Hadoop (1.0.3)	255,563	1,327
Axis2 (1.6.2)	302,315	1,514
ActiveMQ (5.6.0)	247,801	1,873
Total	**1,150,742**	**7,004**

3.1 Vocabulary size and distribution

Table 2 shows the number of class concepts, concepts for short, found in each project. These numbers make clear the benefit of concepts over class names to analyse projects: The projects use between 1.71 (Hadoop) and 3.21 (ActiveMQ) fewer concepts than they do class names, with the ratio getting close to 4 (1,861 concepts for 7,004 public classes) when considering all projects together.

Table 2: Size of concept vocabulary per project

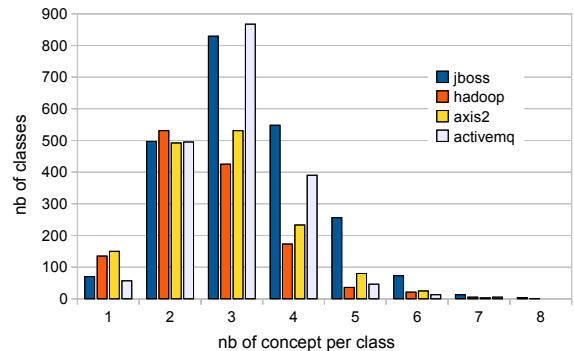| middleware	# concepts	per class (avg)	min	max
JBoss	949	3.31	1	8
Hadoop	776	2.65	1	8
Axis2	646	2.79	1	7
ActiveMQ	583	2.96	1	7
All projects	**1861**	**2.98**	**1**	**8**

Figure 1: Number of concepts per class name

Concepts are rarely used alone. They are instead the basic building blocks from which class names are formed (Figure 1). In all projects, the vast majority of classes (94.12%) use more than two concepts in their names, with more than half (65.35%) containing three concepts or more. This is again reflected in an averaged number of concepts per class of 2.98 (Table 2). This general trend hides however important differences between projects: Hadoop for instance stands out as a project with the lowest number of classes (1,327), but the second highest number of concepts (776). Although it has comparatively more concepts and fewer classes, Hadoop also has counter-intuitively the lowest average number of concepts per class (2.65, Table 2), and a minority of classes (49.81%) with three concepts or more.

In terms of relative frequencies, Figure 2 shows the frequency distribution of concepts in each individual projects, and globally. I.e. Figure 2 shows how many concepts (y-axis) appear in a particular number of classes (x-axis), on a
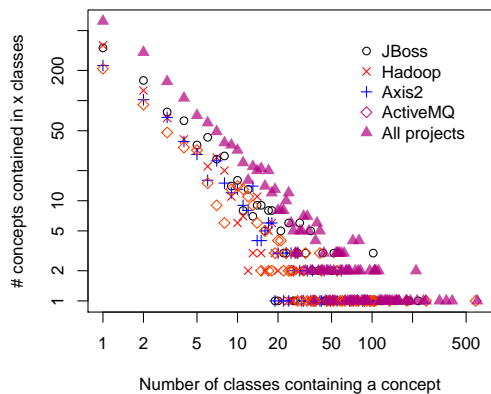
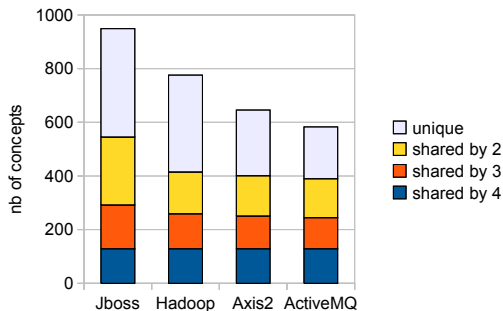**Figure 2: Concept frequencies among class names**

Figure 3: Concept corpus size and sharing

log-log scale. For instance, the top-left black circle indicates that JBoss contains 336 concepts (35.4%) which appear in exactly one JBoss class. Similarly, the top-left dark pink triangle means that across all projects 621 concepts (33.4%) appear in exactly one class. Figure 2 clearly shows how all distributions tend to follow power-laws, a typical features of both natural languages, and human created networks such as folksonomies, and social networks.

