MudPie:
Layers in the Ball of Mud

Daniel Vainsencher (danielv@cs.bgu.ac.il)

The *uses*-hierarchy of a Smalltalk program's packages is not generally available to its maintainer. This sets the stage for a common error - extending a low level package in a way that makes it depend on a higher level package. Such a mistake introduces a cyclic dependency, which prevents the low level package, and all others in the cycle, from being reused independently. This paper describes a tool called MudPie that uses well-known techniques to visualize the dependency structure as it is reflected in the code. We apply these techniques to Smalltalk and show how SUnit tests can detect the cycles as they occur. This can help programmers learn a system's package hierarchy and avoid breaking it.

1. Introduction

A large system is generally partitioned into packages – sets of related code definitions that are used together. Packages make it possible to understand and reuse subsets of the system that do not use and are not dependent on other packages. This mechanism is important to enabling Piecemeal Growth [1]. However, the *uses*-hierarchy of a Smalltalk program's packages is not generally available to its maintainer. This sets the stage for a common error - extending a low level package in a way that makes it depend on a higher level package. Such a mistake introduces a cyclic dependency, which prevents the low level package, and all others in the cycle, from being reused independently. This mistake is part of the process through which systems become Big Balls of Mud [1]. Antoniol *et al.* [2] have used dependency graphs and strongly connected components to show the package dependency structure as it is reflected in the code. We apply these techniques to Smalltalk and show how SUnit tests can detect the flaws causing cycles as they occur. This can help programmers learn a system's package hierarchy and avoid breaking it.

Section 2 sets the context for the rest of the paper by explaining the general problem of cyclic dependencies, and the motivation for this work. Section 3 explains what techniques MudPie applies to help solve this problem. The next two sections demonstrate its use on an existing, well known application, the Refactoring Browser [3]. Section 4 presents the main visualization and analysis features of the system, and Section 5 shows how they are used to identify design problems. The package structure of a system, as analyzed by MudPie, is also operated upon by development tools and package management tools. Section 6 explores how MudPie is integrated with those aspects of the Squeak environment, and how it might interact with different package management systems. Section 7 shows how the

cyclic dependencies analysis provides an appropriate way to continually test that packages that should be independent are in fact independent. Section 8 presents related work in program understanding and reverse engineering. Section 9 presents some conclusions and directions for future work.

2. Background

We define packages as sets of program elements that are used as a unit. Maintainers partition programs into packages in order to be able to reason about parts of the program in a local manner, without having to consider the rest of the program, and in order to make it possible to reuse subsets of the program. The usefulness of packages for these two goals depends on the packages being relatively independent of each other.

Figure 1 illustrates a program partitioned into packages A, B and C. The arc between A and B represents a dependency of package A on package B. Throughout this paper, our diagrams are generated from the code, and this dependency corresponds to the use of an element of package B in package A. We will specify the kinds of code artifacts that create dependencies in the next Section. In terms of reuse, it means that package A cannot be used unless package B is present and functional. Because it depends on B, the behavior of package A cannot be understood completely without access to either the implementation or a precise specification of B.

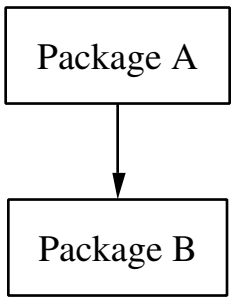

Figure 1. The arc means that package A depends on package B

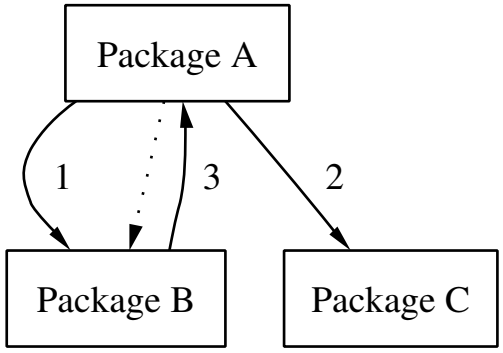

Figure 2. Three possible added dependencies. The preexisting dependency is the dotted line.

Figure 2 illustrates three kinds of new dependencies that can be introduced into this system, which have different impact on our dependency structure:

1. This dependency is equivalent to an existing one. Since it doesn't change which packages are dependent on which, it does not affect package reuse or understanding as defined above.

2. This dependency is new, and changes the graph incrementally. In our case, package A now also depends on package C, which adds constraints on the reuse of A and makes it harder to understand. However this kind of dependency is not unusual - any program that does not include dead code will contain at least as many such dependencies as there are packages, and more if some packages are reused.

3. This dependency is new, and combined with the previously existing arc, it creates a cycle in the dependency graph. This has the effect that neither A nor B can now be reused or understood independently of the other, canceling much of the benefit of having them in separate packages. This example understates the potential effect of a single "bad" dependency, because of its simplicity. A dependency from a leaf to the root of a deep dependency hierarchy will create a cycle that includes all the packages in the path between root and leaf.

This research is designed to help detect and solve dependencies of the third kind, and more generally, all cyclic dependency structures between packages in a program, which when present make it harder to understand and reuse a subset of the program.

