Israel JBoss User Group

Session 08 / 16.4.2008

Java Scripting

By : Iddo Levinson, NexPerience

Introduction

- Java 6 introduced a scripting engine
  - JSR 223: Scripting for the Java Platform.
  - Enables us to execute, within the JVM, script code in any language that has an adapter for this engine.
- In this presentation I will describe the use of Java scripting in NexPerience
  - What we use it for.
  - How we use it.
  - What are the design issues we face.

JSR 223 in Mustang

- Interface to integrate any scripting language
  - Pluggable engine providers framework, supports dynamic engine lookup by language name.
  - Automatic discovery, no registration required.
  - Minimal spec - API to evaluate textual script expression.
- Full spec defines optional interfaces
  - *Compilable*: Improved performance of repeated executions.
  - *Invocable*: Invoke specific script procedure with arguments.
- Language-specific mechanisms to invoke Java code.

Basic Script Invocation

```
javax.script.ScriptEngineManager engineManager
  = new ScriptEngineManager();
javax.script.ScriptEngine engine =
  engineManager.getEngineByName("JavaScript");
try {
  engine.eval("print('Hello, world!')");
}
catch (ScriptException ex) {
  …
}
```

Scripting in NexPerience

NexPerience

- A young startup developing a solution for automatic testing and remote access to mobile handsets
  - Imagine WinRunner for cellular.
  - Combine with remote desktop.
- Product's main components are:
  - IDE for developing test scripts.
  - Server for executing scripts.
- Script developed in IDE are converted to Tcl
  - Java 6 scripting engine used to execute Tcl script.

Why Tcl?

- Mainly product management decision
  - Used for testing in the telecommunication industry.
- Designed to be extended for domain-specific usage
  - We register our handset-oriented functions, which become part of the language.
  - Registration is dynamic – new functions can be added and implementations can be replaced without restarting our product.

Tcl Integration

- Two available implementations for Java
  - » Jacl: Pure Java implementation.
  - » Tcl-Blend: JNI over native implementation.
- Both Supports only minimal JSR spec
  - » Do not implement *Compilable* and *Invocable*.
  - » Possible to invoke Java code from Tcl (but we don’t use it).
- We chose Jacl
  - + Pure Java
  - - Not up to date with latest versions of Java and Tcl

Where We Use Scripts

- Automatically generated scripts
  A. Test scripts
- Manually written scripts
  A. Overrides and external functions
  B. Custom test code

A. Test Scripts

- IDE exposes to user simple script model
  - Functions
  - Simple loops
- Advanced users can modify/extend generated Tcl
  - Our model is limited to keep the GUI simple (e.g., no ‘if’).
  - Limitation – modified script can no longer be edited in GUI.

B. Overrides & External Functions

- Override one of our functions with a different implementation
  - » Plan to support a “selective” override, i.e., provide different implementation for specific handset model, carrier, etc.
- Add a new function not defined in the product
- We also support Java implementation
  - » Developers prefer Java (that’s the default implementation).
  - » In the field (i.e., no development environment) it may be easier to do it with a scripting language.
  - » We want this option to be available also to non-developers (e.g., professional services, support).

C. Custom Code

- Add test-specific functionality to script
  - Not suitable for functions, which are generic.
  - Nicer alternative to editing the generated Tcl scripts, since you can continue editing the script in the IDE.
- Invoked from script with “execute tcl” function
  - We also have ‘execute java’ and ‘execute http’.
  - Tcl is more appropriate for non-developers in the field.

