The Package Blueprint: visually analyzing and quantifying package dependencies

Hani Abdeen, Stéphane Ducasse, Damien Pollet, Ilham Alloui, Jean-Rémy Falleri

The Package Blueprint: visually analyzing and quantifying package dependencies☆

Hani Abdeen[a], Stéphane Ducasse[a,*], Damien Pollet[a], Ilham Alloui[1], Jean-Rémy Falleri[c]

[a]*RMoD INRIA Lille Nord Europe, LIFL, CNRS UMR 8022*
[b]*LISTIC Université de Savoie, France*
[c]*Univ. Bordeaux, LaBRI, UMR 5800, F-33400 Talence, France*

Abstract

Large object-oriented applications are structured over many packages. Packages are important but complex structural entities that are difficult to understand since they act as containers of classes, which can have many dependencies with other classes spread over multiple packages. However to be able to take decisions (*e.g.*, refactoring and/or assessment decisions), maintainers face the challenges of managing (sorting, grouping) the massive amount of dependencies between classes spread over multiple packages. To help maintainers, there is a need for *at the same time* understanding, and quantifying, dependencies between classes as well as understanding how packages as containers of such classes depend on each other.
In this paper, we present a visualization, named Package Blueprint, that reveals in detail package internal structure, as well as the dependencies between an observed package and its neighbors, at both package and class levels. Package Blueprint aims at assisting maintainers in understanding package structure and dependencies, in particular when they focus on few packages and want to take refactoring decisions and/or to assess the structure of those packages. A package blueprint is a space filling matrix-based visualization, using two placement strategies that are enclosure and adjacency. Package blueprint is structured around the notion of *surfaces* that group classes and their dependencies by their packages (*i.e.*, enclosure placement); whilst surfaces are placed next to their parent node which is the package under-analysis (*i.e.*, adjacency placement). We present two views: one stressing how an observed package depends upon the rest of the system and another stressing how the system depends upon that package.
To evaluate the contribution of package blueprint for understanding packages we performed an exploratory user study comparing package blueprint with an advanced IDE. The results shows that users of package blueprint are faster in analyzing and assessing package structure. The results are proved statically significant and they show that package blueprint considerably improve the experience of standard browser users.

Keywords: Software engineering, Software comprehension, Software maintenance, Software visualisation

1. Introduction

To cope with the complexity of large object-oriented software, applications are structured in packages. Packages are units of reuse and deployment, and they play different development roles (*e.g.*, class containers, architectural elements...) [1]. Well structured packages ease software system evolution by supporting the replacement of its parts without impacting the complete system [2, 3]. But as systems inevitably become more complex, their package structure often drifts [4]. As a consequence, it is important to maintain the structure of packages [5, 6].

In literature, there exist many approaches that support automated software remodularization [5, 7, 8]. Despite their success in producing alternative structures, the new modularizations they propose remain difficult to understand and assess. It is thus important to *assist maintainers in understanding detailled dependencies between classes, as well as understanding how packages as containers of such classes depend on each other*.

Before going further, we define *a package as an entity containing classes*. It may or may not have package import declarations and may be or may not be associated with a namespace. This definition captures the situation of a large number of object-oriented languages. Classes can depend on each other via different kinds of dependencies. By definition, there is a direct dependency from a class A to another one B if A *references* B (*i.e.*, A uses B as type of some variable or A methods invoke methods of B).We refer to references that goes from A to B as *outgoing* references of A and/or *incoming* references to B. In the same vein, we say that A is the *referencing* class and B is the *referenced* one.

Inter-classes dependencies can exist among classes belonging to the same package (*internal* package dependencies) or belonging to different packages (*external* package dependencies). Hence, we say that packages depend on each others via those external package dependencies. Thus, the organization

☆This article makes heavy use of colors. To better understand the ideas presented in this paper, please read a colored version of it.

*Corresponding author
Email addresses: hani.abdeen@inria.fr (Hani Abdeen), stephane.ducasse@inria.fr (Stéphane Ducasse), damien.pollet@inria.fr (Damien Pollet), ilham.alloui@univ-savoie.fr (Ilham Alloui), falleri@labri.fr (Jean-Rémy Falleri)

of packages, with the classes they contain and the dependencies among the classes, can be thought of as a compound graph with one type of dependencies among the classes, namely references. Roughly speaking, packages can thus be thought of as *clients* and/or *providers* of classes [3]. We say that there is a client-provider relationship between two packages P1 and P2, if there is at least a class A in P1 that depends on at least another class B in P2.

Several articles proposed visualizations that provide information on packages and their dependencies, by visualizing metrics values, package connectivity and cycles, package evolution or the common usage of package classes (*e.g.*, [9–17]). Relevant body of existing work on software understanding is based on visualization approaches [18, 19], in particular, on node-link visualizations [20–23]. On one hand, some researchers explored matrix-based representation of graphs [24, 25] or of software [26] and its evolution [27]. On another hand, important progress have been made to support navigation over large graphs and to propose scalable and sophisticated node-link visualizations for visualizing the connectivity graph of software entities [23, 28–30]. However, while these approaches are valuable, the proposed visualizations are still not adapted to represent the dependencies induced by classes *while being grouped per package communication*, nor do they show the *importance and impact* of classes (and/or packages) according to their dependencies with other classes (and/or packages). These visualizations fall short of revealing, *at the same time, package structure, internal dependencies, their external dependencies at both class and package levels as well as giving a first estimate of the strength of such dependencies*.

Paper Contribution

To address the limitations, outlined above, we propose a space filling matrix-based visualization, namely Package Blueprint, using two placement strategies that are *enclosure* and *adjacency*[17]. A package blueprint reveals the package structure, its internal dependencies (*i.e.*, package cohesiveness), as well as the distribution of its external dependencies with classes outside it, while being grouped by their packages. Package blueprint is structured around the notion of *surfaces*. A package blueprint' surface groups classes and their dependencies by their package (*i.e.*, enclosure placement); whilst the package blueprint' surfaces are placed next to their parent node which is the package under-analysis (*i.e.*, adjacency placement). However, package blueprint is a hybrid visualization because dependencies are represented as nodes grouped into their corresponding surfaces. We present two specific views that have the same concepts but one stressing the references made *from* a package to the other packages (*outgoing references*), and another stressing the references made *to* a package by the rest of the system (*incoming references*).

We show the usefulness of package blueprint to analyze and understand package structure, and to address the limitations outlined above of existing software visualizations, through several examples taken from real-world software systems. To evaluate the contribution of package blueprint for understanding packages we performed a comparative study comparing the use of package blueprint visualization versus an advanced IDE. The results shows that, overall, users of package blueprint are faster. Package blueprint users performed very well when the task involves a complete identification of all dependencies for a package or a class. Tasks involving more judgment, like identifying important classes or explaining package design are less impacted and standard browser users performed equally well, as they spent more time reading the code, which could give them a better understanding of some inner workings of the system, not just of dependencies between packages and classes. The results promise the usefulness of package blueprint visualization as a complementary tool to standard code browsers, they show that package blueprint could improve the experience of standard browser users.

The work presented in this article extends our previous paper [31] in the following points: (a) visualization improvement based on the feedback and conclusion of the exploratory study, (b) additional and complementary visualization for incoming references (besides that of outgoing references), (c) a detailed presentation of a case study and (d) a separate comparative study analyzing the use of package blueprint versus standard IDE support to perform package maintenance tasks.

Section 2 presents the challenges in supporting package understanding, and summarize the properties expected for effective visualizations. Section 3 presents how related works are positioning themselves in this context. Section 4 presents the structuring principles of a package blueprint, which are then declined to support an outgoing reference and an incoming reference views. Section 5 presents the distinct views of package blueprint at work and detail the different information that can be extracted from package blueprint. The next section presents a comparative study of package blueprint vs. IDE use. In Section 7, we discuss our visualizations and position them w.r.t. related work before concluding.

2. Challenges in Understanding Packages

Packages are complex entities: as they provide another abstraction layer they are not at the same conceptual level than classes. Packages reflect several organizations: they are units of code deployment or units of code ownership; they can also encode team structure, architecture and stratification [1]. Good packages should have only a few clear dependencies to other packages [32–34]. However, as mentioned by R. Martin, importing a class equals importing its complete package and its dependencies recursively [1], therefore importing two classes from the same package is really different from importing them from two different packages. This is a key point, because besides creating references in the system, classes should also be assessed in the context of deployment, within their packages. In practice, some packages may have either a lot of references to other packages or only a couple of them. In summary, there is no canonical form for packages but a diversity of configurations that the maintainers should understand and assess.

In this section, we stress some important points related to package understanding. We sketch some typical maintenance

tasks and provide an overview of data that can help to characterize packages.

2.1. High-Level Maintenance Package Tasks

When maintaining an application, developers face tasks specific to packages. Here we present some key activities.

Package loading/unloading. Knowing whether a package imports too many other packages is one of the first clues to a maintainer trying to reduce code bloat. Typical questions are "can we easily load this package?" and "what are the other ones that should be loaded as well?". Similarly, "can we unload this package?" is an important question in presence of class bloat. For example, this happens for the Java default runtime which often does not fit the memory constraints of small devices.

Repackaging. The corollary to the previous task is to know whether it is possible to repackage a set of classes to minimize external dependencies. In such a context, understanding which classes are the most used internally or the most referenced by other packages is key. But, to help understand the package relations, this information on class references should be presented at the granularity level of a package. Repackaging involves identifying misplaced classes in a package and unnecessary dependencies to other packages. If a class has a lot more references to different packages compared to other classes of the package, it may be an indication that the class should be moved out. Similarly, understanding the cohesion of a package helps maintainers know whether the package can be split.

Architecture violations. Applications often follow architectural patterns, *e.g.*, a layered structure. When developers integrate changes, a key question they face is assessing the impact of each change on the system architecture. For instance, a change should not make a core package depend on its extensions, nor make a database layer depend on UI layer.

2.2. Quantitative and Qualitative Characteristics

To support the understanding of dependencies between packages and classes, it is important to be able to characterize some key aspects. Based on our experience, we identify two main aspects: the general communication size and package cohesion/couplings. Even thought there is no accepted metrics for packages, getting a first impression is important.

Communication size. What is the general size of a package in terms of classes, inheritance definitions, internal and external class references, provider and client packages? This is useful to answer questions like "do we have only a few classes communicating with the rest of the system?"

Cohesion and coupling. Transforming or evolving an application follows natural boundaries defined by coupling and cohesion [33, 35]. One important challenge is then to understand how the cohesion of a given package relates to its coupling. Maintaining package structure sometimes involves deciding whether to split or replace a package; for this, it is important to identify misplaced classes, classes that are tightly coupled within the package and those that are tightly coupled with classes of other package.

2.3. Questions characterizing class/package relations

We conclude that to understand *an observed package* and assist maintainers to take decisions about the package structure, there is a need for capturing the following information phrased as questions that a package maintainer asks

Size: What is the size and complexity of the observed package in terms of number of its classes and number of its provider/client packages and/or classes?

