Abstract—Just like other software, Java profiles benefit from refactoring when they have been used and have evolved for some time. This paper presents a refactoring of the Real-Time Specification for Java (RTSJ) and the Safety Critical Java (SCJ) profile (JSR-302). It highlights core concepts and makes it a suitable foundation for the proposed levels of SCJ. The ongoing work of specifying the SCJ profile builds on subclassing of RTSJ. This spurred our interest in a refactoring approach. It starts by extracting the common kernel of the specifications in a core package, which defines interfaces only. It is then possible to refactor SCJ with its three levels and RTSJ in such a way that each profile is in a separate package. This refactoring results in cleaner class hierarchies with no superfluous methods, well defined SCJ levels, elimination of SCJ annotations like @SCJAllowed, thus making the profiles easier to comprehend and use for application developers and students.

I. INTRODUCTION

Java profiles declare classes and interfaces that embody concepts common for an application area. An application programmer uses the profile to ensure that a concrete implementation conforms to generally accepted construction principles and to get an application that will run on one of several profile implementations. Thus a profile contributes to efficient software development. However, as time passes some features of a profile turn out to be less useful (deprecated) and new features have to be added to cover new technology or there is a desire to specialize the profile to particular areas.

In our work on real-time Java profiles [10], [9], [1], we have observed these phenomena with the Real-Time Specification for Java (RTSJ) which was a first important step in specifying real-time systems in Java [2].

RTSJ was designed to meet the requirements to a Java profile for real-time applications. It extends Java in eight areas: scheduling, memory management, synchronization, asynchronous event handling, asynchronous transfer of control, asynchronous thread termination, physical memory access, and exceptions. However, this profile which in many ways included the “state of the art” as to real-time system development, was (and is) complex to use. Furthermore, RTSJ applications are difficult to analyse, and features like asynchronous transfer of control are hard to use [3].

This has subsequently resulted in different proposals specifying simpler real-time Java profiles, of which Ravenscar-Java [6], [5] and Safety Critical Java (SCJ) [11] are the most prominent.

The SCJ profile is aimed at safety-critical systems which require a safety-critical certification (like DO-178B [7] or ED-12B [8]). Different certification levels in e.g. DO-178B are reflected in SCJ which has three levels: Level 0, 1, and 2, of which Level 0 is the most restrictive. The SCJ profile is based on RTSJ, but with some constraints compared with RTSJ, e.g. the usage of dynamic memory allocation. The programming model is centered on missions, where a mission consists of a set of schedulable objects. At Level 0 only periodic event handlers are permitted; at Level 1, periodic and aperiodic event handlers are permitted; and at Level 2, no-heap real-time threads are permitted too.

Common to those proposals is that they build on RTSJ using subclassing. Generally the advantage of using a subclassing approach is first and foremost code and implementation platform reuse. However, as we show in this paper, RTSJ and SCJ with its three levels do not share implementation code.

The disadvantages of using a subclassing approach are among others:

- the class hierarchies become complicated
- the SCJ classes themselves should be simpler and thus have fewer methods, but because they are subclasses of RTSJ classes, they will automatically inherit the public methods from the RTSJ classes
- annotations are therefore introduced and used extensively to tell at which SCJ level a class or a method is allowed
- for application programmers who start real-time programming in Java, the transition to e.g. SCJ becomes unnecessarily complicated
- the same is the case when teaching IT-students real-time programming using the real-time Java profiles.

Therefore, in [1], we concluded that subclassing was not a good way to extend a specification profile, and we suggested that the implementation reuse was ensured by delegation. However, this still left the definitions of the profiles unrelated. Essentially, a class defines an is-a relation to superclasses, and subclassing should be a specialization; - in algebraic terms, a conservative extension that preserves all properties including implementations introduced in the superclass. A concept of ‘super-classing’ or generalization as suggested in [13] could be used to create smaller profiles. However, a super-class or generalization construct does not exist in Java. Furthermore, such a construct is primarily useful for creating a parent class with common or shared behaviour, and as mentioned, RTSJ and SCJ with its three levels do not share implementation code. What RTSJ and SCJ have in common is better described
through a can-do relation which is implementation independent through the interface concept in Java. The purpose of an interface is exactly to abstract from implementations, which also explains why they support multiple inheritance.

The difference is well illustrated by RTSJ, where for instance the concept of time is implemented by abstract and concrete classes. Presumably, here the RTSJ designers have thought in terms of common implementations of the various methods that manipulate time objects. On the other hand, objects that are scheduled are characterized by an interface, because very different objects (handlers, threads) have to do the same operations with different implementations.

