Helping Program Comprehension of Large Software Systems by Identifying Their Most Important Classes

Ioana Şora

Department of Computer and Software Engineering
University Politehnica of Timisoara, Romania
ioana.sora@cs.upt.ro

Abstract. An essential prerequisite before engaging in any maintenance activities of complex software systems is the good comprehension of the existing code. Program comprehension is supported by documentation, which can be either developer documentation or documentation obtained by reverse engineering. In both cases, but especially in the case of reverse engineered documentation, this means a large amount of detailed documents that have to be carefully studied. Processing such large and detailed information can be made easier if there is an executive summary - a short document pointing to the most important elements of the system.
In our work we propose a tool to automatically extract such a summary, by identifying the most important classes of the system. Our approach consists of modeling the static dependencies of the system as a graph and applying a graph ranking algorithm. How we build the dependency graph is the key for the success of the approach. We empirically determine how different dependency types should be taken into account in building the system graph. The proposed approach has been validated by experiments on a set of open source real systems.

Keywords: reverse engineering, program comprehension, key classes, recommender tool

1 Introduction

Program comprehension [1] is an important software engineering activity, which is necessary to facilitate reuse, maintenance, reengineering or extension of existing software systems.

In the case of large software systems, program comprehension has to deal with the huge amount of code that implements it. When starting with the study of an unknown system, software engineers are overwhelmed by the amount of information, which makes it difficult to filter out the important elements from a lot of details.

Documentation can help with program comprehension. Assuming that the documentation is up-to-date, there are still additional requirements related to the contents of the documentation such that it is effective for facilitating the early stages of work: very useful are documents such as architectural overviews, or describing what is called the core of the system. Detailed and scattered implementation documentation is of little use, as are large class diagrams that are reverse engineered from the code. This has been

confirmed by the experiments described in [2], where most subjects did not appreciate reverse engineered diagrams to be helpful due to the information overload in these class diagrams.

In our work we help program comprehension of object oriented software systems by identifying their most important classes [3], [4]. This gives a set of good starting pointers for studying the system. We consider that the importance of a class is given by the amount and types of interactions it has with other classes. Thus, a natural approach of identifying the most important classes is based on ranking them with a graph-ranking algorithm.

In this work we adapt PageRank [5] to use it for the purpose of ranking classes of software systems according to their importance for the design of the system. The key here for obtaining a ranking which is indeed effective for the goal of program comprehension is to use an adequate graph model of the system.

Section 2 describes our approach of modeling the structure of software systems by static dependencies and the way we use this for identifying the most important classes of the system. We define two parameters of the graph model, given by the weights of dependency types and dependency directions. Section 3 presents experimental results. We do first an empirical fine-tuning of the parameters of the graph model, then apply our approach to a set of relevant open-source projects. In Section 4 we will discuss our results and draw the conclusions of our experiments, while also comparing with related work. Section 5 draws the conclusions of this paper.

2 Ranking Classes According to Their Importance

2.1 Building the Right Model

We model the software system as a graph having as nodes classes or interfaces. If an edge exists from node A to node B, this means, in PageRanks terminology, that node A recommends node B as important. Applying the right strategy for determining where and how to place the recommendation edges is the crucial element for the effectiveness of the ranking approach.

In our model, the recommendations derive from the program dependencies identified by static analysis with help of the model extractors of the ART tool suite [6]. If A depends on B, this means both that A gives a recommendation to B but also that B gives a recommendation to A. We call the edge from A to B a *forward recommendation*, while the edge from B to A is a *back recommendation.*

The forward recommendation, resulting directly from a dependency, is obvious: a class which is used by many other classes has good chances to be an important one, representing a fundamental data or business model. But also the reverse is true: a class which is using a lot of other important classes may be an important one, such as a class containing a lot of control of the application or an important front-end class. If only the directed dependency would be considered as a recommendation, then library classes would rank very high while the classes containing the control would remain unacknowledged. Thus the reason for having back recommendations.

Recommendations may also have weights. A class is not necessarily recommending all its dependency classes with an equal number of votes. It will give more recommendation votes to those classes that offer it more services. Thus recommendation weights are derived from the type and amount of dependencies.

Static dependencies in object oriented languages are produced by various situations. There are different classifications of the mechanisms that constitute dependencies [7]. In accordance with these, we distinguish between following categories of dependencies between two classes or interfaces A and B :

- inheritance: A extends B
- realization: A implements B
- field: A has at least one member of type B
- member access: A member of B is accessed from code belonging to A
- method call: A calls a method of B. We can further distinguish if it is a static method call or a method call on a class instance. Every distinct method of B which is called is counted as a new dependency.
- parameter: A method of A has at least one parameter of type B
- return type: A method of A has the return type B
- local variable: A local variable of type B is declared in code belonging to A
- instantiation: An instance of B is code belonging to A
- cast: A type-cast to B occurs in code belonging to A

Two classes A and B can be at the same time in several dependency relationships: for example, A can have members of type B, but in the same time it can have a method with parameters of type B and overall it can call several different methods of B.

The strength of the recommendation is proportional with the strength of the dependency which takes into account both the number of dependency relationships and the types of dependency relationships between the two classes.

The strength of a dependency can be estimated using an approach based on an ordering of dependency types according to their relative importance. Establishing the relative importance of static dependency types is a subject of empirical estimation and different authors use different frameworks for this [7]. In this work, we continue to use the ordering of dependency types used previously in the context of architectural reconstruction by clustering in [8]. In summary, we take as reference for the weakest type of dependencies the local variables dependency type and assign it weight 1. On the next level of importance, level 2, we put the dependency strength given by one distinct method that is called. Usually several distinct methods of a class are called, thus these weights will sum up to a significant value. Also on level 2 are dependencies generated from creating instances. Dependencies due to parameters, return values or having a member dependency is assigned weight 3 while inheritance and realization have weights 4. We will empirically validate this assumption of dependency weights in the context of class ranking in Section 3.

