The Impact of Programming Languages in Code Cloning

Jaime Filipe Jorge[1] and António Menezes Leitão[2]

1 **Instituto Superior Técnico**
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
jaime.f.jorge@ist.utl.pt
2 **Instituto Superior Técnico**
Av. Rovisco Pais 1, 1049-001 Lisboa, Portugal
antonio.menezes.leitao@ist.utl.pt

Abstract

Code cloning is a duplication of source code fragments that frequently occurs in large software systems. Although different studies exist that evidence cloning benefits, several others expose its harmfulness, specifically upon inconsistent clone management.

One important cause for the creation of software clones is the inherent abstraction capabilities and terseness of the programming language being used.

This paper focuses on the features of two different programming languages, namely Java and Scala, and studies how different language constructs can induce or reduce code cloning. This study was further developed using our tool Kamino which provided clone detection and concrete values.

1998 ACM Subject Classification K.6.3 Software Management

Keywords and phrases Clone Detection, Software Engineering, Programming Languages, Software Management

Digital Object Identifier 10.4230/OASIcs.SLATE.2012.107

1 Introduction

There are a number of bad smells in the process of writing software systems[1]. *Code cloning* is one of them. A code clone is a duplication of source code where programs are copied from one location and subsequently pasted onto other locations.

Studies found that code cloning is ubiquitous in software systems[2][3][4]. Some authors consider cloning to be harmful to software development [2][3][5][6]. A study[7] reports that in some systems half of the changes to code clone groups are inconsistent and that corrective changes following inconsistent changes are rare.

Its existence may be due to a number of reasons. Clones may be caused by programmer inexperience or programming methodology. In this paper we will focus in programming language constructs as causes for code cloning.

Subject to our study are two programming languages: Java and Scala. The Java programming language is one of the *de facto* standards for writing object-oriented systems. Its language constructs are result of years of best practices and it is still passing the test of time by having a large of companies using it. We will provide insights as to how these language constructs influence code cloning.

Scala is a more recent statically typed JVM language that provides different abstraction mechanisms, which are a result of learning from Java's wrong and right doings.

1st Symposium on Languages, Applications and Technologies (SLATE'12).
Editors: Alberto Simões, Ricardo Queirós, Daniela da Cruz; pp. 107–122

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

We provide clone analysis using these two programming languages and we will conclude how abstraction capabilities and verbosity/terseness influence code cloning. This study originated from the development of our tool, Kamino, which is a software clone evolution analyzer that provides developer with means to acknowledge clone duplication, observe and mitigate its consequences.

The rest of the document is structured as follows: section 2 describes software clones and programming language impact, section 3 describes a case study where we instantiate some of the Java problems, section 4 describes in detail some of the problematic situations which fuel software clones and how they can be mitigated using Scala, section 5 will map the scenarios described earlier to real values obtained using our tool Kamino and section 6 presents our conclusions.

2 Software Clones

A code clone is a duplication of source code in which fragments are copied from one location and subsequently pasted onto other locations, which can then be modified or not. Upon duplication, it is often difficult to differentiate the original fragments from duplicated ones, hence the two fragments are uniquely called clones.

Studies show that cloning is ubiquitous in software systems. For instance, Baker et al.[2] found that, on large systems, 13% to 20% of source code can be cloned code. Baxter et al.[3] reported 12.7% source code clones whereas Mayrand et al.[4] experienced 5% to 20%.

Well known software systems have been subject to clone study. In terms of their percentage of code cloning, GCC averages 8,7%[8], X Windows has 19%[2] whereas Linux and JDK have , respectively, 22,7% and 29%[9].

From the developers perspective, there are a number of factors which propel code cloning. On the one hand, we can think of limitations inherent to the programmer himself. For example, when he needs to work with a new system. Whether because it is a large system (which induces difficulty in mastering its domain) or the programmer is dealing with code which he does not own, both aspects tend to promote a kind of example-driven programming approach, in which developers copy fragments of existing functionality and modify them to solve the problem at hands, i.e., each developer's task. Logically, this methodology enables clone propagation.

In what regards programming methodology, there are also reasons which motivate cloning. Systems that rely heavily on generated code (generative programming), tend to have higher clone numbers, since they actually enforce a template which can be seen as a framework of a clone.

Programming languages can ultimately induce code redundancy. Less expressive languages promote code cloning by requiring greater verbosity and need for boilerplate in programming. This code verbosity may hinder the original developers intent. Other problem related to programming languages are lack of reusing mechanisms. Examples of such mechanisms are inheritance composition, generic functions, higher order functions, macros, type classes, etc. Upon nonexistence of such mechanisms, programmers are motivated to copy and paste code.

Software duplication has a function of cost, i.e. , implementing a refactored alternative to a cloned code fragment is not always clearly beneficial [10]. This may be caused by *difficulty in creating or understanding the refactored version* or by *unwanted size increase that it would imply in the system*. Both of these causes can be correlated to the original premise that programming languages may induce cloning namely because refactoring may be complex to

create/understand and may create larger versions of the original cloned fragments.

