A Comparative Framework for Design Recovery Tools

Yann-Gaël Guéhéneuc
Department of Informatics and Operations Research
University of Montreal, Canada
guehene@iro.umontreal.ca

Kim Mens
Département d'Ingénierie Informatique
Université catholique de Louvain, Belgium
kim.mens@uclouvain.be

Roel Wuyts
Département d'Informatique
Université Libre de Bruxelles, Belgium
roel.wuyts@ulb.ac.be

Abstract

While many commercial and academic design recovery tools have been proposed over the years, assessing their relevance and comparing them is difficult due to the lack of a well-defined, comprehensive, and common framework. In this paper, we introduce such a common comparative framework. The framework builds upon our own experience and extends existing comparative frameworks. We illustrate the comparative framework on two specific design recovery tools.

1 Introduction

Ever since the first statements on the "software crisis" [14, 15], work has been conducted to support program maintenance and to develop program comprehension tools. These tools strive to help software engineers in understanding ever larger and more complex pieces of software. In particular, several examples of design recovery tools can be found in academia and industry. *Design recovery tools recreate design abstractions from a combination of source code, existing design documentation, personal experience, and general knowledge about problem and application domains [4].* Design recovery tools are helpful to software engineers when comprehending pieces of software, because they reduce the amount of information with which software engineers must deal, as they abstract raw source code to higher-level easier-to-understand constructs.

Existing design recovery tools have very different characteristics, depending on the software constructs they reason about, on the extraction and abstraction techniques they implement, on the output they produce, and on the users' knowledge level they assume, to name but a few concerns. This wide variety of characteristics prevents researchers and practitioners to compare existing design recovery tools effectively and to distinguish important design issues from technicalities. Moreover, to the best of our knowledge, there does not yet exist a comprehensive set of commonly agreed upon concerns by which to compare and characterise design recovery tools, despite some previous comparative frameworks [2, 3, 8, 10].

We present a comprehensive comparative framework for design recovery tools. This framework is the result of our personal experience with developing design recovery tools, of discussions held with other tool developers, and of reusing existing comparative frameworks. We believe that such a comparative framework is essential to highlight commonalities and differences in existing design recovery tools effectively. Also, a comparative framework is useful for the systematic replication of experimental study and to allow researchers to put in perspective their results. Indeed, replication studies are only interesting if both the programs of which the designs are recovered *as well as* the design recovery tools that are used can be compared. Finally, we hope that our framework can highlight weaknesses of existing design recovery tools, and thus suggest interesting research directions.

In Section 2, we explain the methodology used to build our comparative framework, discuss four existing frameworks and introduce two design recovery tools. In Section 3, we detail our comparative framework, its concerns and criteria, using the two introduced tools as examples. In Section 4, we discuss our comparative framework and state open issues. Finally, in Section 5, we conclude and sketch future work.

2 Constructing the Framework

We devised and adopted an iterative methodology to build our comparative framework. We present our

methodology, which was partly inspired by four existing comparative frameworks, and we introduce two design recovery tools that we use to exemplify our comparative framework.

2.1 Methodology

The construction of our comparative framework was initiated by a research question put forward by the organisers of the 6^{th} Workshop on Object-Oriented Reengineering, 2005 [5]. During the workshop, triggered by the difficulty to compare presented design recovery tools, the organisers emphasised the need for a comparative framework for design recovery tools. (Interestingly, the participants of the 1^{st} STEP Workshop on Empirical Studies in Reverse Engineering, 2005, independently highlighted the same need.) Part of the workshop was dedicated to drafting an initial list of concerns and criteria.

We adopted, refined, extended and validated this initial list, formulating each concern and criterion in more detail. We preferred a descriptive over a goal-oriented methodology because the purpose of the comparative framework is to enable qualitative comparisons among design recovery tools, not to rank them.

We followed an iterative and interactive process. Several iterations over intermediate versions of the framework were required to provide a framework that is as well-defined, comprehensive, and common as possible. During each iteration, we kept only sufficiently discriminating concerns and criteria.

Although we applied our comparative framework successfully on ten design recovery tools[1], we only present two tools for lack of space. Nevertheless, we do not claim to have obtained a complete and final list of concerns and criteria. We are currently validating our comparative framework with even more design recovery tools (including commercial ones) to refine further the presented framework.

