Migration from C++ to Java - Testing Problems and Solutions

Urich Breymann & Andreas Spillner
Migration from C++ to Java
Problems and solutions

Ulrich Breymann
Andreas Spillner

University of Applied Sciences Bremen
Germany

Phone: ++49 421 5905 –3425 / –3467
Fax: ++49 421 5905 –3412
{breymann,spillner}@informatik.hs-bremen.de
Agenda

• Introduction
• Important differences between C++ and Java (pitfalls, examples and recommendations):
  – Assignment operator
  – Equality operator
  – Immutable objects
  – Argument passing
  – General rules
  – Clean up
  – Polymorphic behaviour
• Check list for code inspection
• Conclusion
Introduction

- A strong understanding of object-oriented concepts is necessary.
- Java is more and more the language of choice for a lot of applications.
- The syntax is similar to that of C++, but much simpler, which makes learning and using it easy.

- This talk focuses on the following subjects:
  - Possible pitfalls in program understanding for developers with C++ experience, important for the coding and ‘developer test’-phase of a project.
  - What are the special issues when Java code is reviewed during testing by a C++ expert?
Assignment operator

Primitive data types

// Java and C++
int a; int b = 4;
a = b;
a = 5;              // b is not changed here
Assignment operator

*class objects* in Java

// Java
thisArticle = thatArticle;
   // Two objects of class Article

thisArticle.changeFont("Helvetica");
   // Changes thisArticle *and* thatArticle
Assignment operator

// Java, but C++ like behaviour of assignment

thisArticle = (Article)thatArticle.clone();
    // deep copy

thisArticle.changeFont("Helvetica");
    // Does not change thatArticle!
Assignment operator
Recommendation

• Assigning class objects in Java does not mean making a copy.

• It rather creates an alias for the right hand side object and passes the left hand side object directly to the garbage collector.

• You can focus on this problem by code inspection:
  – Find any assignment to class objects in the code and check the correct use.
  – You can also use a (self-written or off-the-shelf, if there is one) code analyser to list the alias names of all objects.
Equality operator

In C++, the equality-operator compares two objects and returns true if they are equal. What does that mean? For our purposes, we define here:

- Two objects are *identical*, if they cannot be distinguished.

- Two objects are *equal*, if they are of the same type and are in the same state with respect to *all or some selected properties*. 
Equality operator

// Java
Integer I1 = new Integer(77),
    I2 = new Integer(77);
int i1 = 3,
    i2 = 3;
Integer I3 = I2;

i1 == i2; // true
I1 == I2 // false!

I1.equals(I2); // true
I3 == I2 // true
I3.equals(I2); // true
Equality operator

Recommendation

• In Java, operator \(==\) yields false if equal, but not identical class objects are compared.

• It returns true only if the compared *references* have equal values, i.e. the underlying objects are identical.

• You can focus on this problem by code inspection:
  – Find any comparison between class objects with operator \(==\) in the code and check if they should be replaced by the `equals()`-method.
  – You can also use a code analyser to list all comparisons.
Immutable objects

// C++
const string s = "hello";
s.append(" world!");     // Error! s is const
s += " world!";          // Error! s is const

// Java
String s = new String("hello");

Java strings are immutable and there is no method to change a String object. But ...

s += " world!";  // This is ok for Java (equals s=s+" world!";)
System.out.println(s);  // Output: hello world
Immutable objects

Remember: $s$ is a reference, not the object! References to immutable objects are not immutable themselves. They can be made immutable:

```java
final String s = new String("hello");
s += " world!";                      // Error! s is immutable
```

But what about calling a state-changing method?
// Java
```java
final MyClass aMyClass = new MyClass();
aMyClass = anotherMyClass;
    // Error! Reference aMyClass is immutable
    aMyClass.setAttribute(17);
    // (Object attached to) aMyClass is changed!
```
Immutable objects

Only for primitive data types, `final` means the same as `const`. There are some possible workarounds in Java to make also class objects immutable. These workarounds are more or less clumsy and expensive in terms of runtime.

- State checking
- Proxy
- Common Baseclass
Immutable objects

Recommendation

- Trying to implement the C++ `const` for class objects in a Java program is a little bit complicated and therefore will probably introduce errors instead of preventing them.

