# Understanding Packages: The Package Blueprint


| | |
|---|---|
| Journal: | *Transactions on Software Engineering* |
| Manuscript ID: | TSE-2009-08-0202 |
| Manuscript Type: | Regular |
| Keywords: | Software Maintenance, Software Visualization, Program Understanding, Package, Software Evolution |
| | |

# Understanding Packages: The Package Blueprint

Stéphane Ducasse Hani Abdeen Damien Pollet Mathieu Suen Ilham Alloui
RMoD INRIA Lille Nord Europe – LIFL — CNRS UMR 8022 Université de Savoie, France

***Abstract*—Large object-oriented applications are structured over large number of packages. Packages are important but complex structural entities that may be difficult to understand since they play different development roles (*i.e.*, class containers, code ownership basic structure, architectural elements...). Maintainers of large applications face the problem of understanding how packages are structured in general and how they relate to each others. In this paper, we present a compact visualization, named Package Blueprint, that characterizes the relationships a package has with its neighbors. A package blueprint represents a package with the notion of package surfaces: groups of relationships according to the package references to/from other packages. We present two specific views: one stressing the references made by/from a package and another showing the inheritance structure of a package. We applied the visualization on two large case studies: ArgoUML and Squeak and performed a user study.**

***This paper makes heavy use of colors. Please read a colored version of this paper to better understand the ideas presented in.***

## I. Introduction

To cope with the complexity of large software systems, applications are structured in subsystems or packages. It is now frequent to have large object-oriented applications structured over large number of packages. Ideally, packages should keep as less coupling and as much cohesion as possible [26], [6], but as systems inevitably become more complex, their modular structure must be maintained. It is thus important to understand the concrete organization of packages and their relationships. Packages are important but complex structural entities that can be difficult to understand since they play different development roles (*i.e.*, class containers, code ownership basic structure, architectural elements...). Packages provide or require services. They can play core roles or contain accessory code features. Maintainers of large applications face the problem of understanding how packages are structured in general and how they are in relation with each others in their provider/client roles. In addition, approaches that support application remodularization [2], [21], [23] succeed in producing alternative views for system refactorings, but proposed changes remain difficult to understand and assess. Hence even if there is a good support for the algorithmic parts, much work remains to help users understand, compare and assess proposed solutions.

Several previous works provide information on packages and their relationships, by visualizing software artifacts, metrics, their structure or their evolution [7], [8], [11], [20], [24], [29]. However, while these approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the package characteristics (the number of classes it defines, the inheritance relationships among its classes, how the package classes inherit from and interact with classes packaged into other packages...) and help identifying their roles and impact within an application.

In this paper, we propose the Package Blueprint, a compact visualization revealing package structure and relationships. A package blueprint is structured around the concept of a surface, which represents the relationships between the observed package and its provider and client packages. Package Blueprint reveals the overall size and complexity of a package, as well as its relations with other packages, by showing the distribution of references to classes within and outside the observed package. Package complexity is defined by the quantity of relationships among the package classes (*i.e.*, internal complexity) and the quantity of relationships between the package classes and other classes (*i.e.*, external complexity). There is a direct relationship from a class A to another one B if A references B (*e.g.*, A uses B as type of some variable or A methods invoke methods of B), or if A inherits from B. Package Blueprint provides an overview of a system but without losing key details related to package relationships. We applied the Package Blueprint to several large case studies namely Squeak the open-source Smalltalk comprising more than 2000 classes, ArgoUML and Azureus. We performed a limited user study with advanced developers.

The work presented in this article extends our previous paper [12] in the following points: (a) visualization improvements based on the feedback and conclusion of a first user study, (b) addition and complementary visualization for incoming references (in addition to outgoing references), and (c) a detailed presentation of a case study as well as a user study.

Sections II & III present the challenges in supporting package understanding, and summarize the properties expected for effective visualizations. Section IV presents the structuring principles of a package blueprint, which are then declined to support an outgoing reference view, an incoming reference view and an inheritance view in Section V. Sections VI & VII present the distinct views of package blueprint at work. The next section presents the results of a limited case study. In sections IX and X, we discuss our visualizations and position them w.r.t. related work before concluding.

## II. Challenges in Understanding Packages

Although languages such as Java offer a mechanism for modelling the dependencies between packages (*i.e.*, via the

import statement), this mechanism does not really support all the information that is important to understand a package. We present a coarse list of useful information to understand packages. Our goal here is to identify the challenges that maintainers are facing and not to define an exhaustive list of the problems that a particular solution should tackle (note that our solution does not solve all those problems).

*Size:* What is the general size of a package in terms of classes, inheritance definition, internal and external class references, imports, exports to other 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 [6], [3]. The question is then to see the impact (if any) of the transformation on package cohesion and coupling. Assessing these properties is then important.

*Central vs. Peripheral:* Two correlated pieces of information are important: (1) does a package belong to the core of an application or is it more peripheral? and (2) does a package provide or use functionality?

*Developers vs. Team:* Knowing who are the developers and maintainers of the application and packages helps in understanding the architecture of the application and in qualifying package roles [15], [25]. Approaches such as the distribution map are useful in this task [10].

Actually, packages reflect several organizations: they are units of code deployment or units of code ownership; they can also encode team structure, architecture and stratification. Good packages should be self-contained, or only have a few clear dependencies to other packages [6], [3], [19]. A package can interact with other ones in several ways: either as a provider, or as a consumer or both. In addition some packages may have either a lot of references to other packages or only a couple of them. If a package defines subclasses, those can form either a flat or deep subclass hierarchy. It can contain subpackages.

Figure 1 shows different situations involving the same group of classes. For illustrating purpose, Figure 1 only shows references; the same idea holds for inheritance between classes contained in different packages. In both cases (a) and (b), there are only two packages but in case (a) most of the classes of P4 reference a class in P1, while in case (b) most classes of P4 reference internally B2. Revealing this difference is important to the maintainer who wants to understand if s/he can change the relationships between P1 and P4 during a refactoring process. In cases (a) and (c), we have exactly the same relationships between classes but the package structure is different. As mentioned by R. Martin importing a class equals importing the complete package [22], therefore importing two classes from the same package is quite different from importing them from two different packages since in the latter case we import all the classes of the two packages.

Note that understanding packages is also important in the context of remodularization approaches [2], [21], [23]. There it is important to understand how the proposed remodularisation compares with the existing code. This problem is particularly stressed in presence of legacy applications that consist of thousands of classes and hundreds of packages.

## III. Visualization Challenges

In addition to the challenges mentioned above related to the difficulties of understanding packages, the visualization itself raised challenges. Several work identified the characteristics that an efficient visualization should hold [4], [31], [33]. As our focus is on providing a first impression of a package and its context, we would like to exploit the gestalt principles of visualization and preattentive processing[1] as much as possible to help spotting important information [30], [16], [17], [33].

We stress that our visualization should take into account the following properties:

*Good mapping to reality:* The visualization should offer a good representation of the situation that the maintainer can trust and from which s/he can draw and validate hypotheses.

We expect from the visualization to highlight the general tendency of a package in terms of its internal size, internal and external references. In particular we want to spot classes or dependencies that stand out in a given package.

*Scalability and simple navigation:* The maintainer should easily access the information. The visualization should scale *i.e.*, we should be able to have system overview as well as focusing on a particular package. We target a visualization that scales well with the number of packages and of dependencies, unlike those using graph representations [9].

*Low visual complexity:* By being regular and well structured, *i.e.*, reusing the same conventions of color and position, the visualization should help the maintainer to learn it and understand it. In addition, while the visualization should offer a lot of information, it should not be complex to analyze.

## IV. Package Blueprint basic principles

To meet most the requirements cited so far, we propose our compact visualization: package blueprint. A package blueprint represents either how the package under analysis references other packages, or how it is referenced by them. Figure 2 presents the key principles of a package blueprint; these principles are realized slightly differently according to the kind (references or inheritance) and the orientation of the considered relationships (incoming or outgoing class references).

The package blueprint visualization is structured around the ”contact areas” between packages, that we name *surfaces*. A *surface* represents conceptually the relationships between the observed package and another package. In Figure 2(a) the package P1 is in relation with three packages P2, P3, and P4, via different relationships between its own classes and the classes present in the other packages; so P1 has three surfaces.

A package blueprint shows the observed package as a rectangle, vertically subdivided into parts representing its

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

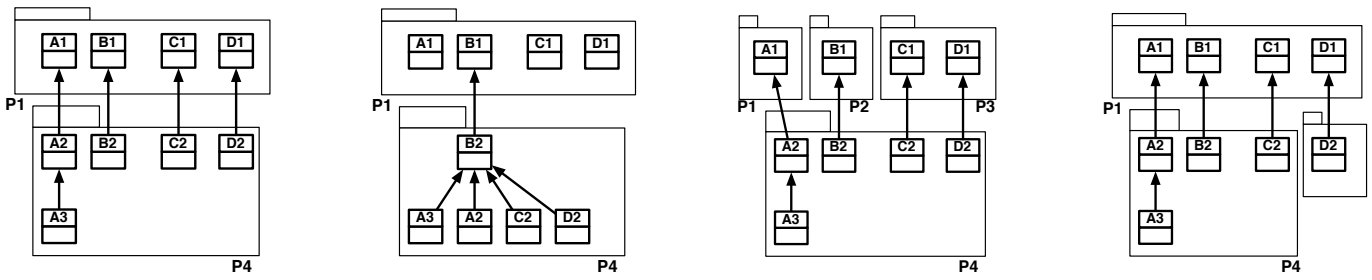
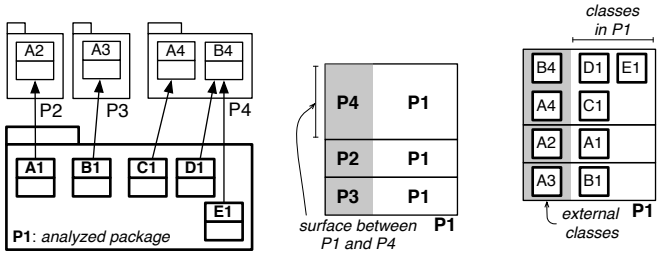



Figure 1. Different package configurations over the same number of classes.

surfaces. Each surface between the observed package and a referenced package is more or less tall, according to the strength of the relationships between them. In Figure 2(b), as P1 references three other packages, its blueprint is formed from three stacked boxes. The box of the surface between P1 and P4 is taller than the others because P1 references more classes in P4 than in P2 and in P3.

In each subdivision, we also show the classes involved in the corresponding surface. By convention, we *always* show the classes in the referenced packages on the leftmost gray-colored column of each surface, and the classes of the observed package on the right. In Figure 2(c), the topmost surface shows that classes D1 and E1 reference class B4, and that C1 references A4. If many classes reference the same external class, we show them all on a horizontal row; we can thus assess the importance of a class by looking at the number of classes on the corresponding row: in Figure 2 (c), the row of B4 stands out because the two referencing classes D1 and E1 make it wider.

To display incoming references and inheritance, we define variants of the layout: to distinguish incoming from outgoing references we rotate the layout (see Figure 4, Section V-B), and to display inheritance we arrange hierarchies as trees instead of rows (see Figure 6, Section V-C).




(a) Analyzed package (b) Stacked surfaces (c) With classes

Figure 2. Consider P1 that references four classes in three other packages (a). A blueprint shows the surfaces of the observed package as stacked subdivisions (b). Small boxes represent classes, either in the observed package (right white part) or in referenced packages (left gray part) (c).

## V. Package Blueprint Detailed Visualizations

To convey more information, we refine the basic layout previously described as illustrated in Figure 3.

### A. Outgoing Reference Blueprints

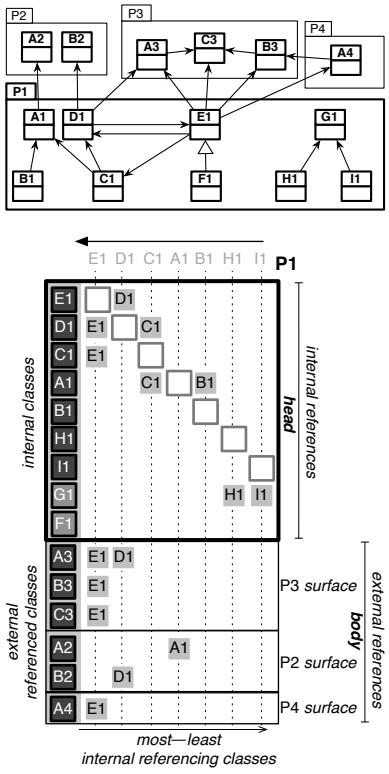



Figure 3. Surface package blueprint detailed view (Outgoing Reference view for P1).

*Internal References:* To support the understanding of references between classes inside the observed package, we add a particular surface with a thick border at the top of the blueprint. This surface is the head of the blueprint, and the rest its body. In the head, the first column represents the internal classes of the package under consideration. Thus among these classes we see those that are referenced from within the package itself: for the package P1 in Figure 3, the class A1 is referenced by B1 and C1; C1 is referenced by E1; D1 by C1 and E1; E1 by D1 and G1 is referenced by H1 and I1. The height of the head surface indicates the number of classes defined within the package.

*Position:* Internal referencing classes are arranged by columns: each column (after the leftmost one) is reserved to the same internal class for all the surfaces. The width of the blueprint indicates the number of referencing classes of the package. Figure 3 shows that class E1 internally references C1 and D1, and externally references B3, C3, A3 and A4.

