BlackBerry.

DEVCON 2010

Layout, Configure and Build Strategies for BlackBerry Apps

Derek Konigsberg

COM12

September 29th 2010

LOVE working together.

Session Surveys

- Remember to complete your breakout session evaluation in one of two ways:
  - On your BlackBerry® smartphone – use the DEVCON 2010 mobile guide, from Pyxis
  - Login to My Scheduler at one of the Cyber Zone locations or on your PC

Introduction

- About me
  - Derek Konigsberg – dkonigsberg@logicprobe.org
  - Desktop Java developer by day
  - Mobile Java developer by night
- BlackBerry platform involvement
  - Active member of the community, having presented on both BlackBerry Java Development and Open-Source Software
  - Best known for the “LogicMail “ standalone E-Mail client application
    - http://logicmail.sf.net/
    - The reason I got started developing for BlackBerry
    - Has become a popular alternative to BES/BIS offerings
    - Supports OS 4.2 through OS 6.0 from a single source tree

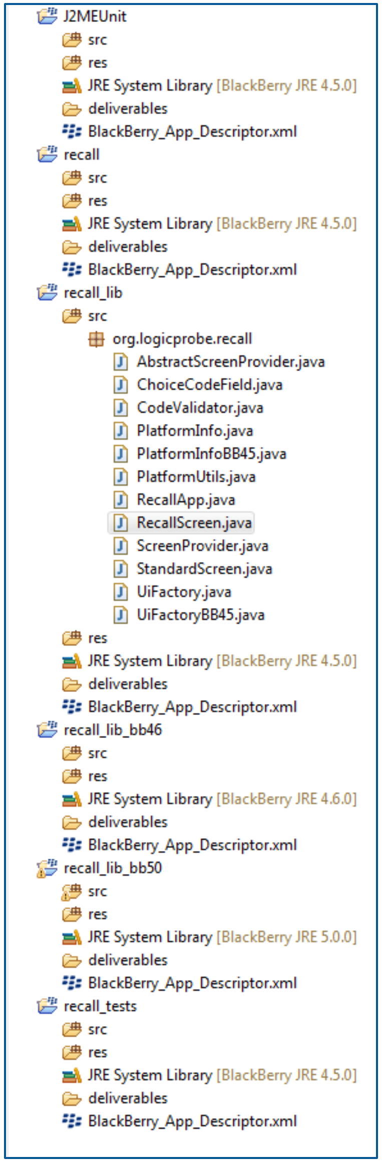
Overview

- Introduction to the environment
- Project layout approaches
- Source control considerations
- Build automation
- Summary and conclusions