- If a C++ program is to be rewritten in Java, we recommend just to ignore the `const` keyword for non-primitive data types.
Argument passing

// same syntax in Java and C++, but different meaning

void f( Window w ) {
    w.setTitle("new Title");
}
Argument passing

// Java (wrong example)
void fullFilename(String path, String filename,
                   String result) {
    // Error: caller's result is not modified, only a local copy of result:
    result = path + filename;
}

The (copied) reference of result is changed, not the object it refers to!
Correct example:
// Java (now ok, but different interface)
String fullFilename(String path, String filename) {
    return new String(path + filename);
}
Argument passing

Recommendation

• Be sure that the parameter passing mechanism is used correctly.

• Is there an assignment to a parameter within a method, it is only a assignment to the local copy of the parameter. It has no effect for the caller of the method, so there must be a special reason, which should be documented in a comment – otherwise it is wrong and the parameter should be declared as `final`.

• Use code inspection to find such cases.
General rules

C++ example  direct semantic Java equivalent

myClass x;   None, there are no automatic (stack) class
objects in Java (only references)

myClass* p = 0;  myClass p = null;

p = new myClass;  p = new myClass();

p->method1()  p.method1()
General rules

**C++ example**  **direct semantic Java equivalent**

**declaration:**

```cpp
void f(myClass)  void f(myClass)
```

**call:**

```cpp
f(*p)  f((myClass)p.clone())
```

**declaration:**

```java
void g(myClass*)  void g(myClass)
```

**call:**

```java
g(p)  g(p)
```

**declaration:**  None, there is no passing by reference in Java

```java
void g(myClass&)
```

**call:**

```java
g(x)
```
General rules

C++ example  direct semantic Java equivalent

myClass* const q  final myClass q = new myClass();
    = new myClass;

const myClass* q  None
    = new myClass;
General rules

Java example  direct semantic C++ equivalent

```java
String s = "hello";    const string* s
    = new string("hello");

s = "abc";  delete s;  s = new string("abc");
char c = s.charAt(0);  char c = s->at(0);
```
General rules
Recommendation

• C++ and Java look similar, but are different.

• Java is often regarded as "C++ without pointers," which obviously is not true.

• Java is more like "C++ with dynamic (=heap) class objects only".

• If your company switches from C++ to Java, special training of developers is a must.

• They have to know the semantic differences of both languages.
Clean up

// C++
{
    Object myObject;  // Constructor acquires resources
    myObject.doSomeThing();
}                     // Destructor is called here automatically
                      // Its task is to release resources

In Java, the finalize() method roughly corresponds to the destructor in C++, but it has to be called explicitly, and it is left to the JVM (Java Virtual Machine) when finalize() will be called.
Clean up

// Java
{
    Object myObject;       // Constructor acquires resources
    myObject.doSomeThing();

    // recommended add-on:
    myObject.cleanup();     // Release resources now!
    myObject = null;        // Release object for garbage collector
                            // and prevent
                            // further access (if more code is to follow)
}   // The Garbage collector can do the rest, when it wants to!
Clean up
Recommendation

- If the immediate release of resources (other than the memory of the object itself) is important, you have to do it by yourself.

- The Java Virtual Machine does not guarantee a defined moment for release.

- Use code inspection to find critical classes and places.
Polymorphic behaviour

- All methods with the exception of static methods are virtual in Java.
- Static methods are class methods.
- This is no problem for those C++ people who write in good style, i.e. they
  - never override non-virtual C++ methods and
  - always call static methods by a class name, not an object's name.
- The second is also our recommendation for Java
Check list for code inspection

- Assignment of class objects:
  Shall an alias (1) be created or a copy (2)?
  1) ok
  2) use clone()

- Comparison of class objects using ==:
  Check of identity (1) or equality (2) intended?
  1) ok
  2) use equals()
Check list for code inspection

• Does parameter passing conform to the ”General rules” above? If not, rethink and recode.

• Is there a local parameter on the left hand side of an assignment in a method? If yes: It has no effect for the caller of the method, so there must be a special reason, which should be documented in a comment, otherwise it is wrong.

