Package Patterns for Visual Architecture Recovery

In Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 2006)

Mircea Lungu *and* Michele Lanza
Faculty of Informatics
University of Lugano, Switzerland

Tudor Gîrba
Software Composition Group
University of Bern, Switzerland

Abstract

Recovering the architecture is the first step towards reengineering a software system. Many reverse engineering tools use top-down exploration as a way of providing a visual and interactive process for architecture recovery. During the exploration process, the user navigates through various views on the system by choosing from several exploration operations. Although some sequences of these operations lead to views which, from the architectural point of view, are mode relevant than others, current tools do not provide a way of predicting which exploration paths are worth taking and which are not.

In this article we propose a set of package patterns which are used for augmenting the exploration process with information about the worthiness of the various exploration paths. The patterns are defined based on the internal package structure and on the relationships between the package and the other packages in the system. To validate our approach, we verify the relevance of the proposed patterns for real-world systems by analyzing their frequency of occurrence in six open-source software projects.

Keywords: Software exploration, architecture recovery, reverse engineering, program comprehension, visualization

1. Introduction

Although the architecture of a system is usually documented at the time of its development, evolutionary processes lead to the decay of the initial design and result in a separation between the *as-designed* and *as-built* architectures [11]. Because the architecture of the software systems is an important asset for many software engineering tasks such as migrations, impact analysis or feature additions, there comes a time in the life of a system when the actual architecture has to be recovered from the most reliable source of information about it: the source code.

There are many approaches to architecture recovery. One approach, called *reverse architecting*, consists in recovering the architectural information using a bottom-up approach, by first modeling the lowest level information and then raising the abstraction level by grouping related entities [27, 28, 29, 34]. The main drawback of this approach is that it requires domain-specific knowledge and it is largely manual and therefore time-consuming.

Another class of approaches uses clustering techniques in order to abstract related sets of artifacts into subsystems [19, 20, 23, 24]. Such approaches can be automated in a very high degree, but they have the drawback of having a high number of possible false positives that require manual verification.

Some of the existing approaches are highly dependent on visualization and interaction [13, 26, 25, 32]. Out of these, a distinct class is the top-down exploration tools which use interactive exploration techniques to navigate hierarchical decompositions of software. The problem of these techniques is that from the many views on the system generated during the exploration, only a subset is relevant for the architecture of the system, and this subset can only be detected by the individual analysis of each view.

In this article we propose a classification of the packages that can be used to augment the exploration by annotating the views with information regarding the worthiness of the possible exploration paths. The classification is based on information regarding the structural properties of the packages and on the way they interact with one another. We validate our approach by analyzing the frequency of occurrence of the proposed package patterns in six open-source systems.

Structure of the article. In the next section we discuss the problems that arise with visual architecture recovery. In Section 3 we introduce the concepts we use in the definition of the patterns. In Section 4 we describe the patterns themselves. We validate our approach on several open-source systems (Section 5). In Section 6 we briefly present a tool that implements the concepts presented in this article and then, in Section 7, we discuss the approach from different points of view. We present the state of the art in Section 8 and in Section 9 we conclude with a brief discussion and an

outlook on our future work in this domain.

2. Visual Architecture Recovery

There are many definitions of software architecture, each of them emphasizig another aspect of the concept. The one that we consider in this work is provided by Bass and Clemens: *"The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them"* [4].

There is a significant variety of tools that support the extraction and recovery of the architecture from a system and in which visualization and interaction play an important role [26, 25, 22]. While some steps of the process (*e.g.,* fact extraction, view generation) are usually automated, none of the tools work without a certain degree of human intervention (*e.g.,* the user has to group related artefacts [25], the user has to compare the architecture as-extracted with the architecture as-predicted [26] or the user decides which navigation paths to follow [22, 21]).

One distinct subclass of these tools, the top-down exploration tools, offer the possibility of navigating a hierarchical decomposition of a system [12, 22, 21]. Although there are many ways in which a system can be decomposed hierarchically (*e.g.,* directory structure, clustered decompositions, implicit namespaces, *etc.*), we limit our discourse to the hierarchical package decomposition of Java systems.

The top-down exploration tools take a hierarchical decomposition of the software system and, starting from the view with the highest abstraction level, let the user generate new views by applying *gardening operations* [30]:

- *Expand.* By expanding a node the view is updated and the node is replaced with nodes representing its children.
- *Collapse.* By collapsing a node corresponding to a package, the node, together with all the nodes representing the siblings of the package are removed from the view and replaced with a node representing the parent package.

