Electronic Communications of the EASST
Volume 8 (2008)

Proceedings of the Third International ERCIM Symposium on Software Evolution (Software Evolution 2007)

Supporting Reengineering Scenarios with FETCH: an Experience Report

Bart Du Bois and Bart Van Rompaey, Karel Meijfroidt and Eric Suijs

14 pages

Guest Editors: Tom Mens, Ellen Van Paesschen, Kim Mens, Maja D'Hondt
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Supporting Reengineering Scenarios with FETCH: an Experience Report

Bart Du Bois[1] and Bart Van Rompaey[1], Karel Meijfroidt[2] and Eric Suijs[3]

[1] bart.dubois@ua.ac.be, bart.vanrompaey2@ua.ac.be
Lab On Reengineering, University of Antwerp, Belgium

[2] karel.meijfroidt@alcatel-lucent.be
Alcatel-Lucent, Antwerp, Belgium

[3] eric.suijs@philips.com
Philips Medical Systems, Best, The Netherlands

Abstract: The exploration and analysis of large software systems is a labor-intensive activity in need of tool support. In recent years, a number of tools have been developed that provide key functionality for standard reverse engineering scenarios, such as (i) metric analysis; (ii) anti-pattern detection; (iii) dependency analysis; and (iv) visualization. However, either these tools support merely a subset of this list of scenarios, they are not made available to the research community for comparison or extension, or they impose strict restrictions on the source code. Accordingly, we observe a need for an extensible and robust open source alternative, which we present in this paper. Our main contributions are (i) a clarification of useful reverse engineering scenarios; (ii) a comparison among existing solutions; and (iii) an experience report on four recent cases illustrating the usefulness of tool support for these scenarios in an industrial setting.

Keywords: Static Analysis, Quality Assurance, Industrial Experience

1 Introduction

To support maintenance scenarios for large software systems, where manual effort is error prone or does not scale, developers are in need of tools. In the past, several engineering techniques have been examined for their contribution to unraveling the evolution knot. In the identification of maintenance hot spots, *metrics* provide quantitative indicators. *Visualization* serves a key role in understanding the system composition, and to facilitate design/architecture communication. *Patterns* have been successfully introduced as a means to document both best practices (e.g., recurring programmatic needs [GHJV94]) or worst practices (e.g., anti-patterns documenting historical design decisions evaluated as suboptimal [BMMM98]). In assistance to impact and effort analysis, *dependency analysis* teaches the developer about component interactions. Together, these techniques form the key means to reverse engineer a given system from source.

All of these techniques require, at least implicitly, the presence of a model of the software system. Such a model abstracts from programming language and variant specific properties,

enabling the user to reason in terms of higher-level building blocks. In research on such models and the techniques using them, tools have been implemented.

In this work, we combined existing components and best practices of earlier tools into FETCH, a Fact Extractor[1] Tool Chain. Implemented as a chain of tools – following a pipes and filters architecture – we support key reverse engineering scenarios in each of the analysis domains (i) *metric calculation*; (ii) *(anti) pattern detection*; (iii) *impact analysis*; and (iv) *design/architecture visualization*.

FETCH aims to be flexible, lightweight, robust as well as open. Evidently, as a research tool, it is vital to facilitate additional analysis scenarios. Moreover, variations in software systems, e.g., with regard to their implementation environment (different languages or language variants) require that the architecture of the tool encapsulates processing steps in such a manner that the accommodation of new variations in input can be limited to the introduction of a new component or a replacement of an existing one. This is why FETCH is organized in a pipes and filter architecture. As tools such as FETCH grow larger, so does the mental load imposed by its usage, and similarly, its performance decreases with the number of processing steps. Accordingly, a lightweight approach is implemented, both with regard to the analysis techniques (e.g., lexical analysis), as well as the use of performance optimized heuristics. Related to the flexibility characteristic, robustness specifically addresses the quality of the output in case of a suboptimal input quality. Robustness is relevant, since the software systems used as input most often do not even compile (out of the box). For that purpose, FETCH's error handling is built to continue processing on a best effort basis. Lastly, the creation of a user and development community surrounding FETCH ensures its future success. On that account, we made FETCH open source, encouraging industrial and academic parters to adopt, extend and integrate with it.

To illustrate the need for, and usefulness of, a tool chain satisfying the above properties, we report on the application of FETCH on recent industrial collaborations.

This paper is structured as follows. The set of existing (C++) fact extractors is discussed in Section 2. As a recent alternative to these existing solutions, FETCH is presented in Section 3. In Section 4, we report on the application of FETCH in recent industrial collaborations. Finally, in Section 5, we conclude and discuss future work.

