Package Reference Fingerprint: a Rich and Compact Visualization to Understand Package Relationships

Hani Abdeen [1]* Ilham Alloui [2] Stéphane Ducasse [1]† Damien Pollet [3] Mathieu Suen [1]

[1] ADAM team, LIFL INRIA Lille-Nord Europe UMR CNRS 8022–France

[2] Université de Savoie France

[3] University of Lugano Switzerland

Abstract

Object-oriented languages such as Java, Smalltalk, and C++ structure their programs using packages, allowing classes to be organized into named abstractions. Maintainers of large applications need to understand how packages are structured and how they relate to each other, but this task is very complex because packages often have multiple clients and different roles (class container, code ownership...). Cohesion and coupling are still among the most used metrics, because they help identify candidate packages for restructuring; however, they do not help maintainers understand the structure and interrelationships between packages. In this paper, we present the package fingerprint*, a 2D visualization of the references made to and from a package. The proposed visualization offers a semantically rich, but compact and zoomable visualization centered on packages. We focus on two views (incoming and outgoing references) that help users understand how the package under analysis is used by the system and how it uses the system. We applied these views on three large case studies: JBoss, Azureus, and ArgoUML.*

This paper uses colors in the figures. Please read a colored printout of this paper .

Index Terms Software packages, Visualization.

1 Introduction

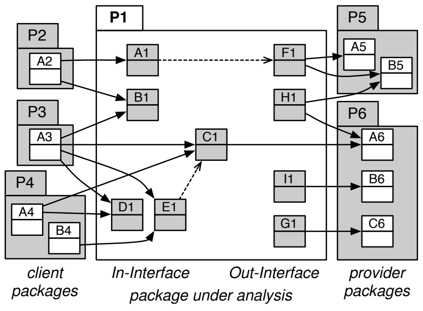
To cope with the complexity of large object-oriented software, developers organize classes into packages or modules. This organization usually follows the conceptual interrelationships between classes, but as a system evolves, its modular structure may change and require maintenance. In this context, it is useful to understand the concrete organization of packages and their interrelationships. Ideally, packages should be kept as less coupled and as much cohesive as possible [3]. Following previous overviews [16, 13, 1, 15], we distinguish two main design approaches. The first approach relates the cohesion of a package to the interconnections between its internal classes. The second approach relates the cohesion of a package to how the system uses this package's internal classes, *i.e.*, if two classes of a package are used from a same client module, then they are considered as conceptually related, regardless of the explicit relationships that exist between them [15]. This second approach is more meaningful to us, because we consider a package as a functionality provider and not only a structural grouping of coupled classes. Many metrics of package cohesion have been defined [3, 13, 1, 15] and help to determine packages that are candidates for restructuring during maintenance. However, those approaches do not help maintainers of large applications when they face the problem of understanding how packages are structured in general and how packages are in relation with each other in their provider/consumer roles.

Several previous works in software visualization provide information on packages and their relationships, by visualizing software artifacts or metrics about their structure or evolution [5, 6, 12, 14, 17, 18]. However, while these approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the structure of packages, their interrelationships within the system, and help identifying their roles within an application.

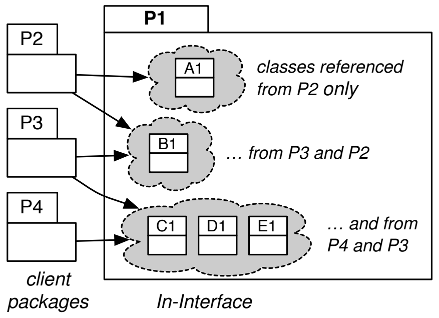
Contribution. In this article, we present the Package Reference Fingerprint: a compact, rich and zoomable visualization supporting the understanding of a package and its relationships. We propose two complementary variations of the Package Reference Fingerprint, structured around the distribution of references from or to the classes of the analyzed package: the *incoming fingerprint* shows how the system uses the package's classes, and highlights the cohesion of the analyzed package, as defined in [15]; the *outgoing fingerprint* shows how the package's classes use the system.

*hani.abdeen@inria.fr

†We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872) and ESUG the European Smalltalk User Group for its sponsoring.



(a) The analyzed package P_1 is composed of nine classes among which only four are involved in internal references (A1 with F1 and E1 with C1). Both In-Interface and Out-Interface of P_1 contain five classes, with C1 in common.



(b) Grouping the classes of the In-Interface of P_1 by common client packages. Similarly, we group classes in the Out-Interface by common provider packages.

Figure 1. Defining our terminology: an example of references between packages.

In Section 2, we first describe the challenges that software engineers face when trying to understand packages. Section 3 presents the Package Reference Fingerprint and its basic principles. Section 4 presents how a fingerprint works in practice. Section 5 presents a coarser level of detail of the fingerprint, and shows what information it provides. Section 6 presents the outgoing fingerprint. In Section 7 we list the most common patterns we identified during our experimentation. In Section 8 we discuss our visualization and we report on related work before concluding.

Vocabulary. Figure 1(a) illustrates the terminology we use in the rest of the paper. First, by *reference*, we mean that a package P_1 refers to an another P_2 if P_1 contains a class C1 that invokes methods of an another class C2 packaged in P_2. In the same vein, we say that P_1 refers to C2, C1 refers to P_2, P_2 exports C2 for P_1, and P_2 is referenced by C1 and by P_1. By language abuse, when we say that a package P refers to another package Q, we mean that classes contained in P refer to classes of Q. We name *In-Interface of P* the set of classes of P which are referenced by classes packaged outside P. Similarly, we name *Out-Interface of P* the set of classes of P that refer to classes packaged outside P.

