This paper has been accepted at CASCON 2004.

A Reverse Engineering Tool for Precise Class Diagrams

Yann-Gaël Guéhéneuc

Département d'informatique et de recherche opérationnelle
Université de Montréal – CP 6128 succ. Centre Ville
Montréal, Québec, H3C 3J7 – Canada
guehene@iro.umontreal.ca

Abstract

Developers use class diagrams to describe the architecture of their programs intensively. Class diagrams represent the structure and global behaviour of programs. They show the programs classes and interfaces and their relationships of inheritance, instantiation, use, association, aggregation and composition. Class diagrams could provide useful data during programs maintenance. However, they often are obsolete and imprecise: They do not reflect the *real* implementation and behaviour of programs. We propose a reverse-engineering tool suite, PTIDEJ, to build precise class diagrams from Java programs, with respect to their implementation and behaviour. We describe static and dynamic models of Java programs and algorithms to analyse these models and to build class diagrams. In particular, we detail algorithms to infer use, association, aggregation, and composition relationships, because these relationships do not have precise definitions. We show that class diagrams obtained semi-automatically are similar to those obtained manually and more precise than those provided usually.

This work has been partly funded by IBM OTI Labs – 2670 Queensview Drive – Ottawa, Ontario, K2B 8K1 – Canada.

1 Introduction

Software developers use UML-like class diagrams to describe the architecture of object-oriented programs intensively during development. Class diagrams represent the structure and global behaviour of programs [13], showing classes, interfaces, and their relationships [16]. They help software developers by abstracting implementation details and by presenting an easier-to-grasp clustered view of the programs lines of code [17].

Class diagrams would help software maintainers to understand programs architecture and to locate places requiring modifications during maintenance. However, they are often obsolete—unsynchronised with the *concrete* implementation of programs—when existing at all [6].

Software maintainers need tools to recover class diagrams from programs source code and binaries, which are the only sources of data available usually during the maintenance process and which can be used to build both static and dynamic models of programs.

We present PTIDEJ (*Pattern Trace Identification, Detection, and Enhancement in Java*), a reverse engineering tool suite to build class diagrams from static and dynamic models of Java programs semi-automatically. This paper summarises our previous work on program architecture recovery started in 2000 at École des

Mines de Nantes and being pursued at University of Montréal. The main contribution of this paper is an overview of our tool suite and a complete example of static and dynamic analyses, using the JHotDraw program, which exemplifies the need for precise architecture recovery for maintainers.

By precise architecture recovery, we mean that the recovered architecture reflects the implementation of the analysed programs and that it provides the same data as if recovered by maintainers manually, in particular with respect to usual UML constituents: Classes, interfaces, inheritance, instantiation, use, association, aggregation, and composition relationships.

In Section 2, we present related work briefly and discuss their limitations. Then, we introduce the Ptidej tool suite: Its models and tools. We also sketch the use of the tool suite. In Section 3, we detail the definitions and algorithms to identify relationships among classes, interfaces, and their instances, and discuss their precision and recall. In Section 4, we apply our tool suite on the real-world JHotDraw program. We show that the class diagram obtained semi-automatically is similar to one obtained manually yet more precise than this provided. Finally, in Section 5, we conclude and present future work on and with the Ptidej tool suite.

2 Tool Suite

There exists several reverse engineering tools to recover class diagrams from program implementation. We present three typical tools briefly, discuss their limitations, and summarise the contributions of our tool suite. Then, we detail our models and tools, and present a short example.

2.1 Related Work

Chava. Korn *et al.* propose Chava [15], a reverse engineering tool dedicated to Java applets. Chava creates a repository that contains the structure of a program from the source code or class files of the program. A repository stores entities that represent classes, interfaces, packages, files, methods, fields, and referenced strings. Entities are linked with subclass, implementation, declaration, field access/modification, and method call relationships. A repository contains a model of the program, which reproduces *exactly* a program static model and does not increase its precision with respect to other kinds of relationships, such as use, association, aggregation, or composition, or with dynamic data.

Womble. Jackson and Waingold propose Womble [13], a tool for the lightweight extraction of object models, which are similar to class diagrams. The latest version of Womble is able to analyse programs class files and to identify inheritance, use, and association relationships. It proposes heuristics to infer multiplicities of the origin and target classes, which distinguish association and aggregation. However, the authors do not attempt to identify composition relationships to increase the level of precision of the recovered object models.

CASE Tools. CASE tools, such as ArgoUML and Rational Rose, offer reverse engineering capabilities, but their capabilities are very limited. They only distinguish use, association, aggregation, and composition relationships graphically. Indeed, they use identical algorithms to reverse engineer use, association, aggregation, and composition relationships, which leads to inconsistency with programs implementations [10].

Discussion. Existing reverse engineering tools are only capable of identifying *structural* relationships, existing *physically* in the static models of Java programs. They are incapable of *abstracting* relationships, which must be *inferred* both from static and dynamic models of programs. In particular, they lack precise definitions and algorithms to identify (or to distinguish) use, association, aggregation, and composition relationships.

The Ptidej tool suite is different from existing reverse engineering tools because it uses both static and dynamic data to infer relationships among classes and interfaces. It is able to infer inheritance, instantiation, use, association, aggregation, and composition relationships among classes and interfaces to represent *precisely* programs. Thus, it helps software maintainers to grasp and to understand programs architecture.

The recovery of programs class diagrams decomposes in two steps: First, class diagrams are built from static data provided by Java programs class files; Second, class diagrams are refined with dynamic data obtained by analysing the runtime behaviour of the programs.

2.2 Models

We use three different models to represent and to analyse static and dynamic data about Java programs and to describe class diagrams.

Static Model. We use class files composing Java programs as static models. Class files embody all the data provided by software developers about a program architecture and about its runtime behaviour statically. They are easier to manipulate than source code, using specialised tools such as CFParse [8] and Javassist [4], because of their structure and of the processing performed at compilation-time, in particular type binding. Also, class files are always available, whereas source code is not.

Dynamic Model. We use traces as models of the runtime behaviour of Java programs. A trace is a history of execution events: Field accesses/modifications; Class loads/unloads; Method, constructor, and finalizer entries/exits; Program end [11]. A program has one and only one static model but several (possibly an infinity of) dynamic models. Thus, dynamic models *approximate* programs behaviour only.

Class Diagram Model. We develop a meta-model, PADL (*Pattern and Abstract-level Description Language*) [2], to describe programs as class diagrams. PADL offers constituents, such as `Model`, `Class`, `Method`, `Relationship`, with which we can build class diagrams representing programs. It offer also methods to manipulate class diagrams easily and to generate other representations of class diagrams, using the Visitor design pattern.

2.3 Tools

The Ptidej tool suite decomposes into three tools[1]: To analyse static models; To generate and to analyse dynamic models; To build class diagrams from the analyses.