2 Related Work

Table 1 compares the usage scenarios provided by related C++ fact extractors. This non-exhaustive list has been composed with the information available to us at the time of writing. As C++ fact extractors, these tools invariably process C++ source code and generate an abstract, query-able representation.

We use the following abbreviations: (i) *MC* for Metric Calculation; (ii) *PD* for Pattern Detection; (iii) *DA* for Dependency Analysis; and (iv) *V* for Visualization. As not all fact extractors were built to support these features, we also indicate other key strengths of the tools listed. Additionally, we indicate whether the tools are open source – i.e., whether they can be freely used and extended. The latter is relevant for readers wishing to use the extractor for their own projects.

[1] A *fact extractor* is an analysis tool that distills information elements – termed *facts* – from software artefacts.

Table 1: Comparison of a non-exhaustive list of C++ fact extractors

Fact Extractor	MC	PD	DA	V	Key strengths	Open source?
Open Visualization Toolkit	√			√	Aggregation	No
TkSee/SN	√		√		Source navigation	Yes
iPlasma	√	√			Quality assurance	No
CPPDM extractor	√		√	√	System decomposition	No
Rigi	√		√	√	Aggregation	Partially
Acacia	√	√	√	√	Dependency analysis	No
SPOOL	√	√	√	√	Design pattern recovery	No
CppX/SWAG Kit	√	√	√	√	Architecture evaluation	Yes
#	8	4	6	6		

Open Visualization Toolkit – uses Source Navigator [2] (SN) in combination with Open Inventor C++ toolkit [TMR02]. The usefulness of the toolkit for reverse architecting large systems has been demonstrated on several software systems from Nokia. A key contribution of this toolkit is the ability to aggregate lower level facts on a higher hierarchical level.
TkSee/SN – uses SN to generate a GXL representation of the Dagstuhl Middle Model (DMM). This tool mostly supports navigation and querying. While TkSee/SN is indicated in [SHE02] to be open source, we were unable to find any reference to the source code nor any binary release.
iPlasma – The Integrated Platform for Software Modelling and Analysis particularly targets quality analysis [MMM+05]. Instances of iPlasma's meta-model (Memoria [Rat04]) can be queried using the Static Analysis Interrogative Language (SAIL) [MMG05].
CPPDM extractor – [MW03] discusses a C++ fact extractor combining SN with Rigi [MK88]. The output model conforms to a C/C++ domain model (CPPDM), and is presented in Rigi Standard Format (RSF). Since the authors do not provide a name for their tool, we abbreviate it as *CPPDM extractor*.
Rigi – presents a scriptable environment for visualizing the structure of large software systems. Parsers are provided for multiple languages (C, C++ and COBOL) [MK88]. The Rigi C++ parser is based on IBM VisualAge for C++ [Mar99].
Acacia – C++ Information Abstraction System developed by AT&T [CNR90]. A tool named Ccia transforms C++ source code to a relational database, queried using Ciao [CFKW95].
SPOOL – The SPOOL environment transforms source code into the Datrix/TA-XML intermediate format using the Datrix C++ parser [SKL+02]. This XML format is then traversed using an XML parser. SPOOL was set out to support pattern-based reverse engineering for program comprehension, and was applied on industrial C++ systems.
CPPX/SWAG Kit – transforms the internal graph structure of `gcc` to a GXL (Graph eXchange Language) representation of Datrix [DMH01]. The dependency on (an out-dated) `gcc` (v3.0) enforces stringent requirements on the source code, which may not be easily satisfied in industrial research projects. CPPX is part of the SWAG Kit that furthermore entails the Grok [HWW02] fact manipulation engine as well as a graph visualization environment.

In [SHE02], a C++ fact extractor benchmark (CppETS) is presented. Four extractors (CCia, CppX, Rigi C++ parser and TkSee/SN) were evaluated using so called *test buckets*, present chal-

[2] http://sourcenav.sourceforge.net

lenging C++ fragments. This evaluation addressed the accuracy and robustness of the extractors. In contrast, we are specifically looking for high-level services provided on top of the raw fact extraction. The resulting comparison[3] is presented in Table 1.

3 Tool Chain

The C++ fact extractors discussed in the previous section successfully supported reverse engineering activities, as demonstrated by the experience reports and case studies in their respective publications. However, researchers and practitioners wishing to reuse these extractors on their projects typically face at least two problems. First, these extractors most often support merely a subset of the features mentioned Table 1. Accordingly, key reverse engineering scenarios – e.g., the ones presented in this work – are not supported. Second, these extractors are more than often unavailable: we were only able to find the source code for CppX/SWAG Kit and Rigi. The latter is a problem in case one wishes to apply the extractors, as we do.

