This paper has been accepted at APSEC 2004.

A Systematic Study of UML Class Diagram Constituents for their Abstract and Precise Recovery

Yann-Gaël Guéhéneuc
Département d'informatique et de recherche opérationnelle
Université de Montréal – CP 6128 succ. Centre Ville
Montréal, Québec, H3C 3J7 – Canada

E-mail: guehene@iro.umontreal.ca

Abstract

Existing reverse-engineering tools use algorithms based on vague and verbose definitions of UML constituents to recover class diagrams from source code. Thus, reverse-engineered class diagrams are neither abstract nor precise representations of source code and are of little interest for software engineers. We propose an exhaustive study of class diagram constituents with respect to their recovery from C++, Java, and Smalltalk source code. We exemplify our study with a tool suite, PTIDEJ*, to reverse-engineer Java programs as UML class diagrams abstractly and precisely. The tool suite produces class diagrams that help software engineers in better understanding programs.*

1 Problem

UML class diagrams describe the static structure of programs at a higher level of abstraction than source code. Class diagrams are an invaluable help for software engineers—both developers and maintainers—to understand programs architectures, behaviours, design choices, and implementations.

However, class diagrams produced during the design phase are often forgotten during the implementation phase, under time pressure usually. They present major discrepancies with implementation frequently. Such divergent class diagrams are of little help to maintainers who must support programs after their release.

Thus, maintainers need means to recover class diagrams from program implementation. These means must be *automated* considering the large size of programs. These means must produce recovered class diagrams with *characteristics* useful to maintainers.

There exist several characteristics to compare reverse-engineering tools and to assess recovered class diagrams. However, we advocate that only abstractness and preciseness of class diagrams matter to maintainers. We show with a simple example that no existing mainstream reverse-engineering tool produce *abstract* yet *precise* class diagrams because of the ambiguous definitions of UML class diagrams constituents[1] [7, 9, 23] and because of the lack of correspondence between source code constructs and constituents. Hence, reverse engineering is a challenge to software engineers and tool builders. This challenge can be met with a detailed and systematic study of class diagram constituents and source code constructs only.

The contribution of this paper is two-fold. First, we propose an exhaustive study of UML class diagram constituents to assess their abstract and precise automated recovery from C++, Java, and Smalltalk source code constructs. Second, we exemplify the correspondence with Java source code using our tool suite for abstract and precise reverse-engineering, PTIDEJ. We compare the results of our tool suite with these of other reverse-engineering tools and show that they provide useful detailed pieces of information to maintainers. This paper is also a step towards unambiguous definitions of UML class diagram constituents through their correspondence with source code.

Section 2 exemplifies the problem of abstractness and preciseness of UML class diagrams and defines our characteristics. It shows that existing reverse-engineering tools recover neither abstract nor precise class diagrams. Section 3 details our systematic study of the abstract and precise recovery of UML class diagrams. Section 4 presents our tool suite and a comparison with other existing tools. Section 5 summarises related work. Finally, Section 6 concludes and introduces future work.

[1] UML does not claim to be unambiguous. Loose definitions ease analysis and design but impede automated tools.

2 Abstractness and Preciseness

2.1 Case Study

Source code 1 describes a Java program composed of a class `Example1` that aggregates instances of a class `A`, *i.e.*, `Example1` declares an array of type `A` as instance variable. The class diagram recovered from this source code must precisely reflect the two classes, their methods, and the aggregation relationship between them—these are pieces of information maintainers would recover *manually*, while abstracting the implementation details of the aggregation relationship, to grasp quickly the architecture and the behaviour of the program. Thus, a class diagram that precisely abstracts Source code 1 must look like this in Figure 1.

The class diagram in Figure 1 shows the two classes `Example1` and `A`. It shows their sets of methods as well as the aggregation relationship between instances of `Example1` and `A`. Although the `addA()`, `getA()`, and `removeA()` methods participate in the aggregation relationship, we believe that they must appear in the class diagram because they would appear in a sequence diagram of the program and in the bodies of methods using instances of class `Example1`.

We could implement the same program using the Java collection API also, for example as in Source code 2, using the `List` interface and its `ArrayList` implementation. The corresponding class diagram would not change because it abstracts the program implementation precisely. Thus, the class diagram in Figure 1 is useful to maintainer because it represents the Java program implemented by either Source code 1 or 2 abstractly yet precisely.

2.2 Characteristics

Thus, class diagrams must possess the characteristics of abstractness and preciseness with respect to a program source code, as shown by the class diagram in Figure 1 with respect to the Java program implementations in Source codes 1 and 2. The Random House dictionary of the English language (1^{st} edition) defines:

- Abstract: Something that concentrates in itself the essential qualities of anything more extensive or more general, or of several things; essence.
- Precise: The extent to which a given measurement agrees with a standard value.

