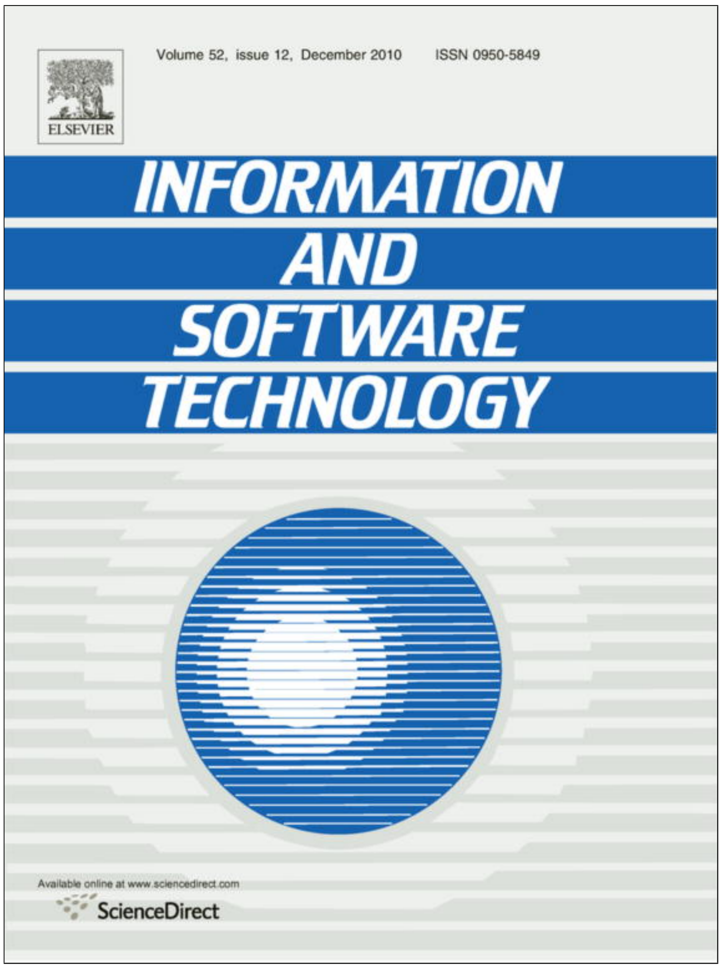

Volume 52, issue 12, December 2010
ISSN 0950-5849
ELSEVIER
INFORMATION AND SOFTWARE TECHNOLOGY
Available online at www.sciencedirect.com
ScienceDirect

Information and Software Technology 52 (2010) 1312–1330

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Package Fingerprints: A visual summary of package interface usage

Hani Abdeen [a], Stéphane Ducasse [a,*], Damien Pollet [a], Ilham Alloui [b]

[a] RMoD, INRIA Lille Nord Europe/USTL LIFL/CNRS UMR 8022 – 40 Avenue Halley, 59650 Villeneuve d'Ascq, France
[b] Listic, Polytech'Savoie – 5, chemin de Bellevue, Domaine universitaire d'Annecy-Le-Vieux, France

A R T I C L E I N F O

Article history:
Received 5 July 2009
Received in revised form 20 July 2010
Accepted 26 July 2010
Available online 3 August 2010

Keywords:
Software package
Visualization
Cohesion
Coupling

A B S T R A C T

Context: Object-oriented languages such as Java, Smalltalk, and C++ structure their programs using packages. Maintainers of large systems need to understand how packages relate to each other, but this task is complex because packages often have multiple clients and play different roles (class container, code ownership, etc.). Several approaches have been proposed, among which the use of cohesion and coupling metrics. Such metrics help identify candidate packages for restructuring; however, they do not help maintainers actually understand the structure and interrelationships between packages.
Objectives: In this paper, we use pre-attentive processing as the basis for package visualization and see to what extent it could be used in package understanding.
Method: 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 views 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.
Results: We applied these views on four large systems: Squeak, JBoss, Azureus, and ArgoUML. We obtained several interesting results, among which, the identification of a set of recurring visual patterns that help maintainers: (a) more easily identify the role of and the way a package is used within the system (e.g., the package under analysis provides a set of layered services), and (b) detect either problematic situations (e.g., a single package that groups together a large number of basic services) or opportunities for better package restructuring (e.g., removing cyclic dependencies among packages). The visualization generally scaled well and the detection of different patterns was always possible.
Conclusion: The proposed visualizations and patterns proved to be useful in understanding and maintaining the different systems we addressed. To generalize to other contexts and systems, a real user study is required.

1. Introduction

To cope with the complexity of large object-oriented software, developers organize classes into packages or modules. This organization usually follows conceptual interrelationships between classes, that the stake-holders would like to maintain over ineluctable system evolution. As the system modular structure changes, its maintenance is required. However, where approaches of system remodularization succeed in proposing system refactorings [49,6,33,32,5], they do not provide good ways for understanding and assessing the changes they propose. There is a wide range of work to define new modularization algorithms but little support to understand the proposed results and their impact on existing systems.

It is important to understand the concrete organization of packages and their interrelationships. Ideally, packages should be kept as less coupled and as much cohesive as possible [9,3]. We distinguish two main approaches of package cohesion in the existing literature [39,34,5,38]. The first approach defines the cohesion of a package in terms of the interconnections between its internal classes. The second approach defines cohesion according to how the system uses the package classes. For instance, if two classes of a package are used from the same client package, then they are considered as conceptually related, regardless of the explicit relationships that exist between them [38]. 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 [9,3,34,5,38] and help maintainers determine packages that are candidates for restructuring. However, those approaches do not help maintainers of large systems 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/client roles.

* Corresponding author.
E-mail addresses: hani.abdeen@inria.fr (H. Abdeen), stephane.ducasse@inria.fr (S. Ducasse), damien.pollet@inria.fr (D. Pollet), ilham.alloui@univ-savoie.fr (I. Alloui).

0950-5849/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2010.07.005

Several previous works on software visualization provide information on packages and their relationships, by visualizing software artifacts or metrics about their structure or evolution [17,15,16,27,37,41,44,22]. While these approaches are valuable, they fall short of providing a fine-grained view of packages that would help maintainers understand the structure of packages, their interrelationships within the system, and identify their roles within a system.

In this article, we present the Package Fingerprint, a compact, rich and zoomable visualization to better support the understanding of a package and its relationships. The goal of this visualization is to help maintainers during their early contacts with unknown packages. We propose two complementary variants of the Package 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 classes, and highlights the cohesion of the analyzed package, as defined in [38]; the *outgoing fingerprint* shows how the package classes use the system.

This article is an extension of previous work [1]. The new contributions presented here are: (1) the description of the outgoing fingerprints and their use, (2) the application of fingerprints to large industrial systems, and (3) additional fingerprint patterns.

In Section 2, we discuss the challenges for understanding packages. Then we present the principles of the incoming and the outgoing fingerprints in Section 3. In Section 4 we show how to use the incoming fingerprint for analyzing and understanding packages in practice. Section 5 presents the different zoom levels of a fingerprint and shows how to read a fingerprint from far away. Section 6 presents the outgoing fingerprint via a simple example, and Section 7 lists the relevant visual patterns in fingerprints. Finally, we discuss our approach and related works in Sections 8 and 9.

2. Challenges in understanding packages

Parnas introduced *information-hiding* as a criterion of the decomposition of systems into modules [35]. The idea is to improve the quality of software, e.g., adaptability and changeability, by decoupling design elements that are likely to change so that they can be changed independently. This idea has been largely adopted in object-oriented design and in software architecture [40]. Object-oriented languages, such as Java and C++, provide the notion of packages, or namespaces, to support the decomposition of systems into subsystems [29,30].

Packages, however, are not mere class containers: they are complex entities that have different usage patterns, often depending on the clients that use them. Packages often represent code ownership, feature containment, team organization or deployment entities. Packages play different roles, some central to the system, others peripheral: Some packages act as reference hubs, others as authorities.

These multiple facets of packages do not ease the understanding of inter-package relationships nor even quick identification of a package clients or providers [17]. Although languages such as Java make dependencies between packages explicit (i.e., via the import statement), developers lack tool support to really understand packages within their context.

To understand the structure and the roles of packages within a system, we need both raw size information (the size of elements and their relationships), and coupling and cohesion related information. In this section, we summarize the information that a solution supporting package understanding should provide.

2.1. Raw size information

To understand the packages of a system and their relevance in the general picture, gathering quantitative information is a good way to offer a mental picture to the [36,25]. Here is a list of relevant questions:

- How many classes are packaged within a given package?
- How many classes are visible to the rest of the system or communicate with it?
- How many packages depend upon a given package?
- How many packages does a given package depend upon?
- What is the ratio of *internal*/*external* class references and inheritance definitions?

2.2. Cohesion and coupling

Robert Martin discussed principles of architecture and package design, addressing package cohesion and package coupling [30]. The package cohesion principles are:

Release Reuse Equivalency (REP): Since packages are the unit of release, they are also the unit of reuse. Therefore a good package should only contain a group of classes that are reusable together.
Common Closure (CCP): To minimize the number of packages that are changed in any given release cycle, it is better to group classes that change together into the same package.
Common Reuse (CRP): Since a dependency upon a package is a dependency upon everything within the package, classes that are reused together should be grouped together. This way, in any given release, changing any class within a considered package will have the same impact-propagation if maintainers change another class within the same package. Thus the impact-propagation of the package changes is always constrained to one graph.

Coupling is always used with cohesion to determine package quality and it is generally defined as: *if changing one package in a program requires changing another package, then coupling between these two packages exists* [7,21]. Robert Martin defines two types of coupling: *Efferent Coupling* and *Afferent Coupling* [30] taking into account the sense of the reference (namely incoming and outgoing).

Cohesion and coupling metrics are among the most used metrics during perfective maintenance, because they help identify which packages should be restructured [34,5,39,8,3,28]. In general, good packages should group classes that are needed for the same task [38], and they should have a few clear dependencies to other packages: they should be highly cohesive and lightly coupled. However, cohesion and coupling alone do not help maintainers understand the structure, roles, or relationships of packages. In particular, they do *not* indicate whether, why and how a package respects Martin's cohesion principles, nor do they help decide what to do if such principles are not respected.

For this, maintainers need more detailed information. For example, it is important to know if some classes in a package are *always used together* or not, and conversely the proportion of package classes that uses the same set of classes/packages. Knowing about the usage relations between a package and its clients and providers offers another perspective on package cohesion, since that gives the maintainer information on the package role and cohesion according to Common Reuse Principle [38].

2.3. Package maintenance scenarios

There are a couple of package centric maintenance scenarios that a good visualization should support. We give here a first open list based on our experience as maintainers of some large software systems such as Squeak [13]. These scenarios set the context we use to detail how we applied fingerprints to real software systems

(Sections 4–7). In particular, we show that fingerprints offer fine information on package use. We limit ourselves to package maintenance actions since this is the focus of fingerprints.

Basic understanding. When working on a package, we want to get an idea of the importance of the package in terms of its size, but also in terms of its actual place and its role within the system. A package that is mainly using other packages will certainly be easier to change than a central package used by many others.

Merging related packages. When maintaining a group of packages, it is interesting to know the coupling of each package, as well as the scope of the coupling and the classes that contribute to it. This helps decide whether the package can be split, or on the contrary whether it should be merged with others. In our experience, we saw that a group of highly coupled packages is also an indication that the group as a whole should probably be repackaged, and that a fine grained decomposition may be counter productive or artificial. To determine whether it is worth merging related packages, we have to look at how some specific classes of these packages are used. In this scenario, we gain valuable information by identifying which classes of a package are used externally, and by grouping these classes according to their users.

Splitting packages. Correlated to the previous point, splitting a package is a useful task since it supports lower memory footprint and easier replacement of functionality. Often, before splitting a package, it can be enough to simply move misplaced classes. We identify such classes by examining the users of the class as well as the internal package cohesion. Class co-use also indicates that the classes may form a coherent package. Nevertheless, co-usage is just a partial view, and it should be complemented with the identification of helper classes and with inheritance constraints (superclasses or subclasses induce diverse situations and may be packaged differently).

3. Package Fingerprint principles

Our aim is to provide an approach that helps maintainers understand packages in their context, regardless of what happens inside packages – since this is considered as a hidden-information from the point of view of its system [38]. We will focus on a package as a provider and/or client offering and/or requiring functionalities to/from other packages within a system.

To meet some of the requirements mentioned in Section 2, we propose two complementary views for *incoming* and *outgoing* references through the fingerprints. The objective of Package Fingerprints is to provide an overview of package cohesion and coupling by stressing the client/provider relationships of the classes contained in the considered package. As such it is complementary to traditional coupling/cohesion metrics [7,3]. Before going in detail, we setup the vocabulary and the intention of fingerprints.