During the exploration of a hierarchical package decomposition, based on the previously mentioned operations, many views can be generated depending on the sequence of exploration operations that have been taken. From all these views, only a subset is relevant for the architecture of the system as only a subset of the packages in the package hierarchy represent subsystems.

Example. Figure 1 shows a system in which the packages with subsystem semantics are X, Y and Z. Views which do not have meaning from the architectural point of view can be considered:

- A view in which X is expanded into its subpackages X1, X2, and X3 will not present architectural information because the three subpackages do not have subsystem semantics: they only help in the implementation of subsystem X.
- A view presenting B unexpanded will not present architectural information because B does not have subsystem semantics: it is merely a container for the two components X and Y.

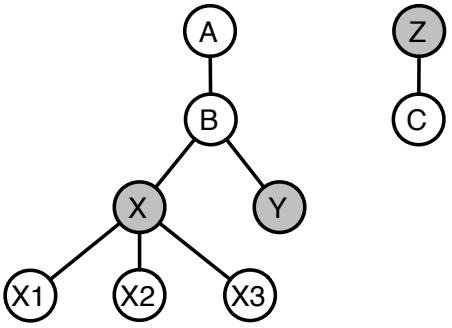

Figure 1. A containment hierarchy of packages where the architectural components are modeled in the packages X, Y and Z.

The example shows that, the responsibility of deciding wether a view is relevant or not is carried by the reverse engineer. During the navigation process, he has to analyze the view and decide whether all the packages in the view are at the right abstraction level or not. After each exploration operation, the reverse engineer has to address the following questions:

- Is a package at a higher abstraction level than needed? Then expand it.
- Is a package at the right abstraction level? Then do not expand it.
- Is a package at a lower abstraction level than needed? Then collapse it.

The challenge that results from here is providing a way to automate the package characterization process. To address this challenge, we propose a classification of packages based on their relation with the other packages in the system and on their internal structure. The result of the classification is a set of package patterns that have associated exploration operations that have to be taken once the package appears in a view.

3. Packages and Dependencies

Packages are the main mechanism for the decomposition and modularization of a system written in Java, and they are essential for the understanding and maintenance of non-trivial programs. However, the packages in Java are defined implicitly, by the mentioning of their name in the definition of a class. Moreover, as the semantics of package containment are not clearly specified, sometimes the term can become ambiguous. We see packages from two points of view:

1. *Restricted Package* - the collection of classes that belong to a package. The classes defined in the subpackages are not considered.
2. *Extended Package* - the collection of classes that belong to a package and all its subpackages.

It is important to be able look at the packages in both ways because for some purposes a package can be regarded as a collection of classes while for others the fact that it contains subpackages and the subpackages contain classes is important.

Although Java offers a language mechanism for supporting the modelling of the dependencies between packages (*i.e.,* via the import statement), this mechanism works only at the Restricted Package level. However, for some understanding tasks, the dependency relation has to be aggregated from lower level packages to higher level ones. Sometimes even the number of the dependencies between two packages is important for the understanding of the relationship between them (In Figure 9 the dependencies between two packages are represented as edges whose width is proportional with the number of inferred dependencies).

In this work we consider the dependency between two packages as being "The set of all the class dependencies between the classes defined in the two packages". A dependency has a direction so it makes sense to talk about incoming dependencies and outgoing dependencies.

Example: Consider the packages in Figure 2. There is no dependency between A and C as restricted packages. On the other hand, there is a dependency between the extended packages A and C. We say that A has an outgoing dependency on C and C has an incoming dependency from A.

Restricted Package Types. During the exploration, we have a set of packages that are simultaneously visible at a given moment in time. We call these packages the *working set*. Our goal is to characterize a package based on its interaction with the other packages in the working set. To do this we caracterize the interaction of each of its subpackages with the packages in the working set. We distinguish four types of restricted packages:

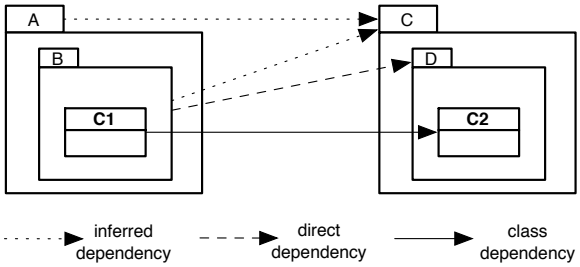

Figure 2. The two types of dependencies between packages

1. *Silent package* - there are no dependency relations between the restricted package and the packages in the working set.
2. *Consumer package* - there is a dependency relation from the restricted package to the packages in the working set.
3. *Provider package* - there is a dependency relation from the packages in the working set to the considered restricted package.
4. *Hybrid package* - there is a bidirectional dependency relation between the restricted package and the packages in the working set.

