A Top-Down Program Comprehension Strategy for Packages

Stéphane Ducasse, Michele Lanza, Laura Ponisio
Software Composition Group, University of Bern, Switzerland.
www.iam.unibe.ch/∼scg.

IAM-04-007

September 23, 2004

Abstract

Understanding packages is an important activity in the reengineering of large object-oriented systems. The relationships between packages and their contained classes can affect the cost of modifying the system. The main problem of this task is to quickly grasp the structure of a package and how it interacts with the rest of the system. In this paper we present a top-down program comprehension strategy based on polymetric views, radar charts, and software metrics. We illustrate this approach on two applications and show how we can retrieve the important characteristics of packages.

Keywords: Program understanding, reverse engineering, software visualization, polymetric views

1 Introduction

It is well-known that 50% to 75% of the overall cost of a software system is devoted to its maintenance [17]. Moreover, during maintenance software professionals spend at least half their time reading and analysing software in order to understand it [7] [2]. The maintenance of object-oriented applications is harder than the ones written in procedural languages [34] because the presence of inheritance and late-binding increases the number of potential dependencies within a program [34, 30, 10, 8].

In addition, nowadays most applications are structured in terms of packages. The current belief is to design packages in a similar way to classes: A package should have a high cohesion and a low coupling with the rest of the system [3, 4]. However, in the context of object-oriented applications and frameworks, packages have different roles, such as containing some key subclasses of a framework. The way a system is decomposed into packages and the way classes are distributed in them represent an important characteristic of the application design and development process constraints. Therefore it is crucial to understand packages in their fine-grained mechanisms. Providing a way to support the understanding of packages (or other sets of classes) is important also in the context of reengineering. Packages are complex entities with multiple facets.

Our approach is based on a limited and simple metamodel of source code (state access, class reference and inheritance) and on the definition of simple measurements based on these relations. Based on this information, package roles within the context of an application are revealed using visualizations enriched with metrics: We propose to support the understanding of packages based on three visualizations, as visualization supports efficiently the combination of properties. The three visualization we propose are: a global polymetric view that illustrates the roles that the packages play in the context of production/consumption of functionality, and two radar diagrams at the level of a single package. The radar diagrams depict how a package is internally structured and how it relates to the rest of the system.

Structure of the paper. In Section 2 we discuss the problem of understanding the packages that compose an application. In Section 3 and Section 4 we introduce our approach and in Section 5 we present the global package view in detail showing how the package characteristics appear compared with the complete application and define the needed measurements. In Section 6 we present two radar chart views that shows how the package contents appears in the context of the package itself. In Section 7 we analyze the results of applying our approach to the case studies and in Section 8 we discuss related work. We summarize our findings in Section 9.

2 Understanding Packages

Chikofsky states that "The primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development" [5]. We focus on the problem of how to provide an understanding of the packages that compose a large application. Our long term goal is to be able to provide a means to assess the quality of packages during the refactoring of a system. In this paper we consider as package a group of classes that a developer has decided to put together or that a clustering algorithm identified [1, 19]. In Java, the term package is mapped to the Java package but without their scoping aspects, *e.g.,* import statements and namespaces. In Smalltalk the term package corresponds to a binary deployment unit and traditional class categories. Our approach is not based on a particular implementation language because the underlying metamodel is language independent [9].

Our aim is to answer the following questions:

- What is the importance of a package in terms of its intrinsic properties such as the number of classes and methods it contains? How many clients rely on it?
- Does the package use several other packages or is it more self-contained?
- What is the impact of changes in the relationships between packages?
- Can we identify patterns or repeating package characteristics?
- How is package structured: does it only extend other packages via inheritance, or does it define itself some complex hierarchies? When classes are subclassing other classes what are exactly the relationships that link them (state, behavior)?

2.1 Challenges and Constraints

Our approach targets the initial phases (*e.g.,* the first couple of weeks) of a reverse engineering process during which a first mental picture of the system is formed [28].

Characterizing packages requires the processing of a lot of information. Software metrics are well-known

to reduce large amounts of information [12]. However, this reduction often leads to only seeing isolated information about a larger phenomenon. In addition, the combination of metrics leads to dimensional inconsistencies and numbers that are meaningless or hard to interpret.

These problems can be partially circumvented by using *software visualization* because visual displays allow us to combine visually multiple aspects of complex problems [27] [33]. However, software visualizations are often too simplistic and lack visual cues for the viewer to correctly interpret them [23]. In other cases the visualizations obtained are still too complex to be of any real value to the viewer. The challenge is to define a visualization that conveys the right level of information while scaling in terms of screen usage so that the human brain can compare and identify multiple packages at the same time.

3 Approach Overview

We adopt a top down approach: The reengineer first uses coarse-grained visualizations of the packages and their connections and then reaches detailed information about packages. Using the polymetric view [16] he can spot *interesting* packages such as core packages, lightweight packages that merely use behavior and state of other packages or packages that are independent from the others. He can then inspect the package in detail using a radar chart. Two radar charts are provided: (1) a global package view where the package is compared with its surrounding context and (2) a local package view where the package is analyzed on its own.

