Developing Data-Intensive Cloud
Applications with Iterative Quality
Enhancements

DICE Framework – Initial version

Deliverable 1.5

Deliverable:	D1.5
Title:	DICE Framework – Initial version
Editor(s):	Marc Gil (PRO)
Contributor(s):	Marc Gil (PRO), Ismael Torres (PRO), Christophe Joubert (PRO) Giuliano Casale (IMP), Darren Whigham (Flexi), Matej Artač (XLAB), Diego Pérez (Zar), Vasilis Papanikolaou (ATC), Francesco Marconi (PMI), Eugenio Gianniti(PMI), Marcello M. Bersani (PMI), Daniel Pop (IEAT), Tatiana Ustinova (IMP), Gabriel Iuhasz (IEAT), Chen Li (IMP), Ioan Gragan (IEAT), Damian Andrew Tamburri (PMI), Jose Merseguer (Zar), Danilo Ardagna (PMI)
Reviewers:	Darren Whigham (Flexi), Matteo Rossi (PMI)
Type (R/P/DEC):	-
Version:	1.0
Date:	31-January-2017
Status:	First Version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

DICE partners

ATC:	Athens Technology Centre
FLEXI:	FlexiOPS
IEAT:	Institutul e-Austria Timisoara
IMP:	Imperial College of Science, Technology & Medicine
NETF:	Netfective Technology SA
PMI:	Politecnico di Milano
PRO:	Prodevelop SL
XLAB:	XLAB razvoj programske opreme in svetovanje d.o.o.
ZAR:	Universidad De Zaragoza

The DICE project (February 2015-January 2018) has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644869

Executive summary

This deliverable documents the DICE Framework, which is composed of a set of tools developed to support the DICE methodology. One of these tools is the DICE IDE, which is the front-end of the DICE methodology and plays a pivotal role in integrating the other tools of the DICE framework.

The purpose of this document is to explain the work done towards releasing this framework and to serve as basis of a DICE workflow, which will guide the user in executing the integrated tools that define the DICE Methodology.

Glossary

API	Application Programming Interface
AST	Abstract Syntax Tree
ATL	ATL Transformation Language
BIRT	Business Intelligence and Reporting Tools
CEP	Complex Event Processor
CI	Continuous Integration
CPU	Central Process Unit
CSS	Cascade Style Sheet
DDSM	DICE Deployment Specific Model
DIA	Data-Intensive Application
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DICER	DICE Rollout Tool
Dmon	DICE Monitoring
DPIM	DICE Platform Independent Model
DSL	Domain Specific Language
DTSM	DICE Technology Specific Model
EMF	Eclipse Modelling Framework
EPL	Eclipse Public License
FCO	Flexiant Cloud Orchestrator
GEF	Graphical Editing Framework
GIT	GIT Versioning Control System
GMF	Graphical Modelling Framework
GUI	Graphical User Interface
HDFS	Hadoop Distributed File System
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IDE	Integrated Development Environment
IP	Internet Protocol
JDO	Java Data Objects
JDT	Java Development Tools
LALR	Look Ahead Left-to-Right
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MBD	Model Based Design
MDSD	Model Driven Software Development
MOF	Meta-Object Facility
MTBF	Mean Time Between Failures
MW	MiddleWare
OCL	Object Constraint Language
OLAP	OnLine Analytical Processing
OMG	Object Management Group
OSGi	Open Services Gateway initiative

PDE	Plug-in Development Environment
PNML	Petri Net Markup Language
POM	Project Object Model (MAVEN)
QA	Quality-Assessment
QVT	Meta Object Facility (MOF) 2.0 Query/View/Transformation Standard
QVTO	QVT Operational Mappings language
RCP	Rich Client Platform
SCM	Source Code Management
SQL	Structured Query Language
SVN	Subversion Versioning Control System
SWT	Standard Widget Toolkit
TC	Trace Checking
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
UML	Unified Modelling Language
UML2RDB	Unified Modelling Language to Relational Data Base
URL	Uniform Resource Locator
VCS	Versioning Control System
VM	Virtual Machine
WST	Web Standard Tools
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

Table of Figures

Table of Contents

1. Introduction

1.1. Objectives of this document

The DICE project goes beyond the basic idea of using model-driven development for Big Data applications. The vision is to develop high-quality applications that can be continuously deployed to satisfy requirements in terms of efficiency, reliability and safety. To achieve these goals the DICE platform implements the architecture and integration patterns defined in deliverable D1.4 "Architecture Definition & Integration Plan".

The DICE Framework is composed of a set of tools developed to help the user to apply the DICE workflows defined in DICE Methodology. The framework will guide the user in executing the integrated tools that define the DICE Methodology.

The DICE Framework is a quality framework for developing data-intensive applications that leverage Big Data Technologies hosted in clouds. The framework will help satisfying quality requirements in data-intensive applications by iterative enhancement of their architecture design.