The weight of the forward recommendation from A to B is given by the dependency strength of the cumulated dependencies from A to B. The weight of the back recommendation from B to A is a fraction F of the weight of the forward recommendation from A to B. We identified that, while a class is important if it is both used by other classes and it is also using other classes, the second argument should have a smaller weight in the

global reasoning, only a fraction F of the dependency strength. We illustrate this idea with the simple example presented in subsection 2.2 and we also empirically investigate values for this fraction in Section 3.

2.2 A Simple Example

We illustrate the idea of our approach using as an example a simplified program structure with four classes A, B, C, D. Class A is the front-end component of the application, B is the main business component, C a helper, and D some utility or library class.

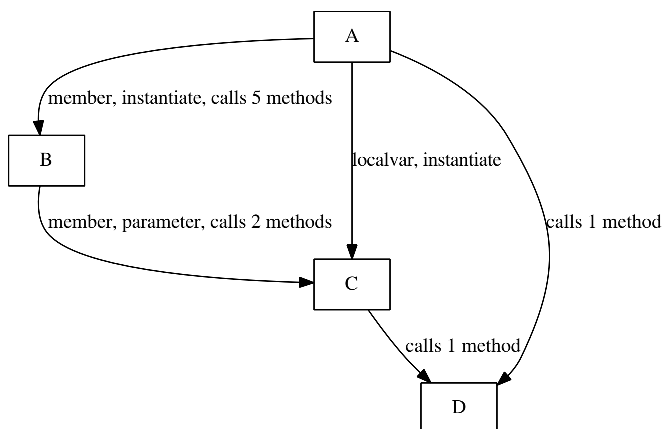

Fig. 1. Example: graph of program dependencies.

Figure 1 depicts the dependencies between the 4 classes. Class A has a member of type B, it instantiates objects of type B and calls five different methods of class B. Also, class A has a local variable of type C and instantiates an object of type C. Class B has a member of type C, has member functions with parameters of type C, and calls 2 different methods of C. Both classes A and C call one static method of class D.

We use this simple example to explain the importance of using a weighted dependency graph, taking into account the dependency strengths induced by different dependency types, and also of using back-recommendations.

In a first try, we consider the dependency graph directed and unweighted. If PageRank is applied on the directed graph of figure 1, without back-recommendations, we obtain the following ranking: D(0.41), C(0.29), B(0.16), A(0.12). This ranking places the deepest classes on a top level, bringing the utility class D on the top position. The utility class D can be considered a library class with high reuse potential, however D is not the most important class of the system and not so important for program comprehension. This shows that simply applying PageRank on the directed graph defined by the dependencies is not a valid method of identifying the classes that are important for program comprehension.

In a second try, back-recommendations are included and the unweighted graph from figure 1 will be completed with a reverse edge for every original edge. Applying PageRank on this new graph results in a new ranking: A(0.29) C(0.29) B(0.21) D(0.21). This order brings on top two classes of medium importance (A and C), while ranking the key class B as low as the utility class D.

In a third try, we introduce weights reflecting the type and amount of dependencies, using the empirical values defined in the previous section. Following weights result: AB=15, AC=3, AD=3, BC=11, CD=2. Back-recommendations are given a fraction F of the weight of the forward recommendation. We experiment with different values for F. If F=0 (no back-recommendations) the ranking results D(0.38), C(0.3), B(0.19), A(0.11), which is wrong since it brings the utility class on top. If F=1, the ranking is B(0.36), A(0.29), C(0.24), D(0.08). If F=1/2, the ranking is B(0.34), C(0.29), A(0.24), D(0.11). These last two rankings reflect very well the situation of B being the most important class, while D plays only a small role as an utility class. A and C are of medium importance. Since this example is generic and small, we cannot argue whether A should be ranked above C or not.

More experiments on real-life systems are described in Section 3 and they will show that PageRank can be used as an effective means to identify key classes for program comprehension if it is applied to a correct model of recommendations. We argue that this model has to take into account both the strength of the dependencies and also include back-recommendations, with a fraction $0 < F < 1$ bringing the best results.

3 Experimental Results

3.1 Experimental setup

In order to validate the proposed ranking tool, we apply it on a set of relevant open source systems. We run our tool that implements the ranking approach described in section 2, using weighted recommendations, according to the type and amount of dependencies as well as back-recommendations.

In all the experiments, we limit the examination of the tool produced ranking to the top 30 ranked classes, independent from the size of the system. We consider that a percentage limit of 15% or even 10% of the system size would result in candidate sets which are too big for the purpose of the tool, that of facilitating an easy start in program comprehension.

Thus we have to experimentally prove that the top 30 ranked classes are indeed the most important classes of the analyzed systems.

Unfortunately, the identification of the most important classes of a system may be, up to a certain degree, subjective to different opinions of different experts. The reference solution will be the reduced set resulting from the intersection of different expert opinions. In order to validate the tool, we could do an experiment asking different software experts to judge the top rankings produced by the tool. This scenario requires a big effort and, in the end, the objectivity of our experts may be questionable.

We chose to rely for the validation of the tool output on the comparison with reference solutions extracted from developers documentation. The kind of developer documentation that is useful for our validation is usually found in documents described

as "architectural overview", "core of the system", "introduction for developers", etc. It may consist either in pruned diagrams or even free text descriptions. Of course, developers documentations may be outdated or not accurate. In order to reduce these risks, we preferred as case studies systems that provide both developers documentation and documentation from other sources, mainly systems included in the *Qualitas Corpus* - a curated collection of software systems intended to be used for empirical studies on code artifacts. These systems have been also analyzed in other works and their structure has been discussed by several sources, thus we can define as reference solution an intersection of different expert opinions. In this way we establish unbiased reference solutions to compare the solutions produced by our tool.