Case studies. We took as case studies Base Visualworks and CodeCrawler.

Base Visualworks is a large portion of the Cincom VisualWorks Smalltalk environment. It is an industrial system, developed over the last 15 years. It defines all the runtime entities of a smalltalk environment (classes, methods, strings, characters, collections, graphical display, memory objects) but also the compiler framework, the coding tools (debugger, code browsers), the OS support and all the widgets offered by the graphical framework.

CodeCrawler, a software visualization tool, is an application written by members of our research group and serves to illustrate examples.

Case Study	Packages	Classes	LOC
Base Visualworks	94	1402	262660
CodeCrawler	8	93	9088

4 Packages and Classes

We now present the information that we extract from source code and show how we use it to model packages and classes. A package contains classes which refer to other classes or are referred to by other classes in the system. We name *clients* the classes that access the state or invoke the behavior of other classes. Consequently the used classes are called *providers*. We call a *client package* a package that depends on another one because its classes refer to classes of the other package.

4.1 Class and Package Dependencies

There are many relationships between classes that can be used to characterize classes in the context of packages. Briand *et al.* [3, 4] propose a complete overview and analysis of the possible metrics to characterize coupling and cohesion. While the propositions are interesting some of them are quite complex to put in place [13].

We chose to take the minimal information such as class references and inheritance relationships and evaluate how far we could get with such simple information. An important influence on this work is the focus on the object-oriented context in which packages exist. We took into account the fact that in object-oriented applications inheritance hierarchies can be spread over multiple packages and that flattening packages according to the inheritance relationships is not satisfactory from an understanding point of view, since packages convey semantics as well as the design intentions of programmers. For example, a package may contain only the abstract core of a framework, or contain only the concrete leaf classes that represent a framework extension, or represent a specific product or the work of a specific development team.

Besides being based on simple size metrics such as the number of classes defined in a package, the information that we use is based on three kinds of dependencies between classes:

1. *Inheritance*: a class is a subclass of another. It inherits its behavior.

2. *State*: a class may use the instance variable inherited from its ancestors.

3. *Class Reference*: a class makes an explicit reference of another *e.g.,* by instantiating the class. This encompasses instance variable types.

The dependencies are *directed* which is important since packages play the roles of clients and providers. If

Name	Description
PP	*(Number of Provider Packages).* Number of *package* providers of a package. PP(P1)=1, PP(P2)=2, PP(P4)=1.
CP	*(Number of Client Packages).* Number of *packages* that depend on a package. CP(P1)=3, CP(P3)=2, CP(P4)=0.
RTP	*(Number of Class References To Other Packages).* Number of class references from classes in the measured package to classes in other packages. RTP(P1)=2, RTP(P2)=1 ,RTP(P3)=1, RTP(P4)=0.
RRTP	*(RelativeNumber of Class References To Other Packages).* RTP divided by the sum of RTP and the number of internal class references.
RFP	*(Number of Class References From Other Packages).* Number of class references from classes belonging to other packages to classes belonging to the analyzed package. RFP(P1)=0, RFP(P2)=1, RFP(P3)=3, RFP(P4)=0
RRFP	*(Relative Number of Class References From Other Packages).* RFP divided by the sum of RFP and the number of internal class references.
PIIR	*(Number of Internal Inheritance Relationships).* Number of inheritance relationships existing between classes in the same package. PIIR(P1)=0, PIIR(P2)=0, PIIR(P3)=3, PIIR(P4)=2
RPII	*(Relative Number of Internal Inheritance Relationships).* PIIR divided by the sum of PIIR and EIP. RPII(P1)=0, RPII(P2)=0, RPII(P3)=1, RPII(P4)=1.
EIC	*(Number of External Inheritance as Client).* Number of inheritance relationships in which superclasses are in external packages. EIC(P1)=0, EIC(P2)=2, EIC(P3)=1, EIC(P4)=1
EIP	*(Number of External Inheritance as Provider).* Number of inheritance relationships where the superclass is in the package being analyzed and the subclass is in another package. EIP(P1)=4, EIP(P2)=0, EIP(P3)=0, EIP(P4)=0
REIP	*(Relative Number of External Inheritance as Provider).* EIP divided by the sum of PIIR and EIP. REIP(P1)=1, REIP(P2)=0, REIP(P3)=0, REIP(P4)=0.
ASC	*(Number of Ancestor State as Client).* Number of accesses to instance variables defined in a superclass that belongs to another package. ASC(P3)=0, ASC(P4)=1
RASC	*(Relative Number of Ancestor State as Client).* ASC divided by the sum of ASC and ASCI. Where ASCI, Number of Ancestor State Client Internal to the Package is the ancestor state class dependencies internal to the package. We consider only dependencies from a class that is inside the package to other classes of the same package.
ASP	*(Number of Ancestor State as Provider).* Number of times that instance variables of classes belonging to the analyzed package are accessed by classes belonging to other packages. ASP(P1)=1, ASC(P4)=0
RASP	*(Relative Number of Ancestor State as Provider).* ASP divided by the sum of ASP and the number of gives ancestor state dependencies between classes when both classes belong to the package.
CC	*(Number of Class Clients).* Number of external *class* dependencies that are clients of a package. Sum over the number of the class dependencies (ancestor state, class reference and inheritance) that refer to a package. CC(P1)=4, CC(P2)=1, CC(P3)=3, CC(P4)=0.
NCP	*(Number of Classes in a Package).* Number of classes in the package. NCP(P1)=2.