These two characteristics are similar to the concepts of semantic distance (abstractness) and sematic accuracy (preciseness) proposed by Gannod and Cheng [12]. We use nominal data (two-values scales) to measure

```
public class A { ... }

public class Example1 {
    private A[] listOfAs = new A[10];
    private int numberOfAs = 0;

    public void addA(final A a) {
        this.listOfAs[numberOfAs++] = a;
    }
    public A getA(final int index) {
        return this.listOfAs[index];
    }
    public void removeA(final A a) {
        // ...
    }
    public static void main(final String[] args) {
        final Example1 example1 = new Example1();
        example1.addA(new A());
        // ...
    }
}
```

Source code 1. A simple case study.

```
public class Example2 {
    private List listOfAs = new ArrayList();

    public void addA(final A a) {
        this.listOfAs.add(a);
    }
    public A getA(final int index) {
        return (A) this.listOfAs.remove(index);
    }
    public void removeA(final A a) {
        this.listOfAs.remove(a);
    }
    public static void main(final String[] args) {
        final Example2 example2 = new Example2();
        example2.addA(new A());
        // ...
    }
}
```

Source code 2. Alternative implementation.

the extent to which class diagram possess these characteristics: A class diagram is either *abstract* or *not abstract*, either *precise* or *not precise*. By extension, recovery process of a constituent is *abstract* (respectively *precise*) if and only if the recovered constituent is an *abstract* (respectively *precise*) representation of a(some) source code construct(s), else its recovery is *not abstract* (respectively *not precise*).

2.3 Reverse-engineering tools

Existing reverse-engineering tools do not recover class diagrams from source code abstractly and precisely. For example, Rational develops a tool, Rose, to uphold round-trip engineering between UML diagrams and program implementation. Figure 2 shows the class diagram recovered from Source code 1 using Rose version 2002.05.20. This class diagram is precise but is not abstract. It is precise because it reflects the program implementation: Classes `A` and `Example1`, their methods, and the relationship between classes `Example1` and `A`, supported by the `listOfAs` array. The class diagram

is not abstract because the relationship is described as an association (through an array), whereas a maintainer would have described it as an aggregation (not detailing its implementation), given the array and the `addA()`, `getA()`, and `removeA()` methods.

Figure 3 shows the class diagrams recovered with ROSE from the alternative implementation in Source code 2. The class diagram is neither abstract nor precise. It shows the two classes and their methods. However, the aggregation relationship between the two classes is recovered as an association relationship with the `List` interface, support of the aggregation relationship: As shown in Source code 1, the aggregation relationship could be implemented with an array equally. (In fact, the aggregation relationship could be implemented with any kind of collection.) Thus, the class diagram is not abstract because it does not hide implementation details of the aggregation relationship, it is not precise because it does not reflect what maintainers would have produced manually.

Other tools, such as ARGOUML version 0.14.1, CHAVA [19], FUJABA version 4.0.1 [21], IDEA [3], Borland TOGETHER/J version 6.2, and WOMBLE [16] recover neither abstract nor precise class diagrams from Source codes 1 and 2: Some might be used with fair success as modelling tools, but they all have limited recovery capabilities, as we show in Section 4.

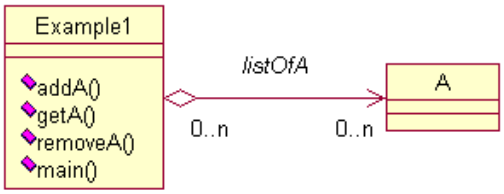

Figure 1. Class diagram recovered manually.

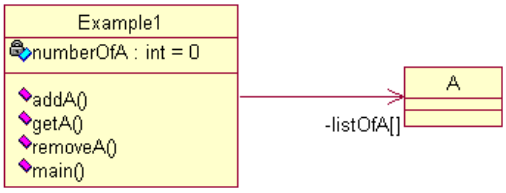

Figure 2. Class diagram recovered from Source code 1 using RATIONAL ROSE.

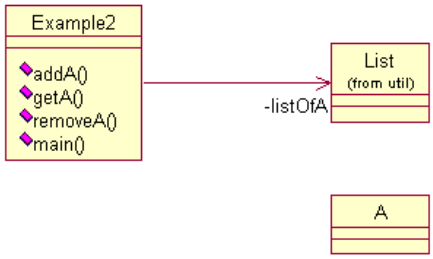

Figure 3. Class diagram recovered from Source code 2 using RATIONAL ROSE.

3 Class Diagram Constituents

Limitations of existing reverse engineering tools come from the lack of systematic study of UML class diagram constituents. Indeed, the abstractness and preciseness of automatically recovered class diagrams depends on the possibility to recover abstractly yet precisely their constituents from source code entirely.

We review all the constituents of UML class diagrams, as defined in the UML v1.5 specifications, Chapter 3, Part 5 "Static Structure Diagrams", sections 3.21 through 3.53 [22] (Part 5 in the following), *i.e.*, classifiers features, classifiers, classifiers relationships, and miscellaneous constituents. We suggest reverse engineering techniques to recover these constituents from source code written in C++, Java, and Smalltalk. We choose C++, Java, and Smalltalk because these are mainstreams programming languages which exhibit interesting non-overlapping features with respect to reverse engineering.

3.1 Classifiers Features

Attributes, methods, and operations define both the behaviours of classifiers and their implementation. Thus, presence of attributes, methods, and operations in source code guides the recovery of classes, types, and interfaces (details follow in Section 3.2). Attributes are named slots that describe ranges of values. Operations are services that instances of classifiers offer. They may be implemented using several methods. Operations *specify* behaviours of classifiers whereas methods *implement* these behaviours. Although methods are not defined in Part 5, we believe important to distinguish methods and operations.

