Butterflies: A Visual Approach to Characterize Packages [*]

Stéphane Ducasse
Software Composition Group
Univ. of Bern, Switzerland
ducasse@iam.unibe.ch

Michele Lanza
Faculty of Informatics
Univ. of Lugano, Switzerland
michele.lanza@unisi.ch

Laura Ponisio
Software Composition Group
Univ. of Bern, Switzerland
ponisio@iam.unibe.ch

Abstract

Understanding sets of classes, or packages, is an important activity in the development and reengineering of large object-oriented systems. Packages represent the coarse-grained structure of an application. They are artefacts to deploy and structure software, and therefore more than a simple generalization of classes. The relationships between packages and their contained classes are key in the decomposition of an application and its (re)-modularisation. However, it is difficult to quickly grasp the structure of a package and to understand how a package interacts with the rest of the system. We tackle this problem using butterfly visualizations, i.e., dedicated radar charts built from simple package metrics based on a language-independent meta-model. We illustrate our approach on two applications and show how we can retrieve the relevant characteristics of packages.

Keywords: Program understanding, package metrics, reverse engineering, software visualization

1. Introduction

It is well-known that 50% to 75% of the overall cost of a software system is devoted to its maintenance [20]: During maintenance, software professionals spend at least half their time reading and analysing software in order to understand it [8] [2]. The maintenance of object-oriented applications is harder than the ones written in procedural languages [30] because the presence of inheritance and late-binding greatly increases the number of potential dependencies within a program [30, 28, 12, 9].

In addition, nowadays most applications are structured in terms of sets of classes, or *packages*. In the context of object-oriented applications with late-binding and frameworks, packages have different roles: they may contain some key subclasses extending a framework [18] or they may contain utility classes used throughout the system. Therefore, packages do not have to follow the same design guidelines as classes to be well designed, as they may represent code ownership or the deployment process. However, the way a system is decomposed into packages and the way classes are distributed in them represent important characteristics of the application design and constrains the development process. Consequently it is crucial to understand packages in their structure and relations with other program elements such as classes. Providing a way to support the understanding of packages (or other sets of classes) is important also in the context of reengineering.

Nowadays packages exist in various languages such as Java, C#, Smalltalk and Squeak, CommonLisp, and Ruby. They present different functionalities *e.g.,* in Java packages serve as namespaces, while in Smalltalk not, and in Smalltalk classes can be extended (new behavior can be changed on a class not belonging to a package). Packages in all these languages share some common key functionalities: they contain classes and classes are in relationships with other classes.

Our approach is based on a language independent meta-model of source code and on the definition of measurements based on these relationships. Using the relationships state access, class reference and inheritance, we build dedicated radar charts named *butterflies* to characterize packages in terms of their internal structure and relationships with the entire application. The contributions of this article are (1) the definition of some simple metrics that support the characterization of packages and (2) the presentation of two butterfly visualizations of packages.

Structure of the paper. In Section 2 we discuss the problems of understanding the packages that compose an application. In Section 3 we present our approach. In Section 4 we show how we characterize packages with the information that we extract from the source code. In Section 5 we present the butterfly views, give examples and analyze the results of applying our approach to the case studies. In

[*] In Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)

Section 6 we elaborate some discussion. In Section 7 we refer to related work before concluding in Section 8.

2. Understanding And Characterizing Packages

Chikofsky and Cross state 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"* [6]. 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 provide a means to assess the quality of packages during the restructuring of a system. In this paper we consider as package a group of classes that a developer has decided to put together. Our approach supports the characterization of packages without being tied to a particular language: Our approach is not based on a particular implementation language because the underlying metamodel is language independent [10]. We leave for future work to refine the proposed framework in the context of specific language construct semantics.

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 it contains and its efferent and afferent relationships? 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 a 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 [27].

Packages are complex entities because they contain classes that can have multiple relationships. Therefore characterizing packages requires processing and reporting a lot of information. We tackle this complexity by combining metrics and visualization.

Software metrics are well-known to reduce and abstract large amounts of information [13]. However, this 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. Our goal in this paper is not to define complex metrics but to use some simple metrics to support the visual characterization of packages.

