Eclipse ESCET™ documentation (Incubation)

Version 0.1.0.20210318-143659

Table of Contents

The Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™) project is an Eclipse Foundation open-source project that provides a toolkit for the development of supervisory controllers in the Model-Based Systems Engineering (MBSE) paradigm. The toolkit has a strong focus on model-based design, supervisory controller synthesis, and industrial applicability, for example to cyber-physical systems. The toolkit supports the entire development process of (supervisory) controllers, from modeling, supervisory controller synthesis, simulation-based validation and visualization, and formal verification, to real-time testing and implementation.

The Eclipse ESCET project and all its tools are currently in the Incubation Phase.

The Eclipse ESCET toolkit features the following languages and associated tools:

- Chi
- CIF
- SeText
- ToolDef

This manual includes information related to the Eclipse ESCET toolkit as a whole, and applies to those tools as well. The following information is available for end users of the toolkit:

- Introduction to Eclipse ESCET project
- Download and install Eclipse ESCET toolkit
- Using Eclipse ESCET tools
- Resolving performance and memory problems
- Eclipse ESCET release notes
- Contact information
- Legal

The following information is available for developers of the toolkit:

- Eclipse ESCET development
- Application framework

1. Introduction to Eclipse ESCET project

High-tech companies increasingly adopt the Model-Based Systems Engineering (MBSE) paradigm. The use of (formal) models for controller design allows validation and verification of controllers long before they are implemented and integrated into the system. Early validation and verification have been shown to lead to less defects and reduced costs.

The Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET™) project is an Eclipse Foundation open-source project that provides a toolkit for the development of supervisory controllers in the MBSE paradigm. The toolkit has a strong focus on model-based design, supervisory controller synthesis, and industrial applicability, for example to cyber-physical systems. The toolkit supports the entire development process of (supervisory) controllers, from modeling, supervisory controller synthesis, simulation-based validation and visualization, and formal verification, to real-time testing and implementation.

Supervisory controller synthesis is a key feature of the toolkit. It involves the automatic generation of supervisory controllers from a specification of the uncontrolled system and the (safety) requirements that the controller needs to enforce. This shifts controller design from 'how should the implementation work' to 'what should the controller do'. Implementation of the controller is achieved through (implementation language) code generation, reducing the number of errors introduced at this stage.

The Eclipse ESCET toolkit features the following languages and associated tools:

- Chi
- CIF
- SeText
- ToolDef

1.1. Chi

The Chi language is a modeling language for describing and analyzing the performance of discrete event systems by means of simulation. The language design is based on decades of successful analyses of various (industrial) systems, aiming to be powerful for advanced users, and easy to use for non-experts.

The language uses a process-based view. A system (and its control) is modeled as a collection of parallel running processes, communicating with each other using point-to-point communication channels. Processes do not share data with other processes, and channels are synchronous (sending and receiving is always done together at the same time), making reasoning about process behavior easier. Processes and channels are dynamic, new processes can be created as needed, and communication channels can be created or rerouted, making for a powerful specification language.

The language is designed to be formal and easily extensible. Models are written as an imperative program, with an intuitive syntax, making it easy to read and write models. A small generic set of statements can be used to describe algorithms, including assignments, *if*, *while* and *for* statements. This set is relatively easy to explain to non-experts, allowing them to understand the model, and participate in the discussions.

The data of the system can be modeled using both basic data types, such as *booleans* and *integer* and *real* numbers, as well as high level structured collections of data like *lists*, *sets* and *dictionaries*. If desired, processes and channels can also be part of that data. Furthermore, timers and (quasi-)random number generation distributions are available for modeling timed and stochastic systems. Features to easily specify repeated experiments, e.g. for stochastic simulation, or simulation for various inputs obtained from files, exist to support large simulation experiments.

While the language is generic, the main application area is modeling of the operation of (manufacturing) systems. During the design process, engineers can make use of analytical models, to get answers about the operation of the system. Simulation studies can provide insights into e.g. the throughput of the system, the effect of set-up time in a machine, or how the batch size of an order will influence the flow time of the product-items.

The Chi toolset allows verification of properties of the actual system by means of simulation, e.g. to optimize the supervisory (logic) control of the system. The Chi language has features that allow for easy specification. Chi aims to make the process of verifying properties for large systems effortless.

Tutorials and manuals demonstrate the use of the language for effective modeling of system processes. More detailed modeling of the processes and e.g performance indicators, or custom tailoring them to the real situation, has no inherent limits.

See the separate Chi documentation for more information.

1.2. CIF

The CIF language is a powerful declarative automata-based modeling language for the specification of discrete event, timed (linear dynamics), hybrid (piecewise continuous dynamics) systems. It can be seen as a rich state machine language with the following main features:

- Modular specification with synchronized events and communication between automata.
- Many data types are available (booleans, integers, reals, tuples, lists, arrays, sets, and dictionaries), combined with a powerful expression language for compact variables updates.
- Text-based specification of the automata, with many features to simplify modeling large non-trivial industrial systems.
- Primitives for supervisory controller synthesis are integrated in the language.

The CIF tooling supports the entire development process of controllers, including among others specification, supervisory controller synthesis, simulation-based validation and visualization, verification, real-time testing, and code generation. Highlights of the CIF tooling include:

- Text-based editor that allows to easily specify and edit models.
- Feature-rich powerful data-based synthesis tool. A transformation to the supervisory controller synthesis tool Supremica is also available.
- A simulator that supports both interactive and automated validation of specifications. Powerful visualization features allow for interactive visualization-based validation.
- Conversion to formal verification tools such as mCRL2 and UPPAAL.
- Implementation language code generation (PLC languages, Java, C, and Simulink) for real-time testing and implementation of the designed controller.

See the separate CIF documentation for more information.

1.3. SeText

SeText is a textual syntax specification language and associated scanner/parser generator. It can be used to specify the syntax of a language, and automatically generate a scanner and LALR(1) parser(s).

SeText is used as scanner/parser technology for the other tools within the Eclipse ESCET project. It is however a generic scanner/parser generator that can also be used for the development of scanners and parsers for other languages.

See the separate SeText documentation for more information.

1.4. ToolDef

ToolDef is a cross-platform and machine-independent scripting language. It supports command line execution, but is also available as plug-in for the Eclipse IDE, providing an integrated development experience.

The ToolDef language features a simple and intuitive syntax to make it easy to write scripts, static typing to catch simple mistakes, a large number of built-in data types and tools, Java integration, and more.

ToolDef libraries with ToolDef compatible tools are available for all tools within the Eclipse ESCET toolkit, allowing cross-platform and machine-independent scripting using Eclipse ESCET tools. ToolDef however is a generic scripting language, and can be used without using any of the other Eclipse ESCET tools. Furthermore, other tools can be made available for use within ToolDef scripts by defining ToolDef libraries for them.

See the separate ToolDef documentation for more information.

2. Download and install Eclipse ESCET toolkit

⚠ The Eclipse ESCET project and all its tools are currently in the Incubation Phase.

You can download Eclipse ESCET from the following locations:

- Eclipse ESCET downloads (for recent versions)
- Eclipse ESCET archived downloads (for older versions)

Downloads for Eclipse ESCET tools are available for the following platforms:

- Windows, x64 (64-bit)
- Linux, x64 (64-bit)
- macOS, x64 (64-bit)

The downloads are archives that can be extracted anywhere on the local system. Each download contains the Eclipse ESCET toolkit, with the tools available in two ways:

- As IDE based on the Eclipse IDE, providing the full experience. Includes full GUI integration and e.g. text editors with syntax highlighting and error checking.
- As command line execution scripts (`bin` directory of the archives), allowing execution in a command line terminal or console, particularly useful for execution of the tools on headless clusters. The command line scripts are only available for Windows and Linux, not for macOS.

The following dependencies need to be available on the system:

- Java Development Kit (JDK):
  - Version 8 update 31 (8u31) or newer is required. Versions 9 and higher are currently not yet supported.
  - An x64 (64-bit) version of Java is required, which also requires a 64-bit operating system.
  - Currently the Eclipse ESCET tools are only tested using Oracle JDKs. Your experience with other JDK vendors may vary.
  - A Java Runtime Environment (JRE) is not sufficient. A JDK is required.

The tools are also available as plug-ins for the Eclipse IDE, by means of an Eclipse P2 update site.

Furthermore, the following documentation is available for download:

- Eclipse ESCET documentation (this documentation)
- Chi documentation

- CIF documentation
- SeText documentation
- ToolDef documentation

The following additional information is available:

- Installation of Eclipse ESCET tools
- Starting Eclipse ESCET IDE for the first time
- Updating Eclipse ESCET tools
- Removing Eclipse ESCET tools
- Finding the tool’s version number

2.1. Installation of Eclipse ESCET tools

Before installing Eclipse ESCET tools, first install a Java Development Kit (JDK), taking into account the following restrictions:

- Version 8 update 31 (8u31) or newer is required. Versions 9 and higher are currently not yet supported.
- An x64 (64-bit) version of Java is required, which also requires a 64-bit operating system.
- Currently the Eclipse ESCET tools are only tested using Oracle JDKs. Your experience with other JDK vendors may vary.
- A Java Runtime Environment (JRE) is not sufficient. A JDK is required.

You can download a JDK from Oracle’s Java SE Development Kit 8 Downloads page. For more information, see:

- Oracle’s JDK Installation for Linux Platforms page.
- Oracle’s JDK Installation for Microsoft Windows page.
- Oracle’s JDK 8 Installation for OS X page.

Ensure that this JDK is in your `PATH`. See e.g. How do I set or change the PATH system variable? for how to achieve this.

Next, obtain the Eclipse ESCET tools:

- Download the Eclipse ESCET tools.
- Extract the downloaded archive somewhere on your hard disk, to a new empty directory where you have read and write access.

You can now start the Eclipse ESCET IDE:

- For Windows, execute `eclipse.exe` from the directory that contains the extracted files.
- For Linux, execute `eclipse` from the directory that contains the extracted files.

- For macOS, installation is more complex and may also depend on the version of macOS. The following instructions have been tested on Big Sur. The downloaded archive contains the Eclipse app in the form of a directory named `Eclipse.app`, which contains all the files of the release. The operating system recognizes this directory as a macOS application and displays it as *Eclipse* with a custom icon. Move/drag the *Eclipse* application icon to the macOS *Applications* folder, to make the Eclipse icon appear among the other macOS applications. Execute the Eclipse application by double clicking it.

  On macOS Big Sur, you will get an error message saying either *"Eclipse.app" cannot be opened because it was not downloaded from the App store* or *"Eclipse.app" cannot be opened because Apple cannot check it for malicious software*. Dismiss the popup by clicking **[OK]**. Go to **Apple menu › System Preferences**, click *Security & Privacy*, then click tab *General*. Under *Allow apps downloaded from:* you should see two tick boxes: one for *App Store* and one for *App Store and identified developers*. Below that, you should see an additional line: *"Eclipse.app"* was blocked from use because it is not from an identified developer, and after that, a button **[Open Anyway]**. Clicking that button will allow the Eclipse app to be executed.

  If executing the Eclipse app now gives an *Alert* message saying *Failed to create the Java virtual machine*, you need to add a setting in Eclipse that points to the Java JDK as follows:

  In the Finder, right click the Eclipse app icon and execute *Show Package Contents*. In the folder `Contents`, edit the file `Info.plist`. You can do this by right clicking the file and execute **Open With › TextEdit**. Near the end of this file, look for the following lines:

  ```
  <key>Eclipse</key>
          <array>
                  <!-- to use a specific ...
                          ...
                          ...
                          ...
                  -->
  ```

  Immediately after these lines, insert the following line:

  ```
  <string>-
  vm</string><string>/Library/Java/JavaVirtualMachines/jdk1.8.0_281.jdk/Contents/Home
  /bin/java</string>
  ```

  where you should change the version number `0_281` to the version number of the JDK that you have installed on your Mac. You can find this version number in the name of the folder that is present in `/Library/Java/JavaVirtualMachines/`. You can go to this folder via the Finder menu **Go › Go to Folder...**. After saving the file, try to execute the Eclipse app.

  In case you now get an *Alert* error message saying *The JVM shared library "/Library/Java/JavaVirtualMachines/jdk..." does not contain the JNI_CreateJavaVM symbol*, click **[OK]** and restart your Mac. After restarting, the Eclipse app should work.

For more information, see the section on starting the Eclipse ESCET IDE for the first time.

You can now also use the command line scripts located in the `bin` directory. These are only available for Windows and Linux, not for macOS. To see which tools are available, simply look in this directory. Each of these tools can be started with the `-h` or `--help` option to get further information. You may want to add the `bin` directory to your `PATH` environment variable. See e.g. How do I set or change the PATH system variable? for how to achieve this.

If the Eclipse ESCET IDE or one of the command line scripts cannot be started, the ESCET tools may not be able to correctly detect the JDK you installed. In such a case, edit the `eclipse.ini` file from the directory that contains the extracted files and add in the following lines at the beginning of the file:

```
-vm
C:/Program Files/Java/jdk1.8.0_211/bin
```

Note that `-vm` and the path to the JDK need to be on separate lines. Obviously, change the JDK path to the actual path for your system. Make sure to save the file and retry starting the Eclipse ESCET IDE or one of the command line scripts.

Adding the `-vm` option to the `eclipse.ini` file on macOS is of no use.

2.2. Starting Eclipse ESCET IDE for the first time

When you start the Eclipse ESCET IDE for the first time, it will ask you to choose a workspace directory. A workspace directory is where all your settings will be stored. It is also the default directory for new projects, in which you will store your files.

Choose a workspace directory and continue. Make sure that you have read and write access to the directory you choose. If you wish, the Eclipse ESCET IDE can remember your workspace directory. Note that if the workspace directory you choose does not yet exist, Eclipse will create it for you.

The first time Eclipse launches in a fresh workspace, you will get a *Welcome* screen. You can close this tab by clicking the 'X' at the right of the tab, or by clicking on the 'workbench' icon (the right most icon on the welcome page).

2.3. Updating Eclipse ESCET tools

There are two ways to obtain a newer version of the Eclipse ESCET tools:

- New download of the Eclipse ESCET tools (including command line scripts), to be used side-by-side the older version.
- In-place update of the Eclipse ESCET tools.

2.3.1. Side by side new installation

To download and install a new version of Eclipse ESCET tools (including the command line scripts) side-by-side an older version, follow these steps:

- Follow the normal installation instructions, extracting the new version to a different directory than the old version.
- You can copy the data (projects, files, and settings) of an earlier installation to the new installation. While neither version of the Eclipse ESCET IDE is running, simply remove the workspace directory of the new installation, and copy the workspace directory of the earlier installation to the new installation.

Using these instructions, it is possible to use multiple installations side by side, at the same time, regardless of whether the installations are the same release or different releases. Simply extract them to different directories and launch them as you would normally do.

2.3.2. In-place update

To perform an in-place update of the Eclipse ESCET tools:

- Select **Help › Check for Updates** within the Eclipse ESCET IDE. Follow the on-screen instructions to perform an in-place update.
- This will *not* update the command line scripts.
- If the tool indicates that no updates are available, while you are sure that an updated version has been released, or if some other problem occurs, please restart the tool and try again. If still an in-place update fails, try to install a new version side-by-side the older version.
- If you get a dialog stating there is a problem, click the **[Details]** button for further details. If may mention `Address family not supported by protocol family: connect`. If so, close Eclipse, and add the following line to the `eclipse.ini` file:

  ```
  -Djava.net.preferIPv4Stack=true
  ```

  Add the line at the end of the file, on a line by itself. By default, `eclipse.ini` is located in the Eclipse ESCET tools installation directory, except for macOS, where instead it is in the `Eclipse.app/Contents/MacOS` directory inside the Eclipse ESCET tools installation directory. Restart the Eclipse ESCET IDE and try again.

