

Knowledge-Based Library Re-Factoring for an Open Source Project

M. Di Penta*, M. Neteler**, G. Antoniol*, E. Merlo***

dipenta@unisannio.it, neteler@itc.it, antoniol@ieee.org, ettore.merlo@polymtl.ca

*RCOST - Research Centre on Software Technology

University of Sannio,

Department of Engineering

Palazzo Bosco Lucarelli, Piazza Roma, 82100 Benevento, Italy

**ITC-irst

Via Sommarive 18

38050 Povo (Trento), Italy

***École Polytechnique de Montréal
Montréal, Canada

Abstract

Software miniaturization is a form of software re-factoring focused on reducing an application to the bare bone. Porting an application on a hand-held device is very likely to require a preliminary step of software miniaturization, plus the development of device drivers dedicated to the new environment and hardware architecture.

This paper presents the process and the lessons learned re-factoring a large Open Source application to get rid of extra fat, to introduce shared libraries, to remove circular dependencies among libraries and, more generally, to minimize inter-library dependencies.

While the final goal was to fully exploit shared libraries capabilities, among the various possibilities we defined a process based on the existing knowledge about the application, and aimed to minimize the maintenance effort required by the miniaturization activities.

Keywords: re-factoring, re-modularization, clustering, software evolution

1. Introduction

Going wireless is one of the new software trends and industry hip; the wireless technology has reached a turning point, where complex applications can be ported on hand-held devices, such as Personal Digital Assistants (PDA).

Wireless communication based on GSM, iDEN, GPRS (part of the so called 2.5G technology [8]) will be further ameliorated by the new third generation protocols promising higher bandwidths.

Moreover, the 2.5G technologies offer enormous possibilities; as an example, GPRS provides up to 171 Kbits/sec.; TCP/IP (PPP) over GPRS allows to connect via cellular phone a laptop or a PDA to information providers, exchange data, download music, images, update databases, or multimedia documents.

To go wireless applications face the miniaturization challenge: the extra fat must be eliminated. Hand-held device resources are limited so that space (i.e., persistent as well as run-time memory) and bandwidth need to be optimized. Software miniaturization can be thought of as a software re-factoring where the focus is on the optimization of resource usage. This particular form of software re-factoring is not peculiar to industrial software, the new trend involves Open Source. Very recently, mixed solutions also appeared; IBM announced a Linux watch [19], a wristwatch computer running Linux, X11 graphics and offering Bluetooth wireless connectivity.

Comparing to proprietary source software (such as most industrial projects), the Open Source movement focuses on different aims. While license fees are requested for proprietary software, Free Software cannot be sold. Software is considered as service, not as product. Money is earned from selling service for the software. Major advantage is the peer review within Open Source projects since the code can be scrutinized by anyone, algorithms are verified and eventually corrections applied. By sharing ideas through code the overall knowledge is broadened, meanwhile, Open Source movement poses new challenges.

Two main development models were identified in [22], the “cathedral type” (proprietary projects) and the “bazaar type” (Open Source projects) development. This has been widely discussed elsewhere [22]. Due to the intrinsic non-organizational nature of Open Source projects, reducing ap-

plications to the bare bone, applying miniaturization processes is even more challenging.

In this paper we present our experience in re-factoring one of the largest Open Source software systems: a Geographical Information System (GIS) named *GRASS* (Geographic Resources Analysis Support System, <http://grass.itc.it>). *GRASS* is a raster/vector GIS combined with integrated image processing and data visualization subsystems [21]. The current *GRASS* development model can be considered as “council type”. The number of team members is small (7-15 active developers), decisions are usually taken by the members most capable of a certain problem. A voting system, as known for the Apache project [18], has not been established at time of this writing. The developers are also users of the system, often focusing on their needs within the general project framework.

GRASS re-factoring aimed at promoting software miniaturization, reducing code duplications, eliminating unused files while restructuring libraries and reorganizing them into relocatable Dynamic Linkable Libraries (DLLs, also referred to as shared libraries).

Normally, a program is statically linked, and thus each executable has its own copy of each library linked. At run-time, the entire object is loaded, including the statically linked libraries. This is also the most common use of shared libraries, in that a single copy of the library exists (thus saving persistent storage), however, at loading time, each loaded executable will be dynamically linked with the entire set of DLLs specified at compile time (or recursively required to resolve symbols). Of course, if two DLLs do not exhibit interdependencies, and an application requires one of the two DLLs, there is no need to link both DLLs to compile the application.

Beside the obvious case, where no dependencies between libraries exist, source code could be restructured relying on the IEEE Std. 1003.1-2001 (the Open Group Base Specifications, Issue 6). The calls *dlopen()*, *dlsym()*, *dlclose()* provide a means to control dynamic loading so that only relevant shared libraries can be loaded (and unloaded). In other words, there might exist tens of DLLs, but only those really needed will be placed in memory at any given time, making the application use less system resources. This, however, requires code transformation, to insert *dlopen()*, *dlsym()*, *dlclose()* primitives where needed.