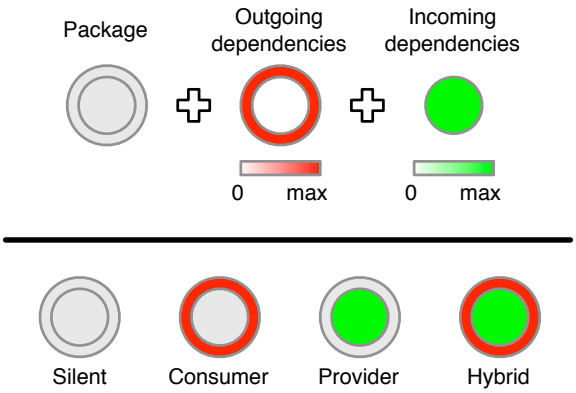

Figure 3. The visual representations of the four types of Restricted Packages

Using these types of restricted packages and their symbolic representation as presented in Figure 3 we characterize the interaction between an extended package and its working set by looking at the types of restricted packages it contains and at the way they are positioned in the structure.

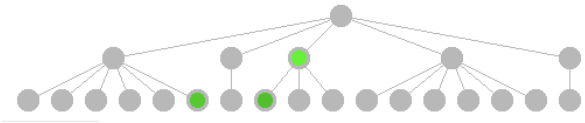

Figure 4. The *org.bouncycastle* package from Azureus

For example, looking at the package in Figure 4 we can see that although it has a rich subpackage structure, only three of its subpackages provide functionality to the packages in the working set. This leads us to think that the package implements a complex functionality which it exposes through a small interface: Indeed, the package in the figure is the *org.bouncycastle* package in the Azureus system, a package which provides cryptographic services.

4. Package Patterns

The main problem when visually exploring a system is deciding which of the packages visible at a given moment should be expanded and which should not. Because doing this analysis after each exploration step is too time consuming we decided to encode the knowledge of the process in a set of patterns that can be automatically applied.

The patterns are organized as a catalog and are presented using the following structure: a short description, suggestion, detection rule, rationale and discussion. The *Detection Rule* is a set of tests which are used to detect wether a package conforms to the pattern or not. For the packages that are conforming an action is suggested in the *Suggestion* section and the reason of doing so is explained in *Rationale*. A *Discussion* ends the description of the pattern.

4.1. Iceberg

An Iceberg is a package on which other packages in the working set depend, but the dependency is limited to the restricted version of the package. This means that from the point of view of the other packages, its subpackages are hidden: all they see is the top of the iceberg.

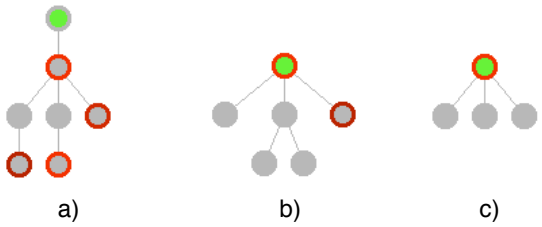

Figure 5. Possible Configurations for Iceberg packages. *a* and *b* are from Azureus while *c* is a Perfect Iceberg from Infoglue.

Suggestion: Do not expand the package

Rationale: Although the subpackages of such a package might use functionality provided by other packages in the working set, the extended package acts as one logical provider of functionality and the understanding of the other packages would not benefit from expanding it.

Detection Rule: An Iceberg is a package for which the following rules hold:

1. The package in the restricted sense is either a Provider package or a Hybrid package.
2. None of the descendant subpackages in the restricted sense is a Provider or Hybrid package.

Discussion: A special case of the pattern is a *Perfect Iceberg* for which all the subpackages are of type *Silent*. Such a package is probably a well delimited component and unit of reuse or it could be an implementation of the *Facade* design pattern. An example of such package is package c) from Figure 5.

A different kind of Iceberg package could be detected by statistical means as being a package for which the *exposed functionality/defined functionality* ratio is very low.

4.2. Autonomous

An Autonomous package is one which contains at least one Provider subpackage and no Consumer or Hybrid subpackages. In other words, an autonomous package does not depend on the other packages in the working set.

