RCE Developer Guide

Build 8.3.0.201907051305

Table of Contents

1. Preface 1
    1. Abstract 1
    2. Intended audience 1
    3. License Information 1
    4. Compatible Operating Systems 1
        4.1. Support of 32 Bit Operating Systems 2
        4.2. Known Issues 2
            4.2.1. KDE on Red Hat Enterprise Linux 7 2
            4.2.2. KDE with Oxygen 2
            4.2.3. Jython scripts are executed sequentially 2
            4.2.4. 32-bit Java is not supported 2
2. Introduction 3
    1. Getting Started with RCE development 3
        1.1. Preparing Eclipse 3
        1.2. Importing and building RCE 3
        1.3. Running RCE from Eclipse 5
        1.4. Configuring Workspace Mechanic (optional) 6
        1.5. Configuring Code Formatting and CheckStyle Rules (optional) 6
        1.6. Building a standalone RCE version from Eclipse 7
        1.7. Building from the command line 8
    2. Common Classes and Interfaces 8
        2.1. General 8
        2.2. Components and Workflows 8
        2.3. Utilities 9
3. Build and Infrastructure 10
    1. Build Structure and Dependencies 10
        1.1. Overview: Build and Versioning Scopes 10
        1.2. Changing Version Dependencies 10
            1.2.1. Switching "RCE Core" to another version of "RCE Platform" 10
    2. Release and Versioning Process 10
        2.1. Overview: The Release and Versioning Process 10
        2.2. Step 1: Trunk preparations 10
        2.3. Step 2: Release candidate building and testing 10
        2.4. Step 3: Publishing the final release 10
        2.5. Step 4: Post-release actions 11
            2.5.1. Upgrading Version Numbers (and Verification) 11
        2.6. Creating a maintenance/hotfix release 11
    3. Build Process FAQ / Tips and Tricks 12
4. Coding Guidelines 14
    1. Developing a new Component 14
        1.1. Component Bundle Setup 14
        1.2. Creating the *execution* Bundle 14
        1.3. Creating a *common* Bundle 18
        1.4. Creating a *gui* Bundle 18
        1.5. Adding your new component to SVN 20
    2. Logging 20
        2.1. General Configuration 20
        2.2. Verbose Logging 20
5. Quality Assurance and Testing 22
    1. Automated GUI Testing 22
        1.1. Getting started 22
    2. Integrated Test Script Runner 23
        2.1. Setup and Configuration 23
        2.2. Usage 24
6. Licensing and Copyright 25

1. Copyright Statements 25
1.1. Current Year Definition 25

List of Tables

4.1. Useful Verbose Logging Identifiers 21

Chapter 1. Preface

This chapter gives an introduction to RCE.

1. Abstract

RCE (Remote Component Environment) is an open source software to help engineers, scientists and others to create, manage and execute complex calculation and simulation workflows. A workflow in RCE consists of components with predefined inputs and outputs connected to each other. A component can be a simulation tool, a tool for data access, or a user-defined script. Connections define which data flows from one component to another. There are predefined components with common functionalities, like an optimizer or a cluster component. Additionally, users can integrate their own tools. RCE instances can be connected between each other. Components can be executed locally or on remote instances of RCE (if the component is configured to allow this). Using these building blocks, use cases for complex distributed applications can be solved with RCE.

2. Intended audience

This document is intended for developers who would like to extend RCE according to their needs and/or contribute to RCE's development.

3. License Information

RCE is published under the Eclipse Public Licence (EPL) 1.0. It is based on Eclipse RCP 3.7.2 (Indigo SR2), which is also published under the Eclipse Public Licence (EPL) 1.0. RCE also makes use of various libraries which may not be covered by the EPL; for detailed information, see the file "THIRD_PARTY" in the root folder of an RCE installation. (To review this file without installing RCE, open the RCE release .zip file.)

For downloads and further information, please visit http://www.rcenvironment.de.

4. Compatible Operating Systems

RCE releases are provided for Windows and Linux. It is regularly tested on

- Windows 7
- Windows 10
- Windows Server 2012 R2
- SUSE Linux Enterprise Desktop ("SLED") 12 SP2
- Ubuntu 16.04 LTS
- Red Hat Enterprise Linux 6

4.1. Support of 32 Bit Operating Systems

Starting with release 8.0.0, RCE is only shipped for 64 bit systems. If you still require 32 bit packages, you can continue to use previous RCE releases, but there will be no standard feature or bugfix updates for them.

4.2. Known Issues

4.2.1. KDE on Red Hat Enterprise Linux 7

On Red Hat Enterprise Linux 7 with KDE 4, RCE (like any other Eclipse-based application) can cause a segmentation fault at startup. If you encounter such an issue, you can try choosing a different GTK2 theme:

1. Open the **System Settings** application (systemsettings).
2. Go to **Application Appearence**
3. Open **GTK** page
4. Switch the GTK2 theme to "Raleigh" or "Adwaita" and click on **Apply**

4.2.2. KDE with Oxygen

On Unix Systems using KDE as desktop environment and Oxygen as theme it can happen that RCE crashes when certain GUI elements are shown. It is a known issue in the theme Oxygen and happens on other Eclipse-based applications as well. If you encounter such an issue, please choose a different theme like "Raleigh" or "Adwaita".

4.2.3. Jython scripts are executed sequentially

The Script component can use Jython for the evaluation of scripts and the pre- and postprocessing of integrated tools always uses Jython. Due to a known bug in the Jython implementation it is not possible to execute several Jython instances in parallel. Therefore, the execution will be done sequentially. If several Script components should be executed in parallel, Python should be used instead.

4.2.4. 32-bit Java is not supported

Running RCE with a 32-bit Java Runtime Environment doesn't work. On some operating systems an error dialog will be displayed in this case, on some other systems nothing will happen at all. Therefore, always make sure a 64-bit Java Runtime Environment is used to run RCE.

Chapter 2. Introduction

1. Getting Started with RCE development

This section covers setting up a development environment for running, modifying and extending RCE on your local machine.

1.1. Preparing Eclipse

- Make sure you have a Java Development Kit [http://www.oracle.com/technetwork/java/javase/downloads/index.html], Version 8, installed on your system.

  Note

  If you are installing a JDK on a centrally administered computer that already has Java installed, it is usually a good idea to *uncheck* the "Install public JRE" option during installation.

- Download and run "Eclipse for RCP/RAP Developers", Version "Mars.2 (4.5.2)" or later from https://www.eclipse.org/downloads/. Using the latest Eclipse release instead is usually fine; however, there may be slight differences between this guide and that version's UI.
- Adapt your eclipse.ini file (see http://wiki.eclipse.org/Eclipse.ini for details):
  - Set the proper path to your JDK installation (the `-vm` parameter; e.g. `<your path>/jdk8/bin` immediately below the `-vm` line).
  - Change the maximum heap size to at least 2 GiB (adapt the `-Xmx` line somewhere below `-vmargs`, e.g. to `-Xmx2048m`, or add it if it does not exist yet)
- Optionally, install these plugins (for example using "Help > Eclipse Marketplace"):
  - Workspace Mechanic [https://code.google.com/a/eclipselabs.org/p/workspacemechanic/] - used to apply common workspace settings. Not strictly required, but convenient.
  - Checkstyle-CS [http://eclipse-cs.sourceforge.net/] - Eclipse integration for Checkstyle (version 6.19 or above; see the note in section "Configuring CheckStyle"). Not required for local development, but mandatory before committing changes into the central code base.