3.1. Terminology

Fig. 1 illustrates the terminology we use in the rest of the paper. First, by *reference*, we mean that a class A refers to a class B that A statically uses the name of, or invokes methods of B. It is worth to note that we extract inter-class references using static analysis the concerned software system. This way, method invocations are linked to the declared type (i.e., class) of the objects whose methods are invoked in run-time. As a consequence, our approach full short of support method late-binding in the presence of polymorphic invocations. In addition, in the case of dynamic typed languages (e.g., Smalltalk) we often can not statically determine the declared type or the class of the invocation target object in the

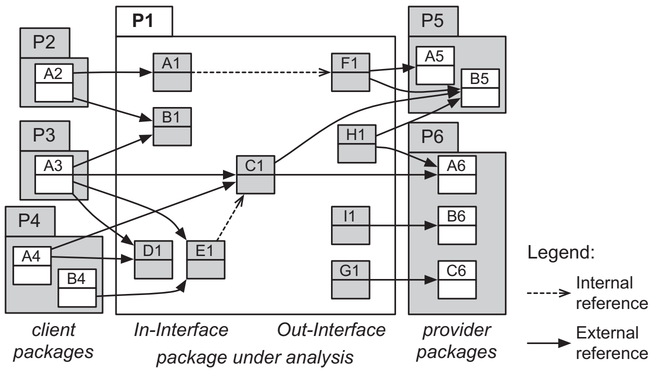


Fig. 1. Terminology—an example of references between packages: P_1 contains nine classes, it has three clients (P_2, P_3 and P_4) and two providers (P_5 and P_6). Both In-Interface and Out-Interface of P_1 contain five classes, with C1 in common.

run-time. In such a case, our strategy consists in creating a reference for every potential candidate class (i.e., every class within it there is a method that has the invoked method signature). These limitations can be addressed with additional dynamic analysis of method invocations.

We mean by internal references the references which are among classes packaged in the same package. Otherwise, references are external. In this context we mean that a package P_i refers to another P_j if P_i contains a class C_i that refers to another class C_j packaged in P_j. In the same vein, we say that P_i refers to C_j, C_i refers to P_j, and P_j is referenced by C_i and by P_i. As a shortcut, when we say that a package P refers to another package Q, we mean that classes contained in P refer to classes of Q. In this context we say that P and Q are coupled [30].

Definition 1 (*In-Interface*). The In-Interface of a package P is the set of classes of P which are referenced by classes packaged outside P.

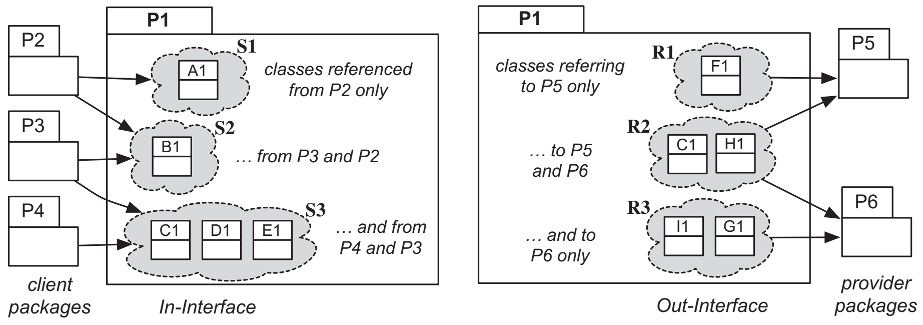
Definition 2 (*Out-Interface*). The Out-Interface of a package P is the set of classes of P that refer to classes packaged outside P.

As shown in Figs. 1 and 2, the size (i.e., number of classes) of the In-Interface gives maintainers a quantified information about the dependency of the system on the package under analysis P_1, while the number of referencing packages shows the importance of P_1 for the system. Similarly, the size of the Out-Interface of P_1 gives maintainers a quantified information about the dependency of P_1 on other packages, while the number of referenced packages shows how much P_1 depends on the system.

Since referencing a class is an indicator of the usage of that class functionalities, referencing a group of classes in a consistent way is an indicator of the usage consistency of those classes. Such a referenced group, that we name a service, represents classes whose functionalities are consistently used together.

Definition 3. ServiceIn the context of a package P, we mean by Service, the set of classes of P In-Interface which are referenced together by the same group of packages. This is related to the Release Reuse Equivalency (REP) principle (see Section 2.2) where good package should only contain a group of classes that are reusable together.

Martin [31] defines a class responsibility as *a reason for change*. From the view point of inter-class references, if a class A refers to another one B, changes in B may be a reason for changes in A. At a high-level of abstraction, if A refers to a package P, changes in P may be a reason for changes in A. In this context, we define a package *reason-for-changing* as follows:



(a) Grouping the classes of the In-Interface of P_1 by common client packages.

(b) Grouping the classes of the Out-Interface of P_1 by common provider packages.

Fig. 2. Grouping incoming and outgoing references into In-Interface and Out-Interface interfaces.

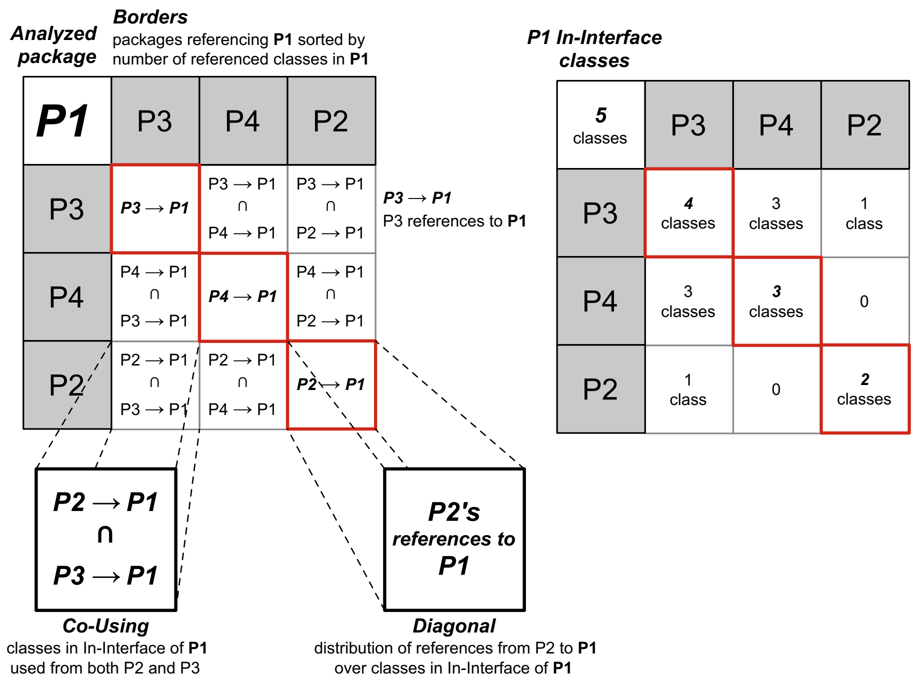


Fig. 3. Skeleton of the incoming fingerprint for P_1 (Fig. 2).

Definition 4 (*Reason for changing*). In the context of a package P, we mean by Reason for Changing, the set of classes of P Out-Interface which refer together to the same group of packages. This is related to the Common Closure (CCP) principle (see Section 2.2) where a good package should group together classes that change together.

3.2. Fingerprints intention

To understand the multiple facets of a package, we group its classes according to their usage by other packages and their usage of other packages. Fig. 2a shows the In-Interface classes of P_1 grouped into clusters as well as the references that point to those clusters, while Fig. 2b shows the Out-Interface classes of P_1 grouped into clusters as well as the references that go out from those clusters. Fig. 2a shows that P_1 provides three services (S_1, S_2 and S_3): the service S_3 is used by the client packages P_3 and P_4; additionally, P_3 with P_2 use the service S_2; the service S_1 is used by the client package P_2 only. Fig. 2b shows that P_1 involves three reasons for changing (R_1, R_2 and R_3): R_1 represents the class F1 which refers to P_5, R_2 represents the classes C1 and H1 which refer to P_5 and P_6, while R_3 represents the classes I1 and G1 which refer to P_6.

Clustering the In-Interface and Out-Interface helps identifying the inter-dependencies between the package under analysis and the system, and thus which classes are conceptually coupled and which classes are not. At a higher level of abstraction, this helps answering the following questions:

- What services does the package provide?
- Which packages use those services?
- Does the package include classes that are always used together or not?
- Does the package include classes that use the same services/packages or not?
- Which are the reasons for changing the package?
- How are those reasons for changing distributed over the package classes?

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. Since we use the same approach for both views, we only present the incoming fingerprint in details and briefly sketch the outgoing fingerprint further on.

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 focus on one package, and *semantically rich* since they present multiple types of information at a glance.

3.3. Fingerprint skeleton

Fig. 3 depicts the key visualization principles of an incoming fingerprint with P_1 from Fig. 1 as the package under analysis. We first present the basic layout before introducing additional features we give to convey more information on package relationships. The skeleton layout of a fingerprint 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. Internal references are explained and illustrated in Section 3.4.
Referencing packages. The cells at the *borders* of the fingerprint, i.e., the leftmost column and the topmost row, both represent the referencing packages placed in the same order horizontally and vertically (i.e., there is a symmetry). Packages are sorted according to the importance of their references: the more referenced classes a package refers to, the closer it is to the top left corner. Fig. 3 shows the three packages that refer to P_1 in Fig. 1: P_3, P_4, and P_2, referencing respectively four, three, and two classes inside P_1.

If two packages make the same number of references, we then group them using a similarity criterion. We define this latter in an incoming fingerprint, as the number of shared referenced classes among packages. For example, in Fig. 2, 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). Conversely, we define the similarity of referenced classes by the number of referencing packages they share. Fig. 2 shows that the similarity between C1 and D1 (two common referencing packages P_3 and P_4) is higher than the similarity between C1 and B1 (1 common referencing package P_3). 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 on the leftmost column and the topmost row, each represents 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 row i and column j, *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* presents the distribution of the In-Interface on the client packages. Fig. 4 shows that *cell*$(3,3)$ represents the classes (C1, D1, E1) referenced by P_4, i.e., *cell*$(3,1)$ and *cell*$(1,3)$.
- The other cells present the classes referenced in *common* by both packages represented by the row and column heads. Fig. 4 shows that *cell*$(2,4)$ contains the class B1, referenced by both P_3 and P_2.

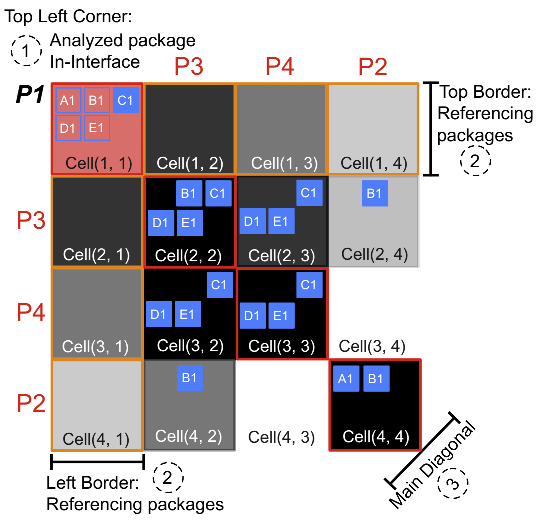


Fig. 4. Showing the incoming fingerprint of P_1 (Fig. 3) with the classes involved in the relations inside each cell.

We define the size of a cell as the number of classes it represents. Hence, in Fig. 4, *cell*$(2,2)$ has size 4 and *cell*$(3,3)$ has size 3: classes C1, D1, and E1 are represented in both cells, but class B1 in *cell*$(2,2)$ only.

3.4. Enriching the fingerprint skeleton layout

We enrich the skeleton of Fig. 3 to convey extra information such as the amount of referenced classes in the analyzed package. For this purpose we use color intensity for 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 [46,48]. Particularly, as our focus is on providing a first impression of a package and its context, we expect that pre-attentive processing will occur but we do not know to what extent[1] as much as possible to help spotting important information [23,48,45].

Cell internals. Inside a cell, we visualize the package referenced classes as small filled squares. To enable pre-attentive processing [23], we give each class a fixed place which is the same in all the cells of a fingerprint. When a cell represents a package reference to a class of the analyzed package, the location of this class is colored: in Fig. 4, since the class B1 is referenced by packages P_3 and P_2, the position corresponding to the class B1 is colored in the *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.
Information on internal references. Information on internal references among classes of the analyzed package is visualized on the *top left corner* (*cell*$(1,1)$). In Fig. 4 we see that among the five referenced classes of P_1, only C1 is referenced internally (as it is colored). Additionally, since not all classes will appear in all

[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.

cells, we use this corner cell to show all the placeholders for the classes that have incoming references, as bordered squares.

Colors. We use color hues to distinguish different entities in the fingerprint (e.g., classes, packages), and to give more information about the references. The colors we use are: (1) shades of gray for all the cells in a fingerprint except the top left corner, (2) blue for the classes (3) red for the top left corner and for highlighting the borders of the main diagonal cells (4) orange to highlight the fingerprint borders, (5) gold to highlight borders of the referencing packages that are outside the scope of the system under analysis (called stubs thereafter).