PADL ClassFile Creator. Different algorithms to analyse static models can be connected to the PADL meta-model, using the Bridge design pattern, to create class diagrams from different sources of data. We offer a default implementation of such a creator, PADL ClassFile Creator, that analyses static models of Java programs and create the corresponding class diagrams by instantiating the constituents of the meta-model.

For each constituent `C` of the PADL meta-model, the PADL ClassFile Creator declares a `recognizeC()` method used to identify constructs corresponding to `C` in static models of Java programs and to instantiate `C` with the appropriate data from the constructs. Methods `recognizeCPriority()` order the identifications of constituents. (Creators for AOL files [3] and C++ files exist also.)

Caffeine. We develop a tool for the dynamic analysis of Java programs. Caffeine [11] is a 100%-pure Java program that generates and analyses on the fly dynamic models of Java programs. Analyses of dynamic models are performed with Prolog predicates. We use Prolog because of its unification and backtrack mechanisms and its high-level pattern-matching capabilities.

A Prolog engine runs as a co-routine of the Java program under analysis. It controls the program execution with the `nextEvent/3` predicate. The `nextEvent/3` predicate unifies a Prolog variable with the last event generated by the program, according to filters and to a list of expected events. Then, dedicated predicates may analyse the set of events.

We develop two Prolog predicates to analyse dynamic models and to assess the presence of composition relationships among classes, interfaces, and their instances: `instanceLevelCompositions/1` and

[1]All tools are available at:
`www.yann-gael.gueheneuc.net/Work/`

`classLevelCompositions/1`. We use the results of these analyses to refine class diagrams of Java programs with dynamic data on relationships among its classes and interfaces.

Ptidej. The Ptidej tool is a front-end for PADL, PADL ClassFile Creator, and Caffeine. We implement this front-end both as a stand-alone 100%-pure Java program and as a plug-in for the Eclipse development environment for Java.

First, a maintainer selects a program static model, for example Java class files. The front-end calls the appropriate creator, for example the PADL ClassFile Creator, to build the corresponding class diagram, using the PADL meta-model. Then, the maintainer refines the class diagram by loading results from the dynamic analysis of the program with the Caffeine tool. Figure 1 summarises the data flow among tools. Thus, a maintainer builds semi-automatically (with user-interactions) a class diagram representing the *concrete* implementation of a program. The front-end displays class diagrams with a dedicated graphic library and different layout algorithms, using the Strategy design pattern.

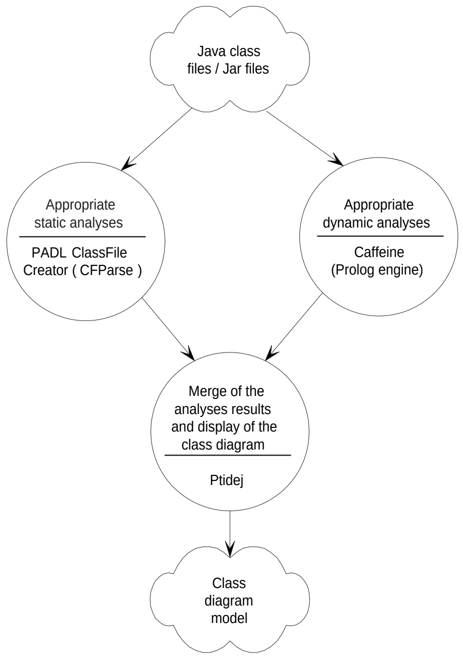

Java class files / Jar files
Appropriate static analyses
PADL ClassFile Creator (CFParse)
Appropriate dynamic analyses
Caffeine (Prolog engine)
Merge of the analyses results and display of the class diagram
Ptidej
Class diagram model

Figure 1: Flow of the data among tools

2.4 Example

We exemplify the Ptidej tool suite with a simple document description program. We present a complete example in Section 4. Figure 2(a) shows the stand-alone Ptidej front-end. It consists of two panels to display class diagrams (left) and to control the tool (right). In addition to classes and interfaces, we can display the names and graphical representations of the relationships among classes, interfaces, and their instances (checkboxes on the right).

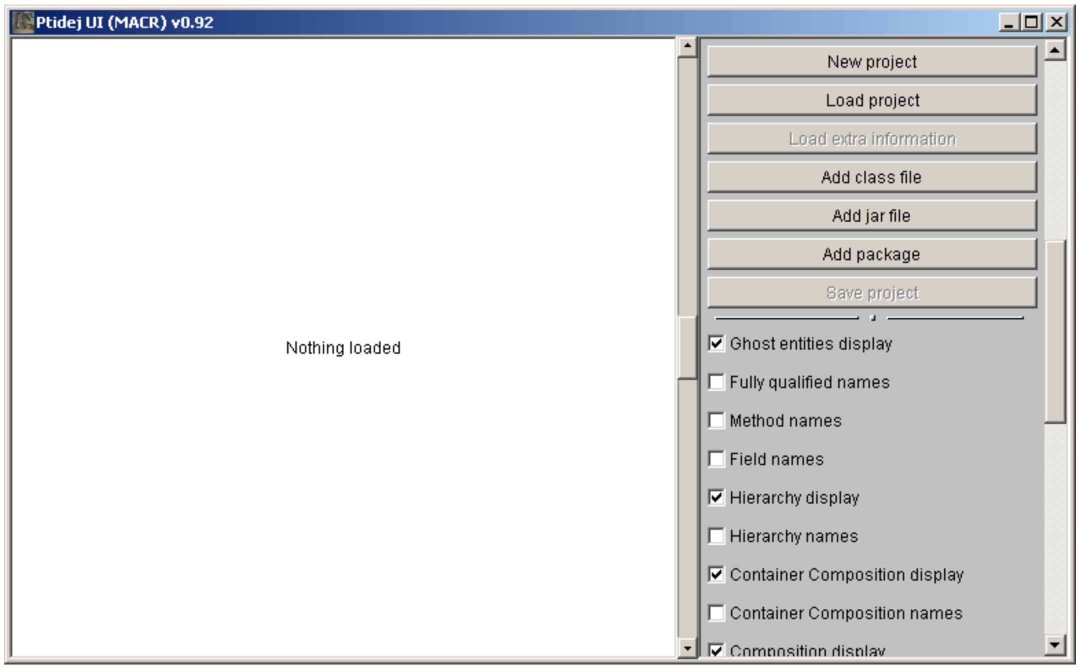
Figure 2(b) shows the class diagram displayed for the document description program, with the classes and interfaces declared in the selected class files (in black) and those that are only known through references[2] (in gray). We only display graphical representations of inheritance, aggregation, and composition relationships, and names of use, association, and instantiation relationships. This class diagram is built by the PADL ClassFile Creator tool and laid out with a simple layout algorithm, which minimises the crossing of inheritance representations. It shows an aggregation relationship (white triangle) between classes `Document` and `Element`, which represent a document and its structure respectively.