Attribute. It is possible to recover the attributes of a classifier precisely and abstractly using static analyses because attributes are source code constructs.

Method and Operation. Operations do not exist as such in source code written in C++, Java, or Smalltalk, which define methods only. However, it is possible to distinguish operations from methods because methods concretely realise operations. Thus, we propose four overlapping rules to recover methods and operations from source code:

- Any abstract public methods[2] defined in a

[2]Abstract protected methods in C++ and Java define template methods [11].

classifier describes operations of this classifier. Thus, operations can be recovered syntactically: Virtual pure methods in C++, `abstract` methods in Java, methods with body `self subclassResponsibility` in Smalltalk, and the methods implementing these operations.

- Public methods (as well as protected and private methods in C++ and Java) can be recovered syntactically as methods in class diagrams.
- Overloaded methods can be recovered as a single operation because they offer a semantically-equivalent service on instances of different classifiers, *e.g.*, methods `compareTo(Date)` and `compareTo(Object)` of class `java.util.Date` can be recovered as operation `compareTo(Object)`.
- Operations could be inferred using data provided by methods comments or programs documentation also. However, it is yet unclear how techniques using comments and documentation could help in reverse engineering programs automatically.

Thus, it is possible to recover methods and operations from source code abstractly and precisely using static analyses of source code constructs and the preceding classification of operations and methods.

3.2 Classifiers

Classifiers are either classes, nested classes, interfaces, parameterised classes, bound elements, or data types. Data types are not defined in Part 5 but are often referenced so it is important to detail their recovery. Classes decompose through stereotypes in either *Enumeration*, *ImplementationClass*, *Metaclass*, *Powertype*, *Type*, or *Utility*. The C++, Java, and Smalltalk programming languages do not distinguish among all these possible classifiers. Recovery of classifiers requires to infer the roles of classes in source code as well as the roles of structures in C++ and of interfaces in Java.

Table 1 summarises the definitions and possible features of classes, implementation classes, interfaces and types. Other kinds of classifiers are discussed in the following. Table 1 shows that it is possible to distinguish classes, implementation classes, interfaces, and types according to their definitions using the presence of attributes, operations, methods, and relationships to other classifiers or from other classifiers. Recovery of these classifiers from source code can be based on attributes, methods, and operations only, because relationships do not further distinguish these classifiers.

Class. Classes can be recovered from source code automatically. Indeed, classes exist as syntactical constructs in class-based object-oriented programming languages such as C++, Java, and Smalltalk. Any class declaring attributes, operations, and methods can be recovered as a class abstractly and precisely

Nested Class. Nested classes can be recovered from source code written in C++ or in Java syntactically. Smalltalk does not allow nested classes. In Java, nested classes can be defined in methods bodies also. Such *local* and *anonymous* classes should not appear on class diagrams because they participate in the inner working of methods only and UML does not define constituents to describe classes in methods explicitly. Thus, recovery is not abstract but precise.

Type and Implementation Class. Types and implementation classes can be recovered from source code syntactically. Classes in source code declaring attributes and only concrete methods can be recovered as implementation classes. Classes declaring attributes and only operations can be recovered as types.

Interface. It is possible to recover interfaces in Java source code syntactically. In source code written in programming languages such as C++, abstract classes defining only pure virtual methods can be recovered as interfaces. In source code written in Smalltalk, interfaces do not exist because of dynamic typing. However, a set of instances answering a same set of messages could be considered as implementing a common interface (if the set of messages changes dynamically, more than one interface must be created). Thus, interfaces can be recovered from C++ and Java source code abstractly and precisely. In Smalltalk, interface recovery requires dynamic analyses or type inference and is thus abstract but not precise.

Parameterised Class. Parameterised classes exist in source code written in C++ (and in Java v1.5, which includes the Java Specification Request n°14, introducing parameterised classes) syntactically. Thus, it is straightforward to recover parameterised classes abstractly and precisely. They do not exist in Smalltalk and thus cannot be recovered abstractly nor precisely.

Bound Element. As for parameterised classes, bound elements exist in C++ source code as well as in Java v1.5 syntactically. Thus, it is possible to recover bound elements from source code abstractly and precisely. Bound elements do not exist in Smalltalk.

Metaclass. Metaclasses as such do not exist in C++ and Java. In these programming languages, it is not possible to declare a metaclass explicitly. Smalltalk provides classes that represents metaclasses and that can be manipulated to modify the behaviours of classes. Thus, it is possible to recover metaclasses from Smalltalk source code abstractly and precisely.

Classifiers	Roles	Attributes	Operations	Methods	Relationships to other | from other classifiers	
Class	Define a set of objects with similar structure, behaviour, relationships	✓	✓	✓	✓	✓
Implementation class	Define the physical structure and methods of objects, as in C++...	✓	×	✓	×	×
Interface	Define a limited part of the behaviour of an actual class	×	✓	×	×	✓
Type	Define a domain of objects with operations, without implementation	✓	✓	×	✓	✓

Table 1. Features of classes, implementation classes, interfaces, and types. (✓ = Yes, × = No)

Powertype. A powertype is a type which instances are subtypes of another type [20]. It is impossible to encounter power types in source code written in C++, Java, or Smalltalk, because instances of classes cannot be classes. Thus, power types are a "conceptual, or analysis, notion" [20, page 256] which cannot be recovered from source code abstractly and precisely.