Color intensity. In addition to color hues, we use color *intensity* to give more information on the visualized entity: (1) for the top left corner, the darker the package, the bigger its In-Interface; (2) for the fingerprint borders, the darker a referencing package, the more classes it references in the analyzed package; (3) for the body, on *a given row*, the darker the cell, the more classes it represents. The darkness of a cell is calculated relatively to the size of the diagonal cell of that row. As consequence, the cells of the diagonal are black. On the fingerprint borders, we consider the color intensity for a referencing package as an additional visual information: as referencing packages are sorted according to the importance of referenced classes and similarity criteria (Section 3.3), we use a same color intensity for referencing packages with a same number of referenced classes. Indeed, those packages are placed in different order but have the same color intensity. Fig. 4 shows that P_3 is darker than P_4: the first package refers to 4 classes in P_1 while P_4 refers to three classes in P_1.

The color of the top left corner is based on an In-Interface size ratio: the size of the In-Interface of P_1 is 5 (Fig. 2a) while the size of P_1 itself is 9 (Fig. 1). Thus the color intensity of this cell equals 5/9.

In Fig. 4, *cell*(2,3) is darker than *cell*(2,4), because the first contains three classes while the latter contains 1 class; *cell*(4,3) is white (i.e., the color intensity is zero) because no referenced class inside. *cell*(3,2) is darker (it is black) than *cell*(2,3) although they both contain the same set of classes: the reason is that the darkness of the former is relative to the size of *cell*(3,3) while the darkness of the latter is relative to the size of *cell*(2,2). This darkness relativity informs us that: for P_1, all the classes referenced by P_4 are also referenced by P_3 but some classes referenced by P_3 (i.e., B1) are not referenced by P_4.

4. Detailling a fingerprint

In this section we present an example that illustrates how a fingerprint is used to analyze package references. Fig. 5 shows the incoming fingerprint of the JBoss *render::renderer* package (referred to as *P* here), visualized in the context of his subsystem, named *them*. As a whole, *Jboss* is composed of 544 packages; *theme* is composed 15 packages totaling up 119 classes.

No internal reference. As depicted by Fig. 5, none of the small squares on the top left corner cell (*P*) is filled: this means that there is no internal reference within the considered package. Actually, this package only contains Java interfaces.

Big number of external incoming references. The top left cell *P* is dark red, therefore most of the classes of *render::renderer* have incoming references from other packages. By looking at the number of squares in cell *P* we can estimate the size of its In-Interface (11 classes here).

Small number of referencing packages. The fingerprint has a relatively small number of rows and columns: only eight other packages reference classes of the package under analysis.

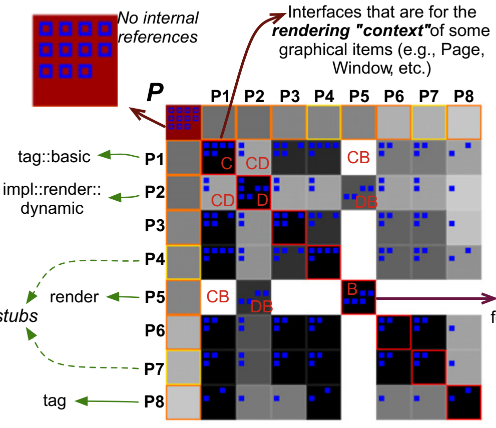

No internal references
Interfaces that are for the rendering "context" of some graphical items (e.g., Page, Window, etc.)
P
P1
P2
P3
P4
P5
P6
P7
P8
tag::basic
impl::render:: dynamic
render
stubs
tag
C
CD
CB
D
DB
B
Interfaces that are for the rendering of some graphical items (e.g., Page, Window, etc.)

Fig. 5. The incoming fingerprint of the package *render::renderer*, from the *them* subsystem of Jboss.

Two external packages, P_4 and P_7, have a gold border color, rather than orange. This means that they are stubs, i.e., they are not part of the system under analysis *theme*. Indeed, when moving the cursor over these cells a fly-by-help reveals their names *test::theme* and *test::theme::renderer*. Thus those two packages are part of the *test* subsystem rather than *theme*, and probably mainly contain test classes. Moreover, since *P* (*render::renderer*) is only used by 6 of the 15 packages of his subsystem and two external test packages, it does not have a direct role outside the subsystem *theme*. Thus we can qualify *render::renderer* as a peripheral package.

Commonly referenced classes. Since the small squares representing classes keep their positions in every cell, they make it possible to spot patterns. For instance, most cells in the rows of P_6 and P_7 show the same 3-square shape, highlighting commonly referenced classes.

Dominant package. As P_1 is the top/left-most package, we know that it makes the most references to *P*. We can also see that all cells in the column of P_1 are black; this means that the corresponding 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 thus a dominant referencer of *P*.

Classes with different reasons for changing. At a first glance, the fingerprint body looks quite filled up: only one cell of the main diagonal (*B*) breaks the fill and causes a white cross hair shape. A white cell means that there is no shared reference to *P* between the two packages for this cell, e.g., there is no shared reference between P_1 and P_5, nor between P_3 and P_5, etc.

The cell *B* contains five squares, for the five classes referenced by the package P_5. Cells denoted by *DB* represent 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*).

Examining cell *CD*, which represents the common referenced classes from both P_1 (cell *C* represents six referenced classes) and P_2 (cell *D* represents six referenced classes), reveals that P_1 and P_2 have only 2/6 referenced classes in common. For this reason cell *CD* is lighter than cells *C* and *D*. Similarly, cell *CB*, which represents the common referenced classes from both P_1 and P_5 (cell *B*), reveals that P_1 and P_5 do not have common referenced classes. For this reason cell *CB* is clearly lighter (i.e., white) than cells *C* and *B*.

Thus we learn that the analyzed package contains two disjointed subsets of classes: the first one with six classes (cell *C*) represents the subset which is referenced by all the client packages except P_5; the second one with five classes (cell *B*) represent the subset which is referenced only by P_5 and P_2. P_2 refers to classes of both subsets, but it refers to four classes from B (DB) and just two from C (CD). These subsets (C and B) hint at a possible way to split P into two more cohesive packages.

Based on that, we suspect that it is possible to remodularize the package, for example by moving C classes to a new package. This will make the package under analysis (P) conceptually more cohesive while providing one group of classes (B) used together by P_5 and P_2. We check this hypothesis by reading the code of B and C classes. We learn that B classes 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.). The referencing package *impl::render::dynamic* (P_2) contains classes that implement some of the interfaces of B. The referencing package *render* (P_5) contains the class *renderContext* that refers to B interfaces. This class *renderContext*, which implements the facade pattern, is responsible of the communication with different objects whose types are declared via the interfaces (e.g., *PageRenderer, WindowRenderer,* etc.). C interfaces are implemented by classes contained in different packages (e.g., *tag::basic, tag*) which are responsible of different contexts of item rendering.

Reading the code reinforced the difference in the usage of both interface collections (B and C) the fingerprint revealed. It consequently reinforced our idea to move C classes to a new package, named for example *render::rendererContext*, for better modularization.

5. Reading and interacting with an incoming fingerprint

Even if we use the same mechanisms for both incoming and outgoing fingerprints, we detail in this section incoming fingerprints. Outgoing fingerprints are briefly described in Section 6. We introduce two levels of zoom-outs to: (a) keep the visualization compact and scalable over a number of referencing packages or the size of the interface; (b) support global visual patterns as presented in Section 7, while minimizing information loss compared to the details presented in Section 4.

Zoom-out level 1. We do not visualize the cell internals. We only visualize in the main diagonal the size of each cell, i.e., the number of referenced classes.
Zoom-out level 2. We visualize the fingerprint without the cell internal information and the size of main diagonal cells.

Fig. 6 shows the fingerprint of the *renderer* package, illustrated in Fig. 5, zoomed-out twice. In the first zoom-out 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 cell which is located on its row. This last information is hidden in the second zoom-out.

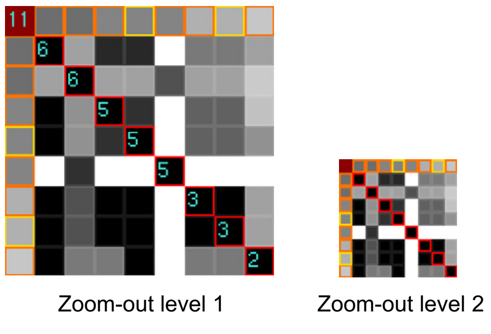


Fig. 6. The incoming fingerprint of *renderer* package (Fig. 5) zoomed-out twice.

5.1. Interacting with the fingerprint

To help users detect quickly information within the fingerprint, we have introduced an interaction mechanism to the visualization, as shown in Fig. 7.

Fig. 7a shows that the selection of a cell makes its fill color gold and its border color green. In addition it automatically selects all cells that display a subset of classes presented by the first selected cell. This highlights a family of packages based on their co-referencing of the analyzed package classes. The fill's color of the cells which are automatically selected is also gold but with different intensity. The cell which contains the biggest number of classes, is the cell with the darkest fill color. We do the same at the class level: The classes that are contained in the selected cell get their fill color green. This highlights a family of the analyzed package classes based on their co-usage.

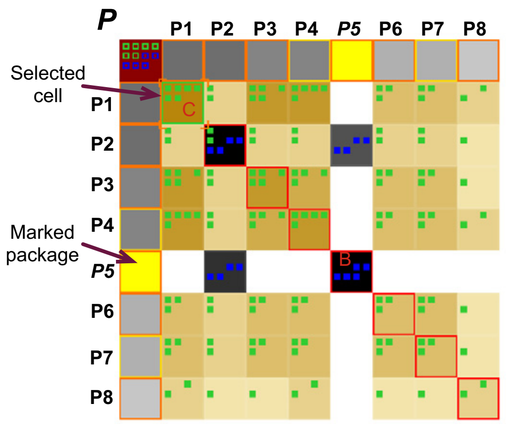
In addition to the selection and marking mechanisms, we have introduced a new interaction with the fingerprint: by moving the cursor over any cell a fly-by-help shows us the size of the cell and the set of the classes it represents (Fig. 7b).

5.2. Reading the fingerprint

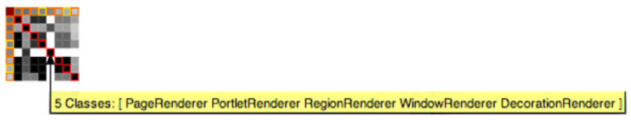
We believe that a package fingerprint, as described in Section 4, helps developers understand and analyze a given package, while the fingerprint zoom-outs help visualize large number of packages, easily navigate in the system and detect global information (e.g., patterns, anomalies, etc.). To understand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

Examples. Fig. 8 shows the incoming fingerprint of the package *utils* of the subsystem *plugins*, taken from *Azureus* system. In the following section we illustrate how to read this incoming fingerprint, and which relevant information we can get.

Size. 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*.
Spread of external incoming references. The top left cell (P) is dark red, which means that most of the package classes are referenced from the outside, i.e., the size of its In-Interface is relatively big.
Distinct part users. The fingerprint fill shows that some cells on the main diagonal (circled in green) are isolated within their row: i.e., the rows 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.
Systematic package external usage. The fingerprint fill shows a black filled rectangle Z_3 at the intersection of the rows and columns of the packages *Pkgs3*. This indicates that the cells within Z_3 represent the same collection of classes that are referenced together by all packages *Pkgs3*. In the same way, we can deduce that those classes are also referenced together by the packages *Pkgs2* and *Pkgs1*: 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 body is an indicator of the package cohesion: *the more black space, the more cohesive the package is.*
Strength of use-based class cohesion. Comparing black filled rectangles according to their size also provides another useful



(a) Interacting with the Fingerprint of renderer package (Figure 5). P_5 is marked in *Yellow* and the cell C is selected (*gold fill and green border).* Thus, all the classes of C are highlighted in *green*. In consequence, each cell that represents *only* a subset of those classes is also selected.



5 Classes: [PageRenderer PortletRenderer RegionRenderer WindowRenderer DecorationRenderer]

(b) Interacting with the zoomed-out Fingerprint of *renderer* package (Figure 5). The cursor is over the cell *B* and a fly-by-help shows us *B* size and the set of the classes it represents.

Fig. 7. Interacting with the fingerprint.