Software visualization allows us to visually combine multiple aspects of complex problems [26] [29]. However, software visualizations are often too simplistic and lack visual cues for the viewer to correctly interpret them [24]. In other cases the obtained visualizations 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 we can *compare* and *identify* multiple packages at the same time. We want our solution to be applicable with limited tool support, *i.e.,* using a tool such as MS Excel, with the advantage that people can apply butterfly views in a broad variety of contexts. Moreover, they should not be limited to a specific language, tool or environment. Moreover, butterfly views can easily be introduced in advanced integrated development environments such as Eclipse.

3. Approach Overview

We adopt a top down approach: The reengineer first uses an adapted polymetric view [19] as coarse-grained visualization of the system with all its packages and their connections and then uses butterfly views to get detailed information about specific packages. We do not present the used polymetric view in this paper to focus on the main contribution, *i.e.,* the butterfly view. The interested reader should refer to [11].

Two butterfly views, *i.e.,* specific radar charts, are provided: (1) a GLOBAL BUTTERFLY where the package is compared with its surrounding context and (2) a RELATIVE BUTTERFLY where the package is analyzed on its own. Both views are described in detail in Section 5.

Case studies. We took as case studies BASE VISUALWORKS and CODECRAWLER:

- BASE VISUALWORKS is a large portion of the Cincom VisualWorks Smalltalk environment[1]. 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.

1 See http://www.cincomsmalltalk.com for more information.

- CODECRAWLER is a small software visualization tool [19][2] and serves to illustrate examples in detail.

Case Study	Packages	Classes	LOC
BASE VISUALWORKS	94	1402	262660
CODECRAWLER	8	93	9088

4. 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

We chose to take the minimal information that reveal the essence of a package in the context of an object-oriented application. For that reason we focus on class references, inheritance relationships, and use of state and behavior. An important influence on this work is the focus on the object-oriented context in which packages exist: In object-oriented applications inheritance hierarchies can be spread over multiple packages. Therefore flattening packages ignoring the inheritance relationships is not satisfactory for a precise characterization, 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, 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 relationships, or *dependencies* between classes:

1. *Inheritance*: a class is a subclass of another. It inherits its behavior.
2. *State*: a class may use instance variables inherited from its ancestors.
3. *Class Reference*: a class makes an explicit reference of another *e.g.,* by instantiating the class.

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

The use of facade classes could obscure the visualization of the structure of the package. However, our approach characterizes packages independently of the specific class to which the dependency is directed, enhancing the understanding of the package structure from its role as client and provider. The effect of increased internal class dependencies, *e.g.,* due to the use of a facade class, is considered independently in an extended visualization that relates the internal and external dependencies.

4.2. Characterizing Packages

To condense the information of a large application at the level of its packages, we use object-oriented metrics based on the dependencies we defined previously.

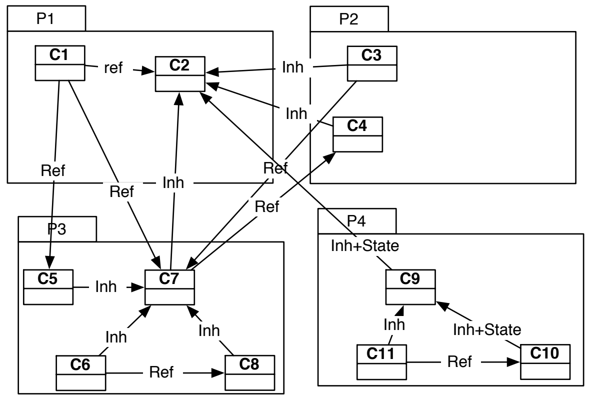


Figure 1. Some packages with class dependencies (C1 refers to C2, or inherits of C2, or is a client of C2).

The metric 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 or vice versa. The metric example values refer to the situation depicted in Figure 1.

We define both *absolute* and *relative* metrics for packages:

- *Absolute metrics* count the dependencies of a given kind and direction. 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 (providers) from classes belonging to the analyzed package (a client). This metric is useful to assess whether a package (and its classes) is heavily using other packages.
- *Relative metrics* show the relationship between the amount of internal and external dependencies of a

2 See http://www.iam.unibe.ch/ scg/Research/CodeCrawler/ for more information.

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.

given type and direction in the package. They follow the pattern:

$$property/(property + internalproperty)$$

For instance, 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. Butterflies: Radar Charts for Packages

Obtaining a detailed understanding of packages is difficult since packages are complex entities: they contain classes which may have different relationships with other classes, either within the same package or defined in other packages and this in presence of late-binding and specialization [30, 28, 12].