While the final goal was to obtain an application making the most parsimonious use of resources, there may be different ways to achieve the same goal. For example, it may be desirable to minimize the time-to-market (e.g., just remap static libraries into DLLs), or resource constraints may force to break static libraries into smaller DLLs eliminating circular dependencies between libraries or reducing the inter library dependencies.

The approach adopted to miniaturize *GRASS* was a com-

promise, attempting to minimize the maintenance effort required to perform the re-factoring process while reducing the time-to-market, under the constraints of eliminating circular dependencies, introducing shared libraries and removing unused object or redundant code. The latter activity was limited to the source code linked into libraries. In the meantime, a brute force approach would be not feasible, given the size of the software system. Thus, the process was based on the already existing knowledge of the application structure, functionalities, and, more generally, software architecture. The application software architecture and the functionalities were used to partition the application source code into components; each component was, in turn, re-factored considering the dependencies within and among components.

The paper is organized as follows. First, the re-factoring activities are described in Section 2. Information on the case study (i.e., *GRASS*) are reported in Section 3, while implemented and adopted tools are described in Section 4. Section 5 presents experimental results, then discussed in Section 6. Finally, an analysis of related work is reported in Section 7, before conclusions and work-in-progress.

2. Re-Factoring Process

As highlighted in the introduction, this paper aims to reorganize the *GRASS* libraries, in order to minimize the amount of resources required when each executable is loaded in memory. This will allow porting the GIS on resource-limited devices, such as palmtops and, in general, to make it lighter, faster and less resource-aware.

Several operations should be performed to obtain significant results. In particular:

- Circular dependencies among libraries must be removed, or, at least, reduced to the minimum. In fact, these dependencies causes a library to be linked each time another one (circularly linked to it) is needed;
- Duplicated (cloned) objects must be identified and, whenever possible, factored out;
- Unused functions/objects should be removed from libraries; and
- Large libraries should be re-factored into smaller ones and, if possible, transformed into dynamic libraries.

The re-factoring process followed the steps detailed in the next subsections.

2.1 Application Identification

Prior to recover dependencies among applications and libraries, and among libraries themselves, applications (i.e.,

executables) composing the software system must be identified. In this paper an approach similar to the one used in [5] was used.

However, in [5] applications were identified detecting all source (.c) files containing the definition of a main function. We experienced that in GRASS there was not always a one-to-one correspondence between source and object files, therefore following that approach could have produced problems in mapping the source file of the main to the corresponding object. Instead, objects defining the main symbol were directly searched and mapped to applications.

2.2 Dependency Graph Construction

Once applications and existing libraries were identified (for the latter the identification process was trivial, in that it consisted in simply searching for .a files), a dependency graph was built. This (direct) graph is defined as follows:

$$DG \equiv \{A, L, D\} \quad (1)$$

where:

- $A = \{a_1, a_2, \dots, a_m\}$ is the set of nodes representing the *applications*; each node is labeled with the application name;
- $L = \{l_1, l_2, \dots, l_n\}$ is the set of nodes representing the *libraries*; each node is labeled with the library name; and
- $D \subseteq \{A \times L\} \cup \{L \times L\}$ is the set of oriented edges $d_{i,j}$ representing dependencies between the application a_i , or the library l_{i-m} (if $i > m$) and the library l_j ($j = 1 \dots n$); each edge is labeled with the name of the symbol an application (or another library) requires from a library.

Given the *use* relation between an object module requiring a symbol and a module defining it, the dependency graph was built determining the *transitive closure* of the *use* relation, starting from the main object of each application and from each library.

In other words, for each application, undefined symbols were identified and recursively (in that new undefined symbols were added to the stack) resolved firstly inside the objects contained in the same path (i.e., other modules of the application), then inside libraries. A similar process was performed to detect dependencies among libraries.

Similarly for DG , an *Internal Dependency Graph* (IDG) was recovered. Such a graph identifies dependencies at object grain-level, and is defined as follows:

$$IDG \equiv \{A, O, D\} \quad (2)$$

where

- $A = \{a_1, a_2, \dots, a_m\}$ is again the set of nodes representing the *applications*;
- $O = \{o_1, o_2, \dots, o_j\}$ is the set of nodes representing the *objects* composing libraries j . Each node is labeled with the object name, and with the name of the library the object belongs to; and
- $D \subseteq \{A \times O\} \cup \{O \times O\}$ is the set of oriented edges representing dependencies among applications and objects, and among objects themselves.

2.3 Removal of Circular Dependencies among Libraries

The analysis was performed on the sub-graph of DG showing dependencies among libraries (i.e. only $D' \subseteq \{L \times L\}$ edges).

