

An Empirical Study of Visualising Types and their Implementations Separately

Sang-won Hwang¹, Yann-Gaël Guéhéneuc², and Yeong-gwang Nam¹

¹ SE Lab., Yonsei University, Won-ju, Korea

² Ptidej Team, École Polytechnique de Montréal, Québec, Canada

E-mails: `arsenal@yonsei.ac.kr`,
`yann-gael.gueheneuc@polymtl.ca`,
`yknam@yonsei.ac.kr`

April 30, 2014

1 Overview

Your participation is necessary to collect **anonymous** data regarding the time and the correctness of your answers when performing typical software engineering tasks using an experimental tools, called in the following the “Ptidej UI Viewer Standalone Swing”. It consists in using a graphical representation of some code and related tools to perform four different tasks. It allows us to collect data and compare this data while taking into account the the visualisation techniques used to display the code and the performed task. During your participation, we will collect three types of **anonymous** data:

- The time that you take to perform the different tasks;
- The answers that you give to the different tasks;
- General demographic / academic data.

At any point before or during the experiment, you can stop your participation without question or penalty whatsoever. We will collect separately your name and e-mail address to send you the expected answers and general results of this experiment in the near future, if desired.

In any case, please do not speak about this experiment, its form and content, to your peers to avoid biasing their participation.

2 Experiment Background

2.1 Type vs. Implementation Classes

Object-oriented programming languages have all essential features that make them different from procedural programming languages (and others). These features are encapsulation, inheritance, types, polymorphism (overloading and overriding), and abstraction.

Of particular interest in this experiment are *types* and their *implementation classes*. In object-oriented programming, a type is typically, in C++, a class declaring only pure virtual member functions while, in Java, it is an **interface** or an abstract class at the top of the inheritance hierarchy.

The developers, who provide types in their library, use types to tell the users about the abstractions that the users can manipulate, without telling them about the concrete implementations of these abstractions. The developers who use types can be confident that changes to the concrete implementations of these types will not change the behaviour of their programs.

Different developers may provide implementation classes for types, for example there exist many implementation classes for the types in the Collection library of Java: the standard implementation classes provided in the Java Development Toolkit but also the implementation classes available in the Apache Commons Collections. These implementation classes must implement the methods declared by the types that they implement. They are hidden to users. Therefore, it is expected from a well-designed library that users must know only about types, not about their implementation classes.

2.2 UML-like Class Diagrams

To visually show types and implementation classes, the OMG Unified Modeling Language (UML) propose a notation for class diagrams. An example of class diagram is shown in Figure 1. This diagram is similar to a UML class diagram. Each box is either a type or an implementation class. 1 shows two types, **AbstractDocument** and **Element**, which are tagged using the `<<interface>>` stereotype as well as five implementation classes, **Main**, **Document**, **Title**, **Paragraph**, and **IndentedParagraph**.

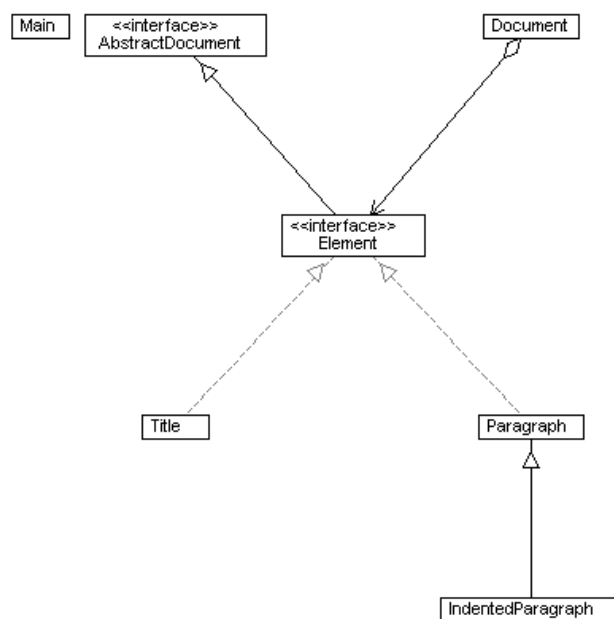



Figure 1: Example of UML-like class diagram

Boxes are connected by arrows describing the relationships between types, between implementation classes, and between types and implementation classes:

- Plain, black arrows with black, empty triangles describe an inheritance relationship between two types, for example type **AbstractDocument** is the syper-type of type **Element**;
- Plain, black arrows with black, empty triangles also describe an inheritance relationship between two implementation classes, for example classes **Paragraph** is the syper-class of class **IndentedParagraph**;
- Dashed, gray arrows with gray, empty triangles describe an implementation relationship between a type and its implementation class(es), for example type **Element** is a direct syper-type of the implementation classes **Title** and **Paragraph** (i.e., the classes **Title** and **Paragraph** implement the type **Element**);
- Plain, black arrows with black arrowheads describe different relationships, depending on the symbol (if any) at the other extremity of the arrow:
 - No symbol: the arrows represents a use relationship, the type or implementation at the origin of the arrow uses the type or class at the destination of the arrow;
 - Black, empty lozenge: the arrows represents an aggregation relationship, i.e., the implementation class at the origin of the arrow aggregate instances of the type or class at the destination of the arrow, for example the implementation class **Document** aggregates objects of type **Element**;
 - Black, filled lozenge: the arrows represents a composition relationship, i.e., the implementation class at the origin of the arrow is composed of instances of the type or class at the destination of the arrow;