We order classes of the concerned package in both horizontal and vertical directions to present important elements according to the (occidental) reading direction. Horizontally, we sort classes from left to right according to the number of classes they reference. Hence classes referencing the most occupy the nearest columns from the left gray column. Figure 3 shows that class E1 occupies the nearest column from the left gray column since it references six classes (D1, C1, A3, B3, C3 and A4) while D1 references three classes and C1 two classes; each of the remaining internal classes references only one class.

We apply the same principle to the vertical ordering, for both surfaces within a blueprint and rows (*i.e.*, external classes) within a body surface. Within a package, we position surfaces that present the most external classes the highest. Within a body surface, we order external classes from the most referenced at the top, to the least referenced at the bottom. This is why in Figure 3 the surface in relation with P3 is the highest and why the surface with P2 is above P4: there are more referenced classes into P2 than into P4.

Within the head surface, the vertical ordering of the internal classes is identical to their horizontal ordering. Figure 3 shows that internal classes (E1, D1, ... and I1) are ordered the same way vertically and horizontally. Bordered squares, that are diagonally placed from the top-left to the bottom-right within the head, help the users to clearly see the symmetry between the horizontal and vertical orderings. This diagonal helps also to detect direct cyclic references within the concerned package: within the head of P1 blueprint (Figure 3), we see that there is a direct cyclic-reference between D1 and E1 – since there are a node of D1 and a node of E1 that have symmetrical positions relatively to the head diagonal; which means that E1 refers to D1 and D1 refers to E1.

Internal classes with no reference to others are placed at the bottom of the left most column in the head (*e.g.*, G1 and F1 do no reference).

The head surface therefore conveys the package size as well as the ratio between defined classes and their internal relationships. Both referencing classes and unreferencing ones together with internal references among them are shown (*e.g.*, the unreferencing G1 is referenced by H1 and I1).

*Color:* Color intensity assigned to a node representing a class conveys the number of references it is doing: the darker the more references it does. 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. Thus, while classes on the left of surfaces will generally tend to be dark, a class that has many references but few in a particular surface will stand up in this surface since it will be light gray. For example, in a blueprint of P1 (Figure 3), within the head surface, the nodes of C1 and E1 should have the same color intensity and both should be darker than the nodes of D1 – since C1 and E1 each references two internal classes (*i.e.*, within the head surface), while D1 references only one class in the same surface. Within P3 surface, E1 should be slightly darker than D1 since the former references three classes within P3 while the latter references only one class.

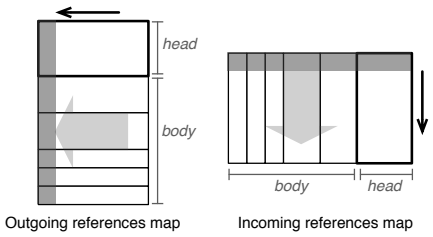



Figure 4. To distinguish it from the outgoing reference blueprint (left), we rotate the incoming reference blueprint (right) by 90°, so that the important details are still read first; in the incoming view, the references are made by the external classes, at the top, to the internal classes below them.

In the first column and within the head surface, we distinguish unreferencing internal classes from others by making their fill lighter than the one of internal referencing classes (*e.g.*, G1 and F1 fill color is lighter than the fill color of I1..E1).

Finally, we want to distinguish referenced classes depending on whether they belong to a framework or the base system, or are within the scope of the application under study. When a referenced class (in the first column and in the external referenced part - see Figure 3) is not part of the application we are currently analyzing, the fill of its node is cyan.

### *B. Incoming Reference Blueprints*

We use a view similar to the outgoing reference blueprint for exploring incoming references. However we visually distinguish the incoming reference blueprint from the former as follows: as shown in Figure 4, the global view is rotated counter-clockwise by 90°; the external referencing classes are placed at the top while the package internal classes are placed below them. The blueprint surfaces are ordered from right to left: the head is at the most right and surfaces of client packages are ordered by the number of referencing classes they enclose. The referencing classes are thus displayed on the top row, and we sort internal referenced classes from the most referenced on the second row, to the least referenced on the bottom row.

Figure 5 shows the incoming reference blueprint of P3 (Figure 3). The blueprint body has two surfaces: P1 surface and P4 surface – since P3 clients are P1 and P4. P1 surface is at the right of P4 surface, since the former involves more referencing classes (two classes: E1 and D1) than the latter (one class: A4). C3 is the most referenced class within P3: C3 is referenced from three classes (A3, B3 and E1), while A3 is referenced from two classes (E1 and D1) and B3 is also referenced from two classes E1 and A4.

In a first version of the package blueprint, views were visually too close and it was difficult to distinguish them in a glance. Finding a view that was really distinct from the previous while sharing the same visual effect was important to avoid confusion.

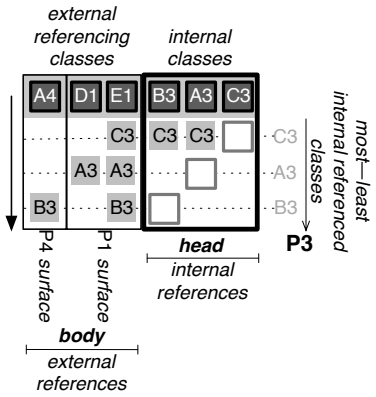



Figure 5. Package blueprint detailed view (Incoming Reference view for P3 Figure 3).

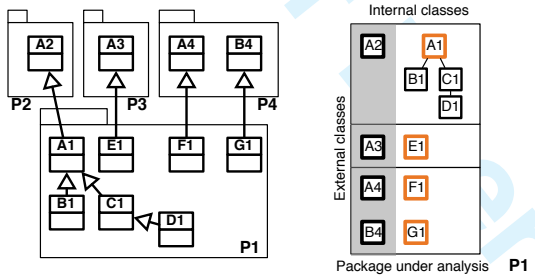



Figure 6. Inheritance package blueprint. Orange bordered classes inherit directly from external classes.

### C. The Case of Inheritance

Up to now, we only discussed references, but inheritance is a really important structural relationship in object-oriented programming. We then offer a specific view which structures the inheritance relationship within the package according to the client packages, as shown in Figure 6.

We consider only single inheritance so we can display all classes and subclasses transitively inheriting from external classes on the same row. We distinguish the direct subclasses of *external* classes by showing them with an orange border; indirect subclasses are black-bordered and arranged in trees under their superclass. Figure 6 shows the inheritance blueprint of P1. P1 classes inherit from classes packaged in distinct packages (P2, P3 and P4): A1 inherits from A2 defined in package P2, while B1, C1, and D1 inherit from A1; E1 inherits from A3 defined in package P3; F1 and G1 inherit respectively from A4 and B4, both defined in package P4. This view highlights internal inheritance roots as well as external inheritance usage.

As explained subsequently, to distinguish root classes such as Object and classes that do not belong to the application under-analysis we use cyan as fill color. Similarly, we use blue as a fill color to distinguish abstract classes. The fill color of other classes in the inheritance view still represent the number of *references* made by the class, but relatively to the *package* and not to the surface like in the reference views. This enables maintainers to correlate inheritance and reference views. For instance, in Figure 7 (c), the inheritance blueprint of the package Network::Kernel shows that most references come from the classes OldSimpleClientSocket and SocksSocket – since they have the darker fill color – which are respectively subclasses of the abstract classes OldSocket and Socket. Blueprint width represents the maximum number of subclasses at one level and its height the depth of inheritance.

## VI. An Example: The Network::Kernel Package

We are now ready to have a deeper look at an example. The Squeak Network subsystem contains 178 classes and 26 packages – making up a library and a set of applications such as a complete mail reader. Figure 7 shows the blueprints (outgoing reference, incoming reference and inheritance) of the Network::Kernel package in Squeak.

Glancing at the package outgoing reference, Figure 7 (a), we see that it has a lot of gray squares inside its head surface, which indicates that there is a lot of interactions among the internal classes of the analyzed package Network::Kernel. This blueprint shows also that there are more gray squares inside the body surfaces than inside the head, indicating that the package classes have more interaction with classes of other packages than internally in the package. This conveys a first impression of the package cohesion even if it is not really precise [6].

The number of the body surfaces indicates that Network::Kernel is in relation with 14 other packages. Most of the referenced classes are cyan, which means that they are not part of the Network subsystem. Indeed they belong to the core libraries (*e.g.*, Collections::Streams, Collections::Arrayed and Collections::Strings) on top of which Network::Kernel is defined. What is striking is that all except one of the referenced classes are outside the application (HTTPSocket in Figure 7 (a)). However, since the package is named Network::Kernel, it is strange that it refers to other classes from the same application, and especially to only one. This is clearly a layering bug.

The outgoing reference blueprint shows clearly which provider packages are important for the analyzed one Network::Kernel and which are less important: some of the referenced packages, such as Collections::Streams and Kernel::Processes, are strongly referenced by Network::Kernel – since there are a lot of gray squares inside the corresponding surfaces; other referenced packages, such as Collections::Strings, are referenced by only one class or a couple of referencing classes.

Analyzing the blueprints and inspecting the class and package names, we found via the outgoing reference blueprint, Figure 7 (a), another improper layering: the Tools::Menus surface shows that Network::Kernel is referencing UI classes (FillInTheBlank and PopUpMenu) via the package Tools::Menus which seems inappropriate.

On another hand, we learn that the class making the most internal references is named Socket: this class is represented in the outgoing reference blueprint by the second column; this latter includes, within the blueprint head, the biggest number of gray squares (4) that are darker than the remaining ones within the head. This means that Socket is referencing 4 classes within the analyzed package Network::Kernel and it is the class which does the biggest number of references within Network::Kernel.

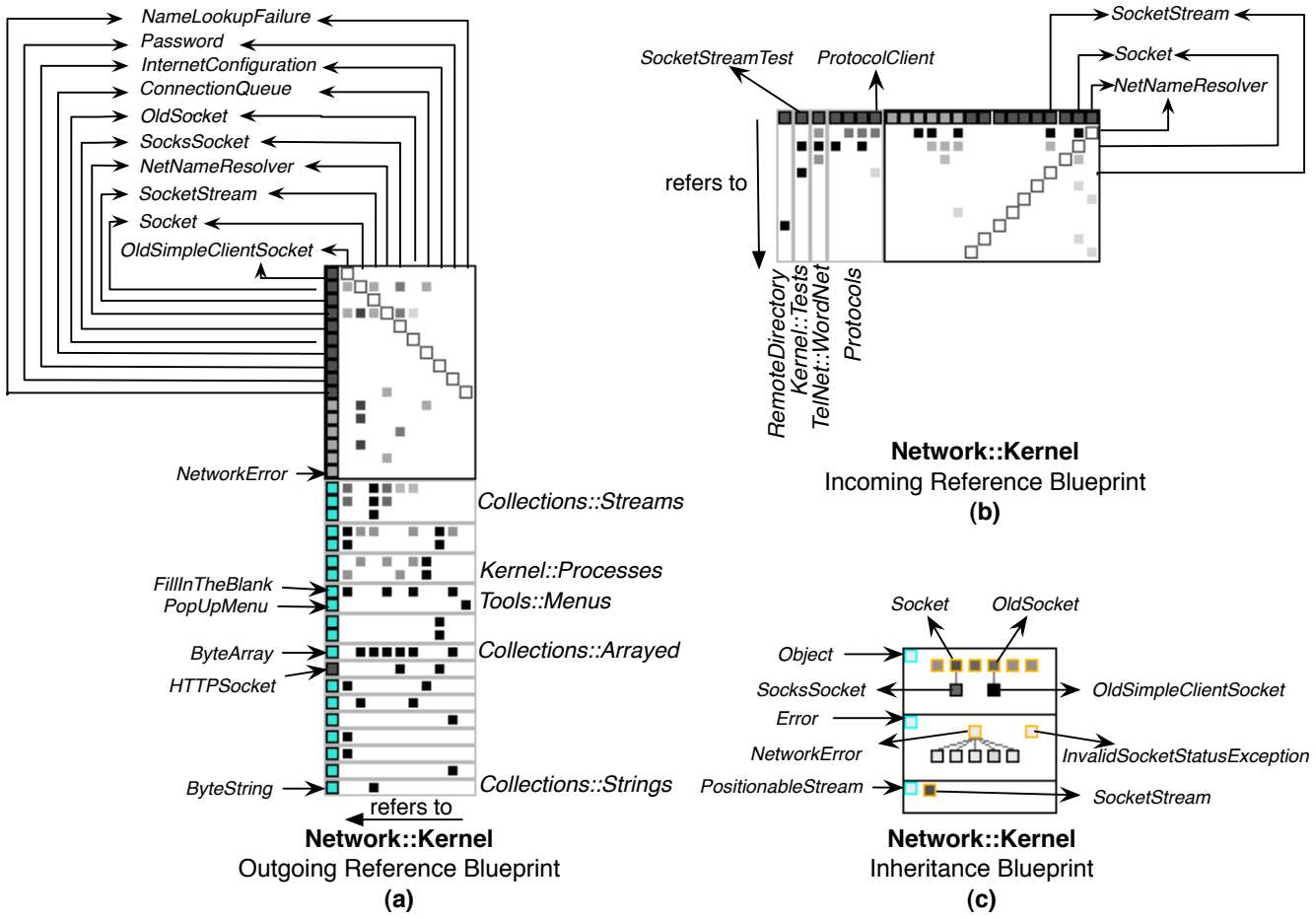



Figure 7. Analyzing the Network::Kernel Package.

We also learn that the class OldSimpleClientSocket, represented by the first column in the outgoing reference blueprint, makes the most external references – the class column includes, within the blueprint body, nine gray squares that are distributed over seven distinct surfaces, which means that this class refers to nine classes into seven packages. However OldSimpleClientSocket refers to only 2 classes within Network::Kernel as shown in the head.