In the latest SCJ Draft from July 2010 [12], interfaces are used to some extent, e.g. p.117, Figure 7.2, but without getting a simpler structure, because they are added to the class hierarchy of RTSJ. The same tendency is found in Figure 4 in [15], due to inconsistencies in the SCJ Draft. The result is even more complicated interface and class hierarchies than the RTSJ hierarchies, making it more difficult to grasp and use SCJ correctly, especially for newcomers. We return to this example in our refactoring.

With this observation we have been able to find a suitable refactoring of RTSJ such that it supports a clean SCJ definition. The refactoring principle is simple: We start specifying the methods common to SCJ and RTSJ, putting these in a separate real-time core package called rtcore. This specification is done entirely by means of Java interfaces. The details of this process are elaborated in Section II.

This common behaviour in rtcore, together with specific methods for SCJ Level0, Level1, Level2, and RTSJ, are next implemented in separate packages which are described in Section III.

The contributions of this paper are thus: A refactoring of RTSJ and the SCJ levels according to the guidelines above, showing

1) a cleaner structure with clear and well defined levels
2) no changes of the semantics of SCJ and RTSJ
3) no use of @SCJAllowed annotations.

Existing applications can thus be compiled with the proposed profiles without any changes, and the resulting code can be run on existing compliant platforms. In the conclusion in Section IV we give a more detailed assessment of the result.

II. REFACTORING

Guidelines for refactoring are found in Fowler’s book [4], which also states the constraints on a refactoring: ‘When refactoring a software system, it has to be done in such a way that it does not alter the external behavior, yet improves its internal structure’. This is followed in the following refactoring of the SCJ and RTSJ profiles. We observe that SCJ is based on RTSJ and is specified as an extended subset of RTSJ. Furthermore SCJ has three compliance levels, Level0, Level1, and Level2, with the requirements, that any application implemented for a specific level must be able to run correctly on an implementation supporting a higher level.

A. Structuring the operations

To get an overview of the operations (methods of classes and interfaces) of all four profiles, SCJ Level0 (SCJ0), SCJ Level1 (SCJ1), SCJ Level2 (SCJ2), and RTSJ, let us look at all the operations of the four profiles. This universe is illustrated by a Venn diagram in Figure 1.

The universe has $2^4 - 1 = 15$ subsets, where

\[ \text{rtcore} = \text{scj0} \cap \text{scj1} \cap \text{scj2} \cap \text{rtsj} \]

This implies that rtcore specifies all the operations which SCJ0, SCJ1, SCJ2, and RTSJ have in common.

More details of the subsets are given in Table I. The numbers 1-15 correspond to the 15 subsets in Figure 1. As an example,

\[ \text{subset}1 = \text{scj0} \cap \text{scj1} \cap \text{scj2} \cap \text{rtsj} \]

and the crosses (\(\times\)) in all the four profiles mean that they have at least one method in common; e.g. MemoryArea.enter is visible in all four profiles and therefore belongs to subset 1. Likewise,

\[ \text{subset}3 = (\text{scj0} \cap \text{scj1} \cap \text{rtsj}) - \text{scj2} \]

with zeros (0), showing that this subset is empty.

As Table I shows, seven of the subsets in Figure 1 are empty, resulting in eight nonempty subsets which are illustrated in Figure 2.

From this analysis follows:

- SCJ1, \{1,2,9,10\}, is a proper subset of SCJ2, \{1,2,9,10,13,14\}, but SCJ0, \{1,2,8\}, is not a subset of SCJ1
- RTSJ, \{1,9,13,15\}, has operations common to all three SCJ-levels, \{1\}
- RTSJ has operations common with both SCJ1 and SCJ2, \{9\}, and also operations common with SCJ2 only, \{13\}. This is the basis for the refactoring.
TABLE I
OVERVIEW OF THE 15 SUBSETS OF OPERATIONS, INCLUDING EXAMPLES

<table>
<thead>
<tr>
<th>Set</th>
<th>Visibility of methods</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>scj0</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>scj1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>scj2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>rtsj</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

B. Refactoring principles

When refactoring RTSJ and SCJ the following principles have been followed:

- Operations common to RTSJ and SCJ0-SCJ2 are specified in a separate package called rtcore.
- rtcore is specified entirely as Java interfaces.
- The four profiles (SCJ0, SCJ1, SCJ2, and RTSJ) have their separate packages.
- The four profiles share operations in line with the illustration in Figure 2.
- None of the four profiles need to share implementation code.

This means that e.g. the SCJ0 profile will consist of (compare to Figure 2):

- the operations specified in rtcore, \{1\}
- some SCJ operations common to SCJ1 and SCJ2, \{2\}
- some SCJ0 specific operations, \{8\}.