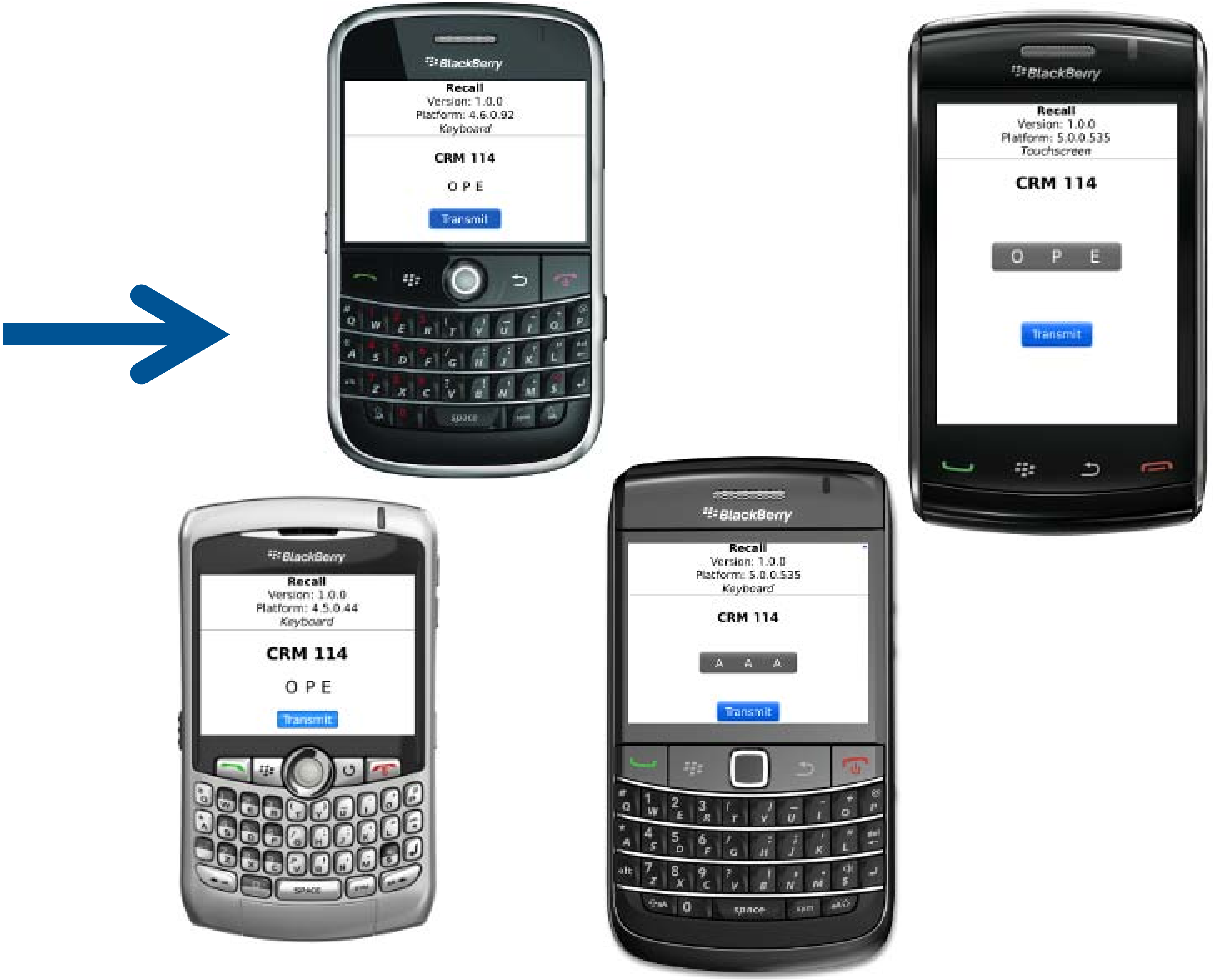
What we want to accomplish

- Support as many BlackBerry OS versions as possible
  - Using a single code base with maximal reuse
  - Taking advantage of features in newer BlackBerry OS versions
  - Releasing for all OS versions in tandem
- Make life easy for the individual developer
  - Single IDE workspace configuration
  - Equally easy to test on all supported OS versions
- Support configuration management, continuous integration, and build automation

Or in other words...



```
J2MEUnit
    src
    res
    JRE System Library [BlackBerry JRE 4.5.0]
    deliverables
    BlackBerry_App_Descriptor.xml
recall
    src
    res
    JRE System Library [BlackBerry JRE 4.5.0]
    deliverables
    BlackBerry_App_Descriptor.xml
recall_lib
    src
        org.logicprobe.recall
            AbstractScreenProvider.java
            ChoiceCodeField.java
            CodeValidator.java
            PlatformInfo.java
            PlatformInfoBB45.java
            PlatformUtils.java
            RecallApp.java
            RecallScreen.java
            ScreenProvider.java
            StandardScreen.java
            UiFactory.java
            UiFactoryBB45.java
    res
    JRE System Library [BlackBerry JRE 4.5.0]
    deliverables
    BlackBerry_App_Descriptor.xml
recall_lib_bb46
    src
    res
    JRE System Library [BlackBerry JRE 4.6.0]
    deliverables
    BlackBerry_App_Descriptor.xml
recall_lib_bb50
    src
    res
    JRE System Library [BlackBerry JRE 5.0.0]
    deliverables
    BlackBerry_App_Descriptor.xml
recall_tests
    src
    res
    JRE System Library [BlackBerry JRE 4.5.0]
    deliverables
    BlackBerry_App_Descriptor.xml
```