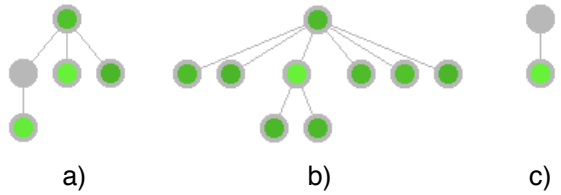

Figure 6. Possible configurations of Autonomous packages. Package c) is from jEdit and is also classified as Fall-Through

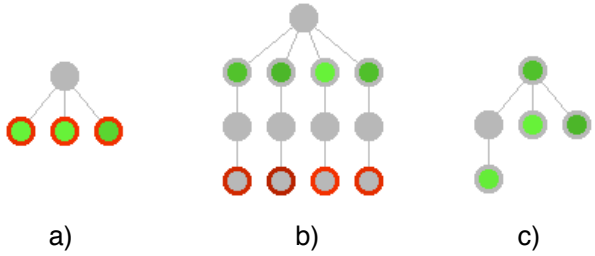

Figure 7. Possible configurations of Archipelago packages. Packages a) and b) display perfect structural symmetry.

Suggestion: Do not expand the package.

Rationale: From the definition it is clear that such a package does not depend on any other packages in the working set. This means that it is an independent provider of functionality.

Detection Rule: In order for a package to be an Autonomous package it has to respect two conditions:

1. At least one descendant of the package, or the package itself, regarded in the restricted sense should be of type Provider.
2. None of the descendant packages or the package itself regarded in the restricted sense can be of type Consumer or Hybrid.

Discussion: The Autonomous pattern is a more strict rule for detecting modular components in the code than the Iceberg. This is due to the second condition which forces an Autonomous package to not depend on any of the other packages in the working set.

If a package is classified as both Autonomous and Iceberg in the same time, the suggestion is reinforced. On the other hand, if a package is detected as being Autonomous and Fall-Through (see package *c* from Figure 6), the Fall-Through suggestion has priority.

The existence of Autonomous packages in a system is a sign of a good modular design.

4.3. Archipelago

An Archipelago is a package which contains at least three direct subpackages which, when regarded in the extended sense, do not depend on one another.

Suggestion: Do not expand the package.

Rationale: Becuase there are no invocations between the subpackages, it means that there is no need for collaboration for achieving the desired functionality. Such a situation can appear in three cases:

1. When the package contains alternative implementations of the same concept (*e.g.,* architecture dependent implementations).
2. When the package represents a collection of entities of the same type (*e.g.,* plugins, entities from the domain model).
3. When the contained subpackages are bundled together for lack of a better alternative (*e.g.,* a utilities package).

Only in the last case it might be argued that it is possible to bring more architectural information by expanding the package. However, it can be assumed that the subpackages are not important for the system if they were bundled in a utilities package, therefore, it is unlikely that the user would obtain architectural information by expanding them.

Detection Rule: In order for a package to be an Archipelago package it has to respect two conditions:

- It should have at least three direct subpackages.
- The direct subpackages in the extended sense should not depend one on another.

Discussion: During the experiments that we have performed, we have detected cases where a package that contained several entities of the same kind was not detected as an Archipelago because two out of seven packages depended very lightly on a third. One solution to this problem and other of the same kind would be to modify the rules in such a way that they use fuzzy logic and return a smaller confidence when the rules are not fully obeyed.

4.4. Fall-Through

A Fall-Through package is one that contains a single subpackage and whose restricted version is a Silent package. Such a package should be expanded as it provides no interesting information.

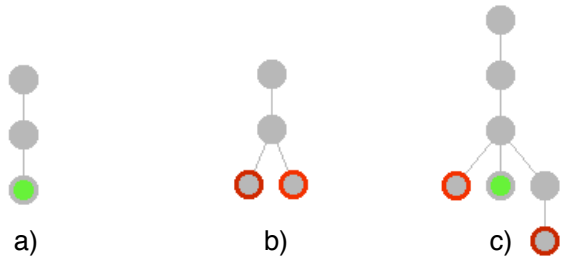

Figure 8. Possible configurations of Fall-Through packages. a) is from Infoglue, b) is from Azureus and c) is from Infoglue

Suggestion: Expand the package.

Rationale: The description implies that the only information that such a package could provide would be conveyed by its name. Therefore, there is no architectural information loss if the user expands the package. Moreover, by expanding the package, the name of the inner package becomes visible and the visible information becomes more precise.

Detection Rule: For a package to be a Fall-Through package it has to respect two conditions:

1. The package should have only one direct subpackage.
2. The package seen in a restricted sense should be a Silent package.

Discussion: Usually, the top level packages in a java system, are Fall-Through packages. This is a result of having the top-level packages in a hierarchy mimic the domain name of the author.

