

# Building a Custom Test Execution Environment

Understanding and Extending the Eclipse™ Test and Performance Tools Platform's (TPTP) Testing Tools Framework

# Agenda

- **Introductions**
- Eclipse Test and Performance Tools Platform (TPTP) Overview
- TPTP Testing Tools Overview
- Demo of existing TPTP Testing Tools
  
- Break (15 minutes)
  
- Test Execution Explained
- Building a custom execution environment for a new test type
- What's next for the TPTP Testing Tools Project?

# Introductions

## Instructors

- Joe Toomey  
IBM Rational  
Eclipse TPTP Project Committer  
[jptoomey@us.ibm.com](mailto:jptoomey@us.ibm.com)
- Sri Doddapaneni  
Intel Corporation  
Eclipse TPTP Project Committer  
[srinivas.p.doddapaneni@intel.com](mailto:srinivas.p.doddapaneni@intel.com)

# Introductions (cont.)

## Attendees

Please introduce yourself:

- Name and company or university affiliation
- Level of experience with Eclipse development and the TPTP Testing Tools Project (previously Hyades)
- Your goals for this tutorial

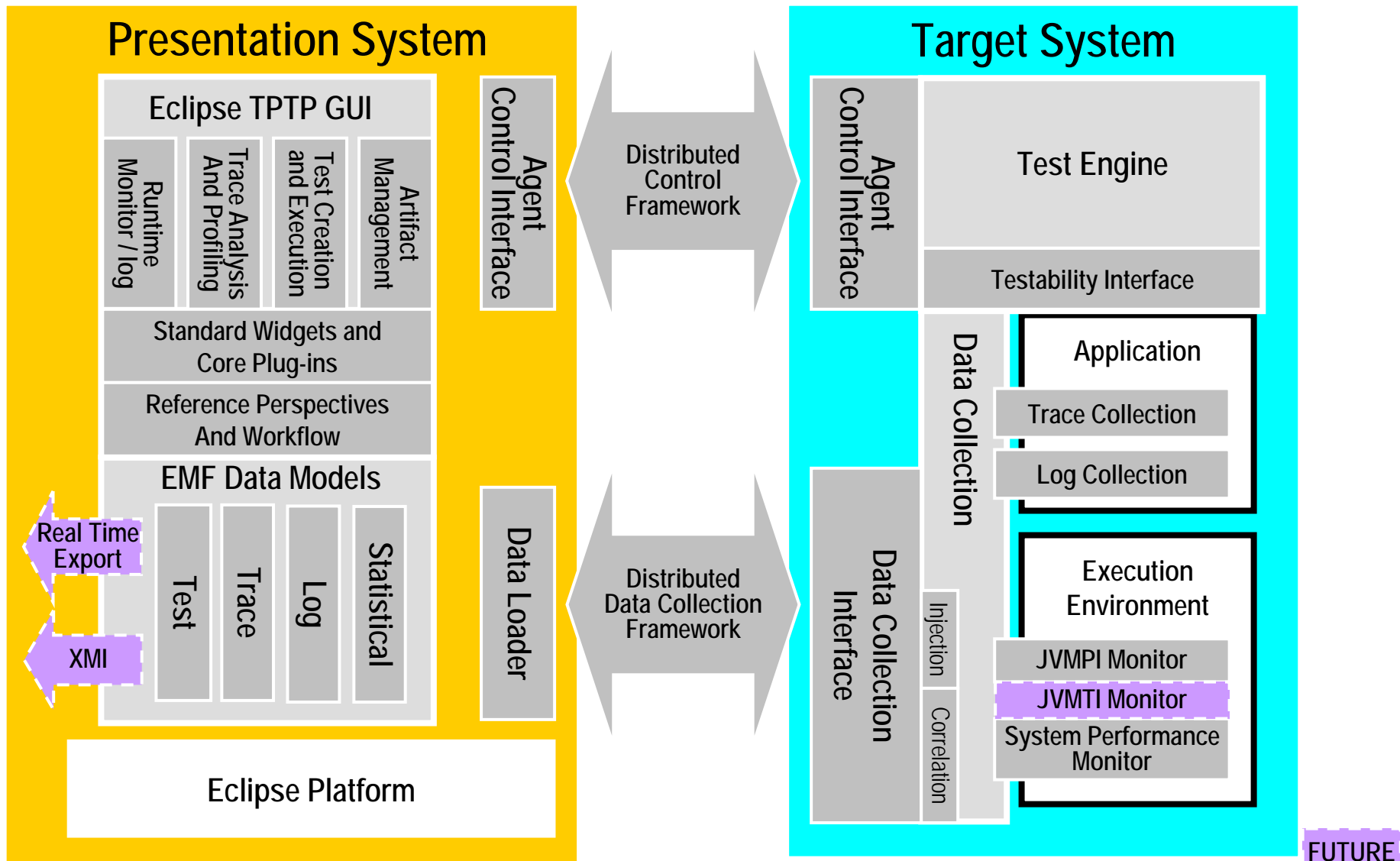
# Agenda

- Introductions
- **Eclipse Test and Performance Tools Platform (TPTP) Overview**
- TPTP Testing Tools Overview
- Demo of existing TPTP Testing Tools
  
- Break (15 minutes)
  
- Building a custom execution environment for a new test type
- What's next for the TPTP Testing Tools Project?

# Eclipse TPTP Overview

- Test and Performance Tools Platform (TPTP) project is an Eclipse top-level project.
  - TPTP project was formed in August 2004 by promoting and expanding Hyades project. The former Hyades project was reorganized into four new projects.
  - Hyades project was formed in December 2002 under Eclipse Tools project.
  - For more on the project visit <http://www.eclipse.org/tptp>
- Mission is “to build a **generic, extensible, standards-based tool platform** upon which software developers can create specialized, differentiated, and interoperable offerings for world class **test and performance tools.**”
- Being developed by three Eclipse Strategic Developer members and a total of eight Eclipse member organizations.

# Eclipse TPTP Architecture



# TPTP Platform Project

- TPTP platform provides common infrastructure for testing, tracing, profiling, monitoring, and logging tools.
- Basic metaphors for interacting with target systems and resources - includes both remote and local systems.
- UI Frameworks and common navigators, viewers, editors, and wizards. It provides many extension points used by TPTP tools and vendor tools.
- EMF-based information model implementations for test, trace, log and statistical data.
- Framework for running rule based queries against data model instances and some simple queries. This is the primary mechanism that ties UI with data models.
- Execution environment that supports deployment, launch, and control of test cases and applications
- Data collection and control frameworks and agents.
- Communication service which is used by the distributed data collection and control frameworks.

# TPTP Testing Tools Project

- Provides testing tools framework by extending the TPTP platform.
- Includes testing tools such as JUnit based component testing tool, Web application performance testing tool, and a manual testing tool.
- Common nomenclature and metaphors facilitate integration of disparate test types and environments.
- Deployment and execution of tests on remote and distributed systems.

# TPTP Tracing and Profiling Tools Project

- Provides frameworks for tracing and profiling tools by extending the TPTP platform.
- Includes profiling tools for both single-system and distributed Java applications.
  - Provides correlation service
- A JVMPi monitoring agent that collects trace and profile data.
  - Collects and analyzes heap and stack information
  - Anticipate JVMTI-based monitoring agent.
  - Anticipate additional language and protocol support.
- A generic tool kit for probe insertion - can instrument byte code of Java applications.

# TPTP Monitoring Tools Project

- Provides frameworks for building monitoring tools by extending the TPTP platform.
- Includes tools for monitoring application servers (JBoss, Jonas, and Websphere) and system performance.
- Collects, analyzes, aggregates, and visualizes data captured in the log and statistical models.
  - A typical example is collection of CPU or memory utilization and viewing, aggregation, and analysis of that data.
- Supports Common Base Event (CBE), provides services for mapping of custom log formats to CBE, and regular expression based log filtering.
- Correlates data across multiple instances of log and statistical models; also across instances of trace and test history models
  - Enables symptom and pattern analysis.

# Technology Domain: User Interface

- Log View: A list and property viewer for Common Base Events.
- Statistical and Performance Viewers: Interactive line charts to show data trends over time, and tabular views to present statistical information about application behavior
- Probe Editor: An Editor for composing contents of a probe and specifying its intended insertion point
- Sequence Diagram: A read-only UML 2 sequence diagram that can be bound to any appropriate data source such as call graph or set of object, class, thread, process, and/or machine interactions over time.
- Charting: A service that renders standard business graphs when fed with proper data. Useful for reporting test histories or CPU utilization over time.
- Profiling and Logging Perspective: A single perspective and navigator useful for showing trace and monitoring data and associated context.

## Technology Domain: User Interface (contd.)

- Test Perspective: A perspective for creating and managing test artifacts such as tests, data pools, execution histories.
- Common Editors: Editors for modeled data such as test metadata, test behavior, deployment, and execution history. These are specialized for JUnit, Web Application and Manual Tests.
- Data Pool Editor: An extensible editor for management of test data and support for import of volume data.
- Code Coverage: Extends table views with method level coverage information.
- Manual Test Client: An application running in target environment that allows user to step through a test case and specify test verdict.
- Reporting: Extension points and usage of SVG and text based execution history reports.