2 Challenges in Understanding Packages

Although languages such as Java model dependencies between packages (*i.e.*, via the import statement), developers lack tool support to really understand packages within their context. Indeed, packages are complex entities that have different usage patterns, often depending on the clients that use them, *e.g.*, code ownership, feature containment. This makes it difficult to understand the inter-packages communication or even to quickly identify the clients or providers of a package [7].

To understand the structure and the roles of packages within a system, we need both quantitative information (the size of elements and their relationships), and qualitative information (coupling and cohesion). In this section, we summarize the information that a solution supporting package understanding should provide. It is complementary to the list described in [7]. Note that this list is not exhaustive and that our approach does not cover all the points raised hereafter.

Quantitative information. What is the general size of a package in terms of classes, inheritance definitions, internal and external class references, imports, exports to other packages? Is there only a few classes communicating with the rest of the system?

The size of the package Out-Interface gives maintainers a quantified information about the dependency of the package on other packages, while the number of referenced packages shows its dependence on the system. The size of the package In-Interface gives maintainers a quantified information about the dependency of the system on the package, and the number of referencing packages shows its importance for the system.

Qualitative information: cohesion and coupling. Good packages should group conceptually coupled classes, *i.e.*, which are needed for the same task [15], and they should have a few clear dependencies to other packages [3, 11]. In such a context, cohesion and coupling are among the most used metrics during perfective maintenance [13, 1, 16].

To understand the multiple facets of a package, we try to group its classes according to their usage by other packages. Figure 1(b) shows referenced classes of P_1 grouped into clusters as well as the references that point to those clusters. Clustering the In-Interface and Out-Interface like that helps identifying the dependencies that the system has with P_1, and thus which classes are conceptually coupled and which classes are not; at a higher level of abstraction, it helps answering the following questions: What functionality does P_1 provide? To which packages?

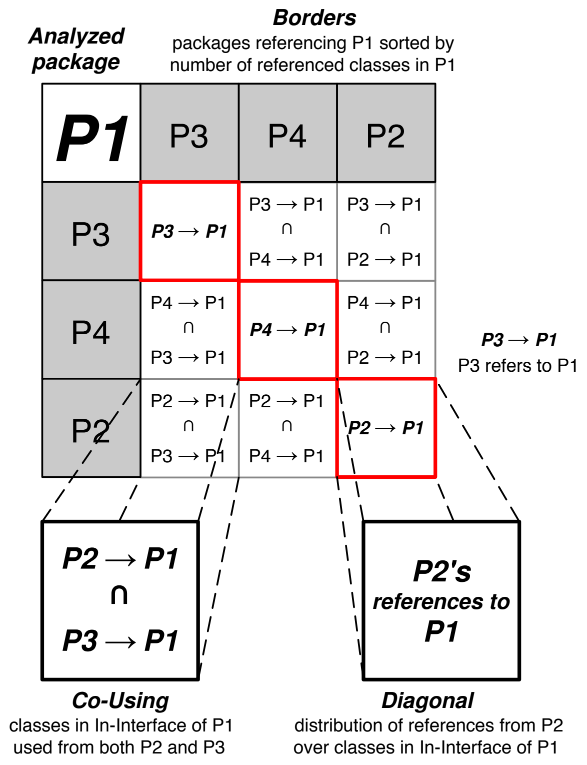


Figure 2. Incoming Fingerprint skeleton.

The other indicator of package design quality that we consider here is coupling. It is generally defined as: *if changing one package in a program requires changing another package, then coupling between these two packages exists* [2, 8]. In this paper we say that a package P is coupled with a package Q if P refers to Q, *i.e.*, at least one class of P refers to one class of Q. Figure 1(a) shows that P_1 is coupled with P_5 and P_6.

3 Package Reference Fingerprint

According to the requirements mentioned in Section 2, we propose two complementary views for *incoming* and *outgoing* references: the Package Reference Fingerprints (fingerprints for short).

The incoming fingerprint shows how the package under analysis is used by the system and how this use is distributed over its classes. The outgoing fingerprint shows how the package under analysis uses the remainder of the system.

For space reasons, and since we use the same approach for both views, we only present the incoming fingerprint in details. Fingerprints have the four following properties: they are *compact* (only the references are shown), *zoomable* (different levels of information are proposed), *entity-based* in the sense that they describe one package, and *semantically rich* since they present multiple information at a glance.

3.1 Fingerprint Skeleton

Figure 2 depicts the key visualization principles of a Package Reference Fingerprint with P_1 from Figure 1 as the package under analysis. In the context of an incoming fingerprint, the skeleton of the layout is the following:

Analyzed Package. The *top left corner* cell indicates global information about the package under analysis (here P_1): the size of its In-Interface and the internal references between its classes.

Referring Packages. The cells at the *borders* of the fingerprint, *i.e.*, the leftmost column and the topmost raw, both represent the referring packages. We sort these packages by order of importance: the most references a package does, the closer it is to the top left corner. Figure 2 shows the three packages that refer to P_1 in Figure 1: P_3, P_4, and P_2, making respectively four, three, and two references to P_1.