These observations suggest that the concepts that make up class names in middleware projects play a similar role to that of words in natural English sentences. As such, they represent a promising entry point to study the abstractions used in modern middleware products. In the following we focus more particularly on concepts shared between projects and ask the following questions: Do different middleware projects tend to share concepts? If yes, which ones? To which extent? What does can shared concepts tell us about individual projects and the state of current middleware practices?

3.2 Concept sharing between projects

Sharing comes in different degrees. At one extreme, concepts that are highly specific to one project (such as 'HA' for High-Availability in JBoss) are unique to this project and not shared. At the other extreme, highly generic concepts (e.g. 'Data') will tend to appear in all projects, with common but less generic concepts falling in-between. To investigate these different degrees of sharing, Figure 3 shows how many concepts are shared in each projects, depending on how many other projects share a particular concept. The

first observation is that all projects share a high proportion of their concepts (between 53.5% for Hadoop and 66.9% for ActiveMQ) with at least one other project. Size plays a role in the amount of common concepts: The smallest project (ActiveMQ, 186kloc) shares more with other projects (66.9% of its concepts) than the largest project (JBoss, 332kloc, 57.4% of shared concepts). Size is not, however, the only factor, as the numbers for Hadoop illustrate. Although Hadoop has a similar number of lines as Axis, and fewer classes, it also shows lower levels of concept sharing (53.5% for Hadoop, against 62.1% for Axis2).

What is the relative importance of these shared concepts? Although they represent a large proportion of the overall concept vocabulary, shared concepts might only appear in a minority of class names, with most classes using concepts unique to the project at hand. To answer this question, Table 3 looks at the number of classes that contain at least one shared concept in their name.

Table 3: Classes with shared concepts in their name

middleware	# shared concepts	# classes	% classes
JBoss	545	2,245	98.0%
Hadoop	415	1,211	91.3%
Axis2	401	1,431	94.5%
ActiveMQ	390	1,860	99.3%

Table 3 shows that shared concepts appear in almost all classes in all projects. As a corollary, very few classes only contain project-specific concepts: less than 1% in ActiveMQ, less than 2% in JBoss, less than 5% in Axis2. Although showing the same trend, Hadoop stands out again with 10% of classes containing no shared concept.

In we now turn to the percentage of classes that only contain shared concepts, the numbers are similarly high, with a majority of classes only containing shared concepts in JBoss (52.4%), Axis2 (56.2%) and ActiveMQ (58.4%), and almost half of classes doing the same in Hadoop (47%).

3.3 Sharing and semantic

The numbers we have just discussed demonstrate the existence of a *substantial shared vocabulary* between the four middleware projects we have selected. This shared vocabulary must be understood by anyone wishing to use and contribute to these projects. More generally it also exposes in a tangible way the existence of a shared foundation to distributed programming platforms.

What is this shared foundation made of? To shed light on this question, we now focus on those concepts shared by all four projects, meant to represent the most generic part of our concept corpus. Table 4 lists the first 15 most common concepts shared by all four projects, sorted according to the total number of lines of code (locs) they cover (first and second columns). For completeness, we also show the number of classes each concept appears in (third and fourth columns), and the percentages of locs they cover in each project (second half of the table). The 'rank' column in the middle indicates the overall rank of each concept (sorted by locs) among all other concepts, whether shared or not. Finally the last column ('CV') reports the *coefficient of variation* of the locs percentages across the projects. This coefficient measures the *dispersion* of each concept: low numbers indi-