2.2 Previous Frameworks

Biggerstaff. In his precursor work [3], Biggerstaff introduces a *program understanding landscape*, which is a two dimensional matrix. The dimensions are *deductive/algorithmic methods* versus *plausible reasoning/heuristic methods* on one axis, and *model-driven methods* versus *model-free methods* on the other axis. He uses this landscape to compare 9 program comprehension tools. It provides an interesting framework, from which we drew inspiration (in particular for our *Technique* concern and its criteria *Method* and *Model*, Section 3.5), although it does not address design recovery tools specifically.

Bellay and Gall. Bellay and Gall [2] compare 4 reverse-engineering tools using 4 functional categories: analysis, representation, editing/browsing, and general capabilities: The *analysis* category characterises their parsing capabilities; The *representation* category classifies their output representations (textual versus graphical and usability); The *editing/browsing* capability compares their editing and navigation capabilities; Finally, the *general capabilities* category includes concerns such as platforms and multi-user support. As Table 1 summarises, we reused most of these categories in our own framework.

Bellay and Gall's categories	Concerns and criteria in our framework (and Sections)		
Analysis	Technique	(3.5)	Semantics
Representation	Output	(3.6)	Representation
Navigation	Technique	(3.5)	Automation
Other	Tool	(3.8)	(Various)

Table 1. Mapping between Bellay and Gall's comparative framework and ours.

Gannod and Cheng. Gannod and Cheng [10] propose criteria to compare commercial and research design recovery tools by their outputs. They introduce a taxonomy of reverse-engineering techniques, consisting of 4 categories: *plan-based*, *parsing-based*, *transformation techniques*, and *translation techniques*, which they further decompose in *research* and *commercial* tools. Then, they present 4 semantic dimensions to compare outputs: *semantic distance* (how abstract is the output *wrt.* source code?), *semantic accuracy* (how accurate is the output *wrt.* source code?), *semantic precision* (how precise is the output *wrt.* source code?), and *semantic traceability* (how equivalent are the output and the source code semantically?). We reuse these categories in the *Method* criterion of our *Technique* concern (Section 3.5). We use the semantic dimensions as *Quality* criteria for our *Output* concern (Section 3.6).

Ducasse and Tichelaar. Ducasse and Tichelaar [8] studied the impact of design decisions on the meta-models and environments of reengineering tools. They introduce a design space to categorise reengineering

[1] Considered design recovery tools and techniques are: LiCoR, Ptidej, JIAD, JFREEDOM, Refactoring Crawler, Hammock graphs, Vanessa, web mining, TraceScraper, and FAMOOS.

tools, based on 9 axes: *Languages*, *level of detail*, *multiple models*, *grouping*, *extensibility*, *incremental loading*, *exchange format structure*, *entity reference*, and *meta-meta-model*. They focused on implementations details of the tools and on the meta-modeling techniques used in the implementations. All but the *grouping* axis are subsumed by our *Input*, *Technique*, and *Output* concerns. Also, they related their different axes to form a design space because choices in one axis typically have some impacts on other axes. For example, the *level of detail* of a tool influences its *scalability*.

2.3 Design Recovery Tools

Although we validated our comparative framework on 10 different design recovery tools, in this paper we exemplify our framework on two tools only. We deliberately choose two design recovery tools that we develop and extensively use, because we know these tools best. We chose our own tools for convenience only, but this choice does not reduce the soundness of the framework.

Ptidej. The Ptidej [2] tool suite (Pattern Trace Identification, Detection, and Enhancement in Java) is a set of tools to evaluate and enhance the quality of object-oriented programs, promoting the use of patterns at the language, design and architectural levels. The core of Ptidej is the PADL meta-model (Pattern and Abstract-level Description Language) to describe code-level models of programs using parsers for AOL [1], C++, and Java (including AspectJ) source code.

Ptidej provides mechanisms to abstract code-level models from programs through several analyses. In particular, it provides an analysis to abstract binary class relations [11] and thus, to promote code-level models into idiom-level models. It also supports other analyses to identify micro-architectures similar to design motifs (solutions of design patterns [9]) in idiom-level models, using the dedicated PtidejSolver constraint solver.