The suggestion might conflict with other suggestions in the case where several patterns apply for the same package (*e.g.,* Autonomous pattern). In case of conflict, the Fall-Through suggestion should have priority.

5. Validation

To validate our approach, we seek answers for the following questions:

- How often do they appear in real world systems?
- Do overlapping patterns occur? Do they contradict or reinforce each other in their suggestions?

To answer these questions, we perform several experiments. For each experiment, we set up a script that simulates all the possibile ways of exploring a system in a top-down manner and automatically gathers the useful data from system. The systems used for the analysis are the following open-source systems:

- ArgoUML (v0.16.1) - a UML modelling tool [1].
- Azureus (v2.2.0.2) - a BitTorrent client [3].
- Columba (v1.0 RC2) - a Java email client [8].
- Hipergate (v2.1.17) - an web based customer relation management tool [15].
- Infoglue (v1.3.2) - a content management platform [16].
- jEdit (v4.3pre2) - a text editor for programmers [17].

5.1. How often do the patterns appear in real-world systems?

To answer this first question, we devised the following experiment. For each system and each pattern we set the script to start from the top-most view on the system and successively expand the packages it encounters. Each time a package is brought into view for the first time, it is characterized in the context of the current working set. For the presented patterns this is enough as the characterization stays the same even if the other packages are expanded or collapsed.

Table 1 presents the aggregated results of the experiment. The *Tested* column of the table presents the number of packages that were expanded and classified. The second and third column present the number of packages for which suggestions were made as an absolute value and as a relative value.

The detailed results of the experiment are present in Table 2. In the followings we detail the analysis separately for each pattern.

Iceberg. It is interesting to see that in all the systems, the Iceberg patterns were around 10% of the packages. It is only in jEdit that the percentage is very low. However, jEdit is not very relevant case study as the package hierarchy is minimal. On the other hand, in the jEdit system that we found *bsh*, a *Perfect Iceberg* (labeled *c* in Figure 5). At a

System	Tested	Patterns	Percentage
ArgoUML	67	15	(22%)
Azureus	250	66	(26%)
Columba	171	13	(7%)
Hipergate	77	13	(16%)
Infoglue	70	31	(44%)
jEdit	40	14	(40%)

Table 1. The results of the exploration simulation.

closer look we understood why the package is so well modularized: it contains the BeanShell Java scripting interpreter [5] which is third party code and therefore, does not depend on the other components of the system.

After checking the packages which conform to the Iceberg pattern we did not find any false positives.

Autonomous. The rules for the detection of the pattern were very strictly defined and this is why there was one system where no conforming packages were detected. Besides the strictness of the rules, another reason for the low occurrence rate is that when the reusable components are not that big they are bundled in a single package and when they are big, they might present their functionality through a facade package. Iceberg does not provide this functionality as it has conditions only regarding the top-level package while sometimes the package that provides the functiuonality is lower in the hierarchy. This is the case with the package *org.bouncycastle* that is presented in Figure 4.

Some of the detected patterns are surprising. For example package b) from Figure 6. On the one hand, the package is totally independent of all the packages in the system. On the other hand, the dependence on all the levels of the hierarchy looked suspect. After closer inspection we found out that the package contains 47 interfaces! This explains why there were no dependencies on other packages and why the package was detected as a Autonomous.

System	Tested	Iceberg	Fall-Through	Autonomous	Archipelago
ArgoUML	67	5	6	3	1
Azureus	250	34	15	9	8
Columba	171	5	4	3	1
Hipergate	77	5	5	3	0
Infoglue	70	6	23	0	2
jEdit	40	1	8	4	1

Table 2. The results of the exploration simulation.

Archipelago. Although Archipelago does not have a high occurrence rate the packages that were detected were in all the cases conforming to the hypotheses presented in the Rationale section of the pattern: they were either collections of packages with parallel functionality or packages gathering together other utility subpackages.

One interesting fact is that some of the Archipelago packages presented surprising symmetries at the structural level (see Figure 7). This is probably the result of detecting sets of similar artefacts packaged together. The structural symmetry of the contained packages could be another way of detecting the Archipelago pattern.

Fall-Through. The pattern is well represented in the analyzed systems. It is interesting to see that the pattern is complementary to Iceberg and Archipelago, but not to Autonomous.

5.2. Do overlapping patterns occur?

As it can be seen from their definitions in Section 4, the patterns are not always mutually exclusive. Some packages can conform to the rules of several patterns at the same time. To find how often this phenomenon happens, we studied its occurrence in the analyzed systems. Table 3 presents the aggregated results.