In the next subsection 3.2 we present the detailed analysis and discussion of one system. We use this system to perform the empirical validation of the value of fraction F representing the back-recommendations and to show the importance of choosing the weights that quantify dependency strengths.

Some more systems are then analyzed and presented in subsection 3.3.

In chapter 4 we will discuss our results and draw the conclusions of our experiments, while also comparing with related work.

3.2 Detailed Analysis of the First Case Study

In this subsection we present the detailed analysis and discussion of a system, Apache Ant. Apache Ant[1] is a Java library and command-line tool to compile, build, test and run Java applications. We analyze release 1.6.1, feeding as input ant.jar containing the core part of ant. It contains 524 classes. A developer tutorial[2] indicates the following key classes to understand the design of the Ant core: `Project`, `Target`, `UnknownElement`, `RuntimeConfigurable`, `Task`, as depicted in Figure 2. Besides these main classes, `IntrospectionHelper`, `ProjectHelper2` and `ProjectHelperImpl` are mentioned in the documentation as important.

The `Project` class is instantiated whenever Ant starts and, with the help of helper classes, the `Project` instance parses the `build.xml` file. The `Target` class represents the targets specified in the `build.xml` file. Once parsing finishes, the build model consists of a project, containing multiple targets. A target is a container of tasks, represented by specializations of the `Task` class.

Each task in Ant has a reference to its `RuntimeConfigurable` instance. Prior to the task being executed, it would need to be configured from its `RuntimeConfigurable` instance.

The class `UnknownElement` was introduced to allow the model to support storing information about tasks whose classes were not known at parse-time. `UnknownElement` extends `Task`, allowing it to be stored in the Ant object model.

We consider the following set of 8 classes as the reference solution: `Project`, `Target`, `UnknownElement`, `RuntimeConfigurable`, `Task`, `IntrospectionHelper`, `ProjectHelper2` and `ProjectHelperImpl`. Ant has been also analyzed for the de-

[1] http://ant.apache.org/
[2] http://codefeed.com/tutorial/ant_config.html

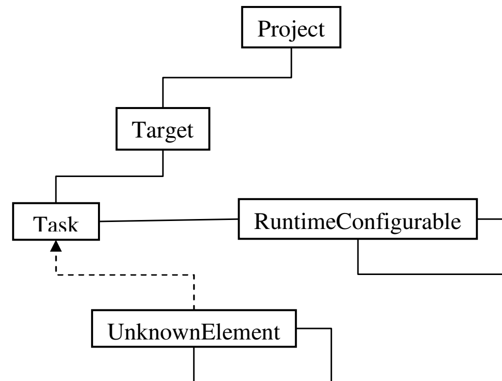

Fig. 2. Core classes of Ant described in the developers tutorial.

tection of key classes in [9], and the same set of classes has been used as a reference solution.

We perform an empirical fine-tuning of our recommender tool in order to get its ranking results as close as possible to the reference solution.

We use the detailed analysis of this case study to answer following questions, for the fine-tuning of the recommender tool:

Q1. Which is the role of dependency directions? More specifically, are back-recommendations needed? If yes, then which is the relative contribution of back-recommendations compared to that of forward recommendations?

In our experiments, we will consider the following possibilities for dependency directions:

- F=0: no back recommendation, only forward recommendation determined by the dependency relationship.
- F=1: back-recommendations have the same weight as forward recommendations.
- F=1/2, F=1/4: back-recommendations have a smaller weight than forward recommendations

Q2. Which types of dependencies are relevant for the goal of this recommender tool? We will investigate whether all dependency types are equally important or if there are some dependency types that can be ignored without affecting the quality of the result or even improving it.

For dependency types and weights, we consider the following profiles:

- *AllDep*: dependencies of all types are considered and summed up, with equal contributions
- *CallsOnly*: only method calls are considered, ignoring all other types of dependencies. The number of distinct methods called is taken into account in the global dependency strength.
- *InterfOnly*: only dependency relationships induced by elements visible from the interface are counted (inheritance, implementation, method parameters), ignoring all details such as local variables, member accesses and method calls.

– *AllWeighted*: all dependency types, but weighted such that interface elements have a higher weight than method calls while local variables use brings the smallest weights. This weighting schema is the one mentioned in subsection 2.1.

In our experiments we will generate and study all the possible combinations resulting from values for dependency weights and the fraction F of back-recommendations. We want to find out which combination favors the retrieval of most of the classes of the reference set. The summary of these experiments is depicted in Table 1. The values in the table represent the percentage of the classes of the reference set that are retrieved in the top N(where N=10, 15, 20, 30 and 50) ranked classes.

Table 1. Experimental results summary for Ant

		F=0	F=1	F=1/2	F=1/4
AllDep	Top 10	0.38	0.38	0.25	0.38
	Top 15	0.63	0.63	0.63	0.63
	Top 20	0.63	0.88	1.00	0.88
	Top 30	0.75	1.00	1.00	1.00
	Top 50	0.75	1.00	1.00	1.00
AllWeight	Top 10	0.38	0.38	0.25	0.38
	Top 15	0.50	0.63	0.63	0.75
	Top 20	0.63	0.88	1.00	0.88
	Top 30	0.75	1.00	1.00	1.00
	Top 50	0.75	1.00	1.00	1.00
CallsOnly	Top 10	0.38	0.38	0.25	0.25
	Top 15	0.44	0.38	0.50	0.50
	Top 20	0.75	0.63	0.88	0.88
	Top 30	0.75	0.89	0.89	1.00
	Top 50	0.75	1.00	1.00	1.00
InterfOnly	Top 10	0.50	0.25	0.25	0.38
	Top 15	0.63	0.25	0.38	0.38
	Top 20	0.75	0.38	0.38	0.50
	Top 30	0.75	0.63	0.63	0.75
	Top 50	0.75	0.75	0.75	0.88

A first conclusion that can be drawn from Table 1 is that all dependency types have to be considered, because ignoring certain dependency types (as in the profiles *CallsOnly* and *InterfOnly*) has a negative impact. While both the *AllDeps* and *AllWeighted* profiles allow for combinations leading to the retrieval of all the classes of the reference solution in the top 20 ranked classes, this is not possible in any combination with the *CallsOnly* and *InterfOnly* profiles.