LiCoR. LiCoR [3] (*Li*brary for *Co*de *R*easoning) has been used to encode a variety of idioms and design patterns and to support program comprehension activities, such as: *checking* whether a piece of source code matches a pattern; *finding* all pieces of code that match a pattern; *searching* for all occurrences of some used patterns; *detecting violations* of and *enforcing* the consistent use of a pattern; and, *generating code* that complies with a pattern [13]. It is also used as a basic library for other tools that need to reason on source code. There are versions of LiCoR for the Smalltalk and Java programming languages.

LiCoR is a logic library written in SOUL [16], a Prolog-like logic programming language that is used to conduct a variety of logic meta-programming experiments. SOUL lives in symbiosis with its implementation language and runtime environment, Smalltalk. This symbiosis provides the ability to reason about and to manipulate Smalltalk objects within the logic paradigm, *e.g.*, SOUL possesses primitive constructs for evaluating Smalltalk expressions in logic rules. LiCoR uses SOUL to reify Smalltalk language constructs and to perform logic queries on these.

3 Comparative Framework

Concerns	Questions on the tool
Context	What is the context for its use?
Intent	What is its purpose?
Users	What is expected from its users?
Input	What input does it accept?
Technique	What algorithms does it use?
Output	What output does it provide?
Implementation	How is it implemented?
Tool	How mature is it?

Table 2. Concerns of design recovery tools.

Our comparative framework characterises a design recovery tool according to the *Context* in which it is applied, its *Intent*, its *Users*, its *Input* and *Output*, the *Technique* which it implements, its actual *Implementation*, and the *Tool* itself. We associate a question with each concern to ease the understanding of its rationale, as summarised in Table 2. The possible answers to each question are discriminating criteria among design recovery tools.

We present each concern together with its different criteria, using a systematic pattern. We introduce concerns with a *definition* and a question revealing their rationale. Then, we detail the associated *criteria* similarly. Finally, we briefly illustrate the concern using Ptidej and LiCoR as *examples*.

3.1 Context

Definition. "What is the context for the use of the design recovery tool?" The context of a design recovery tool imposes external constraints on the tool, such as the targeted software development methodology, the users' settings, the range of uses and the lifespan of the tool.

[2] http://www.ptidej.net
[3] http://prog.vub.ac.be/research/DMP/soul/soul2.html

- *Lifespan.* "Is the tool designed for single or long-term usage?" Tools may be developed at one point in time, for a particular design recovery task, or to be used for an extended period of time.
- *Methodology.* "To what methodology does the tool apply?" The methodology, such as clean-room, waterfall, RUP, or eXtreme Programming (XP), used to develop and maintain software affects the tools available to software engineers greatly. For example, the rapid iterations in XP exclude time-consuming batch tools, while such tools could be appropriate when using the waterfall methodology.
- *Range of Uses.* "What is the range of uses of the tool?" Is the tool dedicated to a particular purpose or is it a general-purpose tool? This criterion overlaps with, but differs from, *Lifespan* because tools used over an extended period of time may either be dedicated to a particular design discovery task or may be general-purpose tools.
- *Settings.* "Is the tool targeting a research or industrial setting?" Along with the methodology, the setting influences possible tools. For example, in a research setting, unstable tools with advanced capabilities could be accepted, while in an industrial setting stability is important.
- *Universe of Discourse.* (UoD) "Does the tool require access to all possible data?" In some contexts, a tool may access all possible data on a program to analyse (in-house program, closed-world), while in others, it may be limited to part of the data only (reverse-engineering, open-world).

Examples. LiCoR is meant to be used over long periods of time, for example to check violations of programming conventions regularly, although ad-hoc logic queries can be performed as well. It is a general-purpose tool that offers to its users the option of performing any kind of logic query over their programs (and includes a comprehensive library of rules). LiCoR primarily targets researchers that need to build more specific tools. Its UoD is open because its logic rules are customisable to any specific case study on which it is applied (for example, particular implementation strategies of patterns or programming conventions).

Ptidej is also developed for use over long periods of time, for example to ease the piecemeal understanding of the architecture of large programs, focusing on the identification of patterns (either at language, design, or architectural-level). It can be included in any development or maintenance methodology but is mainly targeted towards researchers to develop their own analyses and add-ons. The UoD of Ptidej is also open because it can analyse subsets of programs, inferring not-provided (unaccessible) data from provided data.