Cohesion, Coupling and Repackaging: How the observed package classes depend on each others? How package classes depend on classes of other packages (and/or how classes of other packages depend upon them)? What classes of the observed package have more/less references than other classes inside it? Do those classes participate to the package cohesiveness or cause high coupling with other packages?

Loading/Unloading and Architecture Violation: What are the provider/client packages to the observed package? Which ones belong (or do not) to the scope of the application under-analysis? What classes of its provider packages can have a direct impact on it and to which extent? Which client (or provider) packages to the observed package are more/less important than others in terms of communication size and number of classes involved in the communication?

3. Related Work

In this section we overview relevant generic graph visualizations that could be mapped to the problem of understanding package structure and dependencies, then we look at relevant visualizations that are specific to the problem of understanding packages. We briefly discuss their limitations for revealing package structure and dependencies at both package and class levels.

3.1. Generic Graph Visualization

There is an extensive body of work related to the visualization of graphs [17]. In this section we discuss relevant generic graph visualizations. We briefly outline their limitations for characterizing package structure and quantifying the dependencies within and among packages.

Henry *et al.* propose NodeTrix, a hybrid graph visualization mixing matrices for local details and node–link for the overall structure [24]. It is tailored for the globally sparse but locally

dense graphs of social networks. One advantage is that it can present both incoming and outgoing references in a single view. However, it is not clear how NodeTrix can be mapped to show the distribution of dependencies with, at the same time, two types of node (packages and classes). One could represent each package with a matrix which shows internal references between classes, and links connecting classes for external references between packages. This is intuitive but defeats the purpose of NodeTrix: as shown in this paper, some packages are internally sparse, and some have dense external dependencies spread over large number of packages. Or one could follow NodeTrix principles and only focus on the topology of the class dependency graph. Then the package structure is lost.

Abello and van Ham present Matrix Zoom, a hybrid visualization for hierarchical sets of data, using matrices only for fine-grained details [25]. The user navigates into the hierarchy and can get some details on the focused subset in the matrix. We believe both approaches have different requirements which could be complementary. Matrix Zoom offers a single scalable view of the whole dataset. This implies for example that all packages of the studied system are displayed at once, even if few packages have incoming/outgoing dependencies, resulting in lost space in the matrix. However, we believe Marix Zoom can not be adapted to display the distribution of dependencies at both class and package levels.

Holten proposed Hierarchical Edges Bundles (HEB), an approach to improve the scalability of large hierarchical graph visualizations. Edges (links) are bundled together based on hierarchical information; color mapping is used to represent the edge types while saturation mapping and blending are used to represent the edge directions [23, 30]. The technique has been applied to see the communication between classes grouped by packages in large systems and the bundling of edges produces less cluttered display. However, HEB is not adapted to show at the same time the internal and external package dependencies. Moreover, due to occurrences of many link crossings between nodes that are positioned in circle in HEB, it is not easy to identify all the packages (and classes) involved in a given communication and to identify patterns of inter packages dependencies. This limit has been also reported by Henry et al [24].

3.2. Package Visualization

In literature, there are several articles provide or visualize information on software files, classes and/or packages. Many of these approaches address software co-change, looking at coupling from a temporal perspective [19, 27, 36–39], whereas in this paper we focus on the static structure of dependencies.

Chuah and Eick use rich glyphs to characterize software artifacts and their evolution (number of bugs, number of deleted lines, kind of language...) [9]. In particular, the time wheel exploits preattentive processing, and the infobug presents many different data sources in a compact way. D'Ambros *et al.* propose an evolution radar to understand the package coupling based on their evolution [10]. The radar view is effective at identifying outliers but does not detail the structure. In the same vein, to present the evolution of properties over time, Voinea *et al.* [27] propose a view of CVS repositories where each version is represented by a column and colors span over multiple versions to represent a given property over time. Ducasse *et al.* present Butterfly [11], a radar-based visualization that can be used to present the values of several package metrics (*e.g.*, the number of classes, the number of incoming and outcoming dependencies for a package), but only gives a high-level abstractness of package structure. In a similar approach, Pinzger *et al.* use Kiviat diagrams to present the evolution of package metrics [13]. Churcher *et al.* used 3D to visualize class cohesion [21]. It is not clear that their technique can help for large classes and can be applied to packages. In particular the structure of communication around communicating packages seems to be difficult to map to their approach. Using 3D on one hand can help having more information available but on the other hand it adds new problems like occlusion. In a similar approach, Andrian Marcus *et al.* [26] propose a matrix-based representation of files. Each dot represents a line and its color conveys one semantics information (if statement for example). Then they propose a 3D version of the matrix-based structure. The idea behind the matrix-based presentation is to be able to offer a compact representation of code. While those visualizations are valuable, they are not adapted to present the static structure of package dependencies.

Storey *et al.* [14] offer multiple top-down views of an application, but these views do not scale very well with the number of dependencies. Particularly if both classes and packages are displayed, then due to occurrences of many link crossings between nodes it is very difficult to identify packages and/or classes involved in a given communication.

Sangal *et al.* [40] adapt the dependency structure matrix from the domain of process management to analyze software architecture. Lungu *et al.* guide exploration of nested packages based on patterns in the package nesting and in the dependencies between packages [12]; their work is integrated in Softwarenaut and adapted to system discovery. Abdeen *et al.* provide the Package Fingerprint visualization [16, 41]. They focus on the contextual cohesion of a package, the co-use and co-usage of package classes. However, although the visualizations of Abdeen *et al.*, Sangal *et al.* and Lungu *et al.* are good for fast overview on inter-package connectivity, package nesting and software architecture, they can not be adapted for visualizing internal package dependencies and the distribution of dependencies at class level.

Müller *et al.* [29] propose the Subversion Statistics Sifter visualization for exploring the structure and evolution of Subversion repositories, with respect to both developer activity and source code changes. Their visualization is designed as a biological tree, in which the repository under-analysis represent the root node. The tree branches represent the path folders contained in the repository.However, it is not clear how Subversion Statistics Sifter visualization can be mapped to show the distribution of package dependencies with at class level. One could represent the package under-analysis as the root node, whilst referenced packages and their classes are respectively represented by inner nodes and tree leaves. Then the package structure (classes defined in it and its internal dependencies) is lost.

Moreover, using this map, one can not capture the communication size between the package and its provider packages, nor identify the package classes involved into a communication.

Those approaches, while valuable, fall short of providing a fine-grained view of packages that would help in capturing information that we outlined in Section 2.3.

4. The Package Blueprint for Understanding Packages

This section presents the principles and layouts of our visualization, *Package Blueprint*, that we propose to assist maintainers in understanding package structure and characteristics. Before going in the details of the package blueprint views we present the design constraints for our visualization.

4.1. Design Constraints for Visualization

In addition to the challenges raised by package understanding (Section 2), the visualization itself raises challenges. Several works identified the characteristics that an efficient visualization should hold [42–44]. We stress that our visualization should take into account the following properties:

Good mapping of phenomena of interest. We expect the visualization to highlight the general tendency of a package in terms of its size, internal and external dependencies and client/provider packages (and classes). Furthermore, we expect the visualization to reveal package structure and the distribution of its internal and external dependencies at class level. Precisely, we expect the visualization to spot-out classes, dependencies or packages that stand out in the context of an observed package. We stress that we do not target a visualization to present all the packages of a large system in one screenshot. It is important to notice that we target a visualization that reveals, *in details*, the whole structure of *an* observed package, along with its detailed relations with its provider (and/or client) packages.

Good readability and low visual complexity. Although the visualization should offer a lot of information, it should not be complex to analyze. A primary concern for graph visualization is the choice between node–link layout and matrix-based representation. Ghoniem *et al.* [28, 45] performed a comparison study on the readability of node–link and matrix-based representations. According to the study, Node–link representations are more intuitive and compact than matrix-based ones; and are suitable for small and sparse graphs. However, matrix-based representations, unlike node-link ones, do not have the problems of edge crossing and node overlappling. As a consequence, matrix-based representations are more suitable for detailed analysis of dense graphs. Nevertheless, they may suffer from scalability compared to node-link representations.

Since we target a visualization to reveal *in details* and *clearly* the dependencies distribution for an observed package, and this independently of the density of package dependencies, we believe matrix-based representation is more suitable for our visualization than node-link one. Keep in mind that we do not target a visualization to present all the packages of a large system in one screen-shot. Rather, we target a visualization to assist maintainers in their refactoring and assessment decisions when working on a subsystem. Maintainers face such decisions when they are in contact to an observed package or a group of few packages (subsystem). This considerably lowers the stress on the scalability problem of matrix-based representation of our approach.

Moreover, to reduce visual complexity, we target a visualization that reuses the same visual conventions in all its parts and views. From that perspective, by design this visualization does not support semantic-zooming, but geometric-zooming.

Preattentive Visualization. Despite the quantity of information to be offered by our visualization, we expect a visualization to provide also a first impression of a package and its context, in one glance. Therefore, we would like to exploit the gestalt principles of visualization and preattentive processing[1] as much as possible to help spotting important information [44, 46–48]. The preattentive visual features that are used in package blueprints are “length”, “width”, “filed”, “intensity” and “color” (Sections 5.2 and 5.3 summarize how these features are employed).

Simple navigation and interaction. The visualization should enable maintainers to interact with it, to select and mark classes and/or packages. An important point is to be able to identify a class or the communication between packages across multiple packages. It should enable maintainers to collect more information from several packages.

4.2. Package Blueprint basic principles

A package blueprint shows the observed package as a rectangle, subdivided into parts representing the package’ “contact areas”, that we name *surfaces* (see Figure 1):

Surface. Each surface conceptually represents the distribution of the observed package dependencies, that are either internal to the observed package itself (the *“head”* surface) or external pointing to (or coming from) another package (a *“body”* surface).

Head surface. So called the “head”, it is reserved to represent the size of the observed package and the dependencies among its classes. Thus, the head of a package blueprint represents the communication size between the classes of the observed package itself (*i.e.*, the package cohesiveness). The head surface is more or less tall, according to the number of classes involved in the observed package.

[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 length, width, size, shape, filed, curvature, intensity, hue, orientation, motion, and depth of field. However, combining them can destroy their preattentive power (in a context of filled squares and empty circles, a filled circle is usually not detected preattentively). Some of the features are not adapted to our needs. For example, we do not consider motion as applicable.

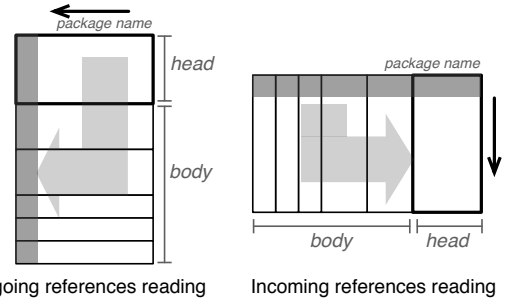

Figure 1: To distinguish between the two main semantics (incoming and outgoing) we use the main orientation of the package blueprint. In both case the important details are still read first; in the incoming view, the references are made by the external classes, at the top, to the internal classes below them.