To overcome these problems, we propose FETCH[4], a 100% open-source tool chain. FETCH bundles and unifies a set of open source tools supporting key issues in reverse engineering. Figure 1 illustrates how FETCH[5] combines these tools to statically analyze source code. Indeed, FETCH currently does not provide a solution for dynamic analysis.

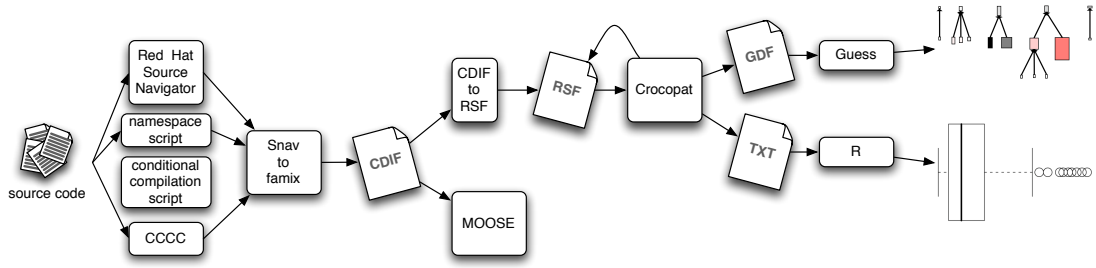


Figure 1: The chain of tools comprising FETCH

Source Navigator (SN) – A source code analysis tool initiated by Red Hat. [TD01] provides a nice overview of the supported functionality. SN has been used previously to develop metric tools [SE01, SKN05] and even some of the C++ fact extractors presented earlier. One of the main benefits of SN is that it parses the source code for structural programming constructs through robust, lexical analysis, making it tolerant towards variants of C and C++. The results of a parse are stored in database tables together with location and scoping information. Subsequently, relations between such structural constructs such as method invocations and attribute accesses are obtained via a cross-referencing step.

CCCC – A metrics tool developed by Littlefair [Lit01]. These metrics concern characteristics not deducible from the structure of the source code, as derived by SN, such as Lines of Code,

[3] Disclaimer: whether or not an extractor supports a scenario is extracted from publications/documentation.

[4] We emphasize the names of home-made tools in a different STYLE to distinguish them from existing ones.

[5] http://www.lore.ua.ac.be/Research/Artefacts/fetch

Cyclomatic Complexity and Comment Lines.

Crocopat – A graph query engine written by Beyer and Noack. Crocopat supports the querying of large software models in RSF using Prolog-alike Relation Manipulation Language (RML) scripts. In contrast to typical binary relation query languages as Grok, Crocopat's support for n-ary relations is particularly expressive for software patterns. Moreover, the highly efficient internal data structure enables superior performance to Prolog for calculating closures (e.g., a call-graph) and patterns with multiple roles (e.g., design patterns) [BNL05].

Guess – A graph exploration system developed by Adar [Ada06]. The input graphs described in ASCII format (GDF) can specify various properties of nodes and edges as required for polymetric views [LD03]. Various layout algorithms are provided. A tree hierarchy is missing, yet can be contributed.

R – The R project for statistical computing[6] initiated by Bell Laboratories. Typical static analyses such as calculating spread, identifying outliers, comparing data sets and evaluating predictive models are supported through the vast library.

The first phase of any usage scenario consists of generating an abstract model corresponding to the Famix (FAMOOS[7] Information Exchange Model) specification. The contents of this model (see Table 2) requires input from (i) Source Navigator; (ii) CCCC; and (iii) home-made script to extract namespace scopes and conditional compilation directives. The facts provided by these three tools are combined and interconnected by SNAVTOFAMIX[8], resulting in the generation of a CDIF (Case Data Interchange Format [Che03]) representation of the Famix model.

Table 2: Abstract model content.

Entity	Origin
File, Include	Source Navigator
ConditionalCompilation	SCRIPT_1
Package	SCRIPT_2
Class, Inheritance, Typedef	Source Navigator
Method, Function	Source Navigator
Attribute, Global Variable	Source Navigator
Invocation,Access	Source Navigator
	+ SNAVTOFAMIX
Measurement	CCCC
	+ Source Navigator

Each of FETCH's usage scenarios requires to query the abstract model. To facilitate querying, the abstract model is translated to a graph format (RSF, the Rigi Standard Format). Crocopat internalizes this RSF model in an efficient in-memory data structure, and executes given RML scripts (Relation Manipulation Language) as queries. These queries result in ASCII output, which we use to generate either (i) reports; (ii) a filtered/extended/altered RSF model; (iii) a visualization in GDF format (Guess Data Format); or (iii) tab-separated data listings.