2.4. Removing Eclipse ESCET tools

Before removing a version of the Eclipse ESCET tools, you may want to preserve (back up) its workspace directory, to keep your data (settings, projects, files). Then, to remove a release, simply remove the directory that contains files that you extracted from the downloaded archive during installation.

Additionally, you may want to remove Java if you no longer need it.

2.5. Finding the tool’s version number

From the Eclipse ESCET IDE, there are multiple ways to find out which version of the toolkit or specific tools you have currently installed:

- Via the Eclipse *About Eclipse ESCET* dialog.

  The *About Eclipse ESCET* dialog can be opened via the **Help › About Eclipse ESCET** menu of the Eclipse ESCET IDE. The dialog has shows the version of the Eclipse ESCET toolkit. An **[Installation Details]** button is available to open the *Eclipse Installation Details* dialog. In this dialog, the *Installed Software* tab shows all the installed software, including their versions under the *Version* column.

- Via the option dialogs of the various tools.

  Most of the Eclipse ESCET tools can be started in a way that shows the option dialog for that tool. All option dialogs for our tools have a *Help* category. By clicking on that category, the help text for that tool is shown. The help text includes the version of the tool that you are using.

- Via the command line option, in a ToolDef script.

  If you start an application using a ToolDef script, you can specify command line arguments in the script as well. Start a tool with the `-h` or `--help` option to see the command line help text, which includes the version.

For command line scripts, the following approach is recommended:

- Start a tool with the `-h` or `--help` option to see the command line help text, which includes the version.

3. Using Eclipse ESCET tools

After you have installed the Eclipse ESCET tools, you can start using them. The following information is available to get you started on using the Eclipse ESCET tools in general, and applies to the various tools in the toolkit:

- Eclipse terminology
- Working with projects, directories, and files
- Editing files and executing commands
- Eclipse ESCET perspective
- Applications view

Consult the documentation of the individual tools for specific information regarding their use.

3.1. Eclipse terminology

The Eclipse ESCET IDE is based on the *Eclipse IDE*, a cross platform Integrated Development Environment (IDE). There is quite a bit of terminology that is used within the IDE. If you are not familiar with Eclipse terminology, it may be difficult to use the Eclipse ESCET IDE. Here we'll explain some basic Eclipse terminology:

- Eclipse workspace
- Eclipse views
- Eclipse projects

3.1.1. Eclipse workspace

Eclipse stores all its settings in a so-called *workspace*. The workspace is simply a directory on your computer. You can choose any directory you like to serve as a workspace, as long as you have write access to that directory. It is usually best to choose an empty directory or a directory that does not yet exist as your workspace.

Typically, and by default, a directory named `workspace` inside your Eclipse ESCET installation directory is used. Eclipse will ask you to choose a workspace directory when you start the Eclipse ESCET IDE for the first time.

The actual settings are stored in a sub-directory of the workspace directory, called `.metadata`. You should avoid manually manipulating this directory. Note that because the name of the directory starts with a dot (`.`), depending on your operating system, file browser, and settings, the directory may be hidden.

You can easily change your workspace directory from within Eclipse, by selecting **File › Switch workspace**. Select one of the workspaces from the list of last used workspaces, or select **Other…** to freely select any directory on your system to use as a workspace directory.

It is possible to run multiple instances of Eclipse at the same time, but each instance must use its own workspace.

The workspace is also the default directory for new projects. However, projects don’t have to be physically located inside your workspace directory. They can be stored in any directory on your system. Whenever you create a project and store it outside of your workspace, or whenever you import an existing project from outside your workspace, it is *linked* to the workspace, but remains physically stored in a directory outside of the workspace.

Having projects stored outside of the workspace has some benefits. The most important benefit is that you can remove the workspace directory, without losing your files.

3.1.2. Eclipse views

Eclipse is an Integrated Development Environment (IDE) with a lot of functionality. Most of the functionality is available through *views*. A view is a part of the Eclipse graphical user interface. Views can be thought of as 'sub-windows'. When you start Eclipse you are likely to see the *Project Explorer* or *Package Explorer* view on the left, and the *Problems* view at the bottom.

Opening a view

To open/show a view, select **Window › Show view** and then choose the view that you wish to open.

If the particular view that you wish to open is not in that menu, choose **Other…** instead. A new dialog opens, in which you can find all available views. The views are organized into categories. Expand a category, select the desired view, and click **[OK]**.

Alternatively, in the **Show view** dialog, enter the name of the view (or the first part of it) in the filter box at the top of the dialog, and observe how views that don’t match the filter are no longer displayed. This makes it easier to find the desired view.

3.1.3. Eclipse projects

Eclipse, being an Integrated Development Environment (IDE), does not only allow you to edit a single file, and simulate it, but also allows you to *manage* your files.

Eclipse works with so-called *projects*. A project is a collection of files and directories. A project may be located anywhere on your system, even though by default project are created in your workspace directory.

A project is essentially a directory on your computer, with a special file named `.project`. This special file stores the information about your project, such as the name of the project. It is recommended to keep the name of the project as Eclipse sees it, the same as the name of the directory in which the project is stored on your hard disk.

You can see the projects that Eclipse knows about in the *Project Explorer* or *Package Explorer* <use-terminology-view,view>>.

For more information on projects, see the Working with projects, directories, and files section.

3.2. Working with projects, directories, and files

Eclipse uses projects to manage files and directories (also called *folders*). This section contains information on the following topics:

- Creating a new project
- Importing an existing project
- Creating a new directory
- Creating a new file
- Renaming a file, directory, or project
- Locating files, directories, and projects
- Refreshing a file, directory, or project

3.2.1. Creating a new project

To create a new project, ensure you have either the *Project Explorer* or *Package Explorer* view visible.

Select **File › New › Project…** to open the *New Project* window. Alternatively, right click somewhere in the *Project Explorer* or *Package Explorer* view, and select **New › Project…**, to open the same window.

In the *New Project* window, from the *General* category select *Project*, and click the **[Next >]** button.

In the next window, enter a project name, for example `models`.

By default, a project is created inside your workspace directory. If you want it at a different place (so you can remove the workspace without losing your projects), uncheck the *Use default location* option, and enter a *Location*. Make sure the location does not yet exist, for example by ending with the project name. It is recommended to always create a project in a directory that has the same name as the project.

Click the **[Finish]** button to create the project. Observe how it appears in the *Project Explorer* or *Package Explorer* view.

3.2.2. Importing an existing project

If you previously created a project, but it is no longer visible in your *Project Explorer* or *Package Explorer* view, you can import it. Imported projects are *linked* to your workspace, but remain in their original location on your hard disk (by default, they are *not* moved or copied to your workspace directory).

Importing existing projects is particularly useful if you removed you workspace directory and started with a fresh one, e.g. for a new installation.

To import one or more existing projects, select **File › Import…** to open the *Import* window. Alternatively, right click somewhere in the *Project Explorer* or *Package Explorer* view, and select **Import…** to open the same window.

In the *Import* window, from the *General* category, select *Existing Projects into Workspace*, and click **[Next >]**.

In the next window, in *Select root directory*, point it to the directory that contains the project you wish to import. The available projects in the given root directory, and all its sub-directories (recursively), are listed in the *Projects* list. Select the project(s) you want to import, by checking the relevant check boxes, and click **[Finish]**. Observe how the project(s) appear(s) in the *Project Explorer* or *Package Explorer* view.

3.2.3. Removing a project from Eclipse

To remove a project from Eclipse, ensure you have either the *Project Explorer* or *Package Explorer* view visible.

Right click the project in the *Project Explorer* or *Package Explorer* view and choose **Delete**. Alternatively, select it and press the `Delete` key on your keyboard. The *Delete Resources* dialog appears. To only remove the project from Eclipse, and keep the files on your hard disk, disable the *Delete project contents on disk (cannot be undone)* option and click the **[OK]** button. Alternatively, to remove the project from Eclipse and also remove all the files in the project from your hard disk, enable the *Delete project contents on disk (cannot be undone)* option and click the **[OK]** button.

Wait for Eclipse to complete the removal operation.

3.2.4. Creating a new directory

You can structure large collections of files, by putting them into different directories (also called *folders*). Directories can only be created in projects, or in other directories.

Select the project or directory in which you want to create a new directory (by left clicking its name in the *Project Explorer* or *Package Explorer* view, and create a new directory by selecting **File › New**

› **Folder**. Alternatively, right click on the project or directory instead, and from the popup menu select **New** › **Folder**.

In the *New Folder* window, enter the *Folder name*, and click **[Finish]**.

3.2.5. Creating a new file

Select the project or directory in which you want to create a new file (by left clicking its name in the *Project Explorer* or *Package Explorer* view), and create a new file by selecting **File** › **New** › **File**. Alternatively, right click on the project or directory instead, and from the popup menu select **New** › **File**.

In the *New File* window, enter the *File name*, and click **[Finish]**. Make sure to give the file the correct file extension. E.g. CIF files should end with `.cif`.

An editor for the new file opens, and you can start editing it.

3.2.6. Renaming a file, directory, or project

To rename a file, directory, or project, select it by left clicking its name in the *Project Explorer* or *Package Explorer* view, and then select **File** › **Rename…**. Alternatively, right click on the file, directory, or project instead, and from the popup menu select **Refactor** › **Rename…**. A second alternative is to select the file, directory, or project, and then press the `F2` key.

In the *Rename Resource* window, enter the *New name*, and click **[OK]**.

3.2.7. Locating files, directories, and projects

Often, it is convenient to be able to manage files not from inside Eclipse, but from outside Eclipse, for instance in a file explorer provided by your operating system. The *Properties* view can be used to find out where the files, directories, and projects that are in Eclipse, are located on your hard disk. With the *Properties* view visible, select a file, directory, or project in the *Project Explorer* or *Package Explorer* view. In the *Property* column of the *Properties* view, look for *location*. The corresponding *Value* indicates where the file, directory, or project is located on your hard disk. Note that you can right click the location and choose *Copy* to copy the location to the clipboard.

As an alternative to the *Properties* view, you can also use the *Properties* window. Right click a file, directory, or project in the *Project Explorer* or *Package Explorer* view and choose **Properties**. In the window that shows, select *Resource* on the left, if it is not already selected. Then, on the right, look for the *Location*.

To directly open the directory that contains a file, directory, or project in your system's file explorer, right click the file, directory, or project in the *Project Explorer* or *Package Explorer* view and choose **Show In › System Explorer**.

Don’t forget to refresh your projects in Eclipse after manipulating them outside Eclipse.

3.2.8. Refreshing a file, directory, or project

Whenever changes are made to files or directories from outside Eclipse, and those files or directories are also in one of the projects inside Eclipse, the changes are *not* always automatically reflected in the *Project Explorer* or *Package Explorer* view. To ensure that the current state of the files and directories are properly reflected in Eclipse, a refresh is required. To refresh a file, directory, or project, right click it, and choose **Refresh**. Any files and directories that no longer exist will disappear from Eclipse. Any new files and directories created outside Eclipse will appear in Eclipse as well.

3.2.9. Checking the size of a file

When working with files, you may occasionally encounter large files. Opening large files in Eclipse can cause serious performance problems. You can use the *Properties* view to check the size of a file. With the *Properties* view visible, select a file in the *Project Explorer* or *Package Explorer* view. In the *Property* column of the *Properties* view, look for *size*. The corresponding *Value* indicates the size of the file.

As an alternative to the *Properties* view, you can also use the *Properties* window. Right click a file in the *Project Explorer* or *Package Explorer* view and choose **Properties**. In the window that shows, select *Resource* on the left, if it is not already selected. Then, on the right, look for the *Size*.

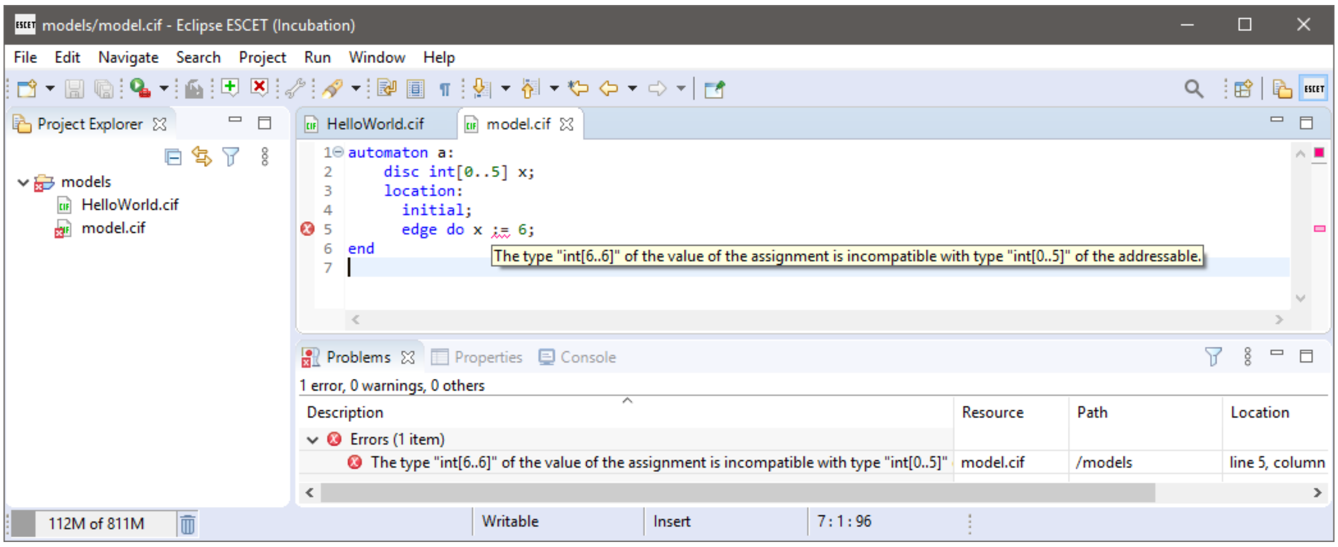
3.3. Editing files and executing commands

This section contains information on editing text files and executing commands (such as starting a simulator).

3.3.1. Editing a text file

To edit a file, double click it in the *Project explorer* or *Package explorer* view.

Whenever you open a text file, change it, or the editor receives the focus, the file is checked for errors. If there are errors (or warnings), they are displayed in the editor as shown below.



```
automaton a:
    disc int[0..5] x;
    location:
      initial;
      edge do x := 6;
end
```

The type "int[6..6]" of the value of the assignment is incompatible with type "int[0..5]" of the addressable.

1 error, 0 warnings, 0 others

Description	Resource	Path	Location
Errors (1 item)			
The type "int[6..6]" of the value of the assignment is incompatible with type "int[0..5]"	model.cif	/models	line 5, column

Note that you can hover over an error in the source code itself to find out what the problem is. Alternatively, you can hover over the error marker in the margin of the editor, or look at the *Problems* view. Also, if a file has an error or a warning, an overlay icon is shown in the *Project Explorer* and *Package Explorer* views, for that file, the directories that contain it, and the project that contains it.

3.3.2. Executing commands

If you have a file without errors, you can execute certain commands on it. The various Eclipse ESCET tools add *commands* to Eclipse. For instance, CIF models can be simulated using a simulation command.

To execute a command on a file, right click the file in the *Project Explorer* or *Package Explorer* view, and select the command. Alternatively, if you have the file open in an editor, right click the editor, and select the command.

The commands that are available are determined by the file extension of the file. That is, only the commands applicable for a certain file are shown.

Besides simulation, other commands may be available, depending on the modeling language and tools you use. Consult the specific documentation for each tool for more details.

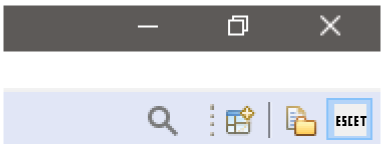
3.4. Eclipse ESCET perspective

The layout of the different parts of the Eclipse ESCET IDE, including the position and size of the various views, can be configured per perspective. Different perspectives allow you to use different layouts for different tasks.