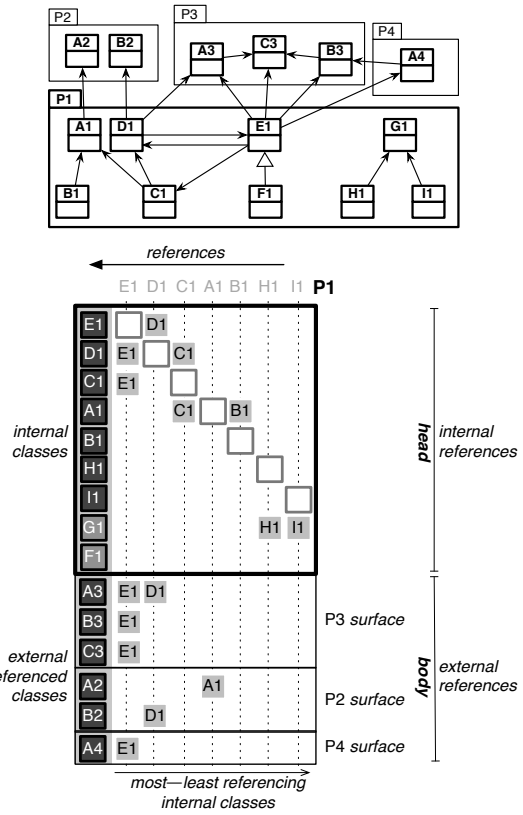

Figure 2: Detailed layout of the outgoing references blueprint for P1.

Body surfaces. For other surfaces, different than the "head", that we name the "body" surfaces, each one conceptually represents the dependencies between the observed package and another one. Hence, all body surfaces (the body) of a package blueprint represent(s) the communication size between the observed package and the rest of its system (*i.e.*, the package coupling); whilst a body surface represents the communication size between the observed package and another one. Each body surface is more or less tall, according to the strength of the communications between the observed package and the package represented by that surface.

4.3. Outgoing Reference Blueprints

Package blueprint is a matrix-based representation of the package dependencies. An outgoing references blueprint, as demonstrated in Figure 2, is organized as follows:

Rows. They represent the *package classes*, *and* other classes that are referenced by the package classes, namely *external referenced classes*. These classes are placed in the first, left-most, column of the matrix and we refer to them by ***classes nodes***.

Columns. They represent the ***referencing classes*** packaged within the observed package. By definition, they are the package classes that point outgoing references to other classes, either inside or outside the observed package. Each referencing class has its reserved column to the right of the left-most one.

Cells. A *filled* cell represents a reference from the column class (referencing class) to the row class (referenced class). Hence, we refer to filled cells by ***reference nodes***.

Enclosures (Surfaces). They group the matrix rows (*i.e.*, represented classes) by their packages. Recall that the first surface, the head, groups the classes of the observed package itself, and thus it represents the observed package and its internal references. Hence, reference nodes within the package blueprint head represent the distribution of internal references to the observed package, whilst those in the package blueprint body represent the distribution of its external references.

In Figure 2, on one hand the head of the blueprint represents the internal structure of the observed package P1. For example, we see that the class E1 internally refers tothe classes D1 and C1. We can also see that, the classes B1, H1, I1 and F1 are not referenced inside P1 (no filled cells in their rows); however, H1 and I1 reference G1 (see the filled cells in the row of G1). On another hand, the surface corresponding to P3 groups the P3' classes that are referenced by the observed package P1. Those external referenced classes in P3 are: A3, B3 and C3. The figure shows that the surface of P3 is taller than the surfaces of P2 and P4, since the observed package P1 reference more classes in P3 than in P2 or P4.

The body surface with their rows (*i.e.*, external referenced classes), and the package blueprint columns (*i.e.*, referencing classes), are ordered according to the number of their associated references. To convey quantitative information about the number of references represented in a reference node (*i.e.*, in a filled cell), we use color intensity for reference nodes.

Order of Columns. The referencing classes of the observed package, which represent the package blueprint columns (and the head rows), are ordered horizontally from left to right (and vertically from top to down), according to the number of classes they refer to. Hence classes referencing the most occupy the nearest columns from the left-most column.

Figure 2 shows that columns of referencing classes are ordered as follows: E1 (references 6 classes); D1 (references 3 classes); C1 (references 2 classes); A1, B1, H1 and I1 (each of them references only one class). Within the head surface, bordered squares highlight the top-left to bottom-right diagonal to help the users clearly see the symmetry between the horizontal and vertical orderings.

To show the size and cohesiveness of the obeserved package,

internal classes that do not make any reference still occupy a row at the bottom of the head (*e.g.*, G1 and F1). Both referencing and not referencing classes are shown together with all internal references among them (*e.g.*, the not referencing G1 is referenced by H1 and I1).

Order of Surfaces and Rows. We apply the same ordering principle to the vertical ordering, both for body surfaces (*i.e.*, provider packages) and for rows within a body surface (*i.e.*, external referenced classes). For example, Figure 2, the surface P3 in higher than the P4 because the class of the package P1 refer more classes in P3 than in P4. For provider packages, we position body surfaces that contain the most referenced classes the highest. For referenced classes within a body surface, we order referenced classes from the most referenced at the top, to the least referenced at the bottom.

Color-Intensity for References Nodes. Color intensity assigned to a reference node within a surface conveys the number of references that go from the referencing class in the cell's column to the provider package represented by the cell's surface. The darker the cell the more references it conveys. Both intensity and horizontal position represent the number of references, but position is computed relatively to the whole package blueprint, while intensity is relative to each surface.

Colors for Classes Nodes. To distinguish different categories of classes, we use colors for classes nodes. In the package blueprint head, we distinguish two categories of the observed package classes: *not referencing* classes and referencing ones. Not referencing classes are colored lighter than referencing ones: *e.g.*, in P1' package blueprint, Figure 2, the classes nodes of G1 and F1 are lighter than those of E1 . . . I1. In the package blueprint body, we distinguish external referenced classes which are *not within the scope of the application under study* by coloring them cyan. Those classes can belong to a third-party framework or to the base system. Otherwise, if a referenced class is part of the analysis scope, we color it gray like in the head.
However, to ease interactions with blueprints, users may use other colors to select and/or mark classes (and/or packages). The interaction mechanisms with blueprints are explained later.

Note. In diagrams explaining the package blueprint principles, we use the name of the class inside its assigned nodes to facilitate the understanding of package blueprint layout; but in the actual visualization, nodes are just colored boxes having the same size. However, users can get more information about nodes/surfaces (*e.g.*, classes/packages names) by interacting with package blueprint, as we will see in next sections.

Finally, we use package blueprint to display incoming references in addition to outgoing references. To display incoming references, the layout works the same, but we rotate it to easily distinguish an incoming from an outgoing blueprint —as shown in Figure 1 and detailed in Section 4.4.

4.4. Incoming Reference Blueprints

These key principles of a package blueprint are realized slightly differently according to the direction of references (outgoing or incoming). To explore incoming references, we use a view similar to the outgoing reference blueprint. In a first version of the package blueprint, the two views were visually too close, and it was difficult to distinguish them in a glance. To avoid confusion, we needed a layout that was really distinct while sharing the same visual effect. Therefore, to visually distinguish blueprints for incoming reference from the ones for outgoing references, we rotate the layout as shown in Figure 1.

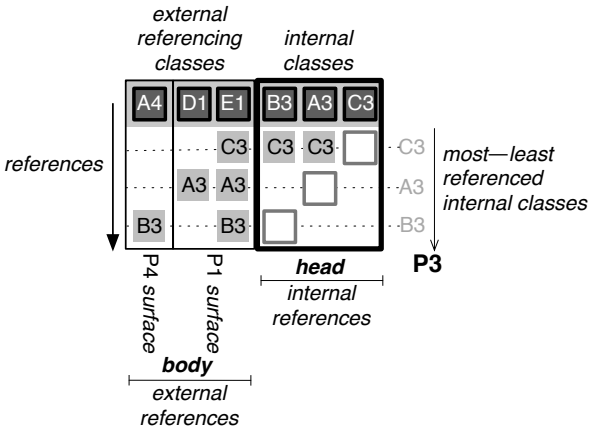

Figure 3: Detailed layout of the incoming reference blueprint (P3, Figure 2).

In an incoming reference blueprint, surfaces are juxtaposed horizontally from right to left: the rightmost surface is the head, and we place surfaces for client packages to the left, ordered by decreasing number of referencing classes. As the direction of references is changed, in an incoming reference blueprint, the columns are reserved for *external referencing classes* which are placed in the top-most row, whilst other rows are reserved for the package *referenced classes*. Similarly to the ordering of an outgoing reference blueprint, in an incoming reference blueprint the package referenced classes (*i.e.*, rows) are sorted top-to-bottom by number of references.

Figure 3 shows the incoming reference blueprint of P3 in the configuration displayed in Figure 2. The blueprint body has two surfaces: one for P1 and one for P4, because these packages are the two clients of P3. P1 surface is at the right of P4 surface, because it involves more referencing classes (E1 and D1, compared to just A4 for P4). C3 is given the topmost row for internal classes, since it is the most referenced class within P3 (from A3, B3 and E1), while A3 and B3 are both referenced from only two classes (A3 from E1 and D1, and B3 from E1 and A4).

5. Analyzing Package Structure With Package Blueprint

Having presented the package blueprint layout, now we can use package blueprints to analyze package structure and dependencies in real contexts. In this section we show the usefulness of package blueprint for analyzing package structure and dependencies within the context of a complete application taken from real object-oriented systems. Relevant questions are for instance the questions outlined in Section 2.3. For this purpose, we use the outgoing and incoming references blueprints for analyzing the Squeak Network subsystem, which contains

178 classes and 18 packages, making up a library and a set of applications such as a complete mail reader. However, before going further in using blueprints for package analysis, we present the interaction mechanisms that package blueprint offers for its users.

5.1. *Interacting with package blueprint*

Users of package blueprint can interact with it in three main ways:

- **Select and Mark classes and/or packages.** Users can select and/or mark a class or a surface (the package represented by that surface). When the user selects a class, the class nodes representing that class and references nodes associated to it gets colored in red. The same happens if the user mark a class with a color. Similarly, the user can also select a surface (or mark it with a given a color). Selecting/marking a surface means selecting/marking the package represented by that surface, so all surfaces in relation with that package are then selected/marked (see Figure 4).