For visualizations, such as polymetric views [LD03], we employ Guess. Guess provides a flexible means to explore visualizations by zooming and navigating, various layout algorithms, and a query/manipulation mechanism for properties of the visualized information. In case more advanced statistical analysis is required, tab-separated data listings are processed using R scripts.

[6] http://www.r-project.org
[7] http://www.iam.unibe.ch/∼famoos
[8] http://sourceforge.net/projects/snavtofamix

4 Experience Report

In this section, we present an experience report. This work is carried out in collaboration with two industrial partners in the ITEA SERIOUS[9] project. As of March 2007, we used FETCH to support several reengineering projects in multi-million line software systems. These reports present applications of the four usage scenarios presented in Table 1, and indicate the relevance of FETCH in industrial projects in which reverse engineering was called for.

4.1 Case 1: Internal Quality Monitoring

To describe the use case at the heart of this usage of FETCH, we reuse a template provided by [Coc95]. Table 3 presents a characteristic description, and enlists the main success scenario.

Table 3: Basic Use Case Template for Internal Quality Monitoring

Item	Value
Goal in Context	User requests comparison of two versions of the same software system with regard to metrics of choice.
Preconditions	A model (abstract representation) of the software system is available.
Success End Condition	User receives file comparing metrics between the two versions.
Failed End Condition	User receives message explaining failed attempts.
Primary Actor	Quality Assurance
Trigger	User launches comparison script
Main success scenario	0. User identifies metrics to be used for the comparison and selects associated script to compare metrics 1. User launches comparison script. 2. Comparison script generates plots and report comparing metric values between the two versions.

To support management in monitoring the improvement of quality characteristics during currently running refactoring efforts, the authors were asked to provide objective data. At large, the refactoring scenario can be summarized as a partial transition from an overly complex implementation in C to a more well-balanced design in C++, serving to (i) introduce abstraction; and improve (ii) understandability; and (iii) testability.

The module in question concerned a protocol implementation at the data link layer, comprising 8.6 kSLOC of C code, of which the lion's share (6.9 kSLOC) was contained in a single source file. Key characteristics of the structural decomposition before and after refactoring are enlisted in Table 4. After refactoring, the largest implementation file was reduced with 40.1% to 4.1 kSLOC by encapsulating functionality in newly introduced abstractions. This encapsulation is clearly noticeable in Table 4. What is not deducible from Table 4 is the extent to which this encapsulation represents a redistribution of functionality. Accordingly, Figure 2 compares the distribution of methods and functions across files before and after

Table 4: Structural comparison.

Entity	# before	# after	Δ
File	9	38	+29
Class	24	46	+22
Method	0	270	+270
Function	255	214	-41
Attribute	191	282	+91
Global Variable	66	60	-6

[9] https://lore.cmi.ua.ac.be/serious

refactoring. The *before* image on the left indicates that all methods and functions were contained in merely two files (a highly centralized implementation). *After* refactoring, the functionality is decentralized through a distribution across multiple files, each containing less methods and functions.

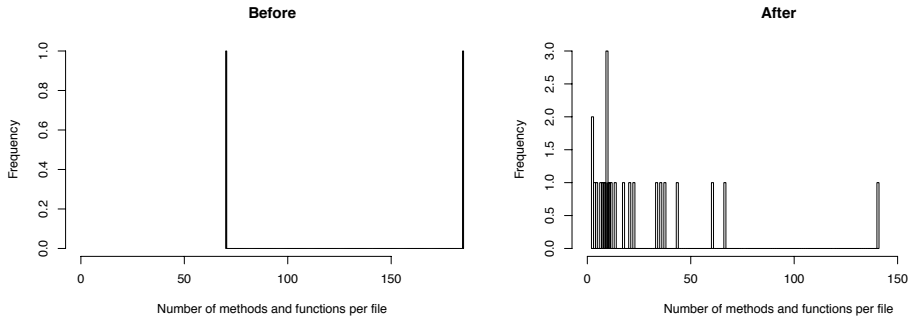
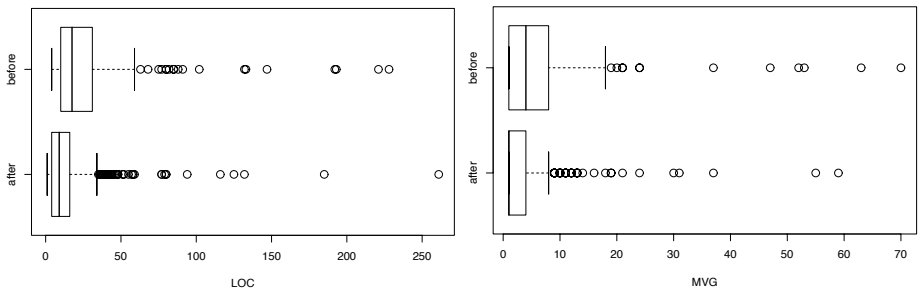