## Technology Domain: User Interface (contd.)

- Generic Log Adapter Configuration Editor: An editor for GLA configuration files that supports rich regular expression language for filtering logs in target environment
- Symptom Viewer: An extension of statistical viewer customized for management of symptom database
- Reporting: Statistical viewer is extended/reused to report data collected by Windows perfmon agent.
- GLA Perspective

# Technology Domain: Data Models

- **Test data model** supports test cases, input data, results and execution history. It consists of three models
  - A data model for creation, definition and management of test artifacts. This implements UML 2 Testing Profile meta model
  - A data model for test case behaviors. It implements UML 2 Interaction meta model.
  - A data model for test execution history. It supports execution traces and results from disparate test types.
- **Trace data model** supports traces of local and distributed execution stacks and heaps
- **Log data model** supports sequence of CBEs and other logged messages that are transformable into CBE.
- **Statistical data model** supports snapshots of arbitrary data over time

# Technology Domain: Execution Environment

- Framework: A set of APIs and interfaces used to control an application, particularly test execution and monitored applications.
- Probekit and Byte Code Insertion: User can specify probes using probekit editor and BCI inserts byte code corresponding to the probes before loading Java class files by the target JVM.
  - Probe instances and run-times enable collection of distributed traces.
- Common Base Event (CBE) and Logging: A set of logging interface implementations to match JSR 47, Apache Commons etc. Implementation supports CBEs as well as Eclipse Workbench logging.
- Correlation Engine is used for associating CBEs based on various data value oriented algorithms such as URI and timestamps.
- Choreography Engine: A dynamically scriptable engine used to run tests without the requirement of generated code. BPEL is used to define the behavioral flow across the various test services.
- Generic Log Adapter: A runtime that supports asynchronous log parsing and formatting.

## Technology Domain: Execution Environment (contd.)

- Execution Harness initiates and manages Hyades test execution
- Common Runner implements generic test suite execution capability and is extended by test runners for each specific test type.
- Manual Runner supports user assisted test execution by the manual test client
- JUnit Runner supports execution of JUnit test suites

# Technology Domain: Data Collection and Agents

- An infrastructure is provided to collect data for the various models. This includes the XML fragment specifications that need to be created as well as the model loaders that will load those fragments into the specific model.
  - Provides service to create instances of XML fragments
  - Correlation service to assist in creating data that can be correlated across machine boundaries.
  - Service to create Common Base Event instances.
- Framework: The basic control flow infrastructure complete with interfaces for the management of an agent and the data it collects. This includes a large set of predefined XML fragment based events that can be sent and will be loaded into the model as appropriate.
- TPTP JVMPI: A JVMPI agent used for profiling a JVM as well as sharing class load events for BCI engines.
- Logging: A default logging agent for this framework as well as an implementation of a Common Base Event.
- JMX based agents for monitoring JBoss, Jonas, and Websphere application servers

## Technology Domain: Data Collection and Agents (contd.)

- **Component Test Agent:** An extensible mechanism for capturing execution events during the execution of a test and sending those events back to the workbench to be loaded into the execution history model.
- **HTTP Proxy Recorder:** A proxy-based recorder to intercept and capture HTTP requests which can be converted to URL test cases.
- **Generic Log Adapter Configurations:** Multiple regular expressions and static parser configurations for various log formats, that extend the Generic Log Adapter execution environment.
- **Perfmon agent:** An agent to monitor system performance on Windows and Linux.

# Technology Domain: Communication

- A light-weight server provided for management of agents; includes registration so they can be shared across process and transaction boundaries in a distributed environment.
- Pluggable communication layer. A default TCP/IP based implementation is provided.
- Security provided using an optional plug-in layered above communication layer.
- Generic interface to facilitate movement of commands and data between the workbench and one or more agents.
- Shared memory pipes provided for data transfers between processes on the same machine.
- The data communication and control interfaces are exposed with C, Java, and Web services bindings.

# Agenda

- Introductions
- Eclipse Test and Performance Tools Platform (TPTP) Overview
- **TPTP Testing Tools Overview**
- Demo of existing TPTP Testing Tools
  
- Break (15 minutes)
  
- Test Execution Explained
- Building a custom execution environment for a new test type
- What's next for the TPTP Testing Tools Project?

# TPTP Testing Tools Project Overview

- Provides testing tools framework by extending the TPTP platform.
- Includes testing tools such as JUnit based component testing tool, Web application performance testing tool, and a manual testing tool.
- Common nomenclature and metaphors facilitate integration of disparate test types and environments.
- Deployment and execution of tests on remote and distributed systems.

# TPTP Testing Tools Tasks and Concepts

- Test Navigator and User Test Projects
- Type of tests – JUnit, Web Application, Assisted Manual
- Create and edit test suite, test location, and behavior
- Generate Java code for a test
- Create and manage test data pools
- Deploy and execute test suite
- Analyze test results – execution history, traces, verdicts, etc.

# Data Models (Test Model)

The TPTP Test Model encompasses:

- UML 2 Testing Profile Implementation
- Behavioral model
- Execution Histories

# Data Models (Test Model)

## UML 2 Testing Profile Implementation

- TPTP was part of the finalization process (as Hyades) in the OMG's acceptance of the U2TP submission. We have implemented our reference implementation based on the MOF independent meta model.

# Data Models (Test Model)

## Behavioral model

- The behavioral portion of the Test Model is an implementation of UML2 Interactions. We have provided a façade on top of this behavioral model for several reasons:
  - To provide a more semantically familiar programming model (loops and decisions, not combined interaction fragments with operators.)
  - To facilitate interop among tests from different test tool vendors (multiple ways of representing the same transaction)
  - To (potentially) allow TPTP based products to access behavior that is not modeled (via another implementation of the same façade interface.)

# Data Models (Test Model)

## Execution Histories

- Stores the results of test executions
  - References to test, trace, deployment info
  - Verdicts
  - Test Log messages and console out from test
  - Provides an extensibility mechanism for custom typed attributes

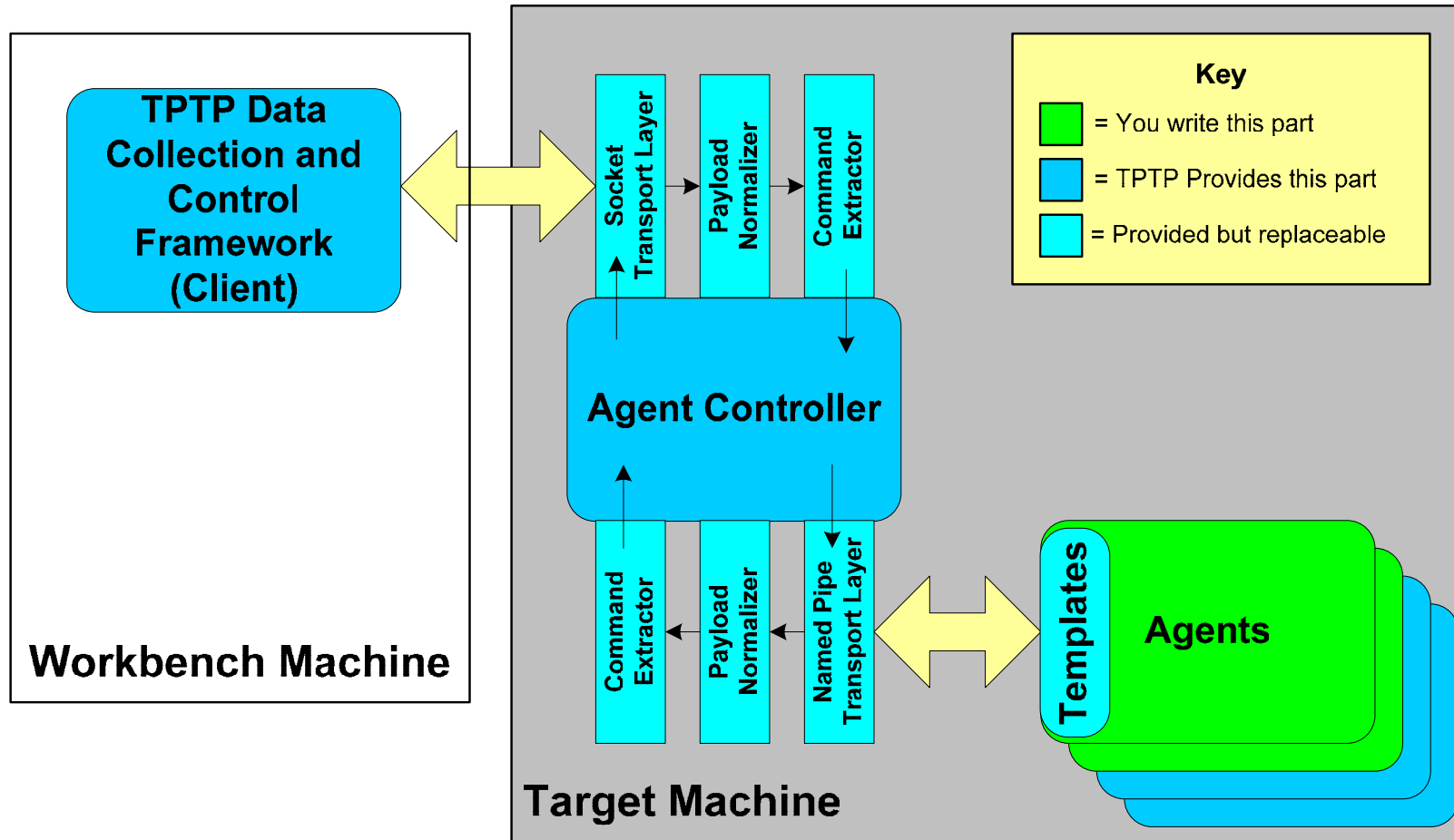
# User Interface Extension Points

- Test Creation Wizard
  - Extension point for defining wizard to create specific types of tests
- Test Editor
  - Extension point provided for associating editor with specific type of test
- Analyze Results
  - Extension point to open Log views use to analyze results of test run
- Publish Reports
  - Extension point for custom report