Table 1. Package Measurements.

there is an inheritance relationship between two classes we do not count it as a class reference.

4.2 Characterizing Packages

To condense the information of a large application at the level of its packages, we use simple object-oriented metrics based on the dependencies we defined previously. We use some simple measurements based on the three kinds of information that we extracted and use these measurements to support the understanding.

The measurements we currently compute are listed in Table 1. In this table the term *external dependencies* denotes dependencies that originate from other packages and target classes of the analyzed package. The metric example values refer to the situation depicted in Figure 1.

We define both *absolute* and *relative* metrics for packages. An example of an absolute metric of a package is RTP (Number of Class References To Other Packages) which is the number of class references to classes belonging to other packages from classes belonging to the analyzed package. This metric is useful to assess whether a package (and its classes) is heavily using other packages, but fails to convey information about the package itself. Relative metrics follow the pattern: $property/(internalproperty + externalproperty)$. The relative metric RRTP (RelativeNumber of Class References To Other Packages) divides RTP by the total number of class references in a package, thus creating a normalized metric (*i.e.,* between 0 and 1) that denotes to what extent a package is self-contained (low RRTP) or not (high RRTP).

5 A Polymetric View for Packages

Polymetric views are a visualization approach for nodes–and–edges graphs enriched with semantic information such as metrics [16]. A node figure is able to render up to five metric values: its width, height, x–

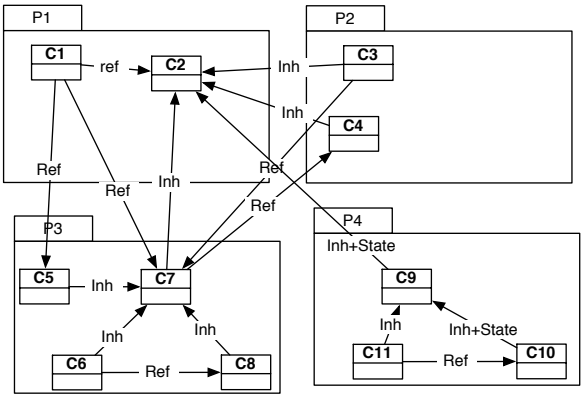

Figure 1. Some packages with class dependencies (C1 refers to C2, or inherits of C2, or is a client of C2).

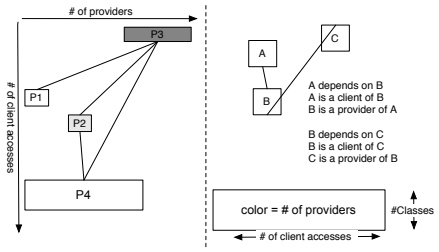

Figure 2. Principles of the Package Sedimentation View**.**

and y–position, and its color. An edge figure is able to visualize two metric values: width and color. By applying metrics to the x– and y–position of the nodes, for example, similar entities are located close together in an easily identifiable region of the visualization exhibiting some of their defining characteristics. Entities with differing characteristics are then placed in a distinct region of the visualization. In this way, the shape of the visualization is able to communicate useful facts about the set of all visualized items.

To support the understanding of application at the level of packages, we define a polymetric view named Package Sedimentation View: The idea behind the view is that heavily used packages are located at the bottom of the view.

It displays all packages of a system as nodes and all the dependencies between them as edges, grouping packages with are heavier (*i.e.,* used the most) towards the bottom of the view. We distinguish provider and client relationships. The position of a node reflects its number of package clients and providers. The node color represents the number of providers of the package. The width represents the number of client accesses. The height represents the number of classes defined in the package. The purpose of this view is to visualize a complete system and give the viewer an idea of its structure in terms of packages and inter-package dependencies.

In Figure 2, the package P3 is a client of P1, P2 and P4. P1 and P4 only provide services to other packages so that are aligned to the left. P2 is below P1 because it has more client accesses.

