Helping manage the concern of object cloning in Java programs
(COMP 762 Project 2 report)

Eric Bodden
Sable Research Group, McGill University
eric.bodden@mail.mcgill.ca

1. Introduction

In previous work [2] we investigated the concern of tagging entities of various intermediate representations of the Soot [9] bytecode analysis and optimization framework with meta-data. The study uncovered a problem with the way such meta-data tags are currently being copied. The latest implementation of Soot copies tags by manual method calls to `Host.addAllTagsOf(Host)`. This necessarily leads to an inconsistent implementation over time, as calls to this method might be forgotten. Indeed, in [2] we pointed out places in the code where such calls were added years after the surrounding code was written, indicating that a latent error was found caused by tags not being copied.

We also suggested that automatic copying of tags could alleviate the problem of inconsistency. Such automatic copying can be implemented by providing consistent implementations of the `clone()` method on all types which can be tagged. (All such types implement the `Host` interface.) Unfortunately, it turned out that almost all of the current implementations of `clone()` are flawed: Instead of calling **`super`**`.clone()` they call a copy constructor. This leads to tags not being copied. Furthermore, it leads to the fact that anybody sub classing any `Host` will lose that sub class instance, when calling `clone()`. This may seriously impede future extensibility of the Soot framework.

2. Solution

We propose a generic solution consisting of two components. Firstly, we expose a set of four heuristic checkers that intend to warn a user whenever cloning in a particular Java class is implemented in a way that might impede software evolution. The heuristics have been implemented as an extension to the Eclipse Integrated Development Environment (IDE) [1]. Whenever a heuristic finds a violation of its rule, it attaches a warning to the resource in question. The user can then invoke a "quick fix" to fix the violation by one of two means depending on the type of warning. (see section 2.2)

[1]Eclipse project `http://www.eclipse.org/`

2.1. Heuristics

In the following we describe the four heuristics in detail. Each heuristic is implemented as a visitor that walks the abstract syntax tree of each compilation unit in the change set of an incremental (or full) compilation in Eclipse.

Consistent class annotation In Java, a class that implements `clone()` most certainly does so because it wants to provide the functionality of cloning objects of that class. Consequently, this class should implement the `Cloneable` marker interface. On the other hand, any class that implements `Cloneable` should provide a non-default implementation of `clone()` at least in one of its super classes.

Hence, this checker issues a warning whenever a class (a) is concrete (not abstract), implements `Cloneable` but only inherits the `clone()` implementation of `Object`, or (b) it declares `clone()` itself but does not implement the `Cloneable` interface.

The warning message given reflects the type of problem detected. The available quick fix resolutions are (a) generating an implementation of `clone()` or (b) making the declaring type implement the `Cloneable` interface.

Returning null During our initial investigation, we found that frequently implementations of `clone()` actually return **`null`** because cloning of those types is not actually supported. As is known from Software Engineering research, returning **`null`** can lead to null-pointer exceptions occurring far from the original error location. Causes of such exceptions are consequently hard to track and fix, especially if the contract of the defined method (here `Object.clone()`) implies that a non-null pointer be returned.

Hence, this checker looks for occurrences of the statement **`return null`**`;` in the body of each `clone()` method.

If an occurrence is found, we actually issue a warning message. The associated quick fix allows to generate a correct implementation of `clone()`. The warning suggests that the method, by contract, can also throw a `CloneNotSupportedException`. No quick fix for generating such an implementation is provided at this time.

Not calling the super class As mentioned in the introduction, Soot suffers from the problem of not calling **super**`.clone()` in order to construct the actual clone object. Instead it relies on the correct implementation of copy-constructors. This causes multiple problems. (1) Copy constructors have to be created and maintained. (2) A copy constructor has to be implemented on a class `C` even if `C` does not add any fields to its super class. (3) If a class `C` is sub classed by a class `S` and `S` calls **super**`.clone()`, an instance of `C` is returned, which can lead to all sorts of errors and violates the contract of `clone()`.