Recall
Version: 1.0.0
Platform: 4.6.0.92
Keyboard
CRM 114
O P E
Transmit

Recall
Version: 1.0.0
Platform: 5.0.0.535
Touchscreen
CRM 114
O P E
Transmit

Recall
Version: 1.0.0
Platform: 4.5.0.44
Keyboard
CRM 114
O P E
Transmit

Recall
Version: 1.0.0
Platform: 5.0.0.535
Keyboard
CRM 114
A A A
Transmit

Two environments to consider

- The Developer's Workstation
  - Uses the Eclipse plugin as it is intended
  - Contains the entire project within a single workspace
  - Testing on different BlackBerry OS versions needs to be simple
  - Running unit tests should be as simple as possible
- The Build Server
  - Uses a scripted build process
  - Build artifacts become more important than the projects they are created from
  - Fully automated, including signing, for multiple BlackBerry OS versions

Project Layout

Preprocessor vs. Libraries

- The preprocessor approach is very tempting…
  - It seems very simple to use
  - It allows one code library to target all supported BlackBerry OS API versions
- However…
  - You might still have issues if you want to exclude whole files
  - Java tools really do not like preprocessors
  - Eclipse will get very annoyed with you, and cover your code in red squiggles
  - You have to change configurations and rebuild for each OS version you want to test on

- Libraries might seem complicated at first…
  - Need more projects, with different OS API version dependencies
  - Selectively loading code can be tricky
- But its well worth it:
  - Eclipse is happy with your source code
  - Will not accidentally break API compatibility
  - Testing a different BlackBerry OS API is as simple as running a different simulator configuration