# Execution Environment

- The TPTP Testing Tools project extends the capabilities provided by the platform to allow distributed execution, control and data collection of tests.
- Extenders of the TPTP Testing Tools project can enable distributed execution, control and data collection of their own test types in two ways
  - By leveraging and extending the components provided by the Testing Tools project (less code to write, less control over behavior)
  - By extending the Platform Project (much more code to write, more control over behavior)
- Later in this tutorial, we will show how to enable support for a new test type by extending the Testing Tools project components

# TPTP Agent Controller



Note: Workbench machine and target machine can be the same machine.

# Agenda

- Introductions
- Eclipse Test and Performance Tools Platform (TPTP) Overview
- TPTP Testing Tools Project
- **Demo of existing TPTP Testing Tools**
  
- Break (15 minutes)
  
- Test Execution Explained
- Building a custom execution environment for a new test type
- What's next for the TPTP Testing Tools Project?

## Performance Testing of Web Application Using TPTP

- Recording a Web application test
- Editing a Web application test
- Generating Java code for a Web application test
- Running a Web application test
- Analyzing Web application test results
- Demo

# JUnit Testing Using TPTP

- Creating a JUnit test
- Editing a JUnit test
- Generating Java code for a JUnit test
- Running a JUnit test
- Analysis of JUnit test results
- Demo

# Assisted Manual Testing using TPTP

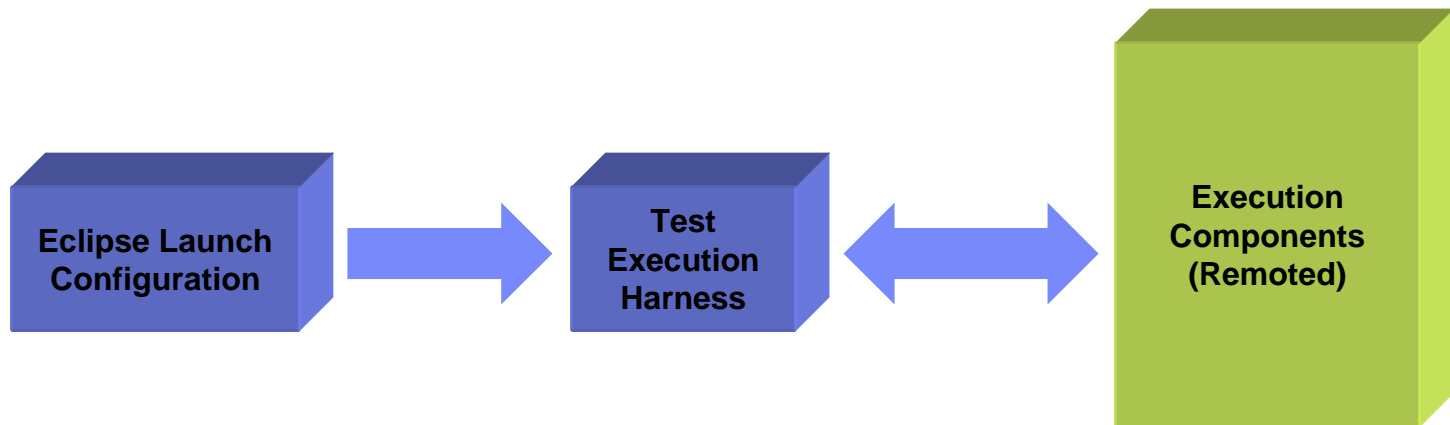
- Creating a manual test
- Editing a manual test
- Running a manual test
- Analyzing manual test results
- Demo

# Agenda

- Introductions
- Eclipse Test and Performance Tools Platform (TPTP) Overview
- TPTP Testing Tools Project
- Demo of existing TPTP Testing Tools
  
- Break (15 minutes)
  
- **Test Execution Explained**
- Building a custom execution environment for a new test type
- What's next for the TPTP Testing Tools Project?

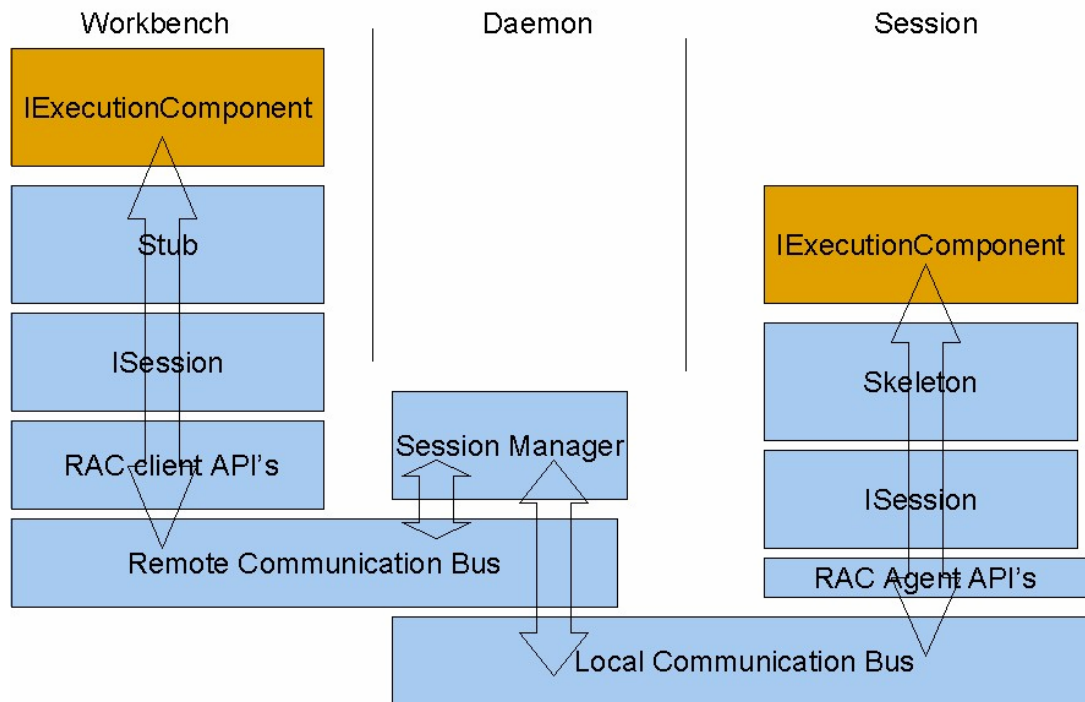
# Test Execution explained

- Within Eclipse Workbench, users initiate test execution by creating a launch configuration for the test.
- The launch configuration performs necessary setup and then launches the test by calling the Test Execution Harness
- The Test Execution Harness is implemented in the `org.eclipse.hyades.execution.harness` plugin