The incoming reference blueprint (Figure 7 (b)) shows that the most internally referenced class is NetNameResolver, since the class row includes the biggest number of squares within the blueprint head and those squares are darker than the other ones. Similarly, we see that the second most referenced is Socket. So this is a sign of good design since important domain classes, namely NetNameResolver and Socket are well used within the package.

On the another hand, the incoming reference blueprint (Figure 7 (b)) shows that Network::Kernel is referenced by only 4 client packages and all belong to the Network system. One of those referencing package is Kernel::Tests which includes test classes. The corresponding surface of Kernel::Tests shows that there is one referencing class (SocketStreamTest) that refers to two classes within the analyzed package Network::Kernel (Socket and SocketStream). We learn that most Network::Kernel classes are not tested and this is a bad sign about the quality of the Network system.

The inheritance package blueprint (Figure 7 (c)) shows that the Network::Kernel package is bound to three external packages containing the three superclasses Object, Error, and PositionableStream. In addition the package, while inheriting a lot from external packages, is inheriting mostly from the same class, here Object. The difference between the two main surfaces is interesting to discuss: the topmost surface shows that most of the classes are directly inheriting from one external superclass (Object), while the second one shows that errors are specialized internally to the package. All in all, this makes sense and provides a good characterization of the package.

## VII. Packages Within Their Application

Understanding a package in isolation is interesting but lacks information about the overall context of classes and packages in relation with it. As shown in the following subsections, our approach also supports the understanding of the usage of a class/package within the context of a complete application. Relevant questions are for instance: Which other packages use a given class? Are two classes always co-used by others? Are two packages used with a same importance by others? and so on.

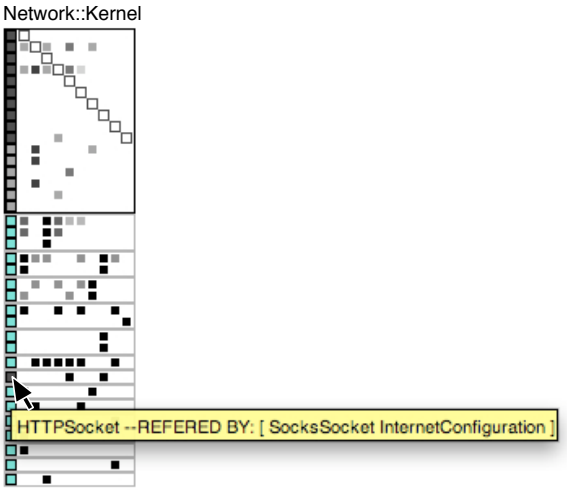



Figure 8. Interacting with package blueprint: using the mouse and pointing at the box shows, through a fly-by-help, the class and package names. In this view, the mouse is pointing to the box representing HTTPSocket and the fly-by-help shows, in addition to the class name, the name of Network::Kernel classes that refer to HTTPSocket.

To help maintainers navigate among packages and classes and to quickly collect information, we introduced a fly-by-help mechanism to the Package Blueprint. In addition to the fly-by-help (see Figure 8), maintainers can select a class (*i.e.*, any box representing the concerned class), or a package (*i.e.*, any surface representing the concerned package), and mark it with a particular color: the fill of boxes/surfaces representing the selected class/package will all have the selected color. Similarly, maintainers can mark several classes and packages with distinct colors (see Figure 9).

### A. Outgoing Reference Package Blueprint Analysis

Figure 9 shows the blueprints of all the packages referencing and defining the class HTTPSocket of the Network application.

**Hub classes.** It is striking to see that HTTPSocket, highlighted in red, is a central class of the package Protocols as it refers to most of the classes referenced by that package. We can deduce the same thing for the class ServerDirectory in RemoteDirectory package. In addition, we can easily see that almost all referencing packages to the package Protocols, whose surface is highlighted in yellow, use the class HTTPSocket of Protocols. Only RemoteDirectory refers, in addition to HTTPSocket, to two classes in Protocols. Note that HTTPSocket has no 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 gray squares.

Figure 9 also shows how the package Kernel, whose surface is highlighted in orange, is used within the Network application.

**Core Packages.** Apparently, Kernel is less important than Protocols, *i.e.*, Kernel is referenced by 3 packages (Protocols, RemoteDirectory and TelNetWordNet), while Protocols is referenced by all other packages in Figure 9 (5 packages including Kernel). However, looking to the orange surface in the body of Protocols package blueprint, we find that 4 classes of Protocols reference 3 classes of Kernel, while the yellow surface in the body of Kernel package blueprint shows that only 2 classes of the latter reference a single class (HTTPSocket) of Protocols. In addition, Kernel does not refer to any other package in Network application (classes colored in cyan do not belong to the subsystem under analysis). Looking more closely at Protocols referenced packages, we can see that Kernel (the orange surface in the body of Protocols package blueprint) is placed above Url and RFC822, the three only packages referencing classes belonging to the Network application. This reinforces the idea that Kernel is the basic package of the Network application core.

**Cyclic References.** On another side, the cyclic reference between Kernel and Protocols raises the known 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 refers to URL package. Therefore moving HTTPSocket to Kernel will result in adding one referenced package to the latter, thus disturbing its status as a core package. To keep Kernel without references to any other Network package, a better solution is to move the referencing classes SocksSocket, colored in blue, and InternetConfiguration, colored in green, to Protocols package. InternetConfiguration has no incoming nor outgoing references inside Kernel package (see in the head of the blueprint, the column and the row for this class), but InternetConfiguration references the HTTPSocket class in Protocols package . So, moving InternetConfiguration to Protocols package will increase the cohesion of both packages ; SocksSocket refers to 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 will increase the Protocols package cohesion –since SocksSocket refers to HTTPSocket in Protocols package . This way Kernel becomes a proper core package for Network application.

**Potentially misplaced classes.** 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.

**Internally loosely connected packages.** Figure 9 shows that HTML::ParserEntities and TelNetWordNet packages are not cohesive from the point of view of inter-class references – since the heads of HTML::ParserEntities and TelNetWordNet blueprints contain respectively only two and one gray boxes.

**Internal Interconnected Packages.** In addition we see in Figure 9 that Kernel and Url packages contain classes that are tightly inter-referenced – since there are a lot of gray boxes within the head of Kernel and Url package blueprints. For example, within Url package, the class Url refers to almost all classes of its package and it does not refer to any class outside the Url package (see the column of Url class in the blueprint body).

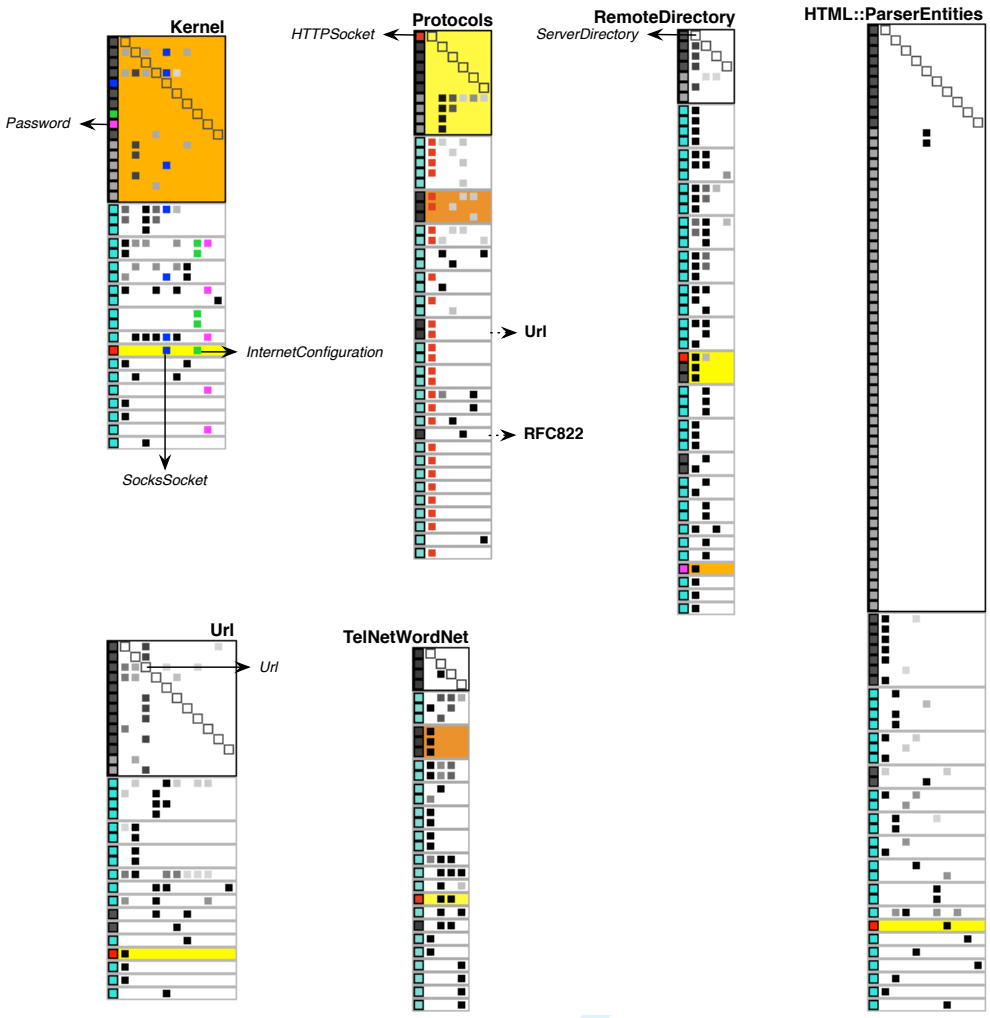



Figure 9. Outgoing reference blueprints of some packages of the Network application. In this view, the Kernel package was selected in orange, surfaces with Protocols package are highlighted in yellow, class HTTPSocket in red, class SocksSocket in blue, class InternetConfiguration in green, and class Password in fuchsia.

Similarly, we can see that RemoteDirectory and Protocols packages are less cohesive than Kernel and Url packages but more cohesive than HTML::ParserEntities and TelNetWordNet packages. It is worth to note that in the Protocols package, all internally referenced classes are classes that do not reference other classes – since all gray boxes within the head of the Protocols blueprint are under the head diagonal.

### B. Incoming Reference Package Blueprint Analysis

Incoming reference package blueprint is similar to outgoing reference package blueprint we described in previous sections. The difference between them is that the incoming reference package blueprint shows the package's relationships with its users while outgoing reference package blueprint shows the package's relationships with the packages it uses.

Figure 10 shows the incoming reference blueprints for Network's packages where only references within the Network application are taken in account (*i.e.*, references from packages outside Network are not shown). In this figure, surfaces of RemoteDirectory package are highlighted in green and those of TelNetWordNet in orange.

Figure 10 shows 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 package Kernel is referenced by only four packages within Network: Protocols, TelNetWordNet, Kernel::Test and RemoteDirectory. Protocols package heavily refers to Kernel package: the Kernel package incoming reference blueprint shows that the surface denoted to Protocols represents 4 external referencing classes. This means that there are 4 classes of Protocols package that refer to Kernel classes. .

Since Kernel package is heavily referenced by the core package Protocols (and that as already explained in Section VII-A Kernel does not refer to packages within Network), Kernel represents the basic package within Network.

**Most referenced package class.** Within this package, Kernel, the most referenced classes (*i.e.*, dominant referenced classes) are NetNameResolver and Socket (see the number of gray boxes within the classes rows of Kernel blueprint). Since the nodes of Socket class, within the body surfaces of Kernel blueprint,

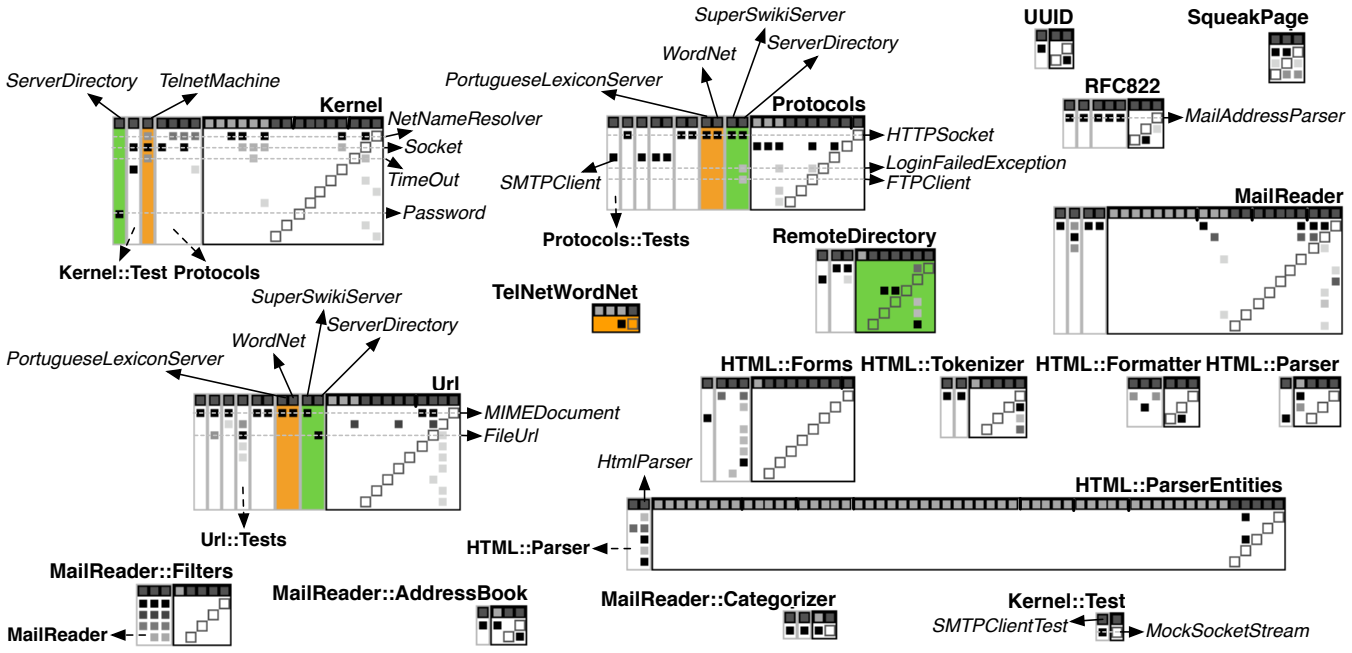



Figure 10. Incoming Reference global view in Network application. In this view, the TelNetWordNet package was selected in orange, surfaces with RemoteDirectory package are highlighted in green.