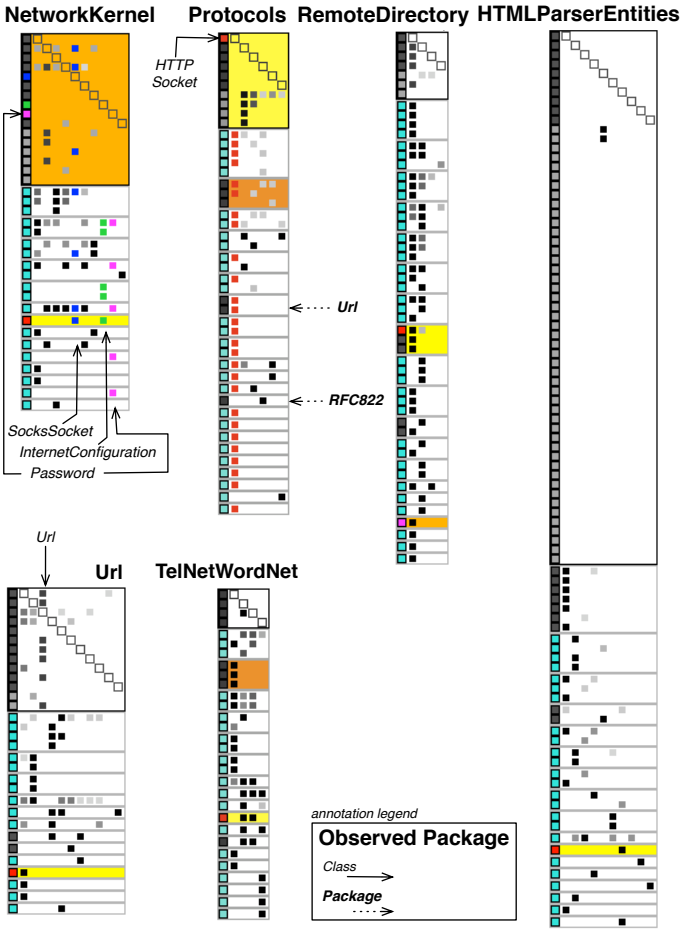

Figure 4: Outgoing Reference Package Blueprint Analysis.

Outgoing reference blueprints of 6 packages of the Network application (annotated screenshot - annotations are in italic). Those 6 packages contain 106 classes (class nodes in the blueprint heads), which expose 326 outgoing references to other classes, intra- and inter-package (reference nodes); causing 104 inter-package relationships (body surfaces). Color marks: surfaces of Kernel package in *orange*; surfaces of Protocols package in *yellow*; HTTPSocket class in *red*; SocksSocket class in *blue*; InternetConfiguration class in *green*; Password class in *fuchsia*. Recall that *cyan* color "" always annotates external referenced classes that do not belong to the application under-analysis.

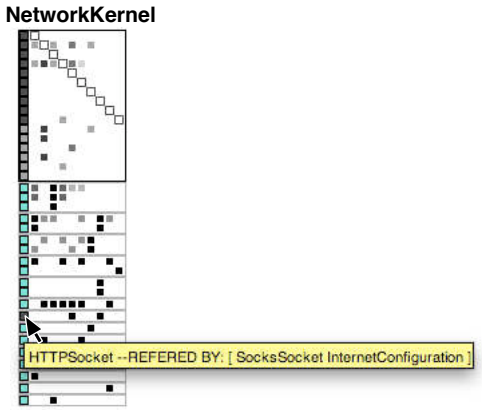

Figure 5: Use fly-by-help to display class/package information.

The outgoing reference blueprint of NetworkKernel package. The mouse is pointing to a class-node representing HTTPSocket class and the fly-by-help shows, in addition to the class name, the name of classes in NetworkKernel that refer to HTTPSocket.

- **Display class/package information.** A user can get information via a fly-by-help on a class or a package. For instance, a fly-by-help displays the class/package name when the mouse is pointing to a class-node/surface in relation with that class/package (see Figure 5). Finally user can jump to the code in a default code browser.

- **Filtering.** To support visual information seeking mantra, we implemented different visual filters that users can use on demand. For instance, users can filter visual information as follows. (a) Display only references that are in the scope of the studied application and/or in the scope of specific group of packages. For example, package blueprints in Figure 4 display all outgoing references, while those in Figure 6 are filtered to display only references in the scope of the analyzed application Network. (b) Users can also filter a package blueprint to display only the package classes that are involved in communications with other classes. In outgoing references blueprint, this filter will result in displaying only classes that have outgoing references to other classes in the package blueprint head. This can reduce the head size of package blueprints, however, users will lose information about package size and/or cohesiveness. In the same way, (c) users may focus only on external package dependencies, so that package blueprints will be displayed without their head surfaces.

5.2. *Outgoing Reference Blueprint Analysis*

Figure 4 shows the outgoing references blueprints of 6 packages of the Network application. We selected those packages

from Network application since they define and refer to a central class in Network, named HTTPSocket (colored in red in Figure 4). In this section we demonstrate how the outgoing references blueprint help in analyzing those packages and answering important questions about their structure and dependencies, such as the questions we outlined in Section 2.3.

Large packages. In Figure 4 we can easily spot tall blueprints, which are HTMLParserEntities, RemoteDictionary and Protocols. Those packages are 'large' in terms of contained classes and external referenced classes. Looking a bit closer to those blueprints, on first hand, we see that among those packages the HTMLParserEntities package is the largest in terms of contained classes in it –since it has the tallest head. On second hand, we see that RemoteDictionary and Protocols are large in terms of external referenced classes –since each of them has a tall body with relatively small/short head. Hence, these two packages are characterized as tightly coupled to the outside: *i.e.*, as demonstrated by the number of surfaces in their body, those packages reference many other packages.

Small packages, but with complex implementations. By looking to the head size of blueprints in Figure 4, we find that the package TelNetWordNet is the smallest one in terms of contained classes: it has the shortest head since it contains only 4 classes. However, giving the body' height of TelNetWordNet blueprint and the number of surfaces in its body, we deduce that the classes in TelNetWordNet most probably provide complex implementations: few classes (only four classes) reference a relatively large number of classes that spread over many packages. The same can be deduced for RemoteDictionary which contains a bit more classes than TelNetWordNet (see the head height), but only four classes in RemoteDictionary (see the head width) reference a relatively large number of classes spread over many packages. As a consequence, loading these small packages require loading all those referenced packages. By comparing the blueprints of TelNetWordNet and RemoteDictionary to those of URL and Kernel, we can easily see that the latter packages contain more classes than the former ones (look to the head size). However, both packages, URL and Kernel, reference less external classes than TelNetWordNet and RemoteDictionary (see the body height).

Cohesive packages. In addition we see in Figure 4 that Kernel and Url packages contain classes that are tightly inter-referenced – since there are a lot of reference nodes (*i.e.*, filled cells) within the head of Kernel and Url package blueprints. However, these blueprints of Kernel and Url packages show also that there are more reference nodes inside the body surfaces than inside the head of blueprints. This indicates that the classes of those packages have more interaction with classes of other packages than inside their packages. This conveys a first impression of the package cohesion even if it is not really precise [33, 35]. Similarly, we can see that RemoteDirectory and Protocols packages are less cohesive than Kernel and Url packages. It is worth to note that in Protocols package, all internally referenced classes are classes that do not reference other classes –since all reference nodes within the head of the Protocols blueprint are under the head diagonal.

Sparse (non-cohesive) packages. Figure 4 shows that HTMLParserEntities and TelNetWordNet packages are not cohesive from the point of view of inter-class references – since the heads of HTMLParserEntities and TelNetWordNet blueprints contain respectively only two and one reference nodes. However, HTMLParserEntities package seems much sparser (non-cohesive) than TelNetWordNet package due to the fact that the former contains much more classes than the latter (see the head length). Hence the density of reference nodes within the head of HTMLParserEntities blueprint seems much low than that within the head of TelNetWordNet blueprint. It is worth to note the blueprints of HTMLParserEntities and TelNetWordNet show that the head of the former seems as a tall rectangle (*i.e.*, its height is much greater than its width), while the head of the later seems as a square. This tells us that only few classes in HTMLParserEntities have outgoing references to other classes, while all classes in TelNetWordNet reference other classes.

Most referencing classes, internally and externally. Glancing at the blueprint of Kernel package, in Figure 4, and focusing on reference nodes, we can spot the class making the most internal references: it is the one represented by the second column. In fact, there are more reference nodes (four) in this column within the head than in other columns. Moreover, those nodes are the darkest reference nodes within the head (we can see this clearly in Figure 5). Using the fly-by-help, we learn the class name, which is Socket. This means that Socket is referencing 4 classes within the analyzed package Kernel and it is the class which does the biggest number of references within Kernel.

We also learn that the class OldSimpleClientSocket, represented by the first column in the Kernel blueprint, makes the most external references – the class column includes, within the blueprint body, nine reference nodes that are distributed over seven distinct body surfaces. This means that this class references nine classes into seven packages. However OldSimpleClientSocket references only two classes within Kernel as shown in the blueprint head.

In the same way, we can see that the class Url, within Url package, references almost all classes inside the Url package, but it does not reference any class outside the Url package (follow the column of Url class, which is third column in the Url package blueprint).

Hub classes. The blueprint of Protocols package shows that that HTTPSocket class (colored in red) is a central class in Protocols as it references most of the external classes referenced displayed in the package blueprint (follow the column of HTTPSocket). We can deduce the same thing for the class ServerDirectory in RemoteDirectory package. Furthermore, by following the surfaces in relation with Protocols, whose are colored in yellow, we can easily see that all its referencing packages reference the class HTTPSocket. Only RemoteDirectory references, in addition to HTTPSocket, two classes in Protocols. Note that HTTPSocket has neither incoming references nor outgoing references inside its package Protocols – since in the head of Protocols blueprint the column and the row of HTTPSocket contain no reference nodes.

Most references point to outside the analyzed application.

All the blueprints in Figure 4 show that most of the external referenced classes are cyan, which means that they are not part of the analyzed application, Network subsystem. Indeed those referenced classes belong to the core libraries (*e.g.*, CollectionsStreams, CollectionsArrayed and CollectionsStrings) on top of which Network library is developed.

Architecture violation with misplaced references. What is striking in the blueprint of Kernel package is that all, except one, of the external referenced classes are outside the application (HTTPSocket which is highlighted in red and defined in Protocols package). Since the package is named Kernel, it is strange that it references other classes from the same application Network, and especially to only one. This is clearly a layering bug.

Potentially misplaced classes and refactoring candidates. Again in Kernel package, we found that the class Password, colored in fuchsia, has no outgoing nor incoming references inside Kernel package – see in the head of Kernel package blueprint the column and the row of this class. Looking closely at Password, we see that it is referenced by only one package: RemoteDirectory refers to Password class in Kernel – see the orange surface and the fuchsia referenced class in the body of RemoteDirectory package blueprint. Thus we think that moving Password class to this last package will increase the cohesion of both packages, Kernel and RemoteDirectory.