Hence, the checker searches the body of each `clone()` method for calls to **super**`.clone()`. If none is found, a warning is issued. The associated quick fix allows to generate a correct implementation of `clone()`.

Use of Java 5 co-variant return types From Java 5 onwards, methods can have co-variant return types. For the particular example of the method `clone()`, it means that it can return subtypes of `Object` and in particular, if defined in a class `C`, it should use return type `C` to avoid casting on the client-side.

Hence, to Java 5-enabled projects, we apply a checker that flags implementations of `clone()` which have a return type different from the type of their declaring class. The flag we create is of type "info" instead of "warning" because this conversion does not really change the behaviour of the program, just its style. The associated quick fix allows to generate an implementation of `clone()` with co-variant return type.

2.2. Quick fixes

As mentioned above, each of the four heuristics provides a set of "quick fixes" to quickly fix the detected problem at hand. In the current implementation we provide two different fixes. One makes a class implement the `Cloneable` interface, while the other one generates an implementation of the `clone()` method. Generally, one could think of further fixes, like making a `clone()` method throw a `CloneNotSupportedException` instead of returning **`null`** but we found that those solutions can easily be coded by hand and also that the problems they solve do not actually occur that often.

While the quick fix for adding the `Cloneable` interface is straightforward, generation of a correct `clone()` method is not an easy task at all and hence we wish to discuss it in more detail. (Note that our implementation allows to generate `clone()` methods also with no warning being present, simply by a menu item in the context menu of an arbitrary source type.)

Generation of `clone()` When designing the user interface for the generation of the `clone()` method, we followed a lot the currently existing support for generating `hashCode()` and `equals(..)` methods within Eclipse because both methods share quite some concepts. In particular, all three implementations depend on the available types of fields and have to call the super class in order to compute their final result. Also, the available implementation for `hashCode()` and `equals(..)` provided best practises for proper integration with the Eclipse IDE.

Compared to generating `hashCode()` or `equals(..)` methods, a complete solution to the problem of generating a `clone()` method is hard. The problem is that not every class implements the `clone()` method, while for `hashCode()` and `equals(..)` this is always the case. Because of this fact, one cannot always assume that `clone()` can be called on any type of instance field. Also, there might be different kinds of clones desirable. Some applications might require a shallow copy where only field references are copied, others might require deep copies where the entire contents of all instance fields are copied (recursively).

At the latest when it comes to deep copies, a general solution is virtually impossible. Creating a deep copy in general requires that all field types and all field types of those types are (again, recursively) cloneable. Furthermore, all those implementations of `clone()` must consistently create deep copies. Virtually all collection classes of the Java runtime library violate this second property; when calling `clone()`, they create shallow copies. Other classes in the runtime library are not cloneable at all, like `String` or `StringBuffer`. A general solution would hence have to generate specialized code for cloning collections and for using copy constructors for non-cloneable classes (on a case-by-case basis). A series of articles by Kreft and Langer [4, 5, 6] (German) give a very detailed assessment of those problems and possible solution strategies. As they show, a general solution even has to make use of reflection in certain situations.

For the scope of this work, we decided for an easier strategy which solves most of the problem but might still leave some manual work to the programmer for exceptional cases. When opting to generate a clone method, our extension presents the user the dialog shown in Figure 1. On the top the user can select fields which should be deeply copied. We allow the creation of deep copies of instance-fields which are of a reference type (the notion of deep copies makes no sense for primitive types) that implements the `Cloneable`

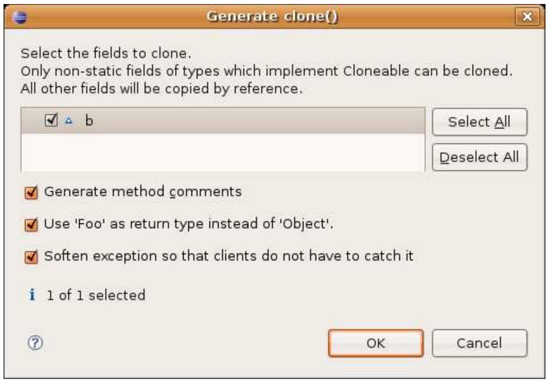

Figure 1. Options dialog for code generation

interface. [2]