The main objective of this document is to explain how the DICE Framework works, and the principles used to drive its definition. To explain the framework the document describes all tools that compose it, with special attention to the Eclipse-based DICE IDE, which is the pivotal tool of the framework. The DICE IDE integrates all the other tools of the DICE framework and it is the base of the methodology. The DICE IDE offers two ways to integrate tools: "fully integrated" or "externally integrated", as it is explained in detail in section 5.

This document defines how the tools are built (implementation and integration) and how users can use them based on the DICE Methodology.

1.2. Relation to DICE Tools

In this section, we present all DICE tools and their main roles in the DICE architecture.

There is a difference between the Framework, which is the entire set of DICE tools, and the IDE, which refers only to the Eclipse environment. The DICE Framework is composed of several tools: the DICE IDE, the DICE/UML profile, the Deployment Design (DICER) and Deployment service provide the minimal toolkit to create and release a DICE application. To validate the quality of the application the framework includes tools covering a broad range of activities such as simulation, optimization, verification, monitoring, anomaly detection, trace checking, iterative enhancement, quality testing, configuration optimization, fault injection, and repository management. Some of the tools are design-focused, others are runtime-oriented. Finally, some have both design and runtime aspects and are used in several stages of the lifecycle development process (see Figure 1).

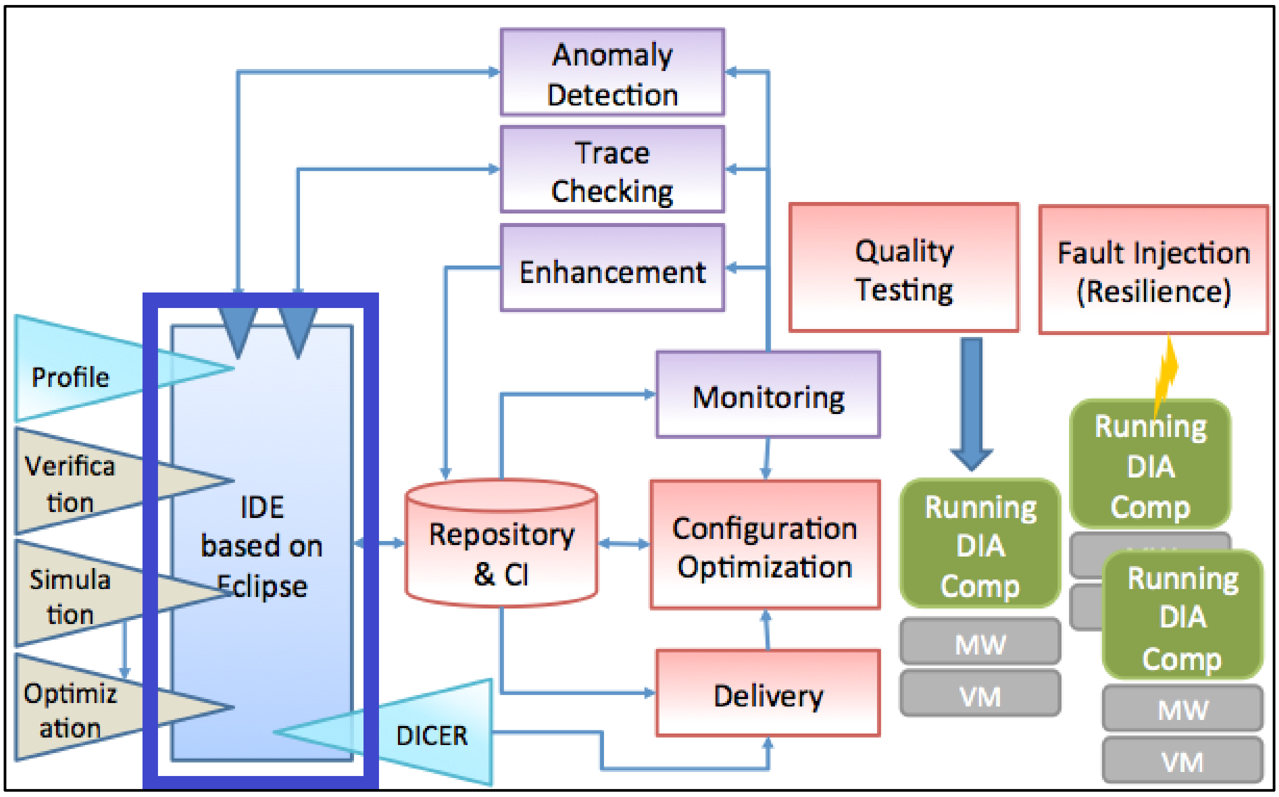


Figure 1:Structure of the DICE Framework components

.

As highlighted in *D1.1 State of the Art Analysis*, there is no IDE for Model Driven Engineering (MDE) on the software market through which a designer can create models to describe data-intensive/Big data applications and their underpinning technology stack. This is the motivation for defining the DICE IDE.

The DICE IDE is intended to integrate most of the tools that belong to the DICE Project. In general, a complex activity of design and prototype validation requires a spectrum of activities ranging from design to operations, where some activities are convenient to do within the IDE, whereas others are not. We therefore provide in this document a rationale for the decision to integrate or not a tool as part of the IDE.