Table 4: The top 15 concepts (in locs) shared by all projects

Concept	locs	%	classes	%	rank	JBoss	Hadoop	Axis2	ActiveMQ	CV
Impl	52,205	4.54%	241	3.44%	2	4.29%	0.63%	10.25%	1.93%	0.86
Service	50,948	4.43%	212	3.03%	3	5.30%	0.26%	9.30%	1.57%	0.86
Data	50,667	4.40%	317	4.53%	4	7.70%	2.51%	1.60%	5.19%	0.56
Message	50,013	4.35%	356	5.08%	5	1.09%	0.02%	4.25%	13.45%	1.12
Factory	42,788	3.72%	394	5.63%	6	7.34%	0.85%	2.22%	3.48%	0.70
File	41,795	3.63%	179	2.56%	7	0.64%	11.86%	1.33%	2.14%	1.15
Manager	37,238	3.24%	153	2.18%	8	7.46%	2.65%	0.91%	0.80%	0.92
Connection	36,856	3.20%	216	3.08%	9	5.52%	0.05%	0.43%	6.62%	0.93
Context	29,051	2.52%	148	2.11%	12	2.58%	0.85%	4.74%	1.48%	0.61
Stream	28,165	2.45%	123	1.76%	13	0.63%	3.35%	2.39%	4.11%	0.50
Abstract	24,690	2.15%	129	1.84%	16	4.45%	0.71%	1.34%	1.40%	0.74
Meta	23,226	2.02%	159	2.27%	18	6.30%	0.05%	0.40%	0.05%	1.56
Info	23,111	2.01%	193	2.76%	19	0.87%	0.63%	0.57%	6.76%	1.19
Command	21,640	1.88%	212	3.03%	20	2.76%	0.18%	0.48%	4.12%	0.87
Utils	21,003	1.83%	77	1.10%	23	0.16%	1.61%	5.34%	0.06%	1.19

cate concepts evenly distributed across projects, while high numbers highlight concepts mainly concentrated in a few projects.

Ranking and dispersion. The first observation is that the top concepts in terms of overall locs are shared concepts: The concepts shared by all projects make up 8 of the top-10 concepts (6th column, 'rank'). The two top-10 concepts not appearing in the table are *Marshaller* (#1) and *Job* (#10) which are only present in 3 projects, and so remain heavily shared as well.

In spite of this observation, top shared concepts are not equally distributed among projects (last column, *CV*), but instead tend to appear heavily in one or two projects, while only sporadically in others. Even the two concepts most evenly distributed in the table (*Data*, #4, and *Stream*, #13) have high CV values (0.56 and 0.50) when compared to the concepts with the lowest CV overall: *Thread* (0.15, #378), *Exception* (0.16, #115), and *Xml* (0.22, #292). This means that in spite of an important foundation of shared concepts, each project retains a strong identity that clearly differentiates it from other projects. This also opens interesting avenues of comparative studies: Why are for instance *Factories* much less common in Hadoop than in other projects? Does this represent an obvious different design? Or simply different naming conventions? Could this suggest opportunities for evolution or refactorisation?

Semantic. Although the concepts captured in the table seem to cover a large range of concerns, they can roughly be classified in three categories (indicated in italics, normal font, and bold, respectively). Some concepts denote *generic* computer science and system-level abstractions (*Data*, *File*, *Context*, *Stream*, *Meta*, *Info*). Other concepts are more *domain-specific* (*Service*, *Message*, *Connection*), and cover abstractions typically associated with distributed systems. A longer table would show more of these, with *Client* and *Server* appearing for instance in all projects at rank #30 and #35. Finally, the remaing concepts capture *design patterns* (even if some might describe utility classes as an anti-pattern), i.e. particular ways to organise code and behaviour in object oriented software. Here again, differences are apparent between projects, with JBoss a clear heavy user of templates, and Hadoop much less so.

Comparing the shared concepts of Table 4 with concepts that are unique to specific projects (Table 5 for JBoss and Hadoop) highlights the transveral nature of pattern con-

cepts. Although some generic abstractions can still be found (*Verifier*, *Domain*, *FS* for File System) in the top-10 concepts unique to JBoss and Hadoop, most of them refer to specific technologies (e.g. CMP and CMR are both taken from the EBJ standard, Thrift is a cross-platform RPC product), with no mention of any design template.

