Laboratory LISTIC*Université de Savoie

Advisors Stéphane Ducasse, Vincent Chauvet

Master report

Package Blueprints

Visualisation de packages

Mathieu Suen

2006 - 2007

Keywords. Object-oriented programming, Visualization, Package, Program understanding, Refactoring.
Mots clés. Programmation orientée objets, Visualisation, Package, Compréhension d'applications, Refactoring.

*Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance

Résumé

Ce master preésente une visualisation compacte appelée Package Surface Blueprint. Cette visualisation montre les relations que les packages ont entre eux. Il est courant que les applications de grande envergure soient composées de nombreux packages. De plus les packages jouent différents rôles (c.-à-d. conteneurs de classes, entités de développement, structures de base, éléments architecturaux). C'est pourquoi les mainteneurs d'applications ont besoin d'outils pour analyser et comprendre comment les packages sont structurés. La vue Package Surface Blueprint montre les relations entre packages à l'aide de la notion de surfaces. Les surfaces sont des groupes de relations entre le package analysé et les packages référencés. Deux vues seront montrées. Une vue sera spécifique a la relation de référence tandis que l'autre montrera la relation d'héritage. Nous avons appliqué ces vues sur plusieurs larges études de cas : ArgoUML et Squeak.

Abstract

This master presents a compact visualization, named Package Surface Blueprint, that qualifies the relationships that a package has with its neighbors. Large object-oriented applications are structured over a 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 other. 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.

Contents

1 Introduction

Large software is hard to maintain. A large part of the development cost goes to maintenance. 50% to 75% of the overall cost of a software system is devoted to it. Software professionals spend at least half their time reading and analyzing software [Dav95]. Visualization takes part of this process ; used correctly, it provides a fast way to analyze software [DL01]. The main purpose of using Package Surface Blueprint visualization is to help understanding and refactoring the package.

Software maintenance. 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 a large number of packages. Ideally a package should contain highly cohesive classes and be as less coupled as possible with the rest of the application. This helps to maintain the application. If coupling between deployment units decrease, then you can easily remove them from the system and deploy them for other people [PN06, BDW99]. 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 (e.g. class containers, code ownership basic structure, architectural elements...). Packages provide or require services. They may play a core role or contain secondary 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 other in their provider/consumer roles. In addition, packages refactoring is based on remodularization algorithms [AL99, MMCG99, MM06]. They succeed in producing alternative views to refactor the system. 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 decide how to apply the changes.

Visualization in the Reengineering Process. Several previous works provide information on packages and their relationships, by visualizing software artifacts, metrics, structure or evolution [LLG06, DLP05, CE98, DL06, PGFL05, SWFM97, FD04]. Metrics can be somehow difficult to understand. They are project dependent and subject to change.

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.

Contribution. In this master, we propose the 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 relations with other packages, by showing the distribution of references to classes within and outside the observed package. We show to the user a direct mapping to the source code. No computation is done. We want to show a raw view to help in understanding how a package should be refactored. 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.

This master presents the work I have made during four months at the LISTIC laboratory. During these months I wrote a paper [DPS+07] for ICSM the International Conference on Software Maintenance. This master is mainly based on this paper. I also wrote a paper for the FAMOOSr workshop [SDP+07].

Chapter 2 presents the challenges that exist to support package understanding, it also summarizes the properties that a visualization should satisfy to be effective. Chapter 3 presents the structuring principles of a package blueprint which are then declined to support a reference view and an inheritance view in Section 3.2. In Chapter 4 we took an example and describe how we use our visualization. In Section 4.2 some striking shapes are present. In Chapter 6 and Chapter 7, we discuss our visualization and position it w.r.t. related work before concluding.

2 Visualization to Understand Packages

2.1 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. I present a coarse list of useful information to understand packages. The 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?

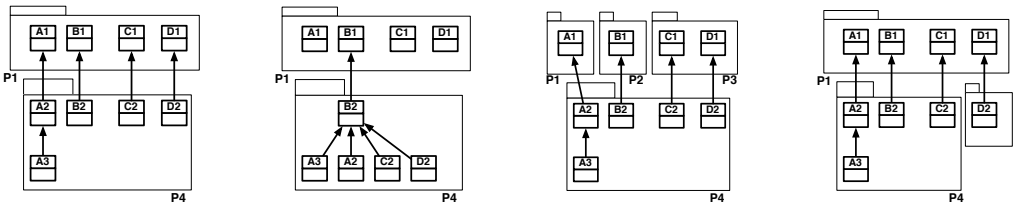
Cohesion and coupling. Transforming an application will follow natural boundaries defined by coupling and cohesion [BDW99, ABF04]. Assessing these properties is then important.

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 [GKSD05, PDP+07]. Approaches such as the distribution map may help in this task [DGW06].

Changes, Bugs and Co-changes. It is valuable to know which packages changed recently and together, or which ones contain more bugs.