The Eclipse ESCET IDE ships with some built-in perspectives. The default *ESCET* perspective is ideally suited for the use of the Eclipse ESCET software.

You can manually open the *ESCET* perspective (or any other perspective), by selecting **Window › Perspective › Open Perspective › Other...**. Then, in the *Open Perspective* dialog, select the *ESCET* perspective from the list, and click the **[OK]** button.

By default, the Eclipse ESCET IDE shows the opened perspectives at the top right corner of the IDE. Each perspective is a button that can be used to active it. If the button appears in a pushed state, that perspective is active. The following image shows an Eclipse ESCET IDE with two open perspectives: the *Resource* perspective and the *ESCET* perspective. The *ESCET* perspective is the currently enabled perspective.



By default, the *ESCET* perspective shows the following views:

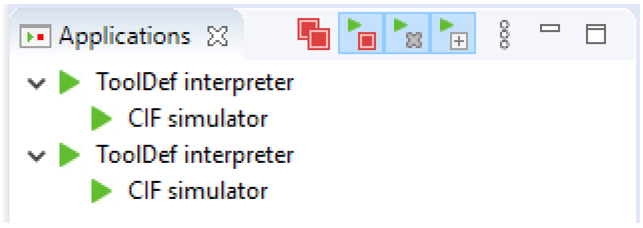
- Project explorer (top left)
- Applications (bottom left)
- Problems (bottom, grouped)
- Properties (bottom, grouped)
- Console (bottom, grouped)

You can manually open additional views, close some views, move views around, change the size of the different views, etc. If at any time you wish to go back to the original layout, you can reset the perspective, by selecting **Window › Perspective › Reset Perspective...**. Click the **[OK]** button to confirm.

Whenever a new release changes the default layout of a perspective, for instance by adding a new default view, you can reset the perspective to get the new view, or you can open that view it manually.

3.5. Applications view

The *Applications* view can be used to manage the applications of the Eclipse ESCET software running within the Eclipse ESCET IDE.



3.5.1. Opening the view

The *Applications* view can be opened by selecting **Window › Show view › Applications**, assuming the ESCET perspective is enabled.

3.5.2. Application status

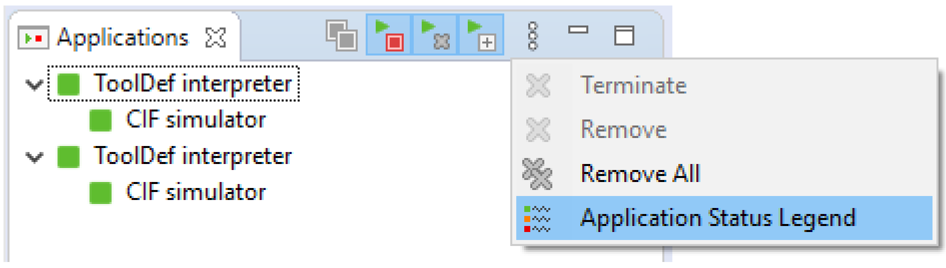
The view shows a list of running applications. If an application starts another application, it is shown as a child, forming a tree structure. Each item of the tree shows a single application. In the example screenshot above, there are four running applications. The first application is a *ToolDef interpreter*, which is running a second application, a *CIF simulator*. The third application is also a *ToolDef interpreter*, which is running a fourth application, also a *CIF simulator*.

To the left of each item, the status of the application is indicated. The following statuses are available:

- ▶ Staring
- ▶ Running
- ▶ Terminating
- 🟩 Terminated
- ✓ Finished
- ✗ Failed
- ! Crashed

When an application is started, it get a *Starting* state. Once the options have been processed from the command line and the option dialog (if requested), the actual application code is executed, and the status changes to *Running*. If the application is not terminated at the request of the user, the application will be done with its execution after some time. If execution finished without problems, it then reaches status *Finished*. If execution failed for some reason, it reaches status *Failed* instead. If the execution crashed due to internal problems, the application reaches status *Crashed*. If however the user terminates the application, the status is first changed to *Terminating*. Once the application has successfully been terminated, the status changes to *Terminated*.

You can also view these statuses and their corresponding icons from within the IDE, by opening the *Application Status Legend* dialog, which is available via the view's drop-down button popup menu:



3.5.3. Termination

The *Applications* view supports terminating running applications. There are several ways to terminate an application using the *Applications* view.

When terminating an application (with a *Starting* or *Running* state), the application will get a *Terminating* state. Applications occasionally poll for termination requests. Therefore, it may take a while for the application to actually process the termination request. Once it has been processed, the application will have terminated, and the status changes to *Terminated* as well.

Applications can’t process a termination request while a modal dialog (such as the option dialog) is open, or when input is requested from the console. They will terminate as soon as possible, after the dialog is closed, or the user has provided input via the console.

The following buttons related to termination are available in the *Applications* view’s toolbar:

- Auto Terminate ()

  Most users will only want to have a single application running at a time, including application started by that application, etc. To facilitate this, the view provides an *Auto Terminate* feature. This feature can be enabled and disabled from the view’s toolbar. It is enabled by default. Your choice whether to enable or disable this option will be remembered, even after Eclipse is restarted. The feature only has an effect if the view is shown in at least one of the opened perspectives.

  If enabled, and a new application (not started by another already running application) is started, all other already running applications are automatically terminated. The new application starts its execution immediately. The already running applications will start to process the termination request, and will terminate as soon as possible.

- Terminate All ()

  The view’s toolbar contains a *Terminate All* button that can be used to terminate all running applications. The button is only enabled when at least one of the listed applications can be terminated. When clicked, all running applications are given a termination request, and will terminate as soon as possible.

- Terminate ()

  The view’s drop-down button popup menu contains a *Terminate* item that can be used to terminate all selected applications that are running. The item is only enabled when at least one of the listed applications that is selected, can be terminated. When clicked, all selected applications that can be terminated, are given a termination request, and will terminate as soon as possible.

Even when a child application is selected, all applications in the entire tree, starting from the root, will be given a termination request, if not already terminated or having finished their execution. In other words, only an entire tree of related applications can be terminated.

3.5.4. Removal

The *Applications* view can get a bit crowded, if already terminated applications are not removed from the list. To keep only relevant applications, the *Applications* view supports removing items from the list.

Only applications which have finished execution can be removed. That is, the root of the tree of applications needs to have a *Terminated*, *Finished*, *Failed*, or *Crashed* status, for the tree to be allowed to be removed. This prevents removing applications that are still running, which would make it impossible to terminate them, or observe their status.

The following buttons related to removal are available in the *Applications* view's toolbar:

- Auto Remove ()

  Most users will only want to only have applications listed in the view, that are either still running, or have just finished execution. To facilitate this, the view provides an *Auto Remove* feature. This feature can be enabled and disabled from the view's toolbar. It is enabled by default. Your choice whether to enable or disable this option will be remembered, even after Eclipse is restarted. The feature only has an effect if the view is shown in at least one of the opened perspectives.

  If enabled, and a new application (not started by another already running application) is started, all already terminated applications, will be removed from the list. If combined with the *Auto Terminate* feature, all other applications that can not be removed immediately because they are still running, will be removed as soon as possible, after they have been terminated.

- Remove All ()

  The view's drop-down button popup menu contains a *Remove All* item that can be used to remove all listed applications that may be removed. The item is only enabled when at least one of the listed applications can be removed. When clicked, all listed applications that can be removed, are immediately removed from the list.

- Remove ()

  The view's drop-down button popup menu contains a *Remove* item that can be used to remove all selected applications that can be removed. The item is only enabled when at least one of the listed applications that is selected, can be removed. When clicked, all selected applications that can be removed, are immediately removed from the list.

Even when a child application is selected, all applications in the entire tree, starting from the root, will be removed. In other words, only an entire tree of related applications can be removed, and only if all applications in that tree have finished execution.

3.5.5. Expansion

When one application starts another application, they are listed in a tree, with the parent containing the child. When running a single application, it may be of interest to see which child applications are being executed by the parent application. However, when executing multiple applications, this may quickly crowd the view. It may then be better to keep all root items collapsed, only showing the status of the root applications. This provides an overview over those multiple applications.

The following buttons related to expansion are available in the *Applications* view's toolbar:

- Auto Expand ()

  Most users will have the *Auto Terminate* and *Auto Remove* features enabled, and will thus only have a single application listed. They will want to automatically expand a parent application, to show its children. To facilitate this, the view provides an *Auto Expand* feature. This feature can be enabled and disabled from the view's toolbar. It is enabled by default. Your choice whether to enable or disable this option will be remembered, even after Eclipse is restarted. The feature only has an effect if the view is shown in at least one of the opened perspectives.

  If enabled, and a parent application starts a child application, the item for the parent application in the view, is automatically expanded to show its children. If disabled, no items will be automatically expanded.

4. Resolving performance and memory problems

During the use of the Eclipse ESCET toolkit, you may encounter performance and/or memory problems. This includes slow execution, performance degradation over time, out-of-memory errors, etc. One way to solve such problems, is to use a computer that is faster and/or has more memory. If that is not an option, or if that doesn't help, the following information is available to help you get rid of these problems:

- Clearing the console
- Reducing console output
- Closing running applications
- Tweaking performance settings

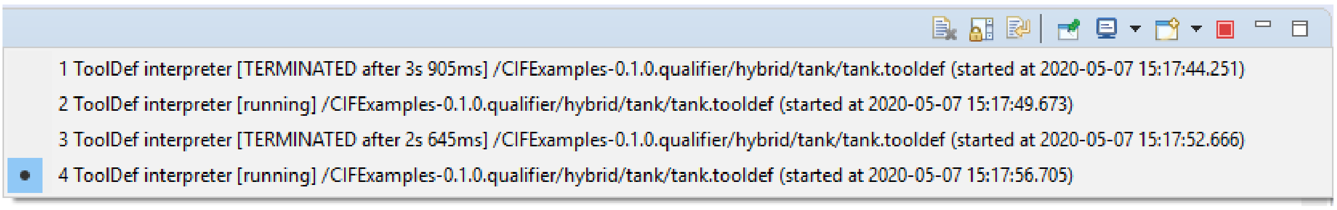
In particular, the Tweaking performance settings section provides information on how to give Eclipse ESCET tools more memory. This solves the most common performance problems and out-of-memory errors.

4.1. Clearing the console

In the Eclipse ESCET IDE, the *Console* view displays the console output generated by the applications that you run. This console keeps all the output in memory. If the application that you run creates a lot of output, this can quickly fill the available memory, and lead to out of memory errors.

By clearing the console, the output is removed and the associated memory becomes free for other uses. To clear the console, right click the console (the part of the view that contains the actual console text) and choose **Clear** from the popup menu. Alternatively, click the *Clear Console* button () of the *Console* view's toolbar.

The Eclipse *Console* view does not just keep the console output of the currently running application or applications in memory, it also keeps the output of all terminated applications in memory. To look at the applications that you executed, click the small arrow next to the *Display Selected Console* icon () of the *Console* view's toolbar.A list of executed applications will appear, that looks something like this:


1 ToolDef interpreter [TERMINATED after 3s 905ms] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:44.251)
2 ToolDef interpreter [running] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:49.673)
3 ToolDef interpreter [TERMINATED after 2s 645ms] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:52.666)
● 4 ToolDef interpreter [running] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:56.705)

In this case, four applications have been launched. The console output for the fourth application is currently displayed on the console, as indicated by the selection indicator on the left. Clicking on any of the other applications will activate the console for that application, and show its console output in the *Console* view. Clearing the console of applications that have terminated can free a lot of memory for other uses, if the applications produced a lot of console output.

Note that instead of clearing the console after a lot of output has been generated, it is often better to prevent that much output from being created in the first place.

4.2. Reducing console output

Console output is expensive. Not only because of the amount of memory the generated console output uses, but also because the console output itself needs to be generated, and displayed on the console. Reducing console output can significantly increase the performance of our tools.

Therefore, instead of clearing the console, it may be better to prevent that much output from being written to the console in the first place. If your model itself generates a lot of console output, consider letting it generate less output. Alternatively, if the tool you use generates a lot of console output, consider checking its options to see if you can disable certain console output.

All Eclipse ESCET applications have an *Output mode* option (*General* category). Changing the value of this option from *Debug* to *Normal*, or from *Normal* to *Warning* may significantly reduce the amount of output that is written to the console. Note however that this is mostly an all or nothing approach. It is often much better to use application specific settings, or change your model, to reduce the amount of output that is generated, as it allows for more control over what output is or isn't generated.

4.3. Closing running applications

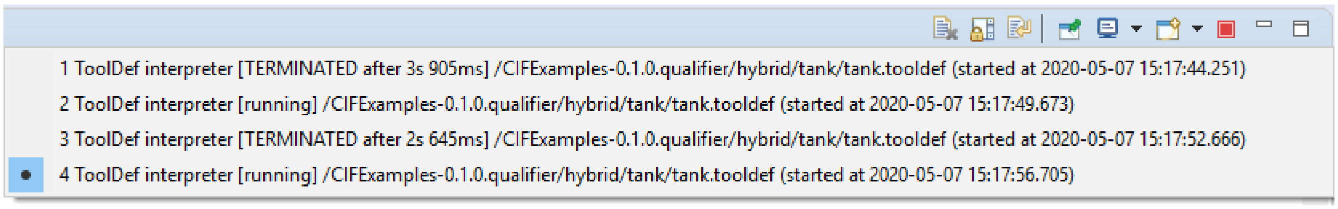
One of the nice features of an the Eclipse ESCET IDE is that it allows the simultaneous execution of multiple applications, as well as the simultaneous execution of a single application on many different inputs. For instance, it is possible to simultaneously simulate two models using a simulator. The downside is that running multiple applications at the same time, costs more memory. Sometimes, if you forget to close an application, it can still consume memory, and may even still be running active computations, thus consuming processing power.

4.3.1. Application management via the applications view

The Applications view can also be used observe the status of running applications, and to terminate them.

4.3.2. Application management via the console view

The Eclipse *Console* view maintains the console output for all running and finished applications. To look at the applications that you executed, click the small arrow next to the *Display Selected Console* icon (🖥) of the *Console* view's toolbar. A list of executed applications will appear, that looks something like this:



1 ToolDef interpreter [TERMINATED after 3s 905ms] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:44.251)
2 ToolDef interpreter [running] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:49.673)
3 ToolDef interpreter [TERMINATED after 2s 645ms] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:52.666)
4 ToolDef interpreter [running] /CIFExamples-0.1.0.qualifier/hybrid/tank/tank.tooldef (started at 2020-05-07 15:17:56.705)

In this case, of the four applications that have been started, the second and fourth are still running. The console for the fourth application is currently displayed, as indicated by the selection indicator on the left. Clicking on any of the other applications will activate the console for that application, allowing it to be terminated, thus freeing resources for other applications.

The application that is currently active in the console can be terminated at any time, by using the *Terminate* button (🟥), located at the upper right corner of the console. Note however that if the console does not have the focus, this button may not be visible. If the button is not visible, click somewhere in the console to make the button appear. If even then the button is still not available, it may still appear if you *Maximize* the console. Also note that the button has no effect while the application interactively asks for input from the console. However, once the console input is provided, and `ENTER` is pressed, the termination request will be processed.

4.4. Tweaking performance settings

If you run into errors related to running out of memory, you may need to tweak some settings. However, even if you don't get errors, tweaking settings can significantly improve performance.

This page provides a lot of background information, to allow you to better understand the impact of the various settings. If you wish, you can skip the background information, and go directly to the Quick and dirty solution section.