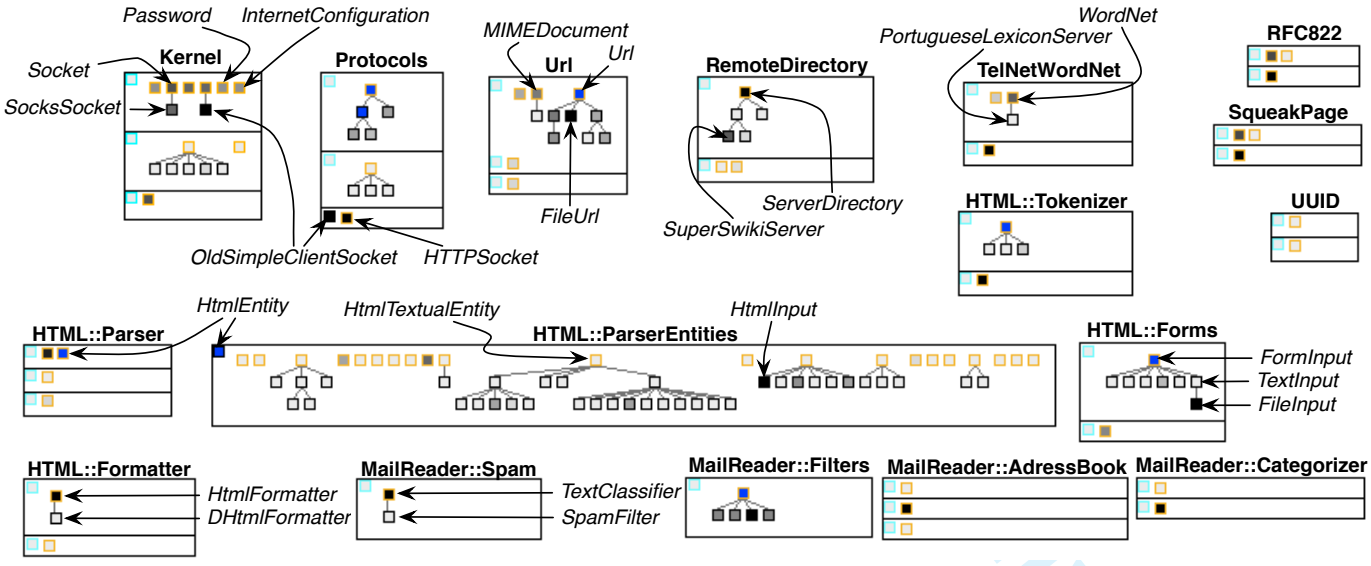



Figure 11. Inheritance global view in Network application

are darker than those of NetNameResolver, thus the former has a bigger number of incoming references than the latter.

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.

**Leaf packages.** Figure 10 clearly shows Network leaf packages (*i.e.*, packages which are referenced by only one package) such as MailReader::Filters which is only referenced by MailReader, or HTML::ParserEntities by HTML::Parser. 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.

**Low test coverage.** During our analysis we found that almost no class is tested. For Protocols package, only the class SMTPClient has incoming references from a test class within Protocols::Tests, while the most referenced class HTTPSocket has none (see the surface denoted by Protocols::Tests within the package blueprint of Protocols). Also for Url package, only four classes from twelve have incoming references from a test class within Url::Tests. Similarly for Kernel package, only two classes have incoming references from Kernel::Test and the class NetNameResolver has none. We thus deduce that the core

packages of Network are not well tested, particularly their most important classes (classes which are heavily referenced from other packages) are not tested.

**Internal/No Internal use.** Package Blueprint stresses the different nature of packages. At first, we was surprised to see that some packages contain classes without any reference among them, while they are heavily referenced by external classes. For example, the reference package blueprint of MailReader::Filters (Figure 10) shows that it contains classes without any package internal reference – since there is no gray boxes within the head surface. On the other hand, almost all classes of MailReader::Filters (four classes from five) are referenced by classes into the package MailReader (see the surface denoted by MailReader within the package blueprint of MailReader::Filter). The package HTML::Forms also presents such characteristics.

During our inspection, we found that these packages are defined around class inheritance instead of inter-class references: HTML::Forms is defined around the inheritance hierarchy of the class FormInput; similarly the package MailReader::Filters does (see inheritance package blueprint in Figure 11).

**Co-Referencers.** Figure 10 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.

Looking more closely at the referenced packages (in Kernel, Protocols and Url blueprints), we see that the similarity between RemoteDirectory and TelNetWordNet is improper at the class granularity level: the referencing classes of RemoteDirectory and TelNetWordNet packages do not reference the same classes within the cited referenced packages.

Except for Protocols package, the classes ServerDirectory and SuperSwikiServer of RemoteDirectory package, PortugueseLexiconServer and WordNet of TelNetWordNet package, all refer to the class HTTPSocket.

For Url package, the classes SuperSwikiServer, PortugueseLexiconServer and WordNet refer to the class MIMEDocument class, while ServerDirectory refers to FileUrl.

However, we see that the classes PortugueseLexiconServer and WordNet of TelNetWordNet package, both refer to the same set of classes (HTTPSocket within Protocols and MIMEDocument within Url). It is worth to note that WordNet is a superclass of PortugueseLexiconServer (we can see that in the inheritance blueprint of TelNetWordNet, Figure 11). Thus PortugueseLexiconServer inherits the behavior of WordNet. We think that it is a design defect that a subclass references the same set of classes as its superclass.

### C. Inheritance Package Blueprint Overview

Finally during our case studies, thanks to the inheritance package blueprint, we identified a few remarkable usage patterns: a package can mainly contain big inheritance hierarchies (potentially a single one); classes in a package may inherit from superclasses within the application itself or from frameworks or the base system; or a package can specialize functionality and have few internal inheritance relationships.

**Mispackaged inheritance root.** Figure 11 shows all the inheritance package blueprints of the Network subsystem in Squeak. It shows that there are only two places where classes inherit from classes within the Network subsystem scope: HtmlEntity and OldSimpleClientSocket. HtmlEntity is defined in HTML::Parser package and directly inherited by a lot of classes within HTML::ParserEntities package; OldSimpleClientSocket is defined in Kernel package and inherited by the class HTTPSocket within Protocols package. Note however that HtmlEntity class has blue fill color; this indicates that it is an abstract class.

Clicking on the HtmlEntity box, we can see that it is defined in the HTML::Parser package, away of all its subclasses defined in HTML::ParserEntities. We consider that it is defined in the wrong package.

**Heavy inheritance structured packages.** We can immediately spot that some packages are heavily structured around inheritance. Examples are: the package HTML::ParserEntities where the main class, in terms of inheritance, is HtmlTextualEntity; the package Url is structured around Url class which is internally inherited by almost all Url package classes; HTML::Forms or MailReader::Filters where both define a single hierarchy.

**Heavy referencing classes.** The overview also shows classes doing a lot of references (indicated as black boxes) such as HTTPSocket in Protocols package, HtmlFormatter in HTML::Formatter package, HtmlInput in HTML::ParserEntities package, FileInput in HTML::Forms package and TextClassifier in MailReader::Spam package.

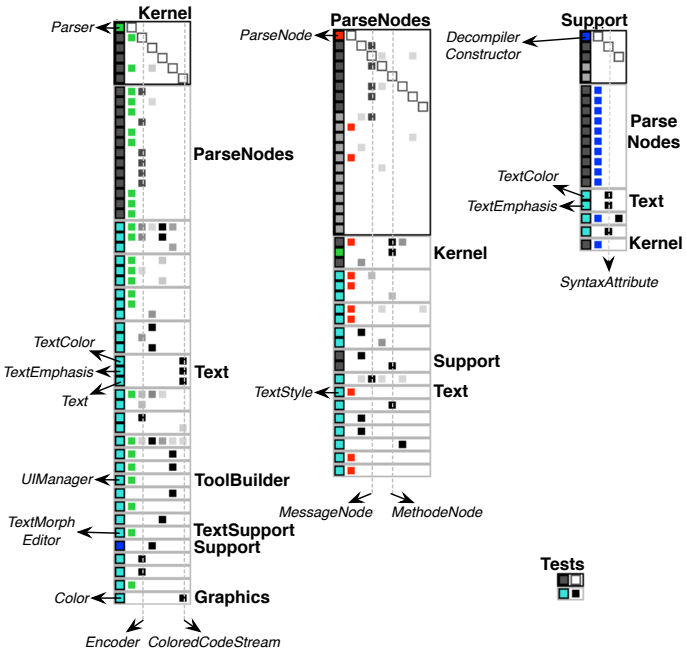
However, in the context of inheritance, we should pay attention to the fact that all the subclasses of a class inherit its behavior and references. The case of HtmlInput in HTML::ParserEntities package and FileInput in HTML::Forms package is interesting: while they are inheritance leaves, they are darker than other classes, in particularly than their superclasses, which means that they make direct references radically more than their superclasses; this indicates that such classes are complex and may heavily specialize their superclass inherited behavior.

This case is the reverse of that of DHtmlFormatter in HTML::Formatter package and SpamFilter in MailReader::Spam package – since these classes do direct references radically less than their superclasses.

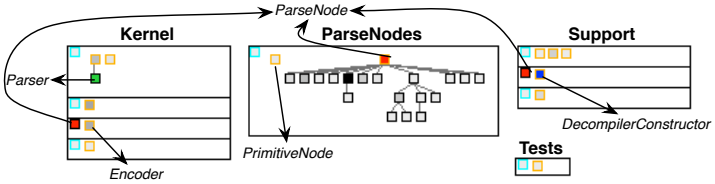
### D. The views together

While the views are simple, they convey powerful information. With inheritance package blueprints, we can see that the percentage of black-bordered boxes reveals the amount of internal reuse. Orange-bordered classes that inherit from a cyan class indicate reuse of functionality from outside the application. Note that this is different from many orange-bordered classes inheriting from a black-bordered one (like with HtmlEntity in HTML::ParserEntities), since a lot of classes inherit from Object and indeed do not share the same domain. In contrast, inheriting from HtmlEntity clearly reuses its domain.

In addition to that, inheritance package blueprint is an interesting complementary view to reference package blueprint.




(a) Outgoing Reference Blueprints for Compiler packages




(b) Inheritance Blueprints for Compiler packages

Figure 12. Global view in Compiler application. In this view, the class Parser is highlighted in green, the class ParseNode in red and the class DecompilerConstructor in blue.

For example, in Section VII-A we proposed to move the classes InternetConfiguration and SocksSocket from Kernel package to Protocols package – since these classes are not referenced within Kernel and reference the class HTTPSocket defined in Protocols package. The inheritance package blueprint of Kernel package (Figure 11) shows that InternetConfiguration class has no inheritance relationship within Kernel, thus moving InternetConfiguration to Protocols package will not break any inheritance hierarchy within the package. Package Blueprint also shows that SocksSocket class inherits from Socket class within Kernel. Indeed the package Protocols has an inheritance relationship with Kernel package (HTTPSocket in Protocols inherits from OldSimpleClientSocket in Kernel) that does not affect SocksSocket. Moving SocksSocket to Protocols package will then not change the inheritance layering within Network application.

## VIII. User Case Study on Squeak Compiler

We performed a controlled user study to assess whether developers could easily use the blueprints. The case study we proposed is the Squeak Compiler application, which is composed of 4 packages (Kernel, ParseNodes, Support and Tests) containing a total of 33 classes (Figure 12). We chose this case as we are familiar with the Squeak compiler and we developed the next compiler. So we can better appreciate the information that the view provides. As in addition, compilers are systems that every one knows more or less, and, it is easier for testers to make hypotheses. Another argument is that compilers generally contain interesting patterns and are not too small nor too big in terms of size and complexity.

### A. Experimental Setup

We first explained the visualization to the testers and gave them the paper [12] as well as related slides. Then we showed them how to use our tool for detecting patterns in the Network application. The demo helps them learn how to use the views. Such demo is important since it shows step by step how to quickly get information from the views.

To define the questions, we have tested them internally to know if they are understandable and meaningful. Hereafter are the questions with comments on the rationale behind them:

1) *Can you identify the main abstractions/classes of each package?*
   With this question we want to know if the reader can quickly identify the main entities, and learn if they are meaningful.
   We know that the main classes in Compiler are: the class ParseNode defined in ParseNodes package, which is the top super class for all the parser node classes. It is also the super class of the classes Encoder and DecompilerConstructor (see Figure 12 (b)). The classes Parser and Encoder in Kernel package and DecompilerConstructor in Support package are main classes in Compiler. These classes uses heavily the ParseNodes classes and are the responsible of code parsing and compiling.
2) *Can you identify how these main classes interact within the package and within the application? Are there classes doing most of the internal/external references?*
   This question checks if testers get how a package blueprint helps them get the relationship between packages. The user learns how to select classes and find if they are referenced or if they make references.
   It was clear for us that the ParseNode class and its subclasses are heavily referenced and used from the classes DecompilerConstructor, Parser and Encoder.
3) *How would you qualify the references from MessageNode class? Compare it to MethodNode?*
   This question checks if the testers are focusing on the understanding of a single package and comparing its classes.
   We know that MessageNode is the main referencing class in ParseNodes package: it refers and uses 5 classes of ParseNodes package and does few references to classes outside ParseNodes. While MethodNode class does not refer to any class in ParseNodes package and does many references to classes outside this package.
4) *How would you qualify the cohesion of Support package?*
   This question brings the user to focus on the head of the package blueprint as it gives information about the cohesion among enclosed classes. We check if the testers have understood that.