The original motivation for this work arose from a long-standing project to modularize Squeak. Squeak is an open source Smalltalk system initially bootstrapped using Apple Smalltalk [4]. It was significantly extended by Squeak Central (a research team at Apple and later at Disney) and, since its release in 1996, also by a wide community that has formed around the project. The first design principle for Smalltalk was "Personal Mastery: If a system is to serve the creative spirit, it must be entirely comprehensible to a single individual" [5]. However, by 2001 the accumulated effects of contributions by the community had made Squeak quite complex.

As people added features and integrated them into the system, basic classes such as Object and Morph collected methods that referenced application-specific classes. These dependencies are in the wrong direction in the sense mentioned above - the basic classes of Squeak should be reusable, regardless of the various applications added to it.

A specific concern was that there was no clear way to create subsets of Squeak, for example a Squeak for business applications that would not include EToys, the scripting system designed for children. The system had included from the start "discard scripts" that stripped various subsystems from the image. But these were fragile and didn't cover all subsystems. The community recognized a better solution is to build up images in a repeatable way from packages that combine cleanly, choosing the subset one wants. This would also have the advantage of allowing separate maintenance of application projects, without having to wait for integration into the image.

This realization motivated two efforts to partition Squeak into packages – the Ginsu [6] portion by Paul McDonough and Joseph Pelrine, which was part of the Squeak World Tour [7] by John Sarkela, and 3.3a Modules by Henrik Gedenryd. While different in approach, both were designed to add a complete package handling system. For various reasons, neither of these approaches were accepted by the community. One reason was that to gain any benefits from their use, they both required significant changes to the that way people use, distribute and contribute code. The creation of MudPie, the tool described

in this paper, began as a response to the perceived constraint that a tool to help analyze the existing code should not require significant changes to other aspects of development.

3. What is MudPie

MudPie is a tool to help maintainers solve the problem described in the previous section. When using MudPie, a programmer defines the code he is interested in by enumerating the packages that should be included in the analysis. MudPie represents dependencies in the code as a graph, allowing the programmer to perform the following kinds of queries and operations.

1. Compute the strongly connected components (SCCs) of the dependency graph.
2. Compute a graph that has the SCCs as nodes and the dependencies between SCCs as arcs. This identifies groups of packages that cannot be reused independently.
3. Generate diagrams of graphs using Connectors, a graphing framework for Squeak. Clicking on the edges shows the code that causes the dependencies they represent.
4. Generate graph descriptions for the GraphViz [8] set of graph processing programs. These perform automatic layout, and generated the figures in this paper.
5. Report some interesting subsets of dependencies, such as direct dependencies or shortest paths of dependencies between sets of classes or of packages.
6. Test whether a number of classes are all in separate SCCs, verifying that important independencies are in still in effect.

It is important to remember that because of the dynamically typed, polymorphic and reflective [9] nature of Smalltalk, the dependencies detected cannot generally be precise, as discussed by Wilde and Huitt [10]. MudPie uses only immediately available static dependencies, which result from three kinds of constructs.

1. Explicitly named references to classes in method implementations. These make the package defining the method dependent on the package defining the named class.
2. Class declarations specify a defined class and its superclass. This makes the package defining each class dependent on the package defining the super class.
3. A package can define methods that are associated with a class that it doesn't define. These are called *class extensions*, and make the package defining them dependent on the package that defines the class to which the methods belong.

The existence of class extensions in Smalltalk is important for structuring dependencies correctly (as discussed in the next two Sections) and depends on the packaging concepts in the dialect. Support of class extensions in Squeak and other dialects is discussed in detail in Section 6.