To order packages that make the same number of references, we group them by similarity; in an incoming fingerprint, we define the similarity of referencing packages as the number of shared referenced classes. For example, in the Figure 1(b) we consider that P_4 is more similar to P_3 (3 referenced classes in common) than to P_2 (no referenced class in common). Symmetrically, we define the similarity of referenced classes by the number of referencing packages they share. Figure 1(b) shows that C1 and D1 (2 common referencing packages) is higher than the similarity between C1 and B1 (1 common referencing package). In any case, the ordering algorithm we have implemented always respects the number of references prior to similarity.

Cells. The *body* cells of an incoming fingerprint, *i.e.*, all cells except those in the leftmost column or the topmost row, each represent a subset of the In-Interface of the package under analysis. This subset contains the classes that are referenced by *both* packages placed at the heads of the cell's row and column. For a package P that is referenced by $P_1, \ldots, P_n$, a cell on line i and column j, $\mathrm{cell}(i, j)$, represents the subset of classes of P that are referenced by both P_i, and P_j (*i.e.*, $cell(i, 1)$ and $cell(1, j)$). Two situations occur: either a cell is on the *main diagonal* or not.

- The *main diagonal*, it presents the distribution of the In-Interface on the using packages. Figure 3 shows that $\mathrm{cell}(3, 3)$ contains the classes (C1, D1, E1) referenced by P_4, *i.e.*, $\mathrm{cell}(3, 1)$ and $\mathrm{cell}(1, 3)$.
- The other cells present the classes accessed in *common* by both packages represented by the row and column heads, as just explained. Figure 3 shows that $\mathrm{cell}(2, 4)$ contains the class B1, referenced by both P_3 and P_2.

We define the size of a cell as the number of classes it contains. Hence in Figure 3, $\mathrm{cell}(2, 2)$ has a size of 4 and $\mathrm{cell}(3, 3)$ a size of 3: both cells represent the classes C1, D1, and E1, but the $\mathrm{cell}(2, 2)$ represents also the class B1.

3.2 Enriching the Fingerprint Skeleton Layout

We enrich the skeleton of Figure 2 to convey extra information such as the amount of referenced classes in the analyzed package. For this purpose we use color intensity of cells, cell borders, and the position of classes within cells.

We selected those visual properties according to several research works that address the characteristics of efficient visualizations [19, 20]. Particularly, as our focus is on providing a first impression of a package and its context, we want to exploit preattentive processing [1] as much as possible to help spotting important information [9, 20].

Cell Internals. Inside a cell, we show the package's referenced classes as small filled squares.

To enable pre-attentive processing [9], we give each class a fixed place which is the same for all the cells of a fingerprint. When a cell represents that a package refers to a class of the analyzed package, the location of this class is colored: in Figure 3, since the class B1 is referred to by the packages P_3 and P_2, the position corresponding to the class B1 is colored in the $\mathrm{cell}(2, 4)$. This way all the cells will have the same geometrical size (*i.e.*, height and width), but the number of classes represented by the cell is given by the number of the colored squares inside that cell.

Internal information. Information on internal references between classes of the analyzed packages are visualized on the *top left corner*. In Figure 3 we see that among the six referenced classes of P_1, only F1 and C1 are referenced internally. Additionally, since not all classes will appear in all cells, we use this corner cell to show all the placeholders for the classes that have incoming references, as bordered squares.

Colors. Colors are used in a fingerprint to distinguish between different entities (*e.g.*, classes, packages), and to give more information about the references. The colors are: (1) shades of grey for all the cells of a fingerprint except the top left corner, (2) blue for the classes (3) red for the top left corner and to highlight the borders of the main diagonal cells and to highlight the fingerprint borders, (4) to highlight packages that are outside the scope of the application under analysis (called stubs thereafter), we use a gold border, and we use the gold color also to highlight selected cells. (5) to highlight the selected classes we use the green color.

[1]Researchers in psychology and vision have discovered a number of visual properties that are preattentively processed. They are detected immediately by the visual system: viewers do not have to focus their attention on a specific region in an image to determine whether elements with the given property are present or absent. An example of a preattentive task is detecting a filled circle in a group of empty circles. Commonly used preattentive features include hue, curvature, size, intensity, orientation, length, motion, and depth of field. However, combining them can destroy their preattentive power. Some of the features such as motion are not relevant in our context.

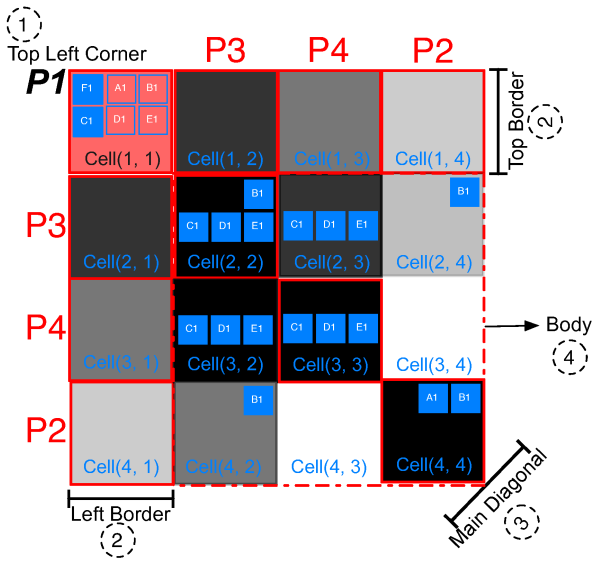


Figure 3. Showing the Incoming Fingerprint of P_1 (Figure 1 (b)) with the classes involved in the relations inside each cell.