Once cycles were identified in that graph, four choices could be taken, with the aid of the IDG , to remove circular dependencies:

1. *Move the object* causing the circular dependence from a library to another. This is only possible if the object does not need resources located in its original library, nor it is required in that library. For example, in Figure 1-a, object o_1 should be moved from library L_1 to library L_2 ;
2. *Duplicate the object*: like the previous case, this is possible if the object does not need resources located in the original library but, differently from the previous case, it is required in that library (therefore moving it outside is not possible). In Figure 1-b, object o_1 should be duplicated in library L_2 (it cannot be moved, in that o_2 depends on it);
3. *Merge the two libraries*: this strategy should be avoided whenever possible; however, it could be the only available solutions when the number of objects causing circular and, in general, inter-library dependencies is very high;
4. *Make dynamic libraries*: instead of merging circularly dependent libraries, one may decide to make them dynamic. Circular dependency problem is not solved, but the average amount of resources needed is reduced (see Section 2.6 for details).

When the dependency graph does not allow to resolve circular dependencies and, for performance reasons, options three and four cannot be adopted, a deeper analysis should be performed, identifying dependencies at function grain-level instead of object grain-level. This should ease the removal of some critical dependencies, as highlighted in the following example.

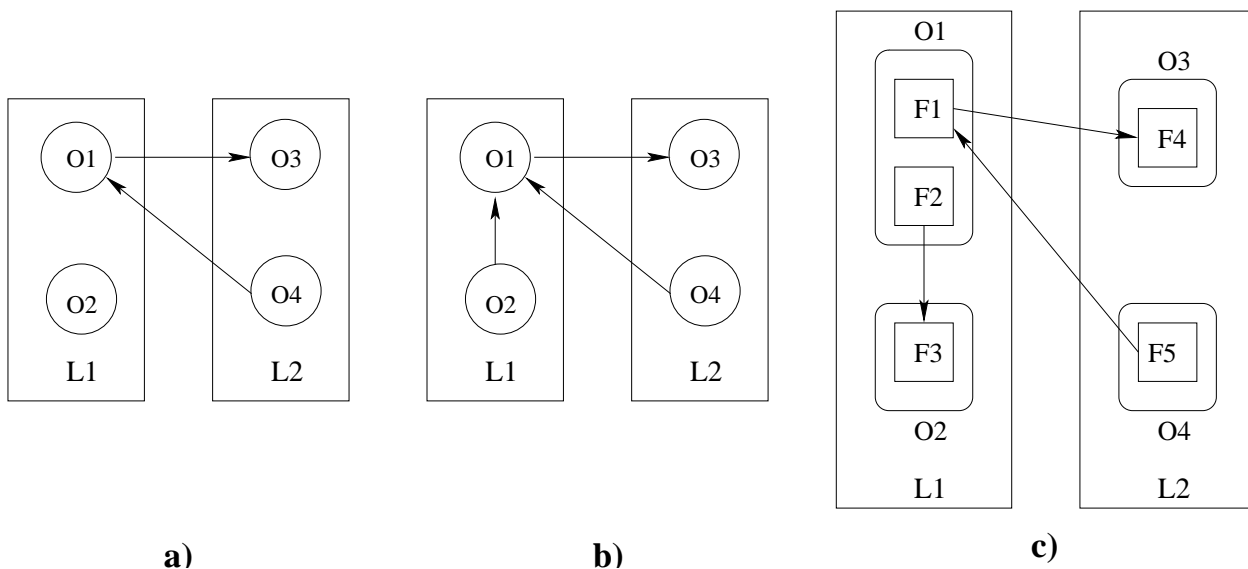


Figure 1. Examples of dependencies among libraries.

Let us consider the example in Figure 1-c: object o_1 cannot be moved, nor duplicated in L_2 (in that it depends from the object o_2). However, splitting o_1 into two chunks, leaving the function f_2 in library L_1 , and moving f_1 in L_2 , would solve the problem.

2.4 Identification of Duplicated Symbols

Comparing the list of symbols defined in each library allows detecting the list of duplicate external symbols. It is worth noting that homonym symbols in different libraries may refer to completely different functions, external variable or data structures.

Therefore, a cloning analysis is necessary. In this paper a metric-based clone detection process [6], aiming to detect duplicated functions, was adopted. The results obtained may suggest different possible actions:

1. If a whole, duplicated, object module has been detected inside two or more libraries, then it should be left in only one of these;
2. If duplicated functions are identified inside different objects, re-factoring could be performed moving them outside, applying the considerations similar to the previous case.

However, preliminary to the above actions, is an analysis of the impact in terms of dependencies (and, above all, circular dependencies) introduced. As explained in Section 2.3, sometimes an object is duplicated to avoid circular dependencies. In general, it may be preferable to duplicate few objects, rather than introducing a dependence that

causes, for a subset of the applications, the linking or the loading (if using DLLs) of one or more additional libraries.

2.5 Handling Unused Objects

Symbols defined in libraries, but not used by applications nor by other libraries, should be identified. Their presence is often due to utility functions inserted in libraries, but not used to the current set of applications, or to feature not yet fully implemented.

The objects defining these symbols should be removed from the library, providing that objects do not define also used symbols (it that case the object should be left into library, or properly split). One possible re-factoring solutions should be to create, from each library, two new libraries, one of which containing all the unused symbols.

2.6 Library Re-Factoring

The last, and most relevant point of the proposed process is devoted to split existing, large libraries into smaller archives, thus reducing the memory footprint of applications.