Cyclic references with deploying or loading problems. The cyclic reference between Kernel (highlighted in orange) and Protocols (highlighted in yellow) indicated by the corresponding colored surfaces, raises problem about the order of deploying or loading the Network application. One possible way to remove this cyclic reference consists in moving class HTTPSocket to Kernel package. However, HTTPSocket also references the URL package in Network application. Therefore moving HTTPSocket to Kernel will result in disturbing the status of Kernel as a core package. To keep Kernel without references to any other package inside Network, a better solution is to move the referencing classes SocksSocket (colored in blue) and InternetConfiguration (colored in green) to Protocols package. On one hand, InternetConfiguration has neither incoming nor outgoing references inside Kernel package (see in the blueprint head the column and the row of this class). So, moving InternetConfiguration to Protocols package will increase the cohesion of both packages; On another hand, SocksSocket references 3 classes inside Kernel but is not referenced inside it. So moving SocksSocket to Protocols will increase a bit the coupling between Protocols and Kernel, but this will finally resolve the problem of cyclic references between these two packages.

5.3. Incoming Reference Blueprint Analysis

As described in previous sections, the difference between an incoming references blueprint and an outgoing one one is: the incoming package blueprint shows how a package is used by the rest of its system while the outgoing shows how a package uses the rest of its system. Figure 6 shows the incoming references blueprints for all Network packages, where only references within the Network application are taken in account (*i.e.*, references from packages outside Network are not shown). In this section we demonstrate how the incoming references blueprints complement the outgoing references ones by capturing further information about characteristics of Network packages.

Core Packages. In Figure 6 we see that Kernel is, surprisingly, less important than Protocols. In fact, Figure 6 shows, on one hand, that the most referenced packages within Network application are Protocols and Url –since they have the biggest number of surfaces within the body of their blueprints (both are referenced from 7 packages within Network). Thus we deduce that these packages are the core of Network. The figure shows, on another hand, that the Kernel package is referenced by only four packages within Network, which are Protocols, TelNetWordNet, KernelTest and RemoteDirectory. However, Protocols package heavily references Kernel package. The incoming reference blueprint of Kernel package shows that the surface in relation with Protocols tells us that Protocols package is the most referencing package to Kernel: that surface is the closest to the head of Kernel and its the largest surface into the blueprint body (it represents 4 referencing classes from Protocols package). Hence, Kernel package represents the basic package within Network.

Leaf and isolated packages. Figure 6 clearly shows Network leaf packages (*i.e.*, packages which are referenced by only one package) such as MailReaderFilters which is only referenced by MailReader, or HTMLParserEntities by HTMLParser. We also identify packages which are completely isolated (*i.e.*, are not referenced by any package), since their blueprints contain only one surface (the head surface), such as SqueakPage and TelNetWordNet packages.

Most referenced package classes. Glancing to the incoming references blueprint of Kernel package, in Figure 6, we easily spot the most referenced classes in Kernel, which are NetNameResolver and Socket. In fact those two referenced classes are represented by the top rows in the blueprint. Moreover, the number of reference nodes within those rows is clearly larger than the number of reference nodes within other rows of the blueprint. However, within the blueprint head, the reference nodes in the row of NetNameResolver class are darker than those in the row of Socket class. Thus, NetNameResolver has more internal incoming references than Socket. In the same way, we deduce that Socket has more external incoming references than NetNameResolver (see the darkness of their corresponding reference nodes within the blueprint body).
Similarly we detect dominant referenced classes into other packages: the dominant referenced class in Url package is MIMEDocument; in Protocols package, it is HTTPSocket; in RFC822 package, it is MailAddressParser.

Co-Referencers. Figure 6 shows that the packages RemoteDirectory and TelNetWordNet are referencing together the same set of packages within Network: both refer to classes into Kernel, Protocols and Url packages (see the green and orange surfaces within the body of blueprints). This gives us an idea about the similarity between RemoteDirectory and TelNetWordNet packages in terms of package co-referencing within Network. However, by looking more closely at the sur-

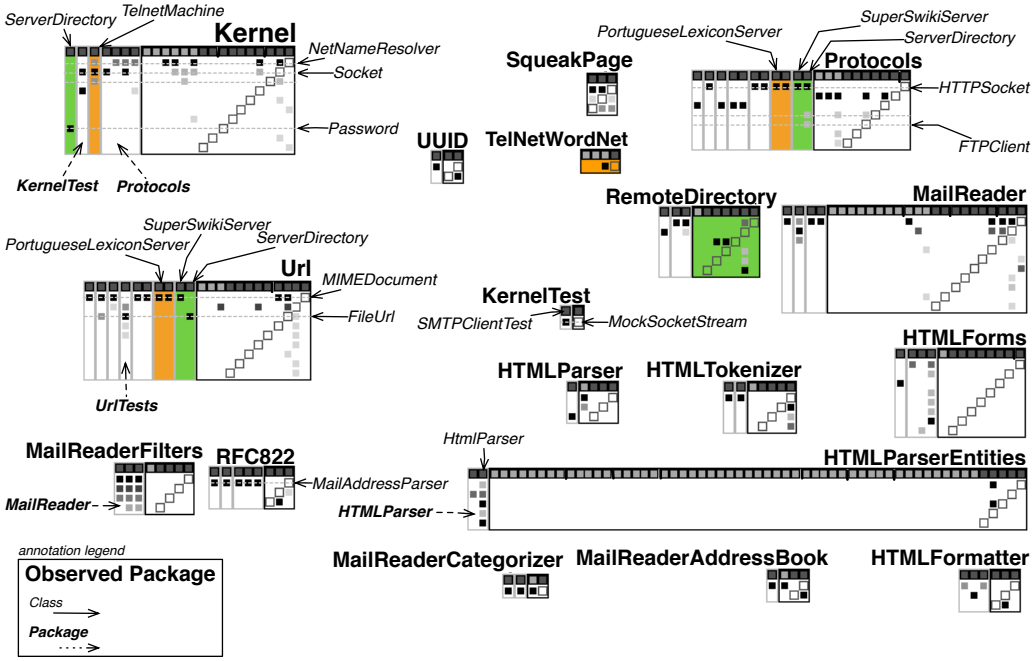

Figure 6: Global view of the incoming references in the Network application (annotated screenshots - annotation are in italic).

In this view, there are 18 packages that contain 178 classes (class nodes in blueprint heads), having, between them, 165 inter-class references (reference nodes) and causing 36 inter-package relationships (body surfaces), within the Network application scope. Color marks: surfaces of TelNetWordNet package in *orange*; those of RemoteDirectory package in *green*.

faces in relation with RemoteDirectory and TelNetWordNet packages (in Kernel, Protocols and Url blueprints), we see that the similarity between RemoteDirectory and TelNetWordNet is improper at the class granularity level. In fact, RemoteDirectory and TelNetWordNet packages do not reference the same classes within the cited referenced packages, except for Protocols package. In Protocols blueprint, we can see in the surfaces of RemoteDirectory and TelNetWordNet that the references nodes are located in one (the same) row. Thus, RemoteDirectory and TelNetWordNet reference the same class in Protocols, which is HTTPSocket. Now glancing at the blueprint of Url package, we see on one hand that RemoteDirectory and TelNetWordNet share one referenced class in Url package –it is the class represented in the first row of the Url blueprint (named MIMEDocument). On another hand, we see that RemoteDirectory references another class in Url package –named FileUrl and represented by the third row in the Url blueprint. Finally, and in the same way, we can easily identify that RemoteDirectory and TelNetWordNet do not share any referenced in Kernel package.

Tightly coupled packages. To identify tightly coupled client packages, we have just to follow large body surfaces in incoming references blueprints. Those body surfaces are always placed close to the blueprint heads. For example, in Figure 6 we easily spot in the blueprint of Kernel package that Protocols package is the most important client to Kernel, and it is tightly coupled to it. Similarly, we see in the blueprint of MailReaderFilters package that MailReader package is tightly coupled to the former one. However, the impact of potential changes of provider packages on their clients is stressed by number of references nodes within the surfaces of those client packages. Again on the blueprint of MailReaderFilters package, giving the density of references nodes within the surface of the client package MailReader, this indicates that any changes in MailReaderFilters may largely impact MailReader package.

6. Comparative Study

The goal of this study is to assess the efficiency of package blueprints with respect to classic navigation tools used for package dependencies found in IDEs. We took the Pharo open-source Smalltalk IDE as a comparison because (1) it is an advanced IDE supporting good code browsing, cross-referencer, debugger [49] and (2) because there is a community of developers willing to spend time to perform our study. The theory behind this work and this comparative evaluation [50] is that *a package blueprint offers a good representation of packages in terms of (a) their internal references, and (b) the package dependencies, by grouping them according to their package clients or providers. Such representation supports faster, and better, analysis of package dependencies than normal IDEs.*

We focus our evaluation on the following hypotheses:

1. users get better results in package-related tasks with package blueprints than with navigation tools;
2. users perform faster in package-related tasks with package blueprints than with navigation tools.

6.1. Experiment Design

17 voluntary participants, from different countries in Europe, North and South America, and from Asia, took part in the experiment. Participant background ranged from master student to professor or full-time developer. Most participants (exactly

14 participants) got a strong background in software development. Each subject was randomly assigned to use either package blueprints or navigation tools. Thus two groups of participants were formed.

We devised a series of eleven questions split in three categories, each category mirroring some developer activities with an increasing look for details but rooted in the tasks we mentioned in Section 2.1.

1. Application assessment: the first three questions target a basic understanding of an application and its dependencies, in particular, the detection of unexpected dependencies to external library.
2. Architecture assessment: four questions ask for an assessment of the architecture of the application, such as the identification of internal implementation packages versus API packages for clients, the identification of key classes, the detection of cyclic dependencies between packages.
3. Detailed assessment: four questions ask for an assessment at class level of package dependencies, in a perspective of understanding/reengineering fine-grained dependencies.

For each task, each subject was asked to play a role: first as an external developer and potential client of the application, second as an architect of the application with a broad view, third as a developer of the application.

We selected the Glamour engine for scripting browsers (http://www.moosetechnology.org/tools/glamour) as the case for study. Glamour was selected as a sample relevant for package-related tasks (11 packages) yet of reasonable size (150 classes). Moreover, it has a well-defined domain (browser engine) which makes it easy for subjects to understand its dependencies to the base platform. Finally, Glamour is shipped with some applications using it, thus participants can also have to look for Glamour clients.

We assessed the expected answers before the study by performing the study ourselves and cross-checking the answers with the developer of Glamour.

6.2. Experimental Setup

Subjects were not supervised by an observer and had to perform the study on their own. The following protocol was proposed for both groups:

1. the subject receives materials for the study, including the questionnaire with instructions, and a ready-made environment for the study itself;
2. he only uses the tool indicated to him, and no other tool;
3. he processes questions in order;
4. he times himself for each question;
5. he should take no more than 20 minutes per question, and 1h30 for the whole study (if one question takes longer, he should stop it).

Subjects using navigation tools were considered experts enough and not given further documentation. Subjects using package blueprints had access to a presentation of package blueprints, a screencast presenting tool usage, and an earlier version of this paper. We did not provide hand-ons advice since a lot of the participants were doing it remotely.