Color Intensity. We vary the color *intensity* to give more information about the visualized entity: (1) for the top left corner, the darker the package, the bigger its In-Interface is; (2) for the borders, the darker a referencing package, the more classes it references in the analyzed package; (3) for the body, on *a given line*, the darker the cell, the more classes it contains. The darkness of a cell is calculated relatively to the size of the diagonal cell of that line. As a consequence the cells of the diagonal are black.

Figure 3 shows that P_3 is darker than P_4: the first package refers to 4 classes in P_1 while P_4 refers to 3 classes in P_1. The color of the top left corner is based on an In-Interface's size ratio: the size of the In-Interface of P_1 is 5 while the size of P_1 itself is 9 (Figure 1). Thus the color intensity of this cell equals to $5/9$.

In Figure 3, $\mathrm{cell}(2, 3)$ is darker than $\mathrm{cell}(2, 4)$, because the first contains three classes while the latter contains only one; $\mathrm{cell}(4, 3)$ is white (*i.e.*, the color's intensity is zero) because it is empty. $\mathrm{cell}(3, 2)$ is darker (it is black) than $\mathrm{cell}(2, 3)$ while both contain the same set of classes, but the darkness of the first is relative to $\mathrm{cell}(2, 2)$ while the darkness of the second is relative to $\mathrm{cell}(3, 3)$. This darkness relativity allows us to know: for P_1, all the classes referenced by P_4 are referenced also by P_3 but some classes referenced by P_3 are not referenced by P_4.

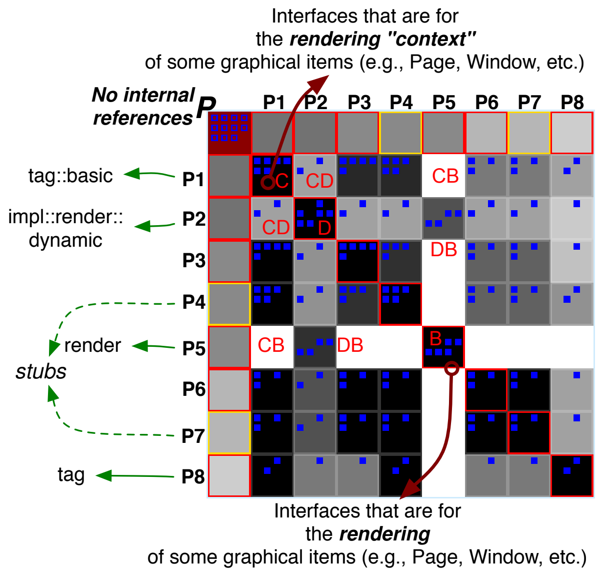


Figure 4. The Incoming Fingerprint of *render.renderer* package of the *them* subsystem.

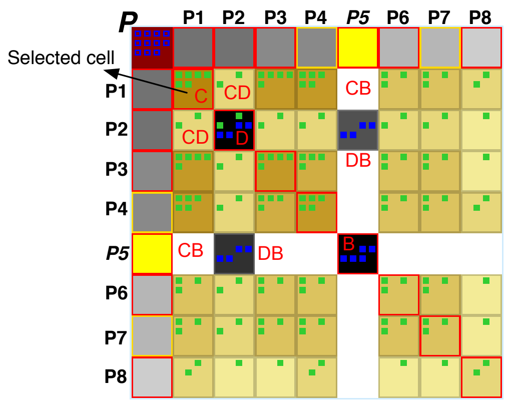


Figure 5. The Fingerprint in Figure 4: P_5 is marked in *Yellow* and the cell C is selected (*gold*). Thus, all the classes of C are selected (*green*). In consequence, each cell that represents *only* a subset of those classes is also selected.

4 Decorticating a Fingerprint

In the following we present an example that illustrates how a fingerprint is used for analyzing package references. Figure 4 shows the incoming fingerprint of the JBoss *render.renderer* package, visualized in the context of the subsystem named *them*. *Jboss* is composed of 499 packages; *them* is composed of 119 classes distributed over 15 packages.

No Internal Reference. In Figure 4, none of the small squares on the top left corner cell (P) are filled, which means that there is no internal references within the considered package. This package only contains interfaces.

Big Number of External incoming References. Many classes in the package have external incoming references. The fill color of P is dark red, thus we conclude that most of the classes of *renderer* package have external incoming references. By looking at the number of squares in P we can estimate the size of the In-Interface (here it is equal to 11).

Small Number of Referencing Packages. The fingerprint has a relatively small size: 8 packages are referencing classes of the package under analysis. Only two among them are stubs (P_4 and P_7, they have a gold border color). By moving the cursor over these packages, a fly-by-help shows their names: *test::them* and *test::them::renderer*. Thus we guess that they should include test classes. Also we understand that P is used only by a few packages within the subsystem *them* and it does not have a direct role outside this subsystem. Thus the it is a peripheral package.

Dominant Package. As P_1 is the topmost package, we know that it is the most referencing package to P. Also we see that the most cells of the column of P_1 are black, which means that the most referencing packages (P_3, P_4, P_6, P_7, and P_8) refer to subsets of the classes that are referenced by P_1: P_1 is a dominant referencer of P.