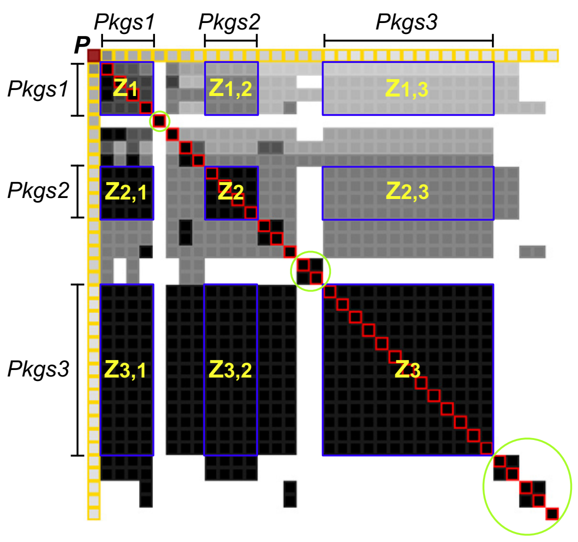


Fig. 8. The incoming fingerprint of *utils* package, from *plugins* subsystem (*Azureus* application).

information related to the cohesion based on usage: *the larger a rectangle size is, the higher the coupling between the classes represented by it* – since more client packages used them together. For example, classes represented by the cells within the rectangle Z_2 are less coupled than the classes represented by the cells within the rectangle Z_3.

User heterogeneous references. The fingerprint body is not symmetrically dark. While the classes that are together referenced by the packages *Pkgs1* and *Pkgs3* are respectively represented by the rectangles $Z_{1,3}$ and $Z_{3,1}$, the former is light gray while the latter is completely black. We deduce then that the classes referenced by *Pkgs3* form a small portion of the classes referenced by *Pkgs1*. Thus, the dissymmetrical darkness of the fingerprint body indicates that the package In-Interface contains classes that are loosely coupled in the context of the package under analysis. As consequence, this is an indicator of a bad organization of classes.

6. Outgoing fingerprint

Up to this point we limited our presentation to incoming references; now we also propose the symmetrical view. The *package outgoing fingerprint* helps maintainers coarsely evaluate the package coupling with the rest of the system and the potential impact of changes on the package, and understand how the package under analysis uses the rest of the system. Also, it focuses on the similarity/coupling between the referencing classes and the cohesion of the considered package, from the point of view of a given provider.

Figs. 9 and 10 depict the key visualization principles of an outgoing fingerprint with P_1 from Fig. 1 as the package under analysis. The principles we described above for an incoming fingerprint (Sections 3.3 and 3.4) are used exactly in the same way, except that we take into account outgoing references instead of incoming ones and referenced packages instead of referencing ones: the referenced packages and the Out-Interface of the package under analysis. In an 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.

Examples. Fig. 11 shows the outgoing fingerprint of *impl::api::user* package. The fingerprint shows several important pieces of information:

A misplaced class. The package Out-Interface involves only two classes, *UserEventBridge* and *UserEventIntercepter*. In the top

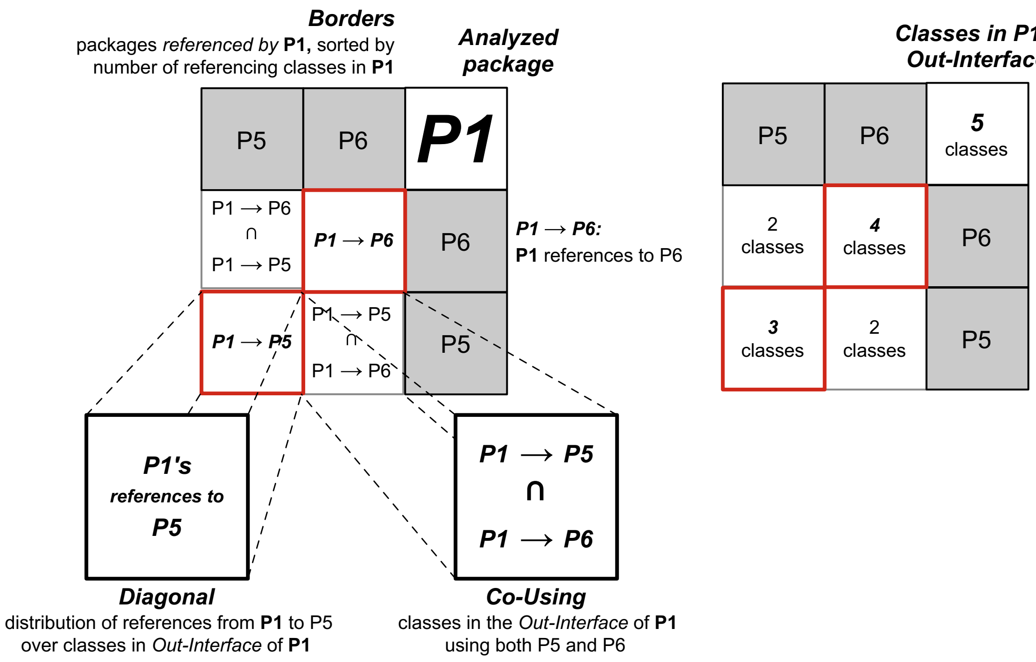

Borders
packages *referenced by* **P1**, sorted by number of referencing classes in **P1**
Analyzed package
Classes in P1's Out-Interface
P5
P6
P1
P1 → P6 ∩ P1 → P5
P1 → P6
P6
P1 → P6: P1 references to P6
P1 → P5
P1 → P5 ∩ P1 → P6
P5
5 classes
2 classes
4 classes
3 classes
2 classes
P1's references to P5
Diagonal
distribution of references from **P1** to P5 over classes in *Out-Interface* of **P1**
P1 → P5 ∩ P1 → P6
Co-Using
classes in the *Out-Interface* of **P1** using both P5 and P6

Fig. 9. Skeleton of the outgoing fingerprint for P_1 (Fig. 2b).

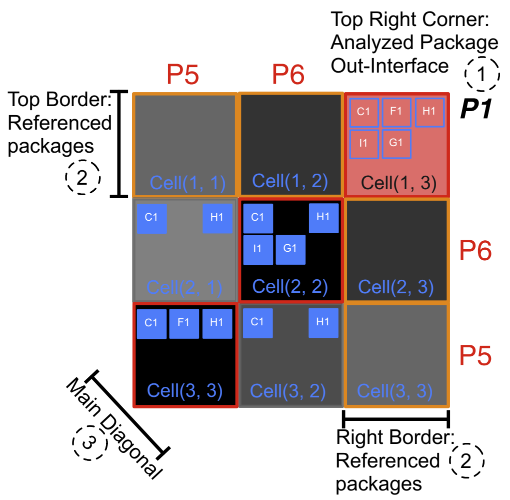


Fig. 10. Showing the outgoing fingerprint of P_1 (Fig. 2b) with the classes involved in the relations inside each cell.

most corner, the square presenting the class *UserEventIntercepter* is not filled, which means that this class does not refer to classes inside the package under analysis *impl::api::user*. On the another hand, this class refers to classes packaged into three packages, the group *Pkgs2*. We suppose then that it is better to move the class *UserEventIntercepter* to one of its provider packages. Inspecting *UserEventIntercepter*, we found that it has neither incoming references nor inheritance inside its own package; it inherits from the class *ServerIntercepter*, which is packaged in *portal::server*, which in turn is one of the provider packages. That enforced our estimation and we think that moving *UserEventIntercepter* to the package *portal::server* will optimize the cohesion of both packages.

Distinct providers used by the package. The classes of the analyzed package reference two *distinct* groups of packages (*Pkgs1* and *Pkgs2* on the figure). Since the cells form two distinct squares of uniform color around the main diagonal, both groups of referenced packages are uniformly accessed.

Distinct reasons for changing the package. The view also reveals the input source for each class of the package Out-Interface. The view shows that each class refers to *distinct* groups of packages/classes. Changes within the group *Pkgs1* directly impacts only the class *UserEventBridge*, while changes within the group *Pkgs2* directly impacts only the class *UserEventIntercepter*. Here we deduce that the package under analysis has two distinct reasons for changing (Definition 4).

7. Visual patterns

While applying fingerprints to large systems (Squeak, Azureus, Jboss, ArgoUML) we identified some recurring visual patterns. We present here the most frequent ones, knowing that several patterns could occur within a single fingerprint. We describe the systems we selected and the reasons why we selected them. We conducted these experiences to show what can be deduced from fingerprints, and how fingerprints help focusing on the code while browsing it. We provide several VisualWorks Smalltalk images with the case studies data presented in this paper loaded, at http://rmod.lille.inria.fr/archives/demos/PackageFingerprints.

7.1. Characterizing the systems under analysis

Table 1 presents some measures about the systems we selected for our experience. The number of packages has been computed by counting only the packages containing the name of the project. For Java projects, we ignore inner classes.

Squeak. Squeak is an open-source Smalltalk environment developed by the team of Alan Kay since 1996. It involves around 20 active developers and 200 committers. It is a really large and complex system containing more than 1600 classes and 32,000 methods in its latest public release (3.8 basic). Squeak includes support for different application domains: two large graphical user-interface frameworks, a complete IDE (including an incremental compiler, debugger and several advanced development tools), a complete language core and its libraries, multimedia sup-

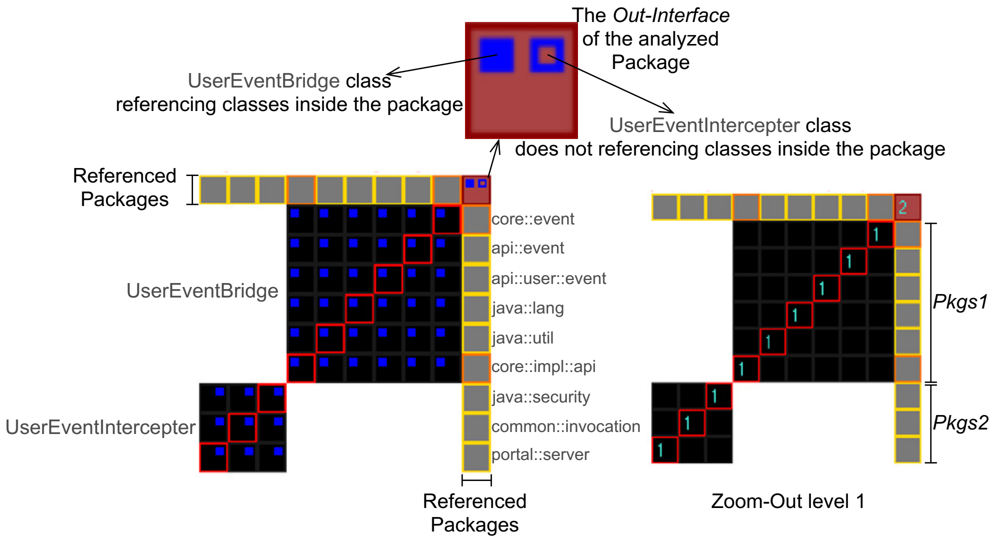


Fig. 11. The outgoing fingerprint of *impl::api::user* package, from the subsystem *Jboss.portal.core*.

Table 1
Overall system data.

System	Release	Packages	Classes	Methods
Squeak	3.8 Basic	223	1659	37,952
JBoss portal	2.6	454	1889	15,498
Azureus	2.5	337	1646	19,745
ArgoUML	0.22	76	2222	11,467

port (images, video, sound, voice generation), eToy (an advanced scripting programming environment for children), various libraries (compression, encryption, networking, XML support).

Our hypothesis: Squeak is based on a monolithic design and was not really engineered with modularity in mind. In addition, it is a large and complex system which evolved over a time span of 15 years. We expect to see co-use, packages with too many responsibilities, as well as some tangling between groups of packages. We are really familiar with the code.

JBoss. JBoss is a widely used Java application server. The domain is more restricted than the one of the previous system. It has a large base of developers. JBoss was also designed in presence of a package system so it should be more modular.

Our hypothesis: JBoss is an industrial standard and we expect it to be of good quality and modularity and to get fingerprints showing cohesive packages. We were not familiar with the code.

ArgoUML. ArgoUML is an UML editing tool. It is a small application, mainly developed by a couple of core developers and several committers. Its design suffers from large facades, but otherwise we were not familiar with the code.

Our hypothesis: ArgoUML has known problems with large classes and we want to see if fingerprints help us to understand how they could be split.

Azureus. Azureus is a peer-to-peer client. It represents a middle size application with a large number of classes.

Our hypothesis: We expect it to be normal application quality and to get fingerprints showing small or medium cohesive packages. We were not familiar with the code.

7.2. Black fill pattern

This pattern is characterized by a complete black fill of the fingerprint as shown in Fig. 12. This pattern occurs when all the package interface classes are conceptually coupled: for an incoming fingerprint, all the In-Interface classes are referenced together by every referencing package, while for an outgoing fingerprint, all the Out-Interface classes refer together to every referenced package.

In our case studies, and in the context of the incoming fingerprint, this pattern occurs for small size In-Interface packages, particularly when they export only one class, or when the package is referenced by a small number of packages. Peripheral packages often present this pattern.

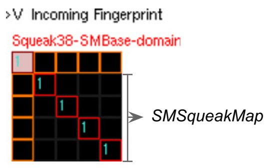
Referenced as a single service. In this pattern, the classes of the package In-Interface are always referenced together as a single service. Thus, such a package is often characterized by a high degree of cohesion because all its classes tend to fulfill a single service, and the package design respects the package cohesion principles REP and CRP (described in Section 2.2).