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 [BDW99, ABF04, LM06]. 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.



(a) (b) (c) (d)

Figure 2.1: Different package configurations over the same number of classes.

Figure 2.1 shows situations where the same group of classes can be dispatched. Note that for the purpose of illustration, Figure 2.1 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 changed. As mentioned by R. Martin, importing a class equals importing the complete package [Mar00], 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 [AL99, MMCG99, MM06]. In this case 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.

2.2 Visualization Challenges

I researched the characteristics that an efficient visualization should hold [Ber74, Tuf01, War00]. As our focus were on providing a first impression of a package and its context, I want to exploit the gestalt principles of visualization and preattentive processing[1] as much as possible to help spotting important information [Tre85,

[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 squared and empty circle, a

Hea92, HBT93, War00].

To support the understanding of packages, I want the visualization to highlight the characteristics of a package in terms of its internal size, internal and external references. In particular I want to spot classes or dependencies that stand out in a given package. 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.

I want the visualization to highlight the general tendency of a package in terms of its internal size, internal and external references. In particular I 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 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 [Her00].

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.

2.3 Selected State of the Art

Langelier *et al.* [LSP05] propose a 3D view for mapping metrics (Figure 2.2). We can easily identify design violation in the treemap or sunburst layout. But it is hard to understand relation between packages. The number of classes make it hard to follow package outline. It is then difficult to make conclusion on the code quality.

Lanza *et al.* [LLG06] proposed a way to recover architecture by visualizing relationship but do not provide a fine grained view. He proposes also Package Patterns for Visual Architecture Recovery [LLG06] which shows incoming and outgoing dependencies map into the hierarchy of packages and classes (Figure 2.3). This visualization is interesting as it helps understanding packages and provides good mapping to the reality but fails to give a fine-grain view.

filled circle is usually not see preattentively). Some of the features are not adapted to our needs. For example, we do not consider motion as applicable.

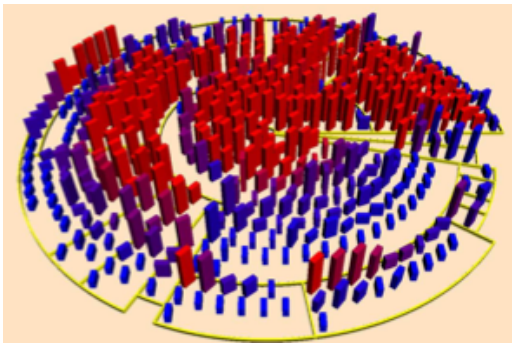


Figure 2.2: A 3D treemap of packages and classes. Red squares are classes strongly coupled according to the selected threshold.

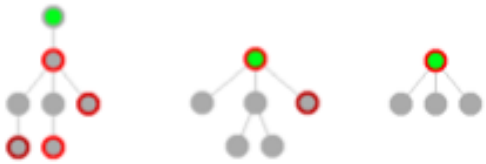


Figure 2.3: Each node of the tree is a package and each leaf is a class. The tree shows the hierarchy of the packages. For each node, red outline shows incoming dependencies while circle fill with green show outgoing dependencies. This figure represents variants of a typical pattern called Iceberg. Iceberg is when the rest of the system knows only the root package

An other visualization, proposed by Ducasse *et al.*, called Distribution Map [DGK06] shows the packages and the distribution of properties on the classes they contain. You only see the package structure but you do not see relationship. The view gives us answer to two main questions.

Spread: how much does a property spread across the reference partition: is it local or global?

Focus: how close does a property match the reference partition: is it well-encapsulated or cross-cutting?

The interesting thing is that you can map every properties you want. In Figure 2.4 we see the distribution of linguistic concepts.

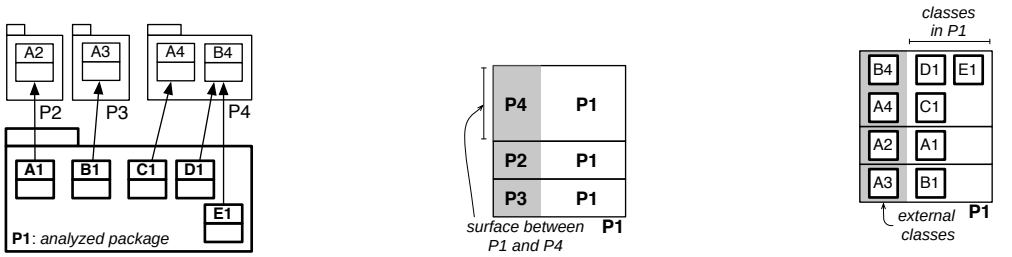
Apart from Lanza *et al.* there are some other 3D visualizations like the one from Marcus *et al.* [MFM03]. They represent lines of code by square boxes in a 3D space and map properties to the height, width and color. In this case the

Figure 2.4: Distribution Map of linguistic concepts over the packages of JEdit

occlusion disturb the reader by hiding some part of the system. They try to solve the problem by adding blending but it makes the view more complex and so less readable.