Table 5: Unique Concepts in JBoss and Hadoop

(a) **JBoss**				(b) **Hadoop**		
Concept	rank	%locs		Concept	rank	%locs
JBoss	24	5.83%		*FS*	46	5.21%
HA	95	2.25%		Hadoop	57	4.69%
Verifier	99	2.21%		Thrift	111	2.68%
Clustered	129	1.78%		DFS	118	2.55%
Statement	137	1.64%		*Namesystem*	141	2.14%
CMP	157	1.39%		*History*	171	1.68%
QL	170	1.26%		Reduce	176	1.65%
Domain	179	1.20%		*Writable*	214	1.36%
CMR	180	1.19%		Zip	263	1.07%
JDK	206	1.05%		CB	262	1.07%

3.4 Inter-concept relationships

Although we have so far analysed the weight, rank, and semantics of individual concepts, most concepts are used in combination with others. Only 3.71% of all concepts (69 out of 1861) are used in complete isolation, never appearing with any other concept in class names. Many of these isolated concepts are concentrated in Hadoop (5.8%) and Axis2 (5.11%), with JBoss and ActiveMQ showing much lowever levels (1.79% and 1.54% respectively).

Table 6: n-grams shared by all projects

ngram	global #	JBoss	Hadoop	Axis2	ActiveMQ
Meta Data	152	149	1	1	1
Input Stream	35	4	16	6	9
Output Stream	25	3	12	3	7
Context **Factory**	15	7	1	5	2

We therefore now focus on the relationships between concepts, using n-grams (Section 2) to capture their tendency

Table 7: n-grams shared by 3 projects

ngram	global #	JBoss	Hadoop	Axis2	ActiveMQ
Connection **Factory**	55	37	–	1	17
Data Source	44	24	–	19	1
Factory Impl	38	11	–	26	1
Proxy **Factory**	26	23	–	2	1
File System	25	–	23	1	1
Class Loader	20	11	–	8	1
Socket **Factory**	16	13	2	1	–
Byte Array	14	1	5	–	8
Connection **Manager**	14	11	–	2	1
Reference **Factory**	6	1	–	4	1
Object Input	6	2	–	3	1
Manager Impl	5	2	–	2	1
Application Context	5	1	–	3	1
Object Input Stream	5	2	–	2	1
Callback **Handler**	5	2	–	1	2
Http Server	5	2	2	1	–
Exception **Handler**	4	1	1	–	2
Xml File	4	2	–	1	1
Round Robin	4	2	1	–	1
Bounded Range	4	2	1	–	1
Handler Factory	3	1	1	1	–
MBean Info	3	1	1	–	1
Java Generator	3	–	1	1	1
File Reader	3	–	1	1	1

to appear together. When applied to our corpus of class names, we found a total of 3757 n-grams appearing more than once, with a majority (61.54%) of bi-grams. In terms of sharing, n-grams show a different behaviour than individual concepts. First, only a small minority (8.25%) of n-grams are shared. Second, the most frequent n-grams tend to be specific to one or two projects. For instance, out of the 20 most common n-grams in terms of frequencies, a majority (12) are not shared, while only 2 (*Meta Data*, #1, and *Input Stream*, #14) can be found in all projects.

These differences between individual concepts (whose most frequent represenatives tend to be shared) and n-grams (showing the opposite trend) highlights the building block nature of concepts. Although the most common concepts tend to appear in most projects, the ways in which they are combined are specific to each project. Our conjecture is that these combinations capture the identity of each project along with its main design philosophy.

For space reason, we focus in the following on shared n-grams, with the n-grams shared by all projects listed in Table 6, and those shared by all but one in Table 7. Most of these generic n-grams fall in two main categories: 13 n-grams (shown in italic) simply capture generic multi-words notions, such as *Input Stream*, or *File System*. Another set of 10 n-grams (shown with some of their words in bold) represent application of design templates (*Factory, Manager, Handler*), some of which were identified in Table 4. This shows for instance that factories, although primarily used in JBoss, are used in other projects to handle connections, sockets, and context (the latter being essentially used for JNDI naming operation, and invocation handling). These "template" n-grams also highlights how design templates are applied to elements of the domain (sockets, connections, naming contexts) to provide desirable properties

(in the case of factories, configurability through dependency injection).