The DICE IDE is based on Eclipse, which is the de-facto standard for the creation of software engineering models based on the MDE approach. DICE customizes the Eclipse IDE with suitable plug-ins that integrate the execution of the different DICE tools, to minimize learning curves and simplify adoption.

1.3. Main objectives of DICE IDE

The main objectives of the DICE IDE are as follows:

- O1: Offer a user-friendly IDE to use the DICE Framework

The IDE is based on Eclipse. Eclipse is a powerful framework that allows one to integrate in an easy way multiple functionalities to reduce the time that is necessary for the user to carry out the application development. In this way, the IDE can serve as the basis for the DICE Framework to interact with all DICE tools.

- O2: Provide an access point for design and prototype delivery

When needed for specific activities, a tool can have an entry point in the IDE. In this way, the developer can use the IDE to create applications in a single environment, without resorting to multiple and diverse frameworks. The IDE environment is particularly suited to orchestrate design and delivery of the application prototypes, and to use interaction patterns such as browser-based dashboards and RESTful APIs to acquire relevant information and data from external tools that operate outside the IDE space.

- O3: Keep the tools always updated

Each tool has its lifecycle. It is possible that a tool is improved and a new version thereof is released after installation of the IDE. The IDE provides update support to keep all tools up-to-date. In addition, the IDE and its components can be updated automatically.

- O4: Integrate most of the framework tools

Most of the tools can be executed using the IDE. Some of them are fully integrated within the IDE, and can be configured within it. Other tools are partially integrated and use external services that can be accessed from the IDE. It is up to each tool to decide how the user interacts with it, for example by providing a user interface to execute it, or by opening an external web browser. Also, the results can be shown within the IDE, thus giving the developer the possibility to use them to execute other tools.

The integration of the tools is an on-going work which is planned to be finished by M30.

- O5: Customize the IDE

As the IDE is based on Eclipse, it brings the possibility to install necessary third party features supported by Eclipse using the Update Site method. This allows the final user to customize the development as he best see fit for their work.

- O6: Allow the user to browse version control repositories

Eclipse adds the support to use SVN or GIT repositories. There exist several components that bring this functionality and allow the user to browse remote files. Such repositories play an important role also in DICE, to manage successive versions of the tools, ensure persistence of the UML models, and allow runtime tools external to the Eclipse environment to access design data (e.g., DPIM/DTSM models).

- 07: Enable advanced UML modelling

Papyrus was chosen as the standard UML editor, as it is fully based on Eclipse and its technologies. In this way, models developed outside the DICE IDE can still be used within the DICE IDE; conversely, users can export their models developed through the DICE IDE to other Papyrus-based IDEs.

1.4. Roadmap and main achievements

This section presents an overview of the progress of the DICE framework and a plan of the activities for the remainder of the project.

In this document, we show the current status (M24) and the plans for the future (M30). A more detailed description of the roadmap from the beginning of the project can be found in **Deliverable 1.4, Section3.**

Integration Plan for M24 (January 2017)

The framework tools integrated with the M24 version of the **IDE (v0.2.0)** are: simulation, verification, monitoring, delivery/deployment, DICE Profiles, Optimization, DICER and Methodology tool.

Integration Plan for M30 (July 2017)

The final goal for M30 is to release **IDE v1.0.0**. This version will include all DICE tools integrated within the IDE, with some exceptions such as the Fault Injection Tool (as mentioned above). Moreover, all interdependencies and relationships among the different tools will be finalised and established.

The tools that will be newly integrated at M30 are: anomaly detection, trace checking, enhancement, fault injection and quality testing tool.

1.5. Document contents

The document is structured as follows:

- Section 2 updated requirements and use case scenarios of the DICE IDE
- Section 3 DICE IDE Components. A description of all the components that are part of the IDE, with an in-depth presentation of those that are relevant to the IDE. Also, the DICE Tools currently integrated are detailed.
- Section 4 Building the IDE. An introduction to the mechanisms underlying the building of the IDE: how it is built in order to obtain an executable application, including the DICE tools, using the Tycho building process by Eclipse.
- Section 5 Integration approaches. Presentation of the two approaches that DICE tools can follow in order to be integrated within the IDE.
- Section 6 Getting started with DICE IDE. An explanation of how to get started with the IDE, including how to download it, how to run it, and how to start working on it.
- Section 7 Conclusions and future plans. Concluding remarks and a list of some future plans concerning the Framework, and in particular which tools will be integrated in the IDE in the future.

A Companion document is added at the end in order to include some useful information about the DICE Framework Tools.

2. Requirements

The requirements and use case scenarios of the DICE Framework, which were first presented in the deliverable "D1.2 Requirements specifications" Requirement Specification – Companion Document.

The outcome of the analysis was a consolidated list of requirements and the list of use cases that define the project's goals that guide the DICE technical activities.

Next, we summarize these requirements for WP1. They will be reviewed in the Conclusions so as to evaluate the technical advances in the scope of WP1.