6.3. Questions and Expected Answers

1. *How big is the application: a) in number of packages; b) in number of classes (among ranges: <100; 100-200; 200-300; >300)?*
   With this question users should make a quick assessment of the system.
   Expected answers are a) 11, and b) in the range 100-200 (precisely 157 classes).
2. *What are the most important packages: a) in terms of outgoing dependencies; b) in terms of incoming dependencies; c) Overall, considering both outgoing and incoming dependencies?*
   Users then proceed to identify important packages of the application. Packages with lots of outgoing dependencies are likely to be core packages and implementation packages. Packages with lots of incoming dependencies provide implementation for clients (internal and external). The third question allows user to catch up packages which score well on both sides, without being important on a single side.
   Expected answers are a) Glamour-Morphic, b) Glamour-Core, and c) Glamour-Core.
3. *Focus on package Glamour-Morphic: a) list all package dependencies which are external to Glamour; b) in this list, please signal any external package which is not part of Pharo base (i.e., package must be loaded with Glamour); c) are there other unexpected/unwanted package dependencies?*
   Finally, users should assess the external dependencies of Glamour. We target Glamour-Morphic as it has an obvious concern relating to the system (bridge to the Morphic UI framework). The user should retrieve quickly all dependencies to check whether Glamour is self-contained and whether it makes reasonable requirements.
   Expected answers: the complete list includes 15 packages, most relating (unsurprisingly) to Morphic, widgets, and system graphics. There are two dependencies which are not part of the base system, Mondrian and Magritte. Another unwanted dependency is DeprecatedPreferences, which is obviously a dependency to be discarded.

The user now switches to architecture assessment for the next four questions.

4. *Please characterize each Glamour package as either: a provider package for external clients (package with which external clients interact); (or) an internal package (package which should not be accessed by external clients).*
   The goal of this question (refining question 2) is to make an assessment of packages as parts of the architecture.
   Glamour-Core, Glamour-Browsers, Glamour-Presentations, and Glamour-Tools are considered as provider packages. Glamour-Announcements, Glamour-Helpers, and Glamour-Morphic are implementation packages.
5. *Are some packages optional/modular (could be unloaded without impacting the application)?*

Users should detect packages without (internal) clients. Unloading such packages would not affect the application. Test packages and the Examples package are modular. More surprising, the Glamour-Morphic package is also modular, as Glamour does not depend on a specific rendering engine (other engines are available for different platforms).

6. *What are the important classes (consider incoming and outgoing dependencies) in Glamour-Core? If possible, explain their roles.*
   In a manner similar to question 2, the user now dives deeper to look for classes and should identify entry points in the package.
   Expected answers should include GLMBrowser and GLMPresentation, all members of package Glamour-Core.
7. *Are there direct cyclic dependencies from Glamour-Core to another package?*
   Finally, the user should be able to detect a violation of a common architecture rule.
   There are no cyclic dependency between Glamour-Core and another package. While it looks like a simple question, the primary goal is to assess how fast the user reaches this conclusion.

Finally, the user switches to a developer's point of view.

8. *What are the most cohesive packages of the application?*
   The goal of this question was to test whether user could form an opinion just by browsing classes (control group) or by scanning the head surface (in package blueprints).
   Expected answers are packages Glamour-Core and Glamour-Browsers.
9. *You notice a dependency to package DeprecatedPreferences in Glamour-Morphic. Can you detect the faulty class? Explain the dependency: do you see an easy way to solve it?*
   Given the above hint, the user should track references to the Preferences class, which is now deprecated. Thus, the user is able to identify classes making references to wrong packages.
   The user should notice 1) class GLMMorphicRenderer and 2) its dependency on the font system in old preferences.
10. *Can you explain the organization of Glamour-Morphic and its relationship with other packages?*
    With this question the user should rationalize the purpose and the implementation of the Glamour-Morphic package, providing a simple picture from multiple levels of information (classes and packages).
    We expected the users to retrieve the following information: 1) package provides widgets, builds on top of Morphic; 2) class GLMMorphicRenderer is important in the package; 3) package also provides an event system; 4) there are no Glamour client to the package (signaling it is a modular package).
11. *Multiple packages of Glamour have dependencies to external library Mondrian. List such packages. Could you extract this dependency and make it optional (you can propose a solution)?*
    The user performs an assessment for a refactoring task. He should be able to retrieve multiples sources of dependencies (in multiple packages) and assess whether such sources could be extracted.
    Expected answers: there are dependencies to Mondrian in the Glamour-Tests, Glamour-Morphic, Glamour-Examples, and Glamour-Presentations packages. Fortunately, it appears all such dependencies relate to Mondrian support in Glamour but are not necessary in the core application. They could be extracted as class extensions in a separate Glamour-Mondrian package.

6.4. Results

For each participant, we computed precision, recall, and f-score for questions Q2–Q6, Q8, and Q10–Q11. For Q1, Q7, and Q9, a normalized score was computed based on participant answer. Finally, we drew boxplots for each group, showing f-score or normalized score and time for each question in Figure 7. All materials used for the study, including evaluation sheet and answers from the participants, are available on our website[2].

1. Most participants performed well in assessing the size of the application. Participants using package blueprints were faster. However, there were minor errors with package blueprints: some counted only 10 packages instead of 11 and some estimated the number of classes above 300.
   The first mistake can be explained by a design defect on our part: one small package was totally empty of internal and external outgoing dependencies. At first we decided that it was not necessary to display an "empty" package blueprint. We revised this decision based on the above result.
2. Users of package blueprints were able to quickly identify important packages based on dependencies, with an average f-score of 0.87 and 7 out of 8 in less than 5 minutes. On the contrary, users of the browser performed poorly, with an average f-score of 0.48 and more than 11 minutes: they often cited Glamour-Core but forgot Glamour-Morphic.
   A possible explanation is that browser users focus on the name Glamour-Core to assess its importance, while package blueprints offered a more factual view of dependencies.
3. Users of package blueprints successfully identified all 15 dependencies of package Glamour-Morphic. 5 out of 8 found external dependency Mondrian and Magritte, and 3 found DeprecatedPreferences. On the other hand, browser users performed poorly once again with an average f-score of 0.46. Two participants found Mondrian and Magritte as external dependencies, and none cited DeprecatedPreferences.

[2] http://rmod.lille.inria.fr/archives/demos/PackageBlueprint

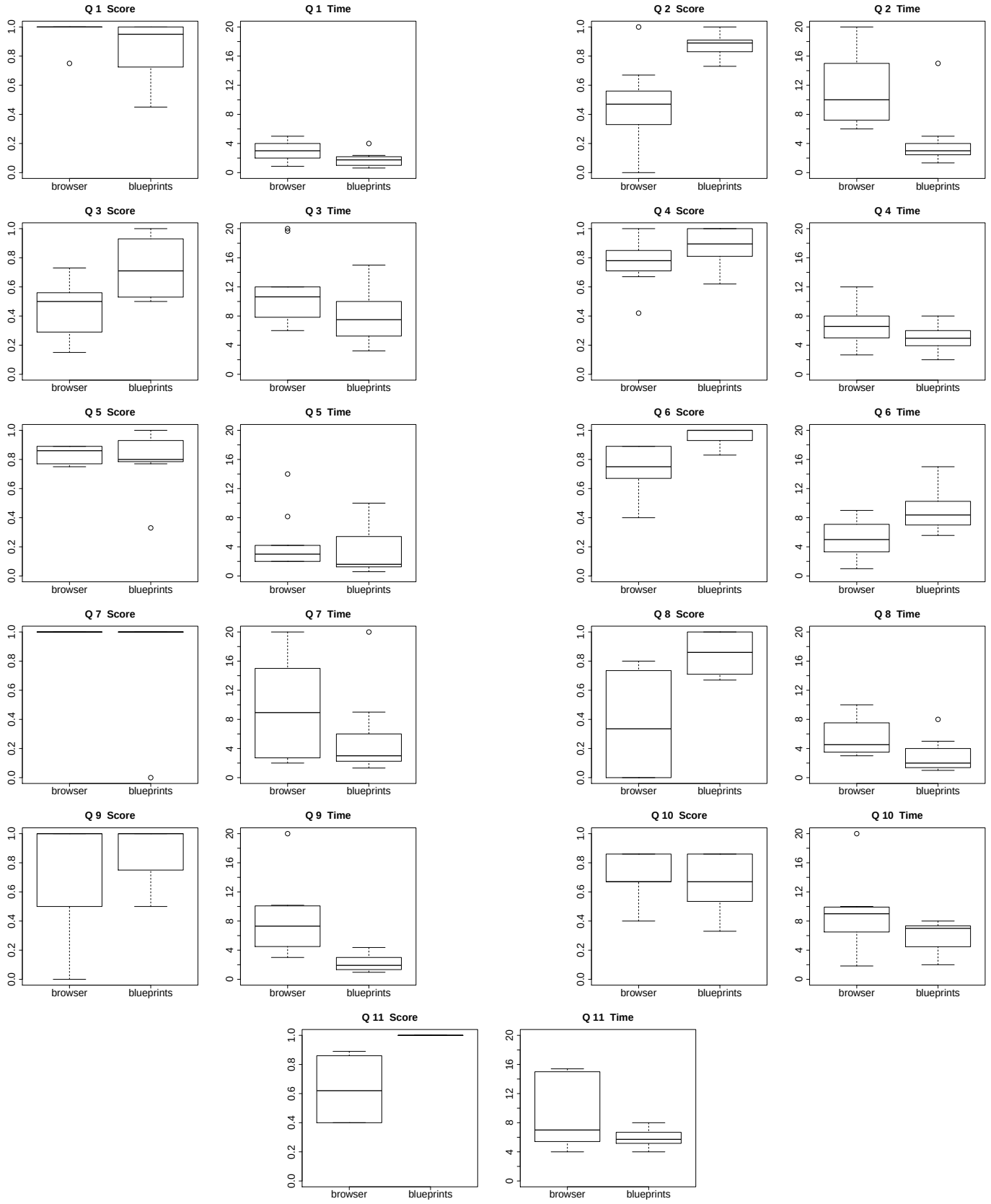

Figure 7: Boxplots showing score and time for questions 1 to 11. Browser group shown on left and package blueprints on right for each question.

4. The identification of provider and internal packages works well with both tools. There is no significant difference in score and time, although users of package blueprints performed slightly faster and better with an average f-score of 0.88.
5. A question of interest was whether participants would be able to detect Glamour-Morphic as modular as intended by Glamour designer. Only 2 people detected the case with package blueprints, and none with the browser. However, both groups performed well overall.
6. Users of package blueprints achieved a near perfect f-score of 0.96 while looking for important classes. Users of browser performed quite well at 0.74. More surprisingly, this is the only question where the former were slower (9 minutes) than the latter (5 minutes).
   A possible explanation is that package blueprint users took time to scan through many classes and their dependencies, while browser users were more successful building a mental map of the inner workings of the system by browsing classes themselves.
7. Participants answered correctly except for one case. The interesting difference is how much time each group used to reach this conclusion. Indeed, users of package blueprints were much faster (with 5 out of 9 answering in 3 minutes or less) while users of browser had different reaction times, ranging from 2 minutes to 20 minutes.
8. This question on package cohesion was the most problematic. 4 out of 8 participants bypassed this question in the browser group, telling they could not perform such a task in a limited time, and one did the same in package blueprint (telling he did not understood the visualization of cohesion in package blueprints). The remaining browser users had indeed poor results with a f-score at 0.37, but package blueprint users performed much better at 0.86.
9. Most participants found and could explain the dependency to DeprecatedPreferences (only two users of browser did not succeed). The significant difference is again in the time used to find the source of the problem. Package blueprint users were much faster with an average of 2 minutes against 8 minutes for browser users.
10. This question was open as it elicited the subjective interpretation of a package design. We still received mostly good answers, with browser users performing slightly better than package blueprint users. Surprisingly, only browser users spotted the importance of events in the package, while only 2 users of package blueprints noticed the absence of internal clients (same ones as for Q5).
11. Package blueprint users all proposed a complete solution to the extraction of Mondrian. In particular, they were able to identify all dependencies to Mondrian across packages. Browser users were less successful with this aspect.