5) *Do you identify some misplaced dependencies with packages outside the compiler application?*
By inspecting Compiler package view, we noticed several misplaced dependencies and we want to know if the user can find them. Those misplaced dependencies are: in Kernel package, the class ColoredCodeStream references classes Text, TextColor and TextEmphasis in Text package; the class Parser references TextMorphEditor class in TextSupport package and UIManager class in ToolBuilder package.
6) *Under the assumption that a package containing classes referenced by other packages should be loaded first, can you identify a loading order for the application?*
We noticed that the three packages of the Compiler application depend on each other cyclicly, this just by clicking on each of them in the view. Does the user easily capture this?
7) *Using the inheritance view, what can you say about the shape of the ParseNodes package?*
The parse node classes belong to the same hierarchy tree and are defined in one package. This is the common way of declaring a parser tree. It allows one to define visitors to walk the tree and it is easier to annotate it. We would know if the user arrive to this conclusion.
8) *Can you tell us something about the class ParseNode hierarchy?*
We took the strangest things in the hierarchical view of the compiler to see whether the user can spot it out or not. All nodes inherit from ParserNode class except for PrimitiveNode class. This is not a good design since the node should be polymorphic to ParseNode.
9) *Do you think that you would have got the answers to our question in the time allocated without the help of visualization?*
We want to know if the user finds our views handy to understand the compiler application. We also want to have suggestion to improve the usability of the view.

### B. Results

The case study was conducted with 20 people, from master students to experienced researchers, with various programming skills and experience with software projects. We gave them a limited amount of time: one hour to perform the study.

For the first question, most of users identified the classes ParseNode and MessageNode in ParseNodes package, Parser in Kernel package, and DecompilerConstructor in Support package, as the main classes of the packages. This was expected: they did not identified Encoder class as a main class. They based their conclusions on the quantity of references and inheritance relationships with these classes, and whether the referenced classes were within or outside of the application boundary. The remainder of users had exactly our estimation: they also identified Encoder as a main class in Kernel package – since it is referenced by the main class Parser.

For the second and third questions, all users have been able to correctly answer those questions.

For the 4th question, about 50% (11 users) of testers have identified the Support package cohesion as very low. Two users have identified Support package as very cohesive, without justifying their answers. The remainder of the users (9) skipped this question. We think that we had to define what we mean by package cohesion. On another hand, in the first version of the Package Blueprint [12], packages that have not internal references do not have a head surface. Some users have found that such a visual mapping is not helpful, since they cannot analyze what they cannot see (*e.g.*, the package size, classes and internal references). In this version, we optimized the visualization in a way that users always see package size, classes and internal references, if any.

For the 5th question, only five users were not able to identify the misplaced dependencies that we identified. Some of them have declared that they did not understand the question. Similarly for the 6th question, most of users have found that the three packages (Kernel, ParseNodes and Support) depend on each other cyclicly and we cannot know the loading order of Compiler packages. Some of those users added that it is more probable to load the package ParseNodes at first – since other packages depend heavily on ParseNodes and extend its class ParseNode – this was a good answer. A couple of users said that we can easily and quickly answer this question by doing an automatic dependency analysis using Smalltalk cross referencer, rather than using the Package Blueprint.

For the 7th and 8th questions, more than 50% (12 users) captured that ParseNodes package contains a single domain defined by the class ParseNode. They found that the hierarchy of ParseNode class is coherent. They also spotted that it is not normal that the class PrimitiveNode does not belong to the hierarchy of ParseNode class. The remainder of users skipped these questions and mentioned that the questions are confusing.

Almost all the users concluded positively to the last question indicating that the visualization was useful. They underlined that the package blueprint was helpful to extract information about the Compiler application in a very short time and indicate that they would need really more time to do the same thing without the package blueprint.

Some of the testers proposed enhancements such as adding a fly-by-help to explain the nodes to ease the learning curve of the visualization. All those propositions were integrated in the package blueprint new version presented in this paper.

## IX. Evaluation and Discussion

### A. Evaluation

As illustrated in Section VII, package blueprints allow software engineer to extract information from the internal structure of a package, its clients as well as packages it uses. Now we revisit some of the information that we listed in the beginning of the paper.

*Size:* Package Blueprint highlights the complexity of the observed package in several dimensions. For outgoing reference blueprints, the height of the body indicates the amount of external classes referenced, whereas the number of surfaces shows the number of referenced packages. Each individual surface height shows how many classes are referenced in the corresponding package. This gives us an estimate of the coupling between the package and this surface; to further

evaluate the coupling strength, we should also look at the intensity of referencing classes in the surface because it represents the number of references. In addition, surface width indicates the number of referencing classes.

Combined together these visual properties offer a quick impression not just about the visualized package, but also about its classes: a thin package with a long body depends on a lot of classes because of few internal classes. If moreover the blueprint is heavily lined, *i.e.*, it references a lot of packages, so some of its referencing classes may be complex and fragile.

The same situation occurs with incoming reference and inheritance blueprints but from the view point of referencing packages/classes and inheritance relationships.

*Central or Peripheral:* For outgoing reference blueprints, by looking at the border color of external classes (cyan or black), we can easily see if a package depends a lot on the framework or on the application. Also, through incoming reference blueprints, we can see if a package is used by different subsystems (central) or just by specific ones (peripheral).

*Cohesion and Coupling:* package blueprint also makes it possible to roughly compare how several packages are coupled with the observed one: larger surfaces indicate coupling to more classes and are positioned nearer to the head surface, while surfaces with more darker class squares represent packages which are more coupled in term of sheer number of references. We can also estimate cohesion by comparing internal coupling (size and overall intensity of the head surface) and external coupling.

*Co-changes and Impact Analysis:* Because package blueprint details how packages depend on each other, it hints at the fragility of the observed package to changes. Selecting a package or a class highlights surfaces or classes that reference the selected entity and are thus sensitive to its changes.

### B. Discussion

Our approach has worked well on our case studies (it helped us to get important structural information efficiently). It should be noted that we were *not* familiar with the case studies such as the Network application before applying our approach. We have been able to locate many conceptual bugs. Our first evaluation with end-users is also promising, even if we are aware that the number of participants was not significant for drawing larger conclusions.

*In conjunction with other tools:* We do not consider that package blueprint should be used in isolation. In our recent work on remodularisation, we use DSM to spot cyclic dependencies, 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 [**?**] to understand how a property (such as developers) spreads on a set of packages.

Let us now discuss the visualization choices we made.

*Position Choices:* We grouped the internal references at the top of the package blueprint, then ordered the surfaces from the ones having the most external references at the top to the least at the bottom; inside a surface, we also ordered the rows from the most referencing ones to the least. 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 top elements than to the bottom ones. We also tried to layout surfaces compactly so that we can easily move them. According to this principle, internal classes that do not do any reference are placed in the bottom of the left most column in 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 [18] to group lines having similar referencing classes. However the resulting views were not as meaningful as with a simple ordering. We thus plan to use seriation to group packages having similar surfaces *i.e.*, packages using similar packages.

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 [12] 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. This is a bit limited in inheritance blueprints because we do not distinguish well the true root classes —*e.g.*, Object or Model in Squeak — from other classes that are packaged outside the analyzed application.

We found it really effective to color surfaces 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.

*Shapes:* For the time being we represent the classes 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.

*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 [20].

*Outgoing vs. incoming:* Having two views showing different flows of relationships 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.

## X. Related Work

Several works provide or visualize information on packages. Many of these approaches treat software co-change, looking at coupling from a temporal perspective, whereas in this paper we focus on the static structure of relationships [5], [13], [14], [28], [32], [34].

Abdeen *et al.* provide the Package Fingerprint visualization [1]. They focus on the package's contextual cohesion, coupling and the co-use and co-usage of internal classes. Although that

Package Fingerprints are good for fast overview on packages, they do not provide a good map for internal references and inheritance relationships.

Lungu *et al.* guide exploration of nested packages based on patterns in the package nesting and in the dependencies between packages [20]; their work is integrated in Softwarenaut and adapted to system discovery.

Sangal *et al.* adapt the dependency structure matrix from the domain of process management to analyze architectural dependencies in software [27]; while the dependency structure matrix looks like the package blueprint, it has no visual semantics.

Storey *et al.* offer multiple top-down views of an application, but these views do not scale very well with the number of relationships [29].

Ducasse *et al.* present Butterfly, a radar-based visualization that summarizes incoming and outcoming relationships for a package [11], but only gives a high-level client/provider trend.

In a similar approach, Pzinger *et al.* use Kiviat diagrams to present the evolution of package metrics [24].

Chuah and Eick use rich glyphs to characterize software artefacts and their evolution (number of bugs, number of deleted lines, kind of language...) [7]. 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 [8]. The radar view is effective at identifying outliers but does not detail the structure.

Those approaches, while valuable, fall short of providing a fine-grained view of packages that would help understanding the package shapes (the number of classes it defines, the inheritance relationships of the internal classes, how the internal classes inherit from external ones...) and support the identification of their roles within an application.

## XI. Conclusion

In this paper, we tackled the problem of understanding the details of packages with a focus on their relationships. We described package blueprint, a visual approach for understanding package relationships. Package Blueprint is a compact visualization supporting large overview without losing the essential details (references and inheritance among classes). 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 principles and preattentive processing. We successfully applied the visualization to several large applications and we have been able to point out core classes, misplaced ones, and badly designed packages. We also introduced interactivity to help the user focus and navigate within the system. We validated the package blueprint usability by conducting tests with several independent software maintainers. The results were positive, even if the numbers of testers was low (20). Testers concluded that the package blueprint is useful for understanding and analyzing packages. They specially underlined that the package blueprint helps them to reduce the time and effort during maintenance tasks.

## References

[1] H. Abdeen, I. Alloui, S. Ducasse, D. Pollet, and M. Suen. Package reference fingerprint: a rich and compact visualization to understand package relationships. In *CSMR*, 213–222. IEEE Comp. Soc., 2008.
[2] N. Anquetil and T. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *WCRE*, 235–255, 1999.
[3] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE TSE*, 30(8):491–506, 2004.
[4] J. Bertin. *Semiology of Graphics*. University of Wisconsin, 1983.
[5] D. Beyer. Co-change visualization. In *ICSM, Industrial and Tool vol.*, 89–92, 2005.
[6] L. C. Briand, J. W. Daly, and J. K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE TSE*, 25(1):91–121, 1999.
[7] M. C. Chuah and S. G. Eick. Information rich glyphs for software management data. *Comp. Graphics and Applications*, 18(4):24–29, 1998.
[8] M. D'Ambros and M. Lanza. Reverse engineering with logical coupling. In *WCRE*, 189 – 198, 2006.
[9] X. Dong and M. Godfrey. System-level usage dependency analysis of object-oriented systems. In *ICSM*, 375–384, Oct. 2007.
[10] S. Ducasse, T. Gîrba, and A. Kuhn. Distribution Map In *ICSM*, 203–214. IEEE Comp. Soc., 2006.
[11] S. Ducasse, M. Lanza, and L. Ponisio. Butterflies: A visual approach to characterize packages. In *METRICS*, 70–77. IEEE Comp. Soc., 2005.
[12] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM*, 94–103, 2007.
[13] S. Eick, T. Graves, A. Karr, A. Mockus, and P. Schuster. Visualizing software changes. *IEEE TSE*, 28(4):396–412, 2002.
[14] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE*, 387–396, 2004. IEEE Comp. Soc.
[15] T. Gîrba, A. Kuhn, M. Seeberger, and S. Ducasse. How developers drive software evolution. In *IWPSE 2005*, 113–122. IEEE Comp. Soc., 2005.
[16] C. G. Healey. Visualization of multivariate data using preattentive processing. Ms. thesis, CS Dep, Bristish Columbia Univ., 1992.
[17] C. G. Healey, K. S. Booth, and E. J. T. Harnessing preattentive processes for multivariate data visualization. In *Graphics Information*, 1993.
[18] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.
[19] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.
[20] M. Lungu, M. Lanza, and T. Gîrba. Package patterns for visual architecture recovery. In *CSMR*, 185–196, 2006. IEEE Comp. Soc.
[21] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *ICSM*, 1999. IEEE Comp. Soc.
[22] R. C. Martin. Design principles and design patterns, 2000.
[23] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE TSE*, 32(3):193–208, 2006.
[24] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *SoftVis*, 67–75, 2005.
[25] D. Pollet and S. Ducasse. Software Architecture Reconstruction: A Process-Oriented Taxonomy. *IEEE TSE*. To appear.
[26] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *SMEF*, 91–103, 2006.
[27] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *OOPSLA*, 167–176, 2005.
[28] M.-A. D. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis*, ACM, 2005.
[29] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Müller. On integrating visualization techniques for effective software exploration. In *InfoVis '97*, 38–48. IEEE Comp. Soc., 1997.
[30] A. Treisman. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing*, 31(2):156–177, 1985.
[31] E. R. Tufte. *The Visual Display of Quantitative Information*. Graphics, 2nd edition, 2001.
[32] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: visualization of code evolution. In *Softviz 2005*, 47–56, 2005.
[33] C. Ware. *Information visualization: perception for design*. Morgan Kaufmann Publishers, 2000.
[34] X. Xie, D. Poshyvanyk, and A. Marcus. Visualization of CVS repository information. In *WCRE*, 231–242, 2006. IEEE Comp. Soc.

# Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships

Accepted at ICSM'2007: International Conference on Software Maintenance

Stéphane Ducasse* Damien Pollet Mathieu Suen Hani Abdeen Ilham Alloui
Language and Software Evolution Group — Université de Savoie, France



## Abstract

*Large object-oriented applications are structured over large number of packages. Packages are important but complex structural entities that may be difficult to understand since they play different development roles (i.e., class containers, code ownership basic structure, architectural elements...). Maintainers of large applications face the problem of understanding how packages are structured in general and how they relate to each others. In this paper, we present a compact visualization, named Package Surface Blueprint, that qualifies the relationships that a package has with its neighbours. A Package Surface Blueprint represents packages around the notion of package surfaces: groups of relationships according to the packages they refer to. We present two specific views one stressing the references made by a package and another showing the inheritance structure of a package. We applied the visualization on two large case studies: ArgoUML and Squeak.*

*This paper makes heavy use of colors in the figures. Please obtain and read an online (colored) version of this paper to better understand the ideas presented in this paper.*

## 1 Introduction

To cope with the complexity of large software systems, applications are structured in subsystems or packages. It is now frequent to have large object-oriented applications structured over large number of packages. Ideally, packages should keep as less coupling and as much cohesion as possible [25, 5], but as systems inevitably become more complex, their modular structure must be maintained. It is thus useful to understand the concrete organization of packages and their relationships. Packages are important but complex structural entities that can be difficult to understand since they play different development roles (*i.e.*, class containers, code ownership basic structure, architectural elements...). Packages provide or require services. They can play core roles or contain accessory code features. Maintainers of large applications face the problem of understanding how packages are structured in general and how packages are in relation with each others in their provider/consumer roles. This problem was experienced first-hand by the first author while preparing the 3.9 release of Squeak, a large open-source Smalltalk [8]. In addition, approaches that support application remodularization [1, 20, 22] succeed in producing alternative views for system refactorings, but proposed changes remain difficult to understand and assess. There is a good support for the algorithmic parts but little support to understand their results. Hence it is difficult to assess the multiple solutions.

Several previous works provide information on packages and their relationships, by visualizing software artifacts, metrics, their structure or their evolution [6, 7, 10, 19, 23, 28]. However, while these approaches are valuable, they fall short of providing a fine-grained view of packages that would help understanding the package shapes (the number of classes it defines, the inheritance relationships of the internal classes, how the internal class inherit from external ones...) and help identifying their roles within an application.

In this paper, we propose Package Surface Blueprint, a compact visualization revealing package structure and relationships. A package blueprint is structured around the concept of surface, which represents the relationships between the observed package and its provider packages. The Package Surface Blueprint reveals the overall size and internal complexity of a package, as well as its relation with other packages, by showing the distribution of references to classes within and outside the observed package. We applied the Package Surface Blueprint to several large case studies namely Squeak the open-source Smalltalk comprising more than 2000 classes, ArgoUML and Azureus.

Sections 2 & 3 present the challenges in supporting package understanding, and summarize the properties wanted for effective visualizations. Section 4 presents the structuring principles of a package blueprint, which are then declined

*We gratefully acknowledge the financial support of the french ANR (National Research Agency) for the project "COOK: Réarchitecturisation des applications industrielles objets" (JC05 42872).

to support a reference view and an inheritance view in Section 5. Section 6 then describes some recurring patterns. In sections 7 & 8, we discuss our visualization and position it w.r.t. related work before concluding.

## 2 Challenges in Understanding Packages

Although languages such as Java offer a language mechanism for modelling the dependencies between packages (*i.e.*, via the import statement), this mechanism does not really support all the information that is important to understand a package. We present a coarse list of useful information to understand packages. Our goal here is to identify the challenges that maintainers are facing and not to define a list of all the problems that a particular solution should tackle.

**Size.** What is the general size of a package in terms of classes, inheritance definition, internal and external class references, imports, exports to other packages? For example, do we have only a few classes communicating with the rest of the system?

**Cohesion and coupling.** Transforming an application will follow natural boundaries defined by coupling and cohesion [5, 2]. Assessing these properties is then important.

**Central vs. Peripheral.** Two correlated pieces of information are important: (1) whether a package belongs to the core of an application or if it is more peripheral, and (2) whether a package provides or uses functionality.

**Developers vs. Team.** Knowing who are the developers and maintainers of the application and packages helps in understanding the architecture of the application and in qualifying package roles [13, 24]. Approaches such as the distribution map may help in this task [9].

In addition, packages reflect several organizations: they are units of code deployment, units of code ownership, can encode team structure, architecture and stratification. Good packages should be self-contained, or only have a few clear dependencies to other packages [5, 2, 18]. A package can interact with other ones in several ways: either as a provider, or as a consumer or both. In addition a package may have either a lot of references to other packages or only a couple of them. If it defines subclasses, those can form either a flat or deep subclass hierarchy. It can contain subpackages.

Figure 1 shows situations where the same group of classes can be dispatched. Note that for the purpose of illustration, Figure 1 only shows references but the same idea holds for inheritance between classes distributed in several packages. In both cases (a) and (b), there are only two packages but in case (a) most of the classes of P4 inherit directly from a class in P1 while in case (b) all the classes of P4 inherit internally from B2 which is a root of an inheritance hierarchy. Revealing this difference is important since we want to understand if we can change the relationships between P1 and P4 during a refactoring process. In cases (a) and (c), we have exactly the same relationships between classes but the package structure is different. As mentioned by R. Martin importing a class equals importing the complete package [21], therefore importing two classes from the same package is quite different from importing them from two different packages since in the latter case we import all the classes of the two packages.

Note that understanding packages is also important in the context of remodularization approaches [1, 20, 22]. There it is important to understand how the proposed remodularisation compares with the existing code. This problem is particularly stressed in presence of legacy applications that consist of thousands of classes and hundreds of packages.

## 3 Visualization Challenges

We researched the characteristics that an efficient visualization should hold [3, 30, 32]. As our focus is on providing a first impression of a package and its context, we want to exploit the gestalt principles of visualization and preattentive processing[1] as much as possible to help spotting important information [29, 14, 15, 32].

To support the understanding of packages, we want the visualization to highlight the characteristics of a package in terms of its internal size, internal and external references. In particular we want to spot classes or dependencies that stand out in a given package. We stress that our visualization should take into account the following properties:

**Good mapping to reality.** The visualization should offer a good representation of the situation that the maintainer can trust and from which it can draw and validate hypothesis.

We want the visualization to highlight the general tendency of a package in terms of its internal size, internal and external references. In particular we want to spot classes or dependencies that stand out in a given package.

**Scalability and simple navigation.** The maintainer should easily access the information. The visualization should

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

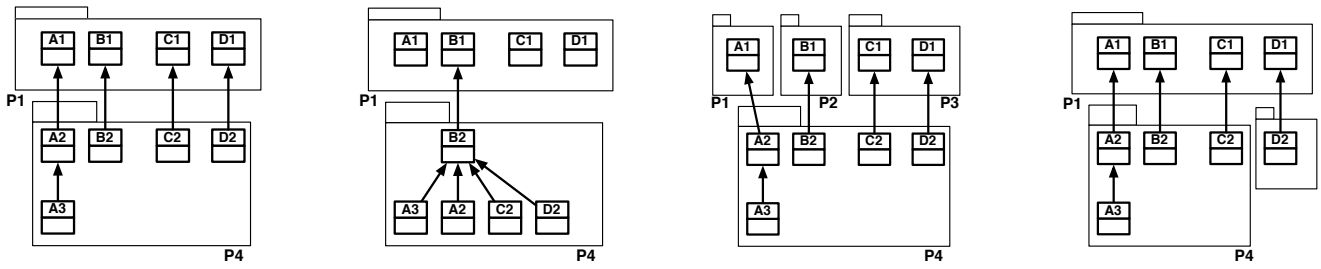



**Figure 1. Different package configurations over the same number of classes.**

scale *i.e.*, we should be able to have system overview as well as focusing on a particular package. We want a visualization that scales well with the number of packages and of dependencies, so we prefer to avoid depicting dependencies with graphs. Given that the graph will contain more than thousands of nodes and much more edges, this will result a unusable view [16].

**Low visual complexity.** By being regular and well structured, *i.e.*, reusing the same conventions of color or position, the visualization should help the maintainer to learn it and understand it. In addition, while the visualization should offer a lot of information, it should not be complex to analyze.

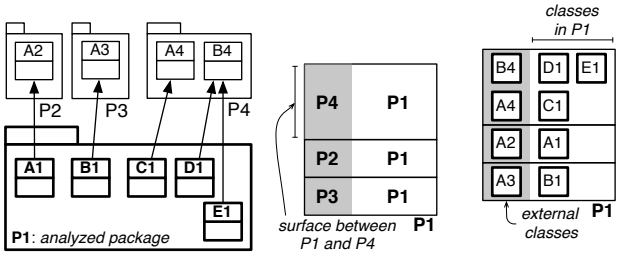
## 4 Package Surface Blueprints

A package blueprint represents how the package under analysis references other packages. Figure 2 presents the key principles of a Package Blueprint. These principles will be realized slightly differently when showing direct class references or inheritance relationships.

### 4.1 Basic Principles

The package blueprint visualization is structured around the "contact areas" between packages, that we name *surfaces*. A *surface* represents the conceptual interaction between the observed package and another package. In Figure 2(a) the package P1 is in relation with three packages P1, P2, and P4, via different relationships between its own classes and the classes present in the other packages, so it has three surfaces.

A package blueprint shows the observed package as a rectangle which is vertically subdivided by each of the package's surfaces. Each subdivision represents a surface between the observed package and a referenced package, and will be more or less tall, depending on the strength of the relation between the two packages. In Figure 2(b), the package blueprint of P1 is made from three stacked boxes because




(a) Analyzed package (b) Stacked surfaces (c) With classes

**Figure 2. Consider** P1 **that references four classes in three other packages (a). A blueprint shows the surfaces of the observed package as stacked subdivisions (b). Small boxes represent classes, either in the observed package (right white part) or in referenced packages (left gray part) (c).**

P1 references three other packages. The box of the surface between P1 and P4 is taller because P1 references more classes in P4 than in P2 or P3.

In each subdivision, we show the classes involved in the corresponding surface. By convention, we *always* show the classes in the referenced packages in the leftmost gray-colored column of each surface, and the classes of the observed package on the right. In Figure 2(c), the topmost surface shows that classes D1 and E1 reference class B4, and that C1 references A4. If many classes reference the same external class, we show them all in an horizontal row; we can thus assess the importance of an external class by looking at how many classes there is in the row: in Figure 2, the row of B4 stands out because the two referring classes D1 and E1 make it wider.

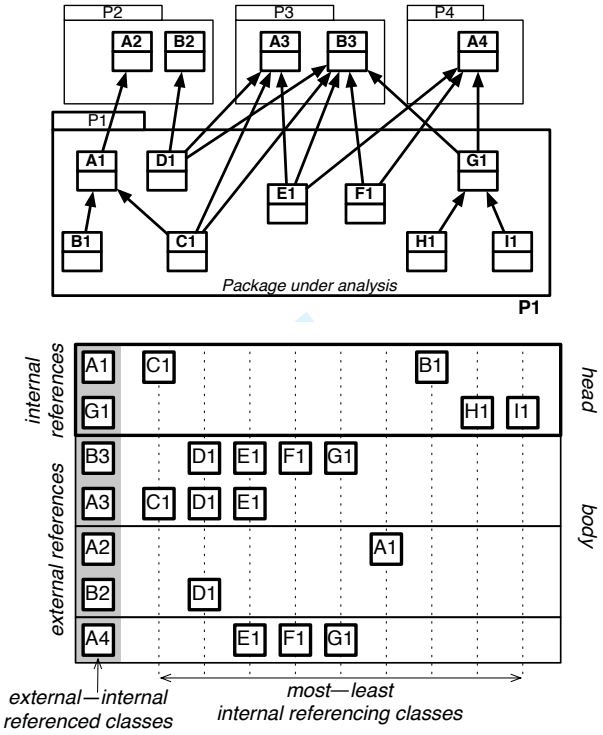



**Figure 3. Surface package blueprint detailed view.**

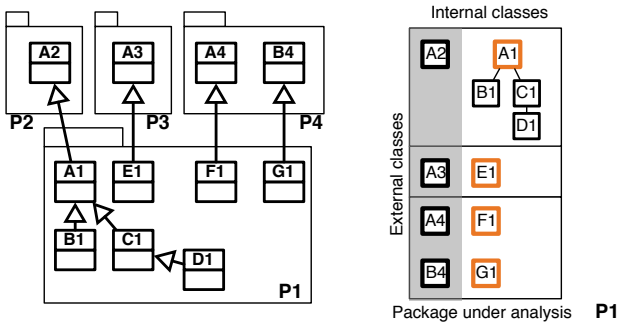



**Figure 4. Inheritance package surface blueprint. Orange bordered classes inherit from external classes directly.**

## 4.2 Detailed Explanation

To convey more information, we add variations to the basic layout described above, as illustrated in Figure 3.