3 Package Surface Blueprints

A package blueprint represents how the package under analysis references other packages. Figure 3.1 presents the key principles of a Package Blueprint. These principles will be realized slightly differently when showing direct class references or inheritance relationships.



(a) Analyzed package

(b) Stacked surfaces

(c) With classes

Figure 3.1: Consider P1 that references four classes in three other packages (a). A blueprint shows the surfaces of the observed package as stacked subdivisions (b). In (c), small boxes represent classes, either in the observed package (right white part) or in referenced packages (left gray part).

3.1 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 3.1 (a) the package P1 is in relation with three packages P3, 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.

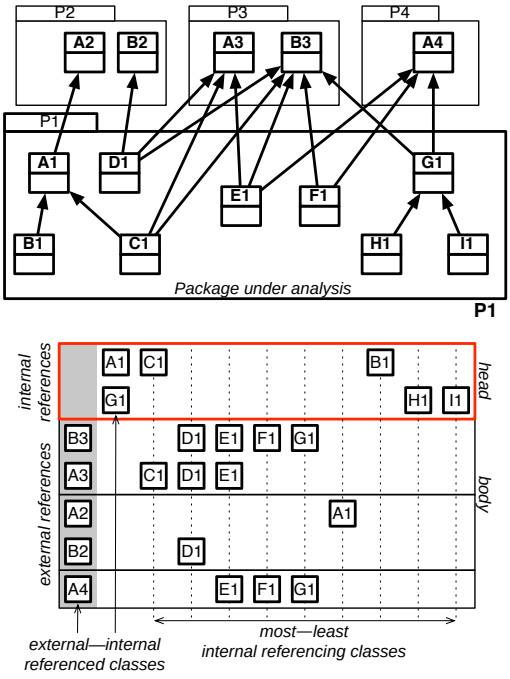


Figure 3.2: Surface package blueprint detailed view.

In Figure 3.1, the package blueprint of P1 is made from three stacked boxes because 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 left most gray-colored column of each surface, and the classes of the observed package on the right. In Figure 3.1, the topmost surface shows that class A1 references class A2.

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 3.1, the row of B4 stands out because the two referring classes D1 and E1 make it wider.

3.2 Detailed Explanation

To convey more information, we add variations to the basic layout described above, as illustrated in Figure 3.2.

3.2.1 Internal references

To support the understanding of references between classes inside the observed package, we add a particular (red bordered) surface at the top of the blueprint. We name this surface the head of the blueprint and the rest its body. Since this surface displays internal references, its left gray column for external classes will remain empty. In addition, 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 red surface indicates the number of classes referenced within the package.

3.2.2 Position

Internal classes are arranged by columns: each column (after the first one of the red surface) 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 3.2 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. In the horizontal direction, 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 leftmost columns in the white area of the package blueprint.

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 3.2 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.

3.2.3 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 done: 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.

3.3 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 3.3. In this variation there is no red surface because we consider only single inheritance, so we 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 (the others are black-bordered). In addition root classes such as Object are filled in cyan and abstract classes in blue. In Figure 3.3 A1 inherits from A2 defined in package P2, while B1, C1 and D1 inherits from A1.

The fill color of classes in the inheritance view represents the number of *references*, relative to the package. This makes it possible to correlate inheritance and references. For instance, the top-right view in Figure 4.1 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 4.2.

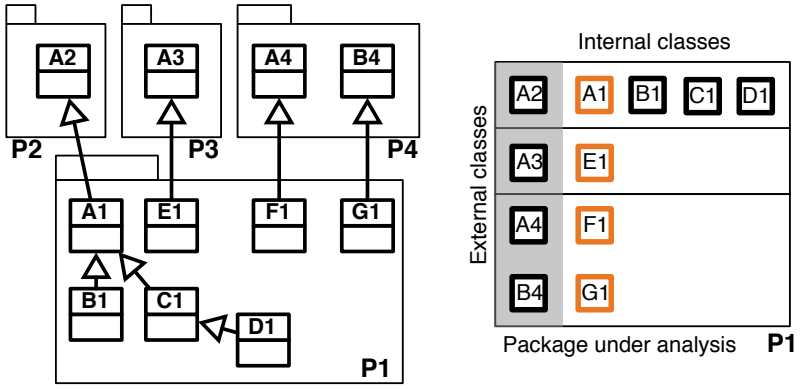


Figure 3.3: Inheritance package surface blueprint. Orange bordered classes inherit from external classes.