The following information is available on this page:

- Quick and dirty solution
- Managed memory and garbage collection
- Different types of memory
- Benefits of increasing the available memory
- Available settings
- Changing memory settings
- Practical hints to solve performance and memory problems
- Monitoring Eclipse heap status
- Monitoring with JVisualVM

4.4.1. Quick and dirty solution

This section explains a 'quick and dirty' solution that gives Eclipse more memory, resolving the most common performance problems and out-of-memory errors.

Find the `eclipse.ini` file. By default, it is located in your Eclipse ESCET installation directory, except for macOS, where instead it is in the `Eclipse.app/Contents/MacOS` directory inside the Eclipse ESCET installation directory. Modify the last line (usually `-Xmx4g`). Replace it by the following to change the maximum available memory from 4 GiB to 8 GiB:

```
-Xmx8g
```

Restart the Eclipse ESCET IDE or command line script to apply the new settings. If the instructions given here don’t fix your problem, or if the IDE or script will no longer start after you changed these settings, you should read the remainder of this page.

4.4.2. Managed memory and garbage collection

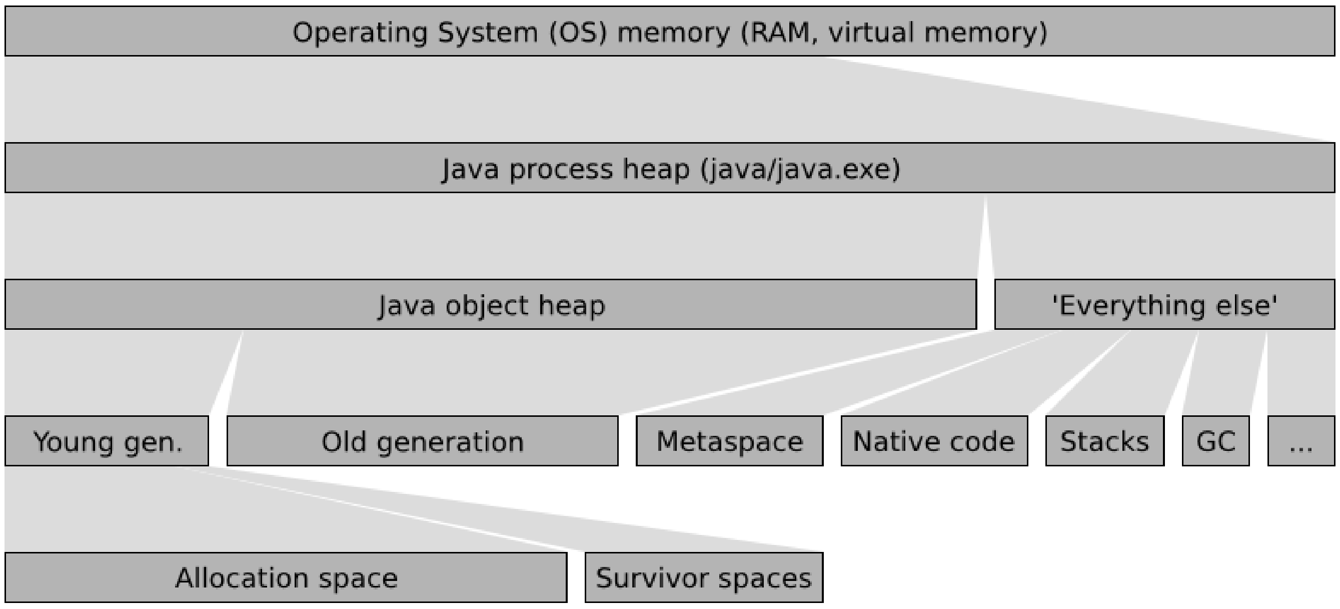
Before going into the actual settings, this section provides a little background on managed memory and garbage collection, to make it easier to understand the following sections. The information here is highly simplified, in order not to complicate matters too much.

The Eclipse ESCET IDE and command line scripts run on Java, a computer programming language. The *Java Virtual Machine* (JVM) manages all memory used by Eclipse, as well as the Eclipse ESCET tools. This page focuses on the Oracle JVM, as that is the JVM that we recommend. JVMs from other vendors may behave differently, and may have different settings. Also, new versions of the JVM often change/tweak their garbage collector, settings, defaults, etc. As such, the information on this page should be used to guide you, but may not be completely accurate.

The JVM keeps track of all data that is maintained by the Eclipse ESCET tools, and releases (frees) the memory once it is no longer needed, so that it can be used to store other data. The JVM frees memory by means of a process called garbage collection (GC). Garbage collection is a complex process, but generally it consists of locking the memory to avoid modification during garbage collection, finding the data that is no longer used (mark the garbage), and then freeing the memory associated with that data (sweep the marked garbage).

4.4.3. Different types of memory

In order to understand the memory related settings, some understanding of Java’s memory architecture is essential. The following figure provides an overview of Java’s memory architecture, and the different types of memory that are used:



The operating system (OS) has memory available, either as physical RAM, or as virtual memory. When Java is executed, the Java program (`java` executable on Linux and macOS, `java.exe` on Windows), becomes one of the running processes. The process uses a part of the operating system's memory to store its data. This memory is called the *Java process heap*. The *Java process heap* is divided into two parts, the *Java object heap* and *'Everything else'*. The *Java object heap* contains all data actually used by the running Java program, which in our case is the Eclipse ESCET IDE and/or Eclipse ESCET command line scripts. The *'Everything else'* part contains various data, mostly used by the JVM internally.

Java uses a generational garbage collector. New data, called *objects* in Java, are created in the *young generation*, or more specifically, in its *allocation space* (also called *eden space*). When the young generation becomes full, the garbage collector will remove all garbage (no longer used data) using a *minor collection*, which removes garbage from the young generation. The garbage collector uses the survivor spaces to store the surviving objects. Objects that survive a few minor collections are moved to the *old generation*, which stores the longer living objects, as well as the larger objects that don't fit in the young generation, which is usually much smaller than the old generation. When the old generation becomes full, the garbage collector performs a *major collection* removing garbage from the entire Java object heap, which is much more work, and thus much more costly than a minor collection.

The *'Everything else'* part of the Java process heap contains various data used internally by the JVM. This includes the *'Metaspace'* with all the Java code of Eclipse and our own plugins, the values of constants, etc. It also includes the *native code*, the highly optimized code generated for the specific architecture of your machine, that can actually be executed on your processor. Furthermore, it includes the stacks of all the threads that are running in parallel. There is also a part that contains the data maintained by the garbage collector itself, for administrative purposes. The 'Everything else' part contains various other types of data, that are irrelevant for the current discussion.

4.4.4. Benefits of increasing the available memory

If Java runs out of available memory, our applications running in Eclipse will terminate with an 'out of memory' error message. In such cases, increasing the available memory will likely solve the problem. However, even if you don’t run out of memory, increasing the amount of memory that is available to Java can significantly improve Java’s performance.

The garbage collector performs a minor collection when the young generation becomes 'full'. Here, 'full' doesn’t necessarily mean 100%, as Java by default tries to keep the heap about 40% to 70% filled. Increasing the size of the young generation makes it possible to allocate more new objects before the young generation becomes 'full'. During garbage collection, program execution may become halted, to ensure that memory doesn’t change during the collection process. The longer one can go without garbage collection, the less halting, and thus the greater the performance of the program.

If an application uses a lot of data that lives for longer periods of time, the old generation may become mostly filled with data. It then becomes harder and harder for the garbage collector to move objects from the young generation to the old generation. This may be caused by fragmentation, due to some objects from the old generation being removed by the garbage collector. In such cases, if the *gaps* are too small to hold the new objects, the old generation may need to be *compacted*, a form of defragmentation. After compaction, the single larger gap hopefully has more than enough free space to contain the new objects. The compaction process is expensive, as a lot of objects need to moved. If the situation gets really bad, Java may need to spend more time performing expensive garbage collection operations than it spends time on actually executing the program you’re running. By increasing the size of the old generation to more than the application needs, a lot more free space is available, reducing the need for frequent compaction, thus significantly increasing the performance of the application.

These are just some of the reasons why increasing the amount of available memory can improve program execution times, even though enough memory was already available to complete the given task. In general, the more memory Java has, the better it performs.

4.4.5. Available settings

The JVM has way too many options to list here, but the settings listed in this section are of particular practical relevance. Most of the settings affect memory sizes. Each setting is described using a name, a command line syntax (between parentheses), and a description. The command line syntax is used to specify the setting, as explained in the Changing memory settings section.

- Initial Java object heap size (`-Xms<size>`)

  The size of the Java object heap when Java starts. Java will increase and/or decrease the size of the Java object heap as needed.

- Maximum Java object heap size (`-Xmx<size>`)

  The maximum size of the Java object heap. Java will increase the size of the Java object heap as needed, but never to more than the amount indicated by this setting.

- Minimum percentage of free heap space (`-XX:MinHeapFreeRatio=<n>`)

  Java will increase the size of the Java object heap as needed. Frequent heap resizing is costly. To prevent frequent resizing, the JVM allocates more space than it really needs. This way, a lot of new objects can be allocated before running out of space, which requires the heap to be increased again.

  This setting indicates the desired minimum percentage of free heap space after each garbage collection operation. This is a desired percentage only, and if it conflicts with other settings, it is ignored. For instance, if this setting is set to 40% (the default), but 80% of the maximum heap size is in use, only 20% free space may be allocated.

- Maximum percentage of free heap space (`-XX:MaxHeapFreeRatio=<n>`)

  Java will decrease the size of the Java object heap if possible, to ensure that Java doesn't keep claiming memory that it no longer needs. Frequent heap resizing is costly. To prevent frequent resizing, the JVM allocates more space than it really needs. This way, a lot of new objects can be allocated before running out of space, which requires the heap to be increased again.

  This setting indicates the desired maximum percentage of free heap space after each garbage collection operation. The default is 70%.

- Ratio of young/old generation sizes (`-XX:NewRatio=<n>`)

  The ratio (1:n) of the young generation size to the old generation size. That is, with a ratio of 1:8, the old generation is 8 times as large as the young generation. In the command line syntax, the `8` is specified. The default value depends on the JVM that is used (Client VM vs Server VM, JVM version, 32-bit vs 64-bit, operating system, etc), but is usually `4`, `8`, or `12`.

- Ratio of allocation/survivor space sizes (`-XX:SurvivorRatio=<n>`)

  The ratio (1:n) of the survivor spaces size to the allocation space size. That is, with a ratio of 1:8, the allocation space is 8 times as large as the survivor space. In the command line syntax, the `8` is specified. The default value depends on the JVM that is used (Client VM vs Server VM, JVM version, 32-bit vs 64-bit, operating system, etc). Some of the defaults include `6`, `25`, and `32`.

- Use garbage collector overhead limit (`-XX:+UseGCOverheadLimit`)

By default, the JVM uses a policy that limits the proportion of the VM’s time that is spent on the garbage collector. If the limit is exceeded, the garbage collector has trouble doing its work (usually due to too little free memory), and performance is impacted so badly, that executed is practically halted. Instead of continuing, the JVM will issue an 'out of memory' error.

- Maximum code cache size (`-XX:ReservedCodeCacheSize=<size>`)

  The maximum size of the code cache for native code. The default value depends on the JVM that is used (Client VM vs Server VM, JVM version, 32-bit vs 64-bit, operating system, etc), and can be anything from `32m` to `2048m`.

- Client VM vs Server VM (`-client`, `-server`)

  The JVM can be run as either the Client VM or the Server VM. The Server VM performs more optimizations than the Client VM, leading to faster execution. However, these optimizations take time as well, making the Server VM start up slower than the Client VM. Note that the JVM compiles and optimizes code even during its execution. For longer running operations, the additional optimizations performed by the Server VM can make the execution significantly faster.

  The Client VM is not available on 64-bit JVMs. If the specified VM is not available, the setting is ignored. The default VM depends on the processor architecture and operating system. See Oracle’s Server-Class Machine Detection page for more information.

- Compile threshold (`-XX:CompileThreshold=<n>`)

  By default, the JVM runs in mixed mode, which means that some code is interpreted, while other code is compiled to native code, which runs much faster. Since compilation takes time as well, compilation is only performed for often used code.

  This setting indicates the number of method (a peace of Java code) invocations/branches before a method is compiled for improved performance. The default is `10000`.

- Thread stack size (`-Xss<size>`)

  The size of the stack of each thread.

The `<size>` part of the command line syntax is to be replaced by an actual size, in bytes. The size can be postfixed with a `k` or `K` for kibibytes, an `m` or `M` for mebibytes, or a `g` or `G` for gibibytes. For instance, `32k` is 32 kibibytes, which is equal to `32768`, which is 32,768 bytes.

The `<n>` part of the command line syntax is to be replaced by an integer number. The values that are allowed are option specific.

The `+` part of the command line syntax indicates that the corresponding feature is to be enabled. Replace the `+` by a `-` to disable the feature instead of enabling it.

4.4.6. Changing memory settings

There are several ways to supply the command line arguments for the settings to Java. The easiest way to do it, when using Eclipse, is to modify the `eclipse.ini` file. By default, it is located in your Eclipse ESCET installation directory, except for macOS, where instead it is in the `Eclipse.app/Contents/MacOS` directory inside the Eclipse ESCET installation directory.

Each of the settings you want to change should be added to the `eclipse.ini` text file, in the command line syntax. Each setting must be put on a line *by itself*. Furthermore, all these JVM settings must be put *after* the line that contains `-vmargs`. Settings on lines before the `-vmargs` line are the settings for the launcher that starts Eclipse, and should *not* be changed.

Note that the default `eclipse.ini` file supplied with Eclipse may already contain some of the settings. If so, don't add the setting again. Instead, change the value of the existing setting. The settings that are present by default, as well as their values, may change from release to release.

After modifying `eclipse.ini`, restart the Eclipse ESCET IDE or command line script for the changes to take effect.

Miscellaneous troubleshooting

If the `ECLIPSE_HOME` environment variable is defined, that directory is used instead of the default directory, to look for `eclipse.ini`. However, most users should not be affected by this.

Using the `-vmargs` command line option replaces the similar settings from the `eclipse.ini` file. For most users, this will not be applicable. If `--launcher.appendVmargs` is specified either in the `eclipse.ini` file, or on the command line, the `-vmargs` settings of the command line are added to the `eclipse.ini` file `-vmargs` instead, instead of replacing them.

4.4.7. Practical hints to solve performance and memory problems

In general, giving Java extra memory only makes it perform better. As such, increasing the maximum Java object heap size (`-Xmx`), is generally a good idea, if you have enough free memory.

If you actually run out of memory, Java will emit a `java.lang.OutOfMemoryError`, with a message to indicate the type of memory that was insufficient. Below the most common out of memory error message are listed, with possible solutions:

- `java.lang.OutOfMemoryError: Java heap space`

  The Java object heap needs more space. Increase the maximum Java object heap size (`-Xmx` setting).

- `java.lang.OutOfMemoryError: GC overhead limit exceeded`

  The 'use garbage collector overhead limit' feature is enabled, and the garbage collector overhead limit was exceeded. The best way to solve this, is to make sure the limit is not exceeded, by giving Java more memory, and thus making it easier for the garbage collector to do its work. Increase the maximum Java object heap size (`-Xmx` setting).

  Alternatively, disable the 'use garbage collector overhead limit' feature (`-XX:-UseGCOverheadLimit` setting, note the `-` instead of the `+`). However, this doesn’t solve the underlying problem, as the limit will still be exceeded. Java will try to continue, and will either fail, or be very slow.

- `warning: CodeCache is full. Compiler has been disabled.`

  This message is not a `java.lang.OutOfMemoryError`, but may still be printed to the console. It is usually followed by `warning: Try increasing the code cache size using -XX:ReservedCodeCacheSize=`. The warnings indicate that the code cache for native code is full. They already indicate the solution: increase the maximum size of the code cache (`-XX:ReservedCodeCacheSize` setting).