Figure 3(a) shows the Caffeine tool to analyse the document description program dynamically. This tool runs as a co-routine of the program and analyses generated events. Figure 3(b) shows the output generated by the tool after analysing an execution of the document description program. The output shows that a composition relationship exists between classes `Document` and `Element`, which is more precise than the aggregation relationship found by static analysis. Figure 3(c) shows the class diagram refined with the results from the dynamic analysis: A composition relationship (black triangle) replaces the aggregation relationship.

3 Relationships

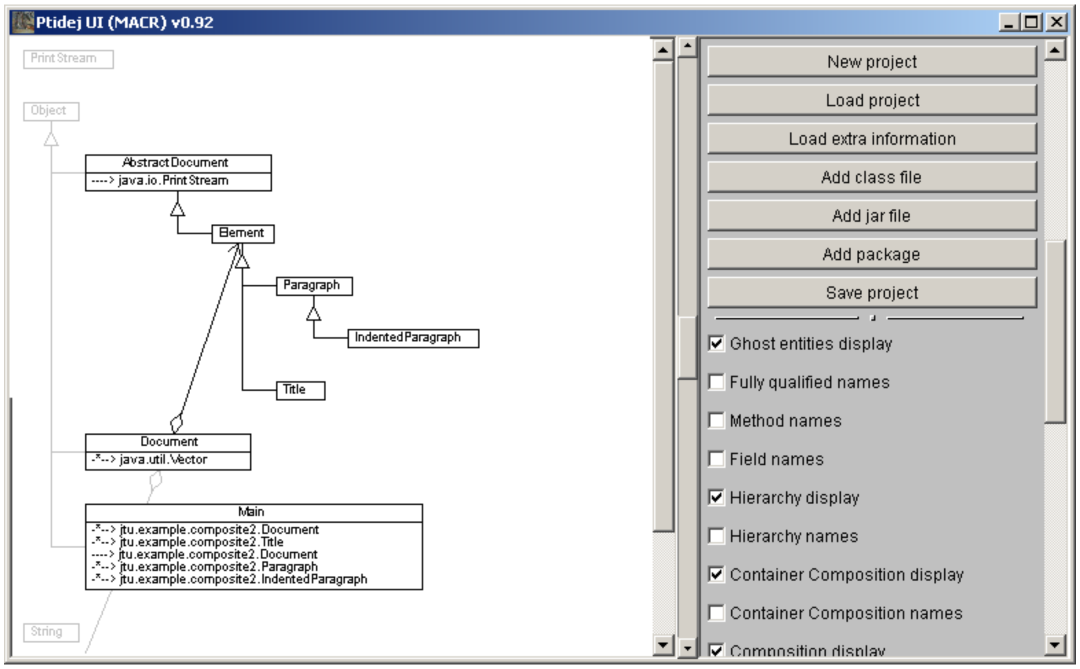
We now detail the definitions and algorithms that we use to identify the inheritance, instan-

[2]We call *ghosts* classes and interfaces known only by references from analysed classes and interfaces: They surround analysed classes and interfaces but we do not know much about them.



[✓] Ghost entities display
[ ] Fully qualified names
[ ] Method names
[ ] Field names
[✓] Hierarchy display
[ ] Hierarchy names
[✓] Container Composition display
[ ] Container Composition names
[✓] Composition display

(a) The Ptidej front-end.



[✓] Ghost entities display
[ ] Fully qualified names
[ ] Method names
[ ] Field names
[✓] Hierarchy display
[ ] Hierarchy names
[✓] Container Composition display
[ ] Container Composition names
[✓] Composition display

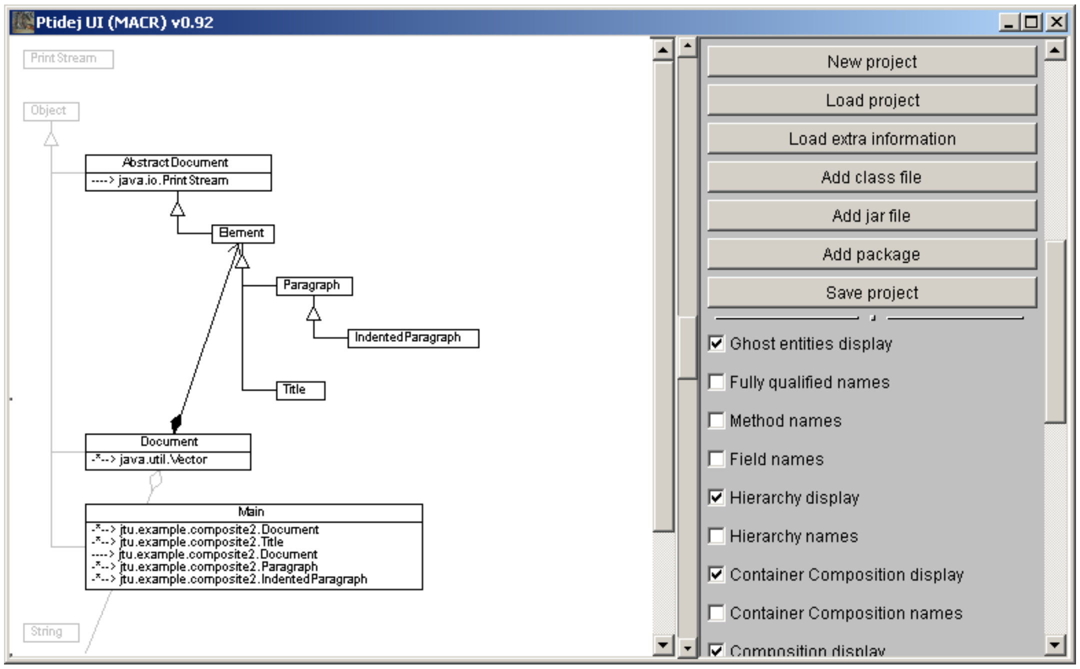
(b) The Ptidej front-end showing the class diagram obtained from the PADL ClassFile Creator tool.

Figure 2: Use of the Ptidej tool suite on a document description program

(a) The CAFFEINE tool.

```
Caffeine initialized.
Caffeine started.
(Remote JVM)
(Remote JVM) ...
(Remote JVM)
List of instance−level compositions:
    composition(
        jtu.example.composite2.Document, 1006,
        jtu.example.composite2.Element, 1011,
        true)
List of class−level compositions:
    composition(
        jtu.example.composite2.Document,
        jtu.example.composite2.Element,
        true)
```

(b) Data obtained from the CAFFEINE tool when analysing the program dynamically.



[✓] Ghost entities display
[ ] Fully qualified names
[ ] Method names
[ ] Field names
[✓] Hierarchy display
[ ] Hierarchy names
[✓] Container Composition display
[ ] Container Composition names
[✓] Composition display

(c) The PTIDEJ tool displaying the class diagram refined with dynamic data.

Figure 3: Use of the PTIDEJ tool suite on a document description program (cont'd)

tiation, use, association, aggregation, and composition relationships. The recovery of classes and interfaces does not pose any problem because classes and interfaces exist in static models explicitly.