Hence, we can also create the parameters on which we can grade programming languages, based on their capabilities towards mitigating code cloning:

Abstraction Capabilities This criteria judges the idiomatic and simple nature of abstraction possibilities to given cloned fragments in the programming language itself. It may be possible to create such abstraction constructs but they may not be valuable in the cost function described above (namely in complexity in construction and understanding). These abstraction capabilities may include method extractions, inheritance, mixins, high-order functions or anonymous inner classes, duck typing, etc.

Language Verbosity This criteria evaluates the size of a refactored alternative to cloning as well as the overall boilerplate syntax required to implement a given feature. These map to the cost function described earlier: if a developer prefers cloning to a refactored version that is larger in size, then one can infer that the language is proliferating code cloning by being verbose.

To instantiate these concerns, we now present a case study of code cloning in which we observe problematic situations of lack of abstraction and a surplus of verbosity.

3 Case Study

To motivate the study of programming languages as fueling agents for software clones we now provide a case study of a mature Java project. Containing approximately 7000 Java classes, with a grand total of 775921 LoC. The project was subject to a clone detection experiment, using our tool Kamino, which analyzed several hundred revisions and provided feedback on possible inconsistent cloning situations.

We used a clone detector named Simian[1] to provide information about clone detection. Simian is a text based clone detector that provides fast execution and some level of token insensitivity (i.e. mark two code fragments as equal even though they present different text tokens).

More specifically, Simian found 57672 duplicate lines in 5230 blocks in 2031 files and processed a total of 385622 significant (775921 raw) lines in 7136 files. Dividing the duplicated lines by the total lines we 7.4%. Furthermore, if we divide the number of classes that contain duplicated code by the total number of Java classes, we obtain 28.5%. Hence, our study using Simian found that, 7.4% of all lines codes are duplicated and that 28.5%

Program 1 shows two classes where code duplication is evident. We can observe that there are two major types of similarities: (1) Classes have the same amount of duplicate fragments with the only difference being the type of the object in consideration (`ChangeCanProc` on the left, `TransferCanProc` on the right); (2) if we compare the methods (of either right and left programs) `getValidCanProc()` and `getValidExtCanProc()`, we can observe that the only difference resides in the `if` condition: one has the negation of the other's boolean expression. We can also observe that there is a pattern involving a `for` cycle, an `if` condition and a method being called inside the body of the `if`.

[1] `http://www.harukizaemon.com/simian/`

Listing 1 Case Study: Two fragments of code (left and right) existing in two different Java classes.

```
    if (degree == null) {
       return Collections.emptyList();
    }

    List<ChangeCanProc> result=
new ArrayList<ChangeCanProc>();
    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
    final ChangeCanProc child =
        (ChangeCanProc) process;
        if (child.isCandidacyValid() &&
        child.hasCandidacyForSelectedDegree(
              degree)) {
            result.add(child);
        }
    }
    return result;
}

public Map<Degree, SortedSet<ChangeCanProc>>
     getValidCanProc() {
    Map<Degree, SortedSet<ChangeCanProc>> result =
new TreeMap<Degree, SortedSet<ChangeCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
        final ChangeCanProc child =
        (ChangeCanProc) process;
        if(child.isCandidacyValid() &&
     !child.getCandidacyPrecedentDegreeInformation()
                .isExternal()) {
            addCandidacy(result, child);
         }
     }

    return result;
}

public Map<Degree, SortedSet<ChangeCanProc>>
     getValidExtCanProc() {
    final Map<Degree, SortedSet<ChangeCanProc>> result
= new TreeMap<Degree, SortedSet<ChangeCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final CandidacyProcess process
          : getChildProcessesSet()) {
        final ChangeCanProc child =
              (ChangeCanProc) process;
        if (child.isCandidacyValid() &&
     child.getCandidacyPrecedentDegreeInformation()
            .isExternal()) {
           addCandidacy(result, child);
        }
    }

    return result;
}

private void addCandidacy(final Map<Degree,
          SortedSet<ChangeCanProc>> result,
      final ChangeCanProc process) {

    SortedSet<ChangeCanProc> values =
    result.get(process.getCandidacySelectedDegree());
    if (values == null) {
      result.put(process.getCandidacySelectedDegree(),
               values = new
             TreeSet<ChangeCanProc>(
              ChangeCanProc
              .COMPARATOR_BY_CANDIDACY_PERSON));
    }
    values.add(process);
```

```
    if (degree == null) {
       return Collections.emptyList();
    }

    List<TransferCanProc> result=
new ArrayList<TransferCanProc>();
    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
    final TransferCanProc child =
        (TransferCanProc) process;
        if (child.isCandidacyValid() &&
        child.hasCandidacyForSelectedDegree(
              degree)) {
            result.add(child);
        }
    }
    return result;
}

public Map<Degree, SortedSet<TransferCanProc>>
     getValidCanProc() {
    Map<Degree, SortedSet<TransferCanProc>> result =
new TreeMap<Degree, SortedSet<TransferCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final IndividualCandidacyProcess process
         : getChildProcessesSet()) {
        final TransferCanProc child =
        (TransferCanProc) process;
        if(child.isCandidacyValid() &&
     !child.getCandidacyPrecedentDegreeInformation()
                .isExternal()) {
            addCandidacy(result, child);
         }
     }

    return result;
}

public Map<Degree, SortedSet<TransferCanProc>>
     getValidExtCanProc() {
  final Map<Degree, SortedSet<TransferCanProc>> result
= new TreeMap<Degree, SortedSet<TransferCanProc>>(
    Degree.COMPARATOR_BY_NAME_AND_ID);

    for (final CandidacyProcess process
          : getChildProcessesSet()) {
        final TransferCanProc child =
              (TransferCanProc) process;
        if (child.isCandidacyValid() &&
     child.getCandidacyPrecedentDegreeInformation()
            .isExternal()) {
           addCandidacy(result, child);
        }
    }

    return result;
}

private void addCandidacy(final Map<Degree,
          SortedSet<TransferCanProc>> result,
      final TransferCanProc process) {

    SortedSet<TransferCanProc> values =
    result.get(process.getCandidacySelectedDegree());
    if (values == null) {
     result.put(process.getCandidacySelectedDegree(),
                values = new
              TreeSet<TransferCanProc>(
               TransferCanProc
                .COMPARATOR_BY_CANDIDACY_PERSON));
    }
    values.add(process);
```