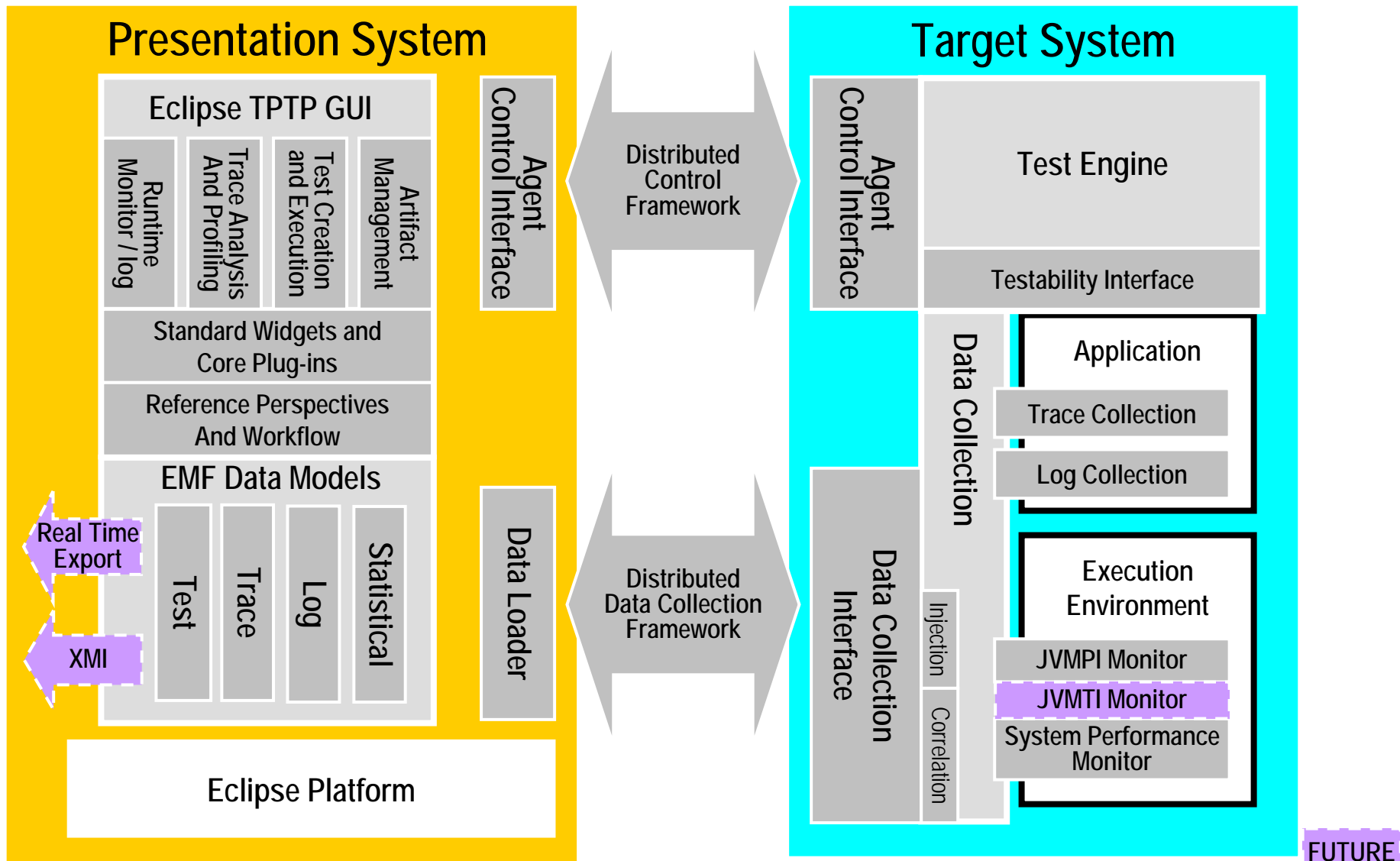


# Test Execution explained (cont)

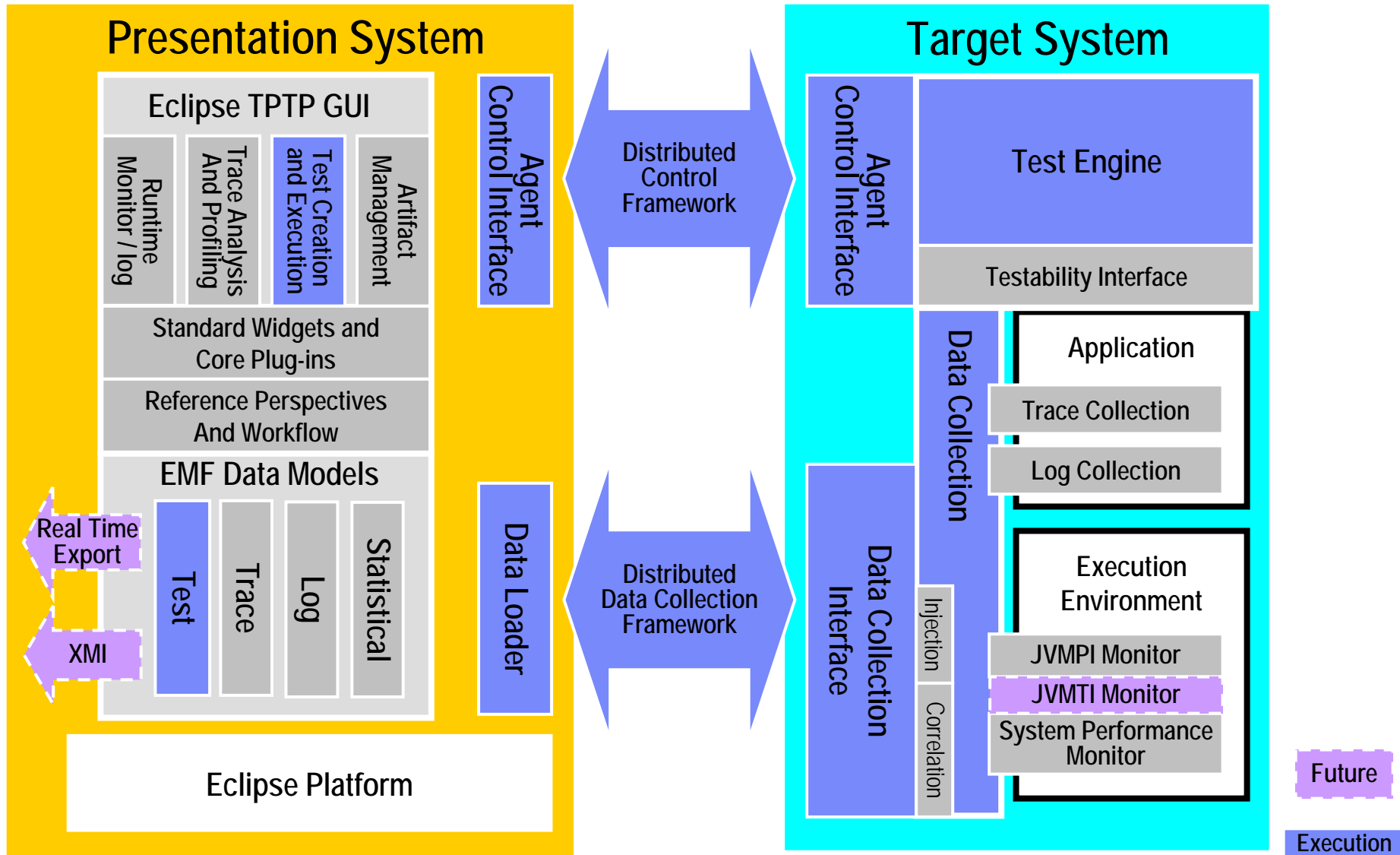
- The Execution Components are remoted via the TPTP Platform Execution Environment
- The Test Execution Harness invokes methods of the Execution Components locally in the Workbench process, and the agent controller marshals and invokes the methods on the target execution environment.



# Eclipse TPTP Architecture



# Eclipse TPTP Architecture



## Test Execution explained (cont)

- The execution components that the test execution harness interacts with, and which you may choose to extend or replace are
  - defined by interfaces in the `org.eclipse.hyades.execution.core` plugin
  - registered for use with the execution harness via extension points

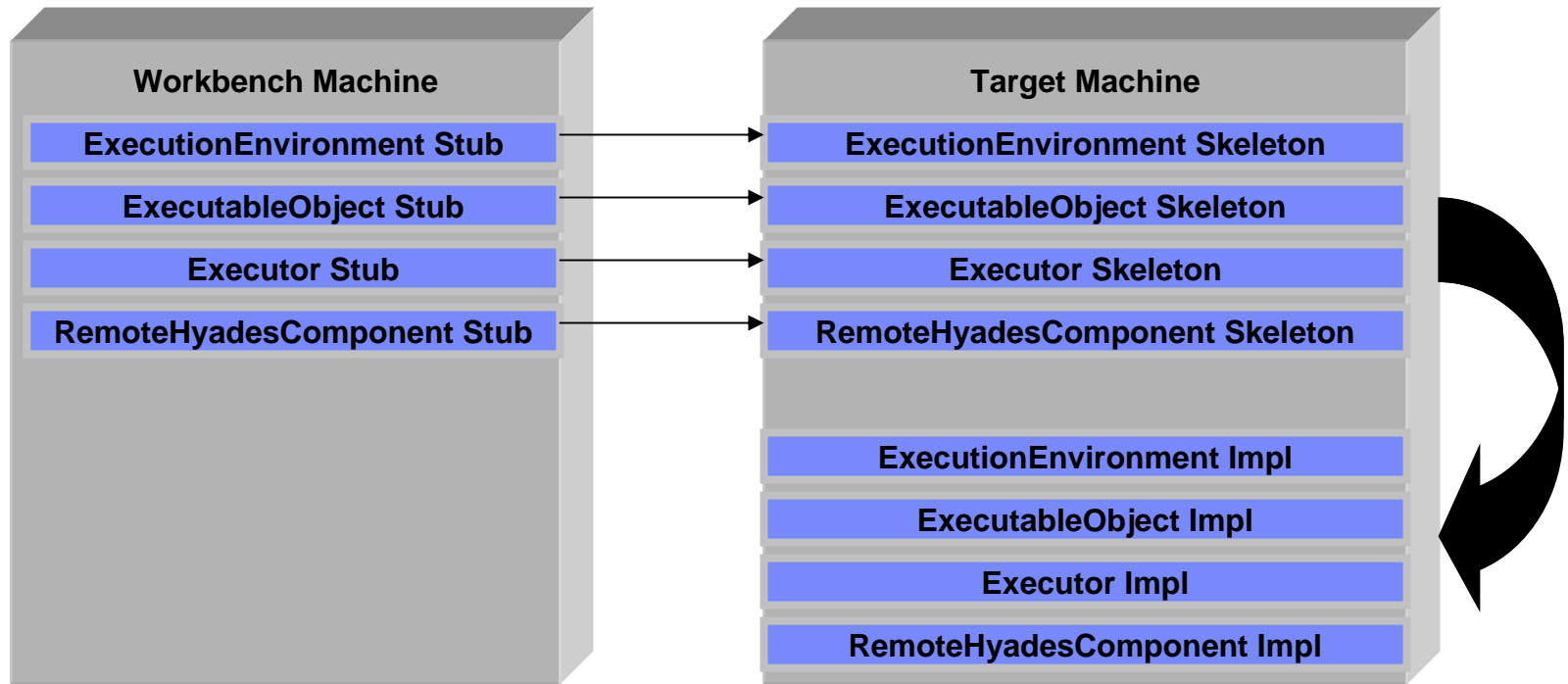
# Test Execution explained (cont)