Data Type. Data types are either primitive types, enumerations, or classes with pure functions only. It is possible to recover primitive types from C++ and Java source code automatically because these programming languages possess primitive types. In Smalltalk, the virtual machine handles the instantiation of a subset of all classes, such as `SmallInteger` or `Float`, to manage "primitives" values, thus the recovery of primitive types in Smalltalk is possible using this subset. Classes with pure functions can be recovered from source code only by analysing method bodies to ensure that parameters are not modified. However, such analyses are time- and space-consuming and often undecidable. Moreover, a class qualifies as a data type only if it inherits from other data types (or from `Object` in Java and Smalltalk): Classes with pure functions but extending non-data types are not data types. Thus, recovery of data types is abstract yet not precise.

Enumeration. Enumerations are data types which instances are sets of literals. Enumerations are source code constructs in C++ and in Java v1.5, which includes the Java Specification Request n°201. In Java, interfaces declaring constant values only (public static final fields) can be recovered as enumerations also. In Smalltalk, variables shared by one or more objects and–or one or more classes are grouped in pools. Pools are used to share constant data. For example, the `TextConstants` pool, shared by classes displaying and editing texts, stores ASCII character codes. Thus, it is possible to recover enumerations from source code abstractly and precisely.

Utility Class. Utility classes are programming convenience constructs only. They declare shared global variables and procedures. We propose to recover utility classes from classes declaring only static methods (and–or attributes) to distinguish utility classes from classes and implementation classes. Thus, it is possible to recover utility classes abstractly and precisely.

3.3 Classifiers Relationships

UML defines several relationships among classifiers: Generalisations, associations, aggregations, compositions, links, and dependencies. These relationships express high-level design concepts and allow software engineers to design and to understand programs at a higher-level of abstraction than source code. Recovery of relationships is important because these relationships differentiate source code and design. We discuss the recovery of binary association, aggregation, and composition relationships briefly, because these relationships present interesting difficulties. We discuss other kinds of relationships separately.

Several authors studied the definitions and–or the recovery of binary association, aggregation, and composition relationships, see [6, 8, 14, 15, 16, 17] to cite but a few. Difficulties in the recovery of these relationships come (1) from the lack of definitions of the relationships at implementation- and design-level and (2) from the lack of precise algorithms to recover these relationships in source code [13]. In particular, we disagree with the assertion by Booch *et al.* that "Simple aggregation is entirely conceptual and does nothing more that distinguish a 'whole' from a 'part'." [5, page 146]. Thus, we propose in another work [13] consensual definitions for these relationships using four programming language-independent properties: Exclusivity, invocation site, multiplicity, and lifetime. We show that the three relationships can be distinguished from one another using their properties and that their recovery decomposes in two steps: Recovery of binary association and aggregation relationships; Conversion of appropriate aggregation in composition relationships.

Association. Association is a generic term denoting binary association relationships, aggregation and composition relationships, and n-ary association relationships. *Binary association* is used to denote the particular association relationship. Recovery of associations depends on the recovery of the different forms entirely.

Binary association. Binary association relationships can be recovered by analysing the source code syntactically: A binary association relationship exists whenever a method invocation exists between a class `A` and a class `B` (or their instances), because the distinguishing property of an association relationship is the presence of a method invocation [13]. Recovery of binary associations is abstract and precise.

Association End. Association ends describe the visual display of associations. They show multiplicities, aggregation and composition adornments. The recovery of association ends depends on the recovery of the different possible adornments only [17].

Multiplicity. Multiplicities specify the number of instances of two classifiers involved in an association. Recovery of multiplicities can be performed both in a static [13, 16] and in a dynamic way [17]. Statically, multiplicities can be recovered by analysing the source code of classifiers declaring fields. If the field is neither an array nor a collection, then multiplicity is $[0, 1]$. If the field is an array or a collection, then multiplicity is $[0, +\infty]$. More extensive analyses of source code can narrow multiplicities, for example to infer instance creations. However, such analyses are generally undecidable. Dynamically, multiplicities can be recovered by counting numbers of instances created [17]. However, dynamic analyses possess well-known limitations and thus should be used in addition to static analysis. Recovery of multiplicities is abstract but not precise.

Qualifier. Qualifiers are attributes (or set of attributes) used to identify instances of a classifier at the target-end of an association uniquely. Thus, it is possible to recover qualifiers for associations when these associations are implemented using arrays or collections. Qualifier for an association implemented with an array (or a collection) is an integer value, which distinguishes instances of the association target classifier uniquely. Qualifier for an association implemented with a map[3] reflects the type of the keys used to store and to retrieve instances uniquely. Recovery of qualifiers for maps requires to analyse the use of methods handling contents of maps in source code. Thus, qualifier recovery is abstract but not precise.

Association Class. Association classes are associations with class properties (attributes, methods, operations). The C++, Java, and Smalltalk programming languages do not offer support for association classes. An association class could be recovered by identifying a class `B` which stands in between two classifiers `A` and `C` linked by two associations, between `A` and `B` and between `B` and `C`. However, this recovery is not satisfying as a great number of classes would then qualify to be association classes. Thus, we do not believe possible to recover association classes abstractly and precisely.