• End of scope: Is an immediate release of resources required? If yes, use a special self-written method for release.
Check list for code inspection

• Is reproducing the C++ `const` in a Java program really necessary for class objects? If not, avoid introducing unnecessary code complexity.

• Are class methods only called by using the class name? If not, rethink and recode.
Conclusion

• There are syntactically identical elements in the programming languages C++ and Java, where Java uses reference semantic, but C++ employs value semantic.

• Developers have to know the semantic differences between these similar syntactic elements.

• The semantic of Java resembles the reference semantic of Smalltalk, not the value semantic of C++.

• On the pretended simple way from C++ to Java there lie some stumbling blocks. Overlooking them can lead to unforeseen errors.
Migration from C++ to Java
Problems and solutions

Ulrich Breymann and Andreas Spillner

Hochschule Bremen
University of Applied Sciences
Neustadtswall 30
D-28199 Bremen
Germany

Phone: ++49 421 5905 –3425 –3467
Fax: ++49 421 5905 –3412
{breymann,spillner}@informatik.hs-bremen.de
http://www.informatik.hs-bremen.de/~breym/
http://www.fbe.hs-bremen.de/spillner/

Abstract

More and more projects use Java as an object-oriented implementation language. Often there is already knowledge of C++ and because of the syntactical similarity of the programming languages migration seems to be simple. Sometimes this is not the case due to important conceptual differences between C++ and Java. Selected problems show the implications for program analysis, understanding and testing. Early detection of problems is desirable, so the paper provides a check-list for the test managers.

Key words:
software testing, object-oriented programming languages, C++, Java, migration
Introduction

The object-oriented programming language Java is more and more the language of choice for a lot of applications. Its platform independence and networking capability promise easier solutions than other programming languages for a broad range of problems. Java [GJS96] appeared at the right time and has a lot of capabilities also beyond the Internet. The syntax is similar to that of C++ [ISO98], but much simpler, which appears to make learning and using it easy.

Migration from C++ to Java has to consider a lot of different things. A strong understanding of object-oriented concepts is necessary, because procedural programs are not possible – an often occurring misuse of C++. Event-driven GUI development and development of distributed applications are main features of Java programs. An effective migration strategy has to include training and self-education, depending on the kind of Java users, for example software developers, software architects, analysts or project managers [SU99]. Our paper, however, focuses on the following subjects:

- Possible pitfalls in program understanding for developers with C++ experience, important for the coding and ‘developer test’-phase of a project.
- What are the special issues when Java code is reviewed during testing by a C++ expert?

Of course, if developers and reviewers have got sufficient cross-training, there will be no big problem. But have they? We used to teach C++ as the first programming language at our institution, the University of applied sciences Bremen, but switched to Java only recently. Many advanced students, having a sound knowledge of C++, now use Java in study projects, learning Java by themselves, reading a textbook, and starting to code quickly. We realize that they often make some special mistakes in Java. Therefore we consider the migration from C++ to Java not as easy as one might think at a first glance. We assume that this problem also occurs in the industrial practice, if software is written or re-written in Java by developers with C++ experience.

Some but not all of the difficulties arise, because a Java program is determined by a reference-semantic behaviour of objects like in Smalltalk, whereas the expectations of Basic/C/C++/Pascal-programmers are based on a value-semantic behaviour. These wrong expectations lead to errors and confusion. Examples demonstrate selected problems and we propose a few hints in order to facilitate the detection of errors.

Caught in the assignment act

Programmers familiar with other languages like Pascal, Basic or C++ approach Java assuming that it behaves in a similar way. So the statement \(A = B;\) is intended to copy object \(B\) to object \(A\) with the result of two equal objects. Furthermore, it is expected that a method call \(A\text{.method1}();\) has no effect on object \(B\). Primitive data types behave as known from C++:

```java
// Java and C++
int a; int b = 4;
a = b;
a = 5; // b is not changed here
```

However, this is not the case with class objects in Java:
Java
thisArticle = thatArticle;     // Two objects of class Article.
thisArticle.changeFont("Helvetica"); // Changes thisArticle and thatArticle!