2.1. WP1 Requirements

ID:	R1.1
Title	Stereotyping of UML diagrams with DICE profile
Priority	Must have
Type	Requirement
Description	Open-source modelling tool with XMI and UML2.X (2.4 or 2.5) support

ID	R1.2
Title	Guides through the DICE methodology
Priority	Must have
Description	An action to open an external website with the guide or document of the DICE Methodology

ID	R1.6
Title	Quality testing tools IDE integration
Priority	Should have
Description	The IDE SHOULD provide the means to configure the QTESTING_TOOLS execution

ID	R1.7
Title	Continuous integration tools IDE integration

Priority	Should have
Description	The CI_TOOLS MUST be integrated with the IDE.

ID:	R1.7.1
Title:	Running tests from IDE without committing to VCS
Priority	Could have
Description	The CI_TOOLS COULD provide an integration with the IDE that enables deployment and execution of tests on the user's local changes without committing the code into the VCS.

ID:	R2IDE.1
Title:	IDE support to the use of profile
Priority	Must have
Type:	Requirement
Description	The DICE IDE MUST support the development of DIA exploiting the DICE profile and following the DICE methodology. This means that it should offer wizards to guide the developer through the steps envisioned in the DICE methodology

ID	R3IDE.1
Title	Metric selection
Priority	Must have
Description	The DICE IDE MUST allow to select the metric to compute from those defined in the DPIM/DTSM DICE annotated UML model. There are efficiency and reliability related metrics

ID	R3IDE.2
Title	Timeout specification
Priority	Should have
Description	The IDE SHOULD allow the user to set a timeout and a maximum amount of memory (2) to be used when running the SIMULATION_TOOLS and the VERIFICATION_TOOLS.

	Then, when the timeout expires or when the memory limit is exceeded, the IDE SHOULDS notify the user of this and gracefully stop the simulation/verification.

ID:	R3IDE.3
Title:	Usability
Priority	Could have
Description	The TRANSFORMATION_TOOLS and SIMULATION_TOOLS MAY follow some usability, ergonomics or accessibility standard such as ISO/TR 16982:2002, ISO 9241, WAI W3C or similar

ID	R3IDE.4
Title	Loading the annotated UML model
Priority	Must have
Description	The DICE IDE MUST include plugins to launch the SIMULATION_TOOLS and VERIFICATION_TOOLS for a DICE UML model that is loaded in the IDE

ID	R3IDE.4.1
Title	Usability of the IDE-VERIFICATION_TOOLS interaction
Priority	Should have
Description	The QA_ENGINEER SHOULD not perceive a difference between the IDE and the VERIFICATION_TOOL; it SHOULD be possible to seamlessly invoke the latter from the former

ID	R3IDE.4.2
Title	Loading of the property to be verified
Priority	Must have
Description	The VERIFICATION_TOOLS MUST be able to handle the verification of the properties to be checked that can be defined through the IDE and the DICE profile

ID	R3IDE.5

Title	Graphical output
Priority	Should have
Description	Whenever needed (for better understanding of the response), the IDE SHOULD be able to take the output generated by the VERIFICATION_TOOLS (i.e., execution traces of the modeled system) and represent it graphically, connecting it to the elements of the modeled system.

ID:	R3IDE.5.1
Title	Graphical output of erroneous behaviors
Priority	Could have
Description	In case the outcome of the verification task is "the property does not hold", the VERIFICATION_TOOLS COULD provide, in addition to the raw execution trace of the system that violates the desired property, an indication of where in the trace lies the problem (i.e., which part of the trace violates the property)

ID	R3IDE.6
Title	Loading a DDSM level model
Priority	Must have
Description	The OPTIMIZATION_TOOLS as part of the IDE MUST provide an interface to load (not design) a DDSM DICE annotated model

ID	R4IDE1
Title	Resource consumption breakdown
Priority	Must have
Description	The DEVELOPER MUST be able to see via the ENHANCEMENT_TOOLS the resource consumption breakdown into its atomic components.

ID	R4IDE2
Title	Bottleneck Identification

Priority	Must have
Description	The ENHANCEMENT_TOOLS MUST indicate which classes of requests represent bottlenecks for the application in a given deployment.

ID	R4IDE3
Title	Model parameter uncertainties
Priority	Could have
Description	The REQ_ENGINEER COULD express uncertainty on some performance/reliability input parameters (e.g., execution times) in the DICE profile by means of a prior distribution or an interval. The ENHANCEMENT_TOOLS COULD take into account these parameters to estimate these parameters from monitoring data.

ID	R4IDE4
Title	Model parameter confidence intervals
Priority	Could have
Description	The ENHANCEMENT_TOOLS COULD return confidence intervals for each inferred parameter of the performance and reliability models.