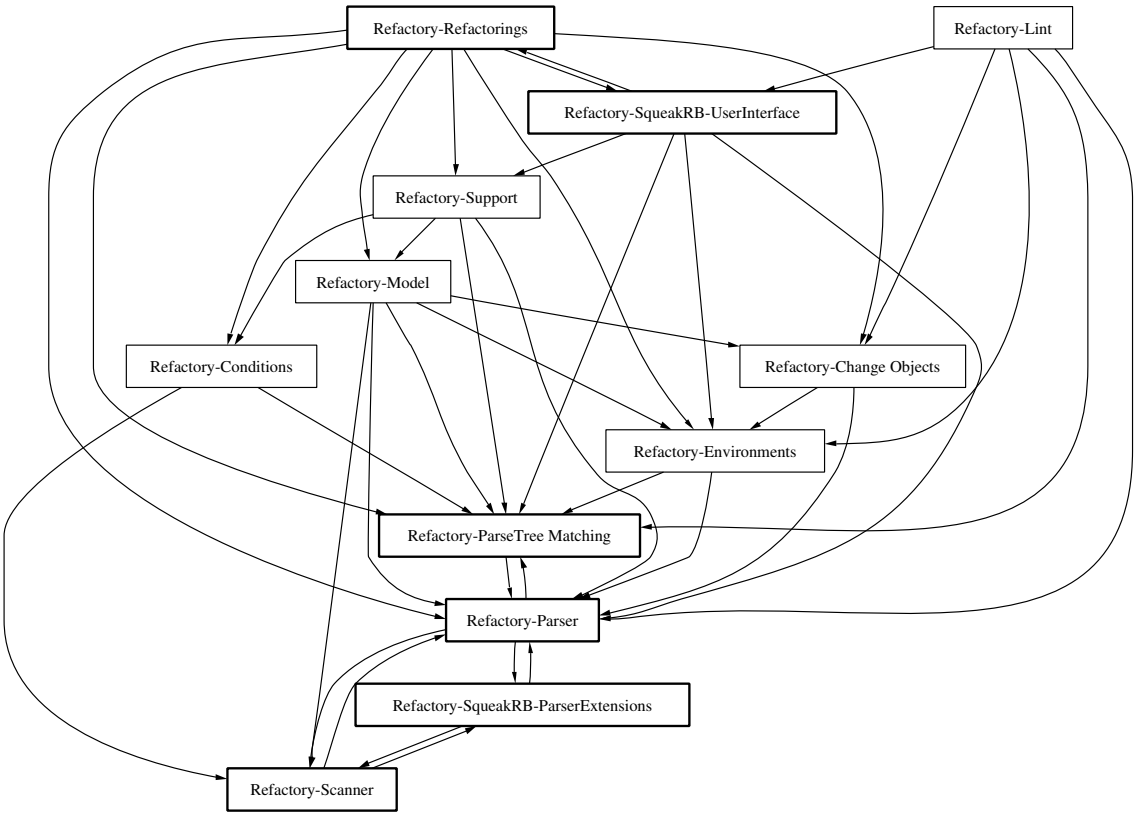

Figure 3. Packages of the Refactoring Browser. Packages involved in cyclic dependencies have bold outlines.

4. Understanding an Existing System

We present the use of MudPie by using it to take a look at an existing system, namely, the Refactoring Browser. The Refactoring Browser was originally developed for VisualWorks Smalltalk. It has since been ported to many platforms, including Squeak. The port to Squeak included quite a few changes by different people including the author of this paper. The Refactoring Browser code is divided into class categories according to function. The Squeak port currently contains 223 classes in 16 different class categories, and 3323 methods.

Before focusing on any one part of the system, our first step is to understand how the system is divided into packages, and what the dependency relationships between those packages are. Extracting all the dependencies from the code of the Refactoring Browser, we model them as a graph. This graph is shown in Figure 3 (for clarity, we exclude throughout three test-related packages).

Note that the graph is a general directed graph containing cycles. This is in contrast with the desired acyclic structure for the "uses" relation. These cycles make the graph

harder to deal with in several ways.

1. It becomes harder to reason about the relations between packages, for example, to say whether a package can be removed or changed without affecting another critical package, or which other packages we need to understand to see what a specific package does.

2. It becomes harder to create a conceptual structure that can be reasoned about, such as a division into layers [11].

3. At a technical level, the diagram becomes harder to lay out properly, either manually or automatically. This might seem a trifle, but it makes the information that it contains harder to deal with.

Our next step is to separate the cycles from the rest of the structural information inherent in this graph. This is done by isolating the strongly connected components in the graph. An SCC is a subgraph in which there is a directed path between every pair of vertices [12]. Finding the SCCs is a well-known problem in graph theory with known linear-time algorithms, for example by R. Tarjan [13]. In our terms, each SCC consists of a set of packages. Each package is mutually dependent on every other package in the same SCC. Therefore, each SCC represents a set of packages that cannot be used separately. If any package in the SCC is changed, every other package in it can be affected. Therefore large SCCs indicate a fragile part of the design.

Figure 3 shows, in the packages marked in bold, that the system we are studying has two problem areas: one showing a mutual dependency between the UI and the Refactoring engine (clearly undesirable, since the functionality represented by a refactoring engine should be available for reuse by a different UI), and another SCC with four packages (near the bottom of the diagram).

Constructing a graph of SCCs using the union of the contained packages' dependencies results in a directed acyclic graph (DAG), because the SCCs encapsulate all mutual dependencies. Visualizing this graph exposes the SCC dependency structure, in which cyclic package dependencies are made explicit.

The SCC diagram in Figure 4 outlines where the problems are, and what the structure of the rest of the system is, by hiding the internal structure of each problem SCC. In the next section, we consider how reasoning about dependency graphs and cycles may help in understanding and dealing with such problems.

5. Refining an Existing System

We shall first examine the simpler of the two problems, that of the SCC containing the UI and Refactorings packages. It is clear from their names that the UI package should use the Refactorings, because the main feature of the Refactoring Browser is that it allows a user to operate the Refactorings engine. This knowledge, combined with the determination that dependencies should be acyclic, implies that the Refactorings package should not depend