Pattern	Iceberg	Fall-Through	Autonomous	Archipelago
Iceberg	-	impossible	0	2 **(+)**
Fall-Through	impossible	-	8 **(-)**	impossible
Autonomous	0	8 **(-)**	-	5 **(+)**
Archipelago	2 **(+)**	impossible	5 **(+)**	-

Table 3. Pairwise overlapping between patterns. The "(+)" and "(-)" signs indicate wether the two overlapping patterns have identical and opposite suggestions, respectively.

We see that there were only a few packages with multiple classifications. These packages can be split in two categories: (1) the ones for which the suggestions of the various matching patterns are the same and (2) the ones for which the suggestions are contradictory.

For example, in the case where a package is classified as both Autonomous and Fall-Through, the suggestions are contradictory. However, as we proposed in the discussion related to the Fall-Through pattern, in such a case the suggestion of expanding the package has priority.

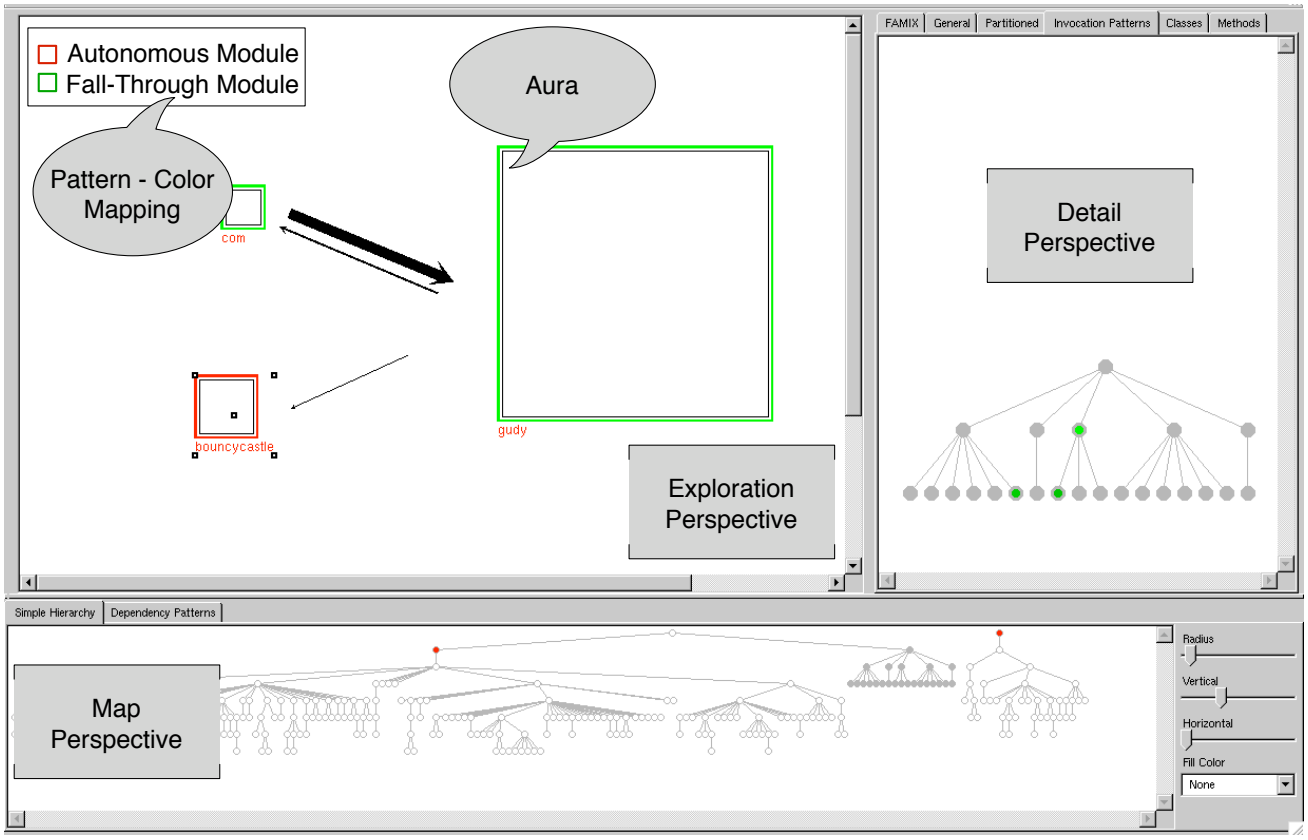

Figure 9. Softwarenaut exploring the Azureus case study. In the Exploration Perspective the packages are annotated with navigation suggestions

6. Softwarenaut

We have implemented the automatic package characterization process in our exploration tool called Softwarenaut [21, 31]. Figure 9 presents a snapshot of Softwarenaut during the exploration of the Azureus system. The tool provides three complementary perspectives on a system. The *exploration perspective* is the one where the exploration takes place. The *map perspective* emphasizes the position of the visible packages in the system's package hierarchy. The *details perspective*, provides details on the package which has the focus in the exploration perspective.