Context	Ptidej	LiCoR
Lifespan	Long-term	
Methodology	Any	
Range of Uses	Pattern-based analyses	General purpose
Settings	Research	
UoD	Open	

3.2 Intent

Definition. "What is the purpose of the design recovery tool?" The intent of a design recovery tool puts in perspective its long- and short-term objectives:

- *Long-term Objectives.* Long-term objectives are the goals beyond the tool itself. Design recovery is not a goal in itself but helps in achieving other, more important goals, such as support for co-evolution, software improvement or system migration.
- *Short-term Objectives.* Short-term objectives are the concrete goals of a tool, such as problem detection, program comprehension, or design (re)-documentation.

Intent	Ptidej	LiCoR
Long-term	Promoting quality through the use of patterns	Co-evolving source-code and design
Short-term	Recovering idiom- and design-level models	Codifying design information

Examples. Although the short term objectives of LiCoR are to codify design information in a logic meta programming language to support different activities, its long-term goal is to support co-evolution of source-code and higher-level designs [6].

The long-term objectives of Ptidej are to promote the use of patterns and to provide a tool suite to experiment with patterns, to assess and to improve the quality of programs designs and architectures. Its short-term objective is to recover idiom-, design-, and architectural-level models of programs.

3.3 Users

Definition. "What is expected from the users of the design recovery tool?" Design recovery tools make implicit and explicit assumptions about users' experience and impose on their users certain modes of interaction:

- *Acquaintance.* "Does the tool assume that its users know the analysed piece of software?" Users might need to know about the application domain, functionalities, physical decomposition, or source code.

- *Experience.* "Does the user need experience with the tool to use its full capabilities?" One tool may be straightforward to apply, while another one may require important learning efforts.

- *Prerequisites.* "What prerequisite knowledge should a user possess?" A tool may assume that its users possess certain knowledge, *e.g.*, users may need to be experienced developers, know a particular modelling or programming language, or have read the design pattern book [9].

- *Targeted User.* "What profile does the expected tool user have?" A tool may assume its users to be developers, maintainers, managers, or any combination thereof.

- *Type.* "Is the tool to be used by software engineers directly?" A tool may either assume that its users interact with it directly through its UI or that it is called indirectly from another tool.

We did not include a criterion on how a user interacts with the tool as this is discussed in the *Data Collection* criterion of the *Input* concern (Section 3.4) and the *Level of Automation* criterion of the *Technique* concern (Section 3.5).

User	Ptidej	LiCoR
Acquaintance	Source code	Design knowledge
Experience	Required	
Prerequisites	Binary class relations [11], design patterns	Logic programming, object oriented design
Targeted User	Educated software developer	
Type	Human/tool	Human

Examples. LiCoR users must have knowledge of logic programming, of design patterns, and of the LiCoR logic library to reason about the source code and to codify design information. Targeted users are (experienced) developers acquainted with the design of the program on which they want to reason.

Ptidej users must be acquainted with the source code of the program under analysis. They also must be familiar with binary-class relations, design patterns, and the other patterns they want to recover. Ptidej can be used as a façade for other tools through its extension and analysis mechanisms.

3.4 Input

Definition. "What input does the design recovery tool accept?" We decompose this concern in:

- *Assumptions.* "What are the assumptions on the input data?" A tool may assume that certain structural or semantic information exists in the input implicitly (*e.g.*, patterns) or that the input conforms to some criteria (*e.g.*, "good" object-oriented style). These assumptions, when not explicit, hinder the comparison of tools with one another.

- *Automation.* "Is the collection of data automated?" The collection of input data can be automatic (from the start of the tool until obtaining the results), semi-automatic (interaction from the user of the tool is required), or manual.

- *Data Collection.* "Where does the data needed by the tool come from?" A tool may use data obtained from static analyses, from dynamic analyses, or a combination thereof. We further distinguish tools that *require* certain kinds of data (if these are missing, the tool cannot work) and tools that can *optionally* complement one kind of data with another (providing the tool with several kinds of data is likely to produce more accurate results).

- *Documentation.* "What input, other than the program, can the tool take?" A tool may use existing design documentation like source code comments and annotations, implementation documentation (implementation choices, improvements, to do lists, bug reports), design and architectural documentation, and user documentation.