3.1 Inheritance, Instantiation

The inheritance and instantiation relationships are direct to identify in programs static models because they exist in source code or class files *physically.*

Inheritance. In a static model of a program, classes and interfaces declare the classes and interfaces they implement or extend explicitly. An algorithm to infer inheritance relationships needs only to iterate over classes and interfaces and to retrieve their subclasses/interfaces syntactically.

Instantiation. Static initialisers, instance initialisers (constructors), and methods may contain objects instantiations. An algorithm to infer instantiation relationships needs only to iterate over the byte-codes of each initialisers and methods, looking for `New`, `NewArray`, `ANewArray`, `MultiANewArray` byte-codes.

3.2 Use, Association, Aggregation, Composition

The use, association, aggregation, and composition relationships are difficult to identify because they lack precise definitions. They do not appear in programs static models *explicitly* and they require the use of dynamic models.

From the literature, we propose consensual definitions of these relationships [10], which decompose in four properties: Exclusivity, invocation site, lifetime, and multiplicity. The four properties are minimal and we use these to develop algorithms to identify the relationships.

Definitions. Links among classes, interfaces, and their instances exist at runtime to allow method invocations and data access. These links are described by different relationships in class diagrams. Conceptually, as in the UML, a relationship is non-oriented and may exist among two or more classes (or interfaces). Practically, however, most authors agree (see [10]) that use, association, aggregation, and composition relationships involve the instances of two classes, an origin and a target, respectively `A` and `B`, and that these relationships are oriented, irreflexive, anti-symmetric at instance and class level, and asymmetric at instance level [12].

First, we propose that an association between `A` and `B` defines the ability of an instance of `A` to send a message to an instance of `B`. Nothing prevents other relationships to exist between classes `B` and `A`.

Second, we say that an association between `A` and `B` is an aggregation relationship if the definition of `A`, the whole, contains instances of `B`, the part. The whole must define a field (or an array field, or field of type collection) of the type of its part. Instances of the whole send messages to the instances of the part. Subclasses inherit the aggregation relationship between `A` and `B`, because subclasses inherit the structure and behaviour of their superclasses.

Third, we define a composition as an aggregation with constraints on the lifetimes of the whole and of the parts and on the ownership of the parts. An instance of the whole owns the instances of its part. The instances of the part are exclusive to the instance of their whole. Parts can be exchanged during the lifecycle of the whole, but all the parts owned by a whole at the moment of its destruction are also destroyed. A composition relationship only allows an association relationship between its part and whole, to ensure the exclusivity and lifetime properties.

Finally, the use relationship is the *default* relationship between two classes when these are not linked through an association, an aggregation, or a composition relationship. For example, a use relationship exists between two interfaces `IA` and `IB` if interface `IA` defines methods which signatures use interface `IB`. The use relationship is similar to the association relationships but only *suggests* that messages may be sent. We do not further consider use relationships in the rest of this section, because their recovery is fairly easy from source code (method parameters, return types. . .).

Table 1 summarises the possible links among classes, interfaces, and their instances, and their representations as relationships. For example, Table 1 states (third row, on the right) that an aggregation relationship exists between two classes `A` and `B` if two instances of these classes, respectively `a` and `b`, are linked together such as `a` sends messages to `b` and `b` is a field of `A` (on the left).

Properties. The definitions of the relationships use four language-independent properties. The association relationship allows *multiple* instances of `A` and `B` to take part in the relationship, while the aggregation and composition relationships allow multiple instances of `B` to be in a relationship with one instance of `A`. With an aggregation relationship, instances of `A` access instances of `B` through a particular *invocation site*: Field, array field, or field of type collection. With a composition relationship, instances of `B` are *exclusive* to their corresponding instance of `A` and instances of `A` and `B` have related *lifetimes*.

The *exclusivity* property states whether an instance of a class involved in a relationship can be in another relationship at a given time.

$$EX(\mathtt{A},\mathtt{B}) \in \{true, false\}$$

We name $\mathbb{B}$ the set $\{true, false\}$. The value *true* states that an instance of `B` can take part in another relationship with another instance of `A` or of another class. The value $false$ indicates that it cannot. The exclusivity property holds at a given time only. It does not prevent possible exchanges.

The *invocation site* property indicates that instances of `A`, involved in a relationship, send messages to instances of `B`.

$$IS(\mathtt{A},\mathtt{B}) \subseteq \{\mathtt{field}, \mathtt{array\ field}, \\ \mathtt{collection}, \mathtt{parameter}, \mathtt{local\ variable}\}$$

The values of the IS property summarise possible invocation sites for messages sent from instances of `A` to instances of `B`. There can be no message sent from `A` to `B`: $IS(\mathtt{A},\mathtt{B}) = \emptyset$, or messages can be sent from `A` through a {`field`} of type `B`, an {`array field`}, a field of type {`collection`}, a method {`parameter`}, or a method {`local variable`}. We name *yes* the set {`field`, `array field`, `collection`, `parameter`, `local variable`}.

The *lifetime* property constrains the lifetime of all the instances of `B` with respect to the lifetime of all the instances of `A`. It corresponds to the time elapsed between the times of destruction LT_d of two instances of `A` and `B` [5]. The time is in any convenient unit, for example in seconds or in CPU ticks.

In programming languages with garbage collection, LT_d matches the moment where an instance is ready for garbage collection.

$$\begin{aligned} LT(\mathtt{A},\mathtt{B}) &= LT_d(A) - LT_d(B) \\ &\in \{+, -\} \end{aligned}$$

We name $\|$ the set $\{+, -\}$. $LT(\mathtt{A},\mathtt{B}) = +$ if instances of `B` are destroyed before the corresponding instances of `A`, $LT(\mathtt{A},\mathtt{B}) = -$ if destroyed after, and $LT(\mathtt{A},\mathtt{B}) \in \|$ if their times of destruction are unrelated (either $+$ or $-$).

The *multiplicity* property describes the number of instances of `B` allowed in a relationship with `A`.

$$MU(\mathtt{A},\mathtt{B}) \subset \mathbb{N} \cup \{+\infty\}$$

For the sake of simplicity, we use an interval of the minimum and maximum numbers of instances to represent the multiplicity. We consider multiplicity at the *target* end of a relationship only. The interested reader may refer to [13] for a discussion on multiplicities at both ends of a relationship.