C. Example

The hierarchy in Figure 3 illustrates the ideas in the refactoring through an abstract example.

The common method \(m\) is specified in the interface rtcore.IA. SCJ1 and SCJ2 have one more method in common, called \(m_{12}\), and RTSJ has a method called \(mrtsj\). All the A classes implement the methods specified in the interface IA which is the interface rtcore.IA or an extension of this interface.

The following test program can be executed, no matter which of the four packages is imported:

```java
import scj0::*; // or scj1, or scj2, or rtsj
public class TestA
{
    public static void main (String[] args) {
        A a = new A();
        a.m();
    }
}
```

And the following test example works only using package scj1 or scj2:

```java
import scj1::*; // or scj2
public class TestA12
{
    public static void main (String[] args) {
        A a = new A();
        a.m();
        a.m12();
    }
}
```
III. THE REFACTORIZING OUTCOME

The refactored specification of SCJ0-SCJ2 and RTSJ consists of the five packages shown in Figure 4. The content of those packages are described in the subsections below.

A. Package javax rtcore

The rtcore package plays a central role. Here is the specification of the common operations of the four profiles. The package consists of 12 interfaces with a total of 39 methods. The names from the corresponding RTSJ classes are left unchanged. We have only prefixed an I (the capital letter I) for the interfaces in rtcore, - except interface Schedulable which already is an interface in RTSJ (as the only interface).

The 12 interfaces are:
- IScheduler, Schedulable, IReleaseParameters, IPriorityParameters, IPeriodicParameters
- IAsyncEventHandler
- IMemoryArea, IPortal
- IHighResolutionTime, IAbsoluteTime, IRelativeTime
- IClock.

The specification of the methods in the interfaces are unchanged compared to RTSJ and SCJ.

As an example, the ReleaseParameters class occurs in both RTSJ and SCJ. In RTSJ, the class has ten methods; in SCJ Level0: no methods; in SCJ Level1 and Level2: three methods. Therefore the interface IReleaseParameters will be an empty interface in rtcore:

```java
public interface IReleaseParameters
{
  // empty
}
```

Similarly, the AsyncEventHandler class has two methods in common with both RTSJ and SCJ, resulting in the following interface in rtcore:

```java
public interface IAsyncEventHandler
extends Schedulable
{
  public void handleAsyncEvent();
  public void run();
}
```

B. The four profiles

Now, let us look at the implementation of the four profiles.

As mentioned in Subsection II-B:
- the four profiles have their separate packages,
- they share operations through interfaces,
- but do not share implementation code.

This implies that the class hierarchies for the four profiles are completely separated, each having their separate package, cf. Figure 4. In that way, only the necessary methods and classes are included in a specific profile.

Let us, as an example, look at the implementation of the IReleaseParameters interface described above. A class diagram is shown in Figure 5.

![Class diagram](image)

Fig. 5. class ReleaseParameters in SCJ0-SCJ2 and RTSJ.

First, the implementation of IReleaseParameters in SCJ0:

```java
package javax.scj0;

import javax.rtcore.IReleaseParameters;

public abstract class ReleaseParameters
  implements IReleaseParameters, Cloneable
{
  protected ReleaseParameters() {..}
  public Object clone() {..}
}
```

Next, let us look at SCJ1 with two new methods specified:

```java
package javax.scj1;

import javax.rtcore.IRelativeTime;
import javax.rtcore.IAsyncEventHandler;

public interface IReleaseParameters extends
  javax.rtcore.IReleaseParameters
{
  public IRelativeTime getDeadline();
  public IAsyncEventHandler
       getDeadlineMissHandler();
}
```
package javax.scj1;

import javax.rtcore.IRelativeTime;
import javax.rtcore.IAsyncEventHandler;

public abstract class ReleaseParameters
    implements IReleaseParameters
{
    protected ReleaseParameters(
        IRelativeTime deadline,
        IAsyncEventHandler missHandler) { .. }
    public Object clone() { .. }
    public IRelativeTime getDeadline() { .. }
    public IAsyncEventHandler
        getDeadlineMissHandler() { .. }
}

SCJ2 is like SCJ1 with no new methods. But RTSJ has seven more methods:
package javax.rtsj;

import javax.rtcore.IAsyncEventHandler;
public interface IReleaseParameters
    extends javax.scj2.IReleaseParameters
    {
    public IAsyncEventHandler
        getCostOverrunHandler();
    // six more methods are specified in RTSJ
    ..
    }
package javax.rtsj;
import javax.rtcore.IAsyncEventHandler;
import javax.rtcore.IRelativeTime;

public abstract class ReleaseParameters
    implements IReleaseParameters, Cloneable
{
    // two constructors
    // a total of 10 methods:
    public IAsyncEventHandler
        getCostOverrunHandler() { .. }
    ..
}

