A Formal and a Cognitive Model of Anaphors in Java

Sebastian Lohmeier
sl@monochromata.de

Abstract
Two prototypical models are in development to demonstrate the feasibility of programming with direct and indirect anaphors. A formal model is developed based on the Eclipse IDE. The model generates executable code, handles referential ambiguity, highlights anaphora relations, and permits programmers to switch between source code with anaphors and normal Java code. A cognitive model is developed to forecast when a programmer will be able to understand specific indirect anaphors and when normal Java code should be presented instead. Both models lay the foundation for indirect anaphors that are resolved at edit-time and that shorten source code in cases when comprehensibility is expected to be maintained.

1. Introduction

While reference in statically typed object-oriented programming languages like Java typically uses local variables that declare an identifier, it would also be possible to refer using anaphors that do not require the declaration of an identifier but exploit readily-available textual information. An instance created by the expression `new ServiceRegistrar()` could e.g. be referred to using the direct anaphor `serviceRegistrar` that is based on the recurrence of the words `service` and `registrar` in the direct anaphor and its related expression `new ServiceRegistrar()`. It would also be possible to refer to parts of wholes. E.g. given a `new RegistrarLocator()` and the knowledge that the getter method `RegistrarLocator.getRegistrar()` returns a `ServiceRegistrar` instance, the indirect anaphor `serviceRegistrar` could be used to refer to the service registrar available from the previously-mentioned `new RegistrarLocator()`.

Lohmeier (2015) attempted to test experimentally when indirect anaphors in Java are understood by programmers and when they are not. In the experiment, it could be shown that indirect anaphors based on less familiar part-whole relations are understood less easily than indirect anaphors based on more familiar part-whole relations. It remained unclear, whether indirect anaphors based on more familiar part-whole relations are understood as easily as normal Java code. It could not be shown reliably that indirect anaphors based on well-known part-whole relations improve the comprehension of expert programmers. Effects of indirect anaphors on task durations were inconclusive. Responses to the post-test questions and statements that participants made during de-briefing indicated that the use of anaphors could benefit from a number of modifications. Programmers might benefit from (indirect) anaphors while authoring (instead of reading) source code. Anaphors should, in addition, come with the typical tool support of IDEs (e.g. marking other occurrences of a name at the current cursor position and going to the declaration of the current name) that was disabled during the experiment.

Because programming can be modelled formally and cognitively, a prototypical formal model of anaphors has been implemented besides the prototypical cognitive model that had been implemented as part of Lohmeier (2015). While the formal model was implemented in the Eclipse IDE to test whether it is possible to switch between anaphors and normal Java code, the cognitive model was implemented to see whether jACT-R[1], an Eclipse-based re-implementation of the cognitive architecture ACT-R (Anderson et al., 2004) can be used to compute activation levels of knowledge representations automatically derived from the abstract syntax tree (AST) of the Eclipse IDE. Both models are briefly described in the following.

2. JDT with anaphors

The Java development tools (JDT) of the Eclipse IDE[2] have been used to create a prototypical editor for anaphors. The editor is based on the Java editor of the JDT. In the editor, anaphors are translated at

[1] `http://www.jact-r.org/`
[2] `http://www.eclipse.org/jdt/`

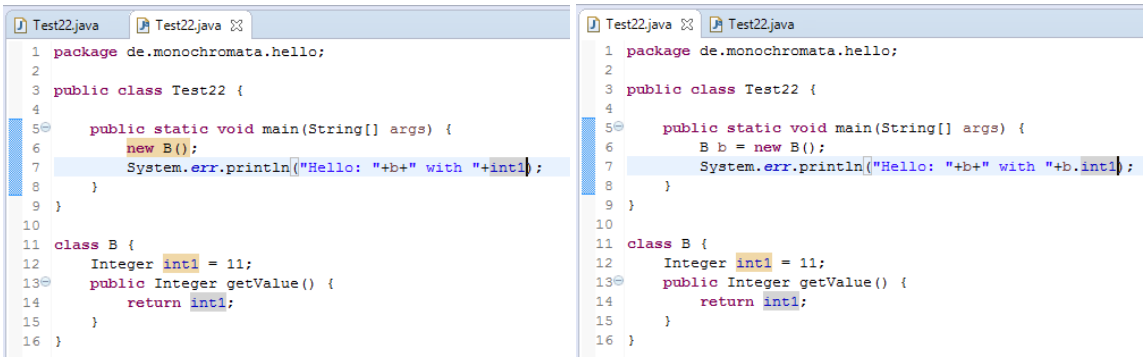

Figure 1 – The direct anaphor `b` and the indirect anaphor `int1` in a Java editor that supports anaphors (left) and the code generated at edit-time in a normal Java editor (right). The editor on the left marks the occurrences of the referent of the indirect anaphor `int1` at the cursor position as well as its related expression `new B()`.