Also to help the user to detect quickly the information we introduce an interaction mechanism to the visualization. Figure 5 shows that the selecting C automatically selects of all cells in the body, except B, D and their intersection DB. Thus we know that the classes that are referenced by P_1 represent all the referenced classes in P, except those that are referenced by P_5 and P_2. The color intensity of a selected cell is relative to how many selected classes it contains. We see then how the classes referenced by P_1 spread on the fingerprint. To quickly detect the occurrences of given packages in other fingerprints, the user can mark arbitrary packages by different colors.

Commonly Referenced Classes. The black cells in the lines of P_6 and P_7 highlight commonly referenced classes. Here we see that three classes are often accessed together. The fact that the classes keep their positions in every cell help spotting such patterns.

Classes with different responsibilities. Glancing at the fingerprint's body we see that it looks filled up: only one cell of the main diagonal (B) breaks the fill and causes a white line/column within it. A white cell shows that there is no shared reference to P between the packages. Here there is no shared reference between P_1 and P_5, P_3 and P_5, ...

The cell (B) contains five classes: the classes referenced by the package (P_5). Cell DB represents the non empty intersection of cell D with cell B, *i.e.*, the four classes referenced from both P_5 (cell B) and P_2 (cell D).

Going further into the details, we see that the referencing package (P_1) is the one that depends most on the package under analysis, because it is the closest to the top left corner. This package refers to six classes: they are the classes represented by the main diagonal cell (C). Also all the cells in the column of P_1, *i.e.*, the second column in the fingerprint, are black except two cells (CD and CB). Thus we know that all the referencing packages refer to classes that belong to the set represented by C, except P_2 that refers to only two classes of this set (CD) and P_5 does not refer to any class of this set.

Thus we assume that the analyzed package contains two collections of classes: the first one with 6 classes (C) referenced by all the referencing packages except P_5; this last with 5 classes (B). P_2 refers to classes of both collections, but it refers to 4 classes among B classes (DB) and just two among C classes (CD). Based on that, we can already suspect that it is possible to re-modularize the package under consideration.

Inspecting B classes we learned that they represent the interfaces of item renderings (*e.g.*, *PageRenderer, WindowRenderer, etc.*) while C classes are the interfaces of item rendering contexts (*e.g.*, *PageRendererContext, WindowRendererContext, etc.*). C interfaces are referenced by the package *render*, within its class *renderContext* that implements the facade pattern. This latter is responsible of the communication with different objects whose types are declared via the interfaces (*e.g.*, *PageRenderer, WindowRenderer, etc.*). The package *impl::render::dynamic* contains classes that implement some of the interfaces of B.

Reading the code, reinforced the difference in the usage of both interface collections reinforced our estimation of a necessary re-modularization.

5 Reading the Fingerprint From Far Away

We introduce two levels of zoom-outs so that the visualization remains compact and scalable over a number of referencing packages or the size of the interface, and supports global visual patterns as listed in Section 7, while minimizing information loss compared to the details presented in Section 4:

1. We do not visualize the cell internals. We visualize, only in the main diagonal, the size of each cell, *i.e.*, the number of referenced classes that are represented by it.

2. We can also visualize the fingerprint without the cell internals and the size of main diagonal's cells.

Figure 6 shows the fingerprint of the *renderer* package, illustrated in Figure 4, zoomed-out twice. In the first zoom-out

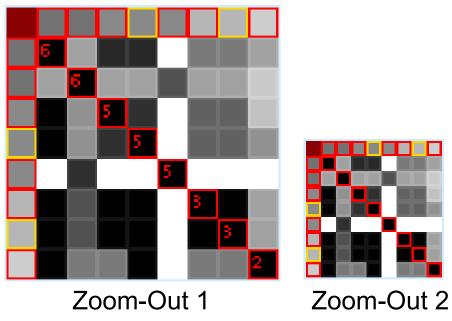


Figure 6. The Incoming Fingerprint of *renderer* package (Figure 4) zoomed-out twice.

we do not see the information about the classes represented by cells, but we can estimate the size of any cell using its darkness and the size of the main diagonal's cell which is located on its line. This last information is hidden in the second zoom-out. To help the user find it we have introduced, in addition to the selection and marking mechanisms represented in Section 4, a new interaction with the fingerprint: by moving the cursor over any cell a fly-by-help shows us the size of the cell and the set of the classes it represents.

Reading the Fingerprint We believe that the package fingerprint, as described in Section 4, helps developers to understand and to analyze a given package, while the fingerprint zoom-outs helps visualizing a big number of packages, easily navigate in the system and detect global information (*e.g.*, patterns, anomalies, etc.). To understand and analyze any package in detail, the developer can select it and zoom to its full fingerprint at any time.

Figure 7 shows the incoming fingerprint of the package *utils* of the subsystem *plugins*, taken from *Azureus* application. In the following we illustrate how to read this incoming fingerprint, and which relevant information we can get out.

At first glance, the size (*i.e.*, width or height) of the fingerprint is relatively large and all referencing packages are golden bordered. That means the *utils* package is referenced by a big number of packages that all are located outside the subsystem *plugins*.

The top left cell (P) is dark red, which means that most of the package's classes are referenced from the outside, *i.e.*, the size of its In-Interface is big.

The fingerprint's fill shows that some cells on the main diagonal (circled in green) are isolated within their row: *i.e.*, the row are nearly completely white. These cells identify services provided by the analyzed package for only a couple of packages. Classes represented by those cells are considered as lightly coupled in the context of the package, and their presence degrades the package cohesion.