ID:	R4IDE5
Title	Visualization of analysis results
Priority	Could have
Description	ENHANCEMENT_TOOLS SHOULD be capable of visualizing analysis results

ID	R4IDE6
Title	Safety and privacy properties loading
Priority	Must have
Description	The ANOMALY_TRACE_TOOLS MUST allow the DEVELOPER/ARCHITECT to choose and load the safety and privacy properties from the Model of the application described through the DICE profile

ID	R4IDE7
Title	Feedback from safety and privacy properties monitoring to UML models concerning violated time bounds
Priority	Could have
Description	In the feedback provided by the ANOMALY_TRACE_TOOLS to the DEVELOPER/ARCHITECT, the tools COULD highlight when a timing requirement is violated, and what is the value of the violation

ID	R4IDE8
Title	Relation between ANOMALY_TRACE_TOOLS and IDE
Priority	Should have
Description	It SHOULD be possible to launch the ANOMALY_TRACE_TOOLS from the IDE

3. DICE IDE Components

3.1. Introduction to Eclipse RCP's

The DICE IDE is an application based on Eclipse. While the Eclipse platform is designed to serve as an open platform for tool integration, it is architected so that its components could be used to build any arbitrary client application. The minimal set of plug-ins needed to build a rich client application is collectively known as the Rich Client Platform (**RCP**). Applications other than IDEs can be built using a subset of the platform. These rich applications are still based on a dynamic plug-in model, and the UI is built using the same toolkits and extension points. The layout and function of the workbench is under fine-grained control of the plug-in developer in this case [1].

An Eclipse **application** consists of several Eclipse **components**, it is possible extend the Eclipse IDE via plug-ins (components) [38]. Eclipse applications incorporate runtime features based on OSGi, that allows update/delete/create features to your application using OSGi Bundles (Components).

The minimum piece of software that can be integrated in Eclipse is called a **plug-in**. The Eclipse platform allows the developer to extend Eclipse applications like the Eclipse IDE with additional functionalities via plug-ins. [38]

Eclipse applications use a runtime based on a specification called OSGi. A software component in OSGi is called a **bundle**. An OSGi bundle is also always an Eclipse plug-in. Both terms can be used interchangeably.

The Eclipse IDE is basically an Eclipse RCP application to support development activities. Even core functionalities of the Eclipse IDE are provided via a plug-in. For example, both the Java and C development tools are contributed as a set of plug-ins. Therefore, the Java or C development capabilities are available only if these plug-ins are present.

The Eclipse IDE functionality is based on the concept of **extensions** and **extension points**. For example, the Java Development Tools provide an extension point to register new code templates for the Java editor.

Via additional plug-ins you can contribute to an existing functionality, for example new menu entries, new toolbar entries or provide completely new functionality. But you can also create completely new programming environments.

The minimal required plug-ins to create and run a minimal Eclipse RCP application (with UI) are the two plug-ins "org.eclipse.core.runtime" and "org.eclipse.ui". Based on these components an Eclipse RCP application must define the following elements:

- Main program - a RCP main application class implementing the interface "IApplication". This class can be viewed as the equivalent of the main method for standard Java application. Eclipse expects that the application class is defined via the extension point "org.eclipse.core.runtime.application".
- A Perspective - it defines the layout of your application. Must be declared via the extension point "org.eclipse.ui.perspective".
- Workbench Advisor- invisible technical component which controls the appearance of the application (menus, toolbars, perspectives, etc.).

Let us now introduce all the components that are part of the DICE IDE.

3.2. The Eclipse required components

The Figure 2 is a Component Diagram made with Papyrus that illustrates the components that take part in the DICE IDE and their dependencies. The components with brown colour are DICE tools (see section 3.3), and the components in blue colour are standard eclipse components (explained in this section).