Basically, the idea is similar to those proposed in [5] to identify libraries: objects used by a common set of programs should be pulled together, trying to minimize the average number of libraries required by each program.

However, in [5] a *concept lattice* was used to group objects into libraries: although the lattice gives useful information (often good libraries are the sets of objects located

on top nodes, or concepts retaining large percentages of objects), it becomes unmanageable when a large number of applications and libraries (as in our case study) must be handled [3].

Instead of pruning information on *concept lattice* like [23, 26], a clustering analysis was performed, similarly to [17, 4, 16]. Let O_j the set of objects of the j library (library archiving p_j objects), A the set of applications (containing m distinguishable applications), and, finally, let L the set of libraries (containing n elements) then a Boolean matrix MD , composed by $n + m$ rows and p_j columns, was built, such that:

$$MD[x, y] = \begin{cases} true & \begin{cases} x \leq m & \text{object } o_y \text{ is used} \\ & \text{by application } a_x \\ x > m & \text{object } o_y \text{ is used} \\ & \text{by library } l_{x-m} \end{cases} \\ false & \text{otherwise} \end{cases}$$

Given that the application contained n libraries, n MD matrices were built; on each matrix an agglomerative-nesting clustering was performed. The idea is to found group of libraries having similar properties (i.e., a similar set of *uses*). More detail on clustering can be found in [2, 10, 11].

Once the *dendrogram* was built and the vector of heights of libraries inside the *dendrogram* itself computed, partitions were identified. The partitioning process was performed inspecting the *dendrogram* (differently from lattice analysis, this is still feasible for hundreds of libraries). It is worth noting that all the unused objects are clustered together (in that they are all characterized by all-false rows), satisfying requirements from Section 2.5.

Then, new candidate libraries should be refined, identifying and removing circular dependencies, as highlighted in Section 2.3.

However, there are cases in which neither the most sophisticated circular dependencies analysis, neither splitting objects suffice. In this case, the most reasonable solution is to split the original library into many small dynamic libraries. Even if these new libraries are circularly dependent each one from another, when an application requires a symbol, the number of libraries needed to cover the transitive closure of the dependencies is, on most cases, smaller than the total. This ensures, even in this case, to reduce the amount of resources needed.

Finally, a measure of the performances of the refactoring process was introduced to assess the process effectiveness. Let c the number of clusters l_{x_1}, \dots, l_{x_c} , obtained from a library l_x . Then, a *Partitioning Ratio* PR_x can be defined as:

Pre-existing libraries	43
Library objects	1056
Applications	517
C source files	7107
C KLOC	1014

Table 1. GRASS key characteristics.

$$PR_x = 100 * \sum_{i=1}^m \frac{\sum_{k=1}^c |l_{x_k}| * d_{i,x_k}}{|l_x| * d_{i,x}} \quad (3)$$

Where:

- $d_{i,x}$ is equal to one if the application i uses the library x (and zero otherwise);
- $|l_x|$ is the number of objects archived into library l_x .

The smaller is the PR , the most effective is the partitioning, in that the average number of objects linked (or loaded) from each application is smaller than using the old whole library.

3. Case Study

The modular structure of *GRASS* allows it to run with a very small memory overhead, therefore the hardware requirements are quite moderate. When running *GRASS* on a PC workstation or a notebook, standard equipment is generally sufficient. The *GRASS* CVS development snapshot of April 5, 2002, downloadable from <http://grass.itc.it> was used as a case study. Its characteristics are summarized in Table 1.

Supported platforms at the date of writing comprise Linux/PC, SUN, HP/UX, MacOSX, MS-Windows/Cygwin, iPAQ/Linux and others. For geospatial data, more RAM is generally more effective than a faster CPU. *GRASS* modules (commands) are organized by name, based on their function class (display, general, imagery, raster, vector or site, etc.). The first letter refers to the function class, followed by a dot and one or two other words, again separated by dots, describing the specific task performed by the module.

GRASS modules are invoked within a shell environment (also the current graphical user interface runs commands within a shell). The *GRASS* parser is a collection of sub-routines which allow the programmer to define options (parameters) and flags that make up the valid command line input of a *GRASS* command. The parser routines behave in the following ways: if no command line arguments are entered by the user, the parser searches for a completely interactive version of the command which may differ from the

command line version. If the interactive version is found, control is passed over to this version. If not, the parser will prompt the user for all programmer-defined options and flags. If all necessary options and flags are entered on the command line by the user, the parser executes the command with the given options and flags, otherwise the parser will pass an error message to the user indicating which required options and/or flags were missing from the command line and cancel execution of the command.

The *GRASS* modules are linked against an internal “front.end”. The “front.end” module will call the interactive version of the command if there are no command-line arguments entered by the user. Otherwise, it will run the command-line version. If only one version of the specific command exists (for example, if there is only a command-line version available) the existing command is executed. Code parameters and flags are defined within each module. They are used to ask user to define map names and other options.

GRASS provides an ANSI C language API with several hundreds of GIS functions which are utilized in the *GRASS* modules, from reading and writing maps to area and distance calculations for georeferenced data as well as attribute handling and map visualization. Details of *GRASS* programming are covered in the “*GRASS 5.0 Programmer’s Manual*” [20].