The second statement changes also the font of thatArticle! The reason is of course, that assignment does not modify an object, but the reference to an object. The result of such an assignment is that there is *only one object*, which can be accessed via *two different names* (aliases).

If we want C++-like behaviour, we have to copy the right-hand-side object (this is also called ”deep copy” in contrast to the ”shallow copy” above):

Java, but C++ like behaviour of assignment
thisArticle = (Article)thatArticle.clone(); // deep copy
thisArticle.changeFont("Helvetica");        // Does not change thatArticle!

The cast is necessary because the type of clone() is Object, the top superclass which is inherited by all other classes. Cloning objects is discussed more in depth in [B97].

**Recommendation**
Assigning class objects in Java does not mean making a copy. Instead, it creates an alias for the right hand side object and passes the left hand side object directly to the garbage collector. You can focus on this problem by code inspection. Find any assignment to class objects in the code and check the correct use. You can also use a (self-written or off-the-shelf, if there is one) code analyzer to list the alias names of all objects.

**Equality is not equality**
In C++, the equality-operator compares two objects and returns true if they are equal. What does that mean? For our purposes, we define here:

- Two objects are *identical*, if they cannot be distinguished.
- Two objects are *equal*, if they are of the same type and are in the same state with respect to *all or some selected properties*. For example, two name-objects (Peggy Sue, Mrs) and (Peggy Sue, Dr) are considered equal for the purpose of sorting, when they are alphabetically listed and titles don’t play any role.

Identical objects are therefore always equal, but the reverse is not true. Applying these definitions to objects in Java or C++ programs, we declare those objects exactly occupying the same address space as identical, i.e. there is actually only one object, maybe with several names. Of course, these definitions depend on the context and may not hold if we think of persistent objects. Objects with different memory addresses are distinguishable (not identical) and can independently change their states. These objects may be equal (or not). In Java, the equality operator == checks *identity* for class objects and *equality* for primitive data types. Equality of class objects is checked by the equals() method as can be seen in the example:

// Java
Integer I1 = new Integer(77), // class objects
I2 = new Integer(77);

int i1 = 3,                        // primitive types!
i2 = 3;

Integer I3 = I2;                 // Create another reference for I2.

i1 == i2;           // true         (primitive type: not identical, but equal)
I1 == I2            // false!        (not identical)

I1.equals(I2);      // true         (equal)
I3 == I2            // true         (identical)
I3.equals(I2);      // true         (equal)

This and some of the following points are discussed more in depth in [A98]. Of course, each author of a class has to write a special equals() method, if objects of this class shall be checkable for equality. If she does so, the method overrides the corresponding method of the top class Object, which is inherited automatically by all classes.

**Recommendation**

In Java, operator == yields false if equal, but not identical class objects are compared. It returns true only if the compared references have equal values, i.e. the underlying objects are identical. You can focus on this problem by code inspection. Find any comparison between class objects with operator == in the code and check if they should be replaced by the equals()-method. You can also use a code analyzer to list all comparisons.

**Constant problem or final confusion?**

If the const keyword is used properly in C++, it helps to create safer programs. The compiler will flag inadvertent incorrect uses of constant or immutable objects. In Java, the keyword final prevents changing an object after initialization. But final is not const!

Example:

    // C++
    const string s = "hello";
    s.append(" world!");          // Error! s is const
    s += " world!";               // Error! s is const

    // Java
    String s = new String("hello");

Java strings are immutable and there is no method to change a String object. But ...

    s += " world!";               // This is ok for Java (equals s = s + " world!");
    System.out.println(s);        // Output: hello world

Remember: s is a reference, not the object! References to immutable objects are not immutable themselves. They can be made immutable:
final String s = new String("hello");

s += " world!"; // Error! s is immutable

But what about calling a state-changing method? Have a look at:

// Java
final MyClass aMyClass = new MyClass();
aMyClass = anotherMyClass; // Error! Reference aMyClass is immutable
aMyClass.setAttribute(17); // (Object attached to) aMyClass is changed!

Only for primitive data types, final means the same as const. There are some possible workarounds in Java to make also class objects immutable. These workarounds are more or less clumsy and expensive in terms of runtime. Some of them are listed here:

**State checking**
Introduce a boolean attribute to the class and the corresponding question-method boolean isConst(). Each setter method has to throw an exception if isConst() yields true. Disadvantages: Only runtime check, careful programming necessary!