1.2. Importing and building RCE

- If you already have other projects in your Eclipse workspace, it is recommended to create a new workspace for RCE. There are some necessary global settings that may interfere with the other projects (e.g. the so-called "target platform").
- Disable "Project > Build Automatically" in the main menu to speed up the next steps.
- There are currently three ways to properly import the complete RCE source code: from the SVN repository, from provided zip files, or from GitHub. At the moment, only the SVN approach provides access to the current development tree; for the time being, the zip files and the GitHub repository are only updated on release.
- Option 1 - Importing from the RCE SVN repository, if you have access to it (as the repository is currently hosted internally at DLR):

- Install Subclipse [http://subclipse.tigris.org/] (or alternatively Subversive [http://www.polarion.com/products/svn/subversive.php]) if you don't already have an Eclipse SVN plugin installed. Note that when using Subclipse, you may have to switch to the "SvnKit" SVN interface in the "Team > SVN" preferences; this is normal.
- Open the "SVN Repositories" view (Window > Show View > Other > SVN).
- Add `https://svn.sistec.dlr.de/svn/rce/new/rce/trunk` as a new SVN repository location.
- Expand the location entry and select all projects (the entries starting with "de.rcenvironment") inside of it.
- Right-click the selected projects, select "Checkout" and confirm if necessary. You should now see a long list of projects in the "Package Explorer" on the left.

  Note

  If the checkout was performed correctly, there should be a small "M" (for "Maven") on most projects' icons, and also a small "J" (for "Java") on most of them.

- Option 2 - Importing from the zip files provided with each release:
  - Browse to the "source" sub-folder of a release's download location, for example `https://software.dlr.de/updates/rce/8.x/products/standard/releases/8.1.0/source/` for the 8.1.0 release. Download both zip files, "source" and "additions"; the latter contains binary artifacts like the Dakota and TiGLViewer executables.
  - Extract both archives into the *same* target directory. You should see a list of more than 200 sub-folders, all except one beginning with "de.rcenvironment".
  - In Eclipse, select "File > Import > General > Existing Projects into Workspace" and choose the directory that you unpacked the archives into. You should see a long list of projects, once again, all except one beginning with "de.rcenvironment". Select all projects confirm the import.

    Note

    Make sure not to change the "Search for nested projects" option in the import dialog; it must *not* be selected/checked.

- Option 3 - Importing from GitHub:
  - For successfully importing the RCE project, you need a Git client with LFS support.
  - Clone `https://github.com/rcenvironment/rce` into a directory of your choice.

    Note

    The clone settings of certain Git clients (e.g. TortoiseGit), have an "LFS" option. Make sure that this option is enabled before you clone.

  - Check out the `master` branch; by default, this points to the source code of the latest release.
  - In Eclipse, select "File > Import > General > Existing Projects into Workspace" and choose the directory that you unpacked the archives into. You should see a long list of projects, once again, all except one beginning with "de.rcenvironment". Select all projects confirm the import.

> **Note**
>
> Make sure not to change the "Search for nested projects" option in the import dialog; it must *not* be selected/checked.

- After you have successfully imported the RCE projects using one of the above methods, the next step is to set the RCE target platform in your workspace. A target platform provides external artifacts like the Eclipse RCP framework and various libraries. To get started with RCE development, the easiest way is to use a precompiled target platform. For convenience, there is a Eclipse .target file inside the code base that always points at an appropriate precompiled target platform release. Follow these steps to apply it:
  - In the Project Explorer, navigate to the `de.rcenvironment/eclipse/tp/remote` folder.
  - Open the `default_release_or_snapshot.target` file by double-clicking it.
  - Select the "Locations" list entry starting with "`https://software.dlr.de/`" and click "Update". After a while, the list entry's description should end with something similar to "242 plugins available" (the exact number may vary). Save the file if necessary.
  - Click "Set as target platform" in the top right corner. You can close the .target file after this.
- It is currently necessary to set the global compiler compliance level to 1.7. To apply this, select "Window > Preferences" from the menu, and navigate to the "Java > Compiler" tab. (This step will become unnecessary in the upcoming 9.0.0 release.)
- Enable "Project > Build Automatically". Eclipse will start building all projects against the new target platform, which provides all required libraries and OSGi bundles.
- At this point, most projects will have a red error marker. To fix this, open the "Problems" view ("Window > Show View > Problems"). You should see a lot of "Plugin execution not covered by lifecycle configuration" entries. Right-click one of them, select "Quick Fix" from the context menu, select "Discover new m2e connectors" and click "Finish". Eclipse should present one or more installation options with "Tycho" in their name. Confirm their installation and restart eclipse.

  > **Note**
  >
  > You only need to do this once per Eclipse installation.

- After this, all RCE bundles should compile without errors (with the exceptions noted below), and you are ready to start developing. If this is not the case, try running "Project > Clean > Clean all projects" from the main menu.

  > **Note**
  >
  > On Linux platforms, there will be compilation errors in some Windows-only Excel and TiGLViewer bundles (5 and 3 projects, respectively). We don't have an elegant solution for this problem yet. You can simply close these projects to get rid of the errors, as they won't be loaded at runtime anyway.

1.3. Running RCE from Eclipse

Before proceeding to the more detailled settings, try running RCE from Eclipse to verify your setup.

- There are several pre-defined launch configurations for RCE. To find them, navigate to `de.rcenvironment/eclipse/launch` in the "Project Explorer" on the left.
- A good starting point is the "default" configuration. Expand the "default" sub-folder, right-click the "rce.default.launch" file and choose "Run As > rce.default.launch" from the context menu.

- RCE should now start and prompt for an RCE workspace location. Confirm the default value or choose another empty folder.

1.4. Configuring Workspace Mechanic (optional)

> **Note**
>
> The Workspace Mechanic project in its original form is not being maintained anymore, and the original project site is gone. However, it has been forked and is being continued by a new maintainer at this location [https://github.com/alfsch/workspacemechanic/]. While this is not an "official" successor, it seems to be the de-facto location of this project now.

Workspace Mechanic (which can be installed via Eclipse Marketplace from this location [https://marketplace.eclipse.org/content/workspace-mechanic]) is an Eclipse plugin that automates common settings in local workspaces. For RCE, the most important settings are the Java code formatting rules and templates. Other settings are provided for convenience, like disabling the console output limit, or showing line numbers in the editor.