This view reveals a certain number of *symptoms*, that the viewer can look for to infer information:

- Wide nodes at the bottom are packages that contain classes that are heavily used by other classes of the system.
- Nodes at the top are packages containing classes that mainly use classes in other packages.
- White nodes on the left are packages that are independent from other packages. Their classes do not have dependencies towards other packages.
- Dark nodes on the right are packages whose classes depend on classes of other packages.
- Flat nodes are packages with few classes.
- Disconnected nodes are packages that do not have package dependencies.
- Packages with many connected edges to their top side represent packages used by many other packages.
- Lightly colored edges are edges representing few class dependencies between packages.

Example 1. Figure 3 shows the Package Sedimentation View applied to CodeCrawler. There are eight packages with their dependencies. The white wide node at the bottom is the package CCBase. Based on its position in the view we learn that this package is a base package of the system as almost all the other packages are clients. It is positioned to the left and is white, so it does not depend on any other package. It is smaller than the other nodes, therefore it contains few classes. There is only one package (CCMooseExtensions) that does not have clients. The package CCCore depends on at least another because it is not totally white. It is also the third from the right. A closer look to the dependencies of CCCore, (in the figure marked by **E**) indicates that it not only depends on CCBase, but

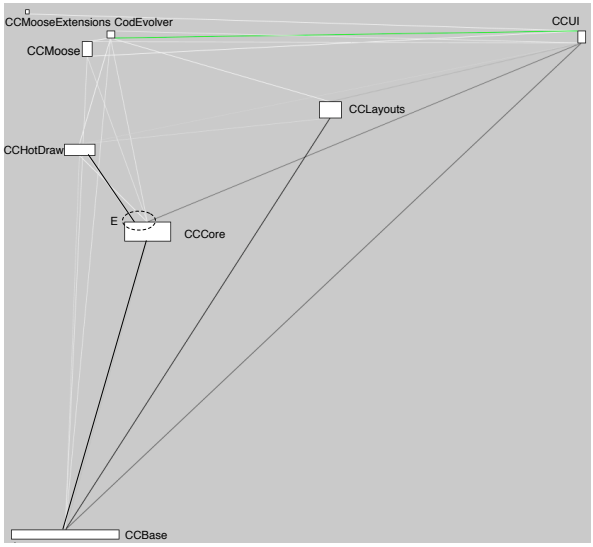

Figure 3. Package Sedimentation View **of the case study** CodeCrawler**.**

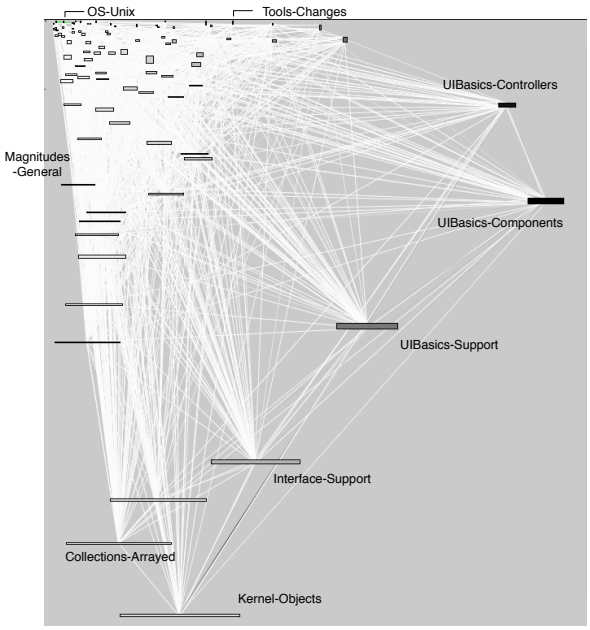

Figure 4. Package Sedimentation View **of the case study** Base Visualworks**.**

also on CCHotDraw. We also see that among the top packages some of them are dark. That is because they depend heavily on the other packages. For instance the package on the top right, CCUI., is the package where the tests are. We expect to see many dependencies from this package to the others, but none the other way around. In that figure we also observe that among the top packages, the ones that are whiter and leaning to the left, barely depend on other packages since they have light gray edges instead of darker ones.

Example 2. Figure 4 shows the application of Package Sedimentation View to the Base Visualworks case study which is composed of 94 packages. The packages on the top correspond to user interface, tools dialogs, printing and operating system related packages. The view spots immediately some key packages: Kernel-Objects, Interface-Support and Collections-Arrayed that are at the bottom of the screen are the foundation on top of which other functionality is built. On the right of the view we see packages that have a lot of providers UIBasics-Controllers and UIBasics-Components.

Discussion. Polymetric views as we implement them are intrinsically interactive and must be interacted with to exploit their full potential. For example, we can highlight the packages that could be affected by a change in a given package. In addition, the Package Sedimentation View can be enriched by visualizing the internal information of dependencies that compose each edge that we see in the view. For instance, we can associate colors to the edges according to the type of class dependency connections that they have. We can also adjust the thickness of the edge to highlight a specific class dependency type that some of its class dependencies have. The viewer can also select the packages that have cyclic dependencies.