Script Execution

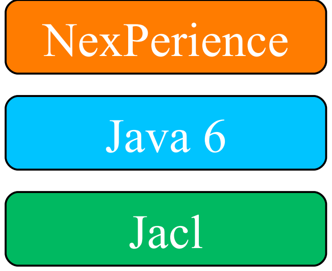
Test Script Execution

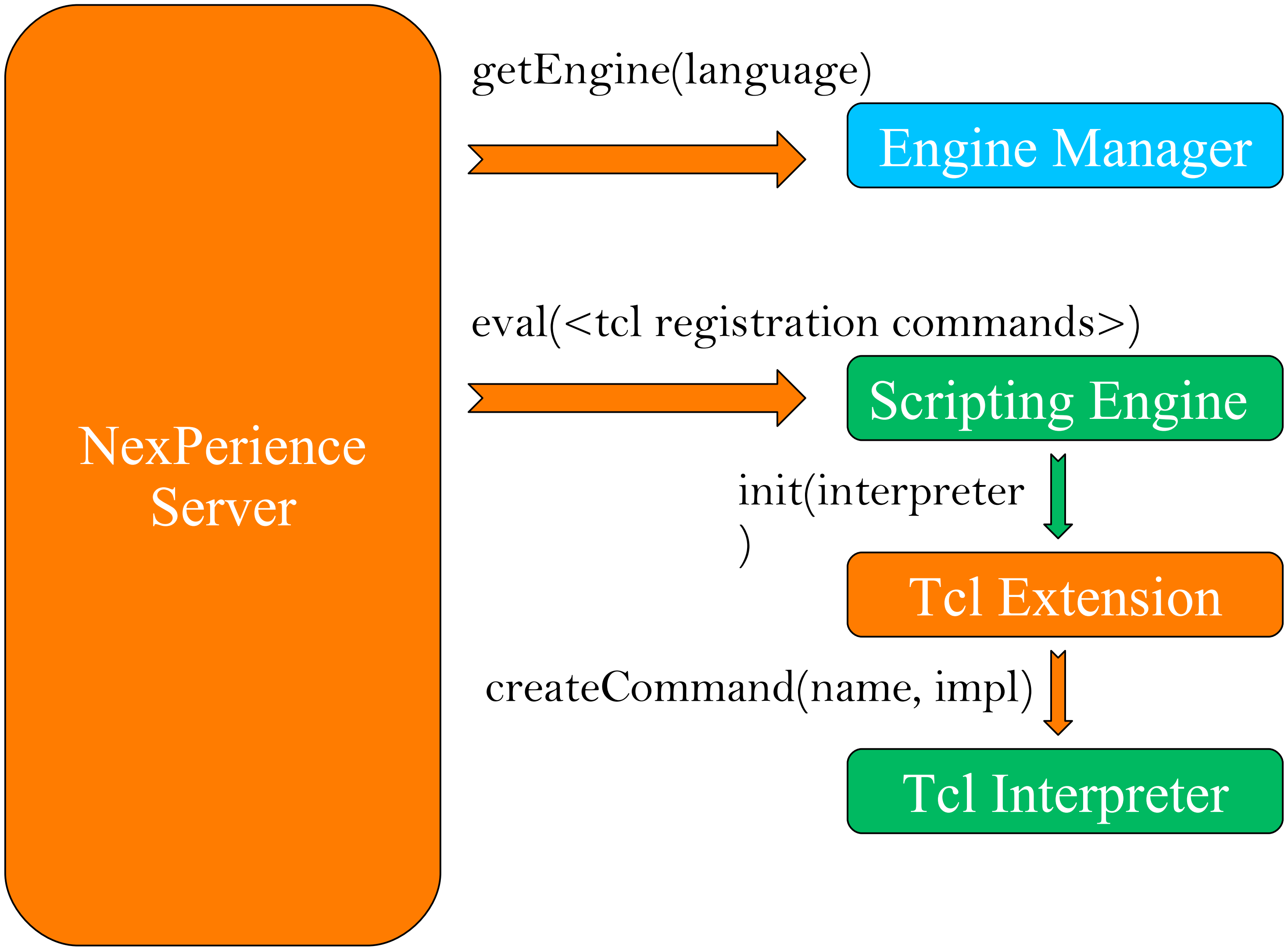
- Prototype:
  - IDE generated an XML representation of script.
  - Server parsed the flow elements and executed script.
- Real Product:
  - IDE generates internal representation of script (XML or Java object model).
  - Server converts script to Tcl and uses scripting engine to execute it.