Configuring Workspace Mechanic consists of copying a set of "rule" files to a location where the plugin can find them. There are two options for this:

- If you want to apply the rules to all Eclipse installations on your machine, use the `.eclipse/mechanic` subfolder in your home directory; by default, this is `C:\Users\<user id>\.eclipse\mechanic`.
- To apply the rules to a single Eclipse installation only, use `<eclipse installation folder>\configuration\com.google.eclipse.mechanic\mechanic`.

Using your system's file browser, navigate to the folder of your choice. Using any SVN tool, check out `https://svn.sistec.dlr.de/svn/rce/new/meta/eclipse/mechanic` into a sub-folder called "`checkout`" within it. (Note that the actual name of the sub-folder is not relevant; adapt if you like.) This sub-folder now contains common rules on its top level, and optional or experimental rules in sub-folders. Copy all common rules to the parent folder (the one you started in), and add any optional rules that you want to apply as well. (TODO add and describe batch/shell files for this.)

> **Note**
>
> These rule files will most likely be integrated into the main project at some point, making this extra checkout step unnecessary. Please note that these rule files are currently not available as part of the GitHub source code mirror [https://github.com/rcenvironment/rce] or the released source zip files either, which makes them inaccessible unless you have access to the internal SVN server.

The next time you open a workspace, Workspace Mechanic should pick up these rule files and show a notice asking if it should apply them. See the plugin's web site [https://github.com/alfsch/workspacemechanic/] for further information.

1.5. Configuring Code Formatting and CheckStyle Rules (optional)

If you only plan to try out or modify RCE locally, you can safely skip this section. If you plan to commit your changes to the central code base, however, you need these settings to get your code accepted into the repository. Code that does not match the style guidelines will be refused on commit. The Checkstyle-CS plugin simplifies development by highlighting violations that need to be fixed.

To configure Checkstyle-CS for RCE:

- Open the Checkstyle preferences (Window > Preferences > Checkstyle).

- Click "New" on the right side. Enter "RCE" as the name of the configuration.
- Choose "Project relative configuration", click "Browse" and choose `de.rcenvironment/checkstyle/checks.xml`.
- Click "Ok" in the main dialog. The list of configurations should now have three entries; select "RCE" and click "Set as Default" on the right side.
- Close the preferences with "Ok" and confirm the rebuild.

> **Note**
>
> We are currently using version 6.19 of the Eclipse CheckStyle plugin within the development team. Using a newer version (e.g. 8.0) works as well, but you may see error markers for constructs that do not actually violate the RCE code guidelines. Versions > 8.0 do not work with current code guidelines. We will most likely adapt/migrate the CheckStyle settings in the near future.

To configure the Eclipse source code formatter:

- Open the code formatter preferences (Window > Preferences > Java > Code Style > Formatter).
- Click "Import", browse to your `de.rcenvironment/eclipse/` checkout folder and choose the "`eclipse-formatter.xml`" file.
- You should now see "RCE" as the "Active Profile".
- Click "Apply and Close" to activate the settings.

> **Note**
>
> There are rare cases where these code formatter settings lead to source files that are not being accepted by our CheckStyle rules. These inconsistencies are being collected and tracked in issue #0005898, and will be fixed/addressed in a future update.

1.6. Building a standalone RCE version from Eclipse

You can also build a standalone version of RCE from Eclipse using Maven.

- There are several pre-defined build configurations for RCE. To find them, navigate to `de.rcenvironment/eclipse/build` in the "Project Explorer" on the left.
- To run a .launch file, right-click on it and choose the single entry in the "Run As" submenu of the context menu that appears.
- Building a standalone RCE installation (which is called a "product" in Eclipse RCP terms) generally consists of two steps: Providing a so-called "platform repository", and then building the actual product on top of it. There are two possible approaches for this:
  - Using the .launch files in the "`using default remote repository builds`" sub-folder, it is possible to avoid building your own platform repository, and fetch a pre-built one from the main repository servers (currently hosted at DLR) instead. One reason for this can be be to ensure that you are building against the exact same platform repository as a certain RCE release. Another reason is to simplify your local development setup if you have no reason to customize the target platform setup (e.g. by adding libraries).
  - The most flexible way to build RCE from source is to compile a local platform repository yourself. This is simply done by navigaring to the "`de.rcenvironment.platform/eclipse` folder (from the root of your workspace) and executing the single .launch file that is located there. On the first run, the build process may download quite a few resources from Maven Central; subsequent runs should be fairly quick. Once the build has finished (there should be a "SUCCESS" message near the bottom of the console output), you can build the main product

using the .launch files in the "`using local repository builds`" sub-folder of the previously mentioned location.

- Regardless of whether you use a pre-build platform repository or compile one locally, running the "`RCE - build default product (snapshot, using <...>)` .launch file is the best way to produce a standard local product build. The other .launch files are intended for more specific use cases.
- The main product build takes several minutes to complete. Once it has finished, you will find the .zip files containing the final product in the folder `de.rcenvironment/target/de.rcenvironment.modules.repository.mainProduct/products` (You may have to refresh the de.rcenvironment project in the Project Explorer to see it.)

1.7. Building from the command line

Building RCE completely from the command line is somewhat complicated as it is a multi-step process in which later steps must reference the output artifacts of previous steps. To simplify this, the command-line build will be further encapsulated by front-end scripts, which will then be documented here. In the meantime, please use the steps described above to trigger the build process from Eclipse.

Note that both the Eclipse-based and the command-line build trigger the same Maven steps in the background. Because of this, the build triggered from Eclipse produces the exact same artifacts as a command-line build (which is used in Continuous Integration and for releases).

Note

TODO document the new command-line build when ready

2. Common Classes and Interfaces

This section lists classes and interfaces that every RCE developer should be familiar with.

Note that at this point, this list is probably incomplete. If you come across a class you wish you had known earlier, please let us know.

2.1. General

ResolvableNodeId, InstanceNodeId, InstanceNodeSessionId, LogicalNodeId, LogicalNodeSessionId (former: NodeIdentifier)	Explanation for NodeIdentifier: This interface represents the "identity" of a node, and is used whenever nodes are specified in API calls. In general, these node identifiers (or "node ids") are stored and reused by nodes, so they are persistent unless the node's operator deletes its settings folder. From a developer's perspective, the inner format of these ids is usually not relevant. TODO: Replace with explanation referring to current node identifier approach Package: de.rcenvironment.core.communication.common

2.2. Components and Workflows

TypedDatum — Represents a chunk of data that is passed between the components of a workflow. This is the central abstraction of all data passing, so you will encounter it when you start writing or modifying workflow components.

Package: de.rcenvironment.core.component.datamodel (Note: may be moved in 5.0.0)

2.3. Utilities

ThreadPool/SharedThreadPool — This is a central thread pool that should be used for all asynchronous operations (except for the SWT GUI thread itself, and GUI-embedded "background tasks"). Always use this instead of creating Thread or Executor/ExecutorService instances.

(TODO add code example(s), explain @TaskDescription, ...)