Referencing all the same services. For outgoing fingerprints, this pattern occurs also for small size package Out-Interface, or when the package refers to a small number of packages. Exhibiting a black fill pattern reveals that all the classes of the package Out-Interface refer together to the same group of packages i.e., same reasons for change. Thus we can conclude that they have a high degree of similarity in terms of required services and responsibility—since all the package classes refer to the same group of packages, they have the same source of changes impact.

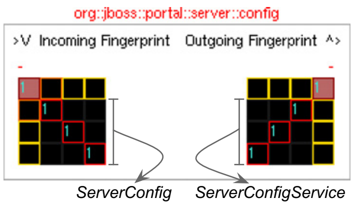
In consequence, packages that exhibit the black fill pattern for incoming and outgoing fingerprints, may represent a good architecture design since: (1) they respect the three cohesion principles, (2) it is easy to know which services the package provides and it provides them to which packages, and (3) maintainer can see quickly which services/packages the package uses. Note that when several classes are doing consistently several and similar references to external classes, leading to an outgoing black fill, this pattern may reveal a lack of factorization within the package violating the DRY (Do not Repeat Yourself) principle [19].

Examples. Fig. 12 shows some fingerprints that present this pattern.

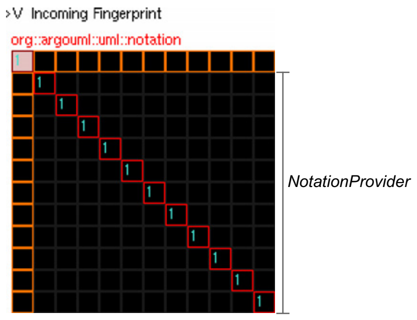
A well encapsulated package. Fig. 12a shows the incoming fingerprint of the package *SMBase::domain* of *Squeak*38, which



(a) The Incoming Fingerprint of *domain* package of the *Squeak38::SMBase* subsystem.



(b) The Incoming and the Outgoing Fingerprints of *config* package of the *jboss::portal::sever* subsystem.



(c) The Incoming Fingerprint of *notation* package of the *argouml::uml* subsystem.

Fig. 12. Examples of *Black Fill* fingerprints.

defines the domain model of a source management system. It shows that *SMBase::domain* exports only one class (*SMSqueakMap*) to only four packages of *Squeak*38 system. Thus we know that the services provided by this package are exactly the role of *SMSqueakMap* class and we know that this class provides specific services – since it is referenced by only four packages within the system. Note that *SMBase::domain* contains 14 classes, but understanding its role requires understanding only one class of those classes. In such a context we say that the package design respects the hidden-information principle.

A provider of abstract service. Fig. 12c shows the incoming fingerprint of *notation* package of *argouml::uml* subsystem. It shows that *uml::notation* exports only one class *NotationProvider*. By reading this class and its hierarchy we found that it is the interface which is implemented by every UML's element notation (e.g., *AttributeNotation*, *MessageNotation*, *ObjectNotation*, etc.). *notation* package includes all those classes (18 classes) but it provides them to the system via their top superclass *NotationProvider*.

A package with a single reason-for-changing. Fig. 12b shows the incoming and outgoing fingerprints of *config* package of the *jboss::portal::server* subsystem. Both fingerprints present the *Black Fill* pattern. The outgoing fingerprint shows that the package Out-Interface contains only one class: *ServerConfigService*. By reading this class we found that it implements the interface *ServerConfig* which is the only class provided by the package: the incoming fingerprint shows that the package In-Interface contain only *ServerConfig*. Thus we deduce that the package has a single reason-for-changing, which is the class *ServerConfigService*. On the another hand, to understand the package role is enough to understand the interface *ServerConfig* or its implementation provided by the class*ServerConfigService*.

*Variation.*The package *invocation*, shown in Fig. 13, 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. We call this variation *Black–White Fill*. The fingerprints that present this pattern are usually larger than those presenting *Black Fill*. Note that the presence of gray layers indicates a degradation of the package cohesion.

Providing a set of layered services. In incoming fingerprints, the *Black–White Fill* pattern indicates that the package In-Interface involves several groups of classes, where each group contains classes that are referenced together, as a single service, by a set of referencing packages. In this pattern, the services are ordered (layered) top-down in the fingerprint: each one uses the services that are layered below it. Fig. 13 shows that the bottommost service *Service1*, which contains the class *PortletInvocation*, is a sub-service of *Service2* and *Service3*: *Service2* augments *Service1* with class *ActionInvocation*, and *Service3* augments *Service2* with class *RenderInvocation*. Relating the importance of a service to the number of packages that refer to it, the provided services are vertically ordered by importance, with the most important package at the bottom. In Fig. 13, the bottommost service *Service1* is referenced by all referencing packages (all cells into that layer are black). In contrast, the three classes of *Service3* are referenced together only by the packages in *Pkgs3*: within the Service3 layer, only cells in the columns of *Pkgs3* are completely black.

Involving a set of layered reasons for changing. For outgoing fingerprints, the *Black–White Fill* pattern indicates that the package Out-Interface involves several group of classes, where each group present classes that refer together to a set of packages. Thus we deduce that each group involves a distinct reason-for-changing. As with services, those groups are layered top-down by the importance of each group's reason-for-changing, i.e., the number of packages the group refers to.

7.3. Arrow pattern

When the only non white cells are the diagonal cells, the fingerprint looks like an arrow.

Providing particular non-coupled services. For incoming fingerprints, this pattern occurs when the package In-Interface involves several independent groups of classes, where the classes in each group are referenced together, as a single service, by only one client package. In other words, each client package refers to only one service: the relationship between provided services and client packages is one-to-one.

This means that the package under study provides non-coupled services to the system. Since each service is used by only one client package, we also deduce that the provided services are particular and of minimal importance, i.e., they are neither general nor core services from the point of view of the package system, and few packages use them.

Fig. 14 shows the incoming fingerprint of *UI* package of the it Squeak38::Monticello subsystem. It shows that the package services are used separately, by individual packages from the same subsystem *Squeak38::Monticello*: the client package *Monticello::Versioning* uses the top service which contains the classes *MCVersionInspector, MCSaveVersionDialog, MCMergeBrowser* and *MCChangeSelector*; the client package *Monticello::Repositories* uses another service containing classes *MCFileRepositoryInspector* and

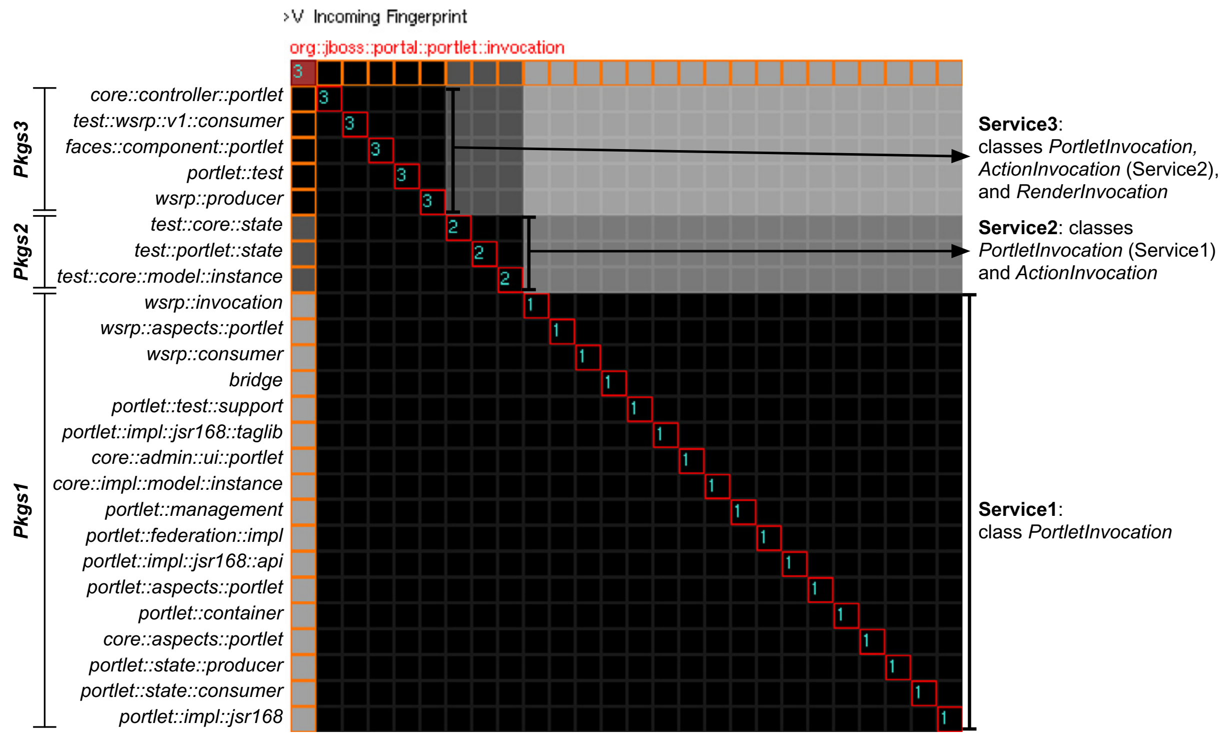

>V Incoming Fingerprint
org::jboss::portal::portlet::invocation
Pkgs3
Pkgs2
Pkgs1
core::controller::portlet
test::wsrp::v1::consumer
faces::component::portlet
portlet::test
wsrp::producer
test::core::state
test::portlet::state
test::core::model::instance
wsrp::invocation
wsrp::aspects::portlet
wsrp::consumer
bridge
portlet::test::support
portlet::impl::jsr168::taglib
core::admin::ui::portlet
core::impl::model::instance
portlet::management
portlet::federation::impl
portlet::impl::jsr168::api
portlet::aspects::portlet
portlet::container
core::aspects::portlet
portlet::state::producer
portlet::state::consumer
portlet::impl::jsr168
Service3: classes PortletInvocation, ActionInvocation (Service2), and RenderInvocation
Service2: classes PortletInvocation (Service1) and ActionInvocation
Service1: class PortletInvocation

Fig. 13. An example of the *Black–White* pattern: the incoming fingerprint of *invocation* package, from *Jboss* system.

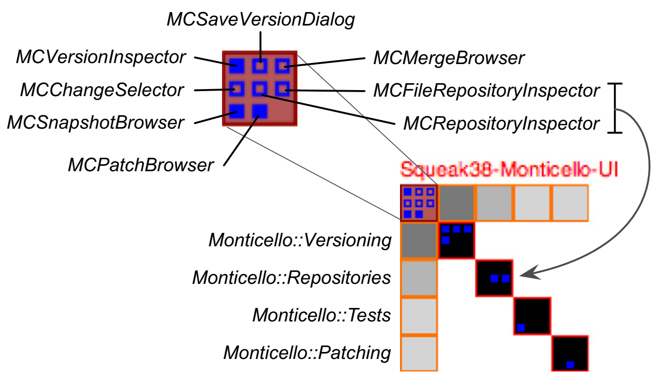


Fig. 14. Arrow Pattern: the incoming fingerprint of *UI* package of the *Squeak*38::*Monticello* subsystem.

MCRepositoryInspector; the last two client packages *Monticello::Tests* and *Monticello::Patching* each use a one-class service.

Involving particular non-coupled reasons for changing. For outgoing fingerprints, the strict occurrence of this pattern appears when the package Out-Interface involves several groups of classes, where the classes in each group refer together, as a single reason-for-changing, to only one provider package. Like with the incoming fingerprint, there is a one-to-one relationship between the provider packages and reasons for changing. The concerned package has thus several non-coupled/mixed reasons for changing, and since these reasons for changing each use only one provider package, they are relatively simple or clear.

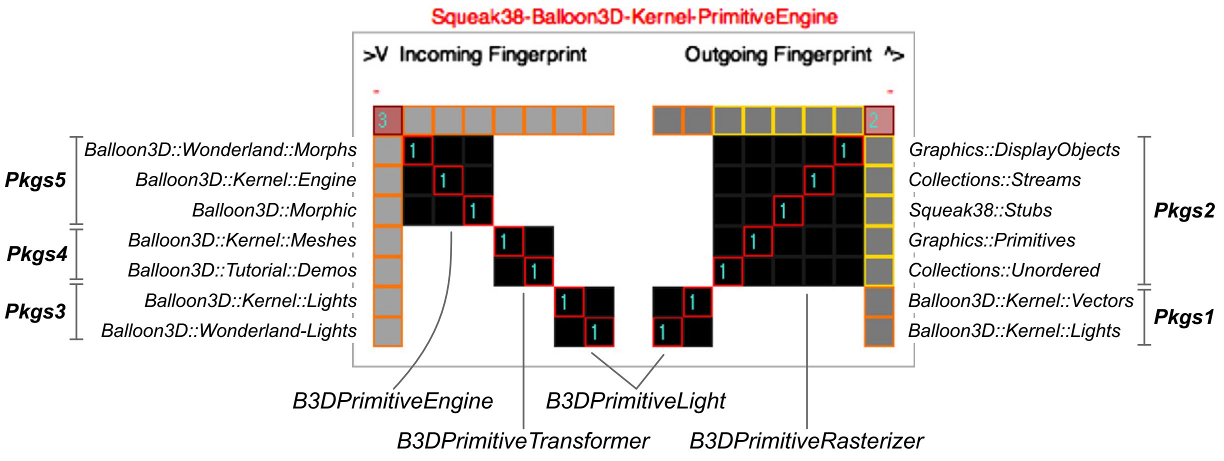
Package may be a candidate for splitting. Since the occurrence of *Arrow* pattern indicates that the concerned package provides particular services that are used separately or/and it has several non-coupled reasons for changing, the pattern indicates that such a package could be a candidate for splitting: moving some classes of the package In-Interface/Out-Interface to their referencer/referenced packages may optimize package internal cohesion and reasons for changing.