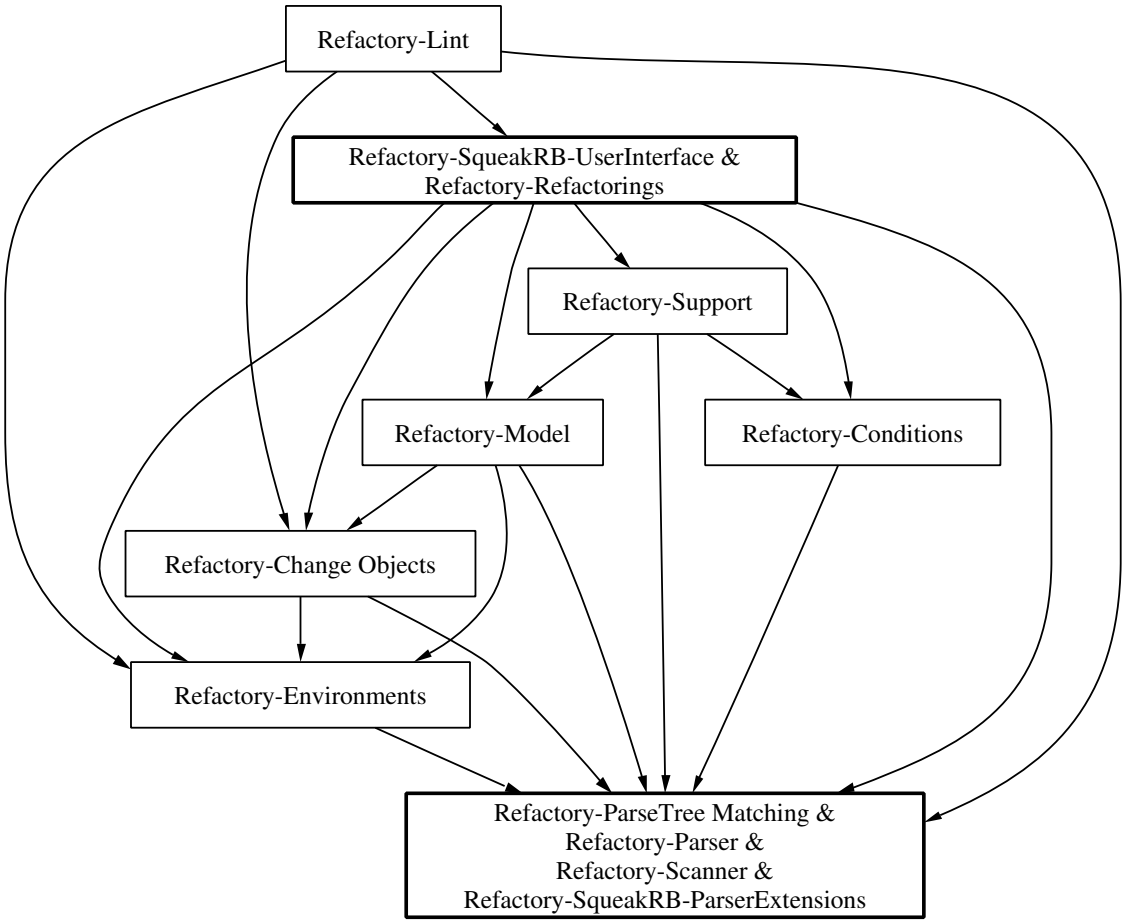

Figure 4. SCC diagram for RB. Each node is an SCC of packages. SCCs that contain more than one package create limits on reuse.

on the UI package. Now the problem that remains is to find the cause of the dependency. We are looking for one of the following:

1. A class in package Refactorings that inherits from a class in package UI;
2. A method defined in package Refactorings that references a class in package UI; or
3. A method defined in package Refactorings that extends a class in package UI.

We query MudPie for all paths from the Refactorings package to the UI package and find only one such dependency. It is a class initialization method in the class Refactoring, that refers to two classes that represent dialogs. Refactorings invoke these dialogs to request additional information from the user. This represents an issue in the design: it ties the engine to use this specific UI. This is a reasonable design decision when creating a single product, but it is less appropriate when the project is thought of as a potential product family (as in fact it has become – there are different UIs in various Smalltalks). We implement a simple solution, by moving the class initialization method to be a class extension in the UI package. This allows one to create alternative, mutually exclusive UIs, each defining its own hooks, but makes the Refactoring package incomplete unless another package provides the class initialization class extension. For this and other reasons, this might not be the best design solution in practice, but it does solve our stated design and dependency problems. Figure 5 is the updated SCC diagram, after having implemented this correction.

Before we go into the next, more complex example, let us reflect on the steps that we went through.

1. We reasoned about the systems architecture, using the dependency diagrams, and identified the improper dependency chain;
2. We identified the causes in the code for that dependency, in this case by a simple query;
3. We read the code to understand the design used, and implemented an alternative design.

We will now look at the SCC that contains the Parser, Scanner, SqueakRB-Parser extensions and ParseTree Matcher SCCs, as seen in Figure 6.

This SCC poses a harder problem than the previous one. It includes more packages, and also, the correct dependency directions are less clear than in the UI/Refactorings case. This case will require more knowledge about the specific design.