Consequences

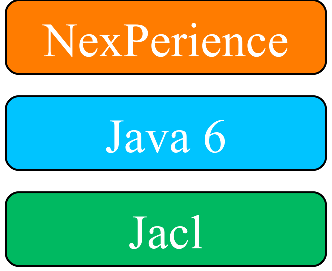
- Advantages:
  - Script language not limited by our implementation – enjoy full features of a real language.
  - No need to develop interpreter – no additional runtime capabilities required to extend our model with additional flow elements (just GUI).
  - Scripting language can be used for additional purposes (e.g., extensions & overrides, testing & debugging).
- Limitations:
  - No complete control over runtime.
  - More difficult to debug.

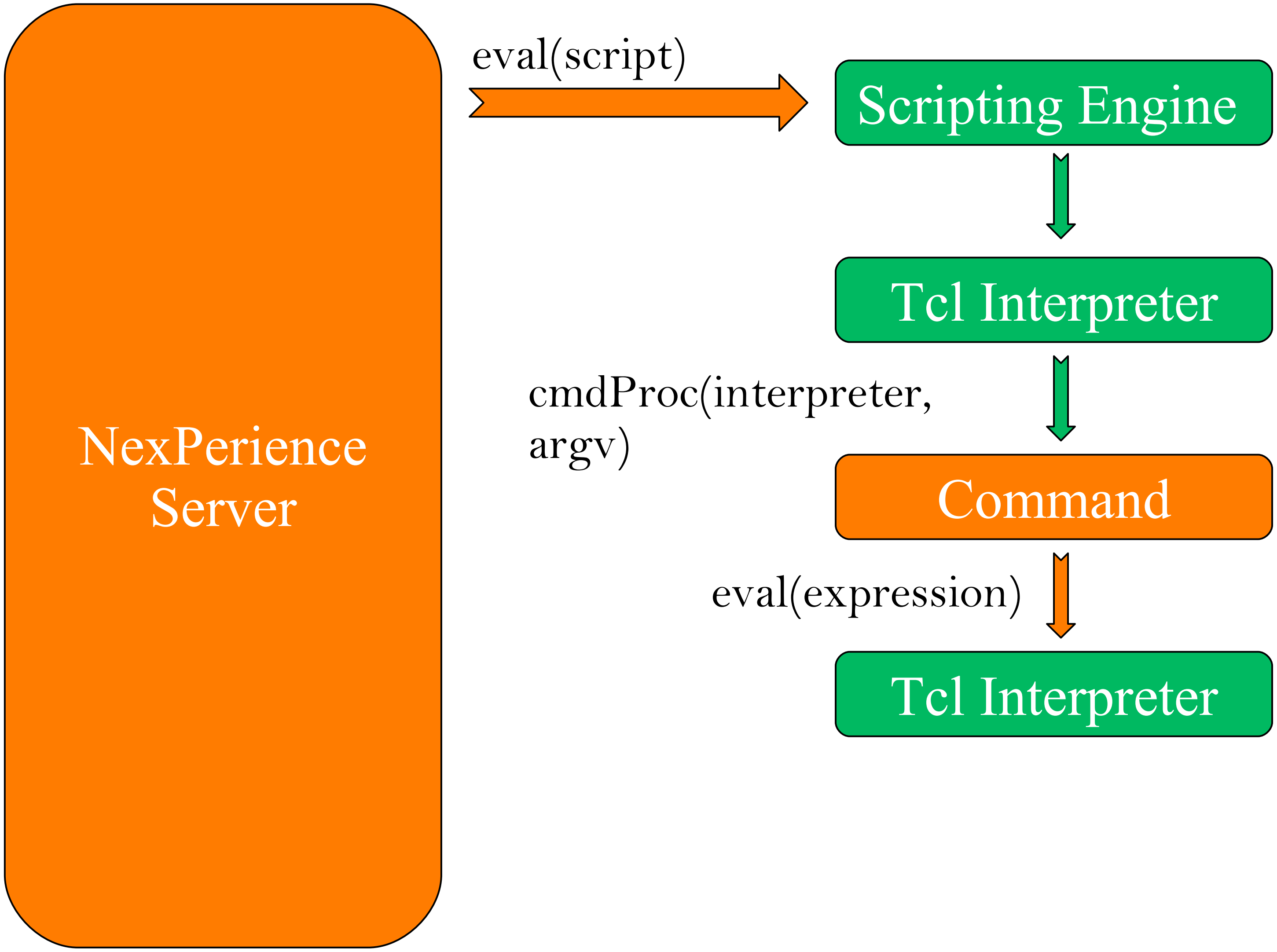