Figure 2: Distribution of methods and functions before/after refactoring

Table 4 indicates that as much as 270 methods were introduced, increasing the total number of invokable entities (methods + functions) from 255 before refactoring, to 484 after refactoring. Accordingly, we ask ourselves whether this refinement improved readability and testability.

The average size and complexity of methods and functions was considerably reduced, as can be seen from Figure 3. Size was shortened from an average of 27 LOC (median=17) to 14 LOC (median=8). Complexity (measured using an approximation of McCabe's cyclomatic complexity) was reduced from an average of 6.3 (median=1) to 3.3 (median=4). Cyclomatic complexity calculates the number of independent paths through a method or function, and is thus directly related to testing effort, since each independent path will need to be covered by a test case to ensure adequate coverage. In the reduction of the spread, we observe that the variation in size and complexity is harmonized as well.



(a) Size (b) Complexity

Figure 3: Variation of method/function size/complexity before and after refactoring

Summarizing, our observations reveal that the refactoring efforts led to a redistribution of functionality. This distribution was accompanied with a reduction of the size and complexity of methods and functions, resulting in an implementation that is both easier to understand and test.

The tools within FETCH providing the required facts for this scenario are Source Navigator (providing information on the structural composition) and CCCC (providing size and complexity measurements). The integration of these two data sources is implemented in SNAVTOFAMIX. This scenario also demonstrates that relatively primitive information can suffice to verify the achievement of quality improvements during software evolution.

4.2 Case 2: Estimating Refactoring and Test Effort

In a second case FETCH was used to estimate the effort involved in a refactoring task inside a 67 kSLOC subsystem, implementing a protocol at the network layer. The associated use case is characterized in Table 5.

Table 5: Basic Use Case Template for Estimating Refactoring and Test Effort

Item	Value
Goal in Context	User requests report about impact and effort of a refactoring task, testing inclusive.
Preconditions	A model (abstract representation) of the software system is available.
Success End Condition	User receives a report with impacted locations and the corresponding effort in lines of code.
Failed End Condition	User receives message explaining failed attempts (e.g. a faulty description of the modification, or a refactoring of non-existant entities).
Primary Actor	Developer
Trigger	User launches estimation script with a description of the modifications in terms of model entities.
Main success scenario	1. User identifies script to perform impact and effort analysis. 2. User translates refactoring task in terms of operations on model entities. 3. User launches script.

Developers noted that the data members of a particular C struct, called `T_Reference`, were directly passed on, accessed and modified throughout the subsystem. One of the features for the next release was translated, at code level, in the introduction of additional data members requiring to change along with the existing members. Developers proposed a proper interface to avoid change ripples in expected future.

In a first step, we determined the change impact of the refactoring. Typically, developers used text search tools to obtain an overview of the usage of such a struct via locations where the type is declared (as formal parameter, local variable, etc.), e.g., `grep` [LH02]. We used FETCH on a model of this subsystem to obtain a list of functions that access data members of *T_Reference* instances. A traditional search does not suffice to identifying these accesses as cross-references between instances of structural entities are required. Part of the (disguised) report is represented in Listing 1. The observation that three of the data members are accessed together in most of the functions confirms the refactoring need. Replacing these direct accesses with an interface invocation is a one line modification. Therefore, we estimated 55 (the number of accesses detected in this case) SLOC of effort.

Listing 1: Change Impact Report for T_Reference generated by FETCH

```
Looking up use of T_Reference
For T_Reference in file types_ifc.hh
  Function to data access:
```

```
  stack.c:frameRecv(buffer *,u_int32 *) -> T_Reference.line
  stack.c:frameRecv(buffer *,u_int32 *) -> T_Reference.vci
  stack.c:frameRecv(buffer *,u_int32 *) -> T_Reference.vpi
  ...

Method to data access:
  configControl.hpp:ConfigControl.changeOperate(T_Bool) -> T_Reference.line
  configControl.hpp:ConfigControl.changeOperate(T_Bool) -> T_Reference.vci
  configControl.hpp:ConfigControl.changeOperate(T_Bool) -> T_Reference.vpi
  ...
```

Secondly, we used this report to estimate testing effort that had to be started from scratch. The testing approach corresponding to this refactoring task was chosen to be on a per function basis. Within testing, we dinstinguish the efforts of (i) stubbing external subsystems; (ii) setup operations to bring the unit under test in the desired state; and (iii) the actual test code. To estimate the number of lines of test code, we incorporate, per function, the number of input parameters (NOP), McCabe's cyclomatic complexity (MVG) [McC76] as well as FanOut [CK91].