The tool assists the user in navigation by annotating the exploration view with suggestions based on package patterns. Figure 9 shows one Autonomous and two Fall-Through packages emphasized with coloured auras. The packages for which the recommendation is *expand* are emphasized with auras coloured in shades of red, while the ones for which the recommendation is *stop* are emphasized with auras coloured with shades of green.

7. Discussion

Human Decision. We consider the approach presented in this paper a first step towards an automatic decomposition of a system based on its package structure. However, in the current stage, the decomposition process can not be fully automated because there are two situations when human decision is needed: when there are no heuristics that apply on a given package and when there are heuristics that propose different actions for the same package. In these cases the user has to decide the operation based on intuition and the available information.

Visualization Variations. There might be systems where both a package and one of its descendants would be relevant for the same view. With the visualization techniques used in the exploration perspective, only one of them can be visible at a given moment. A solution to this problem could

be devised using a visualization technique based on nested graphs as the one used in the SHriMP tool [32].

Data Model. In this paper, we only identified dependencies between packages only by analyzing the invocations. Furthermore, we did not distinguish between different types of invocations. Other kinds of relationships, like inheritance, are also important for the architecture. We plan to explore the different types of relationships in the future.

Patterns Usage. The patterns were developed with the explicit goal of helping in the exploration process. However, there is nothing that impedes their utilization without the visual aids. They could be used to detect violations of good design rules or could detect possible improvements in the package structure of the system.

Extensibility. The simplicity of the approach is both a strength and a weakness. On the one hand, using the same technique other languages and other dependency types can be analyzed. On the other hand, the technique uses only high-level information and more specific patterns can not be found.

8. Related Work

Research on architecture recovery spans a wide area of activities: Approaches such as Bookshelf [13], Dali [18] or Rigi [25] follow the Extract-Abstract-View Metaphor [10], and focus on the creation of condensed high-level views to facilitate program understanding. Most tools differ in the underlying fact extraction technique, in the methods and details of fact representation, and in the analysis and visualization techniques.

Cremer *et al.* [9] described a graph-based approach for reengineering COBOL programs. Since the focus of their work is on source code transformation, their visualizations are very detailed but do no support abstractions to higher levels.

Ebert *et al.* introduced GUPRO which is an integrated workbench that supports program understanding of heterogeneous systems on arbitrary levels of granularity [10]. However, it does not concentrate on the abstraction of higher-level views from source code. Moreover, GUPRO supports program understanding via textual information, but it does not include graphical representations to depict its findings.

Extracting architectural properties from large open source systems such as the Mozilla system has been addressed by Godfrey *et al.* [14]. Their work relied on PBS [13] which is a reverse engineering workbench containing the Relational Algebra tool Grok. PBS does not consider the visualization of metrics to characterize abstracted entities and relationships, or to filter out the information of minor interest leading to more condensed and comprehensible views.

Other works concentrate on diverse coupling metrics: in [6] Briand *et al.* discuss a unified framework for coupling measurement in object-oriented systems based on source model entities. Based on this metrics they verified in [7] the coupling measurements on file level using statistical methods and logical coupling information based on "ripple effects" [33]. In [2] Briand *et al.* describe how coupling can be defined and measured based on dynamic analysis of systems. This recent study shows that some dynamic coupling measures are significant indicators of change proneness and that they complement existing coupling measures that are based on static analysis.

9. Conclusions and Future Work

When only the source code is available, recovering the architecture of a large software system is a difficult task because of its sheer size and complexity.

This paper presents an interactive visual approach to architecture recovery based on package information. In the first part we propose a method of augmenting an exploration tool with complementary views of a system in such a way that it makes possible the characterization of packages and views in terms of their architectural relevance. In the second part, we propose heuristics that automate the package patterns detection, and we use these patterns to guide the user in the exploration of the system. The guidance consists in annotating the visualized packages with information of whether a certain package represents an architectural fragment relevant for the current view, or whether the user needs to drill further down into the package hierarchy.

We believe that more research needs to be done in the direction of augmenting and automating the visual exploration of software systems. For example, many of the clustering techniques generate decompositions which have a hierarchical structure, and we want to research how the clusters can be navigated using our the technique.