The metrics associated with the view can be changed to obtain different information. For example, we change the width and height of the node to represent other metrics such as the number of internal inheritance definitions or the lines of code. Such changes modify the shape of the node but not its location. We obtain flat, narrow or squared nodes conveying different meanings [16].

6 Radar Charts for Packages

While the previous view offers a good overview of a system's structure in terms of packages and their role as clients or providers, it does not provide a finer-grained understanding of single packages. Obtaing such an understanding is difficult since packages are complex entities: they contain classes which may have different interactions with other classes, either within the same package or defined in other packages.

To cope with this situation, we use a radar chart visualization: We apply two radar views which combine several measurements about a package in a single

space. The first view, Global Radar View, presents a package in the context of the complete system and the second view, Relative Radar View, presents how the package is internally structured.

Radar Visualization Principles. A radar visualization is based on dividing a circle area with a certain number of axes and to join the points of each axis as shown in Figure 5. One interesting aspect of the radar visualization is that it generates a surface in the sense that two contiguous axes having high value properties generate more surface. However, using a radar visualization to represent complex constructs is not straightforward since the order of the axes determines the surface and the shapes that the visualization can produce. Therefore it is necessary to determine which criteria are to be abalyzed and how they are to be mapped efficiently on a radar chart.

As packages provide and uses information from other packages, we defined a distribution of the metrics to generate a *butterfly* shape. The left wing of the butterfly represents what the clients of the package use from the package and the right wing what the package in question uses from other packages. The bottom part shows how inheritance is used, *i.e.,* whether the package has classes that are subclassed in other packages and if the package extends other packages.

6.1 Global Radar View

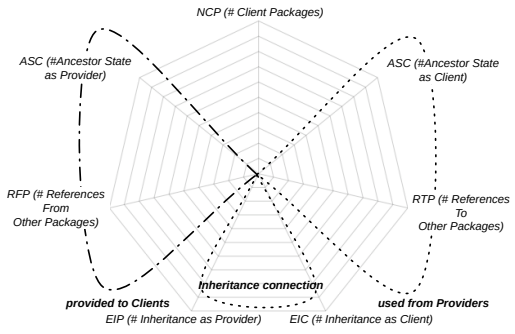

Figure 5. Principles of the Global Radar View.

This first view characterizes a package as presented in Figure 5. It displays information that compares the package in the context of the complete application.

Example. Figure 6 displays the Global Radar View of the packages CCCore, CCBase and CCUI [1].

[1]Refer to Table 1 for an explanation of the metrics used in this section.

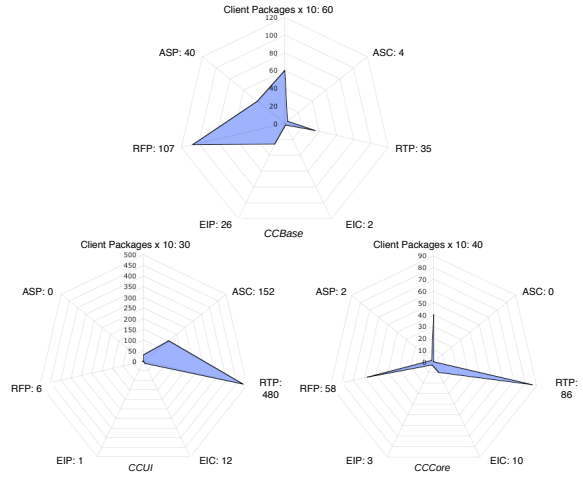

Figure 6. Global Radar View on the CodeCrawler packages CCCore, CCBase and CCUI.

The package CCCore (see also Figure 3) is a central package of CodeCrawler. It uses the package CCBase. This is reflected by the fact that the butterfly has two even narrow wings. The view indicates the following: This package uses 86 external classes while it defines 22 classes (This information was given by the height of the node in the polymetric view). The classes it defines are referenced from other packages too (58 accesses RFP). EIC shows that this package inherits from 10 classes in the other packages (CCBase as we learned in the polymetric view), but this package is also extended (EIP = 3). This package does not directly use state from the superclasses which is an indication of good design. However, we learn that its state is directly accessed by subclasses defined in other packages (ASP = 2). As the package only contains 12 classes (information obtained by the Package Sedimentation View) and that EIC is equal to this number, we learn that this package does not contain an inheritance hierarchy.

The shape of the CCBase package shows that the package is essentially a providing package. In addition it shows that the state of the classes in the package is directly accessed by clients subclasses (certainly CCCore) and that the package also accesses state of other packages. The references to other packages are the ones to default types such as String and Collection. As CCBase is the basis for the complete application and by knowing that only 10 classes inherit from this package class we learn that the application is not flat inheriting solely from a couple of root classes but that it is certainly composed of inheritance hierarchies.