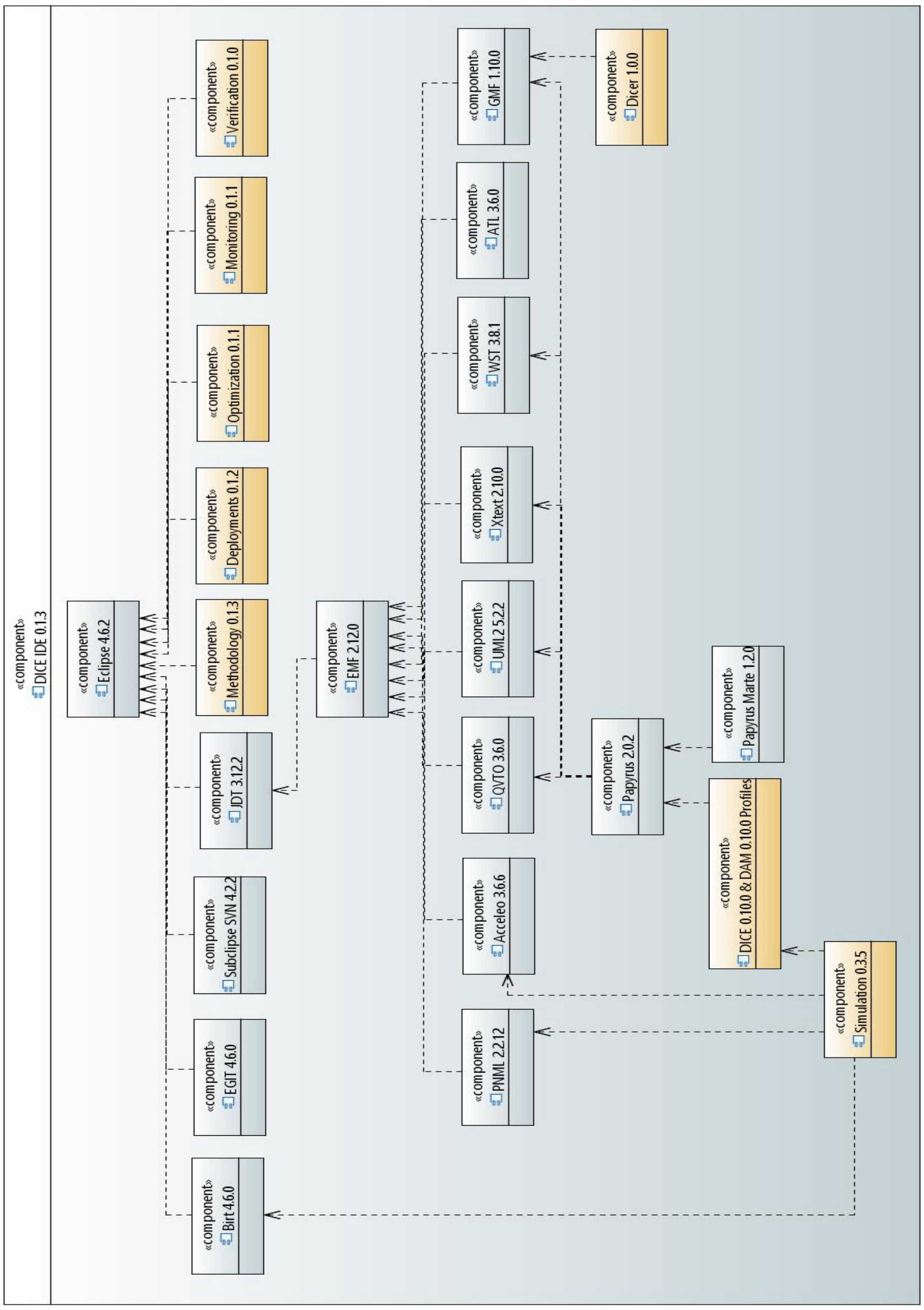

«component» DICE IDE 0.1.3
«component» Eclipse 4.6.2
«component» Verification 0.1.0
«component» Monitoring 0.1.1
«component» Optimization 0.1.1
«component» Deployments 0.1.2
«component» Methodology 0.1.3
«component» JDT 3.12.2
«component» Subclipse SVN 4.2.2
«component» EGIT 4.6.0
«component» Birt 4.6.0
«component» EMF 2.12.0
«component» GMF 1.10.0
«component» ATL 3.6.0
«component» WST 3.8.1
«component» Xtext 2.10.0
«component» UML2 5.2.2
«component» QVTO 3.6.0
«component» Acceleo 3.6.6
«component» PNML 2.2.12
«component» Dicer 1.0.0
«component» Papyrus 2.0.2
«component» Papyrus Marte 1.2.0
«component» DICE 0.10.0 & DAM 0.10.0 Profiles
«component» Simulation 0.3.5

Figure 2:Component diagram of the DICE IDE.

The following are the basic components for running the IDE and Tools.

3.2.1. Eclipse Base

The Eclipse Base component includes all minimum Eclipse plug-ins necessary to launch an Eclipse-based application. All of the contained plug-ins depend on the Eclipse version the DICE IDE is based on. When a new version of Eclipse is released, commonly these plug-ins need to be updated.

These plug-ins can be grouped as follows.

3.2.1.1. Eclipse Core

The core component provides basic platform infrastructure that does not involve any UI.

All the contained plug-ins are extremely generic -- each provides a basic set of services, API and extension points for managing and interacting with those services. The contents of resources are never examined in any domain-specific way and the Core can run equally well with and without a UI. [2]

3.2.1.2. Orbit (Apache Software Foundation, Google)

The Orbit project provides a repository of bundled versions of third party libraries that are approved for use in one or more Eclipse projects. [3]

3.2.1.3. Eclipse e4

e4 is an incubator project for community exploration of future technologies for the Eclipse Platform.

Originally this project was used for the development of the underlying engine of the Eclipse 4 platform, which concluded with the release of Eclipse 4.

After this the project incubated the new E4 Rich Client Tooling, this is used to visually create the Application Model. [4]

3.2.1.4. Eclipse Equinox

From a code point of view, Equinox is an implementation of the OSGi core framework specification, a set of bundles that implement various optional OSGi services and other infrastructure for running OSGi-based systems. The Equinox OSGi core framework implementation is used as the reference implementation and as such it implements all the required features of the latest OSGi core framework specification.