Table 6 shows the effort estimation for the test code based upon the incorporated metrics. We reason that testing one path through a function requires about 15 SLOC, but as we only want to test paths (also see [WM96] on testing using cyclomatic complexity) affected by the refactoring we roughly estimate that half of the paths require testing, resulting in the $0.5 * 15 * MVG$ formula for the actual test code. Summed up, our test estimation sums up to 1730 kSLOC of stubbing, 257 kSLOC of setup and 1582.5 kSLOC of testing effort.

Table 6: Test Effort Estimation Scheme

Task	NOP	MVG	FanOut
Stubbing	-	-	10
Setup	2	-	1
Testing	-	0.5*15	-

Using FETCH in an effort estimation context enabled to calculate the expected change impact objectively, in contrast with the typical combination of gut feeling and search tools. The availability of a software model (provided by Source Navigator), enriched with complexity and coupling metrics (provided by CCCC), and a powerful query mechanism (Crocopat) were essential to support this scenario.

4.3 Case 3: Detecting anti-patterns: Dead code

A use case description for this case is presented in Table 7.

Dead code has many faces. One may find (i) blocks of code that are commented; (ii) functions not called anymore; and thirdly (iii) code that is not compiled due to compiler directives. A particular instance of the latter category uses the `#if 0` directive, a condition that never results in an inclusion of the nested code after preprocessing. As one of the developers noted that the latter approach is common behaviour in a 110 kSLOC subsystem (implementing a network protocol) under study, we introduced the concept of a conditional compilation directive to the FAMIX meta-model [RVW07], in order to reason about conditional compilation and to remove dead code by compilation directive in particular.

We queried the model for all block conditions that contain condition "0" at least as subcondition, thereby identifying 911 SLOC out of the total 110 kSLOC that should be removed according

Table 7: Basic Use Case Template for Detecting Dead Code

Item	Value
Goal in Context	User requests a list of design or architecture elements involved in a given (anti-)pattern.
Preconditions	model (abstract representation) of the software system is available.
Success End Condition	User receives file listing occurrences for a given (anti-)pattern.
Failed End Condition	User receives message explaining failed attempts.
Primary Actor	Q&A
Trigger	Q&A engineer (or an automated process) launches a set of detection scripts for given (anti-)patterns.
Main success scenario	1 User identifies (anti-)pattern to be calculated. 2. User identifies associated script to calculate metrics. 3. User launches detection script. 4. Calculation script generates file reporting on (anti-)pattern occurrences

to developer's opinion that such a construct should not be used. Rather, the versioning system should be used to store code that becomes dead or redundant at a certain time.

FETCH helped to efficiently identify such locations, in what would otherwise have been a cumbersome task of manually identifying blocks that hold this compilation condition. Especially when such blocks are nested, determining the scope of these blocks becomes hard. Note that information on conditional compilations is not provided by Source Navigator (see Table 2).

4.4 Case 4: Detecting key domain abstractions using visualization

Table 8 characterizes the FETCH use case for detecting key domain abstractions. As part of our First Contact to a large C++ software system (a medical image application of around 400 kSLOC), we applied a heuristic for detecting key domain abstractions. This heuristic consists of finding concepts existing in many variations. Typically, sub-variations of a concept are modeled through inheritance, enabling white-box reuse. Templates, however, enable black-box reuse, thereby introducing another way of modeling conceptual variations. In this particular software system, template instantiations were made explicit using `typedef` declarations. Accordingly, we proposed to decorate the inheritance hierarchy with typedef relations to represent both types of variation relations.

Table 8: Basic Use Case Template for Detecting key domain abstractions

Item	Value
Goal in Context	User requests overview of the genericity of classes in variation graph
Preconditions	A model (abstract representation) of the software system is available
Success End Condition	User receives visualization of variation graph.
Failed End Condition	User receives message explaining failed attempts.
Primary Actor	(Newly introduced) maintainer or developer
Trigger	User launches visualization script.
Main success scenario	1. User launches visualization script. 2. Visualization script generates GDF file representing variation graph. 3. User loads GDF file into Guess.