A second conclusion that can be observed by analyzing the columns of Table 1 is that the worst results are obtained when no back-recommendations are used (F=0). Using back-recommendations (F=1) improves the results, but the improvement is bigger when $F < 1$ (with F=1/2 and F=1/4).

Figure 3 presents the top 30 ranked classes when analyzing Ant with our tool configured with the *AllWeighted* profile.

	F=0	F=1	F=1/2	F=1/4
1	***Project***	***Project***	***Project***	***Project***
2	FileUtils	***Task***	***Task***	***Task***
3	Location	Path	BuildException	BuildException
4	BuildException	BuildException	Path	Path
5	***Task***	FileUtils	FileUtils	FileUtils
6	FilterSet	Commandline	Commandline	Parameter
7	***Target***	AbstractFileSet	Parameter	Commandline
8	ChainReaderHelper	Execute	AbstractFileSet	Reference
9	ProjectComponent	Parameter	Execute	***Target***
10	BuildEvent	***ProjectHelper2***	Reference	AbstractFileSet
11	***RuntimeConfigurable***	Java	***Target***	Execute
12	Path	Zip	***UnknownElement***	***UnknownElement***
13	Reference	***UnknownElement***	DirectoryScanner	***RuntimeConfigurable***
14	FilterSetCollection	DirectoryScanner	ComponentHelper	ComponentHelper
15	ComponentHelper	***ProjectHelperImpl***	***ProjectHelper2***	***IntrospectionHelper***
16	PropertyHelper	***Target***	***IntrospectionHelper***	ProjectComponent
17	DataType	DefaultCompilerAdapter	***ProjectHelperImpl***	DirectoryScanner
18	***UnknownElement***	Reference	***RuntimeConfigurable***	***ProjectHelperImpl***
19	Parameter	ComponentHelper	ProjectComponent	Location
20	Os	Javadoc	Zip	BuildEvent
21	BuildListener	***IntrospectionHelper***	TokenFilter	***ProjectHelper2***
22	Condition	TokenFilter	ModifiedSelector	TarEntry
23	***IntrospectionHelper***	Ant	Javadoc	ModifiedSelector
24	LineTokenizer	Javac	Javac	Condition
25	JavaEnvUtils	CommandlineJava	DefaultCompilerAdapter	EnumeratedAttribute
26	Watchdog	MatchingTask	Ant	ZipShort
27	Commandline	Rmic	EnumeratedAttribute	Resource
28	InputRequest	FilterChain	BuildEvent	MailMessage
29	TimeoutObserver	ModifiedSelector	Java	TokenFilter
30	AbstractFileSet	ExecTask	Rmic	FileSelector
Found:	***6/8***	***7/8***	***8/8***	***8/8***

Fig. 3. Top fragment of the ranking of Ant classes using the *AllWeighted* profile.

We can see that with F=0, only 6 out of the 8 classes of the reference set are found. Introducing back-recommendations brings an improvement: with F=1, 7 out of 8 classes are found, while with F=1/2 and F=1/4, all the 8 classes are found in the top 30 ranking. The detailed analysis of Ant validates our assumption, described with help of the simple example in Section 2.2, that back-recommendations are needed but they should be assigned weaker strengths than their forward recommendation counterparts. Taking F=1/2 and F=1/4, all classes of the reference set are found in the top 30 ranking for the analyzed system. Using the value F=1/4 enables to get the last hit on position 21 compared to F=1/2 where the last hit is found earlier, at position 18. In future work, more experiments could be done to fine-tune the value of the back-recommendation fraction F. In this work, the following experiments use the value F=1/2.

By examining the classes that occupy top positions in all rankings, we notice the constant presence of certain classes that were not included in the reference solution, so we manually analyzed them in order to decide if their high ranking can be considered dangerous false positives or if they should be rightfully included in the set of key classes. Among these classes, `Path`, `Parameter`, `Reference`, `Commandline`, and `BuildEception` represent some fundamental data structures that are very much used

and this is the reason that they are ranked on top positions. The classes ComponentHelper, AbstractFileset, DirectoryScanner have a controlling function which makes them interesting to be studied. It is interesting to notice that these 3 classes are also found in top positions in the ranking obtained in [9]. The top ranked classes obtained for Ant in [10] and [11] have also similarities with our ranking.

3.3 More Experimental Results

We completed a series of experiments on an additional set of systems. In the experiments described in this section we use the value F=1/2 for the back-recommendations, as it resulted from the set of experiments described in the previous subsection.

The analyzed systems are: JHotDraw, JEdit, ArgoUML, Wro4j and JMeter.

Analysis of JHotDraw JHotDraw[3] is a highly customizable graphic framework for structured drawing editors. Its source code and binaries are freely available.