- *Model.* "What is the underlying *meta*-model describing the input?" The level of detail or of maturity of the meta-model is an indicator for the maturity of the tool. Moreover, tools working on similar meta-models are potentially more inter-operable. The underlying model could be, for instance, a Smalltalk parse tree representation or the UML meta-model.

- *Precision.* "Can the tool handle imprecisions in the input?" Imprecisions may be syntactic or semantic inconsistencies in the data, *e.g.*, a source-code file which cannot be parsed or compiled fully.

- *Representation.* "How must the input be represented concretely?" The representation describes the actual *physical* embodiment of the input. For

example, the representation can be a running Smalltalk image or a set of Java files.

- *Type of Data.* "What type of data does the tool need?" In addition to the kind of data that has been collected (*Data Collection*), we further distinguish tools based on the actual type of data they use.

- *Versions.* "Can the tool handle multiple versions of the software simultaneously?" We distinguish tools that *can* handle more than one version and tools that *require* two or more versions.

Input	Ptidej	LiCoR
Assumptions	Object-oriented programs with recurring patterns	
Automation	Semi-automatic	
Data Source	Hybrid	Static
Documentation	No	
Model	PADL meta-model	Parse-trees reified as facts
Precision	Correct syntax	Compilable
Representation	PADL model	Smalltalk image
Type of Data	Program entities, binary class relations	Method parse trees
Versions	Multiple	Single

Examples. LiCoR reifies method parse-trees and, thus, requires compilable code. It represents the parse trees as logic facts. It works interactively on a running Smalltalk image and changes to the code are reflected immediately in the logic facts (or even in the image when code is generated). The user indicates where a logic query runs (for example, on all the methods of classes in a certain namespace), and the rest of the collection is automatic. LiCoR reasons about one version of a system at a time, the version that is available in the Smalltalk image in which it runs. The source code must be fairly well structured for LiCoR to be usable and to provide interesting results through logic queries.

Ptidej assumes that the source code follows some loose but "good" programming style (*e.g.*, fields must be private). It builds models of programs described with the PADL meta-model using parsers, which handles missing data (*e.g.*, they can analyse subsets of large programs in which certain class definitions are missing). Models from static data can be enhanced with dynamic data. It does not use existing documentation automatically. Ptidej promotes models into higher-level models through different analyses (*e.g.*, design pattern identification). It can compare models to highlight modifications among versions of programs.

3.5 Technique

Definition. "What algorithms does the design recovery tool use?" The technique is the core of the tool and is a major discriminating concern. We decompose this concern in:

- *Adaptability.* "Is the technique adaptable?" A technique is adaptable if it can adapt fairly easily to other kinds and types of input to produce (possibly different) outputs.

- *Automation.* "Is the tool automated? Is the tool interactive or can it run in batch mode?" The level of automation characterises tools by their required user input. A tool may be fully automatic, may provide intermediary results, or may require user interaction to direct its work. It may also require manual interpretation of the produced results. The "editing/browsing" category of Bellay and Gall [2] is related to this criterion: tools may offer or require browsers for users to guide the tools.

- *Complexity.* "What is the complexity of the technique?" The *computational* complexity evaluates the theoretical performance of the technique. The complexity could be expressed with the big O notation or with a Likert scale.

- *Determinism.* "Is the technique deterministic?" A technique is deterministic if, being applied many times on a same input, it produces a same output.

- *Explicative.* "Does the tool provide explanations on the produced output?" A tool may explain how it computed its output from the given input.

- *Fuzzyness.* "Can the technique return approximate results?" A technique can return approximate results or only complete results (and, thus, *no results at all* if it is too strict *wrt.* to the input).

- *Granularity.* "What is the level of granularity of the technique?" A tool may reason about classes and packages, about methods and method dependencies, or about statements. Granularity is related to the input, but not necessarily identical. For example, the input of the tool could be execution traces from which only the actual names of the messages are used.

- *Incremental.* "How incremental is the technique?" A technique is incremental if, after analysing parts of a program, analysis of additional entities takes advantage of previous analyses and does not require a complete (re-)analysis.

- *Iterative.* "Is the technique iterative?" A technique may be able to refine its results through successive iterations with adjusted inputs and parameters.

- *Language Independence.* "Is the technique language independent?" A technique is 'language independent' if its computation processes do not depend theoretically on specificities of the programming language from which the input is extracted.