The resulting *variation graph*, part of which is represented in Figure 4, contains black circles

and edges to represent classes and the inheritance relations between them, and yellow rectangles and edges to represent typedefs and the relation to the base types for which the typedef introduces an alias. Superclasses in the inheritance hierarchy on the left were confirmed by the maintainers to represent high level concepts that encapsulated generic functionality. The two classes in the hierarchy on the right were confirmed as representations of elements in a data model. Both sets of classes thus indeed represent key domain abstractions, and the latter indicated concepts with many variations not previously observed in the inheritance hierarchy.

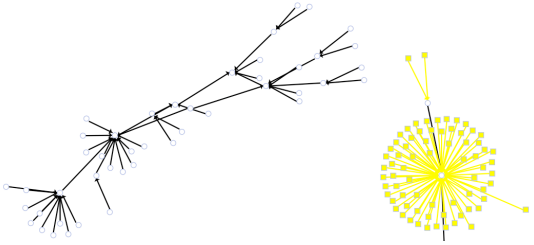


Figure 4: Variation graph decorating a traditional inheritance hierarchy with typedefs

The input required to compose this visualization is fully provided by Source Navigator. The visual recognition of relevant classes is facilitated through Guess's layout algorithms. For this scenario, FETCH's contribution lies in the integration of these two tools.

5 Conclusion and Future Work

In this work we present the following contributions. First, we enlist four features highly relevant for reverse engineering tools, namely (i) metric calculation; (ii) (anti-)pattern detection; (iii) impact analysis; and (iv) visualization. Second, we compare current C++ fact extractors with regard to these features, indicating the need for a freely available integrated solution. Thirdly, we introduce FETCH, a tool chain integrating existing open source tools supporting the above listed features. Finally, we report on four cases in which FETCH supported industrial reengineering projects.

Since no quantitative data was available to serve as an evaluation, this paper merely suggests the industrial usefulness of an integrated tool chain as FETCH, and by no means attempts to validate its efficacy. The validation of FETCH's contribution in each of the reverse engineering scenarios mentioned in this paper will be addressed in the next research steps.

Acknowledgements: This work has been sponsored by the Flemish *Instituut voor Innovatie door Wetenschap en Technologie* under grants of the ITEA project if04032 entitled Software Evolution, Refactoring, Improvement of Operational & usable Systems (SERIOUS) of the Eureka Σ 2023 Programme.

Bibliography

[Ada06] E. Adar. GUESS: a language and interface for graph exploration. In *CHI '06: Proceedings of the SIGCHI conference on Human Factors in computing systems*.

Pp. 791–800. ACM Press, New York, NY, USA, 2006.

[BMMM98] W. Brown, R. Malveau, H. McCormick, T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, 1998.

[BNL05] D. Beyer, A. Noack, C. Lewerentz. Efficient Relational Calculation for Software Analysis. *IEEE Trans. Softw. Eng.* 31(2):137–149, 2005.

[CFKW95] Y.-F. R. Chen, G. S. Fowler, E. Koutsofios, R. S. Wallach. Ciao: a graphical navigator for software and document repositories. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*. P. 66. IEEE Computer Society, Washington, DC, USA, 1995.

[CGK98] Y.-F. Chen, E. R. Gansner, E. Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. *IEEE Trans. Softw. Eng.* 24(9):682–694, 1998.

[Che03] M. Chen. CASE data interchange format (CDIF standards: introduction and evaluation. In *HICS'03: Proceedings of the 26th Hawaii International Conference on System Sciences*. Pp. 31–40. IEEE Computer Society, Washington, DC, USA, 2003.

[CK91] S. R. Chidamber, C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*. Pp. 197–211. 1991.

[CNR90] Y.-F. Chen, M. Y. Nishimoto, C. V. Ramamoorthy. The C Information Abstraction System. *IEEE Trans. Softw. Eng.* 16(3):325–334, 1990.

[Coc95] A. Cockburn. Basic use case template. Technical report HaT.TR.95.1, Humans and Technology, 84121, Salt Lake City, Utah, 1995.

[DDN02] S. Demeyer, S. Ducasse, O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DMH01] T. R. Dean, A. J. Malton, R. Holt. Union Schemas as a Basis for a C++ Extractor. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*. P. 59. IEEE Computer Society, Washington, DC, USA, 2001.

[GHJV94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994.

[HWW02] R. C. Holt, A. Winter, J. Wu. Towards a Common Query Language for Reverse Engineering. Technical report –, Institute for Computer Science, Universität Koblenz-Landau, Koblenz, Germany, 2002.