To cope with this situation, we use dedicated radar charts which combine several metrics about a package in a single space. The first view, GLOBAL BUTTERFLY, presents a package in the context of the complete system. The second view, RELATIVE BUTTERFLY, presents how the package is internally structured.

5.1. Butterfly 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 2. The radar visualization generates an irregular surface which is greater if two contiguous axes represent higher values. However, using a radar visualization to represent complex constructs is not straight-forward 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 analyzed and how they are mapped efficiently on a radar chart.

As packages provide and use 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 package provides to other packages, while the right wing represents what the package 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 whether the package extends other packages.

5.2. Global Butterfly

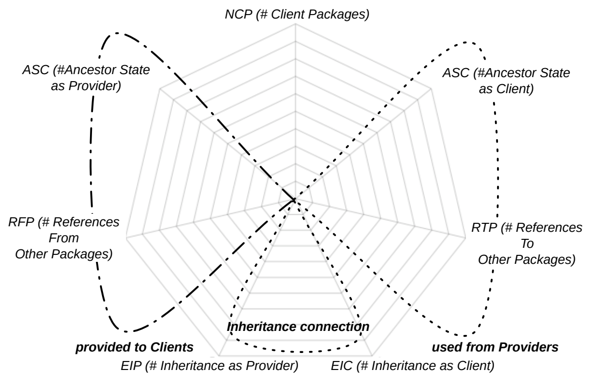

NCP (# Client Packages)
ASC (#Ancestor State as Provider)
ASC (#Ancestor State as Client)
RFP (# References From Other Packages)
RTP (# References To Other Packages)
Inheritance connection
provided to Clients
used from Providers
EIP (# Inheritance as Provider)
EIC (# Inheritance as Client)

Figure 2. Principles of the Global Butterfly.

This view characterizes a package as presented in Figure 2. It displays information that compares the package in the context of the complete application.

Example. Figure 3 displays the Global Butterfly of the packages CCCore, CCBase and CCUI from the CodeCrawler case study.

CCBase. Its shape leaning towards the left shows that this 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 (for example CCCore) and that the package also accesses state of other packages. A close inspection of the code reveals that the references to other packages are the ones to default types such as String and Collection.

CCCore. It is a central package of CodeCrawler. This is reflected by the fact that the butterfly has two even, long and horizontally symmetric wings. It uses the package CCBase. The view indicates that this package uses 86 external classes while it defines 22 classes. The classes it defines are referenced from other packages too (RFP (Number of Class References From Other Packages) = 58). EIC (Number of External Inheritance as Client) shows that this package inherits from 10 classes in the other packages, but this package is also extended (EIP (Number of External Inheritance as Provider) = 3). This package does not directly use state from the superclasses which is an indication of good design. We also learn that its state is directly accessed by subclasses defined in other packages (ASP (Number of Ancestor State as Provider) = 2).