4 Case study: 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 4.1 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 [BDW99]. 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 4.1). 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 4.1). 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

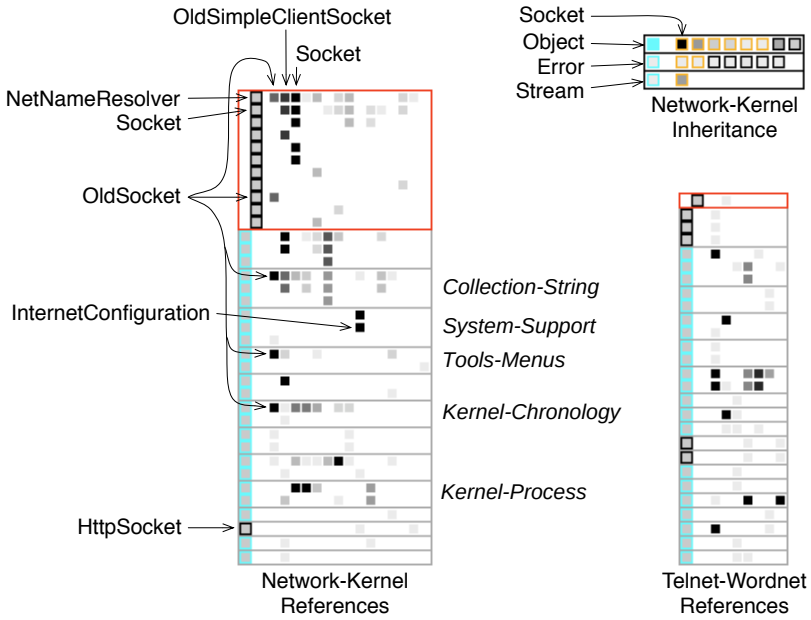


Figure 4.1: Analysing the Network-Kernel Package.

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.

4.1 Packages Within Their Application

Understanding a package in isolation (mainly as a consumer) is interesting but its 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.

4.1.1 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

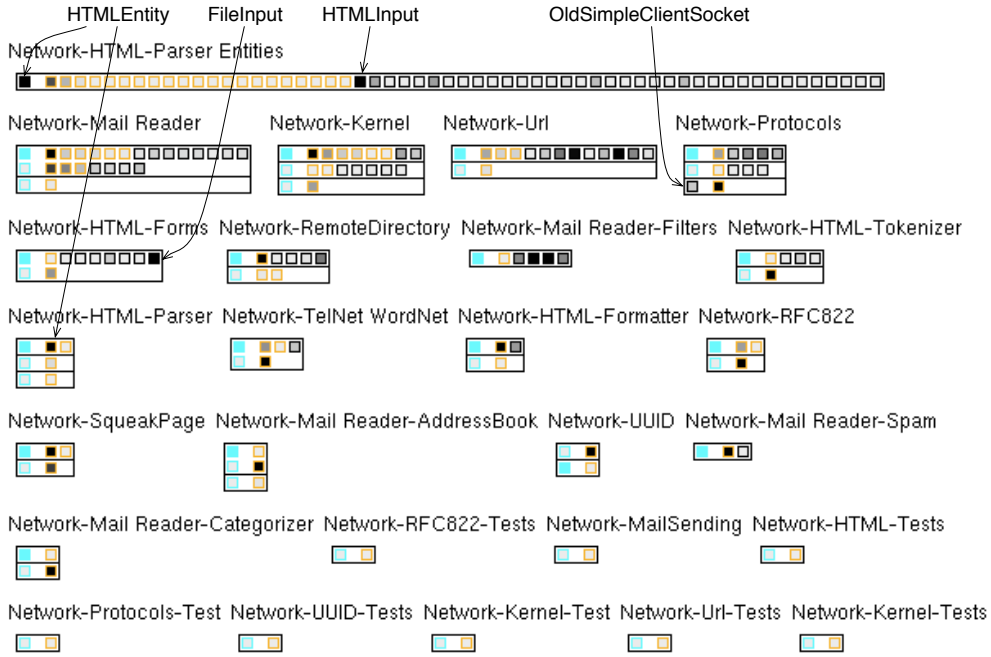


Figure 4.2: Inheritance global view in Network

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: Network. For example, Figure 4.2 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 Networksubsystem 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