This programming library is structured as follows (typical function name prefixes for related library functions are listed in squared brackets):

- GIS library: database routines (*GRASS* file management), memory management, parser (parameter identification on command line), projections, raster data management etc. [G_],
e.g. `G_read_raster_row()`;
- vector library: management of area, line, and point vector data [Vect_, V2_, dig_],
e.g. `V2_read_line()`;
- image data library: image processing file management [I_],
e.g. `I_georef()`;
- site data library: site data management [G_sites_],
e.g. `G_site_new_struct()`;
- display library: graphical output to the monitor [D_],
e.g. `D_new_window()`;
- raster graphics library: display raster graphics on devices [R_],
e.g. `R_open_driver()`; item segment library: segmented data management [segment_],
e.g. `segment_get()`;

- vask library: control of cursor keys etc. [V_],
e.g. `V_ques()`;
- rowio library: for parallel row analysis of raster data [rowio_],
e.g. `rowio_get()`.

4. Tool Support

Several tools, most of which consisting in *Perl* scripts, were developed to re-engineer *GRASS* libraries:

- *The application identifier* that, using the `nm` Unix tool, identifies the list of object modules containing the `main` symbol;
- *The dependency graph extractor*, also based on the `nm tool`, produces the *DG* and the *IDG* graphs. The information produced is also available in `.DOT` format [7], in order to be visualized and analyzed using the `Dotty` [12] graph visualization tool;
- *The circular dependency identifier*: it produces the list of all circular paths among libraries;
- *The unused symbol lister*: it produces, for each library, the list of the symbols (and, for each one, the object containing it) not used by any application or library;
- *The duplicated symbol identifier*: it identifies the list of duplicated-defined external symbols. It is used in conjunction with the metric-based *clone detector* (see [6] for details) and the *dependency graph extractor* to minimize the presence of clones inside libraries; and
- *The library re-factoring tool*: it supports the process of splitting libraries in smaller clusters. As said in Section 2.6, this is performed by clustering algorithms. The cluster analysis is performed by the *cluster* package of the *R Statistical Environment* [1, 9].

5. Experimental Results

This section presents the results obtained applying the process described in Section 2 to *GRASS*.

5.1 Removal of Circular Dependencies among Libraries

Three cases of circular dependencies among libraries were found. The first dependency was between `libstubs.a` and `libdbmi.a`. In particular, we discovered that `libstubs.a` required one symbol, located inside the `error.o` module, from `libdbmi.a`. On the contrary, `libdbmi.a` required 27 symbols from `libstubs.a`.

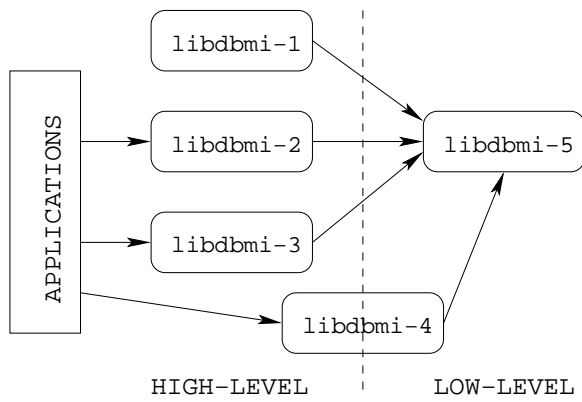


Figure 2. New libdbmi layering structure.

The obvious solution was to move `error.o` into `libstubs.a`: this required moving in that library also the module `alloc.o`, since it depends on `error.o`.

The second circular dependency was found between `libgis.a` and `libcoorcnv.a`. In particular, `libgis.a` required three symbols from `libcoorcnv.a`, all located in the module `datum.o` (while the inverse dependency involved 13 symbols). Moving `datum.o` into `libgis.a` resolved the problem.

Finally, circular dependencies were found between `libvect.a` and `libdig2.a`. It involved 13 symbols in a verse, 31 in another, located in several different objects. The links present in the dependency graph excluded the possibility of resolving circular dependencies simply moving (or duplicating) objects. The decision taken (supported from system’s developers) was to initially merge the two libraries (in effect thought to work together) and then try to re-factor the new library.

It is worth noting that, after each step, new libraries were built, and the *Circular Dependency Identifier* was run again to verify that the problem was actually resolved.

5.2 Identification of Duplicated Symbols

A total of 41 duplicated symbols were found inside GRASS libraries. Three symbols referred to external variables and data structure present in different libraries, while 24 of the remaining 38 effectively corresponded to cloned functions.

A deeper analysis revealed that 16 of the cloned symbols were present in the library `libortho`, and their clones spread across `libimage_sup`, `libgmth` and `libtrans`. Nine of the cloned functions were devoted to perform matrix algebra and, analyzing the *IDG* of matrix `libortho` (see Figure 3), a subgraph composed by such functions was identified (i.e., the box on the right).