Formalisation. We now formalise the definitions of the relationships with the four properties to build identification algorithms. We define an association relationship between `A` and `B`, $AS(\mathtt{A},\mathtt{B})$, as:

$$\begin{array}{rcll} AS(\mathtt{A},\mathtt{B}) = & & & \\ (IS(\mathtt{A},\mathtt{B}) \subseteq yes) & \wedge & (IS(\mathtt{B},\mathtt{A}) = \emptyset) & \wedge \\ (EX(\mathtt{A},\mathtt{B}) \in \mathbb{B}) & \wedge & (EX(\mathtt{B},\mathtt{A}) \in \mathbb{B}) & \wedge \\ (LT(\mathtt{A},\mathtt{B}) \in \|) & \wedge & (LT(\mathtt{B},\mathtt{A}) \in \|) & \wedge \\ (MU(\mathtt{A},\mathtt{B}) & = & [0,+\infty]) & \wedge \\ (MU(\mathtt{B},\mathtt{A}) & = & [0,+\infty]) & \end{array}$$

Link			Is described by	Relationship		
Origin	*Means*	*Target*		*Origin*	*Name*	*Target*
Class/Interface	Any	Class/Interface		Class/Interface	Use	Class/Interface
Instance	Direct	Instance		Class/Interface	Association	Class/Interface
Instance	Field	Instance		Class	Aggregation	Class/Interface
Instance	Field + Lifetime property	Instance		Class	Composition	Class/Interface

Table 1: Definitions and applicability of the relationships

We define an aggregation relationship between A and B, $AG(\mathtt{A},\mathtt{B})$, as:

$$
\begin{array}{l}
AG(\mathtt{A},\mathtt{B}) = \\
\begin{array}{rcll}
(IS(\mathtt{A},\mathtt{B}) & \subseteq & \{\mathtt{field}, \mathtt{array\ field}, \mathtt{collection}\}) & \wedge \\
(IS(\mathtt{B},\mathtt{A}) & = & \emptyset) & \wedge \\
(EX(\mathtt{A},\mathtt{B}) \in \mathbb{B}) & \wedge & (EX(\mathtt{B},\mathtt{A}) \in \mathbb{B}) & \wedge \\
(LT(\mathtt{A},\mathtt{B}) \in \|) & \wedge & (LT(\mathtt{B},\mathtt{A}) \in \|) & \wedge \\
(MU(\mathtt{A},\mathtt{B}) & = & [1,+\infty]) & \wedge \\
(MU(\mathtt{B},\mathtt{A}) & = & [0,+\infty]) &
\end{array}
\end{array}
$$

We define a composition relationship between A and B, $CO(\mathtt{A},\mathtt{B})$, as:

$$
\begin{array}{l}
CO(\mathtt{A},\mathtt{B}) = \\
\begin{array}{rcll}
(IS(\mathtt{A},\mathtt{B}) & \subseteq & \{\mathtt{field}, \mathtt{array\ field}, \mathtt{collection}\}) & \wedge \\
(IS(\mathtt{B},\mathtt{A}) & = & \emptyset) & \wedge \\
(EX(\mathtt{A},\mathtt{B}) & = & true) & \wedge \\
(EX(\mathtt{B},\mathtt{A}) & = & false) & \wedge \\
(LT(\mathtt{A},\mathtt{B}) = +) & \wedge & (LT(\mathtt{B},\mathtt{A}) = -) & \wedge \\
(MU(\mathtt{A},\mathtt{B}) & = & [1,+\infty]) & \wedge \\
(MU(\mathtt{B},\mathtt{A}) & = & [1,1]) &
\end{array}
\end{array}
$$

We show in two steps that the four properties are minimal with respect to our definitions and to other properties of the association, aggregation, and composition relationships: First, we show that the properties are minimal for our definitions; Second, we show that the properties appear in all definitions of the relationships in literature. For lack of space, we cannot detail here these two steps. The interested reader may refer to [10] for the demonstration.

Typically, in the first step, we remove a property from the formalisation of a relationship and we show that we cannot distinguish it from another formalisation. For example, the exclusivity property is the only mean to distinguish an aggregation from a composition relationship because values of the other properties of the aggregation relationship satisfy the composition relationship.

In the second step, we study the definitions of the relationships from the literature and we show that they are all expressed using, at least, these four properties. For example, the definitions of the aggregation and compositions relationships by Henderson-Sellers and Barbier [12, table 4, page 356] use several characteristics, among which: *C1. Propagation of one or more operations* and *C5. Propagation of destruction operation* related to the invocation site and lifetime properties; *C2. Ownership* related to the exclusivity property; *P1. Whole–part* related to the multiplicity property. Thus, algorithms based on these four minimal properties identify and only identify association, aggregation, and composition relationships.

Algorithms. Identification of association relationships requires collecting the value of the IS property only, values of the other properties being indifferent. Identification of aggregation relationships requires inferring the values of the IS and MU properties. Identification of the composition relationships requires the values of the IS and MU properties and the values of the EX and LT properties. We compute values of the invocation site, IS, and multiplicity, MU, properties on static models. We infer the values of the exclusivity, EX, and lifetime, LT, properties from dynamic models.

The computation of the *static* values (IS and MU) of the three relationships is simple to perform by analysing programs static models. The values of the MU property corresponds to the fields and arrays and their multiplicities (i.e., multiplicity 1 and $+\infty$). A difficulty arises when fields are typed as Java collections (`Collection`, `Map`), because these collections are not typed. If we assume that these kinds of collections are homogeneous (containing elements with a common superclass different from `Object`), it is possible to determine their types using well-known Java programming idioms, such as pairs of `add()`–`remove()` accessors [13, 18].

We assign a value to the *IS* property according to invocation sites and message types of method calls. We iterate through the class files, looking for byte-codes corresponding to method calls: `InvokeInterface`, `InvokeStatic`, `InvokeSpecial`, and `InvokeVirtual`.

The computation of the *dynamic* values (*EX* and *LT*) of the composition relationship is based on the dynamic models of programs. We check the exclusivity and lifetime properties of composition relationships with the Caffeine tool and dedicated Prolog predicates, as presented in Section 2. The predicates `instanceLevelCompositions/1` and `classLevelCompositions/1` compute the values of the exclusivity and lifetime properties using the order in which field modifications, finalizer exits, and program-end occur. They infer the presence of composition relationships among instances and their respective classes from the values of *EX* and *LT*.

We performed extensive testing of our algorithms on several programs, in particular Java AWT v1.2.2, JHotDraw v5.1, and JUnit v3.7. We analysed each program manually and compared the results of our analyses with these of our algorithms. We find that the identification of association relationships has a precision of 100% and a recall of 100% (4,925 existing), the identification of aggregation relationships has a precision of 75% and a recall of 96% (32 existing, 24 found, 1 false hit), and the identification of composition relationships has a precision of 100% and a recall of 100% (3 existing). The identification of aggregation relationships does not have a precision and a recall of 100% because the developers did not respect some of the idioms used in our detection algorithms to compute values for the *MU* properties. Also, precision and recall for composition relationships may vary depending on the execution paths taken when running programs. Identification of composition relationships suffer from the common limitations of dynamic analyses.