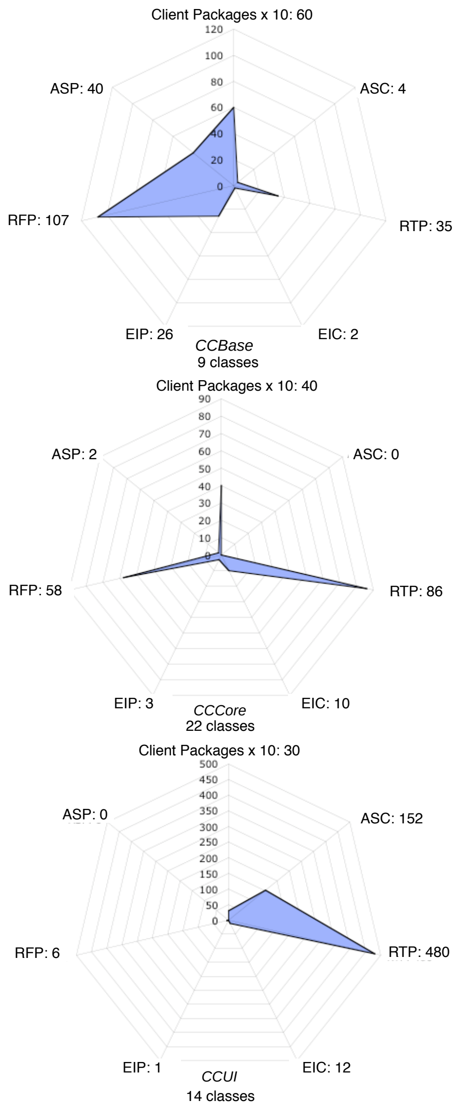


Figure 3. Global Butterfly on the CodeCrawler packages CCCore, CCBase and CCUI.

As the package contains 22 classes and EIC (Number of External Inheritance as Client) is 10, we learn that the package is not flat inheriting solely from a couple of root classes but that it is certainly composed of inheritance hierarchies.

CCUI. The Global Butterfly of CCUI shows that it is mainly a client: its classes directly access attributes of provider superclasses (ASC (Number of Ancestor State as Client) = 152). This package will be impacted if the superclasses located in other packages change. The high-value

(480) of RTP (Number of Class References To Other Packages) is due to the manual building of menus *e.g.*, direct instantiations of MenuItem. This shape was expected, because CCUI contains all the CODECRAWLER UI elements.

5.3. RELATIVE BUTTERFLY

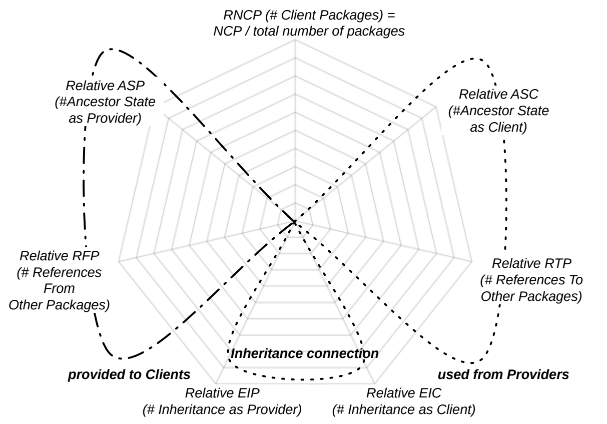

RNCP (# Client Packages) = NCP / total number of packages
Relative ASP (#Ancestor State as Provider)
Relative ASC (#Ancestor State as Client)
Relative RFP (# References From Other Packages)
Relative RTP (# References To Other Packages)
Inheritance connection
provided to Clients
used from Providers
Relative EIP (# Inheritance as Provider)
Relative EIC (# Inheritance as Client)

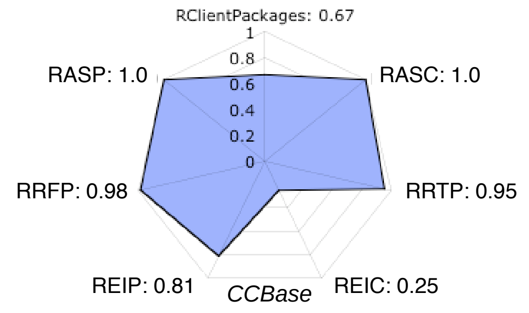
Figure 4. Principles of the RELATIVE BUTTERFLY.

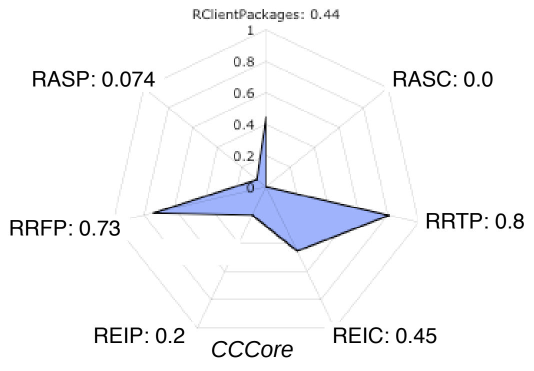
While the GLOBAL BUTTERFLY provides information about a package, it does it by measuring the package in the context of the complete system. However, it is difficult to assess how a property exists *in the context* of the package itself. For example, the information that a package defines a lot of classes 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 BUTTERFLY, whose principles are described in Figure 4. To minimize context-switching it has the same axes as the GLOBAL BUTTERFLY, but uses relative metrics described 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 package compared to the outside. For example, RASP (Relative Number of Ancestor State as Provider) of CCbase in Figure 5 is 1, which means that there is no state access between the classes inside the package.

