Emulating the Java ME Platform on Java SE

Kenneth Russell
Tony Wyant

Sun Microsystems, Inc.

Session TS-2515

Goal of This Talk

Learn advanced uses of Java™ bytecode rewriting techniques to emulate one Java platform version on another

Agenda

Background and Motivation
The Technical Problem
Solution Space
Chosen Solution
How It Works
Automatic Rewriting Tasks
User-Driven Rewriting Tasks
Summary

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

Background and Motivation

- Java Platform, Micro Edition (Java ME) domain includes many specifications and profiles
  - Connected Limited Device Configuration (CLDC)
  - Connected Device Configuration (CDC)
  - Foundation Profile (FP)
  - Personal Basis Profile (PBP)
- Some overlap with Java Platform, Standard Edition (Java SE)
  - Consumer VM (CVM) and CDC stack targets set-top box and larger embedded market

Background and Motivation

- Combinations of some Java ME platform profiles (CDC/FP/PBP) are largely a subset of Java SE Platform APIs
- Basis for some higher-level specifications
  - Blu-Ray Java APIs (BD/J)
- Some licensees want to run Java SE platform under the hood to support Java ME platform applications
  - Reuse existing Java SE platform port
  - Higher performance
  - Example: Software Blu-Ray players on the desktop

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

The Technical Problem

- How to make one Java platform stack (Java SE platform) appear to a running application to be another one (Java ME platform)?

Technical Advantages

- Java ME platform profiles under consideration (CDC/FP/PBP) are largely straight subsets of Java SE platform
- Relatively minimal behavioral changes in specification
- Means that Java SE platform implementation could theoretically be reused largely unmodified
- Taking advantage of this property in our solution to the problem

Technical Disadvantages

- Java ME platform and Java SE platform source bases diverged long ago
  - Results in minor but annoying incompatibilities
- Targeted Java ME platform profiles (CDC 1.1) use different underlying Java platform (1.4) than desired Java SE platform version (5.0)
  - Most incompatibilities due to Java SE platform version differences
- Java ME platform TCKs tend to very strictly test the behavior of the platform
  - In some cases, overly strict tests
- Divergence of Java ME platform and Java SE platform TCKs
  - Similar to divergence of platforms' source bases

Desired Properties

- Avoid changing the underlying Java SE platform implementation at all costs
  - Treat it as a black box
  - Essential for making this work on non-Sun-supported platforms
- Try to support Java SE platform and emulated Java ME platform applications running in the same Virtual Machine for the Java platform (JVM™ implementation)
  - In different class loaders, for example

The terms “Java Virtual Machine” and “JVM” mean a Virtual Machine for the Java™ platform.

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

Solution Space

- Swap in older/different class libraries
  - Doesn't work for Java SE platform ports not done by Sun
- Use wrapper objects/classes somehow
  - Significant issues with getting the semantics right
  - Consider case of wrapping an EventListener and needing to add/remove the underlying one
  - Generally, involves huge global weak hash table
- JVM implementation changes to allow different class library versions to be loaded concurrently
  - Not compatible with desired property #1

Solution Space

- Java bytecodes are a very fluid medium
- Powerful, easy to understand, well-defined semantics
- Application (fortunately) can not introspect on its bytecodes; only on classes, etc. via reflection
  - Not possible to detect bytecode modifications during class loading e.g., in the TCK
- Would like to take advantage of mutability of bytecodes to solve this problem

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

Chosen Solution

- Rewrite incoming bytecodes of Java ME platform application at class load time
- Change application's use of certain core Java APIs
  - Make it appear that it's running on a different platform version and profile
- Similar binary rewriting techniques in use elsewhere in the software industry
  - Purify, VMWare, JDistro, JavaFlow, AspectJ
  - Apparently first time for Java platform emulation

Chosen Solution

- Java ME platform application thinks it is running on a compliant CDC implementation
- Requires no modifications to the Java SE platform implementation
- Theoretically allows Java ME platform and Java SE platform applications to run side-by-side in the same JVM implementation in different class loaders

Current TCK Status (CDC)

CDC TCK	Passed	Failed	Error	Not Run
JDK 5.0 [1]	~8900	~170	~15	~10
JDK 5.0 [2]	9072	79	0	10
JDK 1.4.2	9078	71	2	10
JDK 5.0 + Java ME Emulator	9151	0	0	10

- CDC TCK 1.1 (our initial failure rate)
- CDC TCK 1.1.2, corrected command line args and JVM machine config, up-to-date exclude list, fixed challenges

Source: Sun Microsystems, Inc.