Furthermore, the dialog exposes options for automatically generating method comments (as defined in the Eclipse preferences or project properties), using the co-variant return type (shown in Java 5-enabled projects only) and softening the `CloneNotSupportedException` thrown when calling **`super`**`.clone()`. Through softening, clients of this class do not have to catch this exception again and again. Using softening `CloneNotSupportedException` is common and good practise if used at places where it is known to be safe and can for example be found in many places inside the Java runtime library, e.g. the class `LinkedList`:

```
try {
  clone = (LinkedList<E>) super.clone();
} catch (CloneNotSupportedException e) {
  throw new InternalError();
}
```

The option to (not) soften exceptions is shown only if the super type declares this exception in its interface. In cases where `CloneNotSupportedException` is not declared, it *has to be* softened in order to adhere to this interface. Consequently, in such situations, the option on the dialog is replaced by an appropriate hint that softening will be forced if necessary.

3. Validation

In order to validate the feasibility of our approach, first we applied the four heuristics to the entire Soot code base (as of revision 2665). Then, based on the warning markers, we refactored Soot to implement cloning consistently, using the code generation features explained above.

[2] Note that here we assume that the field type actually creates a deep copy when `clone()` is called. This behaviour is not validated.

3.1. Results of applying the heuristics

Altogether, this Soot revision contained 240 non-abstract declarations of `clone()` methods which we had to to deal with. Out of those, the heuristic for not calling the super class reported 237 implementations not calling **`super`**`.clone()`. The heuristic for consistent class annotation reported that almost all of those types implementing `clone()` did not implement the `Cloneable` interface. In 14 other cases, the interface was implemented but not the `clone()` method. None of the implementations returned **`null`**. All implementations used `Object` as return type which was consistently reported by our heuristic for suggesting co-variant return types. This is because the Soot developers have just currently started to convert Soot to a Java 5 code base.

3.2. Effectiveness of code generation

We manually investigated all of the generated warnings and used the quick fix feature provided by our tool to find a better implementation that would eliminate the warning but not change the behaviour of the program (at least in combination with other changes of the same kind). The largest changes could be made in the packages that resemble nodes of abstract syntax trees for the various intermediate representations in Soot. Formerly, each single node class would implement `clone()` by calling a constructor and cloning the arguments recursively. This requires an implementation of `clone()` on every single node type. Interestingly, after generating a few standard implementations of `clone()` far up in the hierarchy, it turned out that most of the implementations of `clone()` in sub classes could be eliminated. This can always be done when a sub class declares no instance fields.

Altogether, we were able to remove 179 methods that way. Another 37 methods could be replaced by standard implementations we generated with the tool. Those were all either cases where a `clone()` method was necessary because the type did declare instance fields or because the super type of the type was `Object`, whose `clone()` method has only protected visibility. In those cases, an implementation of `clone()` can be used to expose the method to clients. In another 16 cases, we had to replace methods by non-standard implementations. In order to do so, we first generated a default implementation using our tool and then modified it to our needs. Most modifications boiled down to possible null-pointer checks (e.g. for linked lists) or deep copying of arrays or collections.

In one case, an abstract class for constants, we return **`this`** from `clone()`, although it violates the contract. This is because constants are immutable by definition and hence need not to be cloned, hence saving memory. In three cases, the `clone()` method had to be added to interfaces so that it could be called on types implementing that interface. We were happy to see that only few such additions had to be made. This was the fact because almost all types in Soot implement some relatively generic interface, such as `Value`. In another eleven cases, abstract definitions of `clone()` were replaced by concrete, generated implementations. This is because those methods had to be called by sub classes via **super**`.clone()` and Java does not type-check such calls to abstract methods.