When RRTP (RelativeNumber of Class References To Other Packages) is equal to 1, it means that there is a weak class reference dependency between the classes inside the package compared to the class reference dependencies they have with other classes outside the package.

As the following example illustrates, there is an interplay between the two views. In particular the information displayed by the GLOBAL BUTTERFLY allows one to qualify the finer level of description given by the RELATIVE BUTTERFLY.





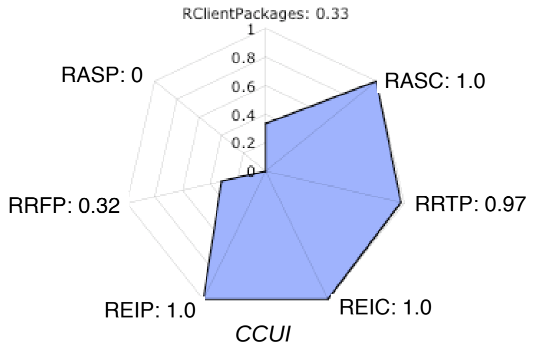


Figure 5. RELATIVE BUTTERFLY on the CodeCrawler packages CCCore, CCBase and CCUI.

Example. Figure 5 shows the RELATIVE BUTTERFLY views of three packages of CODECRAWLER: CCBase, CCCore and CCUI.

CCBase. We see that its classes do not directly access state, since RASP (Relative Number of Ancestor State as Provider) and RASC (Relative Number of Ancestor State as Provider) are 1. This happens even when such classes are