For example, Fig. 14 shows that the *UI* package has a typical *Arrow* incoming fingerprint: the four services provided by package *UI* are never referenced together. Moreover, the service containing the classes *MCFileRepositoryInspector* and *MCRepositoryInspector* could be moved out, because they are not referenced from within *UI* (they appear as hollow squares in the top left cell). In fact, further inspection of these two classes revealed that they participate in circular dependancies, and that moving them to *Monticello::Repositories* would improve its internal cohesion as well as that of *UI*, while freeing *UI* of these circular dependancies.

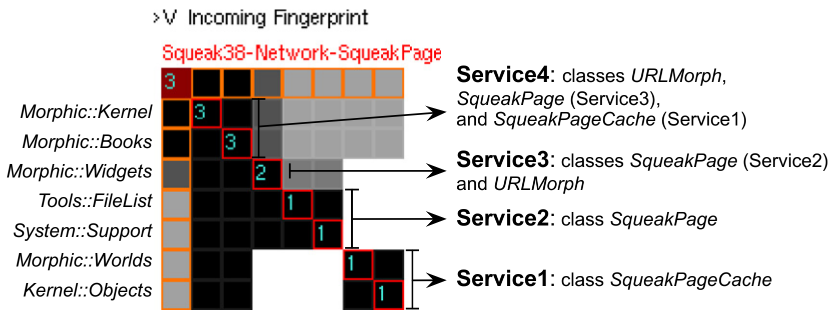
Variant: non-coupled services (reasons for changing) with distinct importances. A frequent variation of the *Arrow* pattern is when the fingerprint's main diagonal shows black squares, rather than individual cells, as in Fig. 15a. Again, the presence of squares around the diagonal only is a good indication that the functionality of the packages is not cohesive from the client/provider point of view. For instance, looking at the outgoint fingerprint in Fig. 15a, we see that the package under study has two non-coupled reasons for changing, the most important one coming from the references made by the class *B3DPrimitiveRasterizer* to the five packages in *Pkgs2*.

Variant: providing loosely-coupled services. A variation of the *Arrow* pattern occurs for incoming fingerprints when some of the provided services, if not all, are used together by a few number of referencing packages. In this case, we say that the package services are loosely coupled and some of the referencing packages appear as dominant over the other referencing packages.

Fig. 15b shows the incoming fingerprint of *Network::SqueakPage* package of the *Squeak38* system. The incoming fingerprint has the



(a) The Incoming and Outgoing Fingerprints of *Kernel::PrimitiveEngine* package of the *Squeak38::Balloon3D* subsystem.


>V Incoming Fingerprint
Squeak38-Network-SqueakPage
Morphic::Kernel
Morphic::Books
Morphic::Widgets
Tools::FileList
System::Support
Morphic::Worlds
Kernel::Objects
Service4: classes URLMorph, SqueakPage (Service3), and SqueakPageCache (Service1)
Service3: classes SqueakPage (Service2) and URLMorph
Service2: class SqueakPage
Service1: class SqueakPageCache

(b) The Incoming Fingerprint of *SqueakPage* package of the *Squeak38::Network* subsystem.

Fig. 15. The variations of *Arrow* pattern.

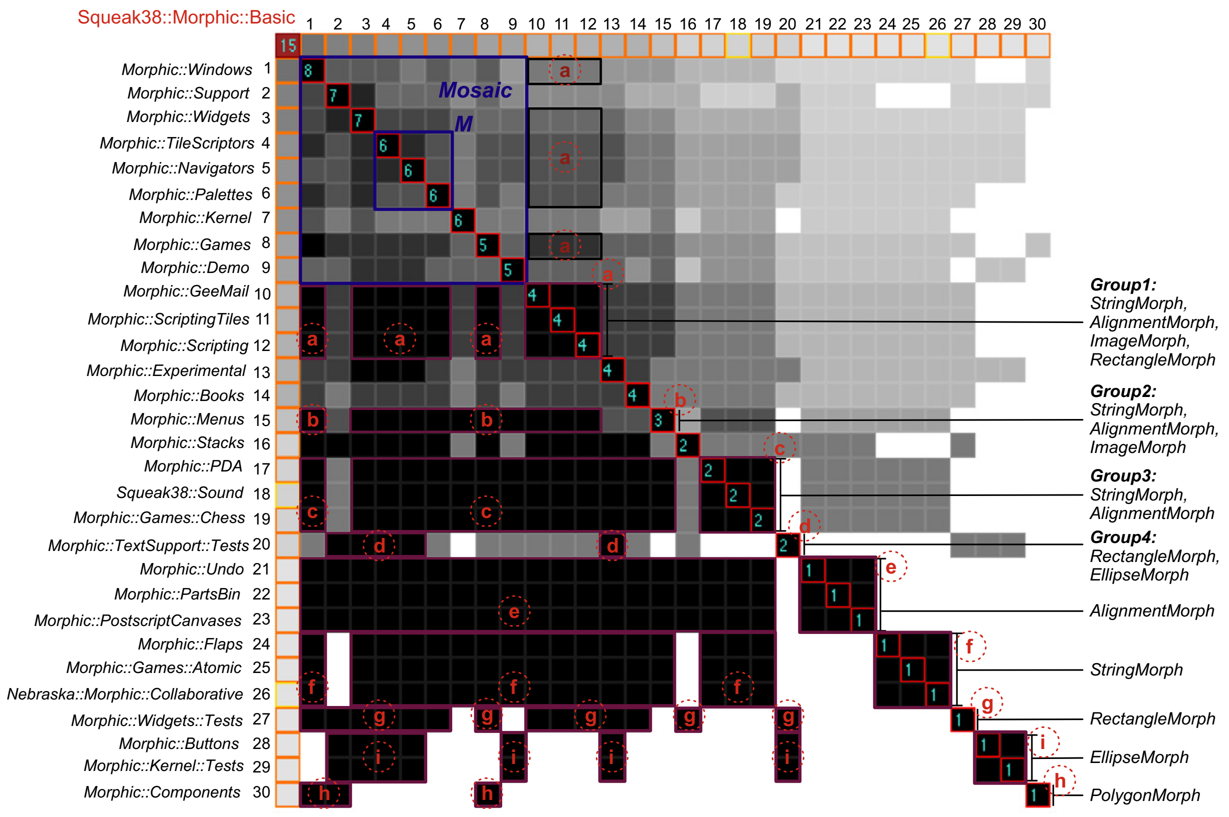


Fig. 16. An example of the *Mosaic* pattern: the incoming fingerprint of *Morphic::Basic* package, from *Squeak*38 system.

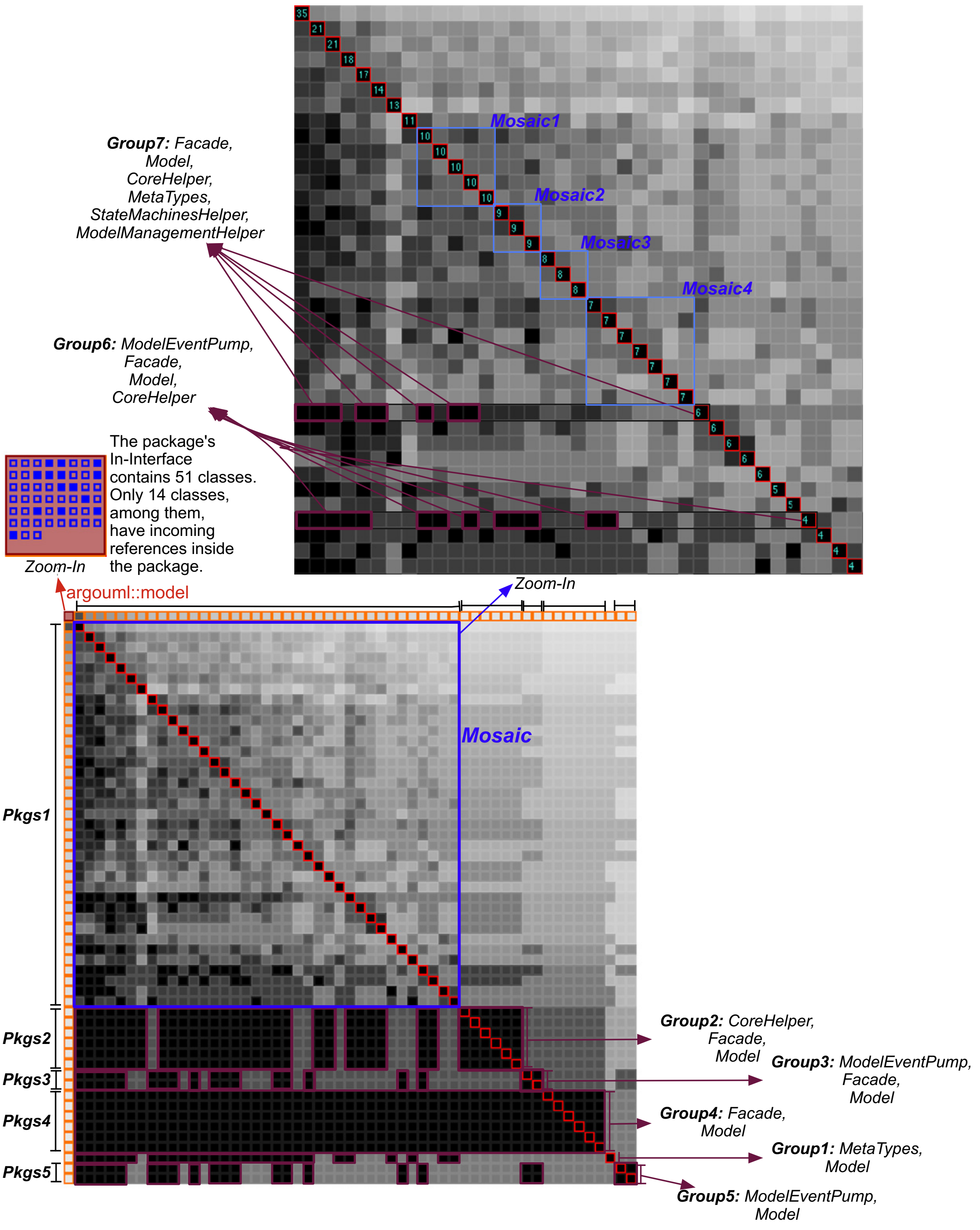

Mosaic1
Mosaic2
Mosaic3
Mosaic4
Group7: Facade, Model, CoreHelper, MetaTypes, StateMachinesHelper, ModelManagementHelper
Group6: ModelEventPump, Facade, Model, CoreHelper
The package's In-Interface contains 51 classes. Only 14 classes, among them, have incoming references inside the package.
Zoom-In
argouml::model
Zoom-In
Mosaic
Pkgs1
Pkgs2
Pkgs3
Pkgs4
Pkgs5
Group2: CoreHelper, Facade, Model
Group3: ModelEventPump, Facade, Model
Group4: Facade, Model
Group1: MetaTypes, Model
Group5: ModelEventPump, Model

Fig. 17. An example of the *Mosaic* pattern: the incoming fingerprint of *model* package, from *Argouml* system.

described pattern variation. It shows that the packages *Morphic::Kernel* and *Morphic::Books* are dominant referencing packages. They refer to all the classes provided by *Network::SqueakPage* package (three classes: *SqueakPageCache*, *SqueakPage* and *URLMorph*). The rest of referencing packages refer to distinct groups of those classes: The referencing packages *Kernel::Objects* and *Morphic::Worlds* refer to the class *SqueakPageCache* (*Service1*); the referencing packages *System::Support* and *Tools::FileList* refer to the class *SqueakPage* (*Service2*); the referencing package *Morphic::Widgets* refers, in addition to (*Service2*), to the class *URLMorph*. Thus the classes of the package In-Interface are used together by only two packages, while the package has seven referencing packages. We then deduce that the provided services are loosely coupled in the context of *Network::SqueakPage* package.