Initialization Flow





Execution Flow

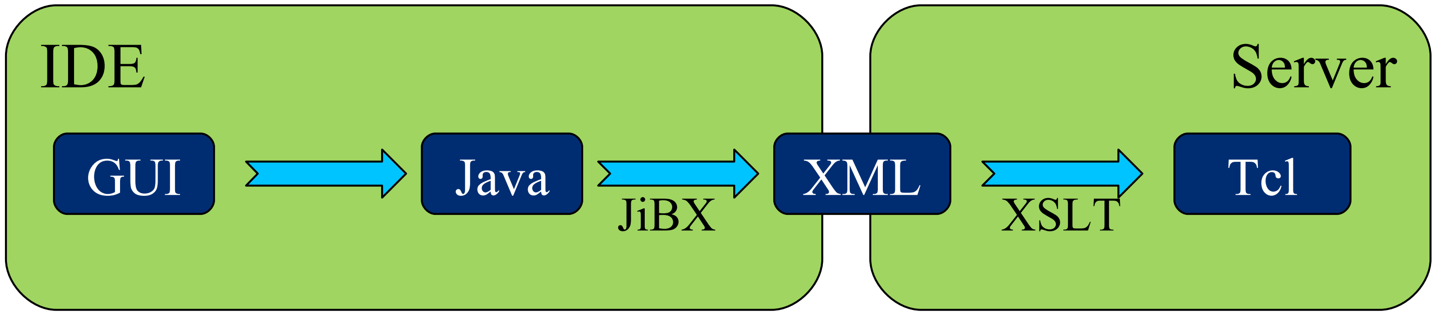




Dynamic Script Generation (1)

1st Generation

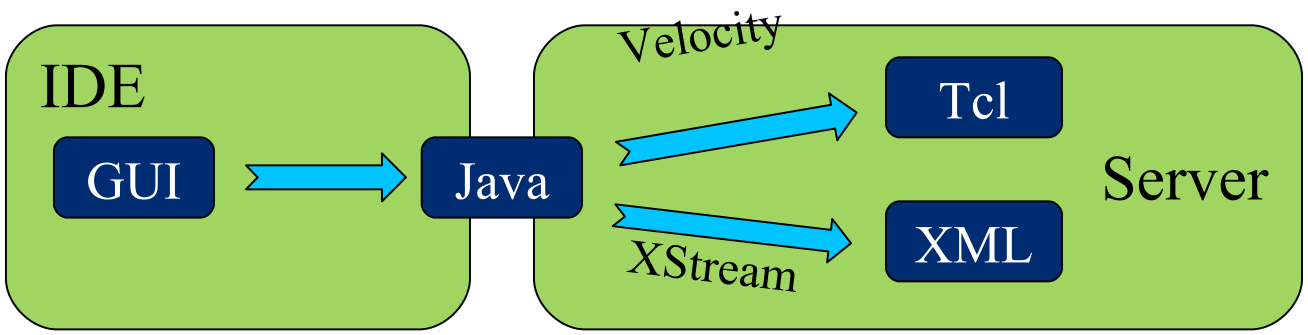
- API between IDE and server was XML script.
- IDE generated XML from Java object model using JiBX.
- Server translated XML to Tcl using XSLT.



Dynamic Script Generation (2)

2nd Generation

- API between IDE and server is the Java object model.
- For execution, server generates Tcl from model using Velocity.
- For persistence, server generates XML from model using XStream.