**Internal References.** To support the understanding of references between classes inside the observed package, we add a particular surface with a thick border at the top of the blueprint. We name this surface the head of the blueprint, and the rest its body. In the head, the first column represents the internal classes that are referenced from within the package itself: here A1 and G1 are the classes referenced respectively by B1 and C1 and H1 and I1. The height of the head surface indicates the number of classes referenced within the package.

**Position.** Internal classes are arranged by columns: each column (after the leftmost one) refers to the same internal class for all the surfaces. The width of the surface indicates the number of referencing classes of the package. Figure 3 shows that class C1 internally references A1, and externally references A3 and B3.

We order classes in both horizontal and vertical direction to present important elements according to the (occidental) reading direction. Horizontally, we sort classes from left to right according to the number of *external* classes they reference from the whole package. Hence classes referencing the most occupy the nearest columns from the gray area.

We apply the same principle for the vertical ordering, both of surfaces within a blueprint, and of rows (*i.e.*, external classes) within a surface. Within a package, we position surfaces that reference the most classes the highest. Within a surface, we order external classes from the most referenced at the top, to the least referenced are at the bottom of the surface. This is why in Figure 3 the surface with P3 is the highest and why the surface with P2 is above P4, since there are more classes references from P2 than from P4.

**Color.** We want to distinguish referenced classes depending on whether they belong to a framework or the base system, or are within the scope of the application under study. When a referenced class is not part of the application we are currently analyzing, we color its border in cyan. In addition the color intensity of a node conveys the number of references it is doing: the darker the more references. Both intensity and horizontal position represent the number of references, but position is computed relative to the whole package, while intensity is relative to each surface. Thus, while classes on the left of surfaces will generally tend to be dark, a class that makes many references in the whole package but few in a particular surface will stand up in this surface since it will be light grey.

**The Case of Inheritance.** Up to now, we only discussed references, but inheritance is a really important structural relationship in object-oriented programming. We adapt the Package Surface Blueprint to offer a view specific to inheritance, as shown in Figure 4. In this variation, we consider only single inheritance so we don't need the head surface: we can display all classes and subclasses transitively inheriting from external classes on the same row. We distinguish the direct subclasses of external classes by showing them with

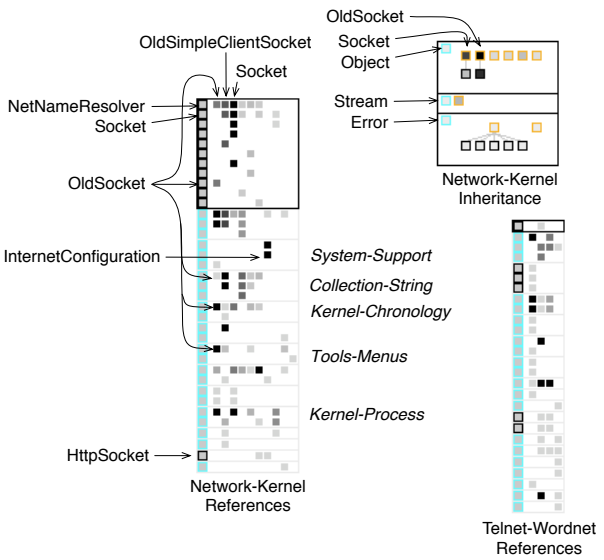



**Figure 5. Analysing the Network-Kernel Package.**

an orange border; indirect subclasses are black-bordered and arranged in trees under their superclass. In addition, root classes such as Object are filled in cyan and abstract classes in blue. In Figure 4, A1 inherits from A2 defined in package P2, while B1, C1, and D1 inherit from A1.

The fill color of classes in the inheritance view still represents the number of *references*, but relative to the *package* and not to the surface like in the references views. This makes it possible to correlate inheritance and references. For instance, the top-right view in Figure 5 shows that most references come from a subclass (Socket) of Object; in other cases, references might come from classes that are lower in the hierarchy as HTMLInput in Figure 6.

### 4.3 An Example: The Network Subsystem

We are now ready to have a deeper look at an example. The Squeak Network subsystem contains 178 classes and 26 packages — this package contains on the one hand a library and a set of applications such as a complete mail reader. The blueprint on the left in Figure 5 shows the references package blueprint of the Network-Kernel package in Squeak.

Glancing at it we see that the package blueprint of the Network-Kernel package has nearly a square top-red surface indicating that most internal classes are referenced internally. This conveys a first impression of the package's cohesion even if not really precise [5]. Contrast it with the package blueprint of the Telnet-Wordnet package which clearly shows little internal references.

We see that Network-Kernel is in relation with thirteen other packages. Most of the referenced classes are cyan, which means that they are not part of the network subsystem. What is striking is that all except one of the referenced classes are classes outside the application (see (HTTPSocket) in Figure 5). However, since the package is named *kernel*, it is strange that it refers to other classes from the same application, and especially only one. We see that half of the referred packages have strong references (indicated by their dark color).

Using the mouse and pointing at the box shows using a fly-by-help the class and package names (indicated in italics in Figure 5). The Tools-Menus surface indicates some improper layering. Indeed it shows that Network-Kernel is referencing UI classes via the package Tools-Menus which seems inappropriate. We learn that the class making the most internal references is named OldSocket; this same class also makes the most external references, to three packages (Collection-String, Tools-Menus, and Kernel-Chronology). The second most referencing class is named OldSimpleClientSocket. It is worth to notice that OldSocket is only referencing itself and that even OldSimpleClientSocket does not refer to it, so it could be removed from this package without problems. The third most referencing class is Socket. Having two classes named Socket and OldSocket clearly indicates that the package is in a transition phase where a new implementation has been supplanting an old one. We learn that the most internally referenced class is NetNameResolver and the second most is Socket. So this is a sign of good design since important domain classes are well used within the package.

The inheritance package blueprint shows that the Network-Kernel package is bound to three external packages containing the three superclasses Object, Error, and Stream. In addition the package, while inheriting a lot from external packages, is inheriting from the same class, here Object. The difference between the two main surfaces is interesting to discuss: the topmost surface shows that most of the classes are directly inheriting from one external superclass (here Object), while the second one shows that errors are specialized internally to the package. All in all, this makes sense and provides a good characterization of the package.

## 5 Packages Within Their Application

Understanding a package in isolation (mainly as a consumer) is interesting but lacks information about the overall context *i.e.*, is a selected class used by other packages? which packages is a selected surface about? As shown in the following subsections, our approach also supports the understanding of the situation of a class/package within the context of a complete application.

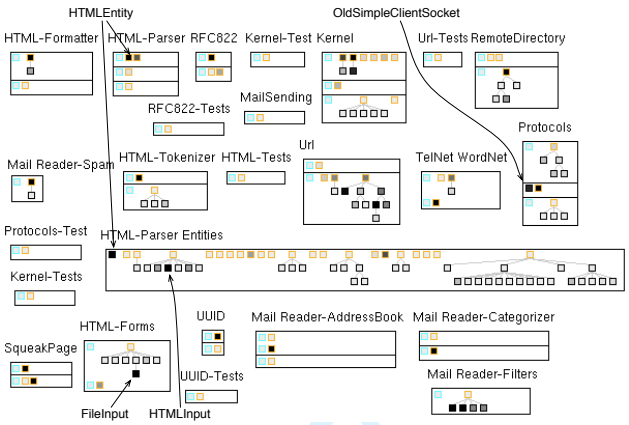



**Figure 6. Inheritance global view in Network**

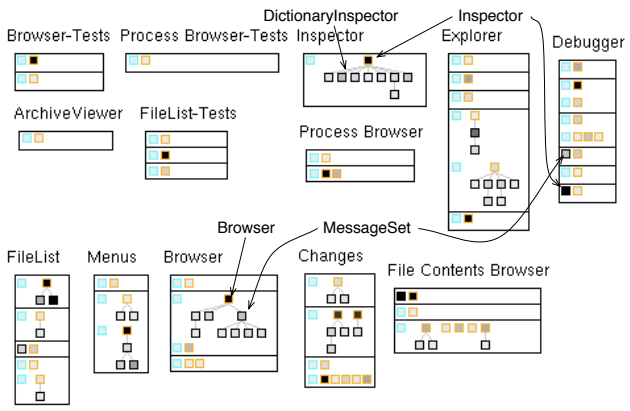



**Figure 7. Inheritance global view in Tools**

## 5.1 Inheritance package blueprint Overview

Overviewing all the package blueprints of an application gives a first impression of how the packages were built and structured. During our case studies, we identified a few remarkable usage patterns: a package can mainly contain big inheritance hierarchies (potentially a single one); classes in a package may inherit from superclasses within the application itself or from frameworks or the base system; or a package can specialize functionality and have few internal inheritance relationships.

**First Case: Squeak's Network.** For example, Figure 6 shows all the package blueprints of the Network subsystem in Squeak, which groups library and application classes. It shows that there are only two places where classes inherit from classes within the Network subsystem scope: HTMLEntity and OldSimpleClientSocket. Note however that OldSimpleClientSocket has a lighter shade of gray than HTMLEntity; this indicates that the former is not referencing other classes as much as the latter.

Clicking on the HTMLEntity box, we can see that it is defined in the Network-HTML-Parser package, away of all its subclasses, and then directly consider that it is defined in the wrong package. We can immediately spot that some packages are heavily structured around inheritance, like the package Network-HTML-Parser Entities or Network-Mail Reader-Filters which define a single hierarchy.

The overview also shows classes doing a lot of references (indicated as black boxes) such as HTMLEntity, FileInput and HTMLInput. However, in the context of inheritance, we should pay attention to the fact that all the subclasses of a class inherit its behavior and references. While we can spot classes doing a lot of references, the view does not convey the tree ordering so it is difficult to evaluate the subclasses of a given class. The case of FileInput is interesting: while it is a leaf in the inheritance tree, it makes a lot of direct references, indicating that the class is complex.

While the views are simple, they convey powerful information. If we analyze a bit, we can see that the percentage of black-bordered boxes reveals the amount of internal reuse. Orange-bordered classes that inherit from a cyan class indicate reuse of functionality from outside the application. Note that this is different from many orange-bordered classes inheriting from a black-bordered one (like with HTMLEntity in HTML-Parser Entities), since a lot of classes inherit from Object and indeed do not share the same domain. In contrast, inheriting from HTMLEntity clearly reuses its domain.

**Second Case: Squeak's Tools.** Figure 7 shows the blueprints of the Tools packages which contain all the Squeak development tools: code browsers, debuggers... Without going into details, we immediately see different shapes. Here, the blueprints are thinner but often higher, showing that there is less internal reuse than in Network. Note that even if the Tools packages contain a large set of development tools, inheritance is actually used to reuse abstractions: The blueprint of Tools-Browser shows that the class Browser, even if it defines a tool, is inherited several times. Other tools reuse the abstraction of Browser: for instance, its subclass MessageSet allows one to browse a group of methods and is reused and extended in Tools-Debugger.

The blueprint of Tools-Debugger shows an interesting shape: it is narrow and has a nearly flat inheritance hierarchy. Moreover, all its classes are inheriting from classes outside the package. Note that this behavior makes sense because the package aggregates functionality defined elsewhere, and the view easily reveals it. The package Tools-FileList defines a tool to browse external files and shows a similar shape.

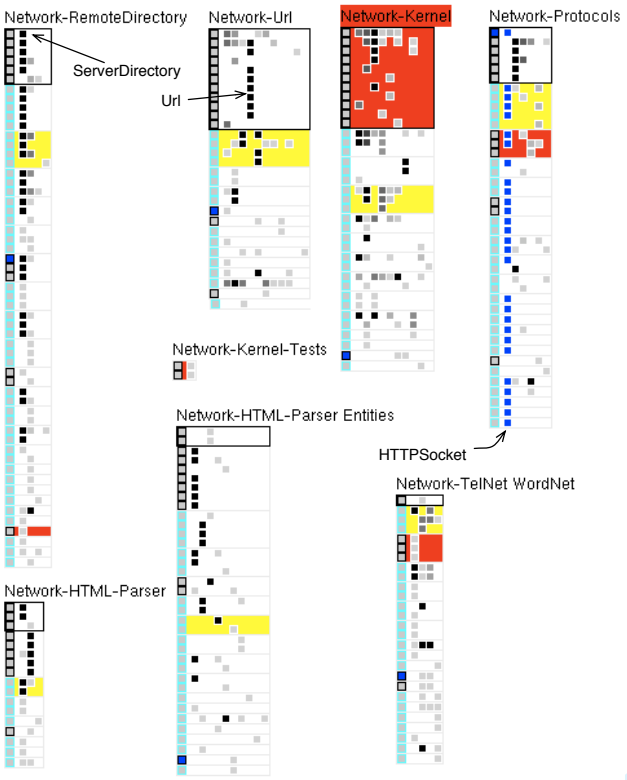



**Figure 8. In this view, the** Network-Kernel **package was selected in red, surfaces with** Collections-Strings **annotated in yellow, and class** HTTPSocket **selected in blue.**

### 5.2 Interactively Querying the Blueprint

The maintainer can also query the system by clicking either on a class or on a surface. This highlights in red all occurences of the class, or all surfaces referring to the same package. In addition, colors can be assigned to a surface to help the maintainer identify all the surfaces communicating with the same packages.

Figure 8 shows the blueprints of all the Network packages referencing and defining HTTPSocket. It is striking to see that HTTPSocket is a central class of the package Network-Protocols as it refers to most of the classes referred by that package. In addition, the surface referencing the package Collections-Strings is annotated in yellow and we see how all the packages refer to this package.

By clicking on the head surface, it gets colored in red and shows the package usage by coloring the surfaces referencing it in red. Figure 8 shows how the package Network-Kernel is used within the application.