- *Maturity.* "Is the technique mature?" A technique is 'mature' if it is well-known in the domain of software engineering and/or has been applied to related domains successfully. A related question is whether the technique is well-documented.

- *Method.* "What algorithm does the tool use?" Biggerstaff [3] distinguishes between *deductive/algorithmic methods* and *plausible reasoning/heuristic methods*. He further distinguishes model-free (or bottom-up) from model-driven (or top-down) techniques. Gannod and Cheng [10] distinguish *plan-based*, *parsing-based*, *transformation techniques*, and *translation techniques*.

- *Model.* "On which model does the tool operate?" This criterion concerns the internal computational model of the tool. For example, in the case of logic inferencing, the model is composed of Horn clauses, while in the case of constraint programming, the model is a constraint network.

- *Scalability.* "How scalable is the technique?" A technique is scalable if it can deal with varying amounts of data as its input. What is the largest system that has been or can be handled by the technique in a reasonable amount of time?

- *Semantics.* "What is the semantic level of the data about which the tool reasons?" This criterion corresponds to Bellay and Gall's *analysis* category [2]. We distinguish between:

  - *Lexical.* Lightweight reasoning about the program at a lexical level: sequences of characters, regular expressions...
  - *Structural.* Reasoning on the structure of the input, *e.g.*, parse trees.
  - *Semantical.* Deep reasoning on the semantics of the input, *e.g.*, type information.

Technique	Ptidej	LiCoR
Adaptability	Yes	
Automation	Semi-automatic	
Complexity	High	
Determinism	Yes	
Explicative	Yes	No
Fuzzyness	Yes	No
Granularity	Method dependencies	Methods and statements
Incremental	Yes	No
Iterative	Yes	No
Language Independence	Yes	
Maturity	Mature	
Method	Explanation-based constraint-programming (algotithmic / model-driven / heuristics)	Logic inferencing (deductive / model-driven / heuristics)
Model	Constraint networks	Horn clauses
Scalability	High	Medium
Semantics	Semantical	Structural

Examples. The method used by LiCoR is partly deductive (logic inferencing) and partly heuristic, because the reasoning about design patterns is partly implemented through heuristics. LiCoR does not return approximative results: logic queries either succeed or fail, approximations must be embedded in the queries. Design recovery with LiCoR is model-driven because it requires models of the patterns to recover or to verify. LiCoR is very adaptable. For example, we have written other logic libraries in LiCoR that reify dynamic data (execution trace information) and that interface with the FAMIX meta-model [7] instead of parse trees. The scalability of LiCoR depends on the logic queries. However, quite a large amount of memory is used because LiCoR reifies source code completely, which prohibits the analysis of large programs at one time.

Ptidej is explicative, fuzzy, and scalable partly because it uses explanation-based constraint programming [12] to perform design pattern identification. Explanation-based constraint programming provides *explanations* on the design pattern found and allows relaxing constraints semi-automatically to provide approximate occurrences. Also, its meta-model has been designed specifically to describe large programs. Ptidej also includes an efficient algorithm for design pattern identification based on bit-vectors (from bioinformatics). It is incremental because program entities can be added to a model at will. It is iterative because analyses produce models that can be further

analysed. It is model-driven like LiCoR because it requires models of the patterns to recover.

3.6 Output

Definition. "What output does the design recovery tool provide?" The output corresponds to all by-products obtained by the user and generated by the tool when applied on a given input. We decompose this concern in:

- *Human-readable.* The output might be human-readable or might require extra processing for visual display or interpretation. For example, a manual interpretation of the output may be required if given as a set of numeric data or Gallois lattices.
- *Level of Detail.* This criterion characterises the information provided as output qualitatively, for example, it distinguishes between tools answering "a Visitor is present" with "Class X, Y, Z, method M, N, O, and ... form a Visitor".
- *Model.* This criterion is similar to the *Model* criterion of the *Input* concern in Section 3.4.
- *Quality.* This criterion assesses the quality of the produced output and can be decomposed further in detailed quality characteristics like those proposed by Gannod and Cheng [10]: semantic distance, semantic accuracy, semantic precision, and semantic traceability.
- *Representation.* This criterion corresponds to Bellay and Gall's [2] representation category and differentiates possible output encoding formats. An important issue here is the usability of the representation by another tool.
- *Type of Data.* The type of data highlights the expected outcome of applying the tools: names of program entities, idioms, hooks and template methods, meta-patterns, design patterns...