N-ary Association. N-ary associations are associations among three or more classifiers. We disagree with Kollmann and Gogolla [17] that only classes at the centre of webs of associations with multiplicities 1 at the target ends are candidates to be n-ary associations. The UML specifications do not constrain multiplicities of classifiers participating in n-ary associations. Thus, we believe difficult to recover n-ary associations from source code abstractly and precisely.

Aggregation. Aggregation are not defined in Part 5 but are often referenced so it is important to detail their recovery. Aggregation relationships can be recovered from source code constructs using their properties [13]. Static analyses of source code can compute values of the invocation site and multiplicity (greater than 1) properties for two given classifiers, from which data to infer the presence of aggregation relationships. Recovery of aggregation relationships is abstract and precise.

Composition. Composition relationships can be recovered from source code after aggregation relationships have been recovered only [13]. Dynamic analyses must be performed to compute the values of exclusivity and lifetime properties of instances of classifiers linked by aggregation relationships. Values of the exclusivity and lifetime properties are used to promote aggregation relationships to composition relationships as required. Thus, composition recovery is abstract but not precise.

Generalisation. The generalisation relationship exist in most object-oriented programming languages as syntactic constructs. However, each programming language gives its own semantics to the generalisation relationship, in particular with respect to static and dynamic binding, co-variance, and contra-variance [4]. We advocate that recovered generalisation relationships should be augmented with data (stereotypes or notes) either on the input programming language or on implementation choices. Thus, generalisation recovery is not abstract but precise.

Dependency. Dependencies indicate semantic relationships between constituents of UML class diagrams. Dependencies add semantics to class diagrams, which does not reflect the structure and–or behaviour of the program being modelled necessarily. Dependencies cannot be recovered from source code only. We believe that there are no means to recover dependencies generically, because recovery of dependencies depends on the semantics given by developers.

[3] A *map* maps keys to values, as template class `map` in C++, interface `Map` in Java, and class `Dictionary` in Smalltalk.

3.4 Miscellaneous

Stereotype Declaration. Stereotypes are user-defined constituents that extend the semantics of UML class diagrams. Stereotypes cannot be recovered from source code automatically because they depend on the designers entirely. However, it is possible to recover some stereotypes when they are well-defined and applied by designers and developers systematically. For example, the authors of the JHotDraw framework [10] documented the design patterns used in their framework using well-defined JavaDoc constructs. This information could be recovered using techniques close to natural language processing and added to the class diagram using stereotypes to show participation of classes to design patterns. Thus, stereotype recovery is abstract but not precise.

Class Pathname. Class pathnames reference classifiers declared in different packages. They can be recovered from source code syntactically. They correspond to fully-qualified classifiers names in C++ and in Java. In Smalltalk, there is no namespace *per se*, thus any class possesses a unique name.

Accessing or Importing a Package. Packages are syntactic constructs of source code written in Java. It is possible to recover accesses and importations syntactically. Packages do not exist as such in source code written with C++ or Smalltalk. In C++, header files using `#define` and `#include` directives could be recovered as packages. For example, in SGI STL library, the `#define STL_H` directive defines a package `STL_H`, the `#include <algo.h>` directive declares an access to the `__SGI_STL_ALGO_H` package. Similarly, namespaces define packages (using the `namespace` and `using namespace` directives). Recovery of access and package import is abstract and precise. In Smalltalk, *Applications* group classes together but do not create namespaces. Cincom VisualWorks for Smalltalk version 5 includes namespaces but there are not part of the language: Recovery is abstract but not precise.

Object. Objects are instances of classifiers. We are interested in UML static class diagrams only, not in UML object diagrams. We do not consider the recovery of objects (their recovery depends on capacities of static and dynamic analyses).

Composite Object. Composite objects are instances of composite classes, *i.e.*, classes with composition relationships to other classifiers. We are interested in UML static class diagrams only, not in UML object diagrams. We do not consider the recovery of composite objects (their recovery depends on the recovery of composition relationships directly).

Link. Links are instances of associations. They define tuple of instances, whose classifiers are linked through associations. We are interested in UML static class diagrams only, not in UML object diagrams. We do not consider the recovery of links (their recovery depends on the recovery of association relationships).

Instance Of. Instantiation relationships exist between a metaclass and classes (classes are *instance of* the metaclass) or between a class and objects (objects are *instance of* the class). A UML static class diagram shows metaclasses and classes, not objects, thus only instantiation relationships among metaclasses and classes should appear. However, metaclasses cannot be recovered from C++ and Java, thus recovery is neither abstract nor precise. Instantiation relationship recovery is possible for Smalltalk source code only, abstractly and precisely.

Derived Element. Derived elements are constituents computed from other constituents of UML class diagrams. They help when designing a program and add no semantics. Derived elements do not exist in source code because they were either eliminated or converted into source code constructs during implementation. There are no means to recover derived elements from source code either abstractly and precisely.

List Compartment. List compartments describe the visual organisation and display of data recovered from source code. Thus, list compartments constraint the layout of data recovered from source code. The recovery of list compartments from source code depend on the recovery of their contents completely.

Name Compartment. Name compartments are containers for data related to classes and interfaces only: Names, stereotypes, tagged values. Thus, the recovery of name compartments from source code depend on the recovery of their contents entirely.