It took in average an hour for package blueprint users to perform the study with an average score of 0.85, against an hour and a half for browser users with an average score of 0.70. Note that two expert programmers refused to perform the experiment without the package blueprint, because they felt the default IDE was not up to the task. Table 1 shows the results of a tow-tailed Wilcoxon test comparing between Package Blueprint and the default IDE, using the results of our 11 questions discussed above. The results confirm our hypothesis mentioned earlier in Section 6. Indeed, the results show that Package Blueprint, overall, registers a better score (+0.11) and a shorter time (-2 minutes) than the default IDE –with regard to the aforementioned results of our 11 maintenance questions. These results are statically significant at $\alpha = 0.001$. Hence, *Package Blueprint supports faster, and better, analysis of package dependencies than normal IDEs.*

Table 1: Two-tailed Wilcoxon test, with $\alpha = 0.05$. Comparison between Package Blueprint (PB) and Pharo IDE using the Score and Time results of our 11 questions (see Figure 7).

	Score ([0..1])	Time (minute)
Δ (PB - Pharo):	**+ 0.11*** **	**- 2.05*** **
0.95% Confidence Interval		
lower bound:	+ 0.03	- 3.40
upper bound:	+ 0.17	- 1.00

Signification code: $\alpha \leqslant 0.001$ (***); $\leqslant 0.01$ (**); $\leqslant 0.05$ (*).

Summary to this section. Overall, users of package blueprints are faster. They performed very well when the task involves a complete identification of all dependencies for a package or a class. Tasks involving more judgment, like identifying important classes or explaining package design are less impacted and browser users performed equally well, as they spent more time reading the code, which could give them a better understanding of some inner workings of the system, not just of dependencies between packages and classes.

6.5. Threats to validity

Having presented the comparative study of package blueprint versus an advanced code browser and described the study results, this section identifies potential threats to validity relevant to the study.

First of all, it is worth to note that most participants to the study performed the evaluation remotely due to geographical distances. As a consequence, for practical reasons, those participants performed the evaluation without our supervision. Hence, we were not able to control the time they took, possible disturbance, and their misinterpretations of the study questions.

Moreover, despite our effort to design precise questions, most questions were open to human interpretations. Due to uncontrollable human-factors, this gave us a large distribution of potential answers for each question, depending on the time the user spent on the question, and his own interpretation of the question.

Other threats to validity can be due to the limited number of participants since we were not able to attract more than 17 voluntary participants to the study. Furthermore, only 4 participants among 17 ones are women. To the best of our knowledge, none of participants suffer from color deficiencies nor had been corrected to normal color vision.

Finally, one can arguments that there exist some tools, may be visualization tools (*e.g.*, [40, 41]), which are more dedicated to browsing package dependencies than standard code browsers, and thus they could have performed better. However, as discussed in Section 3, relevant existing tools that can facilitate package understanding can not be adapted to browse the dependencies at class and package levels, and/or to reveal package internal structure. As a consequence, those tools are not adapted to answer most questions of the study. We decided to compare package blueprint with standard, advanced, code browser as we believe standard code browsers are the most polyvalent tools with respect to the package blueprint. Hence, the role of this study was to assess how much package blueprint could improve the experience of standard browser users.

7. Discussion

Having presented the advantages of package blueprint for analyzing and understanding package structure and dependencies, in this section we discus the design of package blueprint from different perspectives. Finally, we present the main limitations and drawbacks of the proposed visualization.

7.1. Design Choices

Navigation Choices. It is widely argued that visualizations are not fully useful unless they support interactive exploration. There is a large spectrum of means to support navigation such as zoomable interfaces, multiple simultaneous views, fly-by-help showing underlying details. In package blueprint, we did not introduce semantic-zooming support for the following reasons: (a) in this paper we target a visual map that acts as really fast summary of a package and its relations; (a) we want to assess whether the presented visualization, package blueprint, is enough to support the tasks mentioned before. The idea of small multiples presented by Tufte influenced the design of package blueprint [51] even if the fact that package blueprint shows the size of a package breaks the small multiple effect. However, our experiences with package blueprint show that the user really want to know where a reference is made and the name of the packages/classes. For this purpose, we successfully used method source code as fly-by-help in another research topic [52]. As a future work, we would like to experiment and use fly-by-help to show for a given reference node the actual methods causing that reference. We also would like to experiment a semantic-zooming out of package blueprint displaying the visualization without classes and references nodes, as demonstrated in the basic layout of package blueprint in Figure 1. We believe such a semantic-zooming out of package blueprint would give a fast overview of package size and connectivity with other packages, without providing specific details about the the distribution of package internal and external references.

Outgoing vs. incoming. Ideally we would have preferred to have only one view showing in a structured way (*e.g.*, by using enclosure placement strategy: surfaces) the incoming and outgoing references made by packages. Our attempts were not satisfactory. Naturally, displaying both incoming and outgoing references in one view will considerably increase the complexity of package blueprint visualization: (a) increase the view occupied-space to display both referenced and referencing classes and packages; (b) use more visual semantics to distinguish clients from providers, and distinguish whether a reference node represents incoming references, outgoing ones, or both. This is why we decided to split them in two views. Having two views showing different flows of dependencies can be confusing and it took us several attempts and experiments to find a solution so that the reader can distinguish the incoming and outgoing flows. For this purpose, we took the general shape of the package blueprint (horizontal vs. vertical) as distinctive sign between the two semantics. To help the reader, we always keep the reading order of a reference from top to bottom in both views as illustrated in Figure 1. In some cases, it is practical to see at the same time the incoming and outgoing view of the same package. Therefore, our tool lets the user see both views of a single package side by side. Nevertheless, this solution does not work well when we want to see a complete system because it mixes concerns.

Shapes. For the time being we represent the classes and references nodes with squares only. We could convey more information by using several visually distinct shapes. But it is not clear which ones and how efficient the results will be since the shape size is intentionally quite small to provide a compact overview.

Position Choices. Package blueprint uses two placement strategies that are *enclosure* and *adjacency* [17]. We designed package blueprint to be structured around the notion of *surfaces* that groups classes and their dependencies by their package (*i.e.*, enclosure placement); whilst the package blueprint' surfaces have two categories: (1) the head surface which represents the parent node, that is the package under-analysis; (2) body surfaces which represent the packages in relation with the package under-analysis (*i.e.*, the parent node), those are placed next to their parent node (*i.e.*, adjacency placement). However, package blueprint is a hybrid visualization because dependencies are represented as nodes grouped into their corresponding surfaces. We used those placement strategies together to display the distribution of package references in structured manner and at both class and package levels, not only internal references, but also external ones. In this way, users can focus on the first surface (*i.e.*, the top surface in an outgoing references blueprint) to understand the internal package structure; or focus on body surfaces (or one of them) to understand the exact relations between the observed package and its providers/clients (or one of them), in an outgoing/incoming references blueprint. Moreover, to help users in reading package blueprint, body surfaces are ordered, after the head, from most-coupled package to less-coupled one. This way, we do not force the reader to scroll through big visualizations, and use the fact that the reader pays more attention to the surfaces that are close to the head.

Seriation. Rows within a surface are sorted according to the number of references they contain. In an earlier version we

applied the dendrogram seriation algorithm [53] to group lines having similar referencing classes. However the resulting views were not as meaningful as with a simple ordering.
In a package blueprint head, internal classes are ordered so that the head presents a symmetric matrix. This way, when the user focuses on the *i* column (*i.e.*, a column reserved for class *x*) s/he can easily see the information about the internal references within the package of this class by looking to the *i* row in the package blueprint head. Such an ordering reveals also the direct cyclic references within the package under consideration. In previous versions, the head only showed classes performing references [31] and our users suggested such a change to be able to grasp package size.

Impact of Boundaries. We color classes that do not belong to the application in cyan. This way, users distinguish clearly the dependencies from/to classes packaged outside the analyzed application, from the dependencies among the analyzed application classes. Moreover, we found it really effective to use colors for marking packages (surfaces) or classes (classes and references nodes) so that the user can interactively mark entities on which he wants to focus on; this increases the usability of the tool and speeds up understanding packages.

7.2. Limitations

Inheritance dependencies. In this paper we considered only one type of inter-class dependencies, namely reference. However, other types of dependencies, such as inheritances, can exist between classes. We believe that presented package blueprint views, which are specific to outgoing and incoming references, are not well adapted to show inheritance hierarchies of classes. Therefore, in [31] the authors provide a specific view namely inheritance package blueprint, that is tree-based visualization rather matrix-based one, to support understanding of inheritance dependencies at class and package levels. We believe that the inheritance package blueprint, as defined in [31], together within outgoing references blueprint and incoming references one, represent an integral set of views for understanding package structure and dependencies.

Package Nesting. Currently we do not support package nesting. A solution like the one proposed by Lungu *et al.* seems complementary to ours and interesting to deal with package nesting [12].

Dependency Paths. Another limitation of package blueprint is that it fails to support the understanding of dependency paths, *i.e.*, is there a path from one class to another one using multiple intermediaries. This is an advantage for node-link visualizations against matrix-based ones [28], such as the package blueprint visualization. In package blueprint, we are currently limited to see the references made to another class and we have to follow manually the path. Nevertheless, such a limitation could be addressed by visualizing, optionally, cross-links between surfaces (or classes nodes) of different blueprints, so that users can follow the dependency paths. However, this would cause the known drawback of node–link visualizations, regarding link crossing and the unsuitability for dense graphs [28, 45].

7.3. Drawbacks

Naturally, the package blueprint inherits the drawbacks of matrix–based representation. As outlined in Section 4.1, according to the study of Ghoniem *et al.* [28, 45] that compares between node–link and matrix-based representations, matrix-based representations suffer from *scalability*. The study states, on one hand, that node–link representations are more compact than matrix-based ones. Nevertheless, on another hand, node–link representations suffer from problems of edge crossing and node overlappling, and as a consequence, they are not suitable for dense graphs. This drawback of node–link representations was widely outlined [24], even for very advanced node–link visualizations [23, 30].