Package: de.rcenvironment.core.utils.common.concurrent

TempFileUtils — This utility class should be used whenever a temporary file or directory should be created. Its main benefit is that it allows for managed cleanup of leftover temporary files/directories (Note: This is not yet implemented!). Additional benefits are convenience functions (like generating a temporary file with a given filename or name pattern), central handling of cleanup issues (like undeletable files), and making sure that all temporary files are created in a consistent location.

Package: de.rcenvironment.commons (Note: will be moved in 5.0.0)

Chapter 3. Build and Infrastructure

1. Build Structure and Dependencies

1.1. Overview: Build and Versioning Scopes

TODO migrate/add content

1.2. Changing Version Dependencies

1.2.1. Switching "RCE Core" to another version of "RCE Platform"

TODO migrate/add content

2. Release and Versioning Process

2.1. Overview: The Release and Versioning Process

Creating an RCE release and preparing for the next one is a process that can be split into four distinct steps (or phases):

- Trunk preparations - actions that take place in the development trunk before the release branch is split off.
- Release candidate building and testing - creation of the release branch, building RCs on the CI server, and applying fixes if necessary.
- Final release - creating and publishing the final release build, SCM tagging, posting announcements etc.
- Post-release actions - preparing the trunk for the next release.

2.2. Step 1: Trunk preparations

TODO migrate/add content

2.3. Step 2: Release candidate building and testing

TODO migrate/add content

2.4. Step 3: Publishing the final release

TODO migrate/add content

2.5. Step 4: Post-release actions

2.5.1. Upgrading Version Numbers (and Verification)

By convention, the version numbers of all plugins and features are increased in the trunk immediately *after* a release has been performed. This way, every snapshot build is associated with the upcoming release. For example, all snapshot builds after the 8.1.0 release should be named named "8.2.0.xxx_SNAPSHOT".

The version upgrade process of RCE Core is mostly automated:

- Open a shell or command window in "/de.rcenvironment.core/maven/utils/" and run the appropriate "upgrade-core" script for your platform (.bat on Windows, .sh on Linux). Usage:

  `upgrade-core { old core version } { new core version }`

  Example: `upgrade-core 8.0.0 8.1.0`

  > **Note**
  >
  > Due to major structural changes in 8.1.0, the version upgrade script tagged as part of the 8.1.0 release may not be updated to perform the version upgrade to 8.2.0 yet; adapting this script will be one of the first actions after the release.

- Build the platform repository and the full product locally to verify that the build setup is consistent; see the "getting started" section on how to do this.
- Create a Mantis issue "release x.y.z" for the new version if it does not exist yet (usually, it won't).
- Commit the changes under this issue.
- Verify the CI/Jenkins build
  - (Option 1) TODO update this section for 8.1.0+
  - (Option 2) Keep an eye on the standard periodic builds (nightly, "onCommit", ..) and see if they complete normally. IMPORTANT: This is ONLY appropriate if you are around/available for handling possible problems!

2.6. Creating a maintenance/hotfix release

When creating a release that is not based on the current development trunk, the release process is slightly different. Such "maintenance" or "hotfix" releases must *always* be derived from a stable release.

To create a new release based on a previously-released version:

- Create a release branch (similar to a normal release) by copying the SVN release tag folder.
- Check out this release branch to your local machine.
- As in step 4 of the standard process, upgrade the local version numbers, create a Mantis issue for the release and commit the version changes to it. Note the change in ordering: in a standard release, version numbers are upgraded in the trunk *after* the release; in a maintenance/hotfix release, they are upgraded *before* the release, inside the release branch.
- Apply and commit the fixes or changes you want in the release; if you want to include specific trunk changes, consider transfering them by using diff patches.
- Perform standard step 2 (RC building and testing).

- When everything is tested, perform standard step 3 (final release).
- If changes were made in the release branch that should also be in the trunk, merge them back *selectively* . Unlike a normal release, you cannot simply merge all branch changes back to trunk; take special care not to mix up version numbers when merging.

3. Build Process FAQ / Tips and Tricks

This section gives answers and hints to common build issues.

Q: After running a local product build, where do I find the generated product zips and files?

A: The generated files are located in `de.rcenvironment/target/de.rcenvironment.modules.repository.mainProduct/products` .

Q: When running a product build, how can I change the server URL where p2 artifacts (e.g. the "target platform") are loaded from?

A: To support typical build use cases, p2 server URLs are normally assembled from two parts: a common URL "root" part, and a repository-specific URL segment. The default values for these are defined in the build pre-processor script at `/de.rcenvironment/maven/preprocessor/scripts/RCEBuildPreprocessor.groovy` .

> **Note**
>
> For example, the default "target platform" repository URL for the 8.1.0 release is the concatenation of the default URL "root" part `https://software.dlr.de/updates/rce/8.x/repositories/` and the specific repository segment `releases/8.1.0`. The same pattern using the same "root" URL, but different specific segments would also be used for other repositories. However, as of 8.1.0, the "platform" repository is the only one used during the default build. The only other valid option is "intermediate", which is only used in special builds.

There are three ways to change these URLs, depending on the build use case.

- If you want to switch to a different server that provides all of the required repositories, you can simply override the URL "root" part, and all p2 repositories will be loaded from there. This can be done by setting the Maven property `rce.maven.repositories.default.rootUrl`.
- To override the URL root path of a single repository, set the Maven property `rce.maven.repositories.<id>.rootUrl`, with `<id>` being "platform" or "intermediate".
- Alternatively, you can also override the complete URL of a repository by setting the Maven property `rce.maven.repositories.<id>.url` with the same ids as above.

All of these settings can be combined, with more specific settings overriding the more general ones (e.g. a custom repository URL overrides a custom root URL).

Note that these approaches are only intended for adapting the build to your build environment, or for local building and testing. To change the repository paths permanently (e.g. when preparing a new release), edit the default values in the build pre-processor script at `/de.rcenvironment/maven/preprocessor/scripts/RCEBuildPreprocessor.groovy`. Note that there are two sets of specific repository URL segments which are used for snapshot and RC/release builds, respectively.

Q: I created a new snapshot / RC / release build of the "platform" repository. What do I have to edit to make the product build use it?

A: The default repository references are configured in the `defaultRepositoryUrlSuffixes` map within the build pre-processor script at `/de.rcenvironment/maven/preprocessor/scripts/RCEBuildPreprocessor.groovy`. Snapshot and RC/release references are configured separately to support developing against snapshot builds. These references are the repository-specific URL suffixes; see the question above for examples.

Note that when preparing for a new major release, you may also have to adapt the "root" URL part (e.g. changing it from `<...>/rce/8.x/repositories/` to `<...>/rce/9.x/repositories/`), and have to deploy the referenced repository builds to that new location.