We analyze here JHotDraw, release 6.0b.1. We take advantage of the capabilities of our ART model extractor tools [6] that can handle compiled code, and directly feed it as input the jhotdraw.jar file from the binary distribution, which proves to contain 398 implementation classes. The architecture of the system is documented by its developers, the documentation provides a short description of the core architectural classes and interfaces, enumerating the most important artifacts in the opinion of the system developers. The case study of JHotDraw has been analyzed also in [12], in order to produce a more precise class diagram, in terms of relationships, than the one provided by the authors of JHotDraw. We noticed a couple of classes considered important and added to the diagram: DrawingEditor, StandardDrawingView, CompositeFigure. Thus we conclude that the set of important artifacts (classes and interfaces) for an executive summary of JHotDraw is formed by these pointed out by the developers, completed with the three classes added in the study of [12]: Figure, Drawing, DrawingView, DrawApplication, Tool, Handle, DrawingEditor, StandardDrawingView, CompositeFigure. This set of 9 classes is further considered the reference summary of the whole system comprising 398 classes.

The top 30 classes in the ranking produced by our tool are: Figure, DrawingView, FigureEnumeration, DrawingEditor, Undoable, StorableInput, StorableOutput, CollectionsFactory, Drawing, DrawApplication, StandardDrawingView, ConnectionFigure, CommandTool, AbstractCommand, CompositeFigure, DrawApplet, AbstractTool, Connector, HTMLTextAreaFigure, TextFigure, ConnectionTool, HandleEnumeration, PolyLineFigure, Handle, RelativeLocator, Locator, FigureChangeListener, DesktopEventService, DecoratorFigure.

We can see that all the nine classes which are in the reference are ranked in the top 30. This means that our tool finds all the classes of the reference solution, ranking them in the top 7.5% classes of the 398 examined. Eight classes from the reference set are actually ranked in the top 20, while five of them are in the top 10. The first places of the ranking are also taken by the most important classes.

[3] http://www.jhotdraw.org/

Analysis of JEdit JEdit[4] is a cross platform programmer's text editor written in Java. We analyze the code of release 5.1.0, with 1266 classes.

Developer documentation is available[5] and it gives the following introductory overview of jEdit implementation: The main class of jEdit is jEdit, which is the starting point for accessing various components and changing preferences. Each window in jEdit is an instance of the View class. Each text area you see in a View is an instance of JEditTextArea, each of which is contained in its own EditPane. Files are represented by the Buffer class. The Log class is used to print out debugging messages to the activity log. Plugin developers have to extend EBPlugin.

In summary, the developers documentation point out the following classes of interest: jEdit, View, EditPane, Buffer, JEditTextArea, Log, EBMessage. We take this set of 7 classes as the reference solution.

The top 30 classes in the ranking produced by our tool are: jEdit, View, JEditBuffer, Buffer, TextArea, Log, Interpreter, NameSpace, SimpleNode, GUIUtilities, EditPane, TokenMarker, CallStack, ParserRuleSet, MiscUtilities, VFS, VFSBrowser PluginJAR, JEditTextArea, TextAreaPainter, VFSFile, Selection, Mode, Primitive, DisplayManager, Gutter, SearchAndReplace, EditBus, EBMessage, Parser.

We can see that all the seven classes which are in the reference are ranked in the top 30. This means that our tool finds all the classes of the reference solution, ranking them in the top 2.5% classes of the 1266 examined. Out of these, six classes from the reference set are ranked in the top 20. Actually, the only class which did not make it into the top 20, class EBMessage, is not so much a core class but it is mentioned in the summary as important for plugin developers, being important only in this context. Four of the classes in the reference set are found in the top 10. The first places of the ranking are also taken by the most important classes.

Analysis of ArgoUML ArgoUML[6] is a well-known open source UML modeling tool. In this work we analyze its release 0.9.5, having detailed architectural descriptions in Jason Robbins's dissertation[7] which created the fundamental layer for ArgoUML. The analyzed jar contains a total of 852 classes.

The set of key classes as identified from the architectural description is composed by the following 12 classes: Designer, Critic, CrUML, ToDoItem, ToDoList, History, ControlMech, ProjectBrowser, Project, Wizard, Configuration, Argo.

Our analysis resulted in the following top 30 ranked classes: ProjectBrowser, Designer, ToDoItem, ColumnDescriptor, CrUML, Project, UMLUserInterfaceContainer, TreeModelPrereqs, Critic, UMLAction, MMUtil, FigNodeModelElement, NavPerspective, Notation, Wizard, UMLModelElementListModel, PropPanel, Configuration, TableModelComposite, ToDoList, Argo, PropPanelModelElement, ParserDisplay, CodePiece, FigEdgeModelElement, UMLChecklist, ModuleLoader, SelectionWButtons, ArgoEventPump, NotationName.

[4] http://jedit.org/
[5] http://community.jedit.org/cgi-bin/TWiki/view/Main/JEditSourceCodeIntro
[6] http://argouml.tigris.org
[7] http://argouml.tigris.org/docs/robbins_dissertation

We notice that 6 out of the 12 classes in the reference solution are ranked in the top 10, while 9 classes are found in the top 20 and 10 classes are found in the top 30.

Analysis of Wro4j Wro4j[8] is an open source web resource optimizer for Java. We have used release 1.6.3, containing 337 classes.

The classes that are mentioned in the design overview [9] as important for understanding the design of the system, and which are further considered as the reference solution in our experiment, are the following 12 classes: `WroModel`, `WroModelFactory`, `Group`, `Resource`, `WroManager`, `WroManagerfactory`, `ResourcePreProcessor`, `ResourcePostProcessor`, `uriLocator`, `uriLocatorFactory`, `WroFilter`, `resourceType`.

The first 30 classes as ranked by our tool are, in order: `WroManager`, `Resource`, `WroConfiguration`, `BaseWroManagerFactory`, `ResourcePreProcessor`, `WroTestUtils`, `WroUtil`, `WroModelFactory`, `InjectorBuilder`, `ResourceType`, `Context`, `HashStrategy`, `ResourcePostProcessor`, `WroModel`, `WroFilter`, `WroRuntimeException`, `ProcessorDecorator`, `UriLocatorFactory`, `WroManagerFactory`, `CacheStrategy`, `PreProcessorExecutor`, `ReadOnlyContext`, `LifecycleCallbackRegistry`, `Injector`, `LifecycleCallback`, `WildcardExpanderModelTransformer`, `ResourceWatcher`, `DefaultWroModelFactoryDecorator`, `Group`, `UriLocator`.