7.4. Mosaic pattern

In this pattern, mosts cells are gray but they do not have an homogenous darkness, e.g., the incoming fingerprints of the *Basic* package in *Squeak38::Morphic* or the *model* package in *Argouml* (Figs. 16 and 17).

Large package interface size. The *Mosaic* pattern occurs usually for packages whose interfaces (In-Interface and Out-Interface) contain a large number of classes. The incoming fingerprint of *model* package (Fig. 17) shows that this package has a large In-Interface, containing 51 classes, or 45% of the 112 total classes the package contains.

Moreover, out of those In-Interface classes, only 14 have incoming references inside the *model* package.

Core and central package. This patten occurs usually with giant fingerprints, i.e., a pattern that reveals a package with a large interface and which is coupled to a large number of other packages. In the case of incoming fingerprints, this means that the package provides a lot of services (i.e., groups of classes always referenced together, see Section 3.1) that are used by an important number of packages within its system (see Section 7.5).

Examples. The *Basic* package whose incoming fingerprint (Fig. 16) has the *Mosaic* pattern is also a core package within its system *Morphic*. *Basic* package provides 15 classes to 30 packages. Only two packages of the referencing packages are stubs, i.e., they do not belong to the *Morphic* system. Those stubs are *Squeak38::Sound* package (denoted by 18) and *Nebraska::Morphic::Collaborative* package (denoted by 26). Thus, *Basic* package provides 15 classes to 28 of the 45 *Morphic* packages. This means that more than 62% of the *Morphic* packages depend upon *Basic* package and this last is a core and a central package within *Morphic*.

Another example, the *model* package whose incoming fingerprint (Fig. 17) has the *Mosaic* pattern is a core package within its system *ArgoUML*. In addition to the fact that it contains 112 classes of the 1671 *ArgoUML* classes, it is also referenced by 54 packages of the 76 *ArgoUML* packages. Also the package In-Interface contains 51 classes, which means 71% of *ArgoUML* packages depend directly on 51 classes within the *model* package. This means that the whole *ArgoUML* system highly depends on the *model* package and this last plays the role of core and central package within *ArgoUML*.

Imprecision and difficulty in determining package usage and role. The occurrence of *Mosaic* pattern for incoming fingerprints indicates that the package under analysis provides a large number of functionalities that are accessed by a large number of packages in an arbitrary way, i.e., non-consistent way. Thus in presence of this pattern, it is hard to know which functionalities are used together and which are not. As a result, it is hard to identify the role/functionality and to determine the contextual cohesion of the considered package.

A deeper analysis of the Mosaic pattern is described in [2].

7.5. Other patterns

We present some other less frequent but still interesting patterns with less details.

7.5.1. Unbalanced pattern

This pattern occurs when an incoming or outgoing fingerprint appears clearly bigger than its counterpart (i.e., its outgoing or incoming fingerprint). 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. Figs. 8, 17 and 16 show respectively that *plugins::utils*, *argouml::model* and *Morpic::Basic* exhibit 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.

Empty-incoming fingerprint. The incoming fingerprint is empty and the package has no incoming references. It is the case of packages that include abstract classes that are implemented in other packages. This pattern appears for packages that are leaves in the package structure. This is often the case for UI application packages.

7.5.2. Golden border pattern

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 Fig. 8). Such packages represent the border of the analyzed subsystem. This pattern is usually a good sign because it indicates that the system under analysis tends to be well layered.

Ideally, a subsystem should be composed of three distinct layers of packages: the first layer presents packages that refer only to packages outside the subsystem – thus they have *Golden Border* outgoing fingerprints – and are not referenced by packages outside the subsystem; the second layer presents packages that interact only with packages inside the subsystem; the third and last layer presents packages that refer only to packages inside the subsystem and are referenced by only packages outside the subsystem – thus they have *Golden Border* incoming fingerprints. Whatever, analyzing and understanding subsystem architecture/layers need a global view of the analyzed subsystem. DSM views [41] are more suitable for such analysis.

On the another hand, if a package has *Golden Border* outgoing and incoming fingerprints, this means that the concerned package is bad placed within the analyzed subsystem – since it has no incoming or outgoing references with the subsystem packages and it interacts only with packages outside the subsystem.

7.6. Patterns frequency

To give an impression to the readers of the relative importance of the patterns, we studied of their frequency in the four case studies. Our process is the following one: (1) we picked random pack-

Table 2
Overall system data – number of patterns (percentage relative to 396 packages examined).

Pattern	Number		Description
Black fill	114	(28%)	Completely filled with black squares
Black & gray fill	126	(31%)	Uniform blocks of black or gray squares
Arrow	106	(26%)	Square groups around the diagonal
Break	78	(19%)	Filled, except for one spot
Mosaic	78	(19%)	Irregular shades of gray, lots of holes
Filled mosaic	9	(2%)	Filled but irregular shades of gray
Micro	116	(29%)	2–4 interacting packages
Large	62	(15%)	More than 10 interacting packages
Extra large	18	(4%)	More than 25 interacting packages
Empty incoming	88	(22%)	With no cells
Empty outgoing	14	(3%)	With no cells

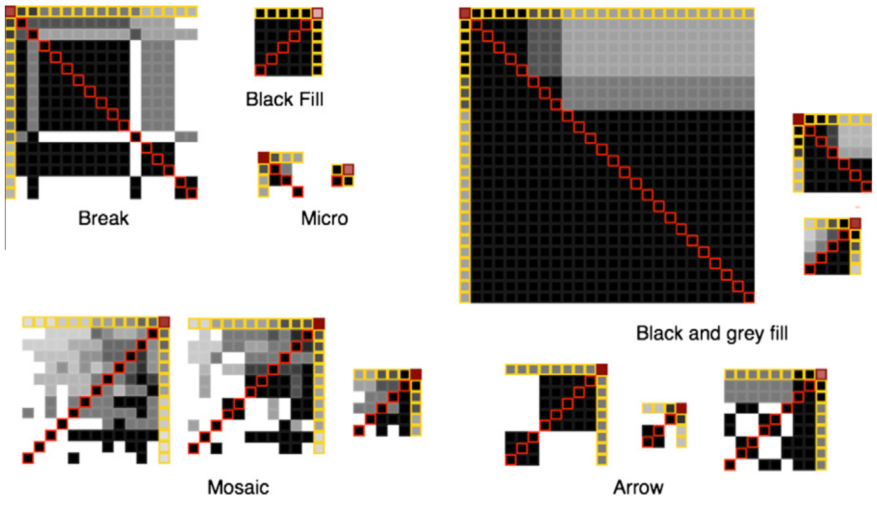


Fig. 18. Some instances of the patterns.

ages and applied the incoming and outgoing fingerprints, (2) we visually identified some of the patterns mentioned previously and that are briefly described in Table 2. In total, we evaluated 396 packages over a total of 1265 (31%). When a fingerprint exhibits several patterns at once, we record one occurrence for each of the patterns. Fig. 18 shows several instances of the patterns to illustrate possible variations. Sometimes the distinction between two patterns is fuzzy, so the categories are not totally rigorous. The size consideration in the patterns is mentioned to give an impression of the system from the class usage point of view. Finally, we are aware that tallying pattern occurrences is not a validation of the approach in itself. We only present these statistics to give an overall impression of its relevance, but the limited number of case studies certainly biases which patterns appeared most.

During our systematic application of fingerprint, we noticed that it is interesting to see packages exhibiting nearly identical patterns. We also got really fast information about packages having classes not been referenced in the system. We could see differences between the projects: for example, in ArgoUML, outgoing fingerprints often contained arrows and breaks, and were larger than in JBoss or Azareus. ArgoUML also presented a couple of really large incoming fingerprints, mainly because they offered a complex UI. Since Squeak defines a complete language we had a large variety of fingerprints and many Mosaics, since the packages were not designed to be modular or cohesive. In general we noticed a large number of two or three cell wide fingerprints, which indicates focused client or user packages.

8. Discussion and lessons learned

8.1. Graphical concerns

Fingerprints show in a condensed form, the existing situation of the code. When packages are not well-designed the patterns are less apparent but the visualization remains useful as it conveys a lot of information on the package usage among the system.

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. Now we discuss some design points.

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 referencing packages in an incoming fingerprint from the most referencing one at the top to the least referencing at the bottom (and similarly with referenced packages in outgoing fingerprints).

Seriation. We ordered referencing/referenced packages that make the same number of references by similarity based on common referenced/referencing classes into the package under analysis: the largest number of common referenced/referencing classes that two client/provider packages have, the biggest similarity the two packages have; this way, the reader can see which packages access, or are accessed by, the same groups of classes. During the design of the fingerprint, 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 system under analysis in gold. We found it really effective to use color to identify the currently selected entities so that the user can interactively mark entities on which s/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 interfaces. This way, the user can visualize large systems, navigate in the system, spot global patterns and conceptual anomalies. Then he can focus on any package by zooming into the detailed fingerprint.

However, during our experiments, we found that detailed fingerprints 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 fingerprints where the package's interface and number of related packages are very large. In fact, in such cases, none of the detailed views we applied has scaled well. Zooming mechanisms [42] helped us solve this problem.

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

8.2. Package cohesion

The presence of dark homogenous zones is a good indicator of the package cohesion. To assess whether fingerprints do offer a good indication on cohesion, we computed the Common-Use

(CU) metrics defined by Ponisio [38] on packages whose fingerprints clearly showed that they were not cohesive (for example the packages *model* (Fig. 17) and *Basic* (Fig. 16)). The CU metric computes package cohesion from the reuse of the classes of the package In-Interface. It takes its value between 0 (no cohesion) and 1 (best cohesion) [38]. Applying the CU metric gave the following results: 0.63 for the *Basic* package and 0.7 for the *model* package. This means that the design of the latter is better than the design of the former and both packages are considered as relatively cohesive, which is not what the fingerprint revealed. This experiment is quite limited and this is one of our future work. However, it does show that the fingerprint is much more than just metrics: the fingerprint shows which classes are coupled, and which ones are not, in a consistent way; it also shows the proportion of those classes within the package In-Interface.

8.3. Hints for code improvements and fingerprint limitations

Incoming fingerprint helps maintainer answer the following questions about a given package:

- Which In-Interface classes are used together, in a consistent way, as a single service? And which are not used together, also in a consistent way, as distinct services?
- Where is a group of In-Interface classes used as a single service? and where is it mixed with distinct classes of the concerned In-Interface?
- Which referencing packages refer to a given group of In-Interface classes? And which ones do not refer to that group?
- Which In-Interface classes are highly coupled (i.e., used together by a large number of referencing packages)? And which ones are loosely coupled (i.e., used together by a small number of referencing packages)?
- Which In-Interface classes are considered as most important (i.e., classes that are referenced by most referencing packages)? And which In-Interface classes are less important?

Fingerprints limitations. Package Fingerprints focus on the package contextual cohesion, coupling, and co-use of internal classes. However, they do not provide a good map for internal references; our aim is to support understanding packages through their interfaces, regardless what happens inside them. With package fingerprint, we consider related packages (e.g., referencing packages in an incoming fingerprint) as black boxes; we only pay attention to package classes while we look at its fingerprint. This is clearly a limitation of fingerprints.

Since package fingerprints do show partial information about package internal references, hints at improvements, which are revealed by fingerprints, should be verified and complemented by other information/views. For example, we illustrated in Section 7.3 that an *Arrow* pattern indicates that the concerned package may be a candidate for splitting—since it provides distinct non-coupled services. For such a case, before deciding to split the concerned package, maintainer needs to know if classes that are not contextually coupled interact with each other. S/he needs to verify if there are references or inheritance relationships among classes that are used by distinct packages before deciding to split the package or to move some classes of that package to other ones.

The primary goal of package fingerprints is to give a fast visual understanding. It helps maintainers decide whether and how to remodularize. However, this is not always possible when code is too complex, as illustrated by the mosaic fingerprints. It should be noted that we do not oppose fingerprints to other techniques; on the contrary, we see them as complementary. For example, independently of this work and this article, we developed complementary tools to help dealing with packages such as Package Blueprints (which is based on a zoom-out visualization and ordering of internal references between packages) [17]; Orion (which supports the prediction of change impact) [26] and a Simulated Annealing search-based technique to help maintainers find good alternative modularizations [4]. Maintainers can then use the fingerprints to compare the results, understand the alternatives, and choose the most suitable modularization.

8.4. A limited user study

Package Fingerprints are a dense and compact visualization, they were designed to have such property. Still, users may have difficulty extracting all the information from them. Our current work lacks a serious user study.

We performed some limited studies with members and students of our team not working on fingerprint or visualization in general but on new language design. The experience was conducted as follows: (1) we presented the fingerprint principles, (2) provided some simple case with explanation, (3) then we presented some fingerprints and asked questions about the fingerprints, (4) at the end we asked if the fingerprints were useful using a set of questions using the Likert scale.[2] The experience is rather limited since we got only 11 participants but we can already draw some conclusions.