Observing the examples, one might ask: is this set of clones a result from the inexperience of developers or is it influenced by programming language deficiencies? We will enlighten this question in the next section where we will describe limitations inherent to the Java programming language and how Scala tackles them.

4 Abstraction Capability Comparison Between Java and Scala

Java is one of the *de facto* programming languages for object-oriented development. Java combines the syntax of C++ with the semantics of classic object-oriented languages such as Smalltalk. The language had a significant investment in the development of virtual machine and compiler research that allowed it to reach mainstream and widespread audience and effectively penetrate business development.

Scala[11] is a statically typed programming language that runs on JVM and blends pure object orientation with functional programming. It is fully interoperable with Java but provides functional aspects such as function literals (lambdas), closures, functions as first class values, currying and partial function application. Scala provides an union of different features from experimental programming languages (such as Pizza[12] and Java's generics) and industry strength programming languages (such as Java's and C# syntax, Smalltalk's uniform object model, ML's functional approach, Haskell's implicits, Lisp's flexible syntax and Ruby and Python's functional approach with object-oriented design).

The next sections will cover the earlier mentioned topics of language abstraction and verbosity. For each, we will try to observe both language's constructs.

4.1 Abstraction Capabilities

We define abstraction capability as the ability of a developer to lift patterns out of his code and place them in a single location where they can be reused. We will provide several abstraction examples that avoid code duplication.

4.1.1 Method Extraction

As motivation, we will provide a programming scenario inspired in the real example presented in listing 1. In our example, a student (described in listing 2) is characterized by an id, a name, an age and a list of courses he's attending. Bean style selectors and modifiers are left out of the Student class for example sake, but they should be assumed as already defined.

Listing 2 Student class.

```
public class Student {
    private int id;
    private String name;
    private int age;
    private ArrayList<String> courses = new ArrayList<String>();
}
```

Assuming we were given a collection of students one could ask for the students with id lower than a given number. We could write method that implemented this functionality (described in the left side of listing 3) This is a common pattern in Java: given a list of objects, each object that satisfies a given criteria is inserted in a container that is returned in the end. The corresponding code is on the left side of listing 3.

We can observe (in the right side of listing 3) an equal rewrite, using non idiomatic use of the language, of the same program in Scala. Notable differences include method declaration (using `def`), type declaration information (after argument declaration), type inference in variable declaration, `for` comprehension of *foreach* (using `<-`) and non obligation of `return` statement.

Now, if in another point in our program we intended to extract the students that are younger than a given age, we would have two approaches: duplicate the code and change the comparison predicate or; refactor the code to provide a more general method that filters students based on their id, age or any other property. As we saw in section 3, this was one of the causes for code duplication since the developer chose the former, i.e., duplicating the method and executing minor changes. For argument sake we will choose the latter.

Listing 3 Naive filter in Java (left side) and in Scala (right side).

```
public ArrayList<Student> filter(
              List<Student> students,
              int id){
    List<Student> resultingStudents
       = new ArrayList<Student>();
    for(Student student : students)
        if (student.getId() <= id)
            resultingStudents.add(student);
    return resultingStudents;
}
```

```
def filter(students:List[Student],
           id:Int):ListBuffer[Student]={

    var resultingStudents = ListBuffer[Student]()

    for(student <- students)
        if (student.getId() <= id)
            resultingStudents.append(student)
    resultingStudents
}
```