On the other hand, seven of the functions in the box on the left were cloned in `libimage_sup`. In particular, the entire structure enclosed in the rounded-dashed-box was replicated in that library. Finally, the decision taken was to split `libortho` in two libraries, corresponding to the two boxes in Figure 3:

1. A library (`libmatrix`) to handle matrices; and
2. A library (`libcamera`) to handle photogrammetric computations for aerial cameras.

Cloned functions contained in these two libraries were removed from `libimage_sup`, `libgmth` and `libtrans`.

The remaining eight clones were found between the `driver` and `app` libraries of the *paint* subsystem. In particular, three were *false positive* (i.e., functions revealed to be different at manual-check), while the others were so small to do not justify re-factoring.

5.3 Handling Unused Objects

Of all 921 objects composing libraries, 89 (spread across almost all libraries) were not used by any applications, nor by other libraries. This result therefore suggests that, when re-factoring libraries, such objects, if present in a consistent number, should be put in separate clusters, thought as a sort of “repositories” where to get these functionalities once applications (or other libraries) will use them.

An interesting example (see also Section 5.4) is the library `libdbmi`: of 98 objects, 19 were not used at all, while 45 were used for internal purposes only.

5.4 Library Re-Factoring

Re-factoring was performed on libraries composed by a large number of objects (see Table 2).

The first analysis was performed on `libdbmi`, already modified (see Section 5.1) to avoid circular dependencies, since it contains a large number (19) of unused objects. A cluster analysis suggested creating five new libraries starting from `libdbmi`:

- A library (`libdbmi-1`) containing unused objects (19 objects);

Library	Objects
<code>libgis</code>	184
<code>libdbmi</code>	97
<code>libproj</code>	119
<code>libvect-new</code>	54

Table 2. GRASS largest libraries.

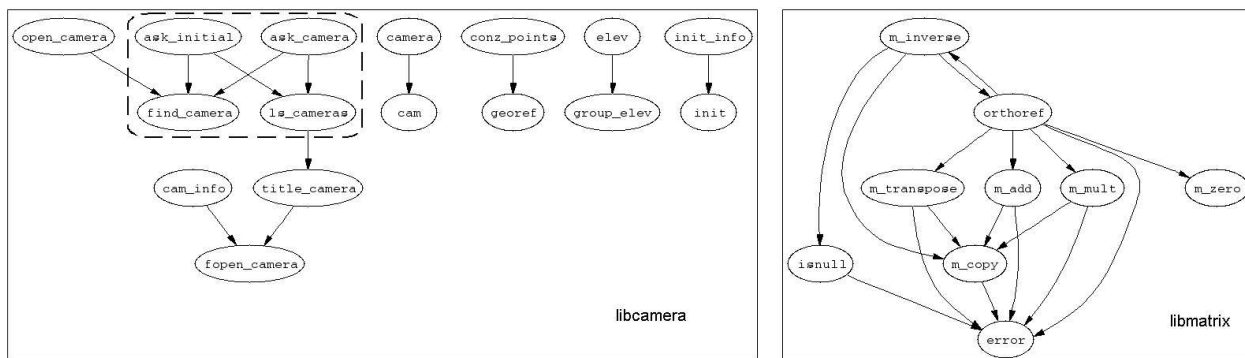


Figure 3. Splitting library libortho.

- Two libraries (libdbmi-2 and libdbmi-3) containing objects used by applications, but not by other libdbmi objects (13 and 12 objects, respectively);
- A library (libdbmi-4) containing objects used both internally to libdbmi and by applications (six objects); and
- A “low-level” library (libdbmi-5), containing 45 objects, used only internally to libdbmi.

Also developers suggested searching for a solution distinguishing high-level and low-level features. The solution obtained, except that for the case of libdbmi-5, resembles a layering structure, as depicted in Figure 2. The *PR* obtained was 16%, indicating that the objective of minimizing the average number of object loaded by each application was obtained. It is worth noting that, after clustering analysis, only an object was moved from libdbmi-5 to libdbmi-2 to avoid circular dependencies. A good compromise between having huge libraries and excessive fragmentation could be pursued merging libdbmi-2, libdbmi-3 and libdbmi-4 (high-level functionalities), while keeping separated libdbmi-1 (unused objects) and libdbmi-5 (low-level functionalities), obtaining a *PR* of 31%.

The new libvect, obtained merging libvect and dibdig (see Section 5.1), was re-factored in three clusters, composed by 20, 26, and 8 objects respectively. Few adjustments were needed to avoid circular dependencies between libvect-new-1 and libvect-new-2, while libvect-new-3 has no dependency links. The *PR* obtained was 46%, acceptable figure considering that libvect was used by a large number (98) of applications.

The biggest library, libgis, was split in six clusters, almost equally sized. In this case removing all dependencies was not possible, due to the complex relations existing among objects. However, the *PR* was good (41%), considering that libgis was used by all applications. Compro-

mises could be pursued splitting the original library in three or four cluster, and obtaining a *PR* of 60% and 54% respectively. Further investigation, devoted to perform analysis at a function grain-level, shall be performed to minimize dependencies.

libproj was not re-factored, as suggested from developers, in that it is under development by a different team and used in GRASS unmodified.