With some practices and patterns, you can completely avoid the preprocessor

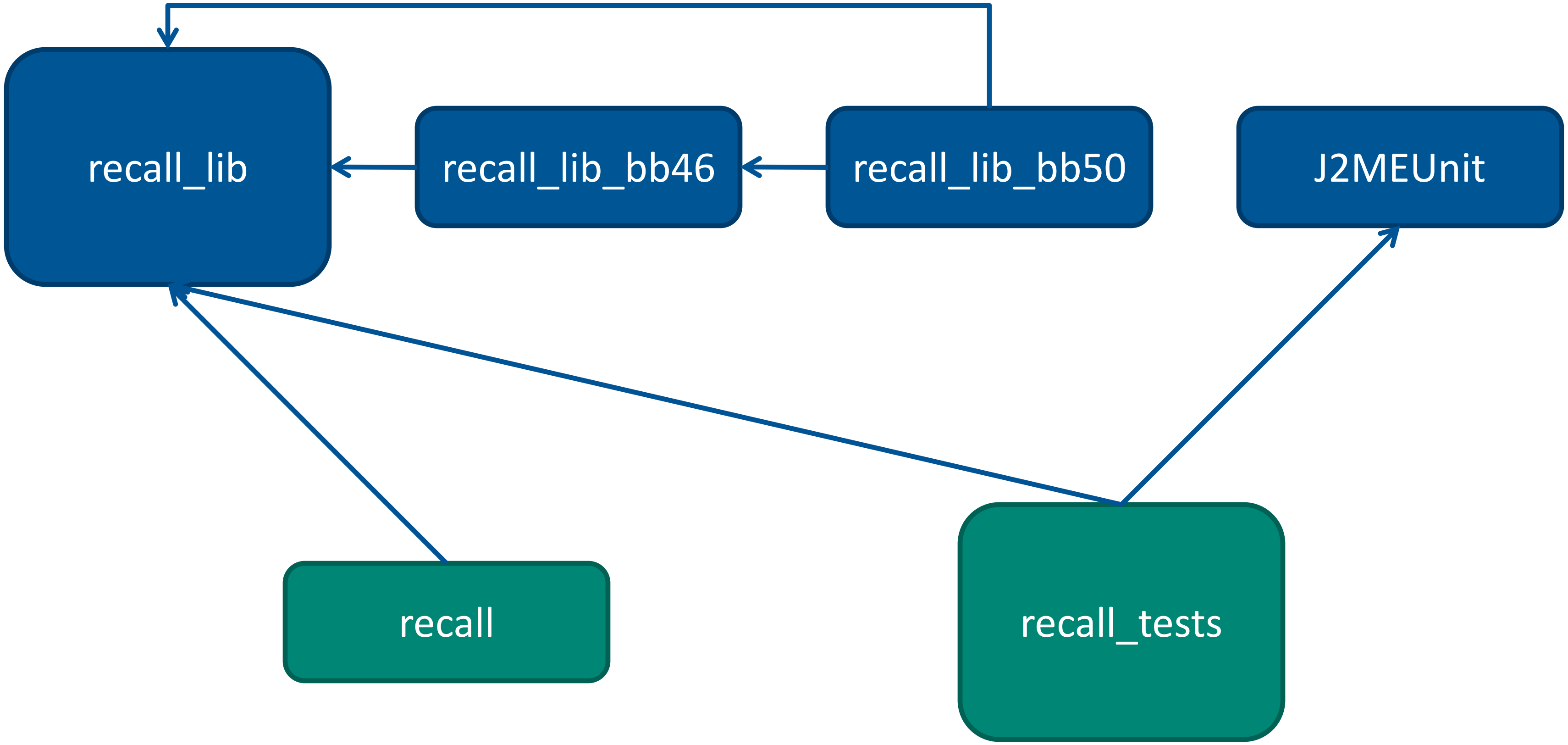
Project Layout

Eclipse Projects

- “Library” projects:
  - **recall_lib** – main application code
    - Targets lowest-common-denominator API version
  - **recall_lib_bb46**, **recall_lib_bb50**
    - Separate libraries for each additionally-supported API version
    - Can depend on each other, for maximal reuse
  - **J2MEUnit** – unit test support library
- “BlackBerry Application” projects:
  - **recall** – main and alternate entry points
  - **recall_tests** – unit tests, and entry point for running unit tests

Project Layout

Dependency Tree



Loading the right code at runtime

- BlackBerry/J2ME lacks real reflection
  - Real reflection would make this a lot easier
  - At least you do have “Class.forName()”
- The “Abstract Factory Pattern” is your friend
  - Top-level abstract class: ThingFactory
    - Knows the fully-qualified names of all subclasses, as strings
    - Has the usual singleton “getInstance()” method
    - Has abstract methods for different functionality
  - Platform-specific subclasses: ThingFactoryBB45, ThingFactoryBB46
    - Can inherit from each other, in order
    - Implement functionality not available in the base API

ThingFactory

```
public abstract class ThingFactory {
    private static ThingFactory instance;
    private static String[] factoryClasses = {
        "net.test.hello.ThingFactoryBB50",
        "net.test.hello.ThingFactoryBB46",
        "net.test.hello.ThingFactoryBB45"
    };

    public static synchronized ThingFactory getInstance() {
        if(instance == null) {
            instance = (ThingFactory)PlatformUtils.getFactoryInstance(
                factoryClasses);
        }
        return instance;
    }

    public abstract Field getThingField();
}
```

PlatformUtils

```
public static Object getFactoryInstance(String[] factoryClasses) {
    // Get a class reference for the concrete factory
    Class factoryClass = null;
    for(int i=0; i<factoryClasses.length; i++) {
        try {
            factoryClass = Class.forName(factoryClasses[i]);
        } catch (ClassNotFoundException e) { }
        if(factoryClass != null) { break; }
    }
    if(factoryClass == null) {
        throw new RuntimeException("Unable to instantiate factory");
    }

    // Instantiate the concrete factory
    try {
        Object instance = factoryClass.newInstance();
        return instance;
    } catch (InstantiationException e) {
        throw new RuntimeException("Unable to instantiate " + factoryClass.getName());
    } catch (IllegalAccessException e) {
        throw new RuntimeException("Unable to instantiate " + factoryClass.getName());
    }
}
```