Listing 4 General method extraction approach.

```
public static <T> Collection<T> filter(
    Collection<T> target,
    ComparePredicate<T> predicate) {
            Collection<T> result =
                   new ArrayList<T>();
            for (T element: target) {
                if (predicate.receive(element)) {
                    result.add(element);
                }
            }
            return result;
        }

public interface ComparePredicate<T> {
    boolean receive(T type);
}

// Client code
final String name = "Jack";
Util.filter(students,
            new ComparePredicate<Student>(){
    public boolean receive(Student student){
         return (student
           .getName().compareTo(name) == 0);
    }
});
```

```
// With function passing
def getStudents(students:List[Student],
        studentSelector:Student => Boolean)={
    for (student <- students;
         returningStudent = student
          if studentSelector(student)
        ) yield student
}
// Client Code:
getStudents(students, student:Student =>
                   student.name == "Jack")
```

```
// Alternative
// Get students using library functions:
// Client Code:
students.filter(_.name == "Jack")
```

A general and abstracted approach in Java requires interfaces, type parametrization and anonymous inner classes (as depicted in the left side of listing 4). This approach works for all Student's properties (provided that the `ComparePredicate` is implemented) but also for all types of Classes.

This alternative is complex and programmers, unless writing libraries or frameworks, do not embark in such generalization (proved by the case study). This example of filtering a list can be extended to various other scenarios, such as: finding the max/min of a list, grouping or splitting elements of a list; reducing the list into a value; etc, each of which should be implemented with each corresponding interface and generalized static method.

This constitutes a burden and a price which most developers are not willing to pay, thus ending up in the original duplication scenario.

In terms of code cloning this means that the developer is left with a deficient function passing mechanism, evidenced by the lack of full function passing mechanisms which does not correctly mitigate the practice of code duplication. There are a number of functional libraries (such as guava-libraries[2], functionaljava[3] and lambdaj[4]) to compensate this limitation in Java.

The most general Java approach to filter a collection of objects can be achieved in Scala as shown in the top right side of listing 4). As we can observe, there is no return type in the method declaration (it is inferred) and the `for` expression is capable of iterating, filtering and `yield`ing. The biggest difference, however, is the function type declaration, which describes a function that receives a `Student` and returns a Boolean. Client code needs only to pass a function value. Notice that the variable `id` inside the function value needs not be final (it can be calculated elsewhere), thus allowing closures to be created.

However, it is possible to achieve terser code. We can observe the last code fragment (on the lower right side of listing 4), which depicts a call to a method inside the List collection: `filter`. Scala was developed to ensure that use cases such as filtering a collection was included inside collection objects. For this reason, a developer is able to filter, find an element, find a max/min element, iterate, map, reduce or group over collections, thus enabling code reduction and mitigating duplication.

4.1.2 Interfaces and Traits

Java's interfaces were originally introduced to enhance Java's single inheritance model[13], much like *protocol*s in Objective C.

Interfaces constitute a contract to a client, so that one can call a given set of functionality, but also constitute a form of polymorphism on a object that implements several interfaces (i.e. can be treated as the type that it implements).

Interfaces implement no functionality nor do they include any variables: they only declare constant values and method signatures that need to be implemented on the classes that implement the interface.

This means that interfaces are considered a poor man's reuse mechanism. They do not factor code nor do they generalize implementation. This, in turn, means that abstracting functionality through inheritance in Java always implies lifting that functionality to a super (regular or abstract) class or delegating that functionality to third party objects which will allow future composition.

From a code duplication perspective, a developer that found a code fragment to be a candidate for a refactoring would lift such code fragment to a super class, making it available to all its sub classes. But what if sub classes do not have a common super class? Take, for instance, an example situation: there is a `Cat` and `Dog` class that have a duplicated method `move`. To reduce duplication, a developer refactors method `move` to their common super class: `Animal`. However, if a `Car` class exists which requires the same method `move`, it does not make sense to include it in the class hierarchy that extends from `Animal`. The developer is forced to allow code duplication in `Car` to to avoid an inconsistent class hierarchy. Hence

[2] http://code.google.com/p/guava-libraries/
[3] http://functionaljava.org/
[4] http://code.google.com/p/lambdaj/

the Java inheritance system forces the programmer to choose between a latent trade off: reusability or coherence. The preferred property is generally the latter, which implies a greater number of code duplication in Java projects.

Scala does not have interfaces, at least not by that terminology. Instead, Scala took inspiration from Ruby's mixins and created Traits, which are similar to Java's interfaces with the difference of Traits allowing method implementation. This way, a class can coherently extend a super class and then mixin a number of traits to insert additional functionality. This is a form of multiple inheritance that avoids the some of problems found in C++, namely diamond problem. Hence, observing the previous example, a Trait containing the method `move` could be created, allowing `Cat`, `Dog` and `Car` to mixin the functionality without disturbing class hierarchies. Thus, coherence and reusability can be achieved using Traits which allow for concern separation, modular design and code reuse.

4.2 Language Verbosity

Programming language verbosity can hinder code comprehension and semantics since the programmer needs to observe a number of keywords and sentences to grasp underlying functionality/behavior. From another point of view, however, certain language formalities can also serve a documentation purpose. Best practices and formalizations can serve the role of documentation as they normalize code to a common standard.

We shall analyze Java's formalizations/best practices from the perspective of describing the intent of the programmer and compare it with a Scala alternative.