6. Lesson Learned

Large software system are not uncommon, *GRASS* size measured as LOC is something above one million LOC; re-factoring such a large system with a finite and scarce amount of resources was the first challenge encountered. The process defined attempted to minimize the time-to-market, given the number of programmers working on the application and minimizing the maintenance effort.

It is worth noting that *GRASS* source code is managed via a CVS server (ssh encrypted transmission) with write access granted only to well know developers. Submissions are controlled by a commit mailing list, communication is ensured by developer’s mailing list, and a RT (Request Tracker) – a trouble ticket system for bug and wish handling. The entire process was performed with the system on-line.

Given the size of the system, a preliminary subdivision based on the existing knowledge of the application revealed itself fundamental. When splitting a library l_x , a compromise should be pursued between the number of newly introduced libraries and a low PR_x . PR_x represents the reduction in percentage of linked objects when splitting library x , thus lower PR_x values are desirable. Notice that, for any given original library, its not ensured that increasing the number of new libraries results in a PR_x decrease. Reaching the right tradeoff is essential: having too many libraries makes the system hard to evolve (application developers may need to include several libraries to use a func-

tions, due to library dependencies), while having few big libraries does not target our initial issues of miniaturization.

Applying agglomerative clustering on components allowed to easily identify potential libraries; the list of identified libraries and the files linked into libraries were validated by GRASS developers. On most cases, the new libraries matched the opinion of the experts. Moreover, clone detection allowed us to obtain a double goal. First and foremost, cloned code was removed from source code linked into libraries. However, the process of clone removal identified other libraries transversal to the system (e.g., a matrix manipulation library).

A weakness of the proposed approach is that, refactoring libraries at object grain-level may increase dependencies among libraries. To obtain better results, we need to perform analysis at a function grain-level: this will avoid the problem that related functions are spread in various files, which is not detected by the object grain-level.

As previously observed on other large software systems, we experienced that the percentage of detected duplicated code snippets is influenced by the size. Short functions tend to be often identified as duplicated, thus on libraries we did not impose any lower bound when detecting clones.

7. Related Work

Many works are reported in literature concerning with software system modules clustering and/or restructuring, identifying objects, and recovering or building libraries. Most of these works applied clustering or concept analysis (CA).

An overview of CA applied to software reengineering problems was shown by G. Snelting in his seminal work [24], where he used CA in several re-modularization problems such as exploring configuration spaces (see also [13]), transforming class hierarchies, and re-modularizing COBOL systems. In [14] Kuipers and Moonen combined CA and type inference in a semi-automatic approach to find objects in Cobol legacy code. As in [13, 14, 15] we believe that with the present level of technology a programmer-centric approach is required: the user is left in charge to choose the proper re-modularization based on his/her knowledge.

A comparison between clustering and CA was presented in [15]. We share with them the idea to apply an agglomerative-nesting clustering to a Boolean usage matrix, although in [15] the matrix indicated the uses of variables by programs.

A survey of clustering techniques applied on software engineering was presented by Tzerpos and Holt in [29]. The same authors presented in [28] a metric to evaluate the similarity of different decompositions of software systems, in [30] a clustering algorithm oriented to program compre-

hension, and they discussed in [27] the problem of stability of software clustering algorithms. Another overview of cluster analysis applied on software systems has been presented in [31].

Applications of clustering to re-engineering can be found with [4] and [17]. In [4] a method for decomposing complex software systems into independent subsystems was proposed by Anquetil and Lethbridge. Source files were clustered based on file names and their decomposition. Merlo et al. [17] exploited comments, as well as variable and function names to clustered files.

Our work shares with [16] the idea of analyzing intra-module and inter-module dependency graphs, finding a tradeoff between having highly cohesive libraries and a low inter-connectivity.

In [5] authors proposed the idea of recovering libraries and creating a source file directory structure using CA. This paper shares with [5] the idea of finding libraries searching for sets of objects using by common groups of applications.

Initial work for GRASS miniaturization has been started for the WILMA wireless project [25]. A GRASS version reduced in functionality was extracted from the original source code and compiled on a portable PC (i.e., a Compaq iPAQ under GNU/Linux). The handhelds support may significantly mature when library re-factoring will be adopted by the GRASS Development Team.

8. Conclusions

The miniaturization process, the lessons learned and the results obtained re-factoring a large Open Source application were presented.

Given the size of the application and the available resources, a brute force approach was not feasible, thus we exploited the available knowledge breaking the large system into components; each component underwent the re-factoring process while inter-component dependencies were minimized. Agglomerative clustering and dendrograms were used to identify libraries based on the as-is dependencies and architecture.

While our final goal is miniaturize as much as possible the application and to fully exploit the possibilities offered by DLLs, the applied re-factoring process attempted to minimize both the maintenance effort while minimizing the time to market. In other words certain activities, such as source code transformation required to dynamically load and unload DLLs, or full clone removal, were not yet applied.