3.5 Visualisation: Hadoop

Finally, we briefly illustrate how concepts can help analyse a particular piece of middleware, Hadoop in our case. An extract of Hadoop's conceptual graph (cf. Section 2) showing Hadoop's most important concepts is shown in Figure 4. The graph provides a good intuitions of the key elements of Hadoop: Hadoop is about a file system (HFS), map and reduce tasks, and jobs. These three different areas have little overlap: FS concepts, task concepts, and job concepts are only connected through concepts denoting cross-cutting concerns (logging, auditing).

Strikingly, most of Hadoop's top concepts in terms of locs are shared (red and orange colours), but none of them are design templates. This second point probably explains some of the anomalies we have noted about Hadoop in terms of class name lengths and sharing: Hadoop class names are mainly focused on domain concepts, with issues of code organisation and configurability left to other means (reflection, annotations, Spring and Guice for instance).

4. RELATED WORK

The importance of identifiers and concepts in software has long been noted [10]. This key role has motivated a growing body of experimental work on identifiers in the software engineering community, in particular in terms of program comprehension, software maintenance and reverse-engineering. None of these works explicitly target middleware, however. Instead they tend to seek general rules and methods , rather than gain domain-specific insights as we have tried in this paper. We review some of these works below.

Some works have focused on the concepts and topics underlying a particular piece of software. For instance, in [1], Alan Blackwell investigated the metaphors used in the javadoc documentation of the standard Java libraries. Blackwell observed that Java objects tend to be presented as members of a society with rights and obligations (contracts). In another work, Kuhn, Ducasse and Tudor [8] applied Latent Semantic Indexing (LSI) to the vocabulary found in software (both in comments and in identifier names) to extract and represent the key topics present in a source code.

Other works have more systematically analysed the nature of the parts making up identifier names. Working across 29 Java projects (including JBoss and Axis), Høst and Østvold [7] investigated how the initial part of method names (usually a verb such a 'get', or 'load') is linked to semantic features of the code (i.e. "the method body contains a loop"). Using this approach, they showed they could distinguish between well-defined concepts, whose meaning vary little across projects, from more variable ones.

In [3], Butler et al found a positive relationship between the quality of identifiers in Java code (as captured by best pratices for naming) and the quality of the code the identifiers refer to (as captured by existing metrics for code quality and complexity). In an other work, Butler et al [5], performed a cross-comparison of the part of speech structure of class names (e.g. "adjective-noun") and types of inheritance in Java (interfaces vs. classes), and found that the structure of 90% of Java class names could be captured using four grammatical rules, while some types of inheritance were more strongly correlated to some rules than others.
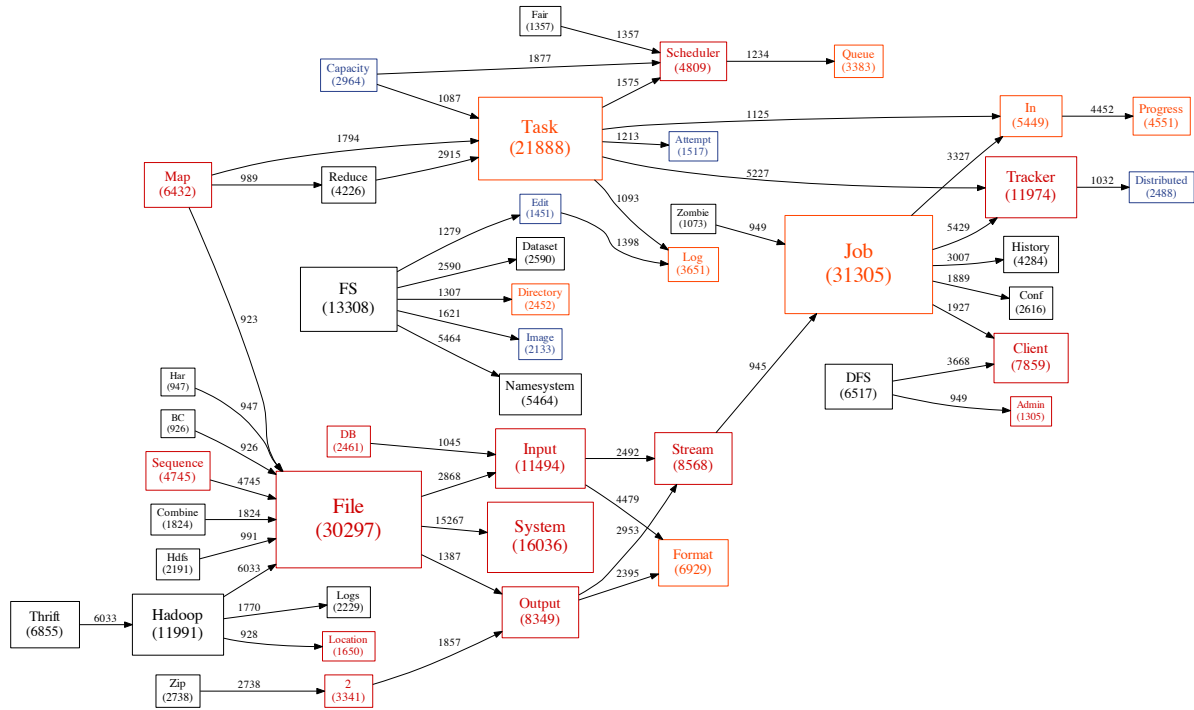
Figure 4: Main concepts in Hadoop 1.0.3. Node labels show the locs count of each concept. Edge labels show the number of locs involved in a connection. Black concepts are unique to Hadoop; blue concepts are shared with one other project; orange ones with two others; and red ones with all others.