4.2.1 Getters and Setters

Listing 5 Getters and setters in Java (left side) and Scala (right side).

```
/**
 * Student Class in Java defining
 * an id, a name and the getters/setters
 * for both atributtes
 */
public class Student{
    private int id;
    private String name;

    public int getId(){
        return id;
    }

    public void setId(int id){
        this.id = id;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }
}
```

```
/**
 * Student class in Scala also defining
 * an id, a name and implicit getters/
 * setters for each attribute
 */
class Student(var id:Int,var name:String)
```

A first example of code that might add unintentional complexity to a given program is Java's *getters* and *setters*. In the Java convention, a class should have its properties declared as `private` and expose a pair of methods that are public getters and setters for those properties. This is so common in Java that Java editors (such as eclipse or netbeans) automatically generate these methods for a given set of properties. They can be harmful since they can hide potential methods that describe functionality. Furthermore, for any

given programming task, these methods could in fact be skipped of code review as best practices state that they do not contain any application functionality or business logic. One can argue that their existence might pollute the code base since the added documentation value is residual.

Scala provides implicit generation of getters and setters for properties of a class. One is still able to create accessors but the language reduces boilerplate by generating what Java defines as best practice.

To create getters, one should mark the attribute with the `val` keyword which creates a method to access the property, e.g. `object.attribute`

To create setters, one should mark the attribute with the `var` keyword. This not only creates the public access method but also creates a public method to set the value with a new value: `example.attribute = newAttribute`.

There is not a direct symmetry to the Java's getter and setter since in Scala the names for both are equal to the attributes they represent, i.e., the getter and setter for a given attribute are the attributes name instead of post-fixing the verb *get* like in Java.

listing 5 depicts the differences of creating a class with two attributes and creating the getters and setters for those same properties. The code reduction achieved in the Scala version loses the Java's documentation purposes but gains more terse and readable code. We left out the constructor of the *Student* in the Java's version since we were concentrated in studying the accessors. However, Scala also includes a constructor with the same argument list as the one provided in the definition of the class.

4.2.2 Anonymous Inner Classes

Listing 6 Sorting using anonymous inner function (left side) and Scala function literals (right side).

```
sort(students,
     new Comparator<Student>(){
  public int compare(Student student1,
                     Student student2){
       // logic of comparing students
    }
});
```

```
// Function literal
(studentA:Student,studentB:Student)
 => /*logic of comparing students*/

// Function literal with type inference
(studentA,studentB) => /*logic*/

// Function literal with placeholders
_ <= _
```

As was seen in the previous section, anonymous inner classes provide an emulation of first class functions, limited by the fact that Java does not currently support full closure behavior. First class functions enable for code reuse by allowing the programmer to inject behavior into another function which can freely choose when and where to evaluate that code. In Java, to enable the creation of functions, one must create an anonymous inner class which has to implement a given contract known by client code. The creation of a anonymous inner class involves the implementation of a method inside a class which, using this paper's terminology, constitutes two code regions.

If there is no contract (i.e. interface or abstract class) for a anonymous inner class to implement, these must be defined. For instance, if we want to define a predicate interface (such as we did in listing 4), we must define a name for that class and for the method that it has to implement. These constitute at least two code regions that can subject to code duplication and also difficult the reading and comprehension of code.

Fortunately there are cases where the Java language already provides such interfaces. *Comparator* is an example of such interface.

As depicted in listing 6 (on the left side), client code is creating a `Comparator` anonymous inner class to pass it to a static sorting function in `Collections`. As we can see, the code that matters is represented by the comment, but it has to be surrounded by a method declaration and class instantiation.

Scala has first class functions, which is a good replacement of anonymous inner classes. In listing 6 (on the right side) there are three types of lambdas.

The first one is a function that receives two arguments of type `Student` and applies a comparison logic. This can be passed as a function value to other functions. However, if we know the types of the arguments to apply to that lambda, one could use type inference to add them, as depicted in the second lambda form. This can be even more concise by removing the intermediate naming by using placeholder syntax ('_'), which is a substitute for variable naming. This is effective upon function calling such as `List(1,2,3,4,5).reduceRight(_ + _)` which sums all the elements of a list and returns 15.

Scala's function literals and values are more concise than Java's anonymous inner classes, which translates into less verbose code.

4.2.3 Constructor Madness

Java allows for the creation of overloaded constructors and methods which provide a way to create optional parameters. The drawback is that this can create a multitude of different combinations (which must be codified in each constructor). Furthermore, upon extension of a given class, which provides overloaded constructors, the sub class must itself define the constructors to enable the propagation of optional parameters. This is called constructor madness [14] (as depicted in the left side of listing 7), since the code is polluted with code duplication of combinations of constructors, which makes inheritance more difficult and reduces comprehension/readability.