Arbitrarily, we shall focus on the Parser and Scanner pair. Listing the dependencies in both directions (since we don't know *a priori* which is correct), we find 12 different methods that contain dependencies from Parser to Scanner, and one method containing a dependency in the opposite direction. Even without reading the code, we should almost

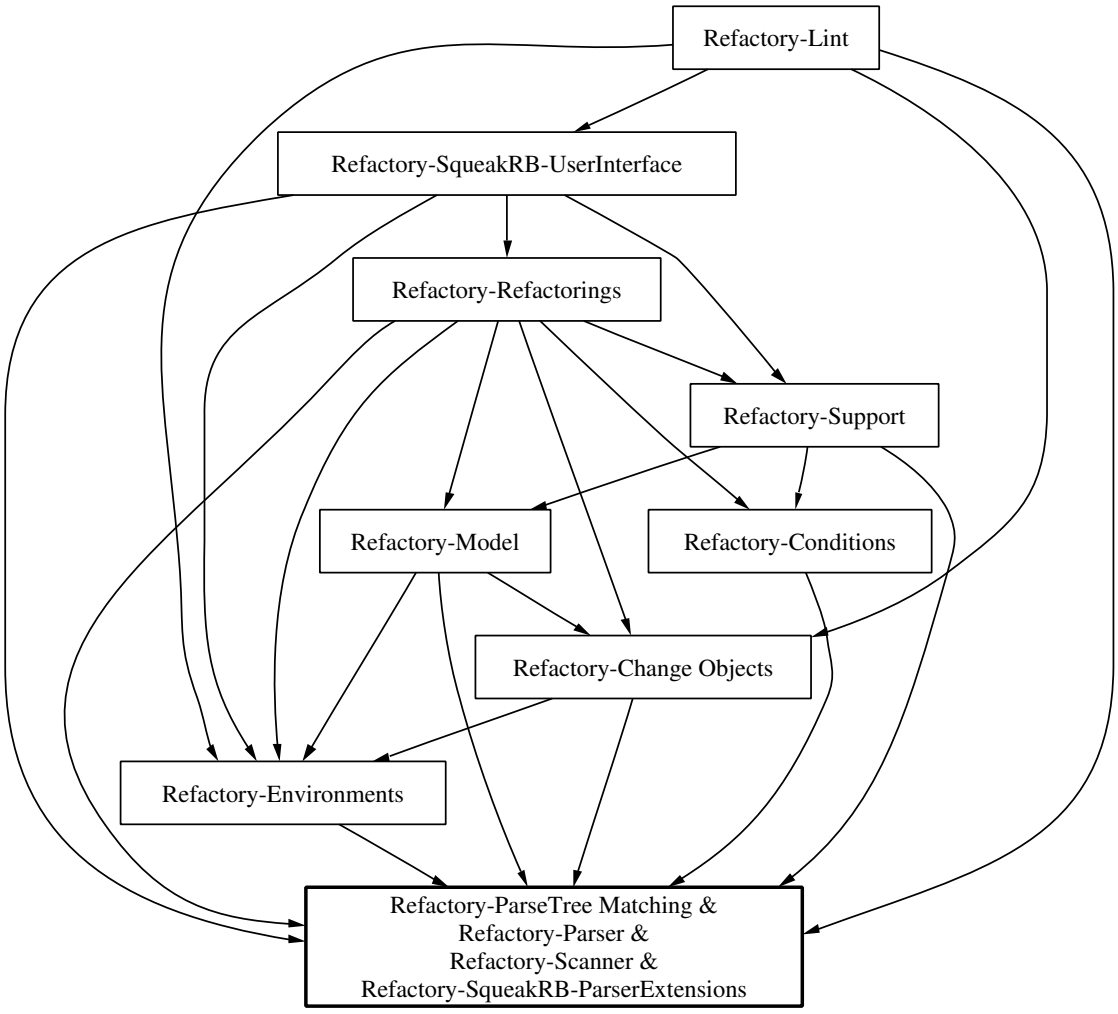

Figure 5. The RB SCC diagram after removing the dependency of Refactoring on UI.

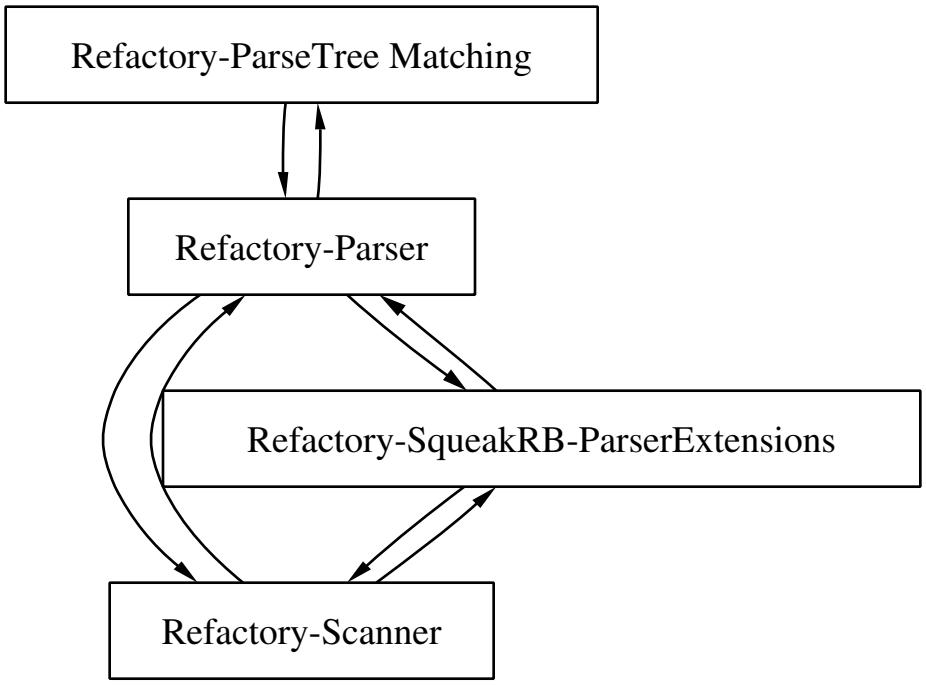

Figure 6. Internal dependencies in the second RB SCC.

certainly remove the dependency of the Scanner on the Parser – it is far more likely to be an isolated flaw.