Output	Ptidej	LiCoR
Human-readable	No	Yes
Level of Detail	Class	Depends on query
Model	PADL	Frames of bindings
Quality	High	High
Representation	Ini files	Bindings of program entities to logic variables
Type of Data	Program entities and patterns	

Examples. LiCoR queries may return discovered patterns and program entities playing a role in those patterns, as bindings to the logic variables of those queries. These results can be understood immediately by a user or used as input for further queries. How detailed is the result depends on the actual query (*i.e.*, on how many parameters it has).

The output of Ptidej is not human-readable because it requires Ptidej's user-interface to display identified patterns for interpretation. It is described with the PADL meta-model and encoded as ini files, which can be saved and reused later for comparisons as programs evolve. Pattern identification focuses on the roles of classes in patterns.

3.7 Implementation

Definition. "How is the design recovery tool implemented?" We decompose this concern in:

- *Dependencies.* "Are there any implementation dependencies?" This criterion evaluates dependencies on libraries needed by the implementation.
- *Language.* "What is the programming language used in the implementation?" This criterion is straightforward to answer for most tools.
- *Maintained.* "Is the tool maintained?" A tool is more interesting for other researchers and practitioners if it is being actively maintained.
- *Platforms.* "What runtime platforms does the implementation support?" The range of platforms on which the implementation runs is important because implementation choices may limit the number of possible platforms.
- *Quality.* "What are the quality characteristics of the implementation?" As for any program, the implementation must possess recognised quality characteristics, such as flexibility, extensibility, readability, ...

Implementation	Ptidej	LiCoR
Dependencies	None	SmaCC
Language	Java	Smalltalk
Maintained	Yes	
Platforms	Sun J2SE	VisualWorks
Quality	High	

Examples. LiCoR has a high-quality implementation in VisualWorks Smalltalk and is still maintained actively. It relies on the SmaCC compiler framework.

Ptidej does not depend on any particular library. It is implemented in Java and is actively maintained and extended, for example to evaluate quality of aspects.

3.8 Tool

Definition. "How mature is the design recovery tool?" Maturity distinguishes simple handy scripts from industrial-strength tools and is highlighted by several criteria. Though some of these criteria seem overlapping with criteria of the *Technique* concern, they are not identical, *e.g.*, a technique may be *language independent* but its implementing tool can be restricted to some languages, for ease of development. This concern somewhat corresponds to Bellay and Gall's category *Other capabilities* [2]. We distinguish:

- *Documentation.* "Is the tool documented?" A tool can be associated with a user-directed documentation or scientific articles describing the technique and its implementation.
- *Kind of License.* "What license applies to the tool?" Licenses are an important factor in the choice of a tool because some licenses may constrain the use of the tools in research or industrial settings.
- *Language Independence.* "Is the tool language independent?" A tool may be language dependent, disregarding the language independence of the technique used.
- *Multi-user support.* "Does the tool support collaborations?" A tool may support many collaborative users simultaneously or may only support one at a time.
- *Quality.* "What are the quality characteristics of the tool?" As for its implementation, a tool must possess qualities intrinsic of software engineering, such as usability, portability...
- *User-base.* "Does the tool have a user-base?" A tool with a user-base (users outside of its development setting) is more interesting because it potentially offers a community to interact with, and to obtain feedback from.

Tool	Ptidej	LiCoR
Documentation	Articles, source code	
Kind of License	GPL	LGPL
Language Independence	AOL, C++, Java (including AspectJ)	Java, Smalltalk
Multi-user	Single user only	
Quality	High	
User base	Other universities	

Examples. LiCoR is distributed as a Visualworks Smalltalk application under the LPGL license and has been used and documented by researchers in several universities. In addition to reasoning about Smalltalk, an extension for Java is available.

Ptidej is available under GPL. It provides parsers for several programming languages and is actively used in several universities around the world. Its documentation is mainly composed of research articles and technical reports.

4 Discussion

We believe that our comparative framework is well-defined, comprehensive, and common because it has been built from clearly defined concerns and criteria, because those concerns and criteria were defined and refined iteratively relying on previous work and on the community's insights, and because it allowed comparing ten different and independently developed design recovery tools (of which two were presented).