Current TCK Status (PBP)

CDC TCK	Passed	Failed	Error	Not Run
JDK 5.0	892	48	0	3
JDK 5.0 + ME Emulator	939	1*	0	10

* Test bug, has been filed

- FP TCK work still underway; new TCK version due

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

How it Works

- Changes invocations of various methods, constructor calls, etc.
- Helper classes emulate behavioral differences between Java ME platform and Java SE platform
- Goal is complete invisibility to application

How it Works

- Implemented as **`java.lang.instrument`** agent
  - One Java Archive (JAR) file, pure Java code
- Agent uses open-source ASM toolkit for most bytecode rewriting tasks
  - **`http://asm.objectweb.org/`**
- Rewriting operations driven by configuration file
  - Very generic directives
  - Designed to be extensible without requiring source code of agent
  - Designed to support more emulation tasks than this one

Behavioral Changes

- Simply hiding inaccessible APIs comprises much of the work
- Need to handle (binary-compatible) situations
  - New superclasses introduced in Java SE platform
  - Methods moved up the class hierarchy in Java SE platform
  - Methods made abstract or final in Java ME platform, etc.
- Significant work needed to “fake-out” reflection and produce similar results even for error cases
  - Largely automatable

Behavioral Changes

- Must handle real changes in behavior
  - Between Java SE platform 1.4.x and 5.0
  - Between Java ME platform (CDC/FP/PBP) and Java SE platform
- Generally not possible to automate
- Handled on a case-by-case basis
  - Using generic bytecode rewriting rules
  - Call into helper classes/code written in Java platform

Examples of Behavioral Differences

- Unicode version differences and other, multi-byte character conversions
- Differing exceptions thrown from some methods (NullPointerException vs. IllegalArgumentException)
- BigDecimal formatting and other changes
- One java.awt.Frame per GraphicsDevice/screen
- java.awt.Component.setName() sends PropertyChangeEvent in Java SE 5.0 platform but not Java ME platform

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

Automatic Rewriting Tasks

- Baseline knowledge of emulated API set is driven by signature files from Java ME platform TCKs
- All core Java SE platform APIs outside this API set are automatically hidden from the application:
  `invokevirtual`
  `java/awt/Container.location()`
- Becomes
  `invokevirtual java/awt/Container.`
  `location_inaccessible_to_PBP()`
- Produces expected **`NoSuchMethodError`** with reasonably descriptive error message

Automatic Rewriting Tasks

- Similar inaccessibility rewrites done for fields and classes
- Constructors are a bit trickier
  - **`<init>`** method name known to verifier; cannot change it
- Inject call to run-time helper routine to throw **`NoSuchMethodError`**

Automatic Rewriting Tasks

- Certain methods are abstract or final in Java ME Platform APIs but not in Java SE platform
- Inject hook methods to manually throw **`AbstractMethodError`** if called
  - Rewrite all other bytecodes in class to call hook method
  - Works for further subclasses of these classes as well
- Attempted overrides of “final” methods detected
  - Not currently possible to throw exception from instrumentation agent
  - Inject attempted override of known final method to produce appropriate **`VerifyError`**

Automatic Rewriting Tasks

- Interpose on reflection to mirror knowledge of visible Java ME Platform APIs
  - Class.forName(), Class.getDeclaredMethods(), …
- Prevent application from reflectively using APIs which are not in the target platform
  - Filter out classes, methods, and fields not present
- Handle cases like methods moving up the class hierarchy
  - …and other binary-compatible changes
  - Interpose on e.g. Method.getDeclaringClass()

Automatic Rewriting Tasks

- Example:
  `invokevirtual java/lang/Class`
  `getDeclaredMethods()`
- Becomes
  `invokestatic com/sun/sepbp/Runtime`
  `getDeclaredMethods()`
  - Filters out invisible methods
  - May register certain methods as requiring further interposition later
- Reflection interposition done using generic bytecode rewriting commands

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

User-Driven Rewriting Tasks

- Several directives available to drive rewrites of incoming bytecodes
- **`IncludeNamespace`**
  `ExcludeNamespace`
  `PreConstructorArgCheck`
  `PostConstructorThisCheck`
  `RewriteNonStaticMethodCall`
  `RewriteStaticMethodCall`
  `RewriteStaticFieldAccess`
  `RewriteSuperclass`
  `ReplaceClassDefinition`