The shape of the CCUI package which contains all the CodeCrawler UI elements, shows that it is mainly a client: Its classes directly access attributes of provider superclasses (ASP = 152 accesses). This package will be im-

pacted if the superclasses located in other packages change. The high-value, 480, of RTP is due to the manual building of menus *i.e.,* direct instantiation of MenuItem.

6.2 Relative Radar View

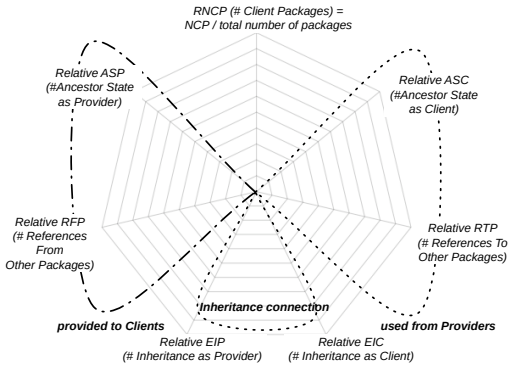

Figure 7. Principles of the Relative Radar View.

While the Global Radar View provides information about a package, it does it by measuring the package in the context of the complete system. However, it is difficult to assess how a property exists *relative* to the package itself. For example, the information that a package defines a lot of classes is refined when we know that most of the classes are inheriting from a class defined inside the package itself or when most of the classes are subclasses of an external class. Presenting such detailed information is the purpose of the Relative Radar View. The Relative Radar View principles are described in Figure 7. Basically it uses the same axes than the Global Radar View but uses relative metrics such as those in Table 1. Note that obtaining 1 as value for a relative metrics indicates that the property does not have a strong value inside the metrics compared to the outside. For example RASP of CCbase in Figure 8 is 1 which means that there is no state access between the class inside the package. Note that that when RRTP is equal to 1, it means there is a weak coupling between the classes inside the package compared to the coupling they have with other class outside the package. This does not mean that the package should be refactored since packages may represent developer intent and do not have to correspond to cohesive packages. For example, grouping framework extensions together makes sense, and it is not mandatory that the extensions are coupled, since the coupling between them is made at the level of the framework.

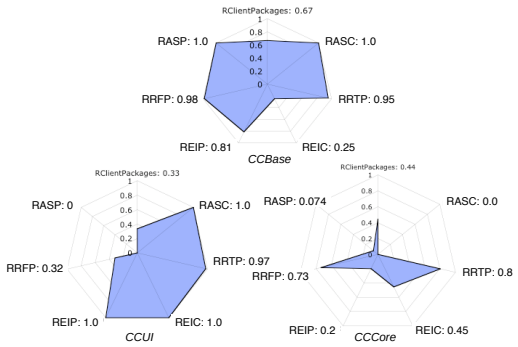

Figure 8. Relative Radar View on the CodeCrawler packages CCCore, CCBase and CCUI.

Example. In Figure 8 we see that the REIC value of CCUI ($REIC = EIC/(EIC+PII)$) is 1: this confirms that it does not define an inheritance hierarchy. Interpreting RRTP whose value is 97%, we learn that the package is not cohesive in the sense that there are 480 references to external classes and only 3% of internal references (*i.e.,* 14 internal references). RRFP is 32% since there are 14 internal references and 6 external ones (RFP).

We learn that in the package CCbase, classes do not directly access state since RASP and RASC have 1 as value, even if such as classes were accessing state of external superclasses (ASC = 4) and its state is accessed by clients classes (ASP = 40 in Figure 6). As the value of REIC is 0.25, we learn that this package, contrary to CCUI, is structured around inheritance hierarchies. It has 3 times more internal inheritance than it is inheriting from others. REIP = 0.81 indicates that it is subclassed from the outside. This does not imply that the classes are not heavily extended in subclasses, as a class can be extended by another class in another package that then acts as another hierarchy root to numerous classes. To get such information we would have to count all the children of the class.

For the package CCCore, we see that it does not access state of other packages (RASC = 0), it has more references to the outside than the references between the classes inside the package (RRTP = 0.8) and it has a bit more references from other packages (RRFP = 0.73). REIP has a value of 0.2 which means that the package has a lot more internal inheritance relationship that it has direct subclasses.

Case study. We applied our approach to a large case study (Base Visualworks) and we selected some characteristic packages displayed in Figure 9. The radars reveals some typical shapes:

Kernel-Objects contains some major inheritance hierarchy root classes such as Object and Model. It contains some important classes such as Boolean, True, False and Error and some key subclasses. This package is heavily subclassed (EIP = 229).