- *Positive points.* Our preliminary results show that a first level of understanding is easy to get: identifying groups of co-referencing/co-referenced classes; identifying distinct provided services and distinct reasons for changing; identifying referencing and referenced packages; etc.

Those users found that, the direction of the diagonal as well as the small annotations we put on top of the fingerprint to distinguish incoming/outgoing fingerprint are very helpful. Fingerprints also support fly-by-help, whose use suggests that showing the names of the packages on the side may really help creating a deeper context. In addition, showing the referenced/referencing classes with the fly-by-help makes the visualizations less abstract.

- *Negative points.* We learned that a deeper level of information extraction is difficult. This is the case with packages that have very large interfaces and classes coupled in a non-consistent way (e.g., the Mosaic pattern). More problematic, this limited study shows that more than a third of the participants (4/11) was unable to quickly understand the semantics of the shades of gray, and why the matrixes are not symmetrical. In addition, understanding the staircase effects as shown in Fig. 13 was difficult to grasp for some participants (3/11).

This suggests that fingerprint is good for a fast overview, but further usability enhancements and studies are required to really prove the usability when too many details are presented.

9. Related work

Several works focus on understanding packages. These can be by supporting high-level analyses of package relationships, visualizations, software metrics or package evolution.

High-level analyses. We are interested here on those based on visualizations. Sangal et al. adapted the dependency structure matrix (DSM) from the domain of process management to analyze architectural dependencies in software [41]. DSM presents a con-

[2] Five levels from "strongly disagree" to "strongly agree" http://en.wikipedia.org/wiki/Likert_scale

sistent visualization that offers a system overview. While the visualization scales for large systems, it is poor in terms of precise information about the package. DSM cells contain a number indicating the number of references made between packages. However DSM did not focus on packages cohesion and co-use or co-usage of classes.

X. Dong et al. [14] present the High-level Object Dependency Graph (HODG) that helps capturing, from a high-level point of view, possible usage dependencies among coarse-grained software entities, namely packages. In their approach, they interpret the usage dependencies between classes in the context of their hierarchy and present a new graph of the system under analysis. While the given graph is helpful for understanding the considered system from a high-level point of view, it does not give any information about package cohesion nor about the co-use or the similarity between classes. Also, their graph visualization still difficult to be interpreted by human eyes because within it, the nodes have different sizes but without any meaningful dimension. The HODG has not visual semantics and it uses numbers to visualize almost information.

Program visualization and navigation. 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 [17] presents a condensed view of a package in terms of the references made between packages. It acts as a map and puts in situation the references between packages. While package blueprint provides a compact view and shows dependencies on a per-class basis, it does not help to group the client/provider packages in terms of their dependencies to the package under analysis.

Kuhn et al. used information retrieval to exploit linguistic information. He introduced semantic clustering to group source artifacts that use similar vocabulary [24]. He uses vocabulary topics to reveal the intention of the code and the similarity between its artifacts, then he provides a consistent visualization.

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 [27]. Storey et al. also worked on system exploration, supporting zoom-out facilities and forces-based graph layouts [44]. However the work did not focus on co-use or co-usage of classes.

Software Metrics. There is a plethora of software metrics on cohesion: from the simple and bogus LCOM ([12]) to more advanced LCOM* metrics [7]. Ponisio et al. introduced the notion of use cohesion [38], which is at the foundation of the fingerprint. E. Hautas defines a new metric that indicates the percentage of changes to be made in order to make a package structure acyclic [22]. While he focuses only on the cyclic dependencies, he does not provide any utility that helps understanding packages or indicating their cohesion or similarity.

Package evolution. 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 [16]; Pzinger et al. show the evolution of package metrics using Kiviat diagrams [37]; Chuah and Eick use rich glyphs to characterize software artifacts and their evolution (number of bugs, number of deleted lines, kind of language...) [11]. In particular, the timewheel exploits pre-attentive 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 [15].

Co-evolution. Other works treat and visualize information about software co-change evolution, looking at coupling from a temporal perspective, and software development teams and activities [10,18,20,43,47,50]. Such approaches are completementary to ours in the sense that we only focus on the static nature of the packages and their relationships. While those approaches are valuable and provide fine-grained views of packages that may help understanding the contextual coupling and cohesion inside packages, they fall short on the analysis of a single version of a system.

10. Conclusion

In this paper, we tackled the problem of understanding the details of package relationships from a usage perspective. 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 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 systems and we have been able to quickly point out badly designed packages, and to extract relevant patterns. While applying fingerprints to large systems that contain radically different packages in terms of size and references, the visualization generally scaled well without hindering the detection of different patterns.

References

[1] Hani Abdeen, Ilham Alloui, Stéphane Ducasse, Damien Pollet, Mathieu Suen, Package Reference Fingerprint: a Rich and Compact Visualization to Understand Package Relationships, IEEE Computer Society Press, 2008. pp. 213–222.

[2] Hani Abdeen, Visualizing, assessing and re-modularizing object-oriented architectural elements, PhD thesis, Université de Lille, 2009.

[3] Erik Arisholm, Lionel C. Briand, Audun Foyen, Dynamic coupling measurement for object-oriented software, IEEE Transactions on Software Engineering 30 (8) (2004) 491–506.

[4] Hani Abdeen, Stéphane Ducasse, Houari A. Sahraoui, Ilham Alloui, Automatic package coupling and cycle minimization, in: International Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press, Washington, DC, USA, 2009, pp. 103–112.

[5] Fernando Brito e Abreu, Miguel Goulão, Coupling and cohesion as modularization drivers: are we being over-persuaded?, in: CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, IEEE Computer Society, Washington, DC, USA, 2001, p 47.

[6] Nicolas Anquetil, Timothy Lethbridge, Experiments with clustering as a software remodularization method, in: Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering), 1999, pp. 235–255.

[7] Lionel C. Briand, John W. Daly, Jürgen K. Wüst, A unified framework for cohesion measurement in object-oriented systems, Empirical Software Engineering: An International Journal 3 (1) (1998) 65–117.

[8] Lionel C. Briand, John W. Daly, Jürgen K. Wüst, A unified framework for coupling measurement in object-oriented systems, IEEE Transactions on Software Engineering 25 (1) (1999) 91–121.

[9] Lionel C. Briand, John W. Daly, Jürgen, K. Wüst, Using coupling measurement for impact analysis in object-oriented systems, in: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999), 1999, pp. 475–482.

[10] Dirk Beyer, Co-change visualization, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM), Industrial and Tool volume, 2005, pp. 89–92.

[11] Mei C. Chuah, Stephen G. Eick, Information rich glyphs for software management data, IEEE Computer Graphics and Applications 18 (4) (1998) 24–29.

[12] Shyam R. Chidamber, Chris F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.

[13] Marcus Denker, Stéphane Ducasse, Software evolution from the field: an experience report from the Squeak maintainers, in: Proceedings of the ERCIM working group on software evolution (2006), Electronic Notes in Theoretical Computer Science, vol. 166, Elsevier, 2007, pp. 81–91.

[14] Xinyi Dong, M.W. Godfrey, System-level usage dependency analysis of object-oriented systems, in: ICSM 2007: IEEE International Conference on Software Maintenance, October 2007, pp. 375–384.

[15] Marco D'Ambros, Michele Lanza, Reverse engineering with logical coupling, in: Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering), 2006, pp. 189–198.

[16] Stéphane Ducasse, Michele Lanza, Laura Ponisio, Butterflies: a visual approach to characterize packages, in: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05), IEEE Computer Society, 2005, pp. 70–77.

[17] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, Ilham Alloui, Package surface blueprints: visually supporting the understanding of package relationships, in: ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance, 2007, pp. 94–103.

[18] Stephen Eick, Todd Graves, Alan Karr, Audris Mockus, Paul Schuster, Visualizing software changes, IEEE Transactions on Software Engineering 28 (4) (2002) 396–412.

[19] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999 (ordered but not received).

[20] Jon Froehlich, Paul Dourish, Unifying artifacts and activities in a visual tool for distributed software development teams, in: Proceedings of the 26th International Conference on Software Engineering, IEEE Computer Society, Washington, DC, USA, 2004, pp. 387–396.

[21] Martin Fowler, Reducing coupling, IEEE Software (2001).

[22] Edwin Hautus, Improving java software through package structure analysis, in: IASTED International Conference Software Engineering and Applications, 2002.

[23] C.G. Healey, K.S. Booth, J.T. Enns, Harnessing preattentive processes for multivariate data visualization, in: GI '93: Proceedings of Graphics Interface, 1993.

[24] Adrian Kuhn, Stéphane Ducasse, Tudor Gîrba, Semantic clustering: identifying topics in source code, Information and Software Technology 49 (3) (2007) 230–243.

[25] Michele Lanza, Stéphane Ducasse, Polymetric views—a lightweight visual approach to reverse engineering, Transactions on Software Engineering (TSE) 29 (9) (2003) 782–795.

[26] Jannik Laval, Simon Denier, Stéphane Ducasse, Andy Kellens, Supporting incremental changes in large models, in: Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2009), Brest, France, 2009.

[27] Mircea Lungu, Michele Lanza, Tudor Gîrba, Package patterns for visual architecture recovery, in: Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering), IEEE Computer Society Press, Los Alamitos CA, 2006, pp. 185–196.

[28] Michele Lanza, Radu Marinescu, Object-Oriented Metrics in Practice, Springer-Verlag, 2006.

[29] Robert C. Martin, Granularity, 1996. <www.objectmentor.com>.

[30] Robert C. Martin, Design principles and design patterns, 2000. <www.objectmentor.com>.

[31] Robert C. Martin, Srp: the single responsibility principle, 2002. <www.objectmentor.com>.

[32] Brian S. Mitchell, Spiros Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Transactions on Software Engineering 32 (3) (2006) 193–208.

[33] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, E.R. Gansner, Bunch: a clustering tool for the recovery and maintenance of software system structures, in: Proceedings of ICSM '99 (International Conference on Software Maintenance), IEEE Computer Society Press, Oxford, England, 1999.

[34] Hayden Melton, Ewan Tempero, The crss metric for package design quality, in: ACSC '07: Proceedings of the Australian Computer Science Conference, 2007.

[35] David L. Parnas, On the criteria to be used in decomposing systems into modules, CACM 15 (12) (1972) 1053–1058.

[36] Marian Petre, Why looking isn't always seeing: readership skills and graphical programming, Communications of the ACM 38 (6) (1995) 33–44.

[37] Martin Pinzger, Harald Gall, Michael Fischer, Michele Lanza, Visualizing multiple evolution metrics, in: Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization), St. Louis, Missouri, USA, May 2005, pp. 67–75.

[38] Laura Ponisio, Oscar Nierstrasz, Using context information to re-architect a system, in: Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06), 2006, pp. 91–103.

[39] Linda Rising, Frank W. Calliss, Problems with determining package cohesion and coupling, Software – Practice and Experience 22 (7) (1992) 553–571.

[40] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, Ben Hallen, The structure and value of modularity in software design, in: ESEC/FSE 2001, 2001.

[41] Neeraj Sangal, Ev Jordan, Vineet Sinha, Daniel Jackson, Using dependency models to manage complex software architecture, in: Proceedings of OOPSLA'05, 2005, pp. 167–176.

[42] Margaret-Anne D. Storey, Theories, methods and tools in program comprehension: past, present and future, in: 13th International Workshop on Program Comprehension (IWPC), 2005, pp. 181–191.

[43] Margaret-Anne D. Storey, Davor Čubranić, Daniel M. German, On the use of visualization to support awareness of human activities in software development: a survey and a framework, in: SoftVis'05: Proceedings of the 2005 ACM Symposium on Software Visualization, ACM Press, 2005, pp. 193–202.

[44] Margaret-Anne D. Storey, Kenny Wong, F.D. Fracchia, Hausi A. Müller, On integrating visualization techniques for effective software exploration, in: Proceedings of IEEE Symposium on Information Visualization (InfoVis '97), IEEE Computer Society, 1997, pp. 38–48.

[45] Anne Treisman, Preattentive processing in vision, Computer Vision, Graphics, and Image Processing 31 (2) (1985) 156–177.

[46] Edward R. Tufte, The Visual Display of Quantitative Information, second ed., Graphics Press, 2001.

[47] Lucian Voinea, Alex Telea, Jarke J. van Wijk, CVSscan: visualization of code evolution, in: Proceedings of 2005 ACM Symposium on Software Visualization (Softviz 2005), St. Louis, Missouri, USA, May 2005, pp. 47–56.

[48] Colin Ware, Information Visualization: Perception for Design, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[49] Theo Wiggerts, Using clustering algorithms in legacy systems remodularization, in: Ira Baxter, Alex Quilici, Chris Verhoef (Eds.), Proceedings of WCRE '97 (4th Working Conference on Reverse Engineering), IEEE Computer Society Press, 1997, pp. 33–43.

[50] Xinrong Xie, Denys Poshyvanyk, Andrian Marcus, Visualization of CVS repository information, in: WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering, IEEE Computer Society, 2006, pp. 231–242.