User-Driven Rewriting Tasks

- Rewriting tasks specified in configuration file
  - Simple text format
  - Generally refer to associated helper classes
- Example usage (all one line)

```
RewriteNonStaticMethodCall
   java/lang/Class
   getDeclaredMethods
   *
   com/sun/sepbp/Reflection
   getDeclaredMethods
```

Configuration File Excerpt

```
# Concession for signature tests, which are in the
# com.sun namespace
IncludeNamespace com/sun/tdk/
# Interpose on all reflective operations on
# java.lang.Class
RewriteStaticMethodCall java/lang/Class forName * \
  com/sun/sepbp/Reflection forName
RewriteNonStaticMethodCall java/lang/Class getClasses * \
  com/sun/sepbp/Reflection getClasses
RewriteNonStaticMethodCall java/lang/Class getConstructor * \
  com/sun/sepbp/Reflection getConstructor
...
# (AWT-related) provide different behavior for
# GraphicsConfiguration.equals() on Windows
# NOTE: it is really unfortunate that we need to
# interpose on all such calls
RewriteNonStaticMethodCall java/lang/Object equals \
  (Ljava/lang/Object;)Z com/sun/sepbp/Runtime objectEquals
# (AWT-related) Interpose on getClass() calls in particular
# to pass the AWT Container serialization tests
RewriteNonStaticMethodCall java/lang/Object getClass \
  ()Ljava/lang/Class; com/sun/sepbp/Runtime objectGetClass
# Interpose on Component.setName so we can filter out its property change
# event and on the Frame constructors so we can prevent multiple ones
# from being created
RewriteSuperclass java/awt/Component com/sun/sepbp/ComponentInterposer
RewriteSuperclass java/awt/Container com/sun/sepbp/ContainerInterposer
RewriteSuperclass java/awt/Frame     com/sun/sepbp/FrameInterposer
```

Include/ExcludeNamespace

- Ordinarily rewriter skips classes in Sun-internal namespaces
  - java/, javax/, sun/, com/sun/
  - Partially a workaround for the fact that it doesn't yet work on a per-ClassLoader basis
- Example use: TCK's signature tests
  - IncludeNamespace com/sun/tdk/
- ExcludeNamespace only used to work around bugs in ASM or TCK provoked by bytecodes never seen in the real world

PreConstructorArgCheck

- Allows an outgoing argument to a constructor to be filtered through a helper method
  - **`PreConstructorArgCheck`**
    `java/math/BigDecimal`
    `(Ljava/math/BigInteger;I)V`
    `1`
    `com/sun/sepbp/BigDecimalFixer`
    `checkScale`
    `(I)I`
- Passes the outgoing int (scale) argument to the BigDecimal constructor to a helper method
  - Check for negative argument, raise exception
  - Compatibility difference between 1.4.x and 5.0

PostConstructorThisCheck

- Passes a newly-constructed argument to a helper method for fixup
  - **`PostConstructorThisCheck`**
    `java/util/Random`
    `()V`
    `com/sun/sepbp/RandomFixer`
    `fixRandomConstruction`
    `(Ljava/util/Random;)V`
- Changes seed for Random objects created with no-arg constructor to be based on current time rather than better seed value
  - Compatibility difference between 1.4.x and 5.0

Rewrite(Non)StaticMethodCall

- Rewrites the specified static or non-static method call into a call to a static helper method
  - **RewriteNonStaticMethodCall**
    java/lang/Class
    getMethods
    *****
    com/sun/sepbp/Reflection
    getMethods
- One of many reflection interposition rules

Rewrite(Non)StaticMethodCall

Issues with inheritance

- ```
  public class A {
    public void m() { println("A.m()"); }
  }

  public class B extends A {
    public void m() { println("B.m()"); }
  }
  ```
  ```

Rewrite(Non)StaticMethodCall

Issues with inheritance

- ```
  public class Test {
    public static void main(String[] a) {
      new B().m();
    }
  }

  public class InterposerHelper {
    public static void aM(A a) {
      println("InterposerHelper.aM()");
    }
  }

  RewriteNonStaticMethodCall
    A  m  *  InterposerHelper aM
  ```
  ```

Rewrite(Non)StaticMethodCall

- **One might expect InterposerHelper to be called in this case**
- Depending on whether class B is loaded at the time the Test class is being instrumented, call to B.m() may or may not be rewritten
  - Rewriter may not force additional class loading
  - Rewriter needs to understand class hierarchy and inheritance relationships
  - Does not appear to be possible to insert run-time checks in the bytecodes for this; resulting bytecodes don't verify (although they would execute correctly if they did)