The package Tools-Changes has a client shape. This is

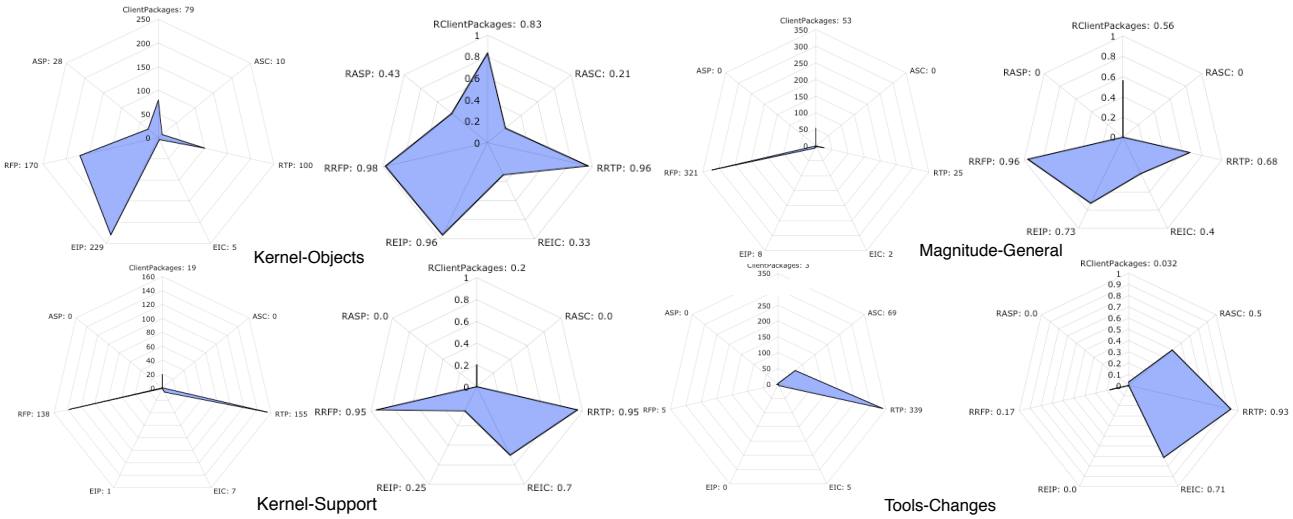

Figure 9. Radar views of selected packages of Base Visualworks**.**

not surprising since it is building all the tools related to the logging facilities of the environment, hence it relies on infrastructure such as the one provided by the package Kernel-Support.

The package Kernel-Support has a shape of both client and provider. Indeed it provides functionality to manage the system such as class externalizers, that are used by the code browsing tools such as the ones of Tools-Changes. To provide such functionality it relies on more primitive packages such as Kernel-Objects.

The package Magnitude is a provider package which merely contains the abstract class Magnitude, and the concrete classes Date and Character. RRFP = 0.96 indicates that there are a few references among the classes in this package while they are still heavily used (RFP = 321).

7 Discussion

Our approach is based on a simple metamodel of source code and metrics. It has proven to be succesful to provide insights about the structure of applications. The Package Sedimentation View provides a global picture of the application while the radar views depict how a package is internally structured and how it relates to the rest of the system.

We support opportunistic understanding [18] in the sense that the user can browse *if necessary* the package and the code it contains, *i.e.,* he can look and interact with the visualization to verify his findings. Our approach compresses information such as all the different relationships between classes. The loss of granularity is balanced by the gain in simplicity and scalability: the packages and the relationships between the packages can be assigned properties and metrics.

We learned that using the surface of the radar to convey information is working well, and it is important to quantify precisely such information. Therefore, having the value of the metrics expressed as part of the axe labels provides useful complementary information.

Even if the current approach is effective for getting a detailed view on packages, there are still questions we plan to investigate:

We do not take into account invocations and we would limit ourselves to a structural view. Introducing invocations may lead to other views on coupling and cohesion but may introduce noise due to late-binding, *i.e.,* an invocation can have multiple potential receivers.

The radar views hide the structural complexity of packages behind easy-to-grasp shapes that allow for a categorization. Due to space and time limitations we do not include a full categorization of packages based on their visualization within the radar views, but plan to include this in our future work.

8 Related Work

Software Visualization. Graphical representations of software have long been accepted as comprehension aids. Many tools enable the user to visualize software using static information, *e.g.,* Rigi [22], Hy+ [6], SeeSoft [11], ShrimpViews [29], and TANGO [26]. The Class, Runtime and Query View approach of Smith and Munro [25] visualizes the internals of classes

using static and dynamic information. The Affinity Browser [24] provides a visual representation of object relationships in terms of dependencies.

Architecture Recovery. Controlling subsystem interactions is one important way to reduce overall complexity of a system. Extensive work has been done to recover abstractions from the source code and to understand inter-subsytems relationships [21] [31]. Bunch [19] is a clustering tool used to deduce software subsystems automatically. ARIS [20] enables software developers to constrain allowable relations between two subsystems and validate existing relations against an interconnection style. Our approach deals with packages in an object-oriented context instead of subsystems and provides first a global view and then a focus on a single package and its interactions with all other packages in the system.

Component recovery is used to control complex legacy systems [15] [32]. These approaches retrieve components whereas we qualify predefined packages. Koschke [14] uses weighted dependencies between program entities to group them.