We observe that 5 out of the 12 classes in the reference solution are found in the top 10 ranked, while 10 classes are found in the top 20 and all 12 classes are found in the top 30.

Analysis of JMeter Jakarta JMeter[10] is a Java application for testing of Web Applications. We analyze version 2.0.1, its core found in ApacheJMeter_core.jar which contains 280 classes. Design documentation[11] and other works that analyzed this system [13] mentions following classes: `AbstractAction`, `JMeterEngine`, `JMeterTreeModel`, `JMeterThread`, `JMeterGUIComponent`, `Sampler`, `SampleResult`, `TestCompiler`, `TestElement`, `TestListener`, `TestPlan`, `TestPlanGUI`, `ThreadGroup`.

The first 30 classes as ranked by our tool are, in order: `JMeterUtils`, `JMeterProperty`, `GuiPackage`, `SampleResult`, `JMeterTreeNode`, `JMeterThread`, `SaveService`, `AbstractTestElement`, `JMeterTreeModel`, `MainFrame`, `JMeterContext`, `PropertyIterator`, `JMeter`, `Sampler`, `CompoundVariable`, `ThreadGroup`, `JMeterTreeListener`, `TestCompiler`, `MenuFactory`, `ThreadGroupGui`, `AbstractJMeterGuiComponent`, `Arguments`, `CollectionProperty`, `SampleEvent`, `ValueReplacer`, `JMeterGUIComponent`, `StandardJMeterEngine`, `ResultCollector`, `GenericController`.

We observe that 3 out of the 13 classes in the reference solution are found in the top 10 ranked, while 7 classes are found in the top 20 and 8 classes are found in the top 30.

[8] https://code.google.com/p/wro4j/
[9] https://code.google.com/p/wro4j/wiki/DesignOverview
[10] http://jmeter.apache.org/
[11] http://wiki.apache.org/jmeter/

4 Discussion and Comparison With Related Work

4.1 Summary of Experimental Results

In Table 2 we summarize the results obtained in our experiments. For each one of the five analyzed systems, we represent in this table the raw data describing it: its size, the size of the reference solution, the number of classes found if the cut threshold is placed after the first 10, 15, 20 or respectively the first 30 ranked classes. The execution time includes both the analysis of dependencies and building the model of the system and the applying of the ranking.

Table 2. Experimental results summary

	JHotDraw	Ant	jEdit	ArgoUML	Wro4j	JMeter	Avg.Precis.	Avg.Recall
System size	398	524	1266	852	337	280		
Execution time	1 min	2 min	3 min	2.5 min	1 min	1 min		
Reference set	9	8	7	12	12	13		
Hits in Top 10	5	2	4	6	5	3	42%	42%
Hits in Top 15	7	5	5	6	8	5	40%	61%
Hits in Top 20	8	8	6	9	10	7	40%	81%
Hits in Top 30	9	8	7	10	12	8	30%	90%

We compute the recall and precision for our approach, defined as in [13]:

The *recall*, showing the techniques retrieval power, is computed as the percentage of key classes retrieved by the technique versus the total number of key classes present in the reference set.

The *precision*, showing the techniques retrieval quality, is computed as the percentage of key classes retrieved versus the total size of the result set.

The last columns of Table 2 present the average values of recall and precision computed from our experimental data concerning the six analyzed systems.

We consider this a good result, since the measured recall guarantees the user a good start for program comprehension, having assured two thirds of the relevant classes by examining a very small number of classes (only 10-15 classes), independently on the size of the whole system. Also, in case of 4 systems out of the six analyzed, all the relevant classes have been found in the top 30.

The precision values in our experiments are disadvantaged by the very small size of the reference solution, which is in average 10 classes. However, we did not add further classes to these reference sets, in order to keep them fair by avoiding subjectivity. Also, while in most systems it would be difficult to rank with precision all classes, this reduced top set is that which is unanimously agreed as the most important. On the other hand, a user which uses our tool to analyze a new system does not know the exact size of this top set. He or she will use the tool with the expectation to find the top 10 or top 20 classes. If we examine the top fragments of the rankings produced by the tool, we notice there several classes that are certainly not irrelevant, although they were not included in the reference top set.

In our opinion, program comprehension is effectively supported by the tool in the following scenario: the tool identifies a small number of classes as key classes. These classes give the starting points for the examination of the system by a software engineer doing maintenance or evolution activities. For practical effectiveness, most often is not worth to move the cut threshold below the top 20 ranked classes, due to the increased effort of manual investigation. The very short and general executive summary of the system is quickly and easy retrieved in this top set. After getting this executive summary, the user can continue the analysis tasks either by parsing the documentation, beginning from the discovered key classes, or he/she may apply other techniques such as feature localization [14] to track more localized areas of interest.

4.2 Comparison with Related Work

There are several approaches trying to identify the most important software artifacts (classes, packages, functions) from a software system. They present differences in following aspects:

- the primary information that is extracted and analyzed: the majority uses static analysis [15], [16] but there are also approaches based on dynamic analysis [9], [11].
- the criteria used to define the importance of a class: the majority derives the importance of a class from the ways it interacts with other classes, given by design metrics (such as coupling), network metrics of the topology of the interactions between the classes, or a combination of these. Other approaches use the number of changes recorded by the versioning system [17] as an indicator of the importance of a class. There were also attempts to use textual information such as class names [18] as hints for the importance of a class.
- the techniques for identifying the key classes are mostly based on network analysis [16], including here also webmining techniques [13], and more recently machine learning [15], [19]. Also interactive tools for pruning of reverse engineered class diagrams are developed [20].