- `java.lang.OutOfMemoryError: unable to create new native thread`

  A new thread could not be created. The best way to solve this problem is to decrease the maximum Java object heap size (`-Xmx` setting), to make room for the 'Everything else' part of the Java memory, including the stack of the new thread.

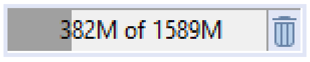
  Alternatively, decrease the size of stacks on all threads (`-Xss` setting). However, decreasing the thread stack size may cause more `java.lang.StackOverflowError` errors, and is thus not recommended.

Giving the JVM too much memory (especially via the `-Xmx` setting), can make the JVM fail to start, or crash shortly after starting. This problem mostly applies to 32-bit JVMs. The JVM needs to reserve a contiguous region of memory, or rather a contiguous region of the address space. On 32-bit JVMs, at most 4 GB of space can be addressed, of which a part is already in use by the operating system, drivers, and other applications. Thus, requesting too much memory, even if it is available, may cause problems, if no contiguous region of address space of that size is available when starting the JVM. If you request too much memory, you may get a `Invalid maximum heap size: -Xmx****m The specified size exceeds the maximum representable size. Could not create the Java virtual` `machine.` or `Error occurred during initialization of VM Could not reserve enough space for object heap` error when starting the JVM. For 64-bit JVMs, the address space is practically infinite, and this should not be a problem.

4.4.8. Monitoring Eclipse heap status

In Eclipse, it is possible to observe the amount of Java object heap space that is being used. In Eclipse, open the *Preferences* dialog, via **Window › Preferences**. Select the *General* category on the left, if not already selected. On the right, make sure the **Show heap status** option is checked, and click **[OK]** to close the dialog.

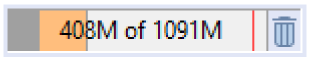
The heap status should now be displayed in the bottom right corner of the Eclipse window:



This example shows that the Java object heap (not the Java process heap) is currently 147 MB in size. Of that 147 MB, 62 MB are in use. The entire graph (the gray background) indicates the total heap size (147 MB), while the yellow part indicates the used part of the heap (62 MB).

Clicking on the garbage can icon, to the right of the heap status, will trigger a major collection cycle of the garbage collector.

By right clicking on the heap status, and enabling the *Show Max Heap* option, the heap status shows more information:



The text still shows the amount of used heap memory (74 MB) out of the total size of the current heap (147 MB). The scale of the background colors however, is different. The entire graph (the gray background) now indicates the maximum heap size. The orange part indicates the current heap size. The yellow part still indicates the part of the heap that is in use. If the used part of the memory gets past the red bar, it will become red as well, to indicate that you are approaching the maximum allowed Java object heap size, and may need to increase it (`-Xmx` setting).

Hover over the heap status to get the same information in a tooltip.

4.4.9. Monitoring with JVisualVM

The *Java Development Kit* (JDK) includes a program called JVisualVM. JVisualVM can be used to monitoring, troubleshoot, and profile running Java applications.

To start JVisualVM:

- On Windows, go to the directory in which the JDK is installed (usually something like `C:\Program Files\Java\jdk1.8.0_55`. Start `jvisualvm.exe` from the `bin` sub-directory, by double clicking on it.
- On Linux, find the directory in which the JDK is installed (may vary depending on the Linux

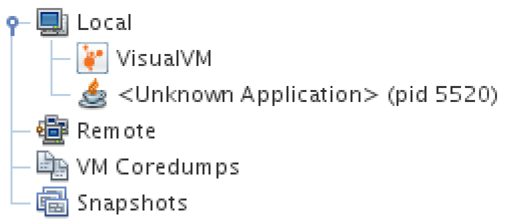
distribution that you use). Start `jvisualvm` from the `bin` sub-directory. You may be able to simply enter `jvisualvm` in a terminal window and press `ENTER`.

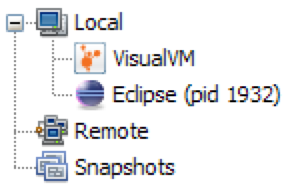
- On macOS, find the directory in which the JDK is installed (may vary depending on the Java version, and whether you use an Apple JDK or an Oracle JDK). A likely candidate is something like `/System/Library/Java/JavaVirtualMachines/1.8.0.jdk/Contents/Home`. Launch the `jvisualvm` binary from the `bin` sub-directory. You may be able to simply enter `jvisualvm` in a terminal window and press `ENTER`.

If your JDK does not contain JVisualVM, you can download it from the VisualVM website.

After you start JVisualVM for the first time, you'll see some dialogs. Just go through the steps until you get to the actual application.

In JVisualVM, you'll see the currently running Java applications, for the local system:

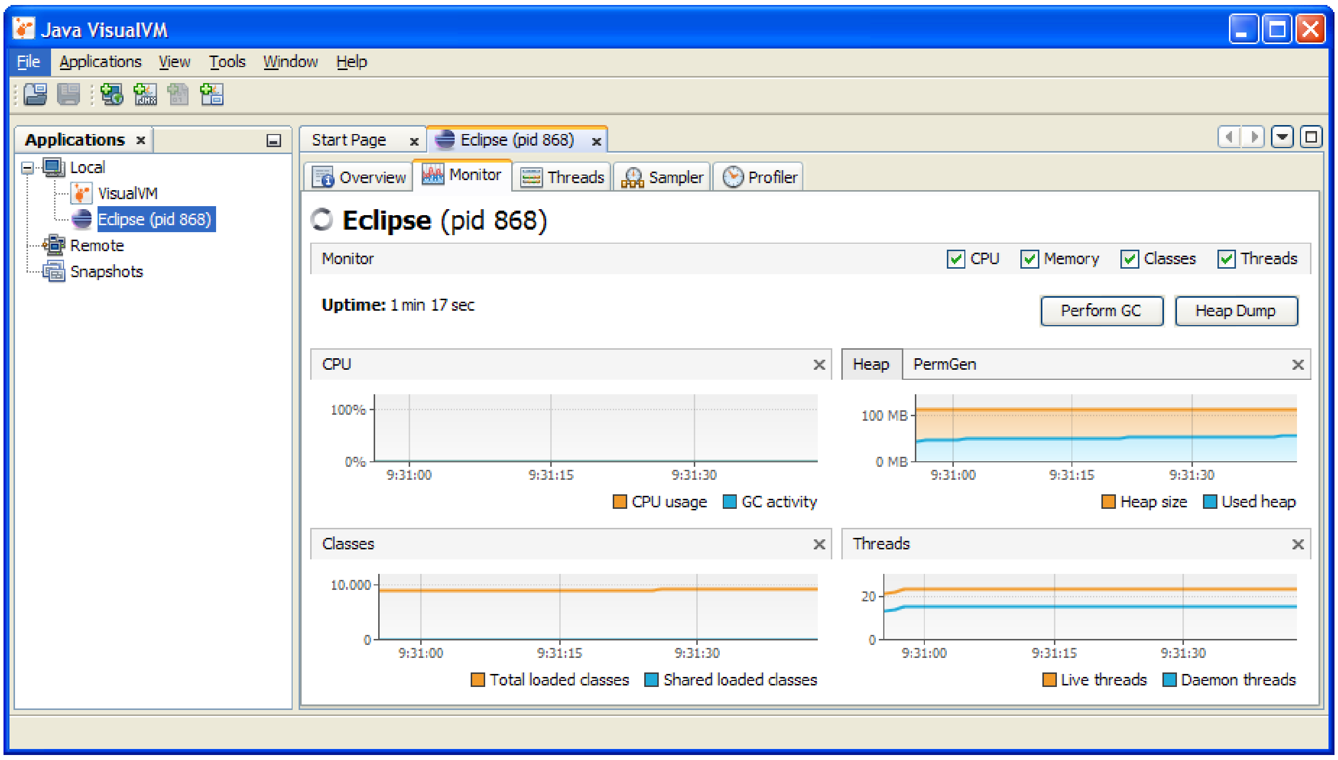




Sometimes JVisualVM can identify the Java applications, sometimes it can't. This may also depend on you operating system, and the version of JVisualVM. Find the application you want to know more about and double click it. A new tab opens on the right. The new tab has various tabs of its own:

- *Overview*: provides various basic information, including the location of the JVM, its command line settings, etc.
- *Monitor*: provides a quick overview of among others the CPU usage, GC activity, Java object heap usage, number of loaded classes, and the number of running threads, over time.
- *Threads*: provides an overview of the running threads, and their status, over time. The **[Thread Dump]** button can be used to dump the stack traces of all currently running threads.
- *Sampler* and *Profiler*: provide CPU and memory profiling, over time, by using sampling and instrumentation respectively.

The *Monitor* tab can be used to determine which type of memory should be increased. The *Sampler* tab can be used to profile an application, and figure out where bottlenecks are. This information can be used by the developers of the application to improve the performance of the application, by removing those bottlenecks.



Via **Tools › Plugins** you can access the *Plugins* window, where you manage the plugins. Various plugins are available. The *Visual GC* plugin is of particular interest. After installing it, restart JVisualVM, or close the tabs of the JVMs you're monitoring and open them again. You'll get an extra tab for monitored JVMs, the *Visual GC* tab. This tab is somewhat similar to the *Monitor* tab, but shows more detailed information about the garbage collector, its various generations, etc.

5. Eclipse ESCET development

Thanks for your interest in Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET) project.

For more information about the project, see:

- Eclipse ESCET project home
- Eclipse ESCET website

Contact the project developers via the project’s 'dev' list.

- Eclipse ESCET 'dev' list

For other means to interact with the Eclipse ESCET community and its developers, see the contact section.

Further topics:

- Development process
- Contributing
- Issue tracking
- Development environment setup
- Git repository
- Coding standards
- Building and testing
- Release process
- Third party tools
- Upgrade instructions

5.1. Development process

If you want to contribute to the Eclipse ESCET project, please refer to the specific contributing information.

The Eclipse ESCET project primarily uses GitLab for its development:

- Eclipse ESCET GitLab

5.1.1. Discussion

It is often a good idea to first discuss new ideas and features with the rest of the project developers, i.e. the project committers and the project community. Discussions can take place on the project’s 'dev' list, especially for radical new ideas and new features that have not been discussed before.

5.1.2. Issue tracking

If the project committers and the community agree that it is a good idea to have the new feature, an issue should be created in the issue tracker. For improvements where extensive discussion is not expected, as well as for obvious bugs, an issue can be created directly, without first discussing it on the 'dev' list. The discussion can also continue in the issue itself, once the issue is created.

When creating an issue, take the following into account:

- An issue must be created in the issue tracker for all development, however small. This ensures we can link issues can commits to keep track of everything.
- Issues are ideally kept relatively small in scope. Bigger tasks can be split up into multiple issues, and follow-up issues can be created as needed. This allows to separate concerns, and also to work in a more agile way, e.g.:
    - Issues can be addressed more quickly.
    - Merge requests can be reviewed more easily as they are not as big.
    - Merge issues are less likely to occur, as branches have a shorter lifespan.
- If a new feature is split up into multiple issues, these can be related together using an epic. In such cases, add the issues to the epic to track the progress of the new feature using the epic.
- Attach the appropriate predefined labels to the issue:
    - Either something is broken ('Bug' label) or we want something more, different, better, etc ('Enhancement' label).
    - Add all relevant component labels (e.g. 'Chi' and 'CIF' labels). Typically at least one such label should be present, but it is also possible to add multiple labels if the issue involves multiple components. The components correspond to the directories in the root of our Git repository.
    - The Eclipse ESCET project committers can add the 'Help Wanted' label to an issue to indicate that they don’t have the time to work on the issue, and that help from the community is wanted.
- Link the issue to any other issues, as relevant, e.g. if an issue requires that another issue is addressed first.

5.1.3. Releases and milestones

For every software version a GitLab milestone is created, to track its scope and progress. First the scope is discussed and agreed upon. Then, a GitLab milestone is created, the relevant issues are created if not yet present, and the issues are associated with the milestone. The issues can then be picked up to be addressed.

A single GitLab milestone is used per software version. Each software version has one or more milestone releases, followed by one or more release candidates, and is completed by a final release.

See also:

- Eclipse ESCET milestones

5.1.4. Working on issues

The process to work on issues is as follows:

- Unassigned issues can be picked up.
- Assign yourself to the issue when you are working on an issue, such that others won't start working on it as well.
- Unassign yourself if you are no longer working on an issue, don't plan to continue, and the issue is not finished.
- Don't unassign yourself after finishing the work, just close the issue.

5.1.5. Working with branches

The Eclipse ESCET project roughly follows the GitFlow branching model. The `master` branch is thus for released content only, and the current development status is captured in the `develop` branch.

If possible, we approach the somewhat heavy GitFlow branching model in a practical way, reducing overhead.

When creating and working with branches, consider the following:

- Always work in a branch for the issue. That is, never commit directly to the `develop` branch, but always use a merge request from a feature branch.
- Branches for work on issues (feature branches) should be relatively short lived. This makes it easier to keep the overview, allows for more agile development, and reduces the chance for merge conflicts.
- The GitFlow branching model allows for sub teams that share work but don't push that to 'origin' (yet). It also allows feature branches that live only locally on a developer's PC and not on 'origin'. To be transparent, Eclipse Foundation open source projects don't do this. We push to our GitLab server regularly, to ensure that the community can see what the project is working on.
- We have no strict branch naming rules. You can Let GitLab create the branch for the issue. For an issue with number #2 named 'Test', it will create a branch named `2-test`. This includes the issue number, which makes it easy to relate a branch to an issue, without having to look inside the branch for commits (if there even are any). It also includes the issue name, which is convenient as it indicates what the branch is about, without having to look up the issue.
- There are many ways to create a branch. One way is to create it from the GitLab issue. On the web page for a GitLab issue, there is a *Create merge request* button. Select the arrow to the right of it to show more options. Select *Create branch*. Adapt the *Branch name* and *Source* as needed. Typically the defaults suffice. Click the *Create branch* button to create the branch.

- We prefer not to create a draft merge request with the creation of the branch, as then commits in the branch lead to commits on the merge requests, which lead to notification emails.

5.1.6. Commits

As is standard for Git commits, the first line of the commit message must be a short summary, and must not exceed 72 characters.

For the Eclipse ESCET project, this line must start with the issue number, to allow GitLab to link commits to issues. For instance: `#1 commit summary`. In case a commit relates to multiple issues, list each of them, e.g. `#1 #2 commit summary`. Merge commits are exempt from this rule.

Furthermore, all commits must adhere to the requirements as defined by the Eclipse Foundation:

- Git Commit Records (Eclipse Foundation Project Handbook)

If you are not an Eclipse ESCET project committer, with write access to our Git repository, see the information on contributing to the Eclipse ESCET project. Don’t forget to sign-off your Git commits.

5.1.7. Merge requests

Once the work on an issue is done and pushed to a branch, it must be reviewed before it is merged back. Reviews are done via merge requests. The process is as follows:

- Create a merge request for merging the branch. You can create a merge request from the Eclipse ESCET Gitlab Branches page. Select the *Merge request* button next to the branch to be merged.
  - Typically a branch is created from and merged back to the `develop` branch, but this can be changed if needed.
  - If you include `Closes #nnn` in the description of the merge request, with `nnn` an issue number, that issue will automatically be closed once the merge request is merged. Use `Addresses #nnn` instead, if the merge request addresses part of the issue, but work remains, to prevent the issue from being closed. Always include either of them to ensure the merge request is properly linked to the issues it addresses.
  - It is not mandatory to select *assignees*, *reviewers*, etc.
- The merge request is reviewed by the Eclipse ESCET project committers.
- Improvements are made as necessary, reviewed again, etc, until the branch is considered to be ready to merge.
- The branch is merged and removed.
- If the branch fully addressed the associated issue or issues, it/they should be closed.