More generally, the goal of the Equinox project is to be a first-class OSGi community and to foster the vision of Eclipse as a landscape of bundles. As part of this, it is responsible for developing and delivering the OSGi framework implementation used for all of Eclipse. [5]

3.2.1.5. GEF

The Eclipse Graphical Editing Framework (GEF) provides Eclipse-integrated end-user tools in terms of a Graphviz authoring and a word cloud rendering environment, as well as framework components to create rich graphical Java client applications, Eclipse-integrated or standalone. [6]

3.2.1.6. PDE

The Plug-in Development Environment (PDE) provides tools to create, develop, test, debug, build and deploy Eclipse plug-ins, fragments, features, update sites and RCP products.

PDE also provides comprehensive OSGi tooling, which makes it an ideal environment for component programming, not just Eclipse plug-in development. [7]

3.2.1.7. SWT

The Standard Widget Toolkit (SWT) is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented. [8]

3.2.2. Eclipse JDT

The Java Development Tools (JDT) project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself. [9]

3.2.3. Eclipse EMF

The Eclipse Modelling Framework (EMF) project is a modelling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor.

EMF (core) is a common standard for data models, on which many technologies and frameworks are based. This includes server solutions, persistence frameworks, UI frameworks and support for transformations. [10]

3.2.4. Acceleo

Acceleo 3 is a code generator implementing of the OMG's Model-to-text specification. It supports the developer with most of the features that can be expected from a top-quality code generator IDE: simple syntax, efficient code generation, advanced tooling, features on par with the JDT. Acceleo helps the developer to handle the lifecycle of its code generators. Thanks to a prototype-based approach, you can quickly and easily create your first generator from the source code of an existing prototype, then with all the features of the Acceleo tooling like the refactoring tools you will easily improve your generator to realize a full fledged code generator. Finally, Acceleo can also help you maintain your generator with its debugger, its profiler and its traceability. [11]

3.2.5. QVTO

QVT Operational component is an implementation of the Operational Mappings Language defined by Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT). In the long term, it aims to provide a complete implementation of the operational part of the standard.

A high-level overview of the QVT Operational language is available as presentations “Model Transformation with Operational QVT” and “The Art of Model Transformation with Operational QVT”. [12]

3.2.6. OCL

The Eclipse OCL Project provides an implementation of the Object Constraint Language (OCL) OMG standard for EMF-based models.

The Core OCL component provides the following capabilities to support OCL integration:

- It defines APIs for parsing and evaluating OCL constraints and queries on EMF models.
- It defines Ecore and UML implementations of the OCL abstract syntax model, including support for serialization of parsed OCL expressions.
- It provides a Visitor API for analysing/transforming the AST model of OCL expressions.
- It provides an extensibility API for clients to customize the parsing and evaluation environments used by the parser.

The Core OCL parser is generated by the LALR Parser Generator, a SourceForge project, licensed under the EPL v1.0. [13]

3.2.7. UML2

UML2 is an EMF-based implementation of the Unified Modelling Language (UML) 2.x OMG metamodel for the Eclipse platform. [14]

The objectives of the UML2 component are to provide:

- a useable implementation of the UML metamodel to support the development of modelling tools
- a common XMI schema to facilitate interchange of semantic models
- test cases as a means of validating the specification
- validation rules as a means of defining and enforcing levels of compliance.

3.2.8. GMF

The Eclipse Graphical Modelling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. [15]

3.2.9. Xtext

Xtext is a framework for development of programming languages and domain-specific languages. With Xtext you define your language using a powerful grammar language. As a result, you get a full infrastructure, including parser, linker, typechecker, compiler as well as editing support for Eclipse, IntelliJ IDEA and your favourite web browser. [16]

3.2.10. WST

The Eclipse Web Tools Platform (WTP) project extends the Eclipse platform with tools for developing Web and Java EE applications. It includes source and graphical editors for a variety of languages, wizards and built-in applications to simplify development, and tools and APIs to support deploying, running, and testing apps. [17]

3.2.11. ATL

ATL (ATL Transformation Language) is a model transformation language and toolkit. In the field of Model-Driven Engineering (MDE), ATL provides ways to produce a set of target models from a set of source models.

Developed on top of the Eclipse platform, the ATL Integrated Environment (IDE) provides several standard development tools (syntax highlighting, debugger, etc.) that aim to ease the development of ATL transformations. [18]

3.2.12. Papyrus & Papyrus Marte

Papyrus is an Open Source UML2 tool based on Eclipse and licensed under the EPL. It has been developed by the Laboratory of Model Driven Engineering for Embedded Systems (LISE) which is a part of the French Alternative Energies and Atomic Energy Commission (CEA-List).