Scala also suffered from this problem. However, since version 2.8, Scala introduced named and default parameters[15] which can be applied to class constructors as well as methods. Given a class constructor, one can define optional parameters by defining a default value for that parameter (as evidenced in class `A` of program 7). This allows for omission of parameters that have a default value upon invocation of constructors/methods. However, this may cause ambiguity upon value assignment to a given property, i.e., in a list of parameters that have default values, if one wishes to invoke a method/constructor that assigns a value to the last parameter, one must include values for all previous parameters. Thus, to remove this restriction, named values were created to selectively indicate the name of the property to be initialized. This allows for increased readability, code conciseness and is less error prone upon method/constructor invocation.

Listing 7 Constructor madness in Java and default and named parameters in Scala.

```
class A {
    public A(int x,int y,int z){
        /*...*/
    }
    public A(){
        this(1,2,3)
    }
    public A(int x){
        this(x,2,3)
    }
    public A(int x,int y){
        this(x,y,3)
    }
}
```

```
class A(x:Int=1, y:Int=2, z:Int=3)
```

5 Code Duplication Implications

In the previous section we observed a number of situations where two programming languages provide different abstraction capabilities. In this section we will map each previous situation to a code cloning scenario using our own clone detector: Kamino.

For each abstraction mechanisms, we will analyze the source code regarding clone detection. This way we will obtain data for judging how one language and, more important, how a language's constructs influence and induce code cloning. This section will also provide values to materialize the discussion provided in the previous sections. For the required clone analysis we will use Kamino, which will be described in the following section.

5.1 The Clone Detector: Kamino

Kamino is a tool for clone evolution analyzer. A clone, as described earlier, is a copied and pasted fragment of code that originates naturally in every code base. By introducing time in the process of clone detection, we allow the cloned code fragments to evolve. Evolving cloned fragments are guided by evolutionary patterns[16] which describe how a cloned fragment changed from one point in time to another. Using these patterns, one can create a genealogy of a cloned fragment. One of the possible evolutionary traits, and also the most harmful, is the inconsistent change. This pattern happens when two previously cloned fragments are no longer similar due to some modification in one of them. The modification is inconsistent because it should have been applied to both instead of just one. If the modification is a bug fix, then there is a code fragment in the code base that will have to be fixed. Naturally, if a system has more cloned code then it can be subject to more inconsistent changes.

Kamino also provides a framework for dealing with different clone detectors and using them to extract clone evolution information.

Figure 1 described Kamino's architecture. As we can observe, four main processes exist: (1) the main Kamino process, which mediates the others and incorporates the mechanisms and algorithms described in our approach; (2) the clone detector, which contains the interface for dealing with the primary clone detectors; (3) the meta clone detector, which describes the Kamino unique approach to clone evolution extraction, and; (4) the repository process, which enables a common interface to different repository types. The Database connection is managed solely by the main Kamino process.

Kamino effectively creates clone genealogies for a given system. The main objective of our tool is to build evolutionary lines of cloned fragments with the purpose of providing insights to code but also warning the programmer when clones are modified inconsistently. For this purpose, Kamino includes two main modules: the clone detector module and the clone evolution inference module. For our analysis we will use the clone detector module of our tool.

Given the scope of this paper, Kamino will incorporate its own clone detector. This clone detector parses the whole code into blocks, where a block is defined as a region of code beginning with '*{*' and ending with '*}*'. There may be nested code regions inside code regions and multiple code regions inside a given source code file. After this pre-processing, each code region is hashed into a value, resulting into a flat list data structure of all the hashed code regions existing in the code. Finally, Kamino will group each region by hash equality, where two regions with the same hash are said to be clones of each other. The clone detection approach in the following scenarios will be the same: equal code regions (i.e. code between curly brackets) are considered as clones. Furthermore, Kamino employs a filtering process: if a clone region is a subset of another clone region (i.e. the line range