4 Implementation and Comparison

From the previous study of the recovery of UML class diagrams constituents in C++, Java, and Smalltalk source code, we implement Ptidej, a tool suite for the abstract and precise recovery of class diagrams from Java source code. Figure 4 shows the UML class diagrams recovered from Source codes 1 automatically. The UML class diagram is abstract and precise with respect to the Java program implementation. It shows the two classes `Example1` and `A`, their sets of methods, and existing relationships:

- The aggregation relationship between `Example1` and `A`: Arrow with white diamond head.

Constituents	Abstract	Precise	Tools[4]	Constituents	Abstract	Precise	Tools[4]
Attribute	✓	✓	All	Composition	✓	×	IDEA, PTIDEJ
Method	✓	✓	All	Association Class	×	×	IDEA
Operation	✓	✓	PTIDEJ	N-ary Association	×	×	IDEA
Class	✓	✓	All	Multiplicity	✓	×	IDEA, PTIDEJ
Nested Class	×	✓	IDEA, PTIDEJ	Qualifier	✓	×	IDEA
Type	✓	✓	PTIDEJ	Generalisation	×	✓	All
Implementation Class	✓	✓	PTIDEJ	Dependency	×	×	None
Interface	✓	✓ ($\times_{Stk}$)	All	Link	N/A	N/A	N/A
Parameterised Class	✓ ($\times_{Stk}$)	✓ ($\times_{Stk}$)	None	Object	N/A	N/A	N/A
Bound Element	✓ ($\times_{Stk}$)	✓ ($\times_{Stk}$)	None	Composite Object	N/A	N/A	N/A
Metaclass	✓	× ($\checkmark_{Stk}$)	None	Derived Element	×	×	None
Powertype	×	×	None	Instance Of	$\checkmark_{Stk}$	$\checkmark_{Stk}$	None
Data type	✓ ($\times_{Stk}$)	✓ ($\times_{Stk}$)	PTIDEJ	Name Compartment	N/A	N/A	N/A
Enumeration	✓	✓	PTIDEJ	List Compartment	N/A	N/A	N/A
Utility Class	✓	✓	PTIDEJ	Class Pathname	✓	✓	All
Binary Association	✓	✓	IDEA, PTIDEJ, WOMBLE	Accessing or Importing Package	✓	✓	PTIDEJ, ROSE, TOGETHER/J
Aggregation	✓	✓	IDEA, PTIDEJ, WOMBLE	Stereotype Declaration	✓	×	None

Table 2. Summary of the recovery of UML class diagram constituents[4]. (✓ = Yes, × = No)

- The binary association relationship resulting from the instantiation of `A` in `Example1`: "`-*--> ptidej.example.apsec.A`".
- The binary association relationship resulting from the use of `A` as parameter of methods of `Example1`: "`-u--> ptidej.example.apsec.A`".
- The aggregation relationship between `Example1` and `A` textually: "`<>-->* ptidej.example.apsec.A`". (It emphasises that the aggregation relationship has cardinality zero or more using the "`*`" symbol.)
- The binary association relationship resulting from the instantiation of `Example1` within itself: "`-*--> ptidej.example.apsec.Example1`".
- The binary association relationship resulting from the use of strings in `Example1`: "`-u--> java.lang.String`".

A similar class diagram is produced when reverse engineering the alternate implementation of the Java program implementation in Source code 2.

We applied our tool suite on many well-known programs. For the sake of brevity, we cannot detail here the recovered class diagrams. However, we found that our tool suite indeed produces class diagrams that are useful to maintainers. For example, the class diagram recovered from the JHOTDRAW framework contains 89% more pieces of information than the one from its documentation, and 67% more than the class diagram recovered using Rational ROSE[5].

Table 2 compares our tools suite with other existing reverse engineering tools. It summarises all the

[4]Subsection 2.3 details the set of tools, Stk means Smalltalk.

[5]We relate the numbers of constituents provided by the documentation and by Rational ROSE with the number of constituents recovered by our tool suite.

Recovered UML constituents (in %)	
ARGOUML	21
CHAVA	21
FUJABA	21
IDEA	48
PTIDEJ	62
ROSE	24
TOGETHER/J	24
WOMBLE	28

Table 3. Percentage of recovered UML class diagrams constituents per tool.

UML class diagrams constituents, indicates if their recovery is possible abstractly and precisely, and shows which existing reverse-engineering tools recover these constituents. Table 3 presents the percentage of recovered UML constituents per tools and emphasises that:

- Existing revere-engineering tools recover a small subset of all UML class diagram constituents only.
- Different programming languages require different recovery techniques. Our study provides concrete evidence to support this well-known point.
- UML provides a rich set of constituents to describe the architecture, the design, and the implementation of programs. These constituents require a greater attention regarding their definitions.

Also, Table 3 shows that it is possible to offer abstract and precise recovery of UML class diagrams with a thorough study of class diagrams constituents.