Reading the code for the specific method, we see that each RBScanner, during its initialization, asks the RBParser class which dialect it is running in, in order to configure itself for small differences in the available tokens. This dependency is quite easy to change, and is the one we choose to refactor. Note that in this case, it is clear that the flaw is the result of maintenance - when it was first designed, and the structure of the system defined, there was no question which dialect it was running on. Section 6 explores a way to warn about this kind of degradation as it happens.

We draw some conclusions from this example:

1. When refactoring the dependencies between two packages, there is a tradeoff between invested time (if we try to learn everything about the packages involved) and risk of error (if we discover additional connections after refactoring has started). By querying the dependency relations in terms of graphs, we obtain the Big Picture efficiently. This makes it easier to make decisions in the refactoring process. It doesn't eliminate the need to read the code, but it adds options to the cost/risk tradeoff.

2. The overall dependency structure can help us prioritize the problems. When faced with multiple different issues, we need to decide the order in which to tackle them. In this case, tackling the Parser/Scanner pair was probably the wrong idea, for two

reasons. The first reason is that we still cannot separate any two packages we couldn't before, because Parser and Scanner are mutually dependent through the SqueakRB-ParserExtensions package. This is clearly visible in the dependency graph.

3. The SCC diagram, along with architectural knowledge, may imply that the package partition itself is wrong. The second reason for which working on the Parser/Scanner issue was a mistake is that the Parser/Scanner/SqueakRBParser Extensions can be reused as a unit for other projects, but are hard to use separately. We might clarify this by unifying these packages.

6. Relationship to Package Management Tools and Concepts

MudPie is designed to solve a specific problem that relates to dependencies between Packages. Other mechanisms exist that solve related problems: it is worth understanding what they are and how they might interact with MudPie.

First, there is the problem of package definition, or in other words, partitioning code as a unit for reuse and understanding. Squeak provides a very simple model for organizing code, in which each class is defined in exactly one class category. Other Smalltalks and package systems have different concepts that provide more freedom. VW Parcels [14] and ENVY Applications [15] both define an abstraction grouping code definitions, and both support the idea of class extensions.

As a simple example of the cases for which class extensions are important, consider the method asUrl, which converts the recipient, a String, into a URL. This method, while it represents behavior that should be defined on class String, is only meaningful in the presence of the URL handling package. This case would be modeled by placing the asUrl method in the URL handling package as a class extension of String.

Because this finer level of granularity is important for allowing packages to be correctly integrated (see the UI/Refactorings discussion in the previous Section), proper use of Mud-Pie requires an enhanced code model that supports it. In the current MudPie version, this is provided by the package PackageInfo [16], but MudPie could be modified to use other partitioning schemes that provide class extensions.

Using the existing code model, extended by PackageInfo, has the following advantages:

1. It leverages existing tool support, provided by multiple browsers, file-in and file-out mechanisms. The existing Browsers display the elements of the code model very prominently, and the operations for moving code between packages are mostly drag and drop.

2. The accepted practices and ideas of the Squeak community are extended in a way that is easy to understand. Maintaining code within packages does not automatically add any learning overhead for the casual programmer.

3. No extra effort is required before starting to create code, so that there is no bias towards up-front over incremental design.

Another related problem is that of package management - that is, given a definition of the contents of a package, support is needed for its storage, for loading it and often for comparing it to find common or different parts between different packages or different versions of the same package. MudPie does not affect these issues, it relies on support for this kind of operation provided by PackageInfo and related tools. We mention this as an area of interaction between MudPie and other tools for two reasons. The first is that it is important that one's analysis tools and package management tools use the same package boundaries. Otherwise, the results of the analysis will not be directly applicable to the packages as they are used. The second reason is that package management tools often require the declaration of dependencies in order to ensure that packages are loaded only in the right order - for example, that a package including class extensions is loaded only after the package that defines the class that is to be extended.

The use of dependency declarations by package management systems implies that definitions in each package can only rely on elements defined in itself or in the packages on which it declares dependency. Note that this in effect controls dependencies of the second kind mentioned in the Section 2. MudPie can be combined with this kind of package management system, as long as it is adapted to use the same package definitions.

In some way or another, package management systems that regulate the load order using dependencies must deal with the issue of cyclic dependencies, because such cycles prevent any valid load order from existing. For example, according to Dolphin-Wiki [17], it is impossible to save cyclically dependent Dolphin packages. MudPie thus complements such systems when dealing with cyclic dependencies - the visualization features of MudPie show these problems in the wider system context, instead of focusing on the specific package. As the next section details, MudPie can also be used to warn about circular dependencies immediately as they occur, before meeting the package systems constraints.