Related expression	Anaphora resolution	Referentialisation
CIC The related expression is a class instance creation expression (e.g. `new Service()`)	**DA1Re** The anaphor refers to the referent of the related expression.	**Rn** The referent of the anaphor has a name that is equal to the simple name that acts as anaphor.
LVD The related expression is a declaration of a parameter or a local variable (e.g. `Service s= new Service()`).	**IA2F** The anaphor refers to the field of the related expression that the referentialisation strategy can be applied to. **IA2Mg** The anaphor refers to the return value of the getter method of the related expression that the referentialisation strategy can be applied to.	**Rt** The referent of the anaphor has a type with a name that is equal to the simple name that acts as anaphor.

Table 1 – Strategies available for anaphora resolution in the formal model

edit-time, i.e. when entered, instead of at compile-time. The prototype currently supports combinations of the related expressions, anaphors resolution strategies and referentialisation strategies listed in Table 1. (The left part of Figure 1 contains the related expression `new B()` in line 6, followed by the indirect anaphor `int1` in line 7 that combines the strategies CIC, IA2F and Rn from Table 1.) Referential ambiguity is eliminated in a small number of cases. When there is a local variable declaration that is initialised with a class instance creation expression, both `B b = new B()` and `new B()` are potential related expressions and the local variable declaration is preferred over the contained class instance creation expression. When a getter method declaration `public Integer getValue() { return int1; }` and a field declaration `public Integer int1;` are found as potential referents of the indirect anaphor `int1`, the getter method declaration is preferred over the field declaration.

Figure 1 shows two anaphors `b` and `int1` (on the left) and the code generated by the refactoring that translates anaphors into normal Java code (on the right). The use of the JDT acknowledges the influence of the programming environment highlighted by Green (1989) not only by shifting the translation of anaphors from compile-time to edit-time but by providing typical guidance to programmers like the *mark occurrences* feature that in the left part of Figure 1 marks the related expression `new B()` (the whole) of the indirect anaphor `int1` (the part). Guidance like *mark occurrences* is expected to ease comprehension of indirect anaphors based on less well-known part-whole relations.

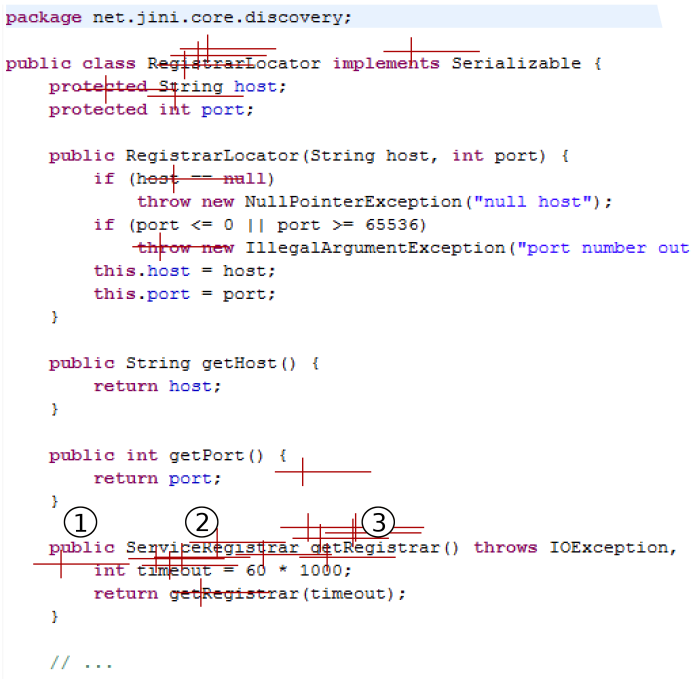

Figure 2 – Source code of the `RegistrarLocator` class with fixations (shown as red crosses) of a single participant. The code shows (1) the accessibility modifier `public`, (2) the return type `ServiceRegistrar` and (3) the method name `getRegistrar()`.