**Proxy**
This means to provide a surrogate or placeholder for the object [GHJV95, p.207 f]. The interface of the Proxy class contains only methods which do not modify the object.

**Common Baseclass**
Split your class X into three classes:

```java
public class XBase {...} /* contains private attributes and get-methods. 
   Only the constructor can set the attributes. */

public final class constX extends Xbase {...} /* inherited API + constX(XBase) 
   constructor */

public class X extends Xbase {...} // inherited API + set-methods
```

constX has to be final because otherwise we could derive a subclass with non-const behaviour.

**Recommendation**
Trying to implement the C++ const for class objects in a Java program is a little bit complicated and therefore will probably introduce errors instead of preventing them. If a C++ program is to be rewritten in Java, we recommend just to ignore the const keyword for non-primitive data types.

**Argument passing surprises**
Arguments including pointers can be passed a C++ method by value or by reference. Arguments are always passed by value in Java. Smalltalk users will not have a problem with this, but people coming from C++ have to realize that parameter passing does not work as they are used to, if objects of class type are involved.
Let’s have a look at some examples:

// same syntax in Java and C++, but different meaning
void f(Window w) {
    w.setTitle("new Title");
}

In C++, a function call f(aWindow) does not change the passed object aWindow, because a local copy is generated, whose method is called. At the end of the function, the local copy will be destroyed. (Of course, in C++ the above function doesn’t seem to make much sense.)

In Java, however, the window (aWindow) gets a new title, because a local copy of aWindow actually means a copy of a reference, i.e. passing a reference by value. The method setTitle() is called via this copy of a reference which refers to the very same object as before the call.

Another example:

// Java (wrong example)
void fullFilename(String path, String filename, String result) {
    // Error: caller's result is not modified, only a local copy of result:
    result = path + filename;
}

The (copied) reference of result is changed, not the object it refers to! Preventing a change of a local copy can be achieved using final, but this does not help here. Correct example:

// Java (now ok, but different interface)
String fullFilename(String path, String filename) {
    return new String(path + filename);
}

The valuable C++ property const is also helpful within methods where arguments are passed as a reference to a constant object (const&) for performance reasons. Unfortunately, there is no similar support in the Java language. Immutable objects need not be copied; therefore the runtime-expensive creation of objects is reduced in C++ with const&. If the caller wants to be sure that a passed object will not be modified, it has to be passed as a clone in Java. This does not, however, prevent the clone from being unnecessarily modified, in contrast to C++.

Examples:

// C++ method:
void f(const X& aX) {
    doThisAndThat();
    aX.setAttribute(1);  // statically detected error!
    doEvenMore();
}

call:
myObject.f(anotherX);
Java method:
void f(X aX) {
    doThisAndThat();
    aX.setAttribut(1);  // neither statically nor dynamically detected!
    doEvenMore();
}

call:
myObject.f((X)anotherX.clone()); /* At least the caller can be sure that the object
remains unmodified. */

Recommendation
You can only pass arguments by value in Java. Be sure that the parameter passing mechanism
is used correctly. Is there an assignment to a parameter within a method, it is only a assign-
ment to the local copy of the parameter. It has no effect for the caller of the method, so there
must be a special reason, which should be documented in a comment – otherwise it is wrong
and the parameter should be declared as final. Use code inspection to find such cases.

General rules
All the examples above are easily understood by C++ experienced people by following the
general rules:
1. Primitive data types like float, char, int etc. are treated equally in C++ and Java.
2. All Java objects of other types (including arrays) are nothing but C++ pointers in disguise.