7. Testing the Dependency Structure

The previous section mentioned the importance of discovering cyclic dependencies between packages early. Extreme Programming and other methodologies stress the importance of unit tests to document and maintain the functionality of the system. This section shows how we can use SUnit [18] tests to document and maintain the system's (in)dependency structure. First we need an appropriate way to express the important assertions about package structure.

If we express the allowed dependencies in terms of the classes at the boundaries of the packages (for example, asserting that only specific classes in the package may be referenced from outside the package, or that a package may reference only specific external classes), these definitions will be fragile when refactorings change the package boundaries. For example, classes implementing a cache strategy that used to serve (and be defined in) a specific part of the application (in which the optimization became important first) might later become infrastructure and move into a database layer.

Even if we use a coarser granularity, defining dependencies in terms of whole packages (for example, using the kinds of definitions we described for package systems - this package

depends on these other specific packages), our definitions are still fragile under reasonable changes. Specifically, such a definition would still be broken by refactorings that make the structure more indirect without actually harming conceptual integrity. In the same example, a Caching package might be created between the application and database layer.

How can we express in a robust way the minimal assertions that define the required independence between packages? To protect against the boundaries shifting, we will make the assertion in terms of core classes that are not likely to move between packages. To maintain validity in the face of additions and other reasonable evolution in the package structure, the test must remain abstract - not repeating the information contained in the dependencies it is supposed to test.

Using the MudPie API, we can assert that "Each of the classes AbstractModelBase, MainUI and DBConnection is in a separate SCC". In general terms, we choose a core class in each package, and assert that they are also in separate SCCs. This statement denies cyclic dependencies between packages, keeping the low level packages reusable, without depending on specific details of the package division. The simple test shown below would have warned about the flaws we mentioned in Section 4.

self assert: ((MultiModuleAnalyzer onCategoriesWithPrefix: 'Refactory') areInDisjoint-Components: #(RBScanner RBParser Refactoring RefactoringBrowser))

8. Related Work

The approach described in this paper has the most in common with that taken in the CANTO [2] module view. CANTO is a program-understanding framework that models C programs. It considers files as modules (used in the sense that the term package is used in this paper), and creates both a call graph and a data-flow graph. These are combined into a graph representing dependencies of both types between the different modules. The module dependencies graph can be visualized, and an operation is provided to cluster strongly connected components [19] (see Section 3). This work adapts similar techniques to the Smalltalk language and programming environment, and uses them to express tests of the dependency structure of the system.

Ginsu and 3.3a Modules both added package/module entities, for which dependencies have to be explicitly defined by the user. The systems then warn about dependencies in the code beyond those (as discussed for package systems in general in the related concepts Section). MudPie instead detects cycles (however indirect) in the dependencies found in the code. Given the expected many correct dependencies, the "reversed" dependencies are likely to show up as cycles. In Smalltalk, this use of cycles allows the verification of a proper dependency structure without requiring a static declaration of dependencies. It thus retains flexibility, and adds no cost to the common case of correct, but unannotated code. MudPie is also consistent with the spirit of Smalltalk in avoiding declarations that can become stale. By showing cyclic dependencies as flaws, it makes it easier to maintain the acyclic structure of the *uses* relation [20], thus making it easier to reuse subsets of the software.

There is another difference between MudPie and the previous systems described. If an

analysis is done only at the level of packages, it is hard to provide helpful information in the case that the existing partitioning of the code is not aligned with dependencies. Unfortunately, this is the state of Squeak. While the examples in this paper focus on the smaller, simpler task of maintaining a smaller, well partitioned system, initial work shows that performing the dependency analysis at the class level provides useful information in the more general case.

Graph-based dependency models are used for different applications of program analysis. These applications include software maintenance [21], regression testing [22,23], and also optimization and debugging. Class Blueprints [24] allow the visualization of references in Smalltalk programs, focusing on a small number of selected classes.

9. Conclusions and Future Work

This paper describes how MudPie can aid in the maintenance of a software system, while adding only the requirement of defining clearly the packages, without requiring a specific package management system. It is designed to add value to the set of different package handling solutions being adopted by the Squeak community (DVS [25], SqueakMap [26] and others), without interfering with them.

An important assumption in this work is the set of dependencies used. By choosing the static forms of dependency, we conform to an easily understandable model for any Smalltalk programmer. This model is also simply and precisely computable, which is both a benefit by itself, and serves to keep the model understandable. On the other hand, adding dependencies for specific cases that can be easily detected (sending a message only implemented by one class) would make the model correspond more closely to the actual dependencies revealed at runtime. Another approach to this would be integration with a type inference engine for Smalltalk. This tradeoff between preciseness and understandability would require careful evaluation.

A limitation in the current implementation is that the graph model is not kept synchronized with the Squeak code. Scanning the Squeak code is the slowest stage in the analysis (taking around a minute for a complete 3.5 image). If the code and model were kept synchronized, tests would be cheaper to run and visual models could provide near real time feedback on a refactoring efforts. This would require separating the definition of the model boundaries (which packages are interesting) from the model building, which would allow the queries to operate on cached information that is invalidated when code changes.

A different future direction would be to make the feedback provided by MudPie more immediately available. For example, by annotating edges with the numbers of dependencies they represent, some links would be more quickly revealed as erronous. More ambitiously, we could try to suggest to the programmer which dependencies should be removed in order to break cycles.