If you are not an Eclipse ESCET project committer, with write access to our Git repository, see the information on contributing to the Eclipse ESCET project.

5.2. Contributing

Thanks for your interest in Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET) project.

It is often a good idea to first discuss your contribution with the project's community and committers, before creating the actual code (e.g. patches), documentation, etc of your contribution. Discussions can take place via an issue in the issue tracker, or on the project's 'dev' list.

To contribute your actual contribution, e.g. code, documentation, examples, or anything else to the project, please first create an issue in the issue tracker.

The easiest way to contribute the actual contribution, is to use GitLab:

- Ensure you're logged in to the Eclipse Foundation GitLab:
  - Eclipse Foundation GitLab
- Clone the official Eclipse ESCET Git repository. You can click the 'Fork' button at the top right of the page. This creates a fork of the official repository under your own account.
  - Official Eclipse ESCET Git repository
- Make your changes in the forked repository under your own account.
- Create a merge request of those changes from the forked repository under your own account. As the target, select the official Eclipse ESCET repository from which you forked earlier. Typically, you should target the `develop` branch.

Before your contribution can be accepted by the project team, you must electronically sign the Eclipse Contributor Agreement (ECA):

- Eclipse Contributor Agreement (ECA)

Commits that are provided by non-committers must have a `Signed-off-by` field in the footer indicating that the author is aware of the terms by which the contribution has been provided to the project. The non-committer must additionally have an Eclipse Foundation account and must have a signed Eclipse Contributor Agreement (ECA) on file.

For more information, including the specific format of commit messages, please see the Eclipse Foundation Project Handbook:

- Eclipse Foundation Project Handbook
- Git Commit Records

Always start a commit message with the issue number, e.g. `#1 Some commit message`.

A contribution by a non-committer will be reviewed by the project committers. This includes adherence to the project's coding standards. Discussions regarding the contribution will take place in the associated issue and/or merge request. If the committers agree with the contribution, they will commit the contribution into the project’s Git repository.

Remember that contributions are always welcome, and contributions don’t have to be perfect. The project’s developers can help to improve your contribution. If you need any help, just ask the project’s developers using the issue or the project’s 'dev' list.

See for more information our development process.

5.3. Issue tracking

The Eclipse ESCET project uses GitLab to track ongoing development and issues:

- Eclipse ESCET milestones
- Eclipse ESCET issues

Be sure to search for existing issues before you create another one. Remember that contributions are always welcome!

To contribute code (e.g. patches), documentation, or anything else, see the contributing section.

To understand how we work with issues, see our development process.

5.4. Development environment setup

Follow these instructions to set up an Eclipse ESCET development environment.

Get the Eclipse Installer:

- Go to https://www.eclipse.org/ in a browser.
- Click on the big **[Download]** button at the top right.
- Download Eclipse Installer, 64 bit edition, using the **[Download 64 bit]** button.

To create a development environment for the first time:

- Start the Eclipse Installer that you downloaded.
- Use the hamburger menu at the top right to switch to advanced mode.
- For Windows:
    - When asked to keep the installer in a permanent location, choose to do so. Select a directory of your choosing.
    - The Eclipse installer will start automatically in advanced mode, from the new permanent location.

- For Linux:
    - The Eclipse installer will restart in advanced mode.
- Continue with non-first time instructions for setting up a development environment.

To create a development environment for a non-first time:

- In the first wizard window:
    - Select *Eclipse Platform* from the big list at the top.
    - Select *2020-06* for *Product Version*.
    - Select a Java 8 JDK for *Java 1.8+ VM*.
    - Choose whether you want a P2 bundle pool (recommended).
    - Click **[Next]**.
- In the second wizard window:
    - Use the green '+' icon at the top right to add the Oomph setup.
        - For *Catalog*, choose *Eclipse Projects*.
        - For *Resource URIs*, enter `https://gitlab.eclipse.org/eclipse/escet/escet/-/raw/develop/org.eclipse.escet.setup` and make sure there are no spaces before or after the URL.
        - Click **[OK]**.
    - Check the checkbox for *Eclipse ESCET*, from the big list. It is under *Eclipse Projects / <User>*.
    - At the bottom right, select the *develop* stream.
    - Click **[Next]**.
- In the third wizard window:
    - Enable the *Show all variables* option to show all options.
    - Choose a *Root install folder* and *Installation folder name*. The new development environment will be put at `<root_installation_folder>/<installation_folder_name>`.
    - For *JRE 1.8 location* make sure to select the same Java 8 JDK that you selected in the first wizard window.
    - Click **[Next]**.
- In the fourth wizard window:
    - Select **[Finish]**.
- Wait for the setup to complete and the development environment to be launched.
    - If asked, accept any licenses and certificates.
- Press **[Finish]** in the Eclipse Installer to close the Eclipse Installer.
- Click the rotating arrows icon in the status bar (bottom right) of the new development environment.
- Observe Oomph executing the startup tasks (such as Git clone, importing projects, etc).

- Wait for the startup tasks to finish successfully.
- NOTE: If you don't open the Oomph dialog, the status bar icon icon will disappear when the tasks are successfully completed.

If you have any issues during setting up the development environment, consider the following:

- Make sure to use a recent version of the Eclipse Installer. These versions include a recent JRE, which includes up-to-date certificates, preventing issues when accessing https URLs.
- You can set the following environment variables to force the use of IPv4, in case of any issues accessing/downloading remote files:

  ```
  _JAVA_OPTIONS=-Djava.net.preferIPv4Stack=true
  _JPI_VM_OPTIONS=-Djava.net.preferIPv4Stack=true
  ```

  After setting them, make sure to fully close the Eclipse Installer and then start it again, for the changes to be picked up.

In your new development environment, consider changing the following settings:

- For the *Package Explorer* view:
  - Enable the *Link with Editor* setting, using the icon.
  - Enable showing resources (files/folders) with names starting with a period. Open the *View Menu* ( ) and choose **Filters…**. Uncheck the `.* resources` option and click **[OK]**.

5.5. Git repository

The Eclipse ESCET project maintains the following source code repositories:

- `https://gitlab.eclipse.org/eclipse/escet/escet.git`

These can also be accessed via a web interface:

- Eclipse ESCET GitLab

The software is written in the Java programming language, as a collection of Eclipse plugins, and using the Eclipse Modeling Framework (EMF).

For ease of programming, the Eclipse IDE is recommended. See also the section on setting up a development environment.

To contribute code (e.g. patches), documentation, or anything else, see the contributing section.

The way we work with our Git repository is explained as part of our development process.

5.5.1. Repository structure

In the Eclipse ESCET source code repository (Git repository), three layers are distinguished:

- The top layer contains user-oriented languages for designing controllers. Currently there are two modeling languages in this layer, CIF and Chi.

  The third language in the top layer is ToolDef, a cross-platform scripting language to run tests, and to automate the various tools that need to be executed while designing a controller.

- The middle layer contains developer oriented support code. It has a language of its own, named SeText. This language implements an LALR(1) parser generator with a few twists to make it easy to use in a Java environment and to connect it to an Eclipse text editor.

  The other part of the middle layer is common functionality shared between the languages.

- The bottom layer is mostly configuration to attach the software to the Eclipse platform, including build and release engineering.

This document describes the structure of the top and middle layers. For the bottom layer, standard Eclipse and Maven/Tycho tools are used, which are described elsewhere.

The three layers are not further distinguished in the repository. Instead, different parts are stored in different sub-directories from the root.

5.5.2. Language directories

Each language has its own subdirectory in the root, `/cif` and `/chi` for the CIF and Chi modeling languages, `/tooldef` for the ToolDef language, and `/setext` for the SeText language.

Within a language directory, a directory exists for each part of the code (often equivalent to a plugin), with the same name as the plugin. The pattern of a plugin name is `org.eclipse.escet.<language>.<plugin-name>` where the plugin-name in different directories has the same meaning. A non-exhaustive list:

Plugin name	Description
`documentation`	User-oriented documentation about the language, such as a user manual or a reference manual.
`metamodel`	Ecore metamodel back bone of the language. Model classes for the central data structure that all tools of the language use. Often generated using modeling tools such as Sirius, but manually written classes exist as well.
`metamodel.java`	Generated Java constructor and walker classes for the Ecore metamodel data structure.
`parser`	SeText input, and generated or manually written code to parse an input file and convert it to an tree of classes that can be given to the type checker.
`typechecker`	Implementation of the type checker to check the parsed input, and annotate it with derived information, resulting in an model instance that can be used by all tools of the language.

Plugin name	Description
io	Text file loading, parsing, and type checking, and possibly writing result specifications.
texteditor	Code for editing source files of the language in an text Eclipse editor, with folding, syntax highlighting, and reporting of errors and warnings in the specification.
tests	Collection of tests to check the tools for the language. Typically a set of input specifications, a set of expected output files, and a ToolDef script to run the tests. These tests can be seen as integration tests.
<plugin-name>.tests	Unit tests for that specific plugin.
codegen	Code generator to convert the input specification to a runnable model.
runtime	Support libraries used by the runnable model.
tooldefs	Interface definitions to make tools of the language available for ToolDef.
common	Common functionality used by many tools of the language.

Other plugin names are often tools with the same name.

5.5.3. Middle layer common functionality

The common code between all languages is stored in the `/common` directory, again with full name of the plugin as sub-directory names. These plugins contain:

Plugin name	Description
org.eclipse.escet.common.app.framework	Common application framework.
org.eclipse.escet.common.app.framework.appsview.ui	The Applications view. User interface of the active applications.
org.eclipse.escet.common.box	Library to generate formatted code-like text.
org.eclipse.escet.common.eclipse.ui	Common Eclipse User Interface code.
org.eclipse.escet.common.emf	Common *EMF* code.
org.eclipse.escet.common.emf.ecore.codegen	Code generators from *Ecore* files.
org.eclipse.escet.common.emf.ecore.validation	Common *EMF* validation code.
org.eclipse.escet.common.emf.ecore.xmi	Common *EMF XMI* serialization code.
org.eclipse.escet.common.java	Common *Java* functions and classes, in particular the *Lists*, *Sets*, *Maps* and *Strings* classes.
org.eclipse.escet.common.position.common	Common functions for (text-file) positions in source files.

Plugin name	Description
`org.eclipse.escet.common.position.metamodel`	The Ecore metamodel for (text-file) positions in source files.
`org.eclipse.escet.common.svg`	Common *SVG* library for viewing and manipulating *SVG* trees.
`org.eclipse.escet.common.typechecker`	Common type checker functionality.

5.6. Coding standards

The Eclipse ESCET development environment has some features that allow developing high quality contributions:

- A Java formatter profile is included. It allows to automatically format Java code for consistency and convenience.
- The Eclipse Checkstyle Plugin and a Checkstyle configuration are included. They can be used to detect various other issues in Java code and other files.

For all contributions to the Eclipse ESCET project, check the following:

- All Java code should be formatted using the provided formatting profile.
- All contributions should be checked using the provided Checkstyle configuration.
- All contributions should be free of warnings and errors, when working with them in the Eclipse ESCET development environment.

Remember that contributions are always welcome, and contributions don't have to be perfect. The project's developers can help to improve your contribution, and ensure it adheres to these coding standards.

For any questions regarding these coding standards, please contact the project's developers.

5.7. Building and testing

The Eclipse ESCET software can be built using Maven/Tycho. The build will build every individual plugin and feature, as well as the update site, product and all documentation. Manually run it in one of the following ways:

- In an Eclipse-based development environment, select **Run** › **Run Configurations...** to open the *Run Configurations* dialog. From the list at the left, under *Maven Build*, select the *build* launch configuration, and click the **[Run]** button to run the build.

  Additional launch configurations are available to build documentation projects separately.

- On Windows, in a command prompt, with the root of the Git repository as current directory, enter `.\build.cmd` and press `ENTER`. This requires Maven to be available on your system (to be on your `PATH`).

- On Linux/macOS, in a shell, with the root of the Git repository as current directory, enter `./build.sh` and press `ENTER`. This requires Maven to be available on your system (to be on your `PATH`).

Builds are in certain cases also automatically executed on our build server:

- Eclipse ESCET Jenkins server

Run tests in one of the following ways:

- As part of the build, all tests will be performed as well.
- Launch configurations to run the integration/regression tests for a specific language are also available, under the *JUnit Plug-in Test* section of the *Run Configurations* dialog.

5.8. Release process

The process to release a new version of the Eclipse ESCET tools involves the following steps:

- Prepare for the next release in the `develop` branch until it is ready to be released.
- Create a GitLab merge request from `develop` to `master`, and merge it. Since `master` is a protected branch for the Eclipse ESCET GitLab, a GitLab merge request is the only way to update it.
- Check that the build on `master` succeeds in Jenkins.
- Add a tag on the commit in `master` that is to be released. Only version tags with a specific syntax will be picked up by Jenkins to be released. For instance, use `v0.1`, `v0.1.1`, `v2.0`, etc for releases, `v0.1-M1` for a milestone build, or `v0.1-RC1` for a release candidate.

  Add the tag via GitLab, at https://gitlab.eclipse.org/eclipse/escet/escet/-/tags/new. Use the *Tag name* also as *Message*. Make sure to select `master` as branch from which to create the tag. For the *Release notes*, use a short text inspired by the release plan.

  As an example, consider milestone 1 of release v0.1: see https://projects.eclipse.org/projects/technology.escet/releases/0.1 for the release plan, https://gitlab.eclipse.org/eclipse/escet/escet/-/tags/v0.1-M1 for the tag and https://gitlab.eclipse.org/eclipse/escet/escet/-/releases/v0.1-M1 for the GitLab release record.

- Edit the GitLab release record, adding the GitLab milestone to the release record. Save the changes.
- Jenkins will automatically pick up the new tag. Log in to Jenkins and manually trigger a build for the tag. Jenkins will then automatically build and release a new version from that tag.
- All releases can be downloaded at https://download.eclipse.org/escet/. For a version `v0.1`, the downloads will be located at `https://download.eclipse.org/escet/v0.1`.

  Note that according to the Eclipse Foundation Wiki page IT Infrastructure Doc, "Once your files are on the `download.eclipse.org` server, they are immediately available to the general public. However, for release builds, we ask that you wait at least four hours for our mirror sites to fetch the new files before linking to them. It typically takes a day or two for all the mirror sites to synchronize with us and get new files."

That same wiki page also notes that "Although you can link directly to `download.eclipse.org/yourfile.zip`, you can also use the *Find a Mirror* script [...]. Using this script allows you to view download statistics and allows users to pick a nearby mirror site for their download." It further indicates that "P2 repositories are not normally accessed via the mirror selection script." The *Find a Mirror* script also transparently handles files moved from `download.eclipse.org` to `archive.eclipse.org`.