Comparing the results obtained by all these different approaches is difficult, because they are using different software systems as case studies and not all publications describe the raw data of their experiments such as the rankings that they obtained. Where such data was available we compared the results with our results for the same systems.

Coderank [21] was one of the first works to introduce the concept of calculating PageRank values for a graph resulting from static dependencies between the software artifacts such as classes of a project. However, there is little experimental validation that supports the claims about their ability to help program comprehension by identifying relevant components of real software systems.

An important work in detecting key classes of software systems belong to Zaidman et al [13], [22], [9]. They uses a graph-ranking algorithm, HITS, in order to detect key classes of a software system. They combine this webmining technique with dynamic and static analysis, and perform experiments on two systems. With dynamic analysis they attain an average recall of 92% and precision 46%. However, a major drawback of

this approach is that dynamic analysis relies very much on the user finding good execution scenarios. It also presents scalability issues and has a high execution time (1h45). Zaidman also combined this webmining technique with static analysis but concluded that the static analysis was not able to achieve a reasonable precision and recall. Here their best reported results were an average recall of 50% and precision 8%, while the execution time is still high (over 1 hour).

In our work we have proven that static analysis can be used to successfully and efficiently identify key classes, our results near the values obtained by [13] with dynamic analysis, while the execution time in our case is just a couple of minutes. We think that a major enabling factor for our positive result here is our recommendation model, which takes into account all possible types of static dependencies with appropriate weights, while Zaidman uses coupling metrics that take into account only method calls. We also appreciate in the work of [13] the extensive description of their result sets in case of Ant and JMeter, which allowed us to compare with our result sets for these two systems. We retrieved a couple of classes from outside the reference set that appear both in their and our top ranking result set, leading to a future reconsideration of the reference sets.

Kamran et al [11] develop their own version of a dynamic coupling metric. Their results, obtained on analysing the Ant system, compete with those obtained in [13] but significantly reduce the execution time. It is interesting to note that the some classes present in the top ranking of Ant, while not included in the reference set, are the same in our approach and in [11].

Steidl et al [16] start from static analysis to retrieve important classes of a system. Their approach calculates a centrality index for the nodes of the dependency graph obtained by static analysis. They performed an empirical study to find the best combination of centrality measurement of dependency graph. They used as baseline for validation of results opinions of several software developers. They found out that centrality indices work best on an undirected dependency graph including information about inheritance, parameter and return dependencies. Using the Markov centrality leads to the best results, with a precision between 60% and 80% in the top 10 recommendation set. Their experiments were performed on a set of 4 systems. However, they do not compute the recall of their method, nor do they mention the members or the sizes of the reference sets. From the data presented, one could conclude that the baseline sets for each system were larger, being reunions of different expert opinion instead of intersection of such, resulting in more that 10 classes in the baseline. Theses larger baseline solutions may have favored the count of hits in the top 10, as opposed to the smaller reference solutions used in our experiments. We appreciate that the retrieval power of this technique is similar with ours.

Meyer et al [10] propose an automated way to identify the important classes of a software system based on K-core decomposition. They show that the classes in the highest K-cores are the ones that are the most important, but in order to reduce the number of classes k-core values should be used in conjunction with other network metrics as centralities. They discuss their results on 3 systems, two of the systems being Ant and JHotDraw and obtaining rankings that have many similarities with the ones obtained in our work. Pan et al [23] use K-core decomposition to find the most important packages of a software system.

Perin et al [24] use PageRank on the graph of static dependencies. They report experiments on several systems, including Ant, Jmeter and Jedit. However, the set of top ranked classes is very different from the sets of top ranked classes reported for these systems in our work, and is different as well from those reported in [9], [11], [10].

Osman et al worked on condensing class diagrams by including only the important classes. They used a very different approach, based on machine learning [15]. They use design metrics extracted from available forward design diagrams to learn and then to validate the quality of prediction algorithms. Nine small to medium size open source case studies are analyzed, taking as baseline available forward design diagrams which contain from 11 to 57 classes, representing between 4% and 47% of the project size. In a follow-up, Osman et. al. [18] built a classifier that is based on the names of classes in addition to design metrics, but the results show that combining text metrics with design metrics leads to modest improvements over using design metrics only.

Thung et al [19] uses machine learning combining design metrics and network metrics in the learning process. Introducing network metrics besides the design metrics improves the results of [15] by almost 10%. However, in [19] network metrics and design metrics are computed as distinct and independent attributes and used in the learning process. In our approach, the network metric (PageRank) is adapted to be computed on the weighted graph resulting after the design metrics (measuring dependency strengths and coupling) are applied, and thus we believe that the concept of recommendation is better adapted to its particular purpose.

The work of Hammad [17] starts from another point of view regarding the importance of classes: they consider that the classes that were important to the design of the system are these often impacted by design changes. They measure the design importance of a class as the number of commits that impact the class, and this is also measured for the sets of classes that collaborate.

5 Conclusions

In this work, we develop a method and tool for automatically identifying the most important classes of a software system, in order to facilitate the start of program comprehension activities.

Our approach is based on static analysis, used to build a graph model of the system, and the PageRang graph ranking algorithm.

In order to obtain a ranking that is relevant for our goal, the graph model of the system has to be carefully built for this purpose. We define two parameters of the graph model: the weights of dependency types and the dependency directions, which we call forward recommendations and back recommendations. We have experimentally determined that all types of static dependencies between classes have to be taken into account, weighted according to the relative importance given by the dependency type and number of occurrences. Also, experiments have shown that back-recommendations are necessary, but should be assigned only a fraction $0 < F < 1$ of the weight of their corresponding forward recommendations.