5 Related Work

Many studies exist that assess characteristics of reverse-engineering tools and of tools by-products. Bellay and Gall [2] proposed a comparison of four reverse engineering tools: REFINE/C, IMAGIX 4D, RIGI,

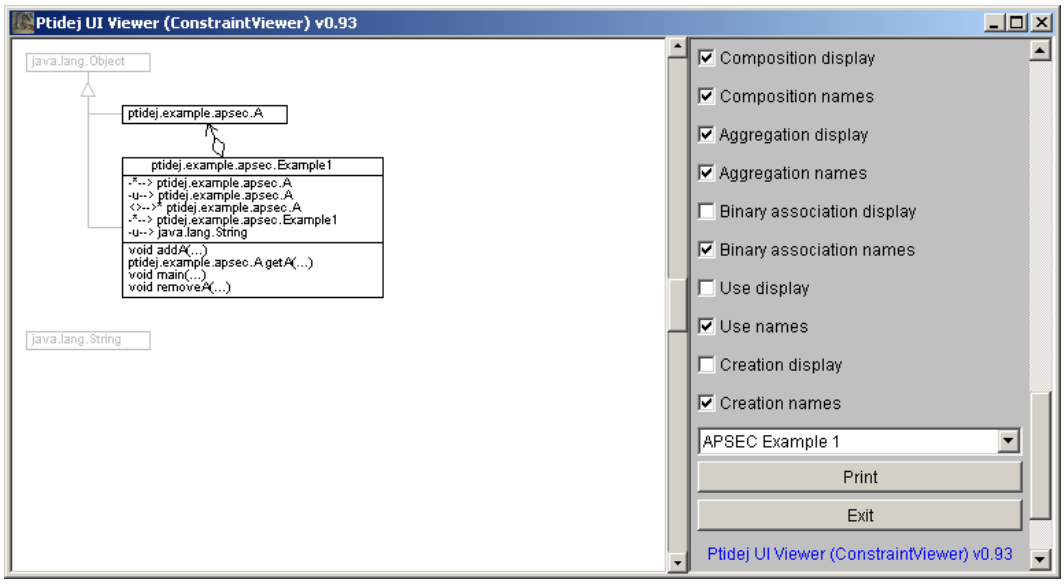

Figure 4. Class diagram recovered from Source code 1 using PTIDEJ**.**

and SNIFF+ to evaluate their applicability to embedded software, their usability, and their extensibility. They introduced four functional categories to assess the tools: Parsing capabilities, by-products representation, editing and browsing capabilities, and general capabilities. The authors performed a thorough assessment of four tools with respect to their *functional* capabilities. However, they did not focus on the tools by-products particularly. Moreover, the tools assessed do not attempt to produce UML class diagrams from source code. Thus, we cannot use the criteria and results of this study for recovered UML class diagrams.

Gannod and Cheng [12] developed a framework to classify and to compare reverse engineering techniques. They classified tools according to the techniques used to reverse engineer programs and according to four semantic dimensions of the tools by-products. The four semantic dimensions are: Semantic distance (levels of abstraction between source code and by-products), semantic accuracy (confidence that a by-product is correct with respect to source code), semantic precision (degree of formality of by-products), and semantic traceability (degree to which by-products can be used to build a program). Then, the authors studied many different reverse-engineering tools. However, none of the assessed tools attempt to produce *UML class diagrams* from source code.

Storey, Fracchia, and Müller [24] studied cognitive issues related to the design of software visualisation tools. They proposed a taxonomy of fourteen cognitive design elements that support the construction of a mental model by maintainers to ease program understanding. Tool builders should integrate these cognitive design elements when designing software visualisation tools to overcome common deficiencies. Cognitive design elements relate to program understanding (bottom-up and top-down) and to maintainer's cognitive overhead (navigability). The authors provided criteria to evaluate the *ease of use* of the by-products: How to enhance understanding, how to ease navigation. Moreover, They did not assess characteristics nor emphasise on reverse engineering tools producing UML class diagrams.

Bassil and Keller [1] reported on a survey and on an analysis of software visualisation tools. More than one hundred participants filled a questionnaire including twenty-one questions. The questionnaire aimed at providing quantitative data on several characteristics of software visualisation tools: Functional, practical, cognitive, and code analysis aspects. The questionnaire surveyed the usefulness of thirty-four functional aspects and of thirteen practical aspects. Among functional aspects were search capabilities, source code visualisation, and hierarchical representations. This study provided important quantitative data on many aspects of software visualisation tools. This data helped to assess the usefulness of functional and practical features of software visualisation and reverse engineering tools. However, this survey focused on characteristics of the *software visualisations tools* rather than on characteristics of their by-products.

Kollmann *et al.* [18] recognised that the reverse engineering capabilities of CASE tools are limited because of the lack of one-to-one mapping between

class diagrams constituents and source code constructs. They introduced a case study of four CASE tools (Rational Rose, Borland TogetherJ, IDEA, and Fujaba) on a Java program and compared the tools using properties of the recovered class diagrams and of the reverse engineering algorithms: Number of classes and of associations, handling of associations, of interfaces, of collection classes, of multiplicities, of role names, of inner classes, and of compartment details. They compared tools by-products using a common research platform automatically also. They concluded on the capacities of current CASE tools to recover but the simplest constituents of class diagrams. However, they did not assess *systematically* the feasibility to recover class diagrams from source code.