[KSRP99] R. K. Keller, R. Schauer, S. Robitaille, P. Pagé. Pattern-based reverse-engineering of design components. In *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Pp. 226–235. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[LD03] M. Lanza, S. Ducasse. Polymetric Views – A Lightweight Visual Approach to Reverse Engineering. *IEEE Transactions on Software Engineering* 29(9):782–795, 2003.

[LH02] T. T. Lethbridge, F. Herrera. *Advances in software engineering: comprehension, evaluation and evolution*. Chapter Assessing the usefulness of the TKSee software exploration tool, pp. 73–93. Springer-Verlag New York, Inc., 2002.

[Lit01] T. Littlefair. *An Investigation into the Use of Software Code Metrics in the Industrial Software Development Environment*. PhD thesis, Faculty of communications, Health and Science, Edith Cowan University, June 2001.

[Mar99] J. Martin. Leveraging IBM visual age for C++ for reverse engineering tasks. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. P. 6. IBM Press, 1999.

[MB89] T. J. McCabe, C. W. Butler. Design complexity measurement and testing. *Commun. ACM* 32(12):1415–1425, 1989.

[McC76] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering* 2(4):308–320, December 1976.

[MK88] H. A. Müller, K. Klashinsky. Rigi-A system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*. Pp. 80–86. IEEE Computer Society Press, Los Alamitos, CA, USA, 1988.

[MMG05] C. Marinescu, R. Marinescu, T. Girba. Towards a Simplified Implementation of Object-Oriented Design Metrics. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*. P. 11. IEEE Computer Society, Washington, DC, USA, 2005.

[MMM+05] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, R. Wettel. iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*. Pp. 77–80. IEEE Computer Society, Washington, DC, USA, 2005.

[MW03] D. L. Moise, K. Wong. An Industrial Experience in Reverse Engineering. In *WCRE '03: Proceedings of the 10th Working Conference on Reverse Engineering*. P. 275. IEEE Computer Society, Washington, DC, USA, 2003.

[NDG05] O. Nierstrasz, S. Ducasse, T. Gîrba. The story of moose: an agile reengineering environment. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. Pp. 1–10. ACM Press, New York, NY, USA, 2005.

[Rat04] D. Ratiu. Memoria: A Unified Meta-Model for Java and C++. Master's thesis, Politehnica University of Timisoara, Romenia, 2004.

[Riv02] C. Riva. Architecture Reconstruction in Practice. In *WICSA 3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*. Pp. 159–173. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 2002.

[RVW07] M. Rieger, B. Van Rompaey, R. Wuyts. Teaching FAMIX about the Preprocessor. In *1st Workshop on FAMIX and Moose in Reengineering (FAMOOSr'07)*. Pp. 13–16. 2007.

[SE01] M. Stojanovic, K. E. Emam. ES2: A Tool for Collecting Object-oriented Design Metrics for the C++ and Java Source Code NRC/ERB-1088. Technical report, National Research Council Canada, 2001.

[SHE02] S. E. Sim, R. C. Holt, S. Easterbrook. On Using a Benchmark to Evaluate C++ Extractors. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*. P. 114. IEEE Computer Society, Washington, DC, USA, 2002.

[SKL+02] R. Schauer, R. K. Keller, B. Lagué, G. Robitaille, S. Robitaille, G. Saing-Denis. *Advances in software engineering: comprehension, evaluation and evolution*. Chapter The SPOOL design repository: architecture, schema, and mechanisms, pp. 269–294. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[SKN05] S. Sarkar, A. C. Kak, N. S. Nagaraja. Metrics for Analyzing Module Interactions in Large Software Systems. In *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference (APSEC'05)*. Pp. 264–271. IEEE Computer Society, Washington, DC, USA, 2005.

[TD01] S. Tilley, M. DeSouza. Spreading Knowledge about Gnutella: A Case Study in Understanding Net-Centric Applications. In *IWPC '01: Proceedings of the 9th International Workshop on Program Comprehension*. Pp. 189–198. IEEE Computer Society, Los Alamitos, CA, USA, 2001.

[TMR02] A. Telea, A. Maccari, C. Riva. An Open Visualization Toolkit for Reverse Architecting. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*. P. 3. IEEE Computer Society, Washington, DC, USA, 2002.

[TTDS07] P. Tonella, M. Torchiano, B. Du Bois, T. Systä. Empirical studies in reverse engineering: state of the art and future trends. To appear in Journal on Empirical Software Engineering, 2007.

[WM96] A. Watson, T. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. Technical report, National Institute of Standards and Technology, USA, 1996.