Chapter 4. Coding Guidelines

1. Developing a new Component

1.1. Component Bundle Setup

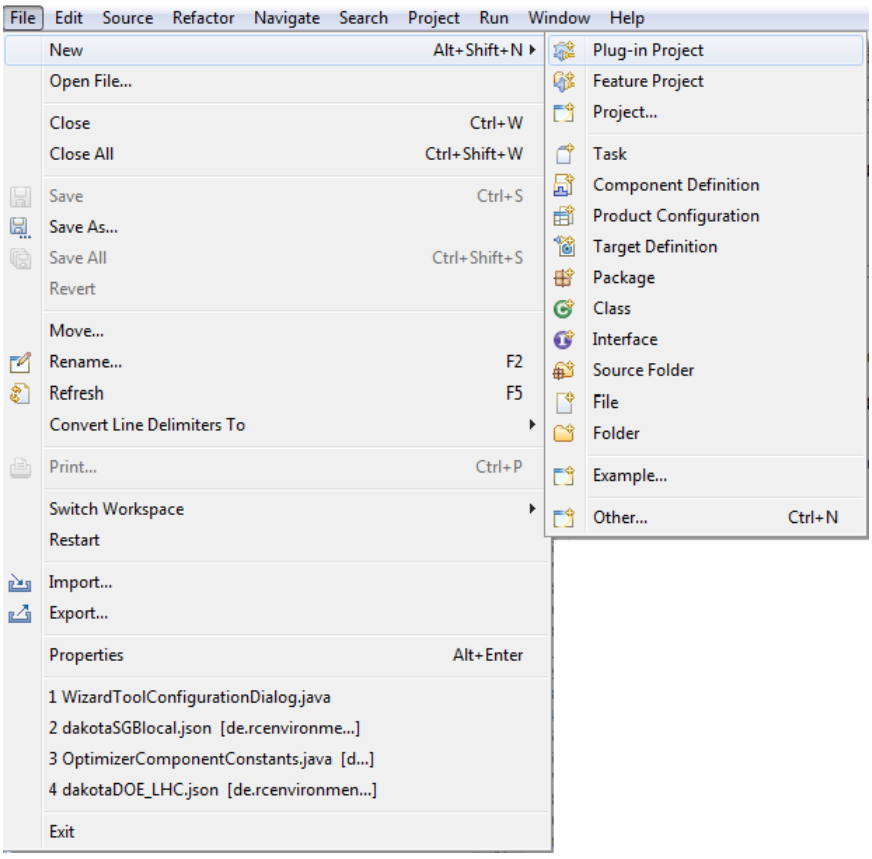
An RCE component usually consists of up to three OSGi bundles: named the *execution*, *gui*, and *common* bundles:

- Execution: Contains the component's lifecycle.
- GUI: Contains the user interface to configure the component. If a component does not require a user interface this bundle can be ommitted.
- Common: Contains code resources which are used by both the execution and the GUI bundle. This bundle is optional, too.

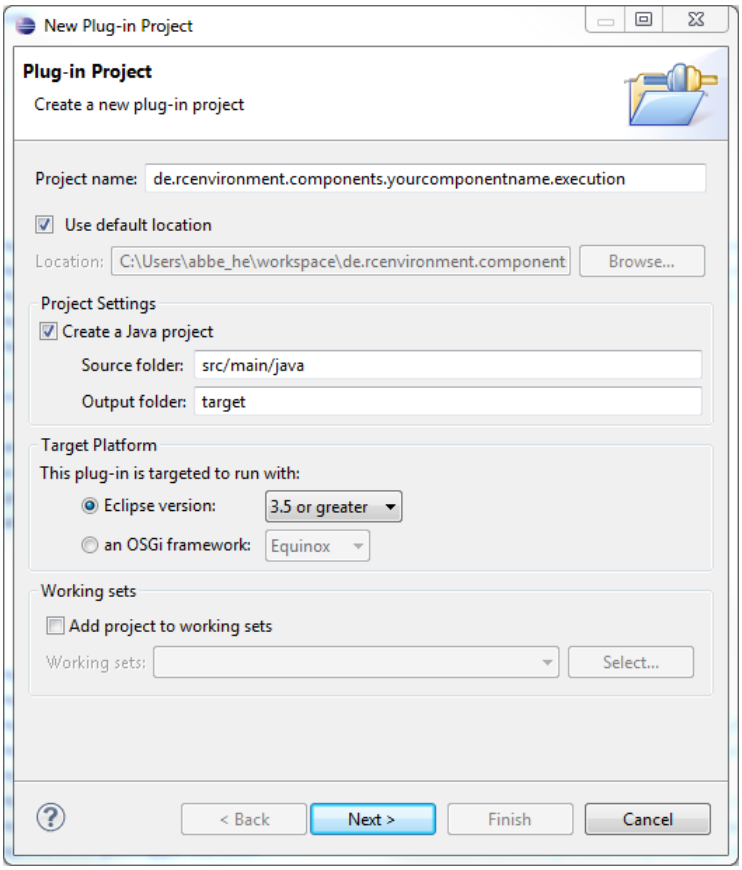
Note that in the context of Eclipse, OSGi bundles are called *Plug-In Projects*, and can be extended with Eclipse-specific features. In practice, the terms *bundle* and *plugin* are often used interchangeably.

1.2. Creating the *execution* Bundle

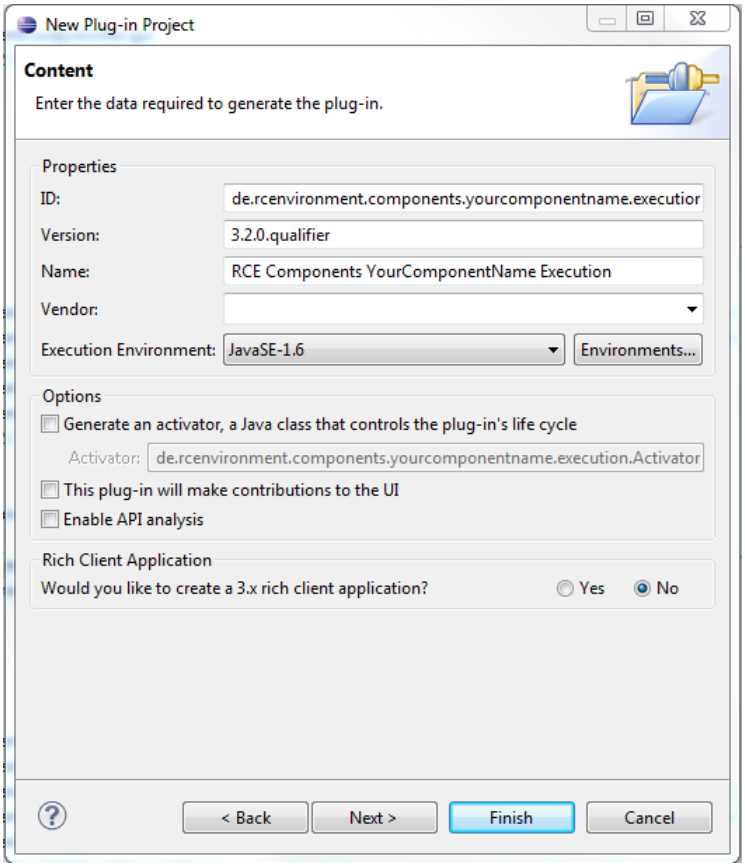
To start developing your first component, generate a new Eclipse Plug-In Project by selecting the corresponding item in the menu bar:



Fill in the dialog properties to configure the plug-in project. The name of the new project should match the RCE naming conventions. That means it should start with "de.rcenvironment.components." and end with ".execution"; so the full name of the execution bundle should be "de.rcenvironment.components.*<yourcomponentid>*.execution", where *<yourcomponentid>* is the ID of your new component. Change the property "Source folder" to "src/main/java" and "Output folder" to "target". Proceed by clicking "Next >".