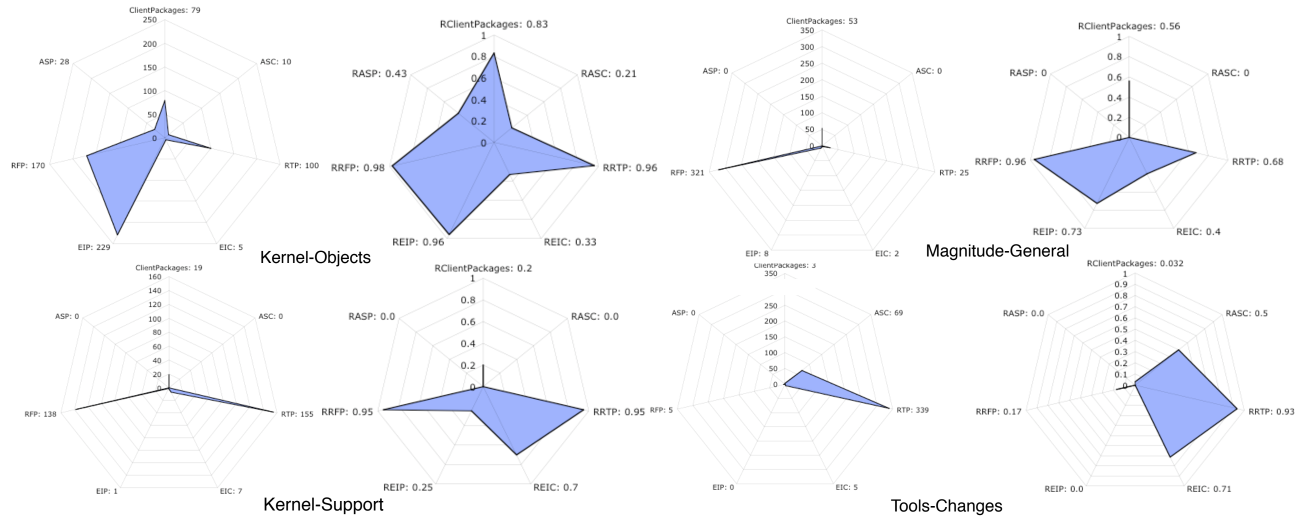


Figure 6. Butterfly views of selected packages of BASE VISUALWORKS**.**

accessing the state of external superclasses (ASC (Number of Ancestor State as Client)= 4) and their state is accessed by clients classes (ASP (Number of Ancestor State as Provider)= 40, as we saw in Figure 3). As the value of REIC (Relative Number of External Inheritance as Client)is 0.25, we learn that this package has 3 times more internal inheritance than it is inheriting from others. REIP (Relative Number of External Inheritance as Provider) = 0.81 indicates that it is more subclassed from the outside than from the inside. In fact it indicates that it is 26 times subclassed from other packages whereas there are only 6 inheritance dependencies in the package. However, it could still be the case that its classes are much more subclassed: A class can be subclassed by a class in another package that then acts as another hierarchy root to numerous classes.

CCCore. Considering CCCore, we see that it does not access the state of other packages (RASC (Relative Number of Ancestor State as Client) = 0). It has more references to the outside than references between the classes inside the package (RRTP (RelativeNumber of Class References To Other Packages) = 0.8) and it has a bit more references from other packages (RRFP (Relative Number of Class References From Other Packages) = 0.73) than internal class references. REIP has a value of 0.2 which means that the package has a lot more internal inheritance relationships than it has direct subclasses.

CCUI. Regarding CCUI we see that the REIC (Relative Number of External Inheritance as Client) value of CCUI ($REIC = EIC/(EIC + PII)$) is 1. This confirms that it does not define an inheritance hierarchy. Interpreting RRTP (RelativeNumber of Class References To Other Packages) whose value is 97%, we learn that the package classes have few class references among them, because there are 480 references to external classes and only 3% of internal references (*i.e.*, 14 internal references). RRFP (Relative Number of Class References From Other Packages) is 32%, since there are 6 external and 14 internal references (RFP (Number of Class References From Other Packages)).

5.4. BASE VISUALWORKS Case Study

We applied our approach to a large case study (BASE VISUALWORKS) and selected some characteristic packages displayed in Figure 6. The butterfly views reveal some typical situations:

Kernel-Objects. It is mainly a provider package. It contains some major inheritance hierarchy root classes such as Object and Model. It contains some important classes such as Boolean, True, False and some key subclasses. This package is heavily subclassed (EIP (Number of External Inheritance as Provider) = 229) and its classes are considerably referenced (RFP (Number of Class References From Other Packages) = 170).

Out of its 16 classes, 5 classes subclass classes belonging to other packages (EIC (Number of External Inheritance as Client) = 5). Kernel-Objects has classes that directly access attributes of their superclasses located in other packages (ASC (Number of Ancestor State as Client) = 10). We learn also that its state is accessed by subclasses defined in other packages (ASP (Number of Ancestor State as Provider) = 28). In the source code we see that the class

Model attribute 'dependents' is referenced by classes in the packages Interface-Models and Interface-Support.

The inheritance connection situated at the bottom of the relative butterfly view shows that there is inheritance inside this package: (REIC (Relative Number of External Inheritance as Client) = 0.33 and REIP (Relative Number of External Inheritance as Provider) = 0.96). Indeed, it contains the hierarchy involving the classes Boolean, False and True, for instance. The relative butterfly view shows that a high percentage of the packages in the system depend on this one (RClientPackages (Relative Number of Client Packages) = 0.83).

Kernel-Support. This package 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. Its clients are only 2% of the packages in the system.

Magnitude-General. This package is a provider package which merely contains the abstract class Magnitude, and the concrete classes Date, Time, TimeZone and Character. The needle-like butterfly left wing in the axis corresponding to RFP indicates that it is heavily used (RFP (Number of Class References From Other Packages) = 321) but it does not use many classes. In fact, its classes have few direct references to classes in other packages (RTP (Number of Class References To Other Packages) = 25) and do not use state from ancestors (ASC (Number of Ancestor State as Client) = 0).

However, its clients are more than half of the packages (RClientPackages (Relative Number of Client Packages) = 0.56), but only accessing it directly or through inheritance (EIP (Number of External Inheritance as Provider) = 8 and ASC (Number of Ancestor State as Client) = 0). The RELATIVE BUTTERFLY indicates that there are few references among the classes in this package (RRFP (Relative Number of Class References From Other Packages)= 0.96) but there is some inheritance.