However, further validation is required to verify whether our framework enables an objective comparison of tools. Such a validation may lead to *refinements* of and *extensions* to the framework, but may also lead to remove criteria that are not sufficiently discriminating. Once a thorough comparison of existing tools has been made using the proposed comparative framework, we will attempt at proposing a fine-grained *taxonomy* of design-recovery tools, characterising similar tools and highlighting holes in the design space of existing tools.

We faced many difficult choices regarding the concerns and criteria while building the comparative framework. We tried to achieve a delicate balance between having sufficient criteria to discriminate among tools, and avoiding to overfeature the framework. We do believe that we have achieved our goal of offering a comparative framework which provides a solid basis on which to build further with the help of the community, though some of our choices could be further discussed.

5 Conclusion and Future Work

Many design recovery tools exist in industry and academia. Although comparing these tools is essential to understand their differences, to ease replication studies, and to discover what tools are lacking, such a comparison is difficult because there is no well-defined, comprehensive, and common comparative framework. We developed such a comparative framework and illustrated it on two design recovery tools: Ptidej and

LiCoR. Our comparative framework comprises eight concerns, which where further decomposed into fifty-three criteria and which we applied on ten design recovery tools successfully. Due to space limitations, we were not able to include a complete comparison and classification of existing and generally known design recovery tools for object-oriented programming languages as well as (possibly) for other paradigms. Moreover, for our framework to be actually usable, it should include tools, such as questionnaires, checklists, and so on, supporting its users during the task of evaluation. An empirical validation to assess the quality of the framework (usability, completeness...) must also be performed. Finally, the framework should be compared against other existing framework thoroughly, such as the ISO/IEC 14 102. This will be the subject of a forthcoming survey paper for which we hope to receive feedback from developers of design recovery tools in the community as a general and concerted effort to provide a comprehensive picture of the current state of the art. Also, we will investigate the construction of a design space similar to Ducasse and Tichelaar's for our larger sets of concerns and criteria.

Acknowledgements

The authors gratefully acknowledge Ademar Aguiar, Carlos López, Raúl Marticorena, and Oscar Nierstrasz' contributions to the concerns and criteria.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In S. Tilley and G. Visaggio, editors, *proceedings of the* 6^{th} *International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In I. Baxter and A. Quilici, editors, *Proceedings of WCRE1997*, pages 2–11. IEEE Computer Society Press, 1997.

[3] T. Biggerstaff, B. Mitbander, and D. Webster. The concept assignment problem in program understanding. In V. Basili, R. DeMillo, and T. Katayama, editors, *Proceedings of ICSE 1993*, pages 482–498. IEEE Computer Society Press / ACM Press, 1993.

[4] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22:36–49, 1989.

[5] S. Demeyer, K. Mens, R. Wuyts, Y.-G. Guéhéneuc, A. Zaidman, N. Walkinshaw, A. Aguiar, and S. Ducasse. Report of the 6^{th} international workshop on object-oriented reengineering, 2005. To be published.

[6] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of SACT 2000*, pages 207–224. Kluwer Academic Publishers, January 2000.

[7] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering Environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, 2005.

[8] S. Ducasse and S. Tichelaar. Dimensions of reengineering environment infrastructures. *International Journal on Software Maintenance: Research and Practice*, 15(5):345–373, Oct. 2003.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[10] G. C. Gannod and B. H. C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In F. Balmas, M. Blaha, and S. Rugaber, editors, *Proceedings of WCRE 1999*, pages 77–88. IEEE Computer Society Press, 1999.

[11] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In D. C. Schmidt, editor, *Proceedings of OOPSLA 2004*. ACM Press, 2004.

[12] N. Jussien. e-Constraints: Explanation-based constraint programming. In B. O'Sullivan and E. Freuder, editors, 1^{st} *CP workshop on User-Interaction in Constraint Satisfaction*, 2001.

[13] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. *Elsevier Journal on Expert Systems with Applications*, 23(4):405–431, November 2002.

[14] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, October 1968.

[15] B. Randell and J. Buxton, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*. Scientific Affairs Division, NATO, October 1969.

[16] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of TOOLS USA 1998*, pages 112–124. IEEE Computer Society Press, 1998.