5. CONCLUSION AND OUTLOOK

In this paper we have explored how the words contained in class name identifiers of middleware products could help capture the key concepts used in these projects. By looking at four popular Java-based middleware products we have found an important vocabulary of shared concepts, which should prove useful to inform training on middleware technologies. Interestingly, a large proportion of concepts are not about distribution, but about code organisation through patterns (e.g. dependency injection). In spite of this sharing, we have also found that different projects use concepts differently. This reflects the different aims of each project. (JBoss being a component container, it is understandable that it uses the concept of Manager more heavily.) These different uses probably also reflect different design philosophies and architectural choices, which opens up the prospect of interesting comparative studies (Is Hadoop less adaptable than JBoss?) and potential paths for cross-pollination between projects.

6. REFERENCES

[1] A. F. Blackwell. Metaphors we Program By: Space, Action and Society in Java. In *Proceedings of the 18th Psychology of Programming Interest Group 2006*, 2006.

[2] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. D. Pietra, and J. C. Lai. Class-based n-gram models of natural language. *Comput. Linguist.*, 18(4):467–479, Dec. 1992.

[3] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *14th European Conf. on Soft.*

[4] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Improving the tokenisation of identifier names. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 130–154, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Mining java class naming conventions. In *IEEE 27th Int. Conf. on Soft. Maintenance, ICSM 2011*, pages 93–102, 2011.

[6] W. Emmerich, M. Aoyama, and J. Sventek. The impact of research on middleware technology. *SIGSOFT Softw. Eng. Notes*, 32(1):21–46, 2007.

[7] E. W. Host and B. M. Ostvold. The programmer's lexicon, volume i: The verbs. In *Proceedings of the Seventh IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '07, pages 193–202, Washington, DC, USA, 2007. IEEE Computer Society.

[8] A. Kuhn, S. Ducasse, and T. Gírba. Semantic clustering: Identifying topics in source code. *Inf. Softw. Technol.*, 49(3):230–243, Mar. 2007.

[9] R. Maia, R. Cerqueira, C. de Souza, and T. Guisasola-Gorham. A qualitative human-centric evaluation of flexibility in middleware implementations. *Empirical Software Engineering*, 17:166–199, 2012. 10.1007/s10664-011-9167-7.

[10] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *10th Int. Workshop on Program Comprehension*, pages 271–278, 2002.

[11] F. Taïani, P. Grace, G. Coulson, and G. Blair. Past and future of reflective middleware: towards a corpus-based impact analysis. In *Proceedings of the 7th workshop on Reflective and adaptive middleware*, ARM '08, pages 41–46, New York, NY, USA, 2008. ACM.

Maintenance and Reengineering, CSMR'10, pages 156–165, Washington, DC, USA, 2010. IEEE Computer Society.