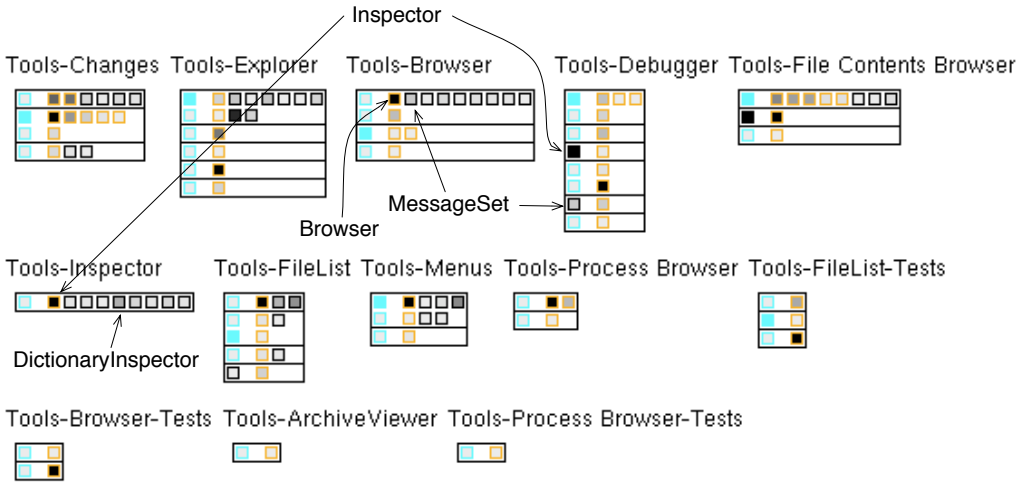


Figure 4.3: Inheritance global view in Tools

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: Tools. Figure 4.3 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 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

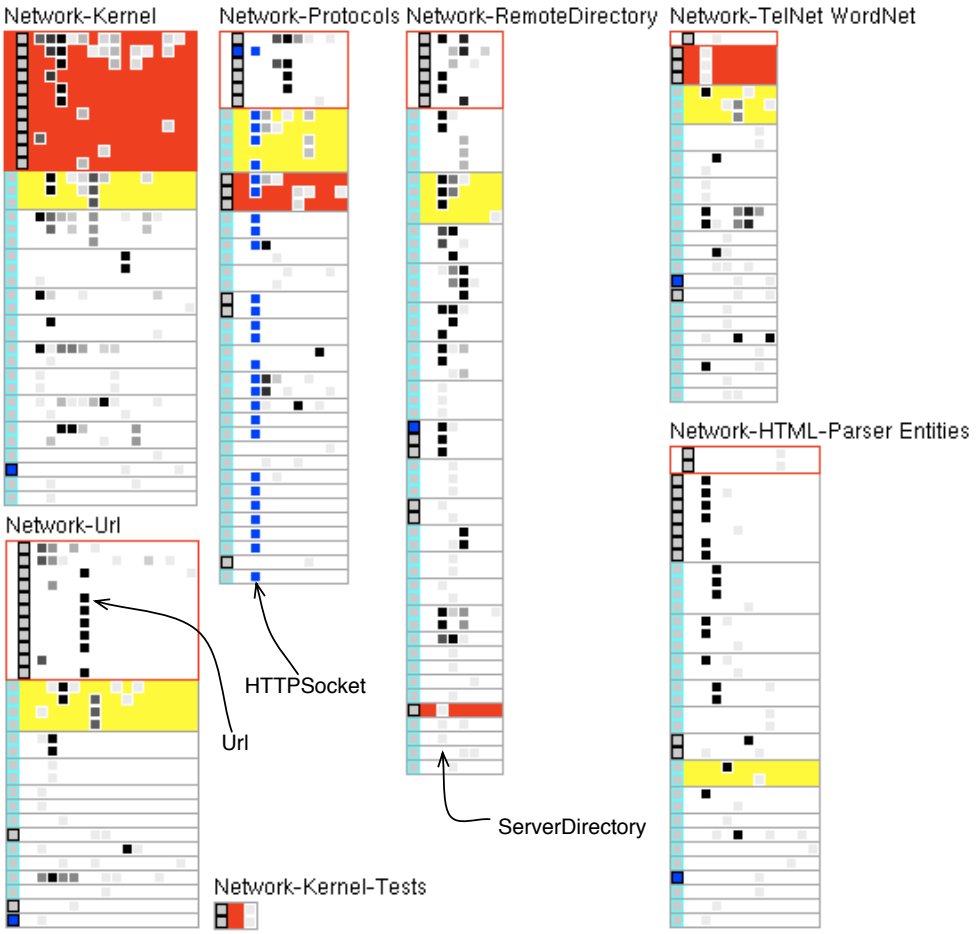


Figure 4.4: In this view, the Network-Kernel package was selected in red, surfaces with Collections-Strings annotated in yellow, and class HTTPSocket selected in blue.

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.

4.1.2 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 occurrences 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 4.4 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 4.4 shows how the package Network-Kernel is used within the application.

4.2 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.