The second point means that there are no automatic (i.e. execution stack) class objects in Java.
What is called a reference in Java, resembles more a C++ pointer than a C++ reference. References in C++ cannot be null or undefined, but pointers can. To show this in more detail, some
examples are listed in the following tables:

<table>
<thead>
<tr>
<th>C++ example</th>
<th>direct semantic Java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>myClass x;</td>
<td>None, there are no automatic (stack) class objects in Java (only references).</td>
</tr>
<tr>
<td>myClass* p = 0;</td>
<td>MyClass p = null;</td>
</tr>
<tr>
<td>p = new MyClass;</td>
<td>p = new MyClass();</td>
</tr>
<tr>
<td>p-&gt;method1()</td>
<td>p.method1()</td>
</tr>
<tr>
<td>declaration: void f(MyClass)</td>
<td>declaration: void f(MyClass)</td>
</tr>
<tr>
<td>call: f(*p)</td>
<td>call: f(MyClass)p.clone()</td>
</tr>
<tr>
<td>declaration: void g(MyClass*)</td>
<td>declaration: void g(MyClass)</td>
</tr>
<tr>
<td>call: g(p)</td>
<td>call: g(p)</td>
</tr>
<tr>
<td>declaration: void g(MyClass&amp;)</td>
<td>declaration: void g(MyClass)</td>
</tr>
</tbody>
</table>
| call: g(x)          | None, there is no passing by reference in Java. It can be simulated by a more circumstantial mecha-
|                    | nism, though. Encapsulation of a reference in a container may be such an indirect semantic equi-
|                    | valent (see below ”Simulating reference parameters in Java”).          |
| MyClass* const q = new MyClass; | final MyClass q = new MyClass();                                   |
| const pointer, very seldom used in |                                                                      |
**C++ example**

<table>
<thead>
<tr>
<th>C++</th>
<th>direct semantic Java equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++. Don’t confuse with pointer to <code>const!</code> (const myClass* t, see below)</td>
<td>None.</td>
</tr>
<tr>
<td>const myClass* q = new myClass;</td>
<td>None.</td>
</tr>
</tbody>
</table>

**Java example**

<table>
<thead>
<tr>
<th>Java example</th>
<th>direct semantic C++ equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>String s = &quot;hello&quot;;</td>
<td>const string* s = new string(&quot;hello&quot;);</td>
</tr>
<tr>
<td>s = &quot;abc&quot;;</td>
<td>delete s; s = new string(&quot;abc&quot;);</td>
</tr>
<tr>
<td>char c = s.charAt(0);</td>
<td>char c = s-&gt;at(0);</td>
</tr>
</tbody>
</table>

**Simulating reference parameters in Java**

Simulating reference parameters has nothing to do with our intention to show possible problems in program understanding during reviews of syntactically alike C++ structures in Java, but is given for completeness. In the function (already shown above)

```java
// Java (wrong example!)
void fullFilename(String path, String filename, String result) {
    result = path + filename;
}
```

the problem is: How to get *out* some information *via the parameter list*? There is no direct way to pass arguments by reference, but a little (not very elegant) trick will do: Just wrap the reference in a container, e.g. an array. The function then looks like

```java
void fullFilename(String path, String filename, String[] cont) {
    cont[0] = path + filename;
}
```

A possible usage is as follows:

```java
String[] container = new String[1];
fullFilename("thePath", "Filename", container);
String result = container[0]; // retrieve reference
System.out.println(result); // thePathFilename
```

**Recommendation**

C++ and Java look similar, but are different. Java is often regarded as ”C++ without pointers,” which obviously is not true. Actually, Java is more like ”C++ with dynamic (=heap) class objects only”. If your company switches from C++ to Java, special training of developers is a must. They have to know the semantic differences between these languages.

**Lazy clean up**

Big advantages of Java are (among others) that memory addresses cannot be taken and misused and that there is an automatic garbage collection. In C++, the destructor of an automatic object is called at the very moment when the object goes out of scope or when `delete` is called for a pointer referring to the object. Normally the destructor is used to release resources held by the object. Example:
// C++
{
    Object myObject; // Constructor acquires resources.
    myObject.doSomeThing();
} // Destructor is called here automatically. Its task is to release resources.

In Java, the finalize() method roughly corresponds to the destructor in C++, but it has to be called explicitly, and it is left to the JVM (Java Virtual Machine) when finalize() will be called, even if the JVM is explicitly asked to do so by calling System.runFinalization(). If resources (other than the memory of the object itself) are to be released a.s.a.p., the best is to write a special routine cleanup() for that purpose and call it directly before transferring the object to the garbage collector.