Additional uses of platform factories

- Instantiating different fields or screens, depending on platform
  - Touchscreen vs. Keyboard
  - BrowserField vs BrowserField2
  - RIM-provided FilePicker vs custom-written FilePicker
- Providing multiple versions of your "open a network connection" code
  - Minor variations from 4.2 to 4.7
  - Completely new API in 5.0+
- Providing top-level platform-info utility methods
  - "What OS am I running?" varies by API version
  - "Do I have a touchscreen?" method call is unavailable pre-4.7

Platform-dependent base class problem

The situation

- Sometimes you have a standard parent class, used all over
  - This class provides hooks to your application's infrastructure code
  - You might want to have different standard behavior depending on OS version or device input type
- This class inherits directly from a RIM framework class
  - That framework class keeps evolving
  - You need to override certain methods that don't exist on older APIs, and those methods have arguments of types that don't exist on older APIs
  - For example: net.rim.device.api.ui.Screen
    - New in 4.7: **`Screen.touchEvent(TouchEvent message)`**
    - More new members in: 3.6.0, 4.0.0, 4.0.2, 4.2.0, 4.3.0, 4.6.1, 4.7.0, 6.0.0

Platform-dependent base class problem

Available choices

- This leaves you with the following choices:
  - Use the preprocessor
    - Sure, its tempting here
    - But then you lose the advantages of being preprocessor-free
  - Create platform-library subclasses
    - Suddenly you have to subclass all of your screens
    - You lose the advantages of a common parent class
  - Refactor to avoid inheriting from the framework
    - We can transform an inheritance problem into composition
    - With composition, this fits very nicely with our factories
    - Can build different screens depending on OS version and input method