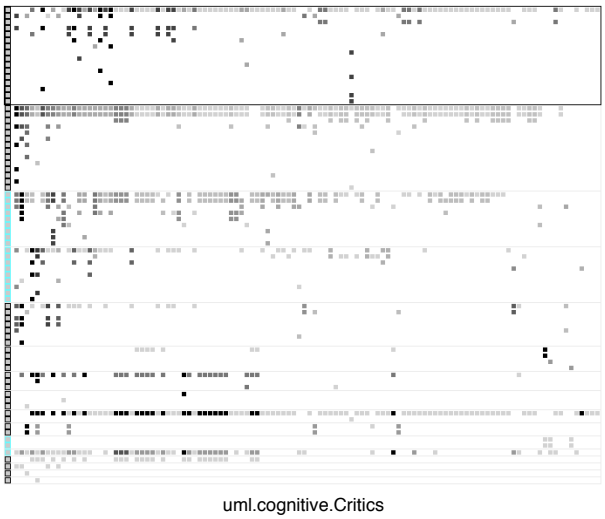



**Figure 9. A Sumo Blueprint: the** Critics **package in ArgoUML.**

## 6 Striking Shapes

While applying blueprints to large applications we identified some striking shapes that the blueprint, a surface or a class within a blueprint would produce. We present here the most frequent ones.

### 6.1 Shapes of Packages and Surfaces

**Sumo Package.** A very large and tall reference blueprint denotes a package that makes a lot of references from many classes. Figure 9 shows an example: the package Critics of ArgoUML that defines all the rules for assessing the quality of models.

**Small House Package.** A small inheritance blueprint with only a couple of surfaces and few inheritance hierarchies often denotes a package that offers a well packaged functionality, like Tools-Debugger or Tools-FileList (Figure 7). These blueprints are usually taller than larger.

**Flat Head Package.** A reference blueprint with a wide but flat head indicates limited internal references. Network-TelNet WordNet and Network-HTML-Parser Entities in Figure 8 are flat head blueprints.

**Exclusive External Referencer Package.** When the first column in a blueprint is almost or completely cyan, the package makes most or all of its external references to classes outside the scope of the analyzed application. These pack-

ages typically extend a framework or a core library; Network-Kernel in Figure 8 is an example.

**Loner Package.** A loner is a package that contains only a couple of classes. It is often containing a single test case class. The blueprint Network-Kernel-Tests in Figure 8 or Network-Mail Reader-Categorizer, Network-UUID, Network-Mail Reader-Spam of Figure 6 are loners. Some of these packages are clearly good candidates for remodularisation.

**Large External Surface.** When the topmost external surfaces are really large, like the four surfaces below the head in Figure 9, they identify packages that we must pay attention to, because changes in these external packages will very probably impact the package under analysis.

**Square Head Package.** A package that references all its own classes will have a blueprint with a square internal surface; this denotes a package that is quite cohesive. In Figure 8, Network-Kernel has a square head and appears to be relatively well packaged.

**Tower Package.** A reference blueprint with a small head and a thin body denotes a package with few internal references but that makes many external references. This package may not be cohesive but highly coupled with the external packages. The package peer in Azureus is an extreme of this shape, as shown in Figure 10. In Figure 8, Network-RemoteDirectory has a more cohesive head and three classes intensively referencing external packages.

### 6.2 Shapes of Classes

**Main Referencer Class.** A vertical alignment of dark squares in the body of a blueprint denotes a class that is responsible for many references to classes in other packages. The classes HTTPSocket and ServerDirectory are the main referencers in packages Network-Protocols and Network-RemoteDirectory; they are candidates to be central package classes (Figure 8).

**Figure 10.** Peer **in Azureus: a Tower Blueprint**

ui.swt.views.Peer

**Main Internal Referencer Class.** When vertical alignments are limited to the head, they reveal classes doing many internal and few external references. These classes often define the abstraction of the application. In Figure 8, the class Url only references classes within Network-Url.

**Omnipresent Referenced Class.** Classes of this kind are referenced by almost all the internal classes, and easily identifiable by filled rows in a surface. This makes sense for a facade class if it occurs a few times, but in ArgoUML we see this shape in most packages for Facade and Model (see Figure 9); we may thus assess that the Facade pattern is misused.

## 7 Evaluation and Discussion

### 7.1 Evaluation

The Package Surface Blueprint shows the internal number of classes as well as the number of classes externally referenced. Hence it conveys whether the package is using a lot of information or not.

**Size.** The Package Surface Blueprint shows the complexity of the observed package in several dimensions. The height of the body indicates the amount of external classes referenced, whereas the number of surfaces shows the number of referenced packages. The height of each individual surface shows how many classes are referenced in the corresponding package. This gives us an estimate of the coupling between the package and this surface; to further evaluate the coupling strength, we should also look at the intensity of referencing classes in the surface because it represents the number of references. In addition, the width of the surface indicates the number of referencing classes.

Those visual properties combine to give a quick impression not just about the visualized package, but also about its classes: a thin package with a long body depends on a lot of classes because of few internal classes. If moreover the blueprint is heavily lined, *i.e.*, it references a lot of packages, so some of its referencing classes may be complex and fragile.

**Central or Peripheral.** By looking at the border color of external classes (cyan or black), we can easily see if a package depends a lot on the framework or on the application. Also, by using the selection mechanism, we can interactively see if a package is imported by different subsystems (central) or just by specific ones (peripheral).

**Cohesion and Coupling.** The package blueprint also makes it possible to roughly compare how several packages are coupled with the observed one: larger surfaces indicate coupling to more classes and are positioned nearer to the head surface, while surfaces with more darker class squares

represent packages which are more coupled in term of sheer number of references. We can also estimate cohesion by comparing internal coupling (size and overall intensity of the head surface) and external coupling.

**Co-changes and Impact Analysis.** Because the package blueprint details how packages depend on each other, it hints at the fragility of the observed package to changes. Selecting a package or a class highlights surfaces or classes that reference the selected entity and are thus sensitive to its changes.

### 7.2 Discussion

Our approach has worked well on our case studies. It should be noted that we were *not* familiar with the case studies before applying our approach. We have been able to locate many conceptual bugs; for instance we found some clearly unwanted dependencies, like the package Network-Telnet WordNet referencing a class in the user interface framework. However one of our future works is to evaluate the view with users. The Package Surface Blueprint answers the main challenges proposed in Section 2 and in Section 3; we further intend to address some remaining challenges.

**Position Choices.** We grouped the internal references at the top of the package blueprint, then ordered the surfaces from the ones having the most external references at the top to the least at the bottom; inside a surface, we also ordered the rows from the most referencing ones to the least. 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 top elements than to the bottom ones. We also tried to layout surfaces compactly so that we can easily move them.

**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 [17] to group lines having similar referencing classes. However the resulting views were not as meaningful as with a simple ordering. We plan to use seriation to group packages having similar surfaces *i.e.*, packages using similar packages.

**Properties.** Instead of the number of references, we could map different properties to the color of classes and surfaces. This can create new striking shapes, adapted to a specific maintenance problem.

**Impact of Boundaries.** We color classes that do not belong to the application in cyan; this is a bit limiting since we do not distinguish well the true root classes —*e.g.*,Object or Model in Squeak— from the classes of a domain library that the analyzed application would extend. We found it really effective to color surfaces so that the user can interactively mark entities on which he wants to focus; this increases the usability of the tool and speeds up understanding packages.

**Shapes.** For the time being we represent the classes 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.

**Package Nesting.** Currently we do not support the nesting of packages. A solution like the one proposed by Lungu *et al.* seems complementary to our approach and interesting to deal with package nesting [19]. We also consider two types of relationships between packages (direct reference and inheritance); therefore we can extend our approach to other types of relationships like method invocation.

**Other Views.** So far we only presented blueprints to understand how a package was referencing or inheriting from other packages and classes. However we developed the reverse view: blueprints that present incoming references made by external classes on the observed package. Due to space limitation we did not present it. This information is useful when supporting package splitting or merging.

## 8 Related Works

Several works provide or visualize information on packages. Many of these approaches treat software co-change, looking at coupling from a temporal perspective, whereas in this paper we focus on the static structure of relationships [4, 11, 12, 27, 31, 33].

Lungu *et al.* guide exploration of nested packages based on patterns in the package nesting and in the dependencies between packages [19]; their work is integrated in Softwarenaut and adapted to system discovery.

Sangal *et al.* adapt the dependency structure matrix from the domain of process management to analyze architectural dependencies in software [26]; while the dependency structure matrix looks like the package blueprint, it has no visual semantics. Storey *et al.* offer multiple top-down views of an application, but these views do not scale very well with the number of relationships [28].

Ducasse *et al.* present Butterfly, a radar-based visualization that summarizes incoming and outcoming relationships for a package [10], but only gives a high-level client/provider trend. In a similar approach, Pzinger *et al.* use Kiviat diagrams to present the evolution of package metrics [23]. Chuah and Eick use rich glyphs to characterize software artefacts and their evolution (number of bugs, number of deleted lines, kind of language...) [6]. In particular, the timewheel 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 [7]. The radar view is effective at identifying outliers but does not detail structure.

Those approaches, while valuable, fall short of providing a fine-grained view of packages that would help understanding the package shapes (the number of classes it defines, the inheritance relationships of the internal classes, how the internal classes inherit from external ones,...) and support the identification of their roles within an application.

## 9 Conclusion

In this paper, we tackled the problem of understanding the details of package relationships. We described the Package Surface Blueprint, a visual approach for understanding package relationships. While designing Package Surface Blueprint, we tried to exploit gestalt visualization principles and preattentive processing.

We successfully applied the visualization to several large applications and we have been able to point out badly designed packages. To help users interpret views, we have identified a list of recurrent striking blueprint shapes. We also introduced interactivity to help the user focus and navigate within the system. We were however rather knowledgeable about both the visualization and the studied systems; in future work, we will validate the package blueprint usability by conducting tests with several independant software maintainers.

## References


[1] Anquetil and Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE*, 1999.

[2] Arisholm, Briand, and Foyen. Dynamic coupling measurement for object-oriented software. *IEEE TSE*, 30(8), 2004.

[3] Bertin. *Semiology of Graphics*. 1983.

[4] Beyer. Co-change visualization. In *ICSM*, 2005.

[5] Briand, Daly, and Wüst. A unified framework for coupling measurement in oo systems. *IEEE TSE*, 25(1), 1999.

[6] Chuah and Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4), July 1998.

[7] D'Ambros and Lanza. Reverse engineering with logical coupling. In *WCRE*, 2006.

[8] Denker and Ducasse. Software evolution from the field: an experience report from the Squeak maintainers. In *ERCIM Working Group on Soft. Evolution*, vol. 166 of *Electronic Notes in Theoretical Computer Science*, Jan. 2007.

[9] Ducasse, Gîrba, and Wuyts. Object-oriented legacy system trace-based logic testing. In *CSMR*, 2006.

[10] Ducasse, Lanza, and Ponisio. Butterflies: A visual approach to characterize packages. In *Int'l Soft. Metrics Symposium (METRICS)*, 2005.

[11] Eick, Graves, Karr, Mockus, and Schuster. Visualizing software changes. *IEEE TSE*, 28(4), 2002.

[12] Froehlich and Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE*, 2004.

[13] Gîrba, Kuhn, Seeberger, and Ducasse. How developers drive software evolution. In *Int'l Workshop on Principles of Soft. Evolution (IWPSE)*, 2005.

[14] Healey. Visualization of multivariate data using preattentive processing. Master's thesis, Univ. Bristish Columbia, 1992.

[15] Healey, Booth, and T. Harnessing preattentive processes for multivariate data visualization. In *Graphics Interface*, 1993.

[16] Herman, Melançon, and Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Trans. on Visualization and Comp. Graphics*, 6(1), 2000.

[17] Jain, Murty, and Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3), 1999.

[18] Lanza and Marinescu. *OO Metrics in Practice*. 2006.

[19] Lungu, Lanza, and Gîrba. Package patterns for visual architecture recovery. In *CSMR*, 2006.

[20] Mancoridis, Mitchell, Chen, and Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *ICSM*, 1999.

[21] Martin. Design principles and design patterns, 2000.

[22] Mitchell and Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE TSE*, 32(3), 2006.

[23] Pinzger, Gall, Fischer, and Lanza. Visualizing multiple evolution metrics. In *SoftVis*, May 2005.

[24] Pollet, Ducasse, Poyet, Alloui, Cîmpan, and Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In *CSMR*, Mar. 2007.

[25] Ponisio and Nierstrasz. Using context information to re-architect a system. In *Soft. Measurement Eur. Forum*, 2006.

[26] Sangal, Jordan, Sinha, and Jackson. Using dependency models to manage complex software architecture. In *OOPSLA*, 2005.

[27] Storey, Čubranić, and German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis*, 2005.

[28] Storey, Wong, Fracchia, and Müller. On integrating visualization techniques for effective software exploration. In *IEEE Symposium on Information Visualization (InfoVis)*, 1997.

[29] Treisman. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing*, 31(2), 1985.

[30] Tufte. *The Visual Display of Quantitative Information*. 2001.

[31] Voinea, Telea, and van Wijk. CVSscan: visualization of code evolution. In *SoftVis*, May 2005.

[32] Ware. *Information Visualization*. 2000.

[33] Xie, Poshyvanyk, and Marcus. Visualization of CVS repository information. In *WCRE*, 2006.

The work presented in this article extends our previous paper, published in ICSM'07 (Package Surface Blueprints: Visually Supporting the Understanding of Package Relationships), in the following points:

(a) Visualization improvements based on the feedback and conclusion of a first user study.

(b) Addition and complementary visualization for incoming references (in addition to outgoing references).

(c) A detailed presentation of a case study as well as a user study.