These Execution Components are:

- **IExecutionEnvironment**
  - Describes the environment in which the test will execute
- **IExecutableObject**
  - Describes the data necessary to execute the test
- **IExecutor**
  - Executes the test described by the **IExecutableObject** in an environment described by the **IExecutionEnvironment**
- **IRemoteHyadesComponent**
  - The agent responsible for collecting data from the running test and transmitting it back to the workbench to be model loaded
  - Sometimes interchangeably called Agent in the code

## Test Execution explained (cont)

- Each remotable Execution Component has a stub class (which runs in the workbench process), a skeleton and an impl class (both of which run on the target machine)



## Test Execution explained (cont)

- The TPTP Testing Tools Project provides implementations of each of these components
- The provided implementations are based on a Java execution environment, thus
  - the IExecutionEnvironment component allows the consumer to specify environment variables (such as classpath)
  - the IExecutableObject component allows the consumer to specify command line arguments (such as JVM args, main class and parameters)
  - the IExecutor component launches a JVM with the specified arguments in the specified environment
  - the IRemoteHyadesComponent monitors the test and provides the communication channel to control it and send back results

## Test Execution explained (cont)

- Execution Environment
  - The provided execution environment allows the consumer to specify the classpath for launching a new JVM in which the test will run
  - `org.eclipse.hyades.execution.local.ExecutionEnvironmentStub`
  - `org.eclipse.hyades.execution.remote.ExecutionEnvironmentSkeleton`
  - `org.eclipse.hyades.execution.core.impl.ExecutionEnvironmentImpl`

## Test Execution explained (cont)

- Executable Object
  - The provided executable object allows the consumer to specify the command line arguments to the process that will be launched (JVM args, main class and arguments to it)
  - `org.eclipse.hyades.execution.local.JavaProcessExecutableObjectStub`
  - `org.eclipse.hyades.execution.remote.JavaProcessExecutableObjectSkeleton`
  - `org.eclipse.hyades.execution.core.impl.JavaProcessExecutableObjectImpl`

## Test Execution explained (cont)

- Executor
  - The provided executor launches a JVM with the arguments specified by the executable object in the environment specified by the execution environment
  - `org.eclipse.hyades.execution.harness.TestExecutionHarnessExecutorStub`
  - `org.eclipse.hyades.execution.remote.JavaProcessExecutorSkeleton`
  - `org.eclipse.hyades.execution.core.impl.ProcessExecutorImpl`

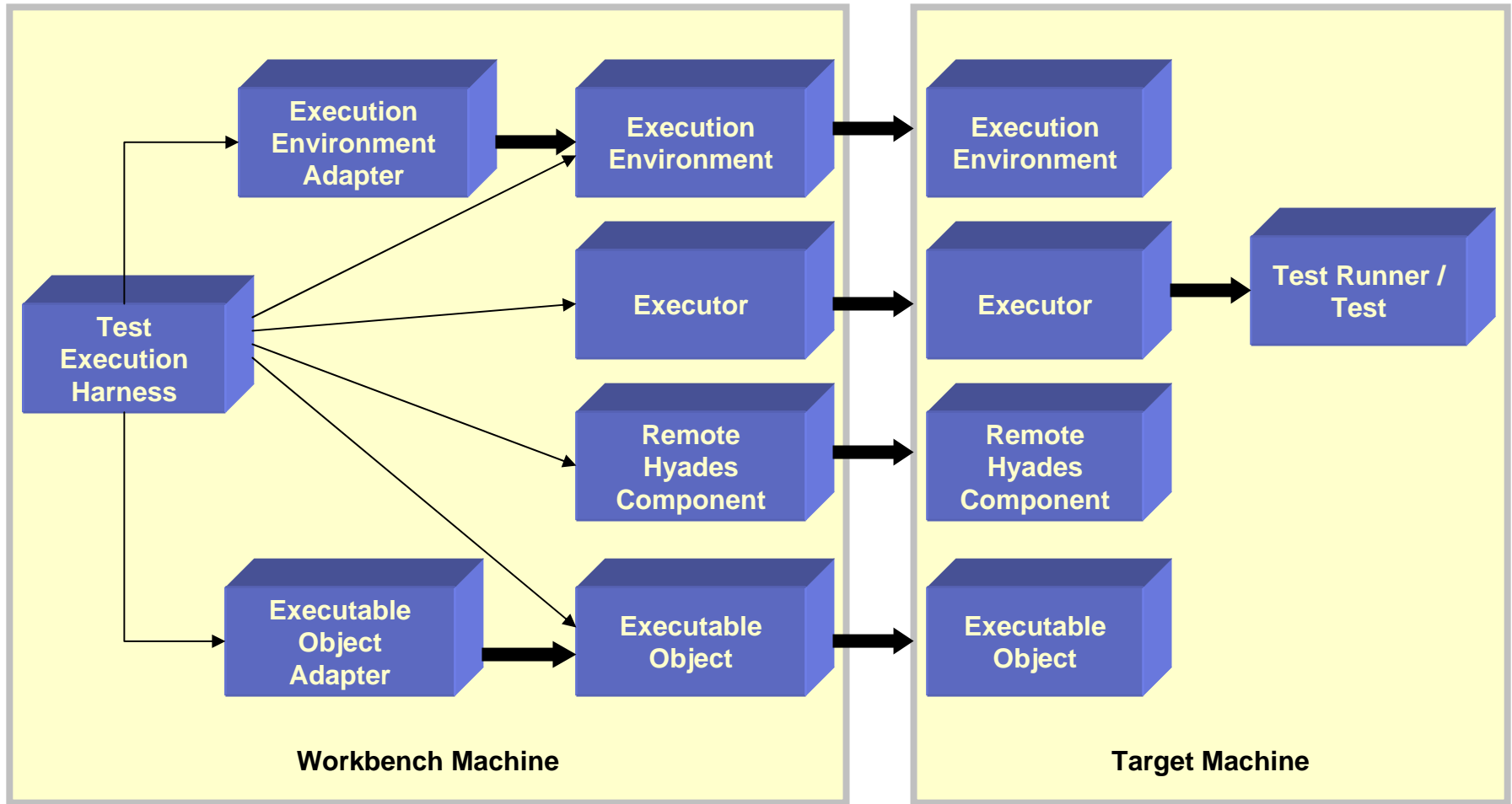
## Test Execution explained (cont)

- Remote Hyades Component (Agent)
  - The provided Remote Hyades Component establishes a control and data channel between the workbench and the test JVM.
  - `org.eclipse.hyades.execution.local.JavaTaskRemoteHyadesComponentStub`
  - `org.eclipse.hyades.execution.remote.JavaTaskRemoteHyadesComponentSkeleton`
  - `org.eclipse.hyades.execution.core.impl.JavaTaskRemoteHyadesComponentImpl`

## Test Execution explained (cont)

- In order to reuse these java-based execution components, you register adapter classes, based on your test type, to populate the necessary data
  - classpath in the execution environment
  - arguments in the executable object
- The test execution harness calls out to your adapters during the test launch process so that they can populate that data.