Acknowledgments. We want to thank the anonymous reviewers for the constructive comments. We also gratefully acknowledge the financial support of the Hasler Foundation for the project "EvoSpaces - Multi-dimensional navigation spaces for software evolution" (Hasler Foundation Project No. MMI 1976), of the Swiss National Science Foundation for the project "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1), and for the project "COSE - Controlling Software Evolution" (SNF Project No. 200021-107584/1), and "NOREX - Network of Reengineering Expertise" (SNF SCOPES Project No. IB7320-110997).

References

[1] Argouml. http://freshmeat.net/projects/argouml/.

[2] E. Arisholm, L. C. Briand, and A. Føyen. Dynamic Coupling Measurement for Object-Oriented Software. *Transactions on Software Engineering*, 30(8):491–506, 2004.

[3] Azureus. http://azureus.sourceforge.net/.

[4] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2/E*. Addison Wesley Professional, 2003.

[5] BeanShell. http://www.beanshell.org/.

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

[7] L. C. Briand, J. W. Daly, and J. K. Wüst. Using coupling measurement for impact analysis in object-oriented systems. In *Proceedings of the 21st International Conference on Software Engineering (ICSE 1999)*, pages 475–482, 1999.

[8] Columba. http://columba.sourceforge.net/.

[9] K. Cremer, A. Marburger, and B. Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance*, 14(4):257–292, 2002.

[10] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. Gupro - generic understanding of programs. In T. Mens, A. Schürr, and G. Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.

[11] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[12] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Mueller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, Nov. 1997.

[13] P. Finnigan, R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The software bookshelf. *IBM Systems Journal*, 36(4):564–593, November 1997.

[14] M. Godfrey and E. H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*, June 2000.

[15] Hipergate. http://freshmeat.net/projects/hipergate/.

[16] Infoglue. http://sourceforge.net/projects/infoglue.

[17] jEdit. http://sourceforge.net/projects/jedit/.

[18] R. Kazman and S. J. Carriére. Playing detective: Reconstructing software architecture from available evidence. *Journal of automated Software Engineering*, 6(2):107–138, April 1999.

[19] R. Koschke. An incremental semi-automatic method for component recovery. In *Working Conference on Reverse Engineering*, pages 256–, 1999.

[20] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[21] M. Lungu, A. Kuhn, T. Girba, and M. Lanza. Interactive exploration of semantic clusters. In *Proceedings of VISSOFT 2005 (3rd IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 95–100. IEEE CS Press, 2005.

[22] C. B. M.-A. D. Storey and J. Michaud. Shrimp views: An interactive and customizable environment for software exploration. In *Proceedings of International Workshop on Program Comprehension (IWPC '2001)*, 2001.

[23] S. Mancoridis and B. S. Mitchell. Using Automatic Clustering to produce High-Level System Organizations of Source Code. In *Proceedings of IWPC '98 (International Workshop on Program Comprehension)*. IEEE Computer Society Press, 1998.

[24] 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 *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[25] H. A. Müller and K. Klashinsky. Rigi – a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, Singapore, 1988. IEEE Computer Society Press.

[26] G. Murphy, D. Notkin, and K. Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of SIGSOFT '95, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28. ACM Press, 1995.

[27] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *Proc. of the 10th International Workshop on Program Comprehension*, pages 53–61, Paris, France, June 2002. IEEE Computer Society Press.

[28] C. Riva. Reverse architecting: an industrial experience report. In *Proceedings WCRE 2000*, pages 42–50. IEEE Computer Society, 2000.

[29] C. Riva. *View-based Software Architecture Reconstruction*. PhD thesis, Technical University of Vienna, 2004.

[30] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Cone trees: animated 3d visualizations of hierarchical information. In *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 189–194, New York, NY, USA, 1991. ACM Press.

[31] Softwarenaut. http://www.inf.unisi.ch/phd/lungu/softwarenaut/.

[32] M.-A. D. Storey, K. Wong, F. D. Fracchia, and H. A. Mueller. On integrating visualization techniques for effective software exploration. In *INFOVIS '97: Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis '97)*, page 38, Washington, DC, USA, 1997. IEEE Computer Society.

[33] S. S. Yau, J. S. Collofello, and T. MacGregor. Ripple effect analysis of software maintenance. In *The IEEE Computer Society's Second International Computer Software and Applications Conference*, pages 60–65. IEEE Press, nov 1978.

[34] T. Zimmermann, S. Diehl, and A. Zeller. How History Justifies System Architecture (or not). In *Proceedings of the 6th International Workshop on Principles of Software Evolution*, pages 73–83. IEEE Computer Society Press, September 2003.