As a consequence, and despite the scalability problem of matrix-based representation, we chose this representation for designing the package blueprint visualization because the following facts:

- There are many visualizations, said scalable, that are adapted to overview the connectivity among packages (*or* classes) in a compact manner. However, those visualizations are not adapted to reveal the distribution of dependencies, both *intra* and *inter* packages, at package and class levels (see Related Work Section 3).

- Thus, there are many zoomed-out visualizations that scale well with few information about package dependencies, but there is no complementary, zoomed-in, visualizations that reveal in details package structure and dependencies.

- We target a visualization that reveals *in details*, and *clearly*, the distribution of package dependencies, at package and class levels. According to the authors' experience, such a graph is usually very dense in real-world applications. In fact, this is why we target a visualization to assist maintainers in their refactoring and assessment decisions about package structure.

- Finally, we believe maintainers do not face such decisions when they overview a whole large system consisting of some hundreds packages, but rather when they are in contact to an observed package or a group of few packages (subsystem). As a consequence, scalability concerns would be considerably mitigated in such a context.

However, to meet scalability concerns and still support visual information, we implemented different visual annotations (colors) that users can use on demand. Those annotations, as described in Section 5.1, help users to understand the multiple interactions that a package can have with other packages.

8. Conclusion

In this paper, we tackled the problem of understanding the details of packages with a focus on their dependencies. We described package blueprint, a visual approach for characterizing package structure and quantifying the distribution of package dependencies, at package and class levels. Although package

blueprint reveals in details package structure and dependencies, we believe it is still a compact visualization supporting overview of software applications in limited display, of reasonable size, without losing the essential details about the distribution of package internal and external references. Therefore it can be used to get a first impression of a system and also to understand fine-grained structures and relations.

While designing package blueprint, we tried to exploit gestalt visualization and small multiples principles [51]. We successfully applied the visualization to real-world software applications and we have been able to point out large and complex packages, core packages and/or classes, isolated packages, cohesive and/or sparse packages, tightly coupled packages, misplaced classes (refacoring candidates) and references (architecture violation), direct cyclic references, and badly designed packages. We also introduced interactivity to help users focus and navigate between several packages within a given software application.

We do not consider that package blueprint should be used in isolation from other tools and/or standard code browsers. In our recent work on remodularization, we use DSM to spot cyclic dependencies [15, 40], then we zoom on the packages and use package blueprint to get a finer understanding of the package references. The synergy between DSM and package blueprint proved to be really useful. In addition, sometimes we complement the view using Distribution Map [54] to understand how a property (such as developers) spreads on a set of packages.

We validated the package blueprint usability by conducting tests with several independent software engineers. The results were positive, even if the numbers of testers was low (22). Testers concluded that the package blueprint is useful for understanding and analyzing packages. We performed a separate comparative study with a group of software developers which shows an improvement in both time and precision for package maintenance tasks compared to standard code browsing tools.

References

[1] R. C. Martin, Design principles and design patterns, www.objectmentor.com (2000).

[2] R. Pressman, Software Engineering: A Practitioner's Approach, 7th Edition, McGraw-Hill, Inc., 2010.

[3] H. Abdeen, S. Ducasse, H. A. Sahraoui, Modularization metrics: Assessing package organization in legacy large object-oriented software, in: International Working Conference on Reverse Engineering (WCRE'11), IEEE Computer Society Press, 2011, pp. 394– 398.

[4] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999.

[5] H. Abdeen, S. Ducasse, H. A. Sahraoui, I. Alloui, Automatic package coupling and cycle minimization, in: Proceedings of the 16th International Working Conference on Reverse Engineering (WCRE'09), IEEE Computer Society Press, Washington, DC, USA, 2009, pp. 103–112.

[6] H. Abdeen, H. A. Sahraoui, O. Shata, N. Anquetil, S. Ducasse, Towards automatically improving package structure while respecting original design decisions, in: Proceedings of the 20th International Working Conference on Reverse Engineering (WCRE '13), IEEE Computer Society Press, 2013, pp. 212–221.

[7] N. Anquetil, T. Lethbridge, Experiments with Clustering as a Software Remodularization Method, in: Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering), 1999, pp. 235–255.

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

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

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

[11] S. Ducasse, M. Lanza, L. 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.

[12] M. Lungu, M. Lanza, T. 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.

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

[14] M.-A. D. Storey, K. Wong, F. D. Fracchia, H. 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.

[15] J. Laval, S. Denier, S. Ducasse, A. Bergel, Identifying cycle causes with enriched dependency structural matrix, in: WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering, Lille, France, 2009.

[16] H. Abdeen, I. Alloui, S. Ducasse, D. Pollet, M. Suen, Package reference fingerprint: a rich and compact visualization to understand package relationships, in: European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society Press, 2008, pp. 213–222.

[17] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J.-D. Fekete, D. W. Fellner, Visual analysis of large graphs: State-of-the-art and future research challenges, Comput. Graph. Forum 30 (6) (2011) 1719–1749.

[18] I. Herman, G. Melançon, M. S. Marshall, Graph visualization and navigation in information visualization: A survey, IEEE Transactions on Visualization and Computer Graphics 6 (1) (2000) 24–43.

[19] M.-A. D. Storey, D. Čubranić, D. 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.

[20] M.-A. D. Storey, H. A. Müller, Manipulating and documenting software structures using SHriMP Views, in: Proceedings of ICSM '95 (International Conference on Software Maintenance), IEEE Computer Society Press, 1995, pp. 275–284.

[21] N. Churcher, W. Irwin, R. Kriz, Visualising class cohesion with virtual worlds, in: APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation, Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2003, pp. 89–97.

[22] S. Karouach, B. Dousset, Visualisation de relations par des graphes interactifs de grande taille, Journal of ISDM (Information Sciences for Decision Making) 6 (57) (2003) 12.

[23] D. Holten, Hierarchical edge bundles: Visualization of adjacency relations in hierarchal data, IEEE Transactions on Visualization and Computer Graphics 12 (5).

[24] N. Henry, J.-D. Fekete, M. J. McGuffin, Nodetrix: a hybrid visualization of social networks, IEEE Trans. Vis. Comput. Graph. 13 (6) (2007) 1302–1309.

[25] J. Abello, F. van Ham, Matrix zoom: A visual interface to semi-external graphs, in: 10th IEEE Symposium on Information Visualization (InfoVis 2004), 10-12 October 2004, Austin, TX, USA, IEEE Computer Society, 2004, pp. 183–190.

[26] A. Marcus, L. Feng, J. I. Maletic, 3D representations for software visualization, in: Proceedings of the ACM Symposium on Software Visualization, IEEE, 2003, pp. 27–ff.

[27] L. Voinea, A. Telea, J. J. van Wijk, CVSscan: visualization of code evolution, in: SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization, ACM, New York, NY, USA, 2005, pp. 47–56.

[28] M. Ghoniem, J.-D. Fekete, P. Castagliola, On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis, Information Visualization 4 (2) (2005) 114–135.
[29] C. Müller, G. Reina, M. Burch, D. Weiskopf, Subversion statistics sifter, in: Proceedings of the 6th international conference on Advances in visual computing - Volume Part III, ISVC'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 447–457.
[30] D. Holten, Visualization of graphs and trees for software analysis, Ph.D. thesis, Computer science department, iSBN 978-90-386-1882-1 (2009).
[31] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, I. Alloui, Package surface blueprints: Visually supporting the understanding of package relationships, in: ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance, 2007, pp. 94–103.
[32] H. Abdeen, Visualizing, assessing and re-modularizing object-oriented architectural elements, Ph.D. thesis, Université de Lille (2009).
[33] L. C. Briand, J. W. Daly, J. K. Wüst, A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on Software Engineering 25 (1) (1999) 91–121.
[34] M. Lanza, R. Marinescu, Object-Oriented Metrics in Practice, Springer-Verlag, 2006.
[35] L. C. Briand, J. W. Daly, J. K. Wüst, A Unified Framework for Cohesion Measurement in Object-Oriented Systems, Empirical Software Engineering: An International Journal 3 (1) (1998) 65–117.
[36] D. Beyer, Co-change visualization, in: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM), Industrial and Tool volume, 2005, pp. 89–92.
[37] S. Eick, T. Graves, A. Karr, A. Mockus, P. Schuster, Visualizing software changes, IEEE Transactions on Software Engineering 28 (4) (2002) 396–412.
[38] J. Froehlich, P. 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.
[39] X. Xie, D. Poshyvanyk, A. Marcus, Visualization of CVS repository information, in: WCRE'06: Proceedings of the 13th Working Conference on Reverse Engineering, IEEE Computer Society, Washington, DC, USA, 2006, pp. 231–242.
[40] N. Sangal, E. Jordan, V. Sinha, D. Jackson, Using dependency models to manage complex software architecture, in: Proceedings of OOPSLA'05, 2005, pp. 167–176.
[41] H. Abdeen, S. Ducasse, D. Pollet, I. Alloui, Package fingerprints: A visual summary of package interface usage, Inf. Softw. Technol. 52 (12) (2010) 1312–1330.
[42] J. Bertin, Semiology of Graphics, University of Wisconsin Press, 1983.
[43] E. R. Tufte, The Visual Display of Quantitative Information, 2nd Edition, Graphics Press, 2001.
[44] C. Ware, Information visualization: perception for design, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.
[45] M. Ghoniem, J.-D. Fekete, P. Castagliola, A comparison of the readability of graphs using node-link and matrix-based representations, in: Proceedings of the IEEE Symposium on Information Visualization, INFOVIS '04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 17–24.
[46] A. Treisman, Preattentive processing in vision, Computer Vision, Graphics, and Image Processing 31 (2) (1985) 156–177.
[47] C. G. Healey, Visualization of multivariate data using preattentive processing, Master's thesis, Department of Computer Science, University of Bristish Columbia (1992).
[48] C. G. Healey, K. S. Booth, E. J. T., Harnessing preattentive processes for multivariate data visualization, in: GI '93: Proceedings of Graphics Interface, 1993.
[49] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, M. Denker, Pharo by Example, Square Bracket Associates, 2009.
[50] S. Easterbrook, J. Singer, M.-A. Storey, D. Damian, Selecting empirical methods for software engineering research, in: F. Shull, J. Singer, D. I. K. Sjoberg (Eds.), Guide to Advanced Empirical Software Engineering, Springer verlag, 2008.
[51] E. R. Tufte, Visual & Statistical Thinking: Displays of Evidence for Decision Making, Graphics Press, Cheshire, CT, USA, 1997.
[52] V. Uquillas Gómez, S. Ducasse, T. D'Hondt, Visually supporting source code changes integration: the torch dashboard, in: Working Conference on Reverse Engineering (WCRE 2010), 2010.
[53] A. K. Jain, M. N. Murty, P. J. Flynn, Data clustering: a review, ACM Computing Surveys 31 (3) (1999) 264–323.
[54] S. Ducasse, T. Gîrba, A. Kuhn, Distribution map, in: Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06), IEEE Computer Society, Los Alamitos CA, 2006, pp. 203–212.