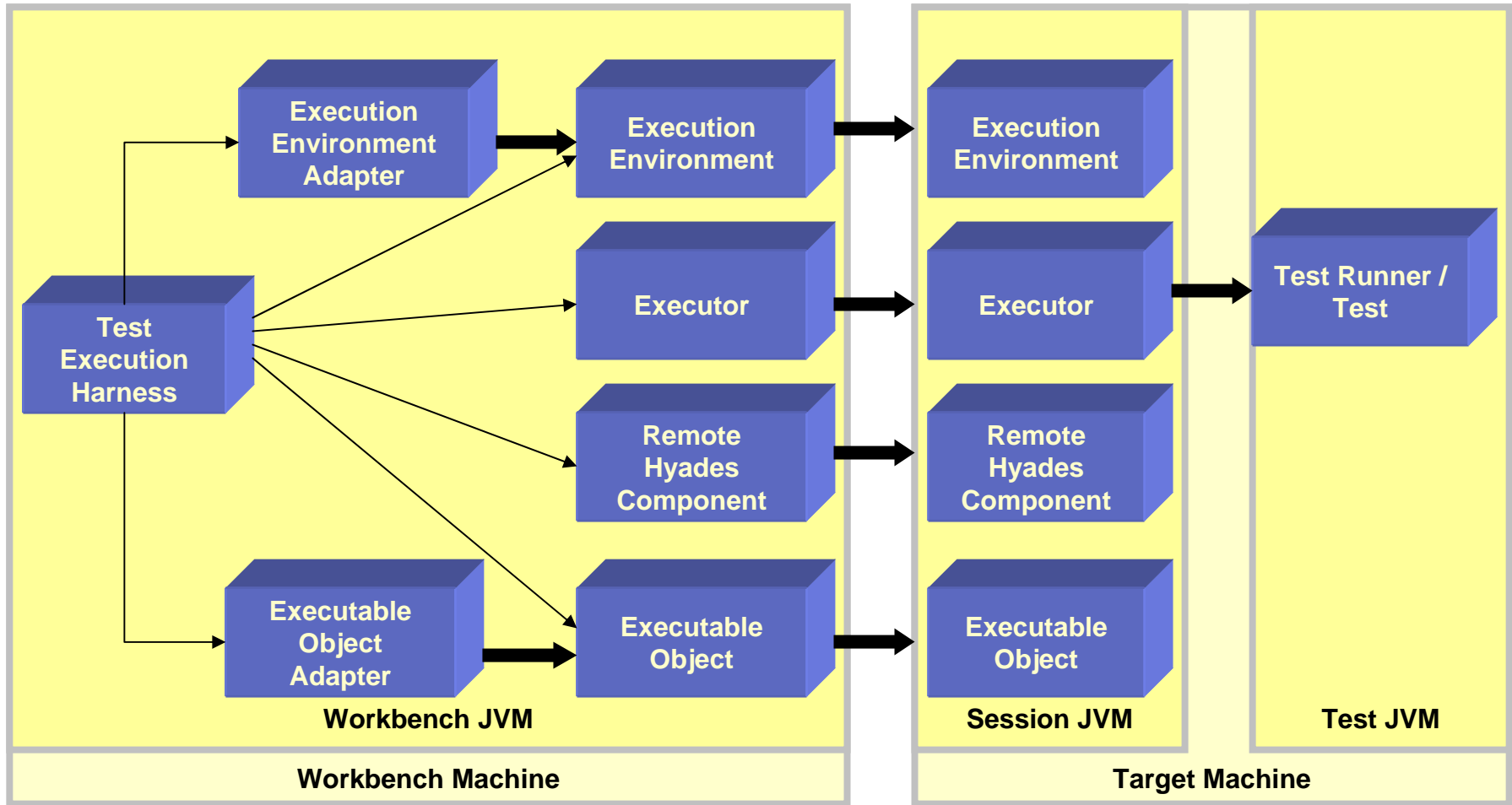
# Test Execution explained (machine boundaries)



## Test Execution explained (cont)

- Process view
  - When executing a TPTP test, the workbench JVM communicates with a JVM that is running within the Agent Controller on the target machine
  - When using the TPTP Test Project execution components, the Executor launches another JVM on the target machine in which the Test Runner (and likely the test) will run.

# Test Execution explained (JVM boundaries)



## Test Execution explained (cont)

- Debugging tips
  - If you add support for your own test type, you may need to debug into each of these JVMs.
  - The Workbench JVM
    - Debugging the workbench is easy – just use the runtime workbench and simply set breakpoints

## Test Execution explained (cont)

- Debugging tips (cont)
  - Debugging the Target Machine JVMs can be accomplished by connecting to the JVM as a remote java application using the eclipse debugger
  - To do this, you must specify the correct JVM options when the JVM is created
    - Turn on debugging
    - Specify the transport and port
    - Specify whether or not to suspend the JVM pending debugger attach

# Test Execution explained (cont)

- Debugging tips
  - Configuring the Session JVM for debugging
    - In the agent controller installation, under `plugins/org.eclipse.hyades.test/config/pluginconfig.xml`, add the text shown in red below:

```
<Application configuration="default" executable="HyadesTestingSession" location="%SYS_TEMP_DIR%" path="%JAVA_PATH%">  
  <Parameter value="-Xdebug -Xrunjdp:transport=dt_socket,server=y,suspend=y,address=8000" position="prepend"/>  
  <Parameter position="prepend" value="&quot;-Dhyades.test.JVM=%JAVA_PATH%&quot;"/>
```

- Attach the debugger to the JVM using port 8000

## Test Execution explained (cont)

- Debugging tips
  - Configuring the Test JVM for debugging
    - Your executable object adapter class specifies the JVM arguments for the test JVM, so you simply pass the appropriate debug JVM arguments

if (DEBUG)

```
jvmArgs += "-Xdebug -Xrunjdw:transport=dt_socket,server=y,suspend=y,address=8005 ";
```

- Attach the debugger to the JVM using port 8005

# Agenda

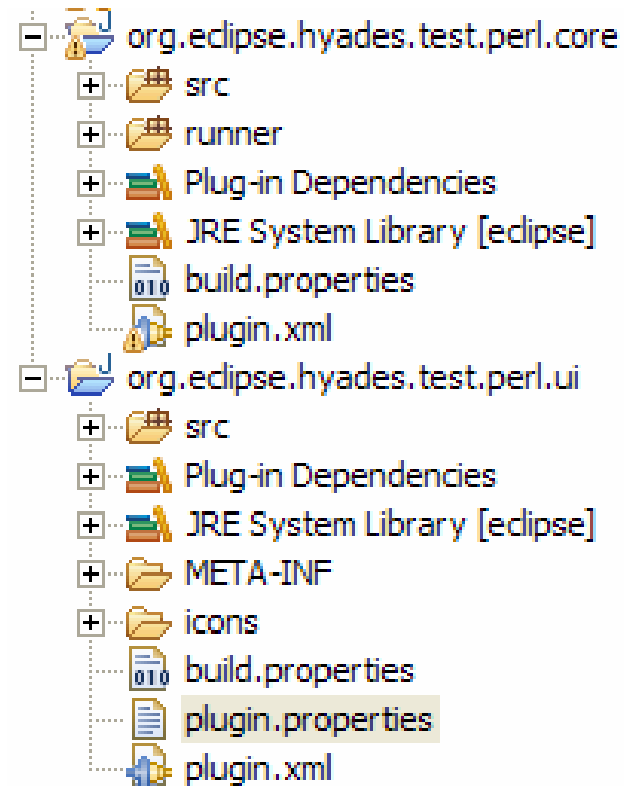
- Introductions
- Eclipse Test and Performance Tools Platform (TPTP) Overview
- TPTP Testing Tools Project
- Demo of existing TPTP Testing Tools
  
- Break (15 minutes)
  
- Test Execution Explained
- **Building a custom execution environment for a new test type**
- What's next for the TPTP Testing Tools Project?

## Building a custom execution environment for a new test type

1. Create and set up plug-in projects
2. Define the test type
3. Define wizards and editors
4. Make the test type launchable
5. Register execution components and adapters
6. Implement the adapters
7. Implement the runner
8. Modify the Agent Controller Configuration

## Create and set up plug-in projects

- Begin with an Eclipse installation including EMF and TPTP/Hyades
- Create two new plug-in projects, one for UI and one for core (with no UI contributions)
  - **org.eclipse.hyades.test.perl.ui**
  - **org.eclispe.hyades.test.perl.core**
- Create two source folders for the core plugin so we can separate the code that runs only in the workbench from code that need to be deployed into the Agent Controller
- Create icons folder for UI



# Create and set up plug-in projects

Set up the following dependencies for the projects:

UI

Core

## Dependencies

**Required Plug-ins**  
Specify the list of plug-ins required for the operation of this plug-in:

- ▶ org.eclipse.ui
- ▶ org.eclipse.core.runtime
- ▶ org.eclipse.hyades.ui
- ▶ org.eclipse.hyades.test.ui

Add...  
Up  
Down

## Dependencies

**Required Plug-ins**  
Specify the list of plug-ins required for the operation of this plug-in:

- ▶ org.eclipse.core.runtime
- ▶ org.eclipse.hyades.execution.core
- ▶ org.eclipse.hyades.execution.local
- ▶ org.eclipse.hyades.models.common
- ▶ org.eclipse.hyades.execution.harness
- ▶ org.eclipse.hyades.test.common