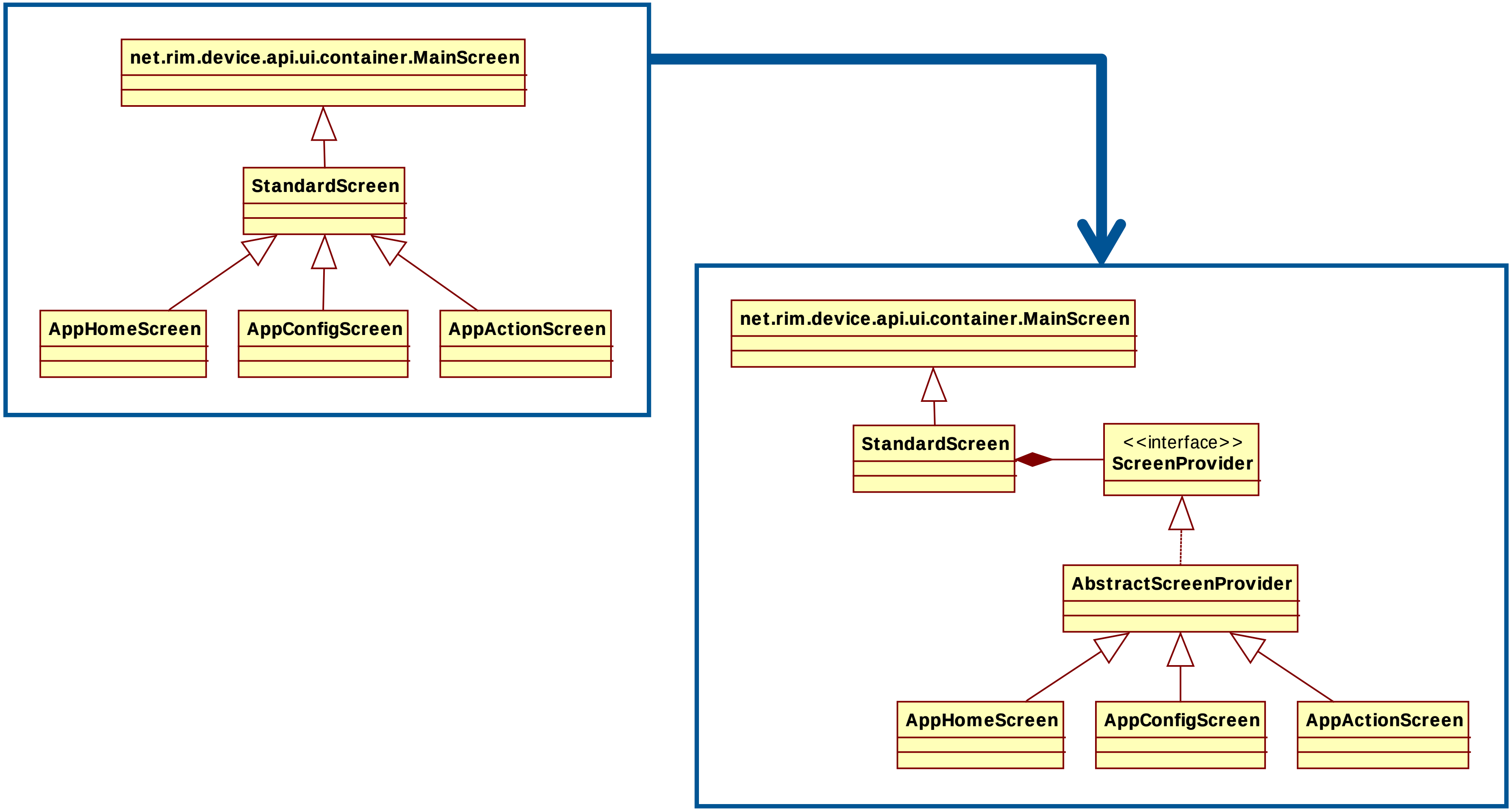
Platform-dependent base class problem

The composition approach

- StandardScreen, StandardTouchScreen
  - Inherits from the framework; fairly lightweight and stub-like
  - Exists in as many variations as necessary for platform support
  - Successive versions can inherit from each other for reuse
- ScreenProvider interface
  - Specifies all methods of RIM's screen class you need access to
  - Has no API dependencies past your base API
  - Contains additional methods your StandardScreen may need
- AbstractScreenProvider
  - Boilerplate implementation of StandardScreen
  - Provides standard implementations of most methods
  - Also has no API dependencies past your base platform

Platform-dependent base class problem

The composition approach



Source control

- What to check in:
  - Nothing at the workspace level
  - At the project level:
    - .project
    - .classpath
    - .settings/org.eclipse.jdt.core.prefs
    - BlackBerry_App_Descriptor.xml
    - res/**
    - src/**

- What to explicitly ignore:
  - At the workspace level:
    - .metadata/
  - At the project level:
    - .locale_interfaces/
    - .preprocessed/
    - bin/
    - deliverables/

Note: These lists may change as RIM updates their development tools

Build Automation

Output Choices

Follow the IDE project structure

- Pros
  - Maintains uniformity between environments
  - Minimizes total number of COD files
- Cons
  - Maximizes the number of per-install COD files
  - Possible library-linking issues
  - Still may have to do multiple builds of the startup project to avoid compatibility mode on touch devices
  - Increases complexity of ALX and JAD files

Ignore the project structure and build it flat

- Pros
  - Minimizes the number of per-install COD files
  - Users only install a minimal, flat, COD sibling set
  - No library-linking issues
  - ALX and JAD files are simpler
- Cons
  - Minor issues with deprecated/changed APIs
  - More total COD files to build/sign
  - Deployment may not exactly match development