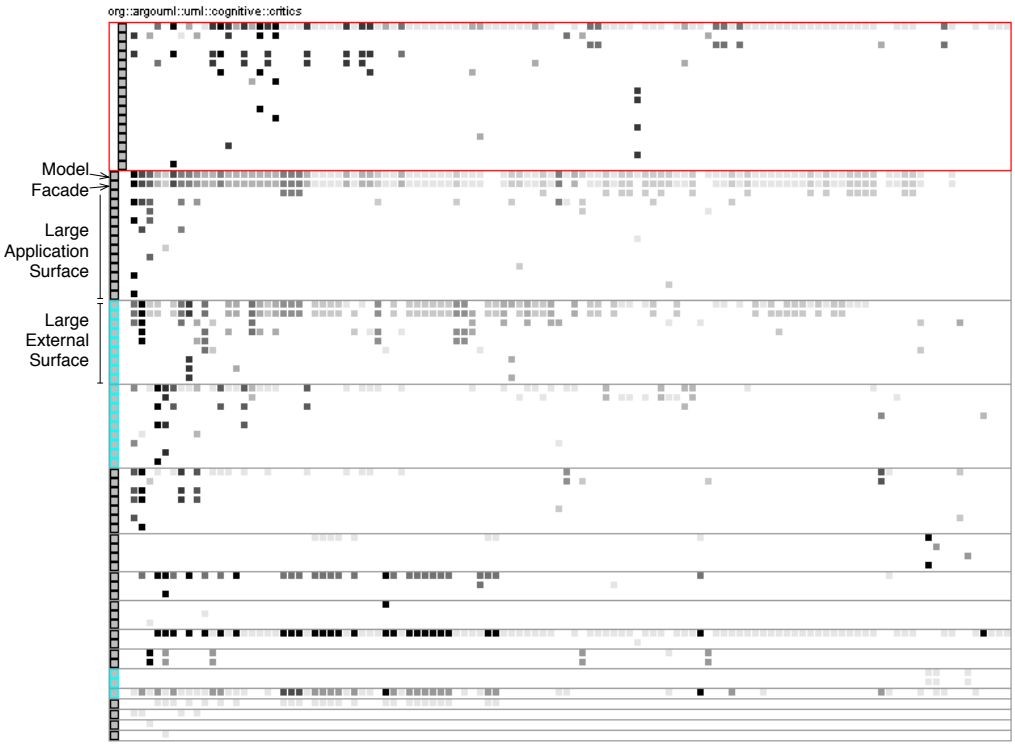


Figure 4.5: A Sumo Blueprint: the Critics package in ArgoUML.

Figure 4.6: A Tower Blueprint: Peer in Azureus

4.2.1 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 4.5 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 4.3). 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 4.4 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 packages typically extend a framework or a core library; Network-Kernel in Figure 4.4 is an example.

Loner Package. A loner is a package that contains only a couple of classes. It often contains a single test case class. The blueprint Network-Kernel-Tests in Figure 4.4 or Network-Mail Reader-Categorizer, Network-UUID, Network-Mail Reader-Spam of Figure 4.2 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 4.5, 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 4.4, 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 4.6. In Figure 4.4, Network-RemoteDirectory has a more cohesive head and three classes intensively referencing external packages.

4.2.2 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 4.4).

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 4.4, 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 4.5); we may thus assess that the Facade pattern is misused.

5 Usability-Study

To prove the intention of our visualization we conduct a survey. This survey has been done over 7 people with different programming skills. We have tested the visualization on the Squeak compiler.

5.1 The Case Study: Squeak Compiler

We chose this case because of our background. We are familiar with the Squeak compiler and developed the next compiler so we can better appreciate the information that the view provides. Compilers are system that every one has knowledge on even if there are limited, so it is easier for the tester to make hypothesis. They also contains some interesting patterns and is not to small or to big.

5.2 Setup

We first explain the visualization to the tester and give them the paper [DPS+07] with slides. Then we show them how to use it for detecting pattern in the Network application. The demo help the user to learn how to use the view. The demo is necessary because we spend time to learn how to get informations fast from the view.

To define the questions we have tested them to know if they are understandable and meaningful. The list of questions is:

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 make sense. the user can also assess whether they are misplaced.
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 helps to focusing on the understanding of relationship between packages. The user learns how to select classes and look for if they are referenced or if they make references.

3. *How would you qualify the references from MessageNode ? Compare it to MethodNode?* When we were looking at the ParserNode package we noticed that the class making most of the references where MessageNode. But MethodNode is the root of the syntax tree of the source code. It should then make more references. There is also ParserNode which is an abstract class for all the parser nodes. It should then define common abstraction for all the nodes and makes more reference then the other nodes.
4. *Do you identify some misplaced dependencies with packages outside the compiler application?* By looking the view we noticed several misplaced dependencies and we would like to know whether or not the user can find them.
5. *Under the assumption that a package containing classes that are referred to by other packages should be loaded first, can you identify a load order for the application?* We also 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.
6. *How would you qualify the cohesion of Compiler-Support?* When we get the habit we can see directly the package with no head and we can assess that cohesion in a such package is low. We want to know if the person could assess the same thing in a first look.
7. *Using the hierarchical view, what can you say about the shape of the Compiler-ParseNode package?* The parser nodes classes belong to the same hierarchy tree and are defined in one package. This is the common way of declaring a parser tree. It allow to define visitors to browse the tree and it is easier to annotate it.
8. *Can you tell us something about the 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 except for PrimitiveNode. This is not a good design since the node should be polymorphic to ParserNode.
9. *Do you think that you would have got the answer that you wrote down about the application 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.