Add...  
Up  
Down

## Define the test type

- In order to add support for a new test type, we must define that test type by extending the `org.eclipse.hyades.ui.typeDescriptions` extension point.
- After defining the test type, we can then (optionally) specify wizards, editors, launch configurations and execution components that may be used by our test type.

```
<extension
    point="org.eclipse.hyades.ui.typeDescriptions">
  <typeDescription
    extension="testSuite"
    type="org.eclipse.hyades.test.perl.testSuite"
    icon="icons/tsuite.gif"
    description="TST_SUITE_PERL_DESC"
    name="TST_SUITE_PERL_NAME" />
</extension>
```

## Define wizards and editors

Next we need to extend the `org.eclipse.ui.newWizards` extension point so create a new wizard that will allow us to create tests of our new type.

```
<extension
  point="org.eclipse.ui.newWizards">
  <wizard
    name="Hyades Perl Test Suite"
    icon="icons/new_perlsuite.gif"
    category="org.eclipse.hyades.test.ui.wizards.new/testSuite"

    class="org.eclipse.hyades.test.perl.ui.wizard.PerlTestSuiteNewWizard"

    id="org.eclipse.hyades.test.perl.ui.wizard.PerlTestSuiteNewWizard">
    <description>
      Create a new Hyades Perl Test Suite
    </description>
  </wizard>
</extension>
```

## Define wizards and editors

Now we need to implement the class that we referenced in our extension –

For now, our wizard will simply extend `TestSuiteNewWizard`, and specify our new test type.

```
public class PerlTestSuiteNewWizard extends TestSuiteNewWizard {  
  
    /* (non-Javadoc)  
    * @see org.eclipse.hyades.test.ui.wizard.TestSuiteNewWizard#getType()  
    */  
    protected String getType() {  
        return "org.eclipse.hyades.test.perl.testSuite";  
    }  
}
```

## Define wizards and editors

If time allows, we will extend the default TPTP Test Suite editor to allow us to specify the location of the perl script we are running.

Since this tutorial is focusing on execution, if time does not allow, the example will be provided, and we will hardcode the script location to illustrate the execution capabilities.

## Define wizards and editors

Let's start up a runtime workbench and see our progress so far!

## Make the test type launchable

Next we want to make our new test type launchable by extending the `org.eclipse.hyades.test.ui.launchableType` extension point

```
<extension
  point="org.eclipse.hyades.test.ui.launchableType">
  <launchableType
    kind="test"
    type="org.eclipse.hyades.test.perl.testSuite">
    <supportedMode
      shortcutAvailable="true"
      mode="run" />
    </launchableType>
  </extension>
```

## Make the test type launchable

The TPTP launch extension points provides a lot of functionality that we won't be leveraging today including

- The ability to add new launch configuration types
- The ability to add new tabs to the launch configuration
- The ability to add additional behavior before the test execution harness is invoked.

See the `org.eclipse.hyades.test.http` and `org.eclipse.hyades.test.manual` plugins for examples.

## Make the test type launchable

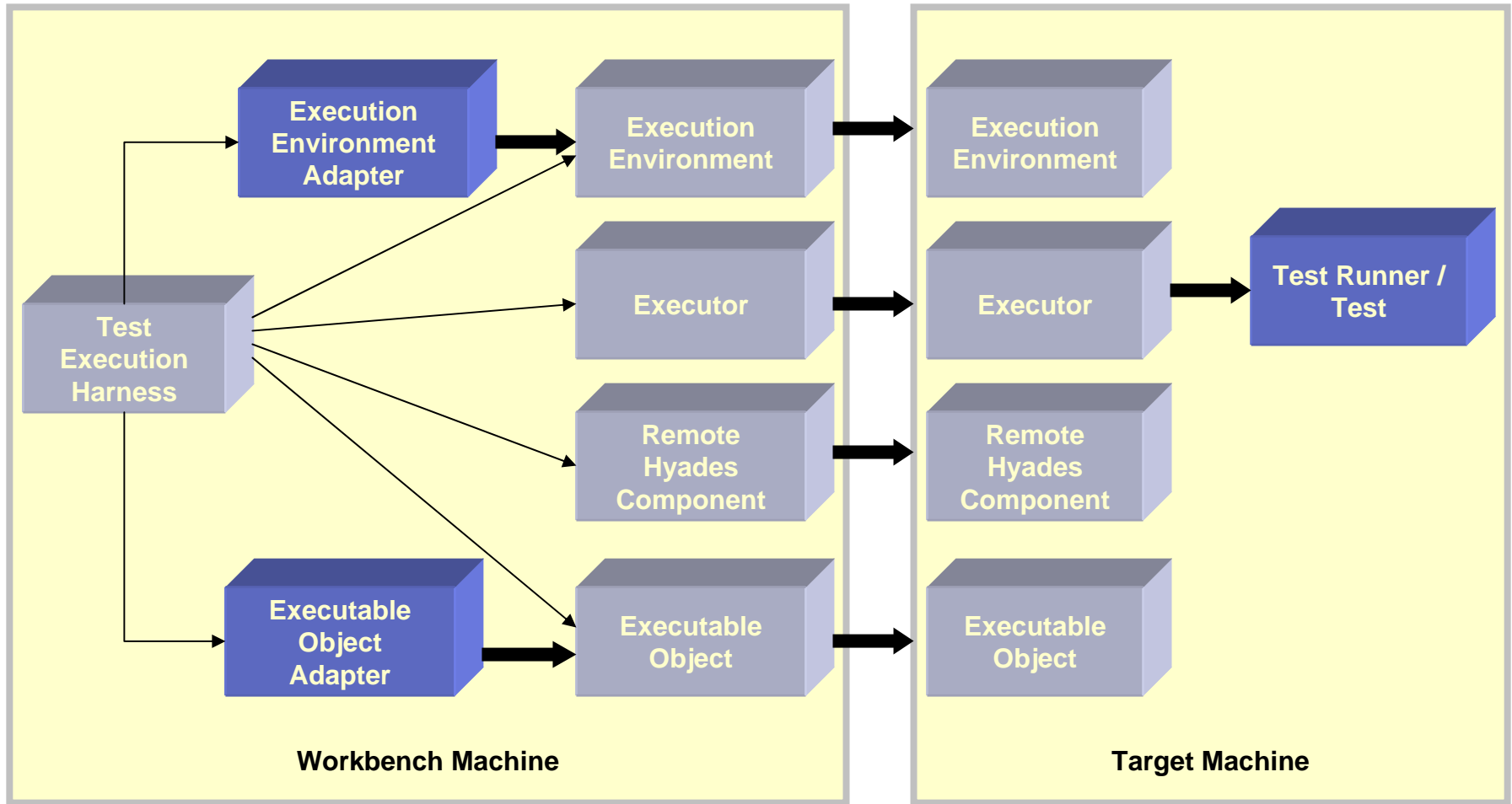
Let's start up a runtime workbench and do another progress check.

# Register execution components and adapters

Turning our attention to the core plug-in, we need to register the execution components and adapters that will support our new test type.

- We will reuse all of the TPTP Testing Tools Project's execution components as previously discussed, and we will implement our own adapters and runner to run our Perl tests.

# Test Execution reusability



# Register execution components and adapters

Extend the RegisteredExecutionComponentImpl extension point in the core plugin.xml and register the execution components that we'll be reusing

```

<!-- Execution Components -->
<extension
  id="org.eclipse.hyades.execution.harness.RegisteredExecutionComponentImpl.JAVA"
  name="JAVA"
  point="org.eclipse.hyades.execution.harness.RegisteredExecutionComponentImpl">
  <RegisteredExecutionComponentImpl
    implClass="org.eclipse.hyades.execution.core.impl.ExecutionEnvironmentImpl"
    name="JAVA_ENVIRONMENT"
    type="ENVIRONMENT"
    stubClass="org.eclipse.hyades.execution.local.ExecutionEnvironmentStub"
    skeletonClass="org.eclipse.hyades.execution.remote.ExecutionEnvironmentSkeleton"
    id="org.eclipse.hyades.execution.core.impl.ExecutionEnvironmentImpl">
    <SupportedTestType
      name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
  </RegisteredExecutionComponentImpl>
  <RegisteredExecutionComponentImpl
    implClass="org.eclipse.hyades.execution.core.impl.ProcessExecutorImpl"
    name="JAVA_EXECUTOR"
    type="EXECUTOR"
    stubClass="org.eclipse.hyades.execution.harness.TestExecutionHarnessExecutorStub"
    skeletonClass="org.eclipse.hyades.execution.remote.JavaProcessExecutorSkeleton"
    id="org.eclipse.hyades.execution.core.impl.ProcessExecutorImpl">
  <SupportedTestType
  