Anaphors are translated into normal Java code by applying a refactoring. The refactoring is applied automatically when the anaphors-enabled editor is used to enter anaphors. While the refactoring generates Java code, the generated statements and expressions are hidden by the anaphors-enabled editor – only the entered anaphor is shown in this editor. The editor therefore separates the user-interface and the knowledge representation functions of the programming language. When the normal Java editor is used instead, no anaphors are display but all Java code is shown. Java code is saved to disk without anaphors. This enables programmers to write anaphors irregardless of whether other programmers might understand them.

When a programmer opens a file she added anaphors to previously, they could be restored automatically, e.g. from a separate meta-data store on disk. The prototype therefore shows that edit-time anaphors can be implemented in an editor for the Eclipse IDE. The editor can be used besides the normal Java editor of Eclipse. It would also be possible to generate anaphors for previously unread code. A cognitive model could be used to decide when to generate such anaphors.

3. A model of anaphor comprehension in jACT-R

Eye movement data obtained in Lohmeier (2015) has been input into a cognitive model of source code reading that re-generates fixation durations. The fixations of a programmer reading source code are therefore mapped to words displayed by the Java editor of Eclipse (see Figure 2). The words in the source code are mapped to nodes in the AST of Eclipse (see the left part of Figure 3) and lead to a sequence of fixation durations on AST nodes. The AST nodes are also used to generate an intermediate knowledge representation (see the right part of Figure 3). The intermediate knowledge representation is fed into a jACT-R model that creates chunks of declarative knowledge for which activation values are computed. The sequence of fixated AST nodes is fed into the jACT-R model and the model re-generates the durations of these fixations. The duration of a re-generated fixation is calculated based on the activation of the chunk in the declarative memory of the jACT-R model that represents the AST node underlying the fixation. The more often a chunk is retrieved, the more active it is, but activation decays with time. Fixations that involve a retrieval of a chunk are shorter the higher the activation

Figure 3 – Abstract syntax tree (left) and intermediate knowledge representation (right) for the code shown in Figure 2. The knowledge representation is input into a jACT-R model that creates chunks in declarative memory from the intermediate representation.

of the chunk at the time of the retrieval, because retrieval is faster for higher activation. The jACT-R model is assumed to model the comprehension of anaphors in source code, if it re-generates the fixation durations input to it. This is not the case so far, the model currently over-estimates fixation durations (Lohmeier & Russwinkel, 2015). If the generated fixation durations are reasonably close to the durations of experimentally-obtained fixations input into the model, the model may be assumed to match cognitive processing in humans reading anaphors in source code in situations resembling the experimental setup. At that point, the activation values of a chunk computed by the model might be used to forecast whether it will be easy or hard to understand an indirect anaphor whose comprehension requires the chunk to be retrieved from memory.

4. Conclusion

The two prototypes demonstrate that it is possible to implement edit-time anaphors in a Java-based editor and to model their comprehension in jACT-R. Both models could be integrated by using the cognitive model to forecast whether a file of source code opened in a new editor should be displayed with indirect anaphors or with normal Java code. That would permit to vary redundancy in source code in order to improve the comprehension of individual programmers: programmers would be presented with indirect anaphors that do not repeat the relations that they already know well, relations that are not well known would be presented as normal Java code that explicates these relations.

5. References

Anderson, J. R., Bothell, D., Byrne, M. D., Douglass, S., Lebiere, C., & Qin, Y. (2004). An integrated theory of mind. *Psychological Review*, *111*(4), 1036–1060.

Green, T. R. (1989). Cognitive dimensions of notations. In A. Sutcliffe & L. Macaulay (Eds.), *People and computers V* (pp. 443–460). Cambridge: Cambridge University Press.

Lohmeier, S. (2015). *Experimental evaluation and modelling of the comprehension of indirect anaphors in a programming language. Version 1.3.* Retrieved 2016/05/16, from `http://monochromata.de/master_thesis/`

Lohmeier, S., & Russwinkel, N. (2015). Explaining eye movements in program comprehension using jACT-R. In *Proceedings of the 13th international conference on cognitive modeling (ICCM).*