- Jenkins will automatically push the website for the new release to the website Git repository, in a directory for the specific release. For a version `v0.1`, the website can be accessed at `https://www.eclipse.org/escet/v0.1`. It may take a few minutes for the Git repository to be synced to the webserver and for the website for the new version to become available.
- If the website for the new release is to be the standard visible website for the project (at `https://www.eclipse.org/escet`), it has to be manually replaced. This is to ensure that a bugfix release for an older version doesn't override the standard visible website. The following steps explain how to 'promote' a website for a specific version to become the standard visible website:
    - Make sure you've uploaded your SSH public key to Eclipse Gerrit. This is a one-time only step. Go to https://git.eclipse.org/r/. Sign in using your Eclipse Foundation committer account. Use the gear icon at the top right to access your account settings. Under *SSH Keys* add your SSH public key. Also make note of your *username*, *Full name* and *Email* address.
    - Clone the Eclipes ESCET website Git repository using `git clone ssh://<username>@git.eclipse.org:29418/www.eclipse.org/escet.git`. Make sure to replace `<username>` by your Eclipse Foundation committer account *username*.
    - In the cloned repository, remove all files/folders in the root of the Git repository pertaining to the current standard visible website. Be sure not to remove any of the directories with websites for specific releases.
    - Copy the files/folders from the directory with the website for the release that you want to make the standard visible website, and put them in the root of the Git repository.
    - Add all changes to be committed, e.g. by using `git add -A`.
    - Make sure to use the *Full name* and *Email* address of your Eclipse Foundation committer account. E.g. use `git config --local user.name "<full_name>"` and `git config --local user.email "<email>"`, replacing `<full_name>` and `<email>` by the appropriate information matching your Eclipse Foundation committer account.
    - Commit the changes. Use as commit message `Set standard visible website to release <version>.`, replacing `<version>` by the release version that will become the new standard visible website. Make sure to sign off the commit to pass Eclipse Foundation automatic commit validation. E.g. use the following to commit the changes: `git commit -s -m "Set standard visible website to release v0.1."`.
    - Push the changes to the Git server. E.g. use `git push`. If successful you should see the changes at the Git server's web view, at https://git.eclipse.org/c/www.eclipse.org/escet.git/.
    - It may take a few minutes for the Git repository to be synced to the webserver, and for the

new standard visible website to become available. The standard visible website can be accessed at `https://www.eclipse.org/escet`. Depending on browser cache settings and other factors, it may be necessary to force refresh your browser for it to pick up the changes on the server.

- Remove/archive old releases:
  - For every release (not a milestone or release candidate), remove all milestones and release candidates of the previous version and older. E.g. for `v0.2`, remove `v0.1-M1`, `v0.1-M2`, `v0.1-RC1`, etc. Also archive all releases older than the current and previous release. E.g. for `v0.3` archive `v0.1` and older, but keep `v0.2`.

    Consider whether archiving or removing P2 update sites will lead to issues for users. See for more information the Eclipse Foundation Wiki page Moving a repo to archive.eclipse.org.
  - Old websites can be removed in a similar way to the above instructions to change the default website. Only remove the directory for the milestone or release candidate.
  - Older downloads can be archived. Go to https://download.eclipse.org/escet/. Make sure you're logged in. This should make check-boxes appear. Select the folders to archive and click the *Archive* button. It may take a few minutes for archiving to complete.
  - Archived downloads can be removed. Go to https://archive.eclipse.org/escet/. Make sure you're logged in. This should make check-boxes appear. Select the folders to delete and click the *Delete* button. It may take a few minutes for deleting to complete.

5.9. Third party tools

As part of development for the Eclipse ESCET project, several third party tools are used. They are used to e.g. run scripts, generate files, etc.

The following third party tools are used to run scripts:

- Bash, to run `.bash` scripts.
- GNU utilities, to use in scripts, e.g. `cat`, `cp`, `diff`, `dirname`, `find`, `grep`, `mv`, `readlink`, `rm`, `sed`, `sort` and `wc`.
- Perl, to run `.pl` scripts.
- Python, version 3, to run `.py` scripts.
- Shell, to run `.sh` scripts.
- Windows command prompt, to run `.cmd` scripts.

The following third party tools are used to build:

- Maven, to run the main build from a console.

The following third party tools are used to generate/convert images:

- `bbox_add.pl` Perl script, used in conjunction with LaTeX, obtained from http://www.inference.org.uk/mackay/perl/bbox_add.pl.

- `eps2png` Perl script, used in conjunction with LaTeX, obtained from https://metacpan.org/pod/eps2png.
- Gnuplot, to generate images.
- ImageMagic, used in conjunction with LaTeX, including `convert`.
- Inkscape, to convert `.svg` images.
- LaTeX, to generate images, including `dvips`, `latex`, `pdfcrop` and `pdflatex`.
- Make, run `Makefile` builds, to generate images.
- LaTeX `rail` package, including `rail`.

The following third party tools are used to build some of the documentation:

- LaTeX, including `bibtex` and `pdflatex`.

The following third party tools are used to generate test classes and package them into a JAR file, for certain tests:

- Java Development Kit (JDK), version 7 or higher, including `javac` and `jar`.

Most of these tools are not needed to run a build or run the tests, as the generated files (e.g. images) are committed into Git.

5.10. Upgrade instructions

To upgrade to a new Eclipse Platform/IDE/SDK version:

- Version updates
  - Look up Orbit version for the new Eclipse Platform/IDE/SDK release, see https://download.eclipse.org/tools/orbit/downloads/.
  - Update Oomph setup, configuring new Eclipse and Orbit versions.
  - Update `dev-env-setup.asciidoc` to match new Eclipse version.
  - Update `org.eclipse.platform` version for the product feature (`org.eclipse.escet.product.feature` project).
  - Update Eclipse and Orbit update site URLs in product.
- New development environment
  - Set up a new development environment.
  - Commit target platform changes after regenerated by Oomph.
  - Check workspace for any errors/warnings and address them if any.
  - Check *New and Noteworthy* (release notes) for changes and adapt as necessary.
- Java formatter profile
  - Make a dummy change to the Eclipse ESCET Java formatter profile, and change it back.
  - Compare the new configuration against the old configuration, to see if there are any new

    settings.
  - In case of new settings, configure them as desired.
  - Reformat all Java code using the new formatter profile.
  - Update the formatter profile in the Oomph setup.
- Java errors/warnings settings
  - Check the properties of the `org.eclipse.escet.common.java` project, under **Java Compiler › Errors/Warnings**.
  - Make a dummy change and change it back.
  - In case of changes to `*.prefs` files in `.settings`, configure the new settings as desired.
  - Run `misc/java-code-style/copy_here.bash ../../common/org.eclipse.escet.common.java` from `misc/java-code-style` to copy the new settings to the central place.
  - Run `misc/java-code-style/copy_there.bash` from `misc/java-code-style` to copy the new settings to all relevant projects.
  - Force a rebuild in Eclipse and check for any warnings/errors, addressing them if any.
- Validation
  - Run a Maven build.

To upgrade to a new Tycho version:

- Update version in `.mvn/extensions.xml`.
- Update version in `releng/org.eclipse.escet.configuration/pom.xml`.
- Check release notes for changes and adapt as necessary.
- Run a Maven build.

6. Application framework

The Eclipse ESCET application framework provides common functionality for applications within the Eclipse ESCET toolkit. The following topics explain the framework in more detail:

6.1. Introduction

The Eclipse ESCET application framework provides common functionality for applications within the Eclipse ESCET toolkit. It has several goals:

- Provide a uniform end-user experience, for example in the form of uniform option dialogs.
- Hide technical details from the end user, for example in the form of crash reports and user friendly error messages, instead of stack traces.
- Provide support for applications to run both as a stand-alone Java program (say, from the command line), as well as within the Eclipse environment.
- Provide the basic functionality needed by most applications, to reduce the overhead needed for developers to develop an application.

The documentation for this framework describes the issues that the application framework attempts to solve, and the way it solves them. It also provides guidance in implementing applications using the application framework.

6.2. Stand-alone execution versus Eclipse IDE

One of the goals of the application framework is to make it easier to allow applications to run as stand-alone Java command line applications, as well as run within the Eclipse IDE. The main problem faced when supporting general applications to run within Eclipse, is that such applications all run within the same instance of the Java Virtual Machine (JVM). In fact, a single application may have multiple instances running at the same time, within a single instance of the IDE. The following sections address the issues that arise when running within the IDE, and how the application framework handles them.

6.2.1. Application static information

Within Java programs, members can be defined with the `static` modifier. Since multiple instances of an application may be running simultaneously, within a single instance of the IDE, one should avoid using static variables that contain information that is specific to a single instance of the application. For instance, assume an application that maintains an integer counter, used to generate unique identifiers. If defined in a class as follows:

```
public static int count = 0;
```

and incremented when needed, the first instance of the application will run just fine. Variable `count` starts at zero, and is incremented over and over again. When a second instance of the application starts however, the static variable keeps its value, as the new application is started within the same Eclipse instance, and thus within the same JVM. The count won't start from zero, thus leading to different results for the application.

The conclusion is that one should be careful to avoid static variables that hold information specific to an application instance.

6.2.2. Application options

Applications often have settings, and they are generally passed as command line arguments. GUI applications however, often use a dialog to configure the options instead. To allow applications within the application framework to work in both scenarios, all applications should use the option framework.

See also the option framework section.

6.2.3. Stdin, stdout, and stderr

Command line applications generally obtain input from stdin, and write output to stdout and/or stderr streams. For applications running within the IDE, those streams are connected to the Eclipse application (IDE) as a whole, and not to the applications running within the IDE. To provide a uniform I/O interface, the application framework includes an I/O framework.

See also the I/O framework section.

6.2.4. Graphical User Interfaces (GUIs) and SWT

The Eclipse IDE uses the Standard Widget Toolkit (SWT) for its graphical user interface (GUI). To be compatible with the Eclipse IDE, all GUI applications should use SWT as well. In order for GUI applications to work seamlessly within the Eclipse IDE as well as stand-alone, the application framework automatically registers the main SWT display thread for stand-alone applications, and uses the Eclipse SWT display thread when running within the Eclipse IDE. This reduces the burden of having to register the main SWT display threads, but also avoids blocking and other thread related issues.

Using the *GUI* option, the GUI can be enabled or disabled. If disabled, headless execution mode is used, which disables creation of a SWT display thread, and thus disables all GUI functionality, including the option dialog.

6.2.5. Application termination

Within Java, the `System.exit` method can be used to immediately terminate an application, by terminating the JVM. For applications running within the Eclipse IDE, this not only terminates the application, but the IDE as well. As such, the `System.exit` method should never be used in applications that are intended to be executed within the IDE.

6.2.6. SIGINT

Stand-alone applications can typically be started from a the command line terminal window. Pressing `Ctrl`+`C` at such a command line terminal window terminates the currently running application (on Unix-based systems, this generates a SIGINT). Applications running within the Eclipse IDE however, don't run in an actual command line terminal. Instead, they run within the IDE, and the stdin, stdout, and stderr streams are coupled to the Eclipse console view. The Eclipse console view does not support termination using the `Ctrl`+`C` key combination. Instead, `Ctrl`+`C` is used to copy console output to the clipboard. To remedy this situation, application framework applications running within Eclipse get a *Terminate* button with their console within the IDE, to allow for easy termination.

Furthermore, the application framework allows termination requests via the `AppEnv.terminate` method. Application framework applications and threads should regularly call the `AppEnv.isTerminationRequested` method to see whether they should terminate.

See also the termination features of the Applications view.

6.2.7. Exceptions

Exceptions are Java feature that allows applications to report error conditions. Exceptions can generally be divided into two categories: internal errors, and end-user errors. Internal errors should generally not happen, and make the application crash. The application framework provides crash reports for end users to report crashes due to internal errors. The application framework also provides exception classes for end-user errors, to provide nice error messages, instead of stack traces.

See also the exception framework section.

6.2.8. System properties

Java uses system properties (`System.getProperty` method etc). Those properties are global to the entire JVM, meaning they are shared between applications running within the Eclipse IDE. The application framework provides functionality to maintain system properties on a per application basis, turning them into application properties. All application framework applications should use the `getProp*` and `setProp*` methods in the `AppEnv` class instead of the property related methods in the `System` class. This ensures that the application properties are used instead of the global system properties.

6.2.9. File paths and current working directory

One of the standard system properties in Java is the `user.dir` property, which refers to the current working directory, or more precisely, the directory from which the JVM was started. Java doesn't allow changing the current working directory. The application framework however, maintains the current working directory on a per application basis. Changing the current working directory is also supported. Application framework applications should use the methods in the `org.eclipse.escet.common.app.framework.Paths` class to get and set the current working directory, to resolve relative paths, etc. These methods also allow both Windows (`\`) and Unix (`/`) separators to be used in paths, on all platforms, transparently.

Furthermore, within the Eclipse IDE, the concept of a workspace is introduced. In order to allow importing of resources from other projects etc, it may be nice to allow end users to specify platform paths (plug-in paths or workspace paths). Eclipse Modeling Framework (EMF) URIs, besides local file system paths, provide functionality for platform URIs as well. EMF URIs can for instance be used to load models that are instances of an Ecore. The `Paths` class mentioned above features methods to create such EMF URIs, from various sources. Those methods also feature smart handling of `platform:/auto/...` paths, an addition to platform URIs, added by the application framework. Such URIs are first resolved in the workspace, and if they can't be found there, they are resolved in the plug-ins. This allows for easier debugging, as the workspace always overrides the plug-ins.

6.3. The Application class

The `org.eclipse.escet.common.app.framework.Application<T>` class is the main class of the application framework. All application should inherit from this abstract class. The generic parameter `<T>` is further explained in the section about the I/O framework.

The next sections introduce the specific parts of the application framework. After that, you'll find a section on how to implement your own application, using the application framework.

6.4. The exception framework

The application framework contains the exception framework. Its main goal is to hide stack traces from end users. Exceptions can generally be divided into two categories: internal errors, and end-user errors.

6.4.1. End-user exceptions

All exceptions that should be presented to the end user are considered end-user exceptions. These messages should be written in terms that the end user should be able to understand. For end-user exceptions, the exception framework does not display stack traces (at least not by default). All end-user exceptions must implement the `org.eclipse.escet.common.app.framework.exceptions.EndUserException` interface, and may inherit from the `org.eclipse.escet.common.app.framework.exceptions.ApplicationException` class. All applications that use the application framework must satisfy these requirements when the error message is to be presented to end users. It is recommended to reuse existing application framework exceptions whenever possible.

6.4.2. Internal exceptions

All exceptions that are not to be presented to end users are considered to be internal exceptions. Internal exceptions crash the application and are always considered to be bugs. The application framework generates crash reports for internal errors, so that end users can easily report them. Also, stack traces are not shown on the console. They are however present in the crash report, along with among others information about the system, the Java version used, the application that crashed (name, version, etc), and if the OSGi framework is running, the available plug-ins etc.

6.4.3. Chained exceptions

Java supports the concept of *chained exceptions*. The end-user exceptions of the application framework support this as well. If an uncaught end-user exception needs to be presented to the end user, the message of the exception is printed to the console, prefixed with the `ERROR:` text. All the causes of the exception are printed as well, each on a line of their own. Those messages are prefixed with the `CAUSE:` text. For exceptions that provide an end-user readable message, only that message is printed after the `CAUSE:` text. For other exceptions, the simple name of the exception class, enclosed in parentheses, is printed between the `CAUSE:` text and the exception message. All end-user exceptions (the ones inheriting from the `org.eclipse.escet.common.app.framework.exceptions.ApplicationException` class), as well as all other exceptions explicitly designed as such (by implementing the `org.eclipse.escet.common.app.framework.exceptions.EndUserException` interface) are considered to provide readable messages. For other exceptions, it is assumed that they don't. This includes all exceptions provided by Java itself.

6.4.4. Development mode

Developers can enable the development mode option (`DevModeOption` class) to always get stack traces for all internal exceptions (thus for crashes, but not for end-user exceptions), instead of crash reports. For more information, see the option framework section.

The development mode option is ideal for automated tests, where a stack trace on stderr is much more ideal than a crash report.

6.5. Exit codes

Application framework applications can terminate with the following exit codes:

- `0`: Application finished without errors.
- `1`: Application finished after reporting an error to the end user.
- `2`: Application crashed after running out of memory.
- `3`: Application crashed for any reason other than running out of memory.

Note that applications themselves should always return a zero exit code. The other exit codes are generated automatically by the exception framework when applicable.

Any exceptions to these rules should generally be avoided, but otherwise must be clearly documented for end users.

6.6. The I/O framework

To provide a uniform I/O interface, the application framework includes an I/O framework. This framework is sometimes also called the output framework, as it mainly handles output. The main goals of this framework are:

- Provide uniform stdin, stdout, and stderr support for applications running on the command line, or within the Eclipse IDE.
- Provide a general framework for output, based on output components that can be registered and unregistered.