The results are not yet available. They will be published in a Software journal (Transaction of Software Engineering or Journal of Software Maintenance).

6 Evaluation and Discussion

6.1 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 concerning its classes: a thin package with a long body depends on a lot of classes because of few internal classes. Moreover if 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.

6.2 Discussion

Our approach has worked well on our case studies. 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 perform a user evaluation. The Package Surface Blueprint answers the main challenges proposed in Section 2.1 and in Section 2.2; 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 [JMF99] 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 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 [LLG06]. 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.

7 Related Works

Several works are trying to provide information on packages. Lungu *et al.* guide exploration of nested packages based on patterns in the package nesting and in the dependencies between packages [LLG06]; 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 [SJSJ05]; 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 [SWFM97].

Ducasse *et al.* present Butterfly, a radar-based visualization that summarizes incoming and outcoming relationships for a package [DLP05], 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 [PGFL05].

Chuah and Eick use rich glyphs to characterize software artifacts and their evolution (number of bugs, number of deleted lines, kind of language...) [CE98]. 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 [DL06]. 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 inherits from external ones,...) and support the identification of their roles within an application.

8 Conclusion

In this master, I 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 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. We have validated the package blueprint usability with several independent software maintainers. In future work I plan to apply the visualization with some clustering algorithm.

Bibliography

[ABF04] Erik Arisholm, Lionel C. Briand, and Audun Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[AL99] Nicolas Anquetil and Timothy Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of WCRE '99 (6th Working Conference on Reverse Engineering)*, pages 235–255, 1999.

[BDW99] Lionel C. Briand, John W. Daly, and Jürgen K. Wüst. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, 1999.

[Ber74] Jacques Bertin. *Graphische Semiologie*. Walter de Gruyter, 1974.

[CE98] Mei C. Chuah and Stephen G. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, July 1998.

[Dav95] Alan Mark Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[DGK06] Stéphane Ducasse, Tudor Gîrba, and Adrian Kuhn. Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, pages 203–212, Los Alamitos CA, 2006. IEEE Computer Society.

[DGW06] Stéphane Ducasse, Tudor Gîrba, and Roel Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006.

[DL01] Stéphane Ducasse and Michele Lanza. Towards a methodology for the understanding of object-oriented systems. *Technique et science informatiques*, 20(4):539–566, 2001.

[DL06] Marco D'Ambros and Michele Lanza. Reverse engineering with logical coupling. In *Proceedings of WCRE 2006 (13th Working Conference on Reverse Engineering)*, pages 189 – 198, 2006.

[DLP05] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th*

IEEE International Software Metrics Symposium (METRICS'05), pages 70–77. IEEE Computer Society, 2005.

[DPS+07] Stéphane Ducasse, Damien Pollet, Mathieu Suen, Hani Abdeen, and Ilham Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, 2007.

[FD04] Jon Froehlich and Paul Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 387–396, Washington, DC, USA, 2004. IEEE Computer Society.

[GKSD05] Tudor Gîrba, Adrian Kuhn, Mauricio Seeberger, and Stéphane Ducasse. How developers drive software evolution. In *Proceedings of International Workshop on Principles of Software Evolution (IWPSE 2005)*, pages 113–122. IEEE Computer Society Press, 2005.

[HBT93] C. G. Healey, K. S. Booth, and Enns J. T. Harnessing preattentive processes for multivariate data visualization. In *GI '93: Proceedings of Graphics Interface*, 1993.

[Hea92] C. G. Healey. Visualization of multivariate data using preattentive processing. Master's thesis, Department of Computer Science, University of Bristish Columbia, 1992.

[Her00] Martin Hermann. Erstellung einer zentralen Kundendatenbank bei Firma W. Gassmann AG Biel. Informatikprojekt, University of Bern, June 2000.

[JMF99] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[LLG06] Mircea Lungu, Michele Lanza, and Tudor Gîrba. Package patterns for visual architecture recovery. In *Proceedings of CSMR 2006 (10th European Conference on Software Maintenance and Reengineering)*, pages 185–196, Los Alamitos CA, 2006. IEEE Computer Society Press.

[LM06] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. Springer-Verlag, 2006.

[LSP05] Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE*, pages 214–223, 2005.

[Mar00] Robert C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. Comprehension of software analysis data using 3d visualization. In *IWPC*

'03: Proceedings of the 11th IEEE International Workshop on Program Comprehension, page 105, Washington, DC, USA, 2003. IEEE Computer Society.

[MM06] Brian S. Mitchell and Spiro Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[MMCG99] Spiros Mancoridis, Brian S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[PDP+07] Damien Pollet, Stéphane Ducasse, Loïc Poyet, Ilham Alloui, Sorana Cîmpan, and Hervé Verjus. Towards a process-oriented software architecture reconstruction taxonomy. In René Krikhaar, Chris Verhoef, and Giuseppe Di Lucca, editors, *Proceedings of 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*. IEEE Computer Society, March 2007. Best Paper Award.