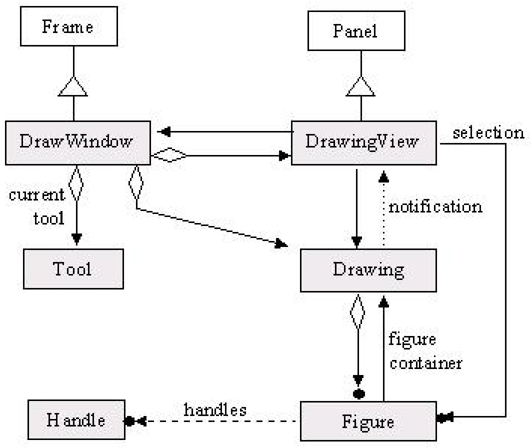


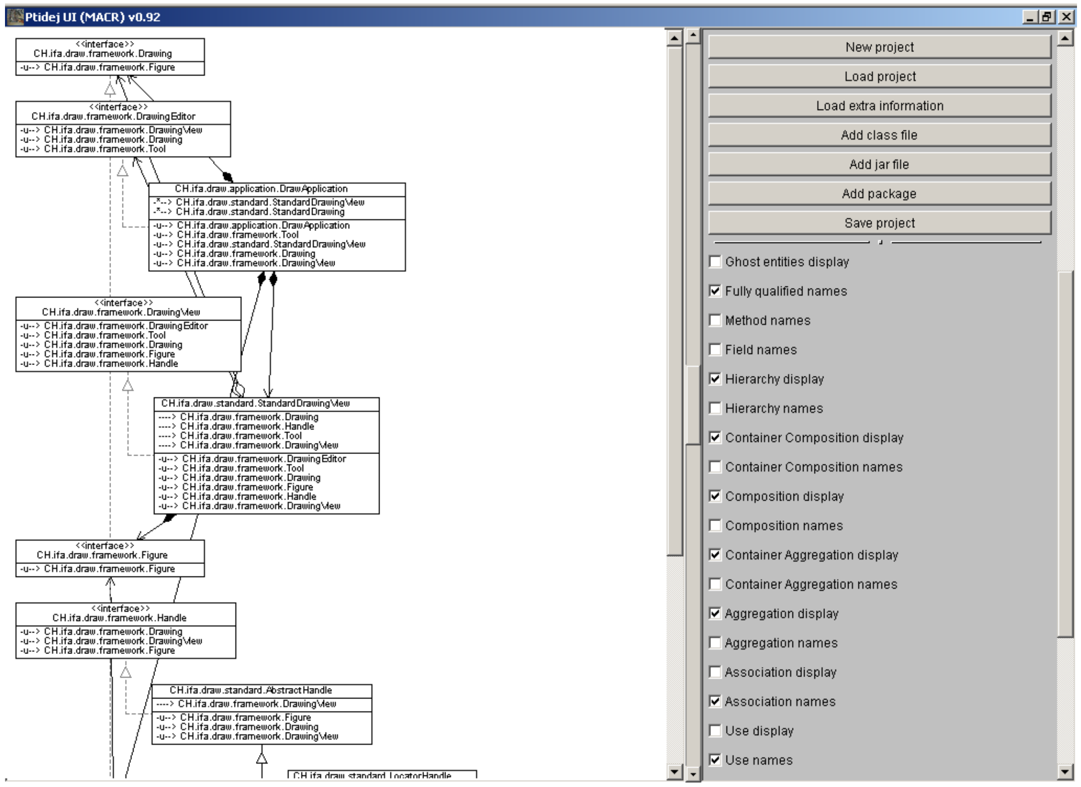
Figure 4: JHotDraw core classes

3.3 Precision

The Ptidej tool suite provide precise class diagrams, representative of programs implementations. Indeed, class diagrams are built with both the static and dynamic models of Java programs using the PADL ClassFile Creator and Caffeine tools. They describe the programs classes and interfaces, and their relationships accurately, using precise definitions and formalisations of the relationships and repeatable algorithms. In particular, we use precise formalisations of the association, aggregation, and composition relationships with four minimal properties, which allow our algorithms to identify these relationships accurately.

4 Application

We present an experimentation of our tool suite on the JHotDraw program. We choose JHotDraw because it is an independent medium-size real-world program. We want to show that class diagrams recovered using Ptidej are (1) easily obtained and (2) more precise than class diagrams usually provided


Ptidej UI (MACR) v0.92
<<interface>>
CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.Figure
<<interface>>
CH.ifa.draw.framework.DrawingEditor
-u--> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.Tool
CH.ifa.draw.application.DrawApplication
-*--> CH.ifa.draw.standard.StandardDrawingView
-*--> CH.ifa.draw.standard.StandardDrawing
-u--> CH.ifa.draw.application.DrawApplication
-u--> CH.ifa.draw.framework.Tool
-u--> CH.ifa.draw.standard.StandardDrawingView
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.DrawingView
<<interface>>
CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.DrawingEditor
-u--> CH.ifa.draw.framework.Tool
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Handle
CH.ifa.draw.standard.StandardDrawingView
----> CH.ifa.draw.framework.Drawing
----> CH.ifa.draw.framework.Handle
----> CH.ifa.draw.framework.Tool
----> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.DrawingEditor
-u--> CH.ifa.draw.framework.Tool
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Handle
-u--> CH.ifa.draw.framework.DrawingView
<<interface>>
CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Figure
<<interface>>
CH.ifa.draw.framework.Handle
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.Figure
CH.ifa.draw.standard.AbstractHandle
----> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.DrawingView
CH.ifa.draw.standard.LocatorHandle
New project
Load project
Load extra information
Add class file
Add jar file
Add package
Save project

- ☐ Ghost entities display
- ☑ Fully qualified names
- ☐ Method names
- ☐ Field names
- ☑ Hierarchy display
- ☐ Hierarchy names
- ☑ Container Composition display
- ☐ Container Composition names
- ☑ Composition display
- ☐ Composition names
- ☑ Container Aggregation display
- ☐ Container Aggregation names
- ☑ Aggregation display
- ☐ Aggregation names
- ☐ Association display
- ☑ Association names
- ☐ Use display
- ☑ Use names

Figure 5: Top view of JHotDraw core classes in Ptidej and their *concrete* relationships

with programs documentation. Thus, we compare the architecture of JHotDraw as described by class diagrams from its documentation with the class diagram obtained using Ptidej. For each class, interface, and their relationships in the automatically reverse-engineered class diagram of JHotDraw, we assess their consistency (existence, absence, characteristics) with the JHotDraw core classes diagram, on Figure 4.

4.1 JHotDraw

JHotDraw is a highly customisable two-dimensional graphic framework for structured drawing editors [14]. It simplifies the development of drawing applications, such as for Pert diagrams, UML diagrams. The 5.1 version weighs 155 classes, distributed across 11 packages, for about 16,000 lines of Java source code. Its source code and binaries are freely available at `http://members.pingnet.ch/gamma/`. Its core classes, interfaces, and their relationships are shown on the class diagram, Figure 4, as provided in the framework documentation.