// Java
{
    Object myObject; // Constructor acquires resources.
    myObject.doSomeThing();

    // recommended add-on:
    myObject.cleanup(); // Release resources now!
    myObject = null; // Release object for garbage collector and prevent further access (if more code is to follow).
} // The Garbage collector can do the rest, when it wants to!

Recommendation
If the immediate release of resources (other than the memory of the object itself) is important, you have to do it by yourself. The Java Virtual Machine does not guarantee a defined moment for release. Use code inspection to find critical classes and places.

Polymorphic behaviour

All methods with the exception of static methods are virtual in Java. Static methods are class methods (not comparable with class methods in Smalltalk, which are always dynamically bound). This is no problem for those C++ people who write in good style, i.e. they
- never override non-virtual C++ methods and
- always call static methods by a class name, not an object’s name.
The second is also our recommendation for Java.

Implications for program analysis, understanding and testing

How can the problems described above be avoided or recognised and solved? The most important precondition is that developers have a good knowledge of Java. The assumption ”Java is a simpler C++” is wrong and leads to erroneous programming.

Besides a thorough training, the following ways may be suitable for finding errors:
reviews and inspections

tools for static code analysis

test methods

There are two cases to be considered for developers with C++ knowledge: new software is to be developed in Java, or software is to be ported from C++ to Java. In the second case, a direct (maybe automated) comparison between the sources is possible. Check-lists for code-inspections should be extended by the following questions:

- Assignment of class objects: Should an alias (1) be created or a copy (2)?
  1) ok
  2) use clone()
- Comparison of class objects using ==: Check of identity (1) or equality (2) intended?
  1) ok
  2) use equals()
- Does parameter passing conform to the ”General Rules” above?
  If not, rethink and recode.
  Is there a local parameter on the left hand side of an assignment in a method?
  If yes, it has no effect for the caller of the method, so there must be a special reason, which should be documented in a comment – otherwise it is wrong. The compiler will prevent this usage if the parameter is declared as final.
- End of scope: Is an immediate release of resources required?
  If yes, use a special self-written method for release.
- Is reproducing the C++ const in a Java program really necessary for class objects?
  If not, avoid introducing unnecessary code complexity.
- Are class methods only called by using the class name?
  If not, rethink and recode.

Many of these difficulties could be detected by a static code analyser [K95], which could print a list of possible cases.

We do not recommend special test methods and test cases because often the test engineer will not be able to decide if there is really an error. Creating an alias with an assignment may just be what is intended, and also a comparison of identical objects with different aliases can make sense.

More important is that software developers really know the semantic differences between similar syntactic elements of C++ and Java.

Conclusions

On the apparently ”simple” way from C++ to Java there lie some stumbling blocks. Overlooking them can lead to unforeseen errors. The semantics of Java resembles the reference semantics of Smalltalk, not the value semantics of C++. There are syntactically identical elements in the programming languages C++ and Java, where Java uses reference semantics, but C++ employs value semantics. Developers have to know the semantic differences between these similar syntactic elements. Also the destruction of objects behaves differently in both languages, and there is no parameter passing by reference in Java. The phrase ”passing an object by reference” is surely acceptable in a colloquial language usage between Java devel-
opers, but the actual, precise meaning is "passing a reference (referring to that object) by value". Developers new to Java have to know this too.

This article shows problems which can arise from these misunderstandings, and gives recommendations and a check-list as an aid to detect possible errors.

Bibliography

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley 1995
Dr. Breymann is currently a professor with the special focus on software engineering and object orientation. Before that he worked several years in industry as analyst and project leader, among others on the subjects process control systems for power plants and software for the international space station project. He was a member of the German working group for the standardization of the programming language C++.

Dr. Spillner, also professor at Hochschule Bremen, has over 20 years experience in software development and teaching. He received his PhD on the dynamic integration testing of modular systems from the University of Bremen in 1990. He has also worked as Associate Professor at the University of Bremen. Dr. Spillner is the chairman of the German Computer Society Special Interest Group for the Testing, Analysis and Verification of Software Systems (GI-TAV). His research interests include the validation of software, test methods (especially for large software systems) and testing of object-oriented software systems.