In order to support the work of modularizing Squeak at larger scales than the application level discussed in this paper, it is likely that higher level tool support would be needed. One direction this could take would be to provide means for planning the refactorings needed using scenarios, and for tracking them using different kinds of dependency tests.

An interesting extension of this work would be to use the hierarchical structure and dependencies recovered, to highlight and order elements in the Browser, making this information more available, and the relevant parts of the code more visible.

MudPie is freely available and posted on SqueakMap, so that those interested in evaluating or using it can either use the web interface to obtain the code or they can install a recent Squeak and then install MudPie using the package loader.

10. Thanks

I thank first Squeak Central and the wide Squeak community, to whom this work is dedicated. Also I owe a debt to those that worked on this problem in Squeak before, and from whom I learned about it, in particular to Joseph Pelrine that gave me my initial useful idea of what packages are for, and how they should fit into a Smalltalk environment.

This system uses packages by several people - Ned Konz's Connectors for visualization, Avi Bryant's PackageInfo to complete the code model.

Reviewers of various versions of this paper who have given me important advice and comment include Uri-David Akavia, Alon Keinan, Shahar Siegman and the commitee reviewers. I am especially grateful to Andrew Black, Stephane Ducasse and Mayer Goldberg, for many useful comments, suggestions and additional references that served to improve this paper.

REFERENCES

1. B. Foote and J. W. Yoder. Big Ball of Mud. In Proceedings of PLoP'97, 1997.
2. Antoniol, G.; Fiutem, R.; Lutteri, G.; Tonella, P.; Zanfei, S. Merlo, E., "Program understanding and maintenance with the CANTO environment," International Conference on Software Maintenance, p. 72-81, Bari, Italy, 1-3 Oct. 1997.
3. Don Roberts, John Brant, and Ralph E. Johnson. A refactoring tool for Smalltalk. Theory and Practice of Object Systems (TAPOS), 3(4):253–263, 1997.
4. Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace and Alan Kay, Back to the Future: the Story of Squeak, a Usable Smalltalk Written in Itself, Proc. OOPSLA '97. Published as SIGPLAN Notices 32(10):318–326.
5. Daniel H. H. Ingalls. Design Principles Behind Smalltalk. Byte Magazine, 6(8):286–298, August 1981.
6. Joseph Pelrine. ModSqueak. http://swiki.squeakfoundation.org/stablesqueak/6
7. John W. Sarkela. Stable Squeak World Tour. ESUG 2000.
8. Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. Software - Practice and Experience, 1999.
9. Brian Foote and Ralph E. Johnson. Reflective facilities in Smalltalk-80. In Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, pages 327-336, October 1989.
10. N. Wilde and R. Huitt. Maintenance support for object-oriented programs. IEEE Transactions on Software Engineering, 18(12):1038–1044, December 1992.

11. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stad. Pattern-Oriented Software Architecture - A System of Patterns. Wiley, 1996.
12. Steven S. Skiena. The Algorithm Design Manual. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc. p. 270–272, 1998.
13. R. Tarjan. Depth-first search and linear graph algorithms. SIAM J. Computing, 1:146-160, 1972.
14. Eliot Miranda, David Liebs. Parcels: a Fast and Flexible Component Architecture. Unpublished article available at: http://wiki.cs.uiuc.edu/VisualWorks/DOWNLOAD/ecoop99-parcels.rtf
15. Joseph Pelrine and Alan Knight. Mastering ENVY/Developer. Cambridge University Press, 2001.
16. Avi Bryant. PackageInfo: Declarative Categorization of Squeak Code. http://beta4.com/squeak/aubergines/docs/packageinfo.html
17. Unknown authors. CycliclyDependentPackages, Dolphin Wiki, http://www.object-arts.co.uk/wiki/html/Dolphin/CycliclyDependentPackages.htm.
18. Kent Beck. "Simple Smalltalk Testing", The Smalltalk Report, October 1994.
19. R. Fiutem; E. Merlo; G. Antoniol; P. Tonella. Understanding the Architecture of Software Systems. WPC '96: Proceedings of the IEEE Fourth Workshop on Program Comprehension. IEEE Computer Society Press. March 1996.
20. D. L. Parnas. Designing software for ease of extension and contraction. IEEE Transactions on Software Engineering, 5(2), 1979.
21. M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: a system for the development of program-analysis-based tools. In Proceedings of the 33rd Annual Southeast Conference, p. 110–119. ACM Press, March 1995.
22. D. J. Richardson. "TAOS: Testing with Analysis and Oracle Support". In Proceedings of the 1994 International Symposium on Software Testing and Analysis, Aug. 1994.
23. M.J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith. Aristotle: a system for the development of program-analysis-based tools. In Proceedings of the 33rd Annual Southeast Conference, p. 110–119, March 1995.
24. Michele Lanza and Stéphane Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. Proceedings of OOPSLA 2001, p. 300-311
25. Avi Bryant. Concurrent Squeaking with DVS. http://beta4.com/squeak/aubergines/docs/dvs.html.
26. Goran Krampe. SqueakMap. http://minnow.cc.gatech.edu/squeak/2726