Now specify the version of your component; you can choose this freely. The name of this plugin should always be "RCE Components *<YourComponentName>* Execution" for the execution bundle, where *<YourComponentName>* again is your component's display name. Press the "Finish" button to complete the general configuration of this plugin.



Eclipse now creates the configured structure of folders, but the plugin is not ready for being used as an RCE component yet.

Create a folder called "resources" in the project you just created. If you have icon files for your component, put them into this folder. Supported formats are PNG, JPG, BMP and GIF. We recommend a resolution of 16×16 and 32×32 pixels. Conventionally these icons are namend "<*yourcomponentname*>16.png" and "<*yourcomponentname*>32.png". Also create an "inputs.json", an "outputs.json" and a "configuration.json" file in the resource folder you just created, where you later define the inputs, outputs and configuration of your component.

As a start, the files can contain an empty JSON object as content. So it suffices to enter the following text in these three files:

```
{ }
```

Note that the files must be present and must not be empty.

Now create a Java class in your source folder by right-clicking on "src/main/java" in the Eclipse Project Explorer. As this will be the central class of your component, give it a name like "<*YourComponentName*>Component.java" (<*YourComponentName*> is your component's display name in camel case, and then add "Component.java"). This Java class must extend `de.rcenvironment.core.component.model.spi.DefaultComponent`.

To implement the functionality of your component override the according methods. The most basic methods to be overriden are:

- processInput: Is called whenever the component receives a new input. In loops this method is called multiple times.
- start: Is called at component start once. Initializations can be placed here.
- dispose: Is called when the component disposes. Clean up methods can be placed here to release resources.

Now create a folder called "`OSGI-INF`" in your project folder, by using the standard Eclipse function (File->New->Folder). This folder will contain all OSGi service definitions for your project. One way to create an OSGi service definition is creating a file "*<yourComponentName>*.xml" ("*<yourComponentName>*" is your own component's display name again) and copying the following source code into it. Replace all occurences of "*<yourComponentName>*" with the display name of your component, and all occurences of "*<yourcomponentname>*" with its id:

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
 factory="de.rcenvironment.rce.component" name="<yourcomponentname>">
 <implementation class="de.rcenvironment.components.<yourcomponentname>.
 execution.<yourComponentName>Component" />
 <service>
  <provide interface="de.rcenvironment.core.component.registration.api.Registerable" />
 </service>
 <property name="rce.component.class" type="String" value="de.rcenvironment.components.
 <yourcomponentname>.execution.<yourComponentName>Component" />
 <property name="rce.component.version" type="String" value="1.0" />
 <property name="rce.component.name" type="String" value="<yourcomponentname>" />
 <property name="rce.component.group" type="String" value="Test" />
 <property name="rce.component.icon-16" type="String" value="/resources/yourcomponentname16.png" />
 <property name="rce.component.icon-32" type="String" value="/resources/yourcomponentname32.png" />
 <property name="rce.component.inputs" type="String" value="/resources/inputs.json" />
 <property name="rce.component.outputs" type="String" value="/resources/outputs.json" />
 <property name="rce.component.configuration" type="String" value="/resources/configuration.json" />
</scr:component>
```

Some of these definitions are optional or refer to elements that do not exist yet. The first lines provide the general XML header, followed by the OSGi root element. The `factory` attribute is the part that links this definition into the RCE framework. The `name` attribute defines your component's name when inspected with OSGi tools and should be the same as the `rce.component.name` property below. The `implementation class` entry defines the main Java class of the component. The `rce.component.group` property sets the GUI group in which your component will appear. (Since the component is still under development, something like "Test" is a good choice for now.) The `rce.component.icon-16` and `rce-component.icon-32` properties are optional and define the icon for your component. The `rce.component.inputs`, `rce.component.outputs` and `rce.component.configuration` attributes define the locations of configuration files that will be described later.

The created plugin project automatically contains a folder named "META-INF". Edit the "MANIFEST.INF" file in this directory by double-clicking it and selecting the "MANIFEST.MF" tab. Add the following lines to it and save:

```
Service-Component: OSGI-INF/*.xml
RCE-Component: true
Export-Package: de.rcenvironment.components.<yourcomponentname>.execution
```

Note that "Export Package: " and the first name have to be in the same line. A second element would be added in the next line with a leading space. Moreover the file must end with a linebreak.

Also note that errors containing "inconsistent hierarchies" can be solved by adding the required referenced packages in the Import-Package property analogous to the Export-Package property.

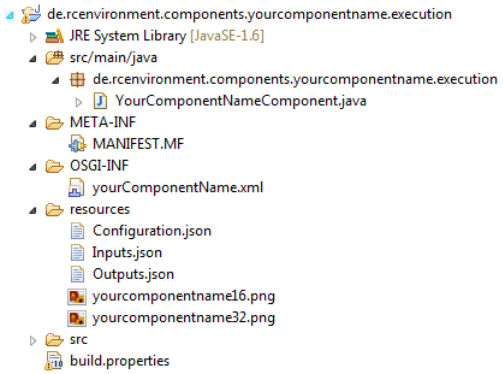
The first line tells the OSGi framework where to look for the component declaration we created before. The second line declares this plug-in as an RCE component. The last line makes the package of your main class visible to other bundles, which is required so RCE can load and initialized the component.

Next, edit the "build.properties" file in the main folder of your plugin project. This file should always look the same for the execution bundles and can be copied from the sample component. It should look like this:

```
source.. = src/main/java
bin.includes = META-INF/,\
OSGI-INF/,\
resources/,\
```

```
.
```

After this step, the plugin configuration is complete. It should look like this in the Eclipse Package Explorer:



When you start RCE from Eclipse and open a workflow file, your component should be shown in the component palette on the right, in the group that you have set in the "OSGI-INF/..." XML file.

1.3. Creating a *common* Bundle

Generate a new Eclipse Plug-In Project by selecting the same option as in the "execution" bundle section above. The project name should follow the RCE naming conventions, similar to the execution bundle but ending with ".common" instead of ".execution".

Add the Java classes to the source folder, within a package with the same name as the bundle's name (or sub-packages of it). For holding shared constants, the usual convention is a Java class named "*<YourComponentName>*ComponentConstants.java". Create this class, and add your first constant for defining the component id and adapt the placeholders in the usual manner:

```
public static final String COMPONENT_ID =
    "de.rcenvironment.components.<yourcomponentname>.
        execution.<YourComponentName>Component";
```

In order to keep track of your constants, it's advisable to give all constants a short comment.

To complete the setup, add the names of the all Java packages you created to the "Export-Package" attribute in the source code of the META-INF/MANIFEST.MF file. Example:

```
Export-Package: de.rcenvironment.components.<yourcomponentname>.common
```

1.4. Creating a *gui* Bundle

To add a graphical user interface for your component, create a "gui" bundle. Start by generating a new blank Eclipse Plug-In Project, as you did for the "execution" and "common" bundle.

The graphical user interface is shown in the properties tab which is usually shown at the bottom of RCE when the component is focused in the workflow editor.

Custom sections can be defined. Therefore create a package called like the project containing it. Inside the package create a class called "*<YourComponentName>*Section.java" which extends `de.rcenvironment.core.gui.workflow.editor.properties.WorkflowNodePropertySectio`
Override the method `createCompositeContent` to fill the GUI.

Each component GUI must provide a *Component Filter*. It is used to determine which sections the GUI for the respective component consists of. Inside the package you just created add a class called "<*YourComponentName*>ComponentFilter.java". extending `de.rcenvironment.core.gui.workflow.editor.properties.ComponentFilter`. To define the component filter, override the following method:

```
@Override
public boolean filterComponentName(String componentId) {
return componentId.startsWith(YourComponentNameComponentConstants.COMPONENT_ID);
}
```

As usual, change *`YourComponentName`* to the name of your own component. This method will only return `true` for the component id used in the "execution" bundle, so it will only be shown for that component.

To provide sections as GUI elements, add a new file called "plugin.xml" to the root of the project folder. It is common to have a section where you can manage inputs and outputs and another section where component specific GUI elements are located. The following code demonstrates this and can be pasted into the plugin.xml you just created. As usual, adapt the component names and IDs accordingly. It is explained below:

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>

   <!-- Property Sections -->
   <extension point="org.eclipse.ui.views.properties.tabbed.propertySections">
      <propertySections contributorId=
      "de.rcenvironment.rce.gui.workflow.editor.WorkflowEditor">

         <propertySection
               tab=
               "workflow.editor.tab.<yourcomponentid>.General"
               class=
               "de.rcenvironment.components.<yourcomponentid>.
               gui.<yourComponentName>Section"
               id=
               "workflow.editor.tab.Properties.Section.General"
               filter=
               "de.rcenvironment.components.<yourcomponentid>.
               gui.<yourComponentName>ComponentFilter"
               enablesFor="1">
         </propertySection>

         <propertySection
               tab=
               "workflow.editor.tab.<yourcomponentid>.InputsOutputs"
               class=
               "de.rcenvironment.core.gui.workflow.editor.
               properties.DefaultEndpointPropertySection"
               id=
               "workflow.editor.tab.Properties.Section.InputsOutputs"
               filter=
               "de.rcenvironment.components.<yourcomponentid>.
               gui.<yourComponentName>ComponentFilter"
               enablesFor=
               "1">
         </propertySection>

      </propertySections>
   </extension>

   <!-- Register Property Sections -->
   <extension point="org.eclipse.ui.views.properties.tabbed.propertyTabs">
      <propertyTabs contributorId=
      "de.rcenvironment.rce.gui.workflow.editor.WorkflowEditor">

         <propertyTab
             label="General"
               category="default"
               id="workflow.editor.tab.<yourcomponentid>.General">
         </propertyTab>

         <propertyTab
               label="Inputs/Outputs"
               category="default"
               id="workflow.editor.tab.<yourcomponentid>.InputsOutputs">
         </propertyTab>
```

```
    </propertyTabs>
  </extension>

</plugin>
```

For each section that you would like to add to the "Properties" view, add the entries like you can see in the propertySection and the propertyTab parts. Make sure that the "filter" attribute contains the correct path of the component filter class created above. The first entry in property section and property tab adds a general custom section to the GUI while the second entry adds a section to manage inputs and outputs.

Remember to keep the file valid by closing the definition blocks with the corresponding end tags:

After assigning the property tabs to their classes, you have to register each tab as a `propertyTab`.

Note that the ID of the propertyTab should be the same as the "tab" in the propertySection above. Customise the section title by editing the `label` attribute.

TODO: Add explanation how to use Messages file.

1.5. Adding your new component to SVN

It is recommended to develop a new component in a separate development branch. Therefore create a new branch. One of the several ways to do so is using TortoiseSVN. Check out the trunk. In the context menu of this folder select "TortoiseSVN > Branch/Tag". Navigate to the destination path which should be located in the branches folder. Call the new branch "dev_*`YourComponentName`*" where *`YourComponentName`* is the display name of your component. Then add the folders of the projects you created above to the structure in your file system. Select them and right click to open the context menu. Then select "TortoiseSVN > Add...". To commit them into the branch, open the context menu again and select "Commit... ".

2. Logging

2.1. General Configuration

(TODO)

2.2. Verbose Logging

Some log messages are disabled by default as they produce large amounts of output, and are only needed in special circumstances (typically, for debugging). This is called "verbose logging", and is controlled by the `DebugSettings` utility class. Typical usage is to initialize a final (static or non-static) field in the logging class with the returned setting, to only incur the configuration checking overhead once:

- non-static field (preferred in most cases, as there is no risk of copying/pasting with the wrong class parameter): `private final boolean verboseLogging = DebugSettings.getVerboseLoggingEnabled(getClass());`
- static field / constant (preferred for classes that are instantiated very frequently): `private static final boolean VERBOSE_LOGGING = DebugSettings.getVerboseLoggingEnabled(TheClassName.class);`

To control the verbose logging, set the `rce.verboseLogging` system property. Example: `rce -p myProfile -vmargs -Drce.verboseLogging=*.NodePropertiesServiceImpl,*Workflow*`. The syntax of the pattern is a comma-separated list of identifiers. A "*" wildcard matches any part of the class name, including the dot ("."). An empty string disables all verbose logging.

> **Note**
>
> As with any other JVM property, this parameter must be placed behind the `-vmargs` delimiter, which separates it from the "direct" RCE command-line arguments (like `--headless` or `-p <profile>`). This rule also applies when adding this parameter to an `rce.ini` file.

If this system property is not set, the `DebugSettings.DEFAULT_VERBOSE_LOGGING_PATTERN` constant's value is used. While developing in Eclipse, it can be useful to enter a verbose logging pattern there, as this affects all local RCE instances at once without editing multiple launch configurations. (As usual, it is your responsibility to make sure these local debug values are not commited into version control.)

Identifiers created before 8.1.0 were fully qualified Java class names; since 8.1.0, identifiers are arbitrary strings. For existing identifiers, these strings were set to the pre-8.1.0 FQNs. Over time, all existing identifiers are planned to be migrated to more intuitive strings.