Build Automation

Toolchain

- Apache Ant - http://ant.apache.org/
- BlackBerry Ant Tools - http://bb-ant-tools.sf.net/
  - Runs the RAPC compiler
  - Runs the SignatureTool
  - Creates the ALX file and directory structure
  - Updates JAD file and extracts COD siblings
- RIM Build Tools (for each target OS version)
  - Known by many names:
    - BlackBerry Java SDK (Eclipse plugin)
    - BlackBerry JDE (legacy IDE)
    - BlackBerry JDE Component Package (just the build tools)

Build Automation

Toolchain – RIM Build Tools

- All you really need is a subset of the distribution:
  - bin/
    - rapc.jar, preverify.exe, SignatureTool.jar
    - sigtool.csk, sigtool.db
  - lib/
    - net_rim_api.jar
- You can build on a *NIX server, if you:
  - Grab another version of "preverify" from non-RIM J2ME tools
  - Fix the hard-coded backslashes in rapc.jar and SignatureTool.jar
    - Only necessary for older versions of the tools
    - The tools for 5.0 and 6.0 work out of the box
  - Handle SignatureTool's need for a GUI, even when running fully automated (Xvfb is useful for this)

Build Automation

Common Processes

- Build steps
  - Build the code
  - Sign the COD files
  - Package for Over-the-Air (JAD) distribution
  - Package for Desktop-Loader (ALX) distribution
- Build server process
  - Checkout latest code from repository
  - Run the build steps
  - Copy artifacts to an accessible location
  - Clean

Build Automation

Server software options

- Cruise Control - http://cruisecontrol.sourceforge.net/
  - Java-based, highly configurable, highly flexible
  - Also quite complicated to configure and manage
  - Originally, I used a CC-based build server, which worked fairly well
- Bitten - http://bitten.edgewall.org/
  - Python-based, designed as a plugin for Trac
  - Switched to this, since I use Trac for everything else I currently do
  - Sufficiently flexible, and simple to configure

Build Automation

Server software considerations

- Many systems (or at least their web examples) oversimplify the problem:
  - They assume its just checkout/build/[test]/done
  - It might actually be more involved than that
- Sometimes you need configuration data or build steps not in the repository:
  - Paths to the RIM tools
  - Passwords for the code signing keys
  - Means of passing the build number to the build process for embedding
  - Scripts to post artifacts in useful locations on file/web servers
- Running unit tests is not so practical here, unfortunately
  - Currently no easy way to run tests without the simulator
  - Not so easy to integrate the BlackBerry simulator into the build process

The “Recall” application

- Provides a complete Eclipse workspace demonstrating:
  - Application and Unit tests with shared code
    - Functional sample application
    - J2MEUnit with a BlackBerry test runner UI
  - Support for multiple OS versions (4.5, 4.6, 5.0+)
    - Abstract factories for instantiating the right classes
    - Composition-based screen construction
    - Not a single use of the preprocessor
  - Automatable builds with BlackBerry Ant Tools
    - Build, Sign, and Package (JAD and ALX)

Summary

- Developers and Build Servers have different needs
- There are many tricks for targeting different OS versions from the same source tree
- You don’t have to build the same way in all contexts
- Automated builds have some additional considerations you may not think about on the developer’s workstation

For More Information

- Build tools
  - Apache Ant: http://ant.apache.org/
  - BlackBerry Ant Tools: http://bb-ant-tools.sf.net/
  - Sun J2ME Wireless Toolkit (for “preverify” on non-Windows)
    - http://www.oracle.com/technetwork/java/download-2-5-1-138417.html
    - http://java.sun.com/javame/downloads/sdk30_mac.jsp
    - With enough web-hunting, you may even find the older version of the J2ME Tools for Solaris
- Build servers
  - Cruise Control: http://cruisecontrol.sourceforge.net/
  - Bitten: http://bitten.edgewall.org/

BlackBerry.

DEVCON 2010

Thank You

Derek Konigsberg

COM12

September 29th 2010

LOVE working together.