Most of the tools that address the problem of large scale software visualization do not have such a fine degree of granularity for the dependencies as our approach. Some of the tools that do have a finer granularity do not scale. Our approach differs in offering a top down approach to first comprehend the system as a whole and then graphically exposing relationships between packages at a fine degree.

Metrics. Metrics are a way to assess the quality and complexity of software [12]. Briand *et al.* provide a conceptual framework to categorize metrics related to cohesion and coupling [3, 4]. However, they flatten inheritance, *i.e.,* a class is the sum of all its superclasses behavior. Relying exclusively on the cohesion of package to understand them is limited since packages convey more semantical information related to the intent of the developer or the organisation in which the application is developed. Therefore weakly cohesive packages make sense. It is our goal to evaluate whether some of the described metrics can provide a better information that the one we obtain with our simple model.

9 Conclusion

We presented an approach that supports the reengineer in obtaining a mental picture of an object-oriented system, understand its packages and cope with its complexity using a top-down reverse engineering approach based on visualization. It targets the first phase of reverse engineering complex software systems.

The main idea is that we consider packages as first class entities that we enrich with semantic information describing the package. We increase the abstraction level as we observe packages instead of classes. We provide a polymetric view and complement it with two radar visualizations that help to understand and categorize packages. The advantage of the polymetric view is that while it visualizes a complete system in terms of packages and dependencies between packages, the reengineer is not flooded with information, but can focus on interesting packages that he can further explore using the two radar views. The radar views not only show how a package relates to the rest of the system, but also how it is internally structured.

References

[1] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *Proceedings of WCRE'99*, pages 235–255, 1999.

[2] V. Basili. Evolving and packaging reading technologies. *Journal Systems and Software*, 38(1):3–12, 1997.

[3] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering: An International Journal*, 3(1):65–117, 1998.

[4] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[5] E. J. Chikofsky and J. H. Cross, II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[6] M. P. Consens and A. O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding of SIGMOD'93*, pages 511–516, 1993.

[7] T. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.

[8] U. Dekel. Revealing java class structures using concept lattices. Master thesis, Technion-Israel Institute of Technology, Feb. 2003.

[9] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — the FAMOOS information exchange model. Technical report, University of Bern, 2001.

[10] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE 2000*, pages 467–476, 2000.

[11] S. G. Eick, J. L. Steffen, and S. Eric E., Jr. SeeSoft—A Tool for Visualizing Line Oriented Software Statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, Nov. 1992.

[12] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1996.

[13] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of ISAAC '95*, 1995.

[14] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[15] R. Koschke. Atomic architectural component recovery for program understanding and evolution. In *Proceedings of ICSM'02*, 2002.

[16] M. Lanza and S. Ducasse. Polymetric views — a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[17] B. P. Lientz and E. B. Swanson. *Software Maintenance Management*. Addison Wesley, 1980.

[18] D. Littman, J. Pinto, S. Letovsky, and E. Soloway. Mental models and software maintenance. In Soloway and Iyengar, editors, *Empirical Studies of Programmers, First Workshop*, pages 80–98, 1996.

[19] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proceedings of ICSM'99*, 1999.

[20] B. S. Mitchell, S. Mancoridis, and M. Traverso. Search based reverse engineering. In *Proceedings of SEKE'02*, pages 431–438, 2002.

[21] B. S. Mitchell, S. Mancoridis, and M. Traverso. Using interconnection style rules to infer software architecture relations. In *Proceedings of GECC'04*, 2004.

[22] H. A. Müller. *Rigi — A Model for Software System Construction, Integration, and Evaluation based on Module Interface Specifications*. PhD thesis, Rice University, 1986.

[23] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.

[24] X. Pintado. The affinity browser. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 245–272. Prentice-Hall, 1995.

[25] M. P. Smith and M. Munro. Runtime visualisation of object oriented software. In *Proceedings of the International Workshop on Visualizing Software for Understanding and Analysis*, page 81. IEEE Computer Society, 2002.

[26] J. T. Stasko. Tango: A framework and system for algorithm animation. *IEEE Computer*, 23(9):27–39, Sept. 1990.

[27] J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price, editors. *Software Visualization — Programming as a Multimedia Experience*. The MIT Press, 1998.

[28] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Software Systems*, 44:171–185, 1999.

[29] M.-A. D. Storey and H. A. Müller. Manipulating and documenting software structures using shrimp views. In *Proceedings of ICSM'95*, 1995.

[30] D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings of ECOOP '89*, pages 25–38, 1989.

[31] M. Traverso and S. Mancoridis. On the automatic recovery of style-specific architectural relations in software systems. In *Proceedings of ASE 2002 (Conference on Automated Software Engineering*, pages 331–360, 2002.

[32] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of ICSE'99*, pages 246–255. ACM, 1999.

[33] C. Ware. *Information Visualization*. Morgan Kaufmann, 2000.

[34] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.