In order to complete the implementation of cloning, we had to add another 26 standard and two non-standard implementations. In seven cases, we refined the return types of existing (correct, partially abstract) methods to the type of the declaring method. Eleven times we had to keep implementations of `clone()` as they were, because they did additional crucial work. In particular this is the case for types representing method bodies in Soot. When cloning bodies, one has to clone everything but local variables which then have to be cloned separately and patched up in a second step. In all those cases, we marked the respective methods with a `SuppressWarnings` annotation. (Currently, our Eclipse plug-in does not yet manage to actually suppress the warning but this will be solved in future versions.)

4. Related Work

The work mostly related to ours is the automatic generation of `equals(..)` and `hashCode()` methods in Eclipse. As mentioned above, opposed to the case of `clone()`, generating those methods is possible in a complete manner, since all types in Java do provide a public `equals(..)` and `hashCode()` method. Also the semantics of those methods is completely defined, while for `clone()` this is not the case (e.g. compare the notions of a deep or shallow copy).

The issue of whether returning **null** from a `clone()` that is not actually capable of cloning or throwing a `CloneNotSupportedException` instead boils down to whether or not to use the so-called "return code idiom". The work of Bruntink et al. [3] analyzes large-scale C programs using this idiom and shows that its use is very error-prone. We take this as a justification for our "return null" heuristic.

The effect of needing less implementations of `clone()` when calling **super**`.clone()` than when using copy-constructors can be explained by the power of virtual dispatch. If a sub class does not add any instance fields, it can reuse the implementation of `clone()` from its super class and virtual dispatch is the most natural form of code reuse in Java. While we that way exploit the natural Java semantics, related work on the topics of Traits [8], Mixins [1] and Virtual Classes [7] tries to maximize code reuse by using *different* forms of dispatch.

5. Conclusion

As we showed in this work, quite simple heuristics can be use to find flaws in the implementation of cloning in Java. Moreover, we were able to provide an Eclipse plug-in that generates a default implementation for `clone()` methods which was useful in almost all cases we investigated. Implementing cloning using code generated that way allowed us to safely eliminate more than 50% of all `clone()` methods, significantly alleviating the problem of code maintenance for Soot. All converted `clone()` methods use co-variant return types. We recommend the use of co-variant return types, as in one case this even revealed a bug: One of the `clone()` methods we converted was previously not even returning an object of the right type.

For future we plan to look into extending the code generation to be able to deal with deep copies of the Java collection classes and arrays. Our study revealed that such cases probably occur less often than one might think, however automated support might be useful for certain applications.

Despite the fact that our experiment went well and helped to support our claims, we came to the conclusion that actually real language support for cloning would be very desirable. For example, annotations could be used to state whether an instance field should be deeply cloned or not. The actual cloning could then be left entirely to the virtual machine. The same could hold for `equals()` and `hashCode()` methods which could be parametrized by the same annotations. We believe that the use of aspect-oriented programming could yield such automation, however, using current technologies, only by the use of reflection which comes at a huge runtime cost.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam - A Smooth Extension of Java with Mixins. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 154–178, London, UK, 2000. Springer-Verlag.

[2] E. Bodden. COMP 762 Project 1 report, February 2007.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 242–251, New York, NY, USA, 2006. ACM Press.

[4] K. Kreft and A. Langer. Das Kopieren von Objekten - Der Sinn und Zweck von clone(). *JavaSPEKTRUM*, September 2002.

[5] K. Kreft and A. Langer. Das Kopieren von Objekten - Prinzipien einer Implementierung von clone(). *JavaSPEKTRUM*, November 2002.

[6] K. Kreft and A. Langer. Das Kopieren von Objekten - Die CloneNotSupportedException. *JavaSPEKTRUM*, January 2003.

[7] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, 2005.

[8] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *European Conference on Object-Oriented Programming*, 2003.

[9] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.