```

# Register execution components and adapters (cont)

```

        name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
</RegisteredExecutionComponentImpl>
<RegisteredExecutionComponentImpl
implClass="org.eclipse.hyades.execution.core.impl.JavaTaskRemoteHyadesComponentImpl"
    name="JAVA_AGENT"
    type="AGENT"
    stubClass="org.eclipse.hyades.execution.local.JavaTaskRemoteHyadesComponentStub"

skeletonClass="org.eclipse.hyades.execution.remote.JavaTaskRemoteHyadesComponentSkeleton"
    id="org.eclipse.hyades.execution.core.impl.JavaTaskRemoteHyadesComponentImpl">
    <SupportedTestType
        name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
</RegisteredExecutionComponentImpl>
<RegisteredExecutionComponentImpl
    implClass="org.eclipse.hyades.execution.core.impl.JavaProcessExecutableObjectImpl"
    name="JAVA_EXECUTABLEOBJECT"
    type="EXECUTABLEOBJECT"
    stubClass="org.eclipse.hyades.execution.local.JavaProcessExecutableObjectStub"

skeletonClass="org.eclipse.hyades.execution.remote.JavaProcessExecutableObjectSkeleton"
    id="org.eclipse.hyades.execution.core.impl.JavaProcessExecutableObjectImpl">
    <SupportedTestType
        name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
</RegisteredExecutionComponentImpl>
</extension>

```

# Register execution components and adapters

Next we need to register the adapters that we'll be implementing.

```
<!-- Adapters -->

<extension

id="org.eclipse.hyades.execution.harness.ExecutableObjectAdapter.PERL"
    name="PERL"

point="org.eclipse.hyades.execution.harness.ExecutableObjectAdapter">
    <ExecutableObjectAdapter
        name="PERL_EXECUTABLE_OBJECT_ADAPTER"

class="org.eclipse.hyades.test.perl.execution.PerlExecutableObjectAdapter"

id="org.eclipse.hyades.test.perl.execution.PerlExecutableObjectAdapter">
    <SupportedTestType
        name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
    </ExecutableObjectAdapter>
</extension>
```

# Register execution components and adapters

```

<extension
  id="org.eclipse.hyades.execution.harness.ExecutionEnvironmentAdapter.PERL"
  name="PERL"
  point="org.eclipse.hyades.execution.harness.ExecutionEnvironmentAdapter">
  <ExecutionEnvironmentAdapter
    name="PERL_EXECUTION_ENVIRONMENT_ADAPTER"
    class="org.eclipse.hyades.test.perl.execution.PerlExecutionEnvironmentAdapter"
    id="org.eclipse.hyades.test.perl.execution.PerlExecutionEnvironmentAdapter">
    <SupportedTestType
      name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
  </ExecutionEnvironmentAdapter>
</extension>

<extension
  id="org.eclipse.hyades.execution.harness.ExecutionDeploymentAdapter.PERL"
  name="PERL"
  point="org.eclipse.hyades.execution.harness.ExecutionDeploymentAdapter">
  <ExecutionDeploymentAdapter
    name="PERL_EXECUTION_DEPLOYMENT_ADAPTER"
    class="org.eclipse.hyades.test.perl.execution.PerlExecutionDeploymentAdapter"
    id="org.eclipse.hyades.test.perl.execution.PerlExecutionDeploymentAdapter">
    <SupportedTestType
      name="org.eclipse.hyades.test.perl.testSuite">
    </SupportedTestType>
  </ExecutionDeploymentAdapter>
</extension>
  
```

## Register execution components and adapters

The third adapter we just registered (`ExecutionDeploymentAdapter`) is not part of the Execution Environment per se, and it does not adapt an execution component. Rather, it provides the deployment behavior for our test type.

## Implement the adapters

- As we discussed earlier, the primary role of the Execution Environment is to specify extensions to the classpath for the test, and the test execution harness calls into an adapter class to make those extensions.
- Since our tests are implemented in Perl, we don't need to make any test-specific modifications to the test JVM's classpath.
  - We will modify the Agent Controller's classpath to pick up the runner that we'll implement, but since that classpath is the same for all executions of our test type, it is not necessary to modify the classpath every time a test is launched.
- So – we will create the Execution Environment Adapter class, but the class will not modify the Execution Environment

`org.eclipse.hyades.test.perl.execution.PerlExecutionEnvironmentAdapter`

## Implement the adapters

- Next we will implement the Executable Object Adapter.
- This class must pass the JVM arguments, main class and arguments which will be user to launch our runner.

# Implement the adapters

```

public class PerlExecutableObjectAdapter implements IExecutableObjectAdapter {
    public static final boolean DEBUG = false;

    /* (non-Javadoc)
     * @see
     org.eclipse.hyades.execution.harness.IExecutableObjectAdapter#setupExecutableObject(org.eclipse.hyades.execution.core
     .IExecutableObject, org.eclipse.hyades.models.common.configuration.CFGClass,
     org.eclipse.hyades.models.common.facades.behavioral.IImplementor,
     org.eclipse.hyades.models.common.testprofile.TPFDeployment)
     */
    public void setupExecutableObject(IExecutableObject execObj,
                                     CFGClass rootResource, IImplementor theImplementor,
                                     TPFDeployment deployment) throws ClassCastException
    {
        JavaProcessExecutableObjectStub objectStub = (JavaProcessExecutableObjectStub)execObj;
        String jvmArgs = "";

        if (DEBUG)
            jvmArgs += "-Xdebug -Xrunjdw:transport=dt_socket,server=y,suspend=y,address=8005 ";

        jvmArgs += "org.eclipse.hyades.test.perl.runner.HyadesPerlRunner " + theImplementor.getResource();
        //jvmArgs += "org.eclipse.hyades.test.perl.runner.HyadesPerlRunner -version";

        objectStub.setArgs(jvmArgs);
    }
}

```

## Implement the adapters

- The last adapter we have to implement is the Execution Deployment Adapter. This adapter is responsible for deploying the files necessary to run the test. For the purposes of this tutorial, we will not write code to perform the deployment, however, if time is available, we will review a deployment adapter and show its use.
- But for now, we will again provide an empty adapter implementation.

`org.eclipse.hyades.test.perl.execution.PerlExecutionDeploymentAdapter`

## Implement the runner

- When using the TPTP Testing Tools Project's execution components, the runner is called by the executor in a new JVM with the appropriate classpath and with the command line arguments necessary to start the test itself.
- The runner class should extend the HyadesRunner class from the `org.eclipse.hyades.test.common` plugin, which contains some behavior that is expected by the TPTP Testing Tools Project's execution components.

# Implement the runner

- Minimally, the runner should perform the following:
  - Construct an instance of itself, and call the super constructor in its constructor
  - Launch the test
  - Pass execution history events back to be model loaded (via `writeExecEvents` inherited from `HyadesRunner`)

# Implement the runner

- Execution history events are passed to the workbench in the form of XML fragments.
  - `org.eclipse.hyades.test.common` has utility classes in the `org.eclipse.hyades.test.common.event` package that will properly construct these fragments for you
  - Or you can construct them yourselves, adhering to the schema published here:  
[http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/hyades-home/docs/components/execution\\_environment/ExecutionEvent.xsd](http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/hyades-home/docs/components/execution_environment/ExecutionEvent.xsd)

## Implement the runner

- Depending on time available, we will either build a runner ourselves, or review a working runner for our Perl example.

# Modify the Agent Controller Configuration

- Our Runner needs to run on the target machine, and thus we need to ensure that the jar file is present and available on the Agent Controller's classpath
- From an install perspective, the way to accomplish this is to drop the jar file into the Agent Controller lib directory, and then add a reference to it from the pluginconfig.xml file

```
<Variable name="CLASSPATH" position="append"  
value="%RASERVER_HOME%\plugins\org.eclipse.hyades.test\lib\perl.runner.jar"/>
```

# Modify the Agent Controller Configuration

- From a development perspective, (when you're making frequent changes and when you're running the test only on the workbench machine), it's much more convenient to instead directly reference the bin directory that contains the runner.

```
<Variable name="CLASSPATH" position="append"  
  value="C:\HyadesPerl\org.eclipse.hyades.test.perl.execution\bin"/>
```

## Building a custom execution environment for a new test type

- Let's try it out!

# Agenda

- Introductions
- Eclipse Test and Performance Tools Platform (TPTP) Overview
- TPTP Testing Tools Project
- Demo of existing TPTP Testing Tools
  
- Break (15 minutes)
  
- Test Execution Explained
- Building a custom execution environment for a new test type
- **What's next for the TPTP Testing Tools Project?**

## Future Work

- Enhanced Data Collection and Control – dynamic deployment of agents, agent metadata, discoverability of agent capabilities, support for other languages
- Enhanced Communication – support for communication across firewalls, optional encryption of data transfers, and pluggable transport layers.
- Export of data using XMI
- Test Reporting and Charting enhancements
- Choreography Framework
- Richer RCP based Manual Test client
- Improved deployment capabilities

# Resources

- Read the website
  - <http://www.eclipse.org/tptp>
  - <http://www.eclipse.org/hyades>
- Join the mailing lists
  - [http://eclipse.org/tptp/home/project\\_info/general/mailnews.html](http://eclipse.org/tptp/home/project_info/general/mailnews.html)
- Read the newsgroup
  - <news://news.eclipse.org/eclipse.tptp>
- Visit the developer resources
  - Linked from main TPTP website

# BACKUP