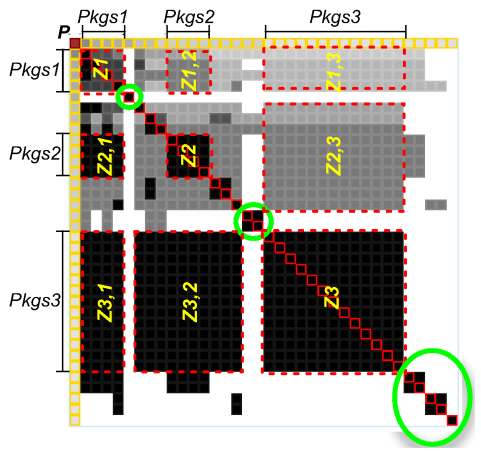


Figure 7. The Incoming Fingerprint of *utils* **package, from** *Azureus.plugins* **subsystem.**

Furthermore, the fingerprint fill shows a black filled rectangle Z_3 at the intersection of the rows and columns of the packages $Pkgs_3$. This indicates that the cells within Z_3 represent the same collection of classes that are referenced together by $Pkgs_3$. In the same way, we can deduce that those classes are also referenced together by the packages $Pkgs_2$ and $Pkgs_1$: see the black filled rectangles $Z_{3,2}$ and $Z_{3,1}$. These set of classes are referenced together from most of the referencing packages: they are highly coupled within the package under analysis. Furthermore, the presence of dark/black rectangles within the fingerprint's body is indicator on the package cohesion: the more black space, the more cohesive the package is.

Comparing black filled rectangles according to their size also provides us with useful information: the larger a rectangle size is, the higher the coupling between the classes represented by it. For example, classes in the rectangle Z_2 are less coupled than the classes of Z_3.

The fingerprint's body is not symmetric in terms of darkness. While the classes that are referenced by the packages $Pkgs_1$ and $Pkgs_3$ are represented by both the rectangles $Z_{1,3}$ and $Z_{3,1}$, the fills of those rectangles have different darkness: $Z_{1,3}$ is light grey and $Z_{3,1}$ is black. We deduce then that the classes referenced by $Pkgs_3$ form a small portion of the classes referenced by $Pkgs_1$. Thus we say that the dissymmetrical darkness of the fingerprint's body is relative to the size of the In-Interface of the package under-analysis. It also prefigures that class usage is not homogeneous from the package perspective: some referencing packages refer to a big number of classes and other packages refer only to a few classes.

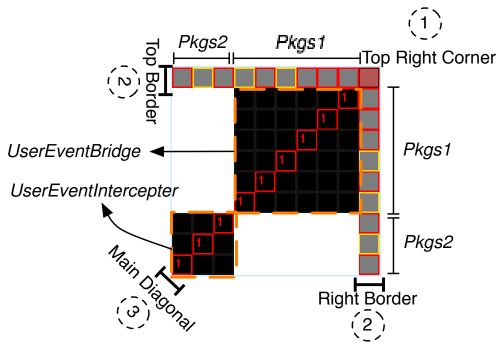


Figure 8. The Outgoing Fingerprint of *impl::api::user* **package, from** *Jboss.core***.**

6 Outgoing Fingerprint

Until here we limited our presentation to incoming references; we also propose the symmetrical view to help understanding how the package under analysis uses the rest of the application. The principles we described above are used exactly in the same way, except that we take into account outgoing references instead of incoming ones: the referenced packages and the Out-Interface of the package under analysis. To help the software engineer to always know which fingerprint he is reading, in a outgoing fingerprint, the package under analysis is located on the top right most corner, *i.e.*, the *top right corner*, and the diagonal is crossing in the other direction. Also the referenced packages form the *right border* of the package outgoing fingerprint.

Figure 8 shows the outgoing fingerprint of *impl::api::user* package. The fingerprint shows two important pieces of information: (1) there are two *distinct* groups of packages ($Pkgs_1$ and $Pkgs_2$ on the figure) been referenced by the classes of the analyzed package – since the cells are black, all the referenced packages are accessed consistently, and (2) there are two classes in the package's Out-Interface since each group has a most one referencing class. The view also reveals the input source for each class, we can then coarsely evaluate the potential impact of changes on the package.

7 Relevant Visual Patterns

While applying Package Reference Fingerprints to large applications (Azureus, Jboss, ArgoUML) we identified some visual patterns. We present here the most frequent ones, knowing that several patterns could occur within a single fingerprint. For space reason, we describe each pattern only in the context of Incoming Fingerprints.

Black Fill. This pattern occurs when all the package In-Interface classes are conceptually coupled: all the In-Interface classes are referenced together by every referenc-

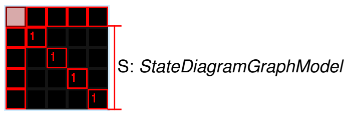


Figure 9. The Incoming Fingerprint of *uml::diagram::state* **package, from** *ArgoUml***.**

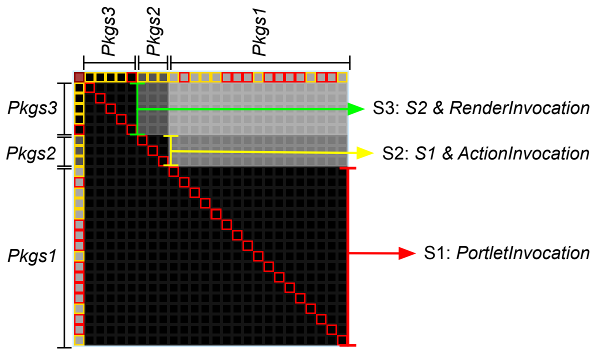


Figure 10. The Incoming Fingerprint of *invocation* **package, from** *Jboss.portlet* **subsystem.**

ing package. In our case studies, this pattern occurs when the size of the In-Interface of the package under analysis is very small, particularly when it exports only one class, or when the package is referenced by a small number of packages. Peripheral packages often present this pattern. In this pattern, all the classes of the package's In-Interface are referenced always together as a single service. Thus such a package is often characterized by a high degree of cohesion because all its classes are related to fulfill a single service.