We have validated our approach by analyzing six open source systems and comparing the top ranked classes with these described as important in developers documenta-

tion. The results have shown our techniques retrieval power, which is able to find an average of 90% of the classes of the reference sets indicated in the developers documentation as ranked in the top 30 by our tool.

References

1. von Mayrhauser, A., Vans, A.: Program comprehension during software maintenance and evolution. Computer **28**(8) (Aug 1995) 44–55
2. Fernández-Sáez, A.M., Chaudron, M.R.V., Genero, M., Ramos, I.: Are forward designed or reverse-engineered UML diagrams more helpful for code maintenance?: A controlled experiment. In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering. EASE '13, New York, NY, USA, ACM (2013) 60–71
3. Sora, I.: Finding the right needles in hay - helping program comprehension of large software systems. In: ENASE 2015 - Proceedings of the 10th International Conference on Evaluation of Novel Approaches to Software Engineering, Barcelona, Spain, 29-30 April, 2015. (2015) 129–140
4. Sora, I.: A PageRank based recommender system for identifying key classes in software systems. In: 10th IEEE Jubilee International Symposium on Applied Computational Intelligence and Informatics, SACI 2015, Timisoara, Romania, May 21-23, 2015. (2015) 495–500
5. Page, L., Brin, S., Motwani, R., Winograd, T.: The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab (November 1999) Previous number = SIDL-WP-1999-0120.
6. Sora, I.: Unified modeling of static relationships between program elements. In Maciaszek, L., Filipe, J., eds.: Evaluation of Novel Approaches to Software Engineering. Volume 410 of Communications in Computer and Information Science. Springer Berlin Heidelberg (2013) 95–109
7. Briand, L., Daly, J., Wust, J.: A unified framework for coupling measurement in object-oriented systems. Software Engineering, IEEE Transactions on **25**(1) (Jan 1999) 91–121
8. Sora, I., Glodean, G., Gligor, M.: Software architecture reconstruction: An approach based on combining graph clustering and partitioning. In: Computational Cybernetics and Technical Informatics (ICCC-CONTI), 2010 International Joint Conference on. (May 2010) 259–264
9. Zaidman, A., Calders, T., Demeyer, S., Paredaens, J.: Applying webmining techniques to execution traces to support the program comprehension process. In: Software Maintenance and Reengineering, 2005. CSMR 2005. Ninth European Conference on. (March 2005) 134–142
10. Meyer, P., Siy, H., Bhomwick, S.: Identifying important classes of large software systems through k-core decomposition. Advances in Complex Systems **17**(07n08) (2014) 1550004
11. Kamran, M., Azam, F., Khanum, A.: Discovering core architecture classes to assist initial program comprehension. In Lu, W., Cai, G., Liu, W., Xing, W., eds.: Proceedings of the 2012 International Conference on Information Technology and Software Engineering. Volume 211 of Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg (2013) 3–10
12. Guéhéneuc, Y.G.: A reverse engineering tool for precise class diagrams. In: Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research. CASCON '04, IBM Press (2004) 28–41
13. Zaidman, A., Demeyer, S.: Automatic identification of key classes in a software system using webmining techniques. Journal of Software Maintenance and Evolution: Research and Practice **20**(6) (2008) 387–417

14. Dit, B., Revelle, M., Gethers, M., Poshyvanyk, D.: Feature location in source code: a taxonomy and survey. Journal of Software: Evolution and Process **25**(1) (2013) 53–95
15. Osman, M.H., Chaudron, M.R.V., Putten, P.v.d.: An analysis of machine learning algorithms for condensing reverse engineered class diagrams. In: Proceedings of the 2013 IEEE International Conference on Software Maintenance. ICSM '13, Washington, DC, USA, IEEE Computer Society (2013) 140–149
16. Steidl, D., Hummel, B., Juergens, E.: Using network analysis for recommendation of central software classes. In: Reverse Engineering (WCRE), 2012 19th Working Conference on. (Oct 2012) 93–102
17. Hammad, M., Collard, M., Maletic, J.: Measuring class importance in the context of design evolution. In: Program Comprehension (ICPC), 2010 IEEE 18th International Conference on. (June 2010) 148–151
18. Osman, M., Chaudron, M., Van Der Putten, P., Ho-Quang, T.: Condensing reverse engineered class diagrams through class name based abstraction. In: Information and Communication Technologies (WICT), 2014 Fourth World Congress on. (Dec 2014) 158–163
19. Thung, F., Lo, D., Osman, M.H., Chaudron, M.R.V.: Condensing class diagrams by analyzing design and network metrics using optimistic classification. In: Proceedings of the 22Nd International Conference on Program Comprehension. ICPC 2014, New York, NY, USA, ACM (2014) 110–121
20. Osman, M., Chaudron, M., Van Der Putten, P.: Interactive scalable abstraction of reverse engineered uml class diagrams. In: Software Engineering Conference (APSEC), 2014 21st Asia-Pacific. Volume 1. (Dec 2014) 159–166
21. Neate, B., Irwin, W., Churcher, N.: Coderank: a new family of software metrics. In: Software Engineering Conference, 2006. Australian. (April 2006) 10 pp.–378
22. Zaidman, A., Du Bois, B., Demeyer, S.: How webmining and coupling metrics improve early program comprehension. In: Program Comprehension, 2006. ICPC 2006. 14th IEEE International Conference on. (2006) 74–78
23. Pan, W., Hu, B., Jiang, B., Xie, B.: Identifying important packages of object-oriented software using weighted k-core decomposition. Journal of Intelligent Systems **23**(4) (2014) 461–476
24. Perin, F., Renggli, L., Ressia, J.: Ranking software artifacts. In: 4th Workshop on FAMIX and Moose in Reengineering (FAMOOSr 2010). (2010) 120