Tools-Changes. This package has a client shape. This is 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 of UIBuilder-Framework. It has 7 classes, 5 of which have their superclasses in other packages. In fact, 4 of its classes subclass ApplicationModel, defined in the package UIBuilder-Framework.

The RELATIVE BUTTERFLY shows that 93% of the direct references are to classes in other packages (RRTP (RelativeNumber of Class References To Other Packages) = 0.93). That leaves only 7% of direct references existing inside the package. The RELATIVE BUTTERFLY also shows that there is inheritance in the package (REIC (Relative Number of External Inheritance as Client) = 0.71).

6. Discussion

Our approach is based on a simple metamodel of source code and measurements. The butterfly views depict how a package is internally structured and how it relates to the rest of the system. Butterfly views has proven to be successful to provide insights about the system structure in terms

Program Understanding Context. The butterfly views are the main contribution of this article, but they are part of a larger process in which polymetric views [19] are used to offer an overview of all the packages an application is composed of [11]. In addition we support opportunistic understanding [21] in the sense that the user browses *if necessary* the package and the code it contains. 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 that allow a precise characterization.

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 axis labels provides useful complementary information. Determining the order of the axes is a challenging task, as a different order can produce different shapes. We tried and evaluated several configurations before obtaining the butterfly configuration that communicates the role of the package presented in this paper. In addition the user should be trained as with any visualization technique.

Visualization Concerns. From a visualization point of view, our approach has the following properties:

- Butterfly views condense information using a minimal amount of screen space. This allows the user to see multiple butterfly views at the same time and compare them.
- Because a butterfly view represents one package, there is a direct mapping between the subject and its representations.
- Butterfly views, contrary to time wheel visualizations [7] are not rotation invariant, however their symmetric shape offers a good *gestalt* effect and makes them easily identifiable.
- Butterfly views are not interactive. This is not intrinsic to them but the result of a choice to have an approach which can be easily done with different tools.

Using Other Metrics. Our current approach uses a set of simple metrics that describe relationships between classes. We would like to investigate the use of other advanced metrics describing the cohesion, coupling or the stability over time of the packages. However, current metrics on cohesion and coupling mainly focuses on classes. Such class cohesion/coupling metrics are difficult to apply to packages since packages are not a simple generalization of classes but play different roles in the software development process. In addition current cohesion metrics for classes do not take into account inheritance as they flatten it [17, 3, 4]. Metrics such as LCOM [5] have been heavily criticized and as such have little value. It is our goal to evaluate whether some cohesion metrics *e.g.,* LCOM* [16] or other cohesion and coupling metrics [1, 23] can support our visual characterization.

Current Limits. Even if the current approach is effective for getting a detailed view on packages, a common problem in radar plots is that not all metrics can be mapped onto a value between 0 and 1, *e.g.,* size. Figure 3 is a case in point. In addition, there are still questions we plan to investigate:

An advantage of our approach is that butterfly views can be generated with MS Excel or any simple chart-drawing tool. More information could be added following the infobug approach [7], but it would be at the expense of simplicity of interpretation. In the future we would like to enhance the butterfly views with information about the classes, or the evolution of the classes in a package.

We do not take into account invocations. Introducing them may lead to other views on coupling and cohesion but may introduce noise due to late-binding, *e.g.,* an invocation can have multiple potential receivers.

Because we only consider the direct relationships a class has, we do not assess whether a class belonging to a package is central to an application. To do this, we plan to introduce transitive relationships such as counting the total number of subclasses instead of the direct ones. Moreover, such information can be retrieved by opportunistic code reading.

The butterfly 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 butterfly views, which is left as future work.

7. Related Work

Researchers have long realized of the usefulness of structural metrics for object oriented systems. They derived many design metrics from the ones originally thought for structured programming. However, these metrics [14] [16] focus on classes or whole systems, and generally not on sets of classes, or packages. Hautus introduced Pasta, a tool to analyze the structure of Java programs and a metric to determine the quality of the package architecture [15]. Allen et al. defined information theory-based — as opposed to counting — coupling and cohesion metrics for modules [1] that are represented as graphs. They define module and intramodule in terms of the subgraph's information and cohesion in terms of intramodule coupling. However this approach does not take into account classes, inheritance and their relationships. Already, metrics for packages such as efferent and afferent couplings, abstractness, instability, distance and cycle have been applied successfully to the empirical analysis of java packages [22].