Variation. The package *metadata*, shown in Figure 10, illustrates a variation of this pattern: the fingerprint fill appears as gray layers: under the main diagonal the cells are black and above it, they are in progressively lighter shades of gray. The classes in the last lines represent the most important classes because they are referenced by all the referencing packages: in Figure 10, the class *PortletInvocation* (S_1) is the most referenced one (S_1 is referenced by $Pkgs_1$, $Pkgs_2$ and $Pkgs_3$). S_2 is referenced by $Pkgs_2$ and $Pkgs_3$. S_3 is only referenced by $Pkgs_3$. This pyramid shows that there is a layering in the terms of the use of the services. Figure 9 shows that *state* package provides only one service (S).

Broken. In this pattern the fingerprint's fill contains white cells. Some cells are isolated within their line and column (*i.e.*, no other dark cell within their line). The packages *utils* (Figure 7), and *render::renderer* (Figure 4) exhibit this pattern. In some cases these cells form together a small square

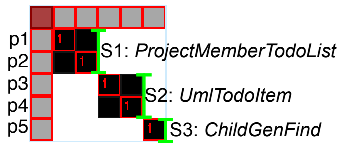


Figure 11. The Incoming fingerprint of the *uml::cognitive* **package in** *ArgoUml* **application.**

around the principal diagonal. Such isolated cells represent particular classes in the context of the package from a client perspective: they offer specific services that are accessed by few clients. This pattern represents a shortcoming in the package's design, and the question of re-factoring the breaking classes, for improving the package cohesion or/and reducing its coupling, takes an important place.

Arrow. The only non white cells in the fingerprint's fill are the diagonal's cells, making the fingerprint look like an arrow. In this case, the package provides non-coupled services to the system.

Variation. The strict occurrence of this pattern appears when there is one client per service and that the services are not mixed. A frequent variation is when the arrows's body is composed of small squares. These squares represent the clients of a given services provided by the package. Again the presence of squares only on the diagonal, is a good indication that the functionality of the packages is not cohesive from the client point of view. Note that the width of the square still conveys an estimate of the cohesion based on client usage.

Figure 11 shows an example of a package that provides three services (S_1, S_2 and S_3), in it S_1 and S_2 are more important than S_3: S_1 and S_2 are both used by two packages but S_3 is used by only one package. These patterns show that we could easily re-modularize the package to improve its contextual cohesion or/and reduce its coupling.

Unbalanced. This pattern occurs when an incoming or outgoing fingerprint appears clearly bigger than its counterpart. The Unbalanced-Incoming Fingerprint pattern indicates that the analyzed package plays a server role within the system, rather than a client role. The Unbalanced-Outgoing Fingerprint pattern indicates the reverse case.

Two variants of this general patterns have special interest:

- *Giant Incoming Fingerprint.* This variant reveals core/central and utility packages that provide basic services for the system. Figure 7 shows that *plugins.utils* applies this pattern.
- *Empty-Outgoing Fingerprint.* The outgoing fingerprint is empty, *i.e.*, the package under analysis does not refer

to any package in the system. This occurs for packages that include only abstract classes or/and interfaces. Such packages are not impacted by the system and could be reused in other applications.

Golden Border-Side. This patterns occurs when all the referencing packages are stubs (*i.e.*, are not part of the system under analysis). Thus, this pattern only occurs when the clients of the package under consideration do not belong to the analyzed subsystem (*e.g.*, Plugins within Azureus). Such packages represent the output of the analyzed subsystem. This pattern is a good sign because it indicates that the application under analysis tends to be well layered.

8 Discussion and Evaluation

8.1 Discussion

Fingerprints are not magic; they show, albeit in a condensed form, the existing situation of the code. When packages are not well-designed the patterns are less apparent, still the visualization conveys the situation and the information about the use of the package by its clients. Our approach has worked well on our case studies and we have been able to locate many conceptual bugs and to spot several visual patterns. It should be noted that we were *not* familiar with the case studies before applying our approach.

Position Choices. A reader often pays more attention to the top elements than to the bottom ones. Therefore, we grouped the internal references at the top corner of the package fingerprint, then ordered the related packages from the most related one at the top to the least at the bottom.

Seriation. We ordered referencing packages that make the same number of references by similarity; this way, the reader can see which packages access the same groups of classes. We tried ordering packages differently, *e.g.*, by similarity regardless of how many references they make, but each time we lost important information *i.e.*, the position of the most/least referencing packages.

Impact of Boundaries. We colored the border of packages that do not belong to the application under-analysis in gold. We found it really effective to color entities so that the user can interactively mark entities on which he wants to focus; this increases the usability of the tool.