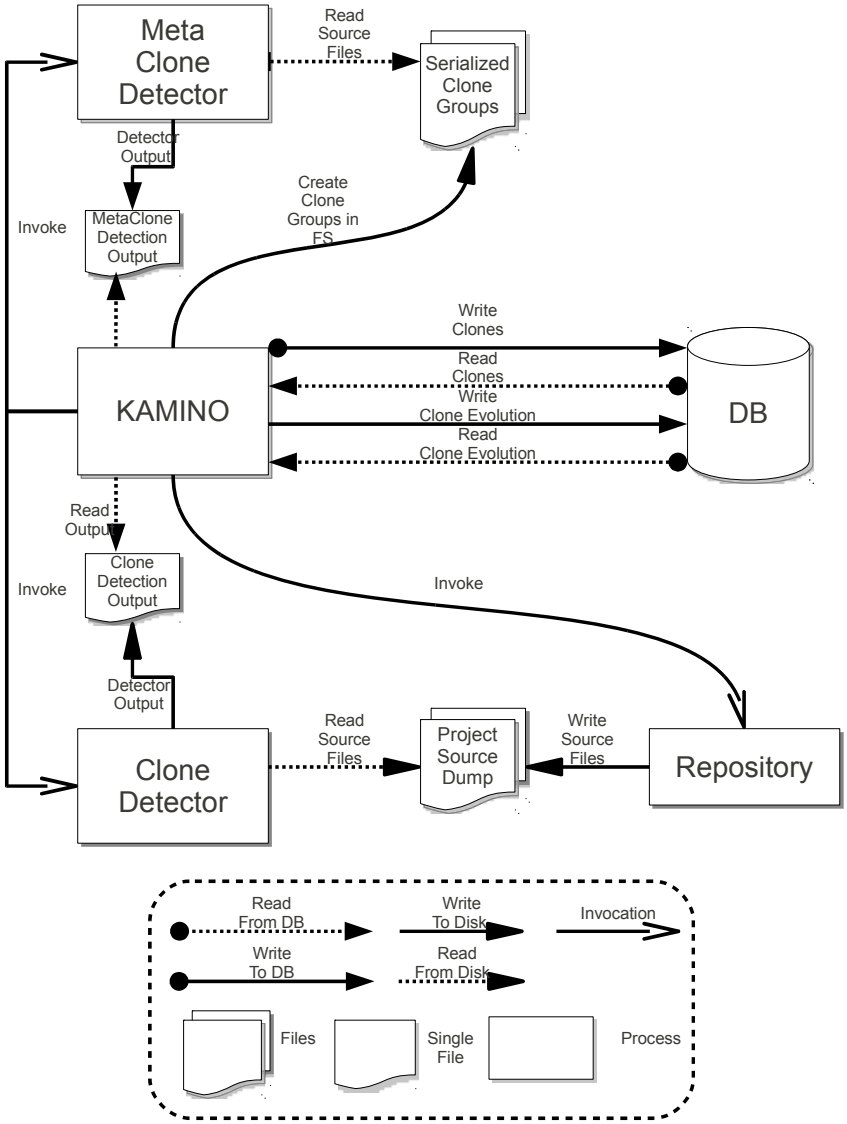

Figure 1 Kamino architecture. Four main processes exist: the main Kamino process, which mediates the others and incorporates the mechanisms and algorithms described in our approach; the clone detector, which contains the interface for dealing with the primary clone detectors; the meta clone detector, which describes the Kamino unique approach to clone evolution extraction, and; the repository process, which enables a common interface to different repository types. The Database connection is managed solely by the main Kamino process.

is within a line range of another clone) then the larger clone is considered and the smaller clone is filtered.

Experiments have showed that there are clones associated with the languages constructs described in the previous section. Throughout this section we will detail these experiments and provide concrete clone data.

5.2 Method Extraction

As we have seen in the final part of section 4.1.1, by having first class functions we can greatly reduce and refactor code upon method extraction. But how does this translate into code cloning?

As we have stated, our code clone is based on the equality between code regions. A general approach would be to detect clones in the worst case scenario: whole file duplication. In a situation where one duplicates all the code in the listing 4, i.e., copy the whole Java program into a file and copy the whole Scala program into another file, the clone detector returns **3** duplicated code regions in the Java program where it returns **1** in the Scala program. Inner block regions are not considered since Kamino filters clones that are subset of other clones, i.e., the larger clone is always selected.

Because Scala has first class functions, there is no need to create an interface nor the client code needs to create an anonymous inner class. This translates into a reduction in the number of code regions. Furthermore, if we only consider the third code fragment in the Scala program, there would be no code clones detected. As we have stated this is a typical programming pattern in Scala and Java which means that there is a high probability of finding code very similar to this in every code base. Thus, in this case, our clone detector finds that there is a **3:1** clone ration in the introduction of first class function in method extraction.

5.3 Interfaces and Traits

As we have seen in section 4.1.2, Java's interfaces help in the design of a system (by defining implementation contracts) but do not guarantee code reuse (because class that implement a given interface must implement the methods defined by that interface, notwithstanding of having the exact same implementation). One could create a scenario where a single interface is defined and two classes implement its methods equally, i.e. have the same implementation for the methods required by that interface. Instantiating for an interface with one method, there would be (at least) **1** clone. In the Scala counterpart, i.e. using traits, there would be no clones since all the code is refactored into one place which can then be mixed in, hence no code duplication. Thus, traits successfully remove code duplication caused by Java's interface.

5.4 Getters and Setters

As referenced in section 4.2.1, Java's best practices require the programmer to provide two methods for each private attribute: a getter and a setter. In terms of code cloning, this would mean two more code regions for each attribute that can be detected as equal in another class. This may constitute two problems.

The first problem is the confusion created to the clone detector. If there are regions of code in which the programmer has no interest in knowing whether they are duplicated or not, then the clone detector should remove them. They can cloud other clones that are

much more meaningful in the eyes of the programmer. This makes the clone detector much more complex for it has to incorporate these filtering rules.

Furthermore, a second problem exists. If a programmer adds more logic to a getter/setter than the best practices advise, then, if we instruct the clone detector to ignore such regions, there may be false negatives (i.e. clones that the clone detector fails to detect) in the results of the tool. Kamino has the premise: *it's better to have false positives (which can further be filtered when showing results to the user) then false negatives.*

If a programmer introduces an attribute with the same name in a different class and provides a getter and a setter with equal implementation (which there is a high probability of since they are guided by best practices), there would be **2** code regions marked as clones.