Briand *et al.* provide a conceptual framework to categorize metrics related to cohesion and coupling [17, 3, 4]. However, they flatten inheritance, *i.e.,* a class is the sum of all its superclasses behavior and rely exclusively on the cohesion of package to understand them. This approach is limited because packages convey more semantical information related to the intent of the developer or the organisation in which the application is developed.

Graphical representations of software have long been accepted as comprehension aids. Many tools enable the user to visualize software using static information. Sharble and Cohen introduce the use of a compass-type plot for eight metrics [25]. We apply this idea to describe packages by their role in the system. The difference with their work is that our approach exploits the vertical symmetry to characterize the package as client or provider.

To the best of our our knowledge, only the infobug visualization [7] tries to support the understanding of files in a glyph oriented way as our butterflies do. Chuah and Eick present a way to visualize project information through glyphs. Glyphs are graphical objects representing data through visual parameters. The difference with our work is that they use glyphs for viewing project management data (*i.e.,* evolution aspects, programming languages used, and errors found in a software component), while our work focuses on describing how a package relates to the rest of the system in a fine-grained way, i.e. visualizing how a package relates with others according to different dependency types (inheritance, state, or class reference) and the role that the package plays in the system (client, central, or provider).

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. Our approach differs in exposing relationships between packages at a fine degree.

8. Conclusion

We presented a novel approach that supports the reengineer in obtaining a mental picture of an object-oriented system, understand its packages and cope with its complexity.

The main idea is that we consider packages as first class entities that we enrich with metrics describing them. We provide two radar visualizations named butterfly views that help to understand and categorize packages. The butterfly views not only show how a package relates to the rest of the system, but also how it is internally structured. The goal of the work presented here is to support reengineers but also researchers working on remodularisation to get a better understanding of object-oriented applications.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, Sept. 2002 - Aug. 2006).

References

[1] E. Allen and T. Khoshgoftaar. Measuring coupling and cohesion of software modules: An information theory approach. In *Seventh International Software Metrics Symposium*, 2001.

[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] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[6] E. J. Chikofsky and J. H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[7] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, pages 24–29, July 1998.

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

[9] U. Dekel. Revealing JAVA Class Structures using Concept Lattices. Diploma thesis, Technion-Israel Institute of Technology, Feb. 2003.

[10] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[11] S. Ducasse, M. Lanza, and L. Ponisio. A top-down program comprehension strategy for packages. Technical Report IAM-04-007, University of Berne, Institut of Applied Mathematics and Computer Sciences, 2004.

[12] A. Dunsmore, M. Roper, and M. Wood. Object-Oriented Inspection in the Face of Delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

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

[14] N. Fenton, S. L. Pfleeger, and R. L. Glass. Science and Substance: A Challenge to Software Engineers. *IEEE Software*, (7):86–95, July 1994.

[15] E. Hautus. Inmproving java software through package structure analysis. In *International Conference Software Engineering and Applications*, 2002.

[16] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[17] M. Hitz and B. Montazeri. Measure coupling and cohesion in object-oriented systems. *Proceedings of International Symposium on Applied Corporate Computing (ISAAC '95)*, Oct. 1995.

[18] R. E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92*, volume 27, pages 63–76, Oct. 1992.

[19] 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.

[20] B. Lientz and B. Swanson. *Software Maintenance Management*. Addison Wesley, 1980.

[21] 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. Ablex Publishing Corporation, 1996.

[22] Maven. http://maven.apache.org.

[23] V. B. Mišić. Cohesion is structural, coherence is functional: Different views, different measures. In *Proceedings of the Seventh International Software Metrics Symposium (METRICS-01)*. IEEE, 2001.

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

[25] R. C. Sharble and S. Cohen. The object-oriented brewery: a comparison of two object-oriented development methods. *ACM SIGSOFT, Software Engineering Notes*, 18(2):60–63, 1993.

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

[27] 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.

[28] D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings ECOOP '89*, pages 25–38, Nottingham, July 1989. Cambridge University Press.

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

[30] N. Wilde and R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.