A `DrawingWindow` is a window, subclass of `Frame`, displaying a `Drawing` through a `DrawingView`, subclass of `Panel`, with a `Tool` to manipulate the drawing. An instance of `DrawingWindow` aggregates instances of `DrawingView`, `Drawing`, and `Tool` (white lozenges). An instance of `DrawingView` knows its containing `DrawingWindow`, contained `Drawing`, and selected `Figure` (arrows). Instances of `Drawing` use instances of `DrawingView` (dash arrow). A `Drawing` is composed of `Figures` (white lozenge with black circle) which know their containing `Drawing` (arrow) and create `Handles` to allow user-interactions (dash arrow with black circle).

4.2 JHotDraw and Ptidej

The class diagram shown in Figure 4 does not reflect the *real* implementation of the JHotDraw framework: It is obsolete and inaccurate. It is neither complete nor precise enough to allow maintainers to perform modifications with confidence. The class diagram shows classes and relationships that do not exist in the *real* implementation and only represents a *simplification* of the framework. Indeed, we built the JHotDraw class diagram using the Ptidej tool suite and we found several differences with the provided class diagram. Figures 5 and 6 show the JHotDraw core classes, from the documentation, in Ptidej.

The `DrawingWindow` class has been renamed `DrawingEditor` in the implementation. Core classes are *in facts* interfaces (`<<interface>>` stereotype) and only use relationships exist among them (`-u-->` symbol). We must add classes implementing these interfaces to display relationships *really* existing among them. Figures 5 and 6 show the JHotDraw core interfaces and implementation classes, recovered by static and dynamic analyses.

The `DrawApplication` class, implementing the `DrawingEditor` interface, is composed of (black lozenge) the `Drawing` interface (and its implementation class `StandardDrawing`). It is also composed of the `StandardDrawingView` and `Tool` classes. These composition relationships are conform to what we could expect: An instance of class `DrawApplication` represents the JHotDraw editor, composed of a view and a drawing panel, when the editor is destroyed (closed), view and drawing panel are destroyed also.

The `StandardDrawingView` class is composed of instances of classes implementing the `Figure` interface, through the composed instance of class `StandardDrawing`, implementing the `Drawing` interface and extending the `CompositeFigure` class. The `StandardDrawingView` class aggregates instances of classes implementing the `Drawing` and `DrawingEditor` interfaces (white lozenges). Instances of `StandardDrawingView` use instances of the class implementing interface `DrawingEditor` as *backpointers* to send messages to their parents (instances of `DrawApplication`).

The `CompositeFigure` class implements the Composite design pattern: The `Figure` interface plays the role of Component, the `CompositeFigure` of Composite, and other classes implementing interface `Figure` play the role of Leaves.

Finally, classes are associated with, use, or create various instances of other classes to perform their tasks, for example class `DrawApplication` creates its composing instances of classes `StandardDrawing` and `StandardDrawingView`.

4.3 Class Diagrams Comparisons

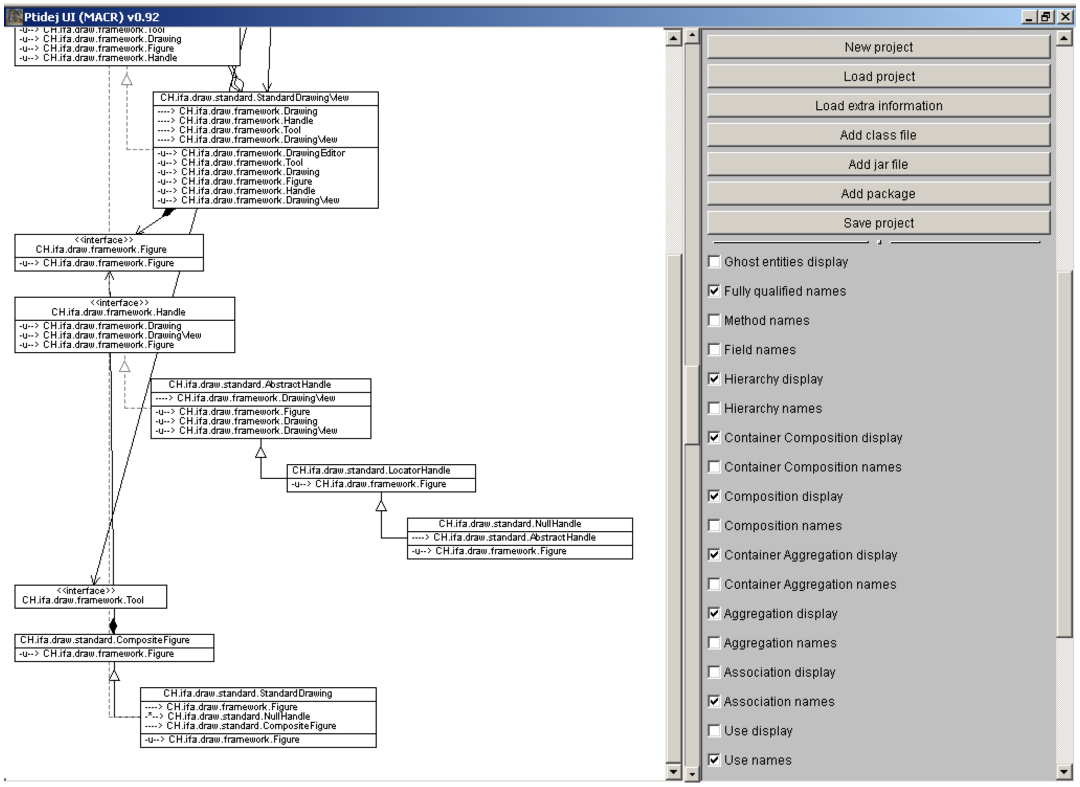
The class diagram shown in Figures 5 and 6 provides maintainers with a more precise view of the JHotDraw framework than this provided by the authors, in Figure 4.

The class diagram is more precise because it is built from both static and dynamic models of JHotDraw, using precise and consensual definitions of the use, association, aggregation, and composition relationships and all data required to distinguish these relationships. Thus, it distinguishes clearly classes, interfaces, and inheritance, instantiation, use, association, aggregation, and composition relationships.

The class diagram is built semi-automatically (with user-interactions), using the Ptidej tool suite, and does not require any manual analysis. It is created by the PADL ClassFile Creator tool in about 2 seconds (along with its graphical representation) on an AMD Athlon 64bits processor at 2GHz. It is refined with data obtained from Caffeine, which computation-time depends on the number of generated events [11]. Typically, execution time may be slowed down by a factor between 100 and 5,000 because of the inefficient yet frequent exchange of data between the Prolog engine performing the analyses and the analysed program.