2.3 Visualisation

Figure 2 shows the visualisation tool, Ptidej UI Viewer Standalone Swing, that you will use to answer four different questions about three different software programs. The tool shows a graphical representation of the constituents of a program, using a UML-like class diagram as explained before. Such a diagram shows the types (Java interfaces) and the implementation classes (Java classes) in a program. It also shows the relationships among types and implementation classes, including inheritance, use (invocation), association, aggregation, and composition.

You can choose the entities (types and/or implementation classes) and the relationships to be displayed as well as show fields and methods (unnecessary to answer the questions). Also, you can also increase/decrease the size of the areas by dragging the divider in between. Finally, you can display one particular entity of interest by selecting, in the hierarchical view, the blue arrow  next to its name.

Please turn your attention to the computer, the tool is available, showing a system that you will *not* use during the experiment. You can play with the tool to become familiar with the various checkboxes and viewer options. If you have any question, please ask the experimenter.

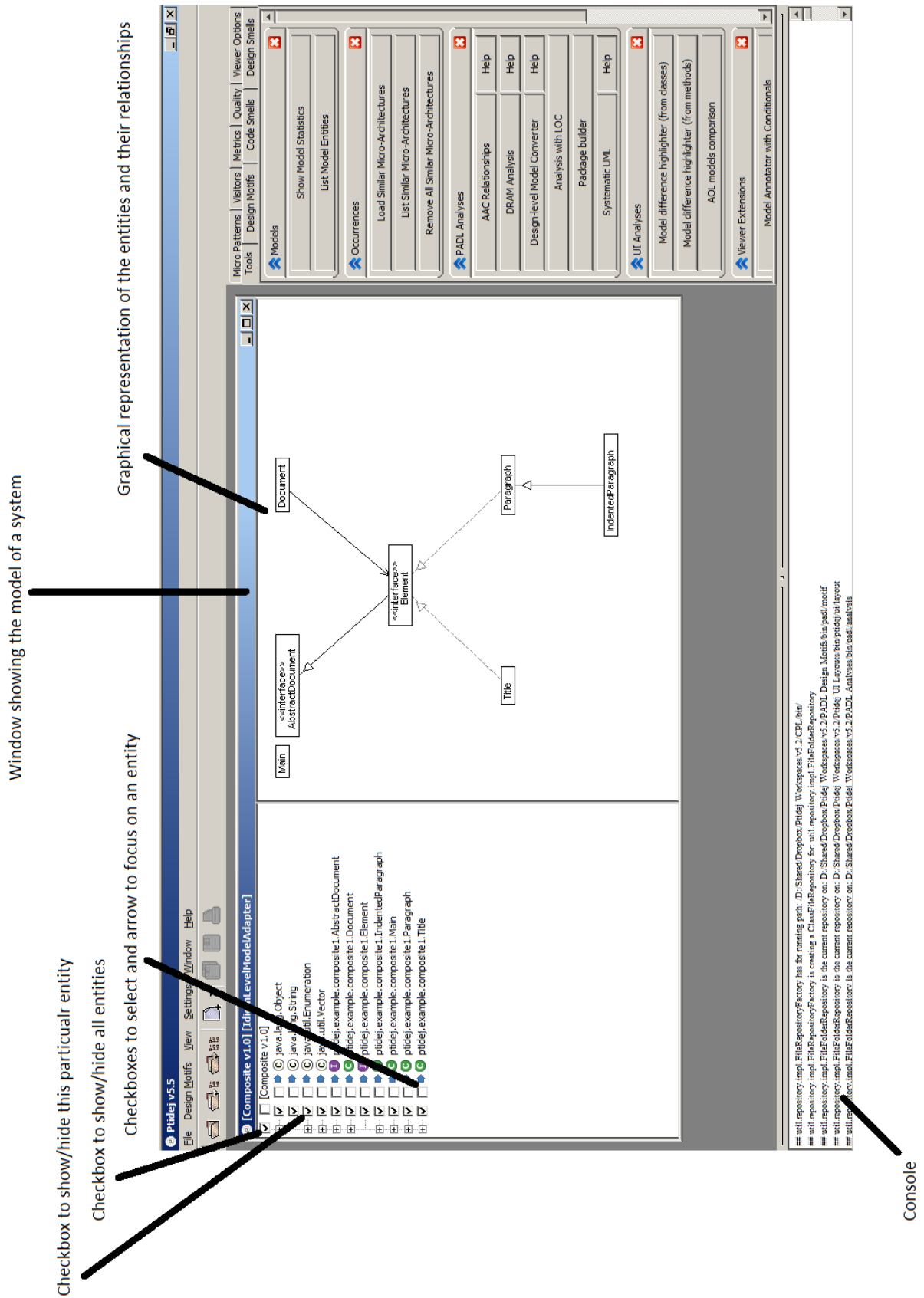


Figure 2: Summary explanation of the tool

3 Data Collection

3.1 Demographic / Academic Data

Please answer the following questions:

1. What is your country of origin?
2. What is your current level of education?

The following questions use a 3-point Lickert-scale, where 1 is poor, 1 is average, and 2 is best. Please tell the experimenter, in your own opinion, what are your levels?

Java knowledge	1	2	3
UML knowledge	1	2	3
Software engineering knowledge	1	2	3

3.2 Experimental Data

Now, it is time to performed some tasks. We will collect the time that you take to perform the tasks as well as your answers to the task questions. The experimenter will close the tool and reopen it, showing a new system. He will tell you which system and which question to answer and will refer you to the following Table for the exact questions. Then, he will start a chronometer to measure the time that you take answering the question and, when you are ready, will record your answer and the time. Finally, he will again close and reopen the tool to repeat this procedure a total of four times, once for each question.

But, first, please read these short descriptions of the programs:

- **JHotDraw:** JHotDraw is a Java GUI framework for technical and structured Graphics. JHotDraw defines a basic skeleton for a GUI-based editor with tools in a tool palette, different views, user-defined graphical figures, and support for saving, loading, and printing drawings. The framework can be customized using inheritance and combining components. It includes some predefined figures like **AbstractFigure**, **CompositeFigure**, and **AttributeFigure**. It offers JHotDraw various tools, for example **CreationTool**, **ConnectionTool**, **SelectionTool**, and **TextTool** as well as related commands. It can handle various events as well as connections among figures.
- **JRefactory:** A refactoring tool for the Java programming language, it includes the **JavaStyle** pretty printer, a UML Java class diagram viewer, a coding standards checker and computes program metrics. Plugins for JEdit, Netbeans, JBuilder and other IDEs are available. The core of JRefactory is a parser to create a model of some Java code, among other classe **ParseFactory** and **JavaParserFactory**. This model is then used to apply various transformations, either to modify how the code is written, **PrettyPrintString**, or to move code constituents around, for example **AddClassRefactoring**.
- **PADL:** PADL stands for Pattern and Abstract-level Description Language. It is a meta-model to describe programs at different levels of abstractions. There are three different levels of abstractions to model programs, collectively called abstract-level models, for example **ICodeLevelModel**. Typical design information is obtained from the idiom-level model, for example occurrences of design motifs or of code and design smells. The PADL meta-model includes an extensive hierarchy of interfaces to describes binary class relationships, which are **IRelationship** and include, for example, **ICreation**.

JHotDraw	Type 1	Where is there any code involved in the implementation of this behaviour: <code>CH.ifa.draw.framework.Handle.owner()</code> ?
	Type 2	What does the definition or declaration of this look like: <code>CH.ifa.draw.standard.RelativeLocator.locate(Figure)</code> ?
	Type 3	How are these types or objects related: <code>CH.ifa.draw.framework.Drawing</code> and <code>CH.ifa.draw.framework.DrawingChangeEvent</code> ?
	Type 4	What is the mapping between these types and implementation classes: <code>CH.ifa.draw.framework.Tool</code> and <code>CH.ifa.draw.standard.SelectAreaTracker</code> ?
JRefactory	Type 1	Where is there any code involved in the implementation of this behaviour: <code>org.acm.seguin.parser.Node.jjtAddChild(Node,int)</code> ?
	Type 2	What does the definition or declaration of this look like: <code>org.acm.seguin.refactor.field.AddFieldVisitor.visit(ASTClassBody,Object)</code> ?
	Type 3	How are these types or objects related: <code>org.acm.seguin.refactor.Refactoring</code> and <code>org.acm.seguin.uml.refactor.RefactoringDialog</code> ?
	Type 4	What is the mapping between these types and implementation classes: <code>org.acm.seguin.metrics.MetricsFrame</code> and <code>org.acm.seguin.metrics.ProjectMetricsFrame</code> ?
PADL	Type 1	Where is there any code involved in the implementation of this behaviour: <code>padl.kernel.IMethod.getReturnType()</code> ?
	Type 2	What does the definition or declaration of this look like: <code>padl.kernel.impl.MemberGhost.attachTo(IElement)</code> ?
	Type 3	How are these types or objects related: <code>padl.kernel.IContainer</code> and <code>padl.kernel.IFilter</code> ?
	Type 4	What is the mapping between these types and implementation classes: <code>padl.kernel.IUseRelationship</code> and <code>padl.kernel.impl.ContainerComposition</code> ?