The following table lists some identifiers that may be useful for debugging:

Table 4.1. Useful Verbose Logging Identifiers

Identifier	Description
NetworkRequests	outgoing and received network requests and responses, and possibly other related operations (e.g. conversions)

Chapter 5. Quality Assurance and Testing

1. Automated GUI Testing

1.1. Getting started

This section describes the required steps to run existing or create new automated GUI tests for RCE using the RCP Testing Tools (RCPTT):

- Download RCPTT from https://eclipse.org/rcptt/download/
- Download RCE from http://rcenvironment.de/pages/download.html
- Start RCPTT
- In the view "Applications" add RCE as Application under test (AUT) via "New... ". Use RCE's main folder as location.
- Configure AUT (Rightclick on entry in "Applications" view -> Configure... -> Advanced...):

1. **Set a profile** different from the default one to make sure your productive RCEs won't interfere with RCPTT's RCE and vice versa. Therefore go to "Arguments" tab and add to Program arguments e.g "-p my_rcptt_profile".
2. **Set dev_config:** Download from dev_config.ini from https://svn.sistec.dlr.de/svn/rce/new/rce/trunk/de.rcenvironment/eclipse/launch/installation_data/ [https://svn.sistec.dlr.de/svn/rce/new/rce/trunk/de.rcenvironment/eclipse/launch/installation_data/dev_config.ini]

   In "Configuration" tab select "Use an existing config.ini file as a template" and navigate to the file you just downloaded.
3. **Set launcher:** In "Arguments" tab add the following VM argument: "-Dde.rcenvironment.launcher=de.rcenvironment.launcher"
4. **Set allocate console for StdIn and StdOut:** To access StdIn and StdOut directly from console view in RCPTT go to "Common" tab and check "Allocate console (necessary for input)

- Check if RCE can be started from RCPTT by double clicking on the entry in the "Applications" view
- Now you can either run existing test cases (A) or create your own test cases (B):

A: Run existing test cases:

- Create an **RCP Testing Tool Project** in the Test Explorer on the lefthand side
- Checkout the following folder and add it to the project: https://svn.sistec.dlr.de/svn/rce/new/rce/trunk/de.rcenvironment/eclipse/ui-testing/RCPTT
- In the folder "Testsuites" navigate to "AllPlatforms" and execute the testsuite (Rightclick -> Run As -> Test Cases)
- Find the Execution View on the bottom left which shows the progress of the testcases

- Do the same for the testsuite that matches your platform.

B: Create your own test cases:

- For the next steps also refer to RCPTT's getting started guide: https://www.eclipse.org/rcptt/documentation/userguide/getstarted/
- Create an **RCP Testing Tool Project** in the Test Explorer on the lefthand side
- Create a **Test Case** within this project
- **Capture script** via "Record" button in the upper right corner and then clicking around in RCE
- **Capture verification** by switching to Assertion Mode in the menu bar of the Control Panel and then selecting some element in RCE
- Click save, stopp and then the Return to RCPTT/Home button
- Create a **Context** within the project (New -> Context). Contexts are proconditions for tests, e.g. to make sure there is a clean setup. For instance, choose a context of type "Workspace" an leave the default settings so it will clear the workspace upon being applied.
- Add the context to the test case by selecting the "Contexts" tab in the test case, click "+" and choose the one you just created.
- **Replay script** by clicking "Replay" in the upper right corner.

2. Integrated Test Script Runner

The integrated "Test Script Runner" (TSR) was created to automate test sequences on one or more RCE standalone installations. These installations are automatically set up using an RCE feature called "Instance Management" (IM), which is still under development, and therefore not fully documented yet. However, the instructions below should be sufficient to configure this feature as needed for the TSR.

2.1. Setup and Configuration

The most common way of using the TSR is simply running it from Eclipse. (It is also intended to be installable into a standalone RCE product; however, this requires certain APIs to be available in the standalone Eclipse, which makes this approach not feasible yet.) Using it from Eclipse is fairly simple, as the project containing it (`de.rcenvironment.supplemental.testscriptrunner`) is part of the standard RCE checkout, and should therefore be present in your development workspace already. You can verify this by starting RCE from Eclipse and executing the command "`run-test`" in the RCE command console. If it prints usage information, the extension is available as it should.

The only manual setup that is required is adding an Instance Management configuration to the RCE instance launched from Eclipse. This configuration defines the root directory where work files and directories will be stored, and the URLs that standalone RCE build will be downloaded from. Locate the profile directory that is being launched and edit its `configuration.json` file. In this file, add this configuration block on the root JSON level, and adjust the settings as necessary:

```
"instanceManagement": {
  "dataRootDirectory": "C:\\ExampleWorkdir\\rce-im",
  "downloadSourceFolderUrlPattern":
      "https://software.dlr.de/updates/rce/8.x/products/
```

```
                                          standard/*/zip/",
  "downloadFilenamePattern": "rce-*-standard-win32.x86_64.zip"
}
```

The asterisks in the URL patterns mark the locations where the specified "build id" (see below) and the automatically determined qualified build number will be inserted.

Note

It is *strongly* recommended to use a short full filesystem path as the `dataRootDirectory`, as RCE installations will be placed inside of it, and long filesystem paths are known to cause problems with these. The exact maximum lenght will be determined, documented, and maybe also automatically checked in the future.

2.2. Usage

The TSR is invoked by a single console command (`run-test`), with an alias for readability (`run-tests`). The general syntax is:

```
run-test[s] <comma-separated list of test ids>|--all <build id>
```

The actual test scripts are located in `*.feature` files in the `/de.rcenvironment.supplemental.testscriptrunner.scripts/resources/scripts` directory. The "test ids" in the command description are the annotation-like tags above the indivual test scenarios, e.g. "@Test01". These can be specified in the command with or without the "@" character. "`--all`" executes all available test scenarios. The reserved "`@Disabled`" and "`@disabled`" tags can be used to mark tests as excluded in the `*.feature` files.

The <build id> is usually the part of the configured download URL marked by an asterisk ("*"); for example, `snapshots/trunk` or `releases/8.1.1`. This id is used within the test scripts as part of certain test steps. For example, the test step "Given instances ... using the default build" sets up one or more test instances to run with the build specified in the command, which allows test scripts to be executed on different builds instead of hardcoding the release or snapshot version to test.

Example: `run-test Test04 snapshots/trunk`

To support running test scripts on locally generated builds, a special form of build id is supported as well: By prefixing it with "`local:`", a local path to a build can be specified. This build is then executed from that location without copying it. Note that the given path must point to an unpacked RCE build, unlike the standard download URL, which points to zipped release packages.

Chapter 6. Licensing and Copyright

1. Copyright Statements

1.1. Current Year Definition

In copyright statements, the current year is defined. Each year, it must be updated at following places (files):

- Header in Java source files
- feature.xml files
- About dialog (de.rcenvironment.core.start/about.mappings)
- Splash images ([..].gui.branding.[..]/splash.bmp)