In the Scala counterpart, the getters and setters are effectively generated, which leaves the source file without the extra two code regions to be marked as clones. Thus, in respect to getters and setters, Java has **2** possible clones where Scala nullifies this problem.

5.5 Anonymous Inner Classes

We already saw how anonymous inner classes versus first class function translate into code cloning in a previous example. However, in the previous scenario, one considered the whole program. Hence the anonymous inner classes clones were removed since they represented subset clones (i.e. clone regions that are inside other greater clone regions).

If we consider the examples showed in section 4.2.2, we could verify that for each anonymous inner class (implementing one method) a function could be written in Scala without any code region whatsoever. One could argue that, because Scala also has anonymous inner classes, the number of code regions should be same. However, in the example we considered, the use of anonymous inner classes is a way of achieving lambdas, i.e. functions defined in place that can be returned or received as arguments. As such, function literals in Scala are considered.

In conclusion: if a programmer duplicates code using the creation of an anonymous inner class (which, because this is often client code, there is a high probability of), there will be at least **1** clone region that can be marked as clone of another code region in another location of a software project. In Scala there is no such code region to be marked as clone, hence there are effectively no code duplication.

5.6 Constructor Madness

Our final example of language limitation regarding verbosity is in section 4.2.3 and is related to the constructor madness problem in Java. When trying to achieve default parameters (i.e. parameters that have a default value if not provided) in the constructor of a given class in Java, one must provide concrete implementations for all of the combinations of default parameters. Observing the code regions inside each overloaded constructor call in the Java listing 7, a programmer could classify all the similar calls to *this* as clones. Since our clone detector is based on equal code regions, it would not detect these code regions as clones.

However, in the worst case scenario (if one duplicates the whole source file, changes the class name and adds an attribute), there would effectively be **4** equal code regions marked as clones. We are being specific to constructor calls but methods have the same problem: the only way to achieve default parameters is through overloading. This causes more code that can be copied and pasted into subsequent locations.

Scala has default parameters which reduce the number of code regions and nullify the constructor madness problem. If we simplify this scenario to one default parameter (requir-

ing one constructor overloading of the constructor), one could obtain **1** clone (if we do not count the primary constructor) whereas in Scala there would be **0** clones.

6 Conclusion

We studied programming languages in the perspective of code cloning, specifically in the Java and Scala programming languages.

We exemplified a real case study where we could observe several code duplication problems which have a high probability of being caused by language verbosity and abstraction mechanisms. We observed abstraction capabilities, namely: anonymous inner classes versus first class functions; interfaces versus traits and; language verbosity (getters/setters, anonymous inner classes verbosity and constructor madness).

We concluded that cloned code is more prone to be created when a language is more verbose (since there is a higher probability of identical code) and also when the effort of refactorization is higher than the simple duplication of source code (in which abstraction mechanisms are of great influence).

We applied our tool, Kamino, to provide clone detection analysis in the comparison of both languages, where we observed that more terse and compact language constructs leave less room for clones. Evidence showed that the more expressive a language is lesser confusion is created in the clone detector and the number of clones drops significantly.

We postulate that a tool like Kamino could be useful in less expressive/more verbose languages where a high volume of cloned fragments can found (and possibly managed) and where language limitations induce the programmer to copy and paste source code to ease the task of software development.

As future work, we would like to study other abstraction mechanisms, such as type classes in haskell, and how software architectures influence the origin of cloned code.

References

1 K. Beck and M. Fowler, "Bad smells in code," *Refactoring: Improving the design of existing code*, 1999.

2 B. Baker, "On finding duplication and near-duplication in large software systems," in *wcre*, p. 86, Published by the IEEE Computer Society, 1995.

3 I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *icsm*, p. 368, Published by the IEEE Computer Society, 1998.

4 J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proceedings of the International Conference on Software Maintenance*, vol. 96, pp. 244–253, 1996.

5 T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, pp. 654–670, 2002.

6 K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1, pp. 77–108, 1996.

7 J. Krinke, "A study of consistent and inconsistent changes to code clones," in *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pp. 170–178, IEEE, 2007.

8 S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *icsm*, p. 109, Published by the IEEE Computer Society, 1999.

9 Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Transactions on Software Engineering*, pp. 176–192, 2006.

10 C. Roy and J. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, p. 115, 2007.

11 M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "An overview of the scala programming language," tech. rep., Citeseer, 2004.

12 M. Odersky and P. Wadler, "Pizza into java: Translating theory into practice," in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 146–159, ACM, 1997.

13 J. Gosling and H. McGilton, "The java language environment," *Sun Microsystems Computer Company*, 1995.

14 A. Leitao, "The next 700 programming libraries," in *Proceedings of the 2007 International Lisp Conference*, p. 21, ACM, 2007.

15 L. Rytz and M. Odersky, "Named and default arguments for polymorphic object-oriented languages: a discussion on the design implemented in the scala language," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 2090–2095, ACM, 2010.

16 M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 187–196, ACM, 2005.