[PGFL05] Martin Pinzger, Harald Gall, Michael Fischer, and Michele Lanza. Visualizing multiple evolution metrics. In *Proceedings of SoftVis 2005 (2nd ACM Symposium on Software Visualization)*, pages 67–75, St. Louis, Missouri, USA, May 2005.

[PN06] Laura Ponisio and Oscar Nierstrasz. Using contextual information to assess package cohesion. Technical Report IAM-06-002, University of Berne, Institute of Applied Mathematics and Computer Sciences, 2006.

[SDP+07] Mathieu Suen, Stéphane Ducasse, Damien Pollet, Hani Abdeen, and Ilham Alloui. Package surface blueprint: A software map. In *FAMOOSr, 1st Workshop on FAMIX and Moose in Reengineering*, 2007.

[SJSJ05] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[SWFM97] Margaret-Anne D. Storey, Kenny Wong, F. D. Fracchia, and Hausi A. Müller. On integrating visualization techniques for effective software exploration. In *Proceedings of IEEE Symposium on Information Visualization (InfoVis '97)*, pages 38–48. IEEE Computer Society, 1997.

[Tre85] Anne Treisman. Preattentive processing in vision. *Computer Vision, Graphics, and Image Processing*, 31(2):156–177, 1985.

[Tuf01] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 2nd edition, 2001.

[War00] Colin Ware. *Information Visualization.* Morgan Kaufmann, 2000.

A Smalltalk syntax

The table A.1 show the different syntaxe element of Smalltalk

.	period separates statements.
:=	affectation.
^	returns statement ; exit the method where it appears.
'a string'	singles quotes delimit strings.
[statements]	square brackets delimit closures (AKA block).
[:arg1 :arg2 | bloc]	block can tack argument.
| tmpVar1 tmpVar2 |	the pipes is use to declare temporary variable. the scope is limited to the method
"commented"	double quotes denote comments.
$a	a character instance of Character.
#symb	symbol instance of ByteSymbol.
#(a b)	array containing a and b instance of Array.

Table A.1: Basic syntactic elements

In Smalltalk we have 6 pseudo variables. They are given in the Table A.2.

true and false	boolean instance of True and False respectively.
nil	value given to undefined variable.
self	current object. self have the same semantic than this in Java.
super	current object. Lookup have to start in the super class where super is used.
thisContext	stack of the execution.

Table A.2: Pseudo Variables

Smaltalk have only methods. We can define method in 3 way:

As unary messages. messages that do not take any arguments.

```
aCollection removeAll.
```

As binary messages. messages that take one argument and they are written with special token(',' '+' '-'...).

```
1 + 3.
"Hello ", "world !"
```

As keyword messages.. messages that can take as many parameters as you want. Keyword are compose of token ended by a semi column. Each argument are written after each of the token.

```
aCollection replaceFrom: 1 to: 6 with: anOtherCollection.

"Equivalent in Java to:"
aCollection.replaceFromToWith(1, 6, anOtherCollection);
```

Message sending priority. When the compiler parses the statement, it read it with the following priority:

1. Unary messages ;
2. Binary messages ;
3. Keyword messages.

For example:

```
5 factorial + 5 gcd: 5
```

Should be read as:

```
((5 factorial) + 5) gcd: 5
```

Thus mathematics operation order are not preserved. So `3 + 4 * 3` is equal to 21 and not 15. You should write `3 + (4 * 3)` if you want to give priority to the multiplication.

Cascades. You also can send several messages to the same object. To do so, you use the ';' operator.

```
aCollection
    add: anObject;
    add: anOtherObject;
    add: aThirdObject.
```

Remark Is common to see messages begining by a #. This convention help to read them in a plain text.

Methods definition are written as the following example:

```
String>>lineCount
  "Answer the number of lines represented by the receiver (a string), where every
  carriage return (cr) adds one line."

  | count |
  count := 1.
  self do:
    [:c | (c == Character cr)
            ifTrue: [count := count + 1]].
  ^ count
```

The above code should be read as:

1. new method call lineCount is defined in the String class.
2. the new token is a comment explaining the purpose of this method.
3. new count temporary is declare and initialized to 1.
4. message #do: is sent to the current object (self). It takes a block as argument, the block should be written to accept one argument. It will be evaluated for each character of the current object.

   The #do: send is equivalent to foreach in C# or to mapcar in Common Lisp.
5. block tack c as argument. c will contain each of the character of the current string.
6. body of the block starts by comparing the current character with a line feed. In Smalltalk line feed is given by calling the #cr message on Character.
7. result of the comparison is true or false.
8. true and false understand the message #ifTrue:. #ifTrue: take a black as argument. The block is evaluate only if the receiver is true.
9. if c is a line feed the argument of #ifTrue: is evaluated and count is increased by 1.
10. at the end, the #lineCount method return the number of line feed count in the receiver string.