Command Definition

External Command Definition

- Commands defined in XML files
  - Can be reloaded at runtime, so commands can be added, removed or modified without restarting the server.
  - File hierarchy defines command menu hierarchy in script designer GUI.
  - Function documentation generated with DocBook.
- User Functions
  - Functions can be added in the field.
  - Functions we provide can be overridden in the field.
- Database will be considered in the future
  - We're talking about dozens, not thousands of functions.
  - Files are more convenient, especially for user functions.

Command Definition

Command

- Name
- Display name, help message, tool tip
- Implementation (name of Spring bean, Java class or Tcl script)
- Error policy

Parameter

- Name
- Order
- Optional / mandatory
- Data type
- Value restrictions (enumeration, range)
- # of occurrences
- Display name, help message, tool tip
- View hints

Annotated Spring Beans

- Don’t want to specify bean of each function in XML application context
  - Want to support additions and overrides in the field.
  - Field people aren’t expected to know app-context syntax.
  - XML bean definition doesn’t add anything (just implementation class, no additional properties).
- Use annotations in code:

```
@Scope("prototype")
@Component("sms.send")
public class SmsSendCommand extends ScriptCommandBase {
…
```

Spring 2.5 Application Context

```
<?xml version="1.0" encoding="UTF-8"?>
<beans>
  <context:component-scan
    base-package="com.nexperience.function"/>
  <aop:aspectj-autoproxy/>
  <context:annotation-config/>
</beans>
```

Command Implementation

Runtime Command Input

- Receive command-line from Tcl interpreter
- Use commons-cli to process command-line
  - Parses command-line options and arguments.
  - Performs basic validation (e.g., unexpected parameter).
  - Generates usage message.
- Developed generic mechanism to define CLI syntax
  - Uses function definition.
  - No need for command-specific code to work with commons-cli API.
- Developed generic mechanism for syntax checks
  - Uses function definition.
  - No need for command-specific code for input validations.

Runtime Command Output

- Return value
  - » Use Jacl utility method to automatically convert each Java type to the corresponding Tcl type.
- Error code
  - » Use Jacl to set Tcl error code and message.
- Exceptions
  - » Throw Jacl exceptions to abort flow.
- Stdout and stderr
  - » We had to write custom code that captures it.

Runtime Command Control

- Generic code runs before and after each command, taking care of:
  - Progress indication
  - Abort requests
  - Reporting results
- In the future:
  - Pause/resume
  - Step-by-step execution

Tools, Testing, Issues

Tools

- API for remote script execution
  - Use Spring's JMX annotations to expose API for executing XML scripts.
  - Can be used for integration with 3rd-party software (e.g., test management products).
- Commandline tool to invoke scripts
  - Uses remote script execution API (exposed with JMX).
  - Can be used to invoke a batch of scripts.
- Interactive console
  - For debugging new functionality.
  - For isolating between GUI and functional bugs.

JMX Console

Use Spring annotations to expose methods and attributes through web:

- Class Annotation

```
@ManagedResource(objectName="nexperience.engine.function:name=functionMa
nager", description="Function Manager")
public class FunctionManager … {
  …
```

- Attribute annotation

```
@ManagedAttribute
public Set<IFunctionDef> getAllFunctions() {
  …
```

JMX Console (cont.)

- Operation Annotation

```
@ManagedOperation(description="Get function descriptor by function name")
@ManagedOperationParameters({
  @ManagedOperationParameter(name="command", description="Command"),
  @ManagedOperationParameter(name="subcommand", description="Subcommand")})
public IFunctionDef getFunctionByName(String command, String
subcommand) {
   …
```

Testing with Scripts

- Unit tests run scripts
  - Test script execution framework.
  - Test command functionality.
- Automatic regression tests by QA
  - (Not implemented yet).
  - Will use command-line tool.
- Some test-only commands take advantage of our automation capabilities
  - e.g., run load test of adding/removing handset.

Issues

- Design issues
  - Concurrent script executions.
  - Interpreter reuse.
- Script debugging
  - No debugger.
  - Need to implement special commands to integrate debug messages into main log.