Zooming. We introduced two levels of zoom-outs with minimal information loss, so that the visualization remains compact and scalable over the number of the related packages or the size of the interface. This way, the user can visualize large applications, navigate in the system, spot global patterns and conceptual anomalies. Then he can focus on any package by zooming in to the detailed fingerprint.

However, during our experiments, we found that detailed fingerprints have do not scale as well as the zoomed-out views. Detailed fingerprints expose a lot of information, which makes it difficult to spot patterns or gather general information about the visualized package; this is especially true for giant packages whose interface and number of related packages are very large. In fact, in such cases, no detailled view that we applied has scaled well. We think that zooming mechanisms are very important in software visualization to solve this problem.

Placeholders. The placeholders in cell internals are essential to make preattentive processing work and thus to help users see quickly which classes are coupled and where they are coupled. The negative impact of this principle is that all cells should be large enough to contain all possible classes in the package's interface. This is one of the reason why the detailed fingerprints do not scale so well.

Fingerprints Limitations. Package fingerprints focus on the package's contextual cohesion, afferent and efferent coupling, and co-use of internal classes. However, they do not provide a good map for internal dependencies; our aim is to support understanding packages through their interfaces, regardless what happens inside them. Similarly, we consider related packages (*e.g.*, referencing packages in an incoming fingerprint) as black boxes; we only care about a package's classes while we look at its fingerprint.

8.2 Related Work

Sangal *et al.* adapted the dependency structure matrix from the domain of process management to analyze architectural dependencies in software [17]; they presented a consistent visualization by focusing on all the system at a time, thus the visualization does not scale well over large systems; they used only numbers to visualize the quantified information. Package fingerprints are based on similar principles but provide more visual information and help identify groups of packages with similar dependencies. A fingerprint exploits pre-attentive processing using color, contrast, and the principle of placeholders. In addition, a fingerprint by focusing on a package at a time qualifies in a finer-grained way the dependencies.

A Package Blueprint [7] presents a condensed map of a package. It shows dependencies on a per-class basis, but it does not help compare and group the client/provider packages in terms of their dependencies to the package under analysis.

Several works explore packages and their structure but few of them reveal information on their relationships and dependencies. In Softwarenaut, Lungu *et al.* help system discovery by guiding exploration of nested packages based nesting and dependencies [12]. Storey *et al.* also worked on system exploration, but their views do not scale very well with the number of relationships [18]. Kuhn *et al.* used information retrieval to exploit linguistic information, he introduced semantic clustering to group source artifacts that use similar vocabulary. He based on linguistic topics to reveal the intention of the code and the similarity between its artifacts, then he provide a consistent visualization [10].

A number of approaches give summarized information on package relationships and their evolution: the Butterfly by Ducasse *et al.* gives a high-level client/provider trend of package dependencies [6]; Pzinger *et al.* show the evolution of package metrics using Kiviat diagrams [14]; Chuah and Eick use rich glyphs to characterize software artifacts and their evolution (number of bugs, number of deleted lines, kind of language...) [4]. In particular, the timewheel exploits preattentive processing, and the infobug presents many different data sources in a compact way; finally, D'Ambros *et al.* reveal package coupling by showing evolutions that are correlated in time [5].

Those approaches, while valuable, do not provide a fine-grained view of packages that helps understanding the contextual coupling and cohesion inside packages, and that reveals possible remodularisations.

9 Conclusion

In this paper, we tackled the problem of understanding the details of package relationships. We described the package fingerprints, and their use as a visual approach for understanding package relationships, contextual cohesion, and the conceptual coupling of their classes. While designing Package Reference Fingerprint, we exploited pre-attentive processing using color properties and saving placeholders principle. We also introduced interactivity and multi-selection mechanism to help the user during the analysis task.

We successfully applied the visualization to several large applications and we have been able quickly to point out badly designed packages, and to extract relevant patterns.

While applying Package Reference Fingerprints to large applications that contain radically different packages in terms of internal size and package references, the visualization scaled well and the detection of different presented patterns was always possible.

We plan to apply our approach for creating other views (*e.g.*, the inheritance relationships) and to validate the visualization usability with independent software maintainers.

References

[1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: Are we being over-persuaded? In *Conference on Software Maintenance and Reengineering (CSMR), Lisbon*, 2001.

[2] 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.

[3] 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.

[4] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[5] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pp. 189 – 198, 2006.

[6] S. Ducasse, M. Lanza, and L. Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pp. 70–77. IEEE Computer Society, 2005.

[7] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, 2007.

[8] M. Fowler. Reducing coupling. *IEEE Software*, 2001.

[9] C. G. Healey, K. S. Booth, and E. J. T. Harnessing preattentive processes for multivariate data visualization. In *GI '93: Proceedings of Graphics Interface*, 1993.

[10] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243, Mar. 2007.

[11] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[12] M. Lungu, M. Lanza, and T. Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pp. 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[13] H. Melton and E. Tempero. The crss metric for package design quality. In *ACSC '07: Proceedings of the Australian Computer Science Conference*, 2007.

[14] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pp. 67–75, St. Louis, Missouri, USA, May 2005.

[15] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pp. 91–103, 2006.

[16] L. Rising and F. W. Calliss. Problems with determining package cohesion and coupling. *Software - Practice and Experience*, 22(7):553–571, 1992.

[17] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pp. 167–176, 2005.

[18] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On integrating visualization techniques for effective software exploration. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis '97)*, pp. 38–48. IEEE Computer Society, 1997.

[19] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 2nd edition, 2001.

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