The implementation of the AsyncEventHandler looks similarly, see class diagram Figure 6.
Here, the AsyncEventHandler has the same methods in SCJ0-SCJ2, which are all specified in
rtcore.IAsyncEventHandler, but in RTSJ another 21 methods are specified in the rtsj.Schedulable
interface, so that all together the rtsj.AsyncEventHandler class has 32 methods and 7 constructors:
package javax.rtsj;

import javax.rtcore.IAsyncEventHandler;

public class AsyncEventHandler
    implements IAsyncEventHandler, Schedulable
{
    // seven constructors
    public void handleAsyncEvent () { .. }
    public void run () { .. }
    // thirty more methods:
    ..
}

The remaining classes in the profiles are implemented in the same way, see link in footnote 1
for the complete source code.

C. Example

With a much simplified structure of SCJ and with the specification of the common behaviour of the SCJ and RTSJ profiles in the package rtcore using interfaces, it is now possible to simplify some of the class hierarchies in SCJ. As an example of this simplification, let us look at the event handling hierarchy from the SCJ Draft [12], and the revised SCJ event handling hierarchy from [15], both mentioned in the Introduction and shown in Figure 7 and Figure 8.

This event handling hierarchy is a consequence of using implementation inheritance. By this, the AsyncEventHandler class with 32 methods is inherited through the BoundAsyncEventHandler class to the event handler classes in SCJ, see Figure 7. This results in some inconsistencies in SCJ because a bound asynchronous event handler is permanently bound to a dedicated real-time thread which is self-suspending, and in SCJ Level0 and Level1 the handlers cannot self-suspend. Wellings [15] has solved this inconsistency at the price of an even more complicated class hierarchy, see Figure 8.

Instead of retaining the event handling class hierarchy described above, simpler and more comprehensible hierarchies

1http://www.it-engineering.dk/HSO/index.html
can be constructed by following the ideas of "inheritance for specialization". This is the idea illustrated by the class diagram in Figure 9.

The problems in SCJ with self-suspending / not self-suspending event handlers, observed and solved by Wellings in [15], are solved in a much cleaner manner as follows:

- **SCJ Level0**: The periodic event handler is implemented according to its specification which tells that the handler is not self-suspending
- **SCJ Level1**: Similarly at this level for both periodic and aperiodic event handlers
- **SCJ Level2**: The handlers can self-suspend and are implemented in accordance with this specification.

This emerges clearly from Figure 9, in contrast to the class diagram in Figure 8.

The following example illustrates how a simple SCJ Level0 mission could be written. As described in the Introduction, one of the extensions in SCJ compared to RTSJ is the mission concept, where a SCJ application consists of one or more missions, and a mission consists of a bounded set of ManagedEventHandler objects. In SCJ Level0, only periodic event handlers are allowed, cf. Figure 9.

```java
import javax.scj0.*; // or scj1, or scj2

class AMission extends Mission {
    ManagedEventHandler[] eventHandlers;

    public AMission () {
        eventHandlers = new ManagedEventHandler[1];
    }

    public void initialize () {
```

Fig. 7. SCJ Handler Hierarchy, from [12].
Fig. 8. Revised SCJ Handler Hierarchy, from [15].

Fig. 9. Refactored Event Handler Hierarchy for SCJ and RTSJ.
class Periodic extends PeriodicEventHandler {
    static final int priority = 13;
    static final int nativeStackSize = 1000,
    javaStackSize = 1000;
    static final long storeSize = 10000,
    memSize = 10000;

    Periodic (RelativeTime start,
    RelativeTime period) {
        super (
            new PriorityParameters(priority),
            new PeriodicParameters(start, period),
            new StorageParameters (storeSize,
            nativeStackSize, javaStackSize),
            memSize);
    }

    public void handleAsyncEvent() {
        // the logic to be executed every period
    }
}

IV. CONCLUSION

The presented refactoring of the Real-Time Specification for Java (RTSJ) highlights core concepts and is a suitable foundation for the proposed levels of the Safety Critical Java (SCJ) profile (JSR-302). The specification of the common behaviour of the SCJ and RTSJ profiles in the package rtcore using interfaces, and the subsequent implementation of the profiles, show that it is possible to refactor the profiles without changing the semantics of them.

In a similar way, a mission for Level1 can be implemented containing both periodic and aperiodic event handlers. Such a mission can also be executed at Level2, but not at Level0.

REFERENCES