Future works will be devoted to automatically transform the source code of executables and libraries, removing remaining code duplication and introducing the calls required to load and unload DLLs at run-time. A fine-grained refactoring, at a function level, will also be performed.

9. Acknowledgments

We are grateful to the *GRASS* development team for the support, the information provided, and the feedback on the refactored artifacts.

Giuliano Antoniol and Massimiliano Di Penta were partially supported by the ASI grant I/R/ 091/00. Markus Neteler was partially supported by the FUR-PAT Project WEBFAQ.

References

- [1] The R project for statistical computing.
<http://www.r-project.org>.
- [2] M. R. Anderberg. *Cluster Analysis for Applications*. Academic Press Inc., 1973.
- [3] N. Anquetil. A comparison of graphs of concept for reverse engineering. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 231–240, June 2000.
- [4] N. Anquetil and T. Lethbridge. Extracting concepts from file names; a new file clustering criterion. In *Proceedings of the International Conference on Software Engineering*, pages 84–93, April 1998.
- [5] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. A method to re-organize legacy systems via concept analysis. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 281–290, Toronto, ON, Canada, May 2001. IEEE Press.
- [6] G. Antoniol, G. Casazza, M. Di Penta, and E. Merlo. Modeling clones evolution through time series. *Proceedings of IEEE International Conference on Software Maintenance*, pages 273–280, Nov 6-10 2001.
- [7] AT&T. Doty directed graph editor (part of graphviz).
<http://www.research.att.com/sw/tools/graphviz/>.
- [8] L. Garber. Will 3G really be the next big wireless technology. *IEEE Computer*, 35(1):26–32, January 2002.
- [9] R. Ihaka and R. Gentleman. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314, 1996.
- [10] A. Jain and R. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.
- [11] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley, 1990.
- [12] E. Koutsofios. Editing graphs with *doty*. Technical report, AT&T Bell Laboratories, Murray Hill, NJ, USA, July 1994.
- [13] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proc. of the 16th International Conference on Software Engineering*, pages 49–57, Sorrento Italy, May 1994.
- [14] T. Kuipers and L. Moonen. Types and concept analysis for legacy systems. In *Proceedings of the IEEE International Workshop on Program Comprehension*, pages 221–230, June 2000.
- [15] T. Kuipers and A. van Deursen. Identifying objects using cluster and concept analysis. In *Proceedings of the International Conference on Software Engineering*, pages 246–255, June 1999.
- [16] S. Mancoridis, B. S. Mitchell, C. Corres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *IEEE Proceedings of the 1998 Int. Workshop on Program Comprehension (IWPC'98)*, 1998.
- [17] E. Merlo, I. McAdam, and R. De Mori. Source code informal information analysis using connectionist model. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1339–44, Load Altos Calif, 1993.
- [18] A. Mockus, R. T. Fielding, and J. D. Herbsleb. A case study of open source software development: the apache server. In *International Conference on Software Engineering*, pages 263–272, 2000.
- [19] C. Narayanaswami, N. Kamijoh, M. Raghunath, T. Inoue, T. Cipolla, and Others. IBM's linux watch: The challenge of miniaturization. *IEEE Computer*, 35(1):33–41, January 2002.
- [20] M. Neteler, editor. *GRASS 5.0 Programmer's Manual. Geographic Resources Analysis Support System*. ITC-irst, Italy, <http://grass.itc.it/grassdevel.html>, 2001.
- [21] M. Neteler and H. Mitasova. *Open Source GIS: A GRASS GIS Approach*. Kluwer Academic Publishers, Boston/U.S.A.; Dordrecht/Holland; London/U.K., 2002.
- [22] E. Raymond. *The cathedral and the bazaar. Musings on Linux and Open Source by an accidental revolutionary*. O'Reilly & Associates, Cambridge, 1999.
- [23] M. Siff and T. Reps. Identifying modules via concept analysis. *IEEE Transactions on Software Engineering*, 25:749–768, Nov-Dec 1999.
- [24] G. Snelting. Software reengineering based on concept lattices. In *Proceedings of IEEE International Conference on Software Maintenance*, pages 3–10, March 2000.
- [25] J. Stankovic and M. Neteler. GRASS GIS on handhelds project.
<http://grass.itc.it/grasshandheld.html>.
- [26] P. Tonella. Concept analysis for module restructuring. *IEEE Transactions on Software Engineering*, 27(4):351–363, April 2001.
- [27] V. Tzerpos and R. Holt. the stability of software clustering algorithms, 2000.
- [28] V. Tzerpos and R. C. Holt. MoJo: A distance metric for software clusterings. pages 187–195.
- [29] V. Tzerpos and R. C. Holt. Software botryology: Automatic clustering of software systems. In *DEXA Workshop*, pages 811–818, 1998.
- [30] V. Tzerpos and R. C. Holt. ACDC: An algorithm for comprehension-driven clustering. In *Working Conference on Reverse Engineering*, pages 258–267, 2000.
- [31] T. A. Wiggerts. Using clustering algorithms in legacy systems remodularization. In *Proceedings of IEEE Working Conference on Reverse Engineering*, 1997.