Rewrite(Non)StaticMethodCall

- These directives work best when the target method holder is final
  - Essentially the only option in this case
- RewriteSuperclass directive is more general and solves this case more completely
  - Has a couple of other issues
  - To be described later

RewriteStaticFieldAccess

- Rewrites all accesses to a given static field to point to another one
  - **`RewriteStaticFieldAccess java/awt/event/WindowEvent WINDOW_LAST com/sun/sepbp/Runtime windowEventConstantsWindowLast`**
- Rarely needed
  - Only to pass a couple of handwritten TCK tests using the getfield bytecode on a static final field
  - Would be compiled/inlined away by javac in a real application

RewriteSuperclass

- Most general rewrite rule
  - Supports functionality of PreConstructorArgCheck, PostConstructorThisCheck, RewriteNonStaticMethodCall
- Two basic operations
  - For all subclasses of a given class, rewrite their superclass to be another one
  - For all **`new`** bytecodes referencing a given class, rewrite them and any constructor invocations to refer to another class

RewriteSuperclass

- Example: handling one Frame per screen as well as setName() not sending PropertyChangeEvent

RewriteSuperclass

- **`RewriteSuperclass`**
  `java/awt/Frame`
  `com/sun/sepbp/FrameInterposer`
- Inheritance hierarchy changes from

```
java.awt.Frame
      ^
(potential application subclass)
```

- To

```
java.awt.Frame
      ^
com.sun.sepbp.FrameInterposer
      ^
(potential application subclass)
```

RewriteSuperclass

```
public class FrameInterposer extends Frame {
   public FrameInterposer() {
     super();
     Runtime.checkFrameConstruction(this);
   }

   public FrameInterposer(
       GraphicsConfiguration gc) {
     super(gc);
     Runtime.checkFrameConstruction(this);
   }

   public void setName(String name) {
     // Do not call super.setName()
     Runtime.componentSetName(this, name);
   }
   // ... more ...
 }
```

RewriteSuperclass

- New instances of Frame created by the application create a FrameInterposer instead
  - Type compatible
  - Additional work in constructor and setName() method
- Application classes which subclass Frame now rewritten to subclass FrameInterposer instead
  - Same properties
  - Additional checks and modified functionality inserted
  - Works correctly even if e.g. setName() overridden

RewriteSuperclass

- Potential issues with serialization
  - FrameInterposer class likely not present on other side
  - Haven't experimented extensively, but believe deserialization should work (per serialization spec)
- On the principle of minimum perturbation, have minimized use of this directive in our CDC emulation
  - Although it's probably safe and certainly simpler and more correct in some situations
- More on this with ReplaceClassDefinition

ReplaceClassDefinition

- Allows completely different set of bytecodes to be substituted for a given class
  - e.g., revert it wholesale to a previous implementation
- Used only in situations of last resort
  - Only when impossible or extremely difficult to change behavior of a class using other techniques
  - Serialization
  - Any other place where creation of object of a given class is hidden inside the core Java libraries
- Barrier to hosting Java ME platform and Java SE platform applications in the same JVM machine

ReplaceClassDefinition

- At points during development, used to revert java.text.DecimalFormat and java.util.GregorianCalendar to 1.4.x states
  - DecimalFormat had serialization-related problems
  - GregorianCalendar had algorithmic changes
- Fortunately, TCK tests in these areas were declared invalid
- Not using this directive in current product

Agenda

Background and Motivation

The Technical Problem

Solution Space

Chosen Solution

How It Works

Automatic Rewriting Tasks

User-Driven Rewriting Tasks

Summary

Summary

- Have successfully bridged the API gap between desktop Java platform and Java platform for set-top boxes
  - Without changing the Java SE platform source code
  - Without requiring changes to the Java ME platform application
- Believe we have a viable solution for starting to build compliant Blu-Ray Java support on Java SE platform
- Technology under consideration for emulation of “previous” Java SE release on “current” release
- Aiming for open-source release in near future

Related Technical Session

- TS-1326, “Bytecode Manipulation Techniques for Dynamic Applications for the Java Virtual Machine”
  - Friday, May 11, 1:30 PM, North Meeting Room

Q&A

Emulating the Java ME Platform on Java SE

Kenneth Russell
Tony Wyant

Sun Microsystems, Inc.

Session TS-2515