6 Conclusion and Future Work

We define two characteristics to qualify recovered UML class diagrams: Abstractness and preciseness, inspired by the work of Gannod and Cheng. These two characteristics are important to assess the usefulness of recovered class diagrams for maintainers. We show that existing mainstream reverse-engineering tools produce neither abstract nor precise class diagrams from source code. We advocate that the limitations of tools come from a lack of systematic study of the UML class diagram constituents. We perform an exhaustive study of these constituents with respect to their abstract and precise recovery from C++, Java, and Smalltalk source code constructs, which depends on programming languages and on the constituents. We implement algorithms to recover class diagrams abstractly and precisely from Java source code in our tool suite, Ptidej. We show the usefulness of the recovered class diagrams using a case study. Future work includes refining the characteristics of recovered class diagrams, studying the new 2.0 version of the UML specifications, and improving our algorithms. Also, we plan to study alternative techniques (natural language processing) and other programming languages, such as Eiffel.

References

[1] S. Bassil and R. K. Keller. Software visualization tools: Survey and analysis. In *proceedings of the 9^{th} International Workshop on Program Comprehension*, pages 7–17. IEEE Computer Society Press, May 2001.

[2] B. Bellay and H. Gall. A comparison of four reverse engineering tools. In *proceedings of the 4^{th} Working Conference on Reverse Engineering*, pages 2–11. IEEE Computer Society Press, Oct. 1997.

[3] F. Bergenti and A. Poggi. IDEA: A design assistant based on automatic design pattern detection. In *proceedings of the 12^{th} conference on Software Engineering and Knowledge Engineering*, pages 336–343. Springer-Verlag, Jul. 2000.

[4] A. Beugnard. OO languages late-binding signature. In *proceedings of the 9^{th} workshop on Foundations of Object-Oriented Languages*. ACM Press, Jan. 2002.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, Oct. 1999.

[6] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *proceedings of the 11^{th} European Conference for Object-Oriented Programming*, pages 344–366. Springer-Verlag, Jun. 1997.

[7] J.-M. Bruel, B. Henderson-Sellers, F. Barbier, A. L. Parc, and R. B. France. Improving the UML metamodel to rigorously specify aggregation and composition. In *proceedings of the 7^{th} international conference on Object-Oriented Information Systems*, pages 5–14. Springer-Verlag, Aug. 2001.

[8] F. Civello. Roles for composite objects in object-oriented analysis and design. In *proceedings of the 8^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 376–393. ACM Press, Sep. 1993.

[9] R. B. France. A problem-oriented analysis of basic UML static requirements modeling concepts. In *proceedings of the 14^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 57–69. ACM Press, Nov. 1999.

[10] E. Gamma and T. Eggenschwiler. JHotDraw. Web site, 1998.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] G. C. Gannod and B. H. C. Cheng. A framework for classifying and comparing software reverse engineering and design recovery techniques. In *proceedings of the 6^{th} Working Conference on Reverse Engineering*, pages 77–88. IEEE Computer Society Press, Oct. 1999.

[13] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *proceedings of the 19^{th} conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Oct. 2004. To appear.

[14] T. Hartmann, R. Jungclaus, and G. Saake. Aggregation in a behavior oriented object model. In *proceedings of 6^{th} European Conference for Object-Oriented Programming*, pages 57–77. Springer-Verlag, Jun.–Jul. 1992.

[15] B. Henderson-Sellers and F. Barbier. A survey of the UML's aggregation and composition relationships. In *L'objet : Logiciel, Base de données, Réseaux*, 5(3/4):339–366, Dec. 1999.

[16] D. Jackson and A. Waingold. Lightweight extraction of object models from bytecode. In *proceedings of the 21^{st} International Conference on Software Engineering*, pages 194–202. ACM Press, May 1999.

[17] R. Kollmann and M. Gogolla. Application of UML associations and their adornments in design recovery. In *proceedings of the 8^{th} Working Conference on Reverse Engineering*, pages 81–91. IEEE Computer Society Press, Oct. 2001.

[18] R. Kollmann, P. Selonen, E. Stroulia, T. Systä, and A. Zündorf. A study on the current state of the art in tool-supported UML-based static reverse engineering. In *proceedings of the 9^{th} Working Conference on Reverse Engineering*, pages 22–33. IEEE Computer Society Press, Oct. 2002.

[19] J. Korn, Y.-F. Chen, and E. Koutsofios. Chava: Reverse engineering and tracking of Java applets. In *proceedings of the 6^{th} Working Conference on Reverse Engineering*, pages 314–325. IEEE Computer Society Press, Nov. 1999.

[20] J. Martin and J. J. Odell. *Object-Oriented Methods: A Foundation, UML Edition*. Prentice Hall, Dec. 1997.

[21] J. Niere, J. P. Wadsack, and A. Zündorf. Recovering UML diagrams from Java code using patterns. In *proceedings of the 2^{nd} workshop on Soft Computing Applied to Software Engineering*, pages 89–97. Springer-Verlag, Feb. 2001.

[22] Object Management Group, Inc. *UML v1.5 Specification*, Mar. 2003.

[23] M. Saksena, R. B. France, and M. M. Larrondo-Petrie. A characterization of aggregation. In *proceedings of the 5^{th} international conference on Object-Oriented Information Systems*, pages 363–372. Springer-Verlag, Sep. 1998.

[24] M.-A. D. Storey, F. D. Fracchia, and H. A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. In *Journal of Systems and Software*, 44(3):171–185, Jan. 1999.