6.6.1. Output components

The I/O framework works with output components. All output that the application generates, is given to the output components. Each output component can decide for itself what to do with that output. All applications include at least a `StreamOutputComponent`, that redirects stream output to the console. For stand-alone applications, this means redirection to stdout and stderr. For application running within the Eclipse IDE, this means redirection to a *Console* view.

Applications that only need to provide error, warning, normal, and debug textual output, the default output component interface (`IOutputComponent`) suffices. Applications that want to provide additional (typed) output, should create a derived interface that inherits from `IOutputComponent`, and extends the interface with additional callback methods. For an example of this, see the `org.eclipse.escet.cif.simulator.output.SimulatorOutputComponent` interface.

The `OutputComponentBase` class can be used as a base class for output components. It implements the full `IOutputComponent` interface, but does nothing with the output that is generated by the application. Derived classes can easily override one or more methods to process output.

6.6.2. Output provider

Each instance of an application has its own output provider. The output provider keeps track of the output components that are registered, and allows sending of output to the output components through static methods.

If an application uses the default `IOutputComponent` as its output interface, an instance of `OutputProvider<IOutputComponent>` can be used. This will suffice for most applications. If an extended output component interface is defined, the `OutputProvider` class should be extended to provide additional static methods. For an example of this, see the `org.eclipse.escet.cif.simulator.output.SimulatorOutputComponent` class.

For details on how and where to create an instance of the output provider for an application, see the section on how to implement your own application.

6.6.3. Stdout and stderr

Command line applications generally write output to stdout and/or stderr streams. For applications running within the Eclipse IDE, those streams are connected to the Eclipse IDE as a whole, and not to the applications running within Eclipse. The I/O framework solves this issue, by providing a uniform I/O interface.

The `org.eclipse.escet.common.app.framework.output.OutputProvider<T>` class provides several static methods that can be used to generate output. Several forms of output are supported by default:

- Error output is automatically generated by the exception framework, for uncaught exceptions. It is however possible to manually generate error output, by using the `OutputProvider.err` method. This could for instance be useful if multiple error messages are to be outputted.

- Warning output can be generated by applications, by using the `OutputProvider.warn` method. The application framework counts the number of warnings generated by an application, and the count can be retrieved using the `OutputProvider.getWarningCount` method.
- Normal output can be generated by applications, by using the `OutputProvider.out` method. To support structured output, the I/O frame maintains an indentation level, which can be increased and decreased one level at a time.
- Debug output can be generated by applications, by using the `OutputProvider.dbg` method. To support structured output, the I/O frame maintains an indentation level, which can be increased and decreased one level at a time.

One of the default options of the application framework is the output mode option (`OutputModeOption` class). It can be used to control what output gets forwarded to the output components. For performance reasons, it may be useful to query whether certain output gets forwarded. The `OutputProvider` class provides the `dowarn`, `doout`, and `dodbg` methods for this.

It should now be clear that application should never access `System.out` and `System.err` directly. Instead, they should use the output provider.

6.6.4. Stdin

There is no equivalent to the `OutputProvider` for stdin. Instead, use the `AppEnv.getStreams()` method to obtain the streams for the current application. The `AppEnv.getStreams().IN` streams can be used to read data from the stdin stream associated with the current application.

6.6.5. Buffering and flushing

The I/O framework buffers all input and output streams by default, and also automatically performs line based flushing for output and error streams.

6.7. The option framework

Applications often have settings, and they are generally passed as command line arguments. GUI applications however, often use a dialog to configure the options instead. To allow applications within the application framework to work in both scenarios, the application framework provides the option framework.

6.7.1. The option class

All options of applications that use the application framework, should be specified as application framework options. Each option is a derived class of the `org.eclipse.escet.common.app.framework.options.Option<T>` class. The generic type parameter `<T>` indicates that options are strongly typed with respect to their values.

6.7.2. Command line options and the option dialog

The option framework requires all options to work via the command line, but options can also support the option dialog. It is recommended for all options to support the option dialog. The option framework process options as follows:

- All registered options are first initialized to their default values.
- The pre-options hook for the application is fired.
- The command line options are parsed.
- If the command line options enabled the option dialog option (a standard application framework option that controls whether the option dialog is to be shown), the option dialog is shown. The option values as processed so far, are shown to the user in this dialog. The user can modify the options via the dialog and choose **[OK]** to continue.
    - If the user chose **[Cancel]** in the option dialog, terminate the application.
    - All registered options are reset to their default values. This also clears the remaining arguments option, if any.
    - The options set in the dialog are parsed. This overwrites the values of all options.
- The post-processing hook is fired for all options that have it.
- All option values are checked (validated).
- The post-options hook for the application is fired.

6.7.3. Option categories

Options can be ordered into categories. Categories can be combined into a hierarchical structure. This allows the option dialog to show options per category, and allows the command line help message to show command line option help per category. In both cases, this adds structure to the possibly large amount of options, and makes it easier for end users to find the option they are looking for.

6.7.4. Instantiating options

For every option, there may be at most one instance. Therefore, never use the constructors of options directly. Instead use the following:

```
Options.getInstance(MyOption.class)
```

to get an instance of an option.

6.7.5. Getting and setting option values

Applications don’t have access to the command line arguments. The option framework automatically process the command line arguments based on the options registered for the application. Applications always retrieve the values of options through static methods defined in the option classes.

Options are usually set via command line arguments, or via the option dialog. It is however also possible to set option values at run-time:

```
Options.set(MyOption.class, <value>);
```

6.7.6. Option processing order

If possible, options should not depend on the order in which they are parsed. If the value of one option depends on the value of another option, use the post-processing hook to achieve consistency.

6.7.7. Command line option syntax

All options have a long form (`--option`), optionally with a value (`--option=VALUE`). They can also have short form (`-o`), optionally with a value (`-oVALUE` or `-o VALUE`). All arguments that do not start with a dash symbol (`-`) are considered to be the 'remaining arguments'. It is possible to register one option that processes those remaining arguments. Such special options have `*` as long option name.

6.7.8. Implementing your own options

Simply derive from the `Option` class, and study its API to implement your own options. You can also look at existing options for best practices. Furthermore, the option framework provides several options that can be used in applications:

- `BooleanOption`: convenience base class for boolean options.
- `FilesOption`: multiple remaining arguments input file paths option.
- `InputFileOption`: single remaining argument input file path option.
- `OutputFileOption`: output file path option (`--output` / `-o`).

6.7.9. Standard options

The application framework provides several options that must be registered for every application:

- `DevModeOption`: option to enable/disable development mode. Developers can enable this option to get stack traces in case of internal exceptions, instead of crash reports. See also the chapter on the exception framework.

- `HelpOption`: option to show the application help text at the console.
- `LicenseOption`: option to print the license text of the application at the console, and terminate the application.
- `OptionDialogOption`: option to show the option dialog.
- `OutputModeOption`: option to control the amount of output produced by the application. See also the I/O framework section.
- `GuiOption`: option to disable the GUI (enabled headless execution mode). See also the section on GUIs and SWT.

See also the section on how to implement your own application.

6.8. The compiler framework

For performance reasons, it can be better to generate and compile code at runtime, than to use an interpreter. The Java compiler supports this. However, in an Eclipse/OSGi environment, some additional effort is required to make it all work. The application framework contains a compiler framework in the `org.eclipse.escet.common.app.framework.javacompiler` package. It supports in-memory compilation of in-memory code, with full transparent OSGi support. That is, whether used from inside the Eclipse IDE, or from a stand-alone application, the compiler framework takes care of the details. The framework supports various representations of in-memory code, and can be extended with additional representations.

The compiler framework requires the use of a Java Development Kit (JDK). A Java Runtime Environment (JRE) is not sufficient.

6.9. How to implement your own application

This section more or less explains step by step how to implement your own application, by using the application framework.

- Decide whether it is enough to use the `IOutputComponent` interface, or that you need more. See also the I/O framework section.
- Create a new class, deriving from `Application`.
- Add a `main` method to your application class. For instance:

```
/**
 * Application main method.
 *
 * @param args The command line arguments supplied to the application.
 */
public static void main(String[] args) {
    MyApp app = new MyApp();
    app.run(args);
}
```

This allows for standalone execution.

- Add constructors as needed. You'll probably want to implement some or most of the constructors provided by the `Application` class. In order to support standalone execution, the following constructor is required:

```
/** Constructor for the {@link MyApp} class. */
public MyApp() {
    // Nothing to do here.
}
```

In order to support the ToolDef `app` tool, which can be used to run application framework applications from ToolDef scripts, the following constructor is required:

```
/**
 * Constructor for the {@link MyApp} class.
 *
 * @param streams The streams to use for input, output, and error streams.
 */
public MyApp(AppStreams streams) {
    super(streams);
}
```

This constructor is also required by the `ChildAppStarter` class, to support starting one application framework application from another application framework application.

- Implement the mandatory methods `getAppName` and `getAppDescription`.
- Implement mandatory method `getProvider`. If you use `IOutputComponent`, then you can implement it as follows:

```
return new OutputProvider<>();
```

If you don't use `IOutputComponent`, return a new instance of a derived class of `OutputProvider` that implements the derived interface of `IOutputComponent`.

- If you don't use `IOutputComponent`, override the `getStreamOutputComponent` method, and return a new instance of a derived class of `StreamOutputComponent` that implements the derived interface of `IOutputComponent`. Such a class usually ignores all other output, and thus behaves exactly as `StreamOutputComponent`, but implements the full output interface of the application.
- Implement mandatory method `getAllOptions`. You'll need to return an option category that wraps the actual option categories of the application. Use the `getGeneralOptionCategory` to obtain the default application options category, which must always be the first category of options for your application. An example of an implementation of this method:

```
@Override
@SuppressWarnings("rawtypes")
protected OptionCategory getAllOptions() {
    OptionCategory generalOpts = getGeneralOptionCategory();

    OptionCategory debugOpts =
        new OptionCategory("Debug", "Debugging options.", list(),
                           list(Options.getInstance(DebugOption.class)));

    OptionCategory options =
        new OptionCategory("My Application Options",
                           "All options for My Application.",
                           list(generalOpts, debugOpts), list());

    return options;
}
```

- Implement mandatory method `runInternal` with the actual application code.
- Override optional method `getHelpMessageNotes` if applicable.
- Override optional methods `preOptions` and `postOptions` if applicable.
- Override optional method `getAppVersion` if applicable.

6.9.1. The runInternal method

Some things to consider when implementing the `runInternal` method:

- If you want to support stand-alone execution, register all Eclipse Modeling Framework (EMF) metamodels with the EMF metamodel registry. Also register any parsers, constraints, etc. For instance:

```
if (!Platform.isRunning()) {
    // Register languages and parsers for stand-alone execution.
    LanguageRegistry.register...(...)
}
```

- The start of the `runInternal` method is a good place to add output components, as all options have been fully processed at this point. Output components can be registered by using the application's output provider (though static methods).

- The code in this method and all code directly or indirectly executed by this method, should regularly call the `AppEnv.isTerminationRequested` method, to find out whether the application should be terminated.
- For the return code of this method, always use value zero, to indicate successful termination. Other exit codes are automatically generated by the exception framework, if applicable. See also the exit codes section.

6.10. Application registration

Applications that use the application framework maintain their own data. This includes options, output components (via an output provider), streams, etc. Only a single application can be registered for each thread. Only once the application terminates and automatically unregisters itself, can a new application register itself in that thread. To run multiple applications in parallel, simply run them on different threads.

6.10.1. Multi-threaded applications

All data stored for the application is wrapped in the `AppEnvData` class, and stored by the `AppEnv` class, on a per-thread basis. If your application uses multiple threads, you need to register each thread with the application framework. Use the `AppEnv.registerThread` method for this. This method requires the current application environment data as parameter, which may be obtained by using the `AppEnv.getData` method. To avoid managed memory leaks, always unregister threads once they are no longer used, by using the `AppEnv.unregisterThread` method.

6.10.2. Unit tests

If unit tests use methods that depend on the application being registered, then the unit test will need to register an application. Examples of method using the application framework are methods that use options, or produce output via the application framework. Especially for unit tests, the `AppEnv.registerSimple` method can be used to register a dummy application. This method uses a default application environment, without an actual application, registers a default stream output provider, sets the output mode to errors and warnings only (no normal or debug output), and disables development mode. It can be used in a unit test class as follows:

```
/***/ @BeforeClass
public static void oneTimeSetUp() {
    AppEnv.registerSimple();
}

/***/ @AfterClass
public static void oneTimeTearDown() {
    AppEnv.unregisterApplication();
}
```

If any options are used, they will need to be available as well. For instance, one could add the following to the `oneTimeSetUp` method, or at the start of the actual unit test method:

```
Options.set(SomeOption.class, <value>);
```

6.10.3. Running an application from another application

As noted above, only a single application can be registered for a single thread. To start one application from another application, simply run the second application in a fresh thread. In the new thread, do the following:

- Construct the child application, using a constructor with the `AppStreams` argument, to pass along the streams from the parent application.
- Set the current working directory to the current working directory of the parent application.
- Obtain the Eclipse IDE console (if any) from the parent application, and couple it the child application.
- Run the child application.

After the child application thread finishes, make sure you:

- Restore the coupling between the Eclipse IDE console (if any) and the parent application.
- If the child application finished due to a termination request, request termination for the parent application.
- Decide what to do with the exit code of the child application. If it is non-zero, you'll probably want to terminate the parent application.

To make it easier to follow this approach, the `ChildAppStarter.exec` methods can be used.

6.11. Execution

Application framework applications can be executed in the following ways:

- As plain Java application, from the command line.

  Using the *GUI* option, the application can be executed either with full GUI support, or as headless application.

  The OSGi framework will not be running, and the Eclipse workbench will not be available.

- As application within the Eclipse IDE, with full GUI support.

  The OSGi framework will be running, and the Eclipse workbench will be available.

- As headless Eclipse application.

  Using the *GUI* option, the application can be executed either with full GUI support, or as headless application.

  The OSGi framework will be running, but the Eclipse workbench will not be available.

  The `org.eclipse.escet.common.app.framework.AppEclipseApplication` application can be provided to the `-application` command line argument of Eclipse to start any application framework application. This functionality is implemented by the `org.eclipse.escet.common.app.framework.AppEclipseApplication` which provides a generic implementation of Eclipse's `IApplication` interface that supports execution of any application framework application.

  The following command line arguments are expected:

  - The name of the plug-in (OSGi bundle) that provides the application.
  - The full/absolute name of the Java class that implements the application. Must extend the `Application` class and have a parameterless constructor.
  - The remaining command line arguments are the command line arguments for the application itself.

7. Eclipse ESCET release notes

The release notes for the releases of the Eclipse ESCET tools, as part of the Eclipse ESCET project, are listed below in reverse chronological order.

See also the release notes for the specific tools for more information:

- CIF release notes
- Chi release notes
- ToolDef release notes
- SeText release notes

7.1. Version 0.1

The first release of the Eclipse ESCET project and toolkit. This release is based on the initial contribution by the Eindhoven University of Technology (TU/e).

Most notable changes compared to the last TU/e release:

- A JDK is no longer bundled with the downloads. A JDK must be installed separately and manually. Future releases will again include a JDK.

This release is based on the Eclipse IDE version 2020-06 and supports Java 8.

8. Contact information

Thanks for your interest in Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET) project.

You can interact with the Eclipse ESCET community and its developers in various ways:

Eclipse ESCET forum

If you have any questions regarding the Eclipse ESCET project, any of its tools, or how to use them, feel free to ask them on the project forum.

Issue tracking

If you wish to browse existing issues or report new ones, then see the issue tracking section for more information.

Developer information

If you specifically want to contact the Eclipse ESCET developers concerning development related activities, want to contribute to the Eclipse ESCET project, or want to browse the source code, then see the developer information.

9. Legal

Index