5 Conclusion

We presented Ptidej, a tool suite for the precise semi-automatic reverse engineering of Java programs as UML-like class diagrams; i.e., classes and interfaces, inheritance, instantiation, use, association, aggregation, and composition relationships. Ptidej uses both static and dynamic models of programs. Static models are analysed using the PADL ClassFile Creator tool, dynamic models using the


Ptidej UI (MACR) v0.92
-u--> CH.ifa.draw.framework.Tool
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Handle
CH.ifa.draw.standard.StandardDrawingView
----> CH.ifa.draw.framework.Drawing
----> CH.ifa.draw.framework.Handle
----> CH.ifa.draw.framework.Tool
----> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.DrawingEditor
-u--> CH.ifa.draw.framework.Tool
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Handle
-u--> CH.ifa.draw.framework.DrawingView
<<interface>>
CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Figure
<<interface>>
CH.ifa.draw.framework.Handle
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.Figure
CH.ifa.draw.standard.AbstractHandle
----> CH.ifa.draw.framework.DrawingView
-u--> CH.ifa.draw.framework.Figure
-u--> CH.ifa.draw.framework.Drawing
-u--> CH.ifa.draw.framework.DrawingView
CH.ifa.draw.standard.LocatorHandle
-u--> CH.ifa.draw.framework.Figure
CH.ifa.draw.standard.NullHandle
----> CH.ifa.draw.standard.AbstractHandle
-u--> CH.ifa.draw.framework.Figure
<<interface>>
CH.ifa.draw.framework.Tool
CH.ifa.draw.standard.CompositeFigure
-u--> CH.ifa.draw.framework.Figure
CH.ifa.draw.standard.StandardDrawing
----> CH.ifa.draw.framework.Figure
-*--> CH.ifa.draw.standard.NullHandle
----> CH.ifa.draw.standard.CompositeFigure
-u--> CH.ifa.draw.framework.Figure
New project
Load project
Load extra information
Add class file
Add jar file
Add package
Save project

[ ] Ghost entities display
[✓] Fully qualified names
[ ] Method names
[ ] Field names
[✓] Hierarchy display
[ ] Hierarchy names
[✓] Container Composition display
[ ] Container Composition names
[✓] Composition display
[ ] Composition names
[✓] Container Aggregation display
[ ] Container Aggregation names
[✓] Aggregation display
[ ] Aggregation names
[ ] Association display
[✓] Association names
[ ] Use display
[✓] Use names

Figure 6: Bottom view of JHotDraw core classes in Ptidej and their *concrete* relationships

Caffeine tool. PADL ClassFile Creator and Caffeine compute values of four minimal properties (exclusivity, lifetime, multiplicity, and invocation site) that we use to formalise the use, association, aggregation, and composition relationships. We exemplified the Ptidej tool suite on a simple document description program and detailed its application on the JHotDraw framework. We showed that the class diagram obtained for the JHotDraw framework semi-automatically is more precise than the class diagram provided with the documentation from the authors.

Currently, we work on replacing dynamic analyses with type analyses of single uses of values. Also, we are extending PADL, using its implementation of the Visitor design pattern, with recovery algorithms for more UML constituents, such as data-types, implementation classes, utility classes. Future work includes analyses of real-world programs (thousand of classes), such as telecommunication systems or development environments to assess the usefulness of recovered class diagrams for maintainers. Also, we intend to implement sophisticated layout algorithms to improve the visual appealing of the reverse engineered class diagrams [7, 17]. Finally, we investigated the use of the reverse engineered class diagrams to identify automatically design patterns [1]. We plan to extend our experience to design defects [9] to help maintainers further.

References

[1] Hervé Albin-Amiot, Pierre Cointe, Yann-Gaël Guéhéneuc, and Narendra Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In Debra Richardson, Martin Feather, and Michael Goedicke, editors, *proceedings of the 16th conference on Automated Software Engineering*, pages 166–173. IEEE Computer Society Press, Nov. 2001.

[2] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and

code synthesis. In Bedir Tekinerdogan, Pim Van Den Broek, Motoshi Saeki, Pavel Hruby, and Gerson Sunyé, editors, *proceedings of the 1st ECOOP workshop on Automating Object-Oriented Software Development Methods*. Centre for Telematics and Information Technology, University of Twente, Oct. 2001. TR-CTIT-01-35.

[3] Giuliano Antoniol, Roberto Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In Scott Tilley and Giuseppe Visaggio, editors, *proceedings of the 6th International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, Jun. 1998.

[4] Shigeru Chiba. Javassist – A reflection-based programming wizard for Java. In Jean-Charles Fabre and Shigeru Chiba, editors, *proceedings of the OOPSLA workshop on Reflective Programming in C++ and Java*. Center for Computational Physics, University of Tsukuba, Oct. 1998. UTCCP Report 98-4.

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

[6] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. Finding refactorings via change metrics. In Doug Lea, editor, *proceedings of 15th conference on Object-Oriented Programming Systems, Languages and Applications*, pages 166–177. ACM Press, Oct. 2000.

[7] Holger Eichelberger and Jürgen Wolff von Gudenberg. On the visualization of Java programs. In Stephan Diehl, editor, *proceedings of the 1st international seminar on Software Visualization*, pages 295–306. Springer-Verlag, May 2002.

[8] Matt Greenwood. *CFParse Distribution*. IBM AlphaWorks, Sep. 2000.

[9] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Quioyun Li, Richard Riehle, Gilda Pour, and Bertrand Meyer, editors, *proceedings of the 39th conference on the Technology of Object-Oriented Languages and Systems*, pages 296–305. IEEE Computer Society Press, Jul. 2001.

[10] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In Doug C. Schmidt, editor, *proceedings of the 19th conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Oct. 2004. To appear.

[11] Yann-Gaël Guéhéneuc, Rémi Douence, and Narendra Jussien. No Java without Caffeine – A tool for dynamic analysis of Java programs. In Wolfgang Emmerich and Dave Wile, editors, *proceedings of the 17th conference on Automated Software Engineering*, pages 117–126. IEEE Computer Society Press, Sep. 2002.

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

[13] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In David Garlan and Jeff Kramer, editors, *proceedings of the 21st International Conference on Software Engineering*, pages 194–202. ACM Press, May 1999.

[14] Wolfram Kaiser. Become a programming picasso with JHotDraw – Use the highly customizable GUI framework to simplify draw application development. *JavaWorld*, Feb. 2001.

[15] Jeffrey Korn, Yih-Farn Chen, and Eleftherios Koutsofios. Chava: Reverse engineering and tracking of Java applets. In Kostas Kontogiannis and Françoise Balmas, editors, *proceedings of the 6th Working Conference on Reverse Engineering*, pages 314–325. IEEE Computer Society Press, Nov. 1999.

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

[17] Jochen Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In Giuseppe Di Battista, editor, *proceedings of the 5th international symposium on Graph Drawing*, pages 415–424. Springer-Verlag, Sep. 1997.

[18] Paolo Tonella and Alessandra Potrich. Reverse engineering of the UML class diagram from C++ code in presence of weakly typed containers. In Gerardo Canfora and Anneliese Amschler Andrews-Von Maryhauser, editors, *proceedings of the 9st International Conference on Software Maintenance*, pages 376–385. IEEE Computer Society Press, Nov. 2001.