Papyrus can either be used as a standalone tool or as an Eclipse plugin. It provides support for Domain Specific Languages and SysML. Papyrus is designed to be easily extensible as it is based on the principle of UML Profiles. [19]

MARTE is the OMG Modelling and Analysis of Real-Time Embedded Systems profile. [20]

Papyrus provides an integrated, user-consumable environment for editing any kind of EMF model and particularly supporting UML and related modelling languages such as SysML and MARTE. Papyrus provides diagram editors for EMF-based modelling languages (amongst which UML2 and SysML) and the glue required for integrating these editors (GMF-based or not) with other MBD and MDSD tools. It also offers a very advanced support for UML profiles that enables users to define editors for DSLs based on the UML 2 standard and its extension mechanisms. [21]

3.2.13. Subclipse SVN

Subclipse is an Eclipse Team Provider plugin for Apache Subversion. [22]

3.2.14. EGIT

EGit is an Eclipse Team provider for the Git version control system. Git is a distributed SCM, which means every developer has a full copy of all history of every revision of the code, making queries against the history very fast and versatile.

The EGit project is implementing Eclipse tooling on top of the JGit Java implementation of Git. [23]

3.2.15. Birt

The Business Intelligence and Reporting Tools (BIRT) Project is an open source software project that provides reporting and business intelligence capabilities for rich client and web applications, especially those based on Java and Java EE. BIRT is a top-level software project within the Eclipse Foundation, an independent not-for-profit consortium of software industry vendors and an open source community.

The project's stated goals are to address a wide range of reporting needs within a typical application, ranging from operational or enterprise reporting to multi-dimensional online analytical processing (OLAP). Initially, the project has focused on and delivered capabilities that allow application developers to easily design and integrate reports into applications.

BIRT has two main components: a visual report designer within the Eclipse IDE for creating BIRT Reports, and a runtime component for generating reports that can be deployed to any Java environment. The BIRT project also includes a charting engine that is both fully integrated into the report designer and can be used standalone to integrate charts into an application.

BIRT Report designs are persisted as XML and can access a number of different data sources including JDO data stores, JFire Scripting Objects, POJOs, SQL databases, Web Services and XML. [36]

3.2.16. PNML

PNML Framework is a free and open-source prototype implementation of ISO/IEC-15909, International Standard on Petri Nets.

PNML Framework supports Part 2 of the Standard, which defines the Petri Net Markup Language, the XML-based exchange format for Petri nets. Part 1 provides formal definitions of Petri nets, for which Part 2 defines the abstract syntax and exchange format (PNML).

The primary purpose of PNML is to put into practice the interoperability among Petri nets tools. Thanks to this standardized transfer format, tools should be able to exchange Petri nets models, according to ISO/IEC 15909-2 specifications. [24]

3.3. Tool-related components

Most of the components introduced in section 2.2 are the minimum needed dependencies that all DICE Tools have in order to be integrated within the IDE. This section explains the dependencies for each DICE tool.

3.3.1. DICE Profiles

This component contains the UML profiles for the DICE project. Users can use this component when creating their UML models, by applying the profiles provided by the component. More information about this component can be found in the corresponding GitHub project page. [25]

3.3.2. Simulation Tool

This component contains the plug-ins that are necessary to integrate the Simulation Tool. Users can use this component through the integrated launcher that is available on the launching configurations of the IDE. More information about this component can be found in the corresponding GitHub project page. [26]

3.3.3. Deployment Tool

This component contains the plug-ins integrating the Deployment Tool. This tool is integrated externally from the IDE. That means that the real tool is not integrated within the IDE, but in an external web server. So, users can use this component via a menu action from the IDE that will open an external web server where the tool is running. More information about this component can be found in the corresponding GitHub project page. [27]

3.3.4. Optimization Tool

This component includes the plug-ins for the integration of the Optimization Tool. This tool is integrated externally from the IDE. This means that the real tool is not integrated within the IDE, but in an external web server. So, users can use this component via a menu action from the IDE that will open an external web server where the tool is deployed. The next release at M18 will be integrated at least for the front-end functionalities and will rely on an integrated launcher, available on the launching configurations of the IDE.

3.3.5. Monitoring Tool

This component contains plug-ins used to integrate the Monitoring Tool. This tool is integrated externally from the IDE. That is, the real tool is fully accessible within the IDE, but it effectively resides outside the Eclipse environment and it is accessible through the browser within the IDE through an external service or web service. So, users can use this component via a menu action from the IDE that will open an external web server where the tool is deployed.

3.3.6. Verification Tool

This component contains the plug-ins integrating the Verification Tool. Users can use this component through the integrated launcher that is available on the launching configurations of the IDE. More information about this component can be found in the corresponding GitHub project page. [28]

3.3.7. DICER Tool

This component includes the plug-ins for the integration of the DICER Tool. Users can use this component through the integrated launcher that is available on the launching configurations of the IDE. More information about this component can be found in the corresponding GitHub project page. [39]

4. Building the IDE

As the IDE is based on Eclipse, there exists a recommended RCP building process that was introduced in the latest versions of the platform. This building process is named Tycho [29] and it is based on Maven.

Tycho is focused on a Maven-centric, manifest-first approach to building Eclipse plug-ins, features, update sites, RCP applications and OSGi bundles. Tycho is a set of Maven plugins and extensions for building Eclipse plugins and OSGi bundles with Maven. Eclipse plugins and OSGi bundles have their own metadata for expressing dependencies, source folder locations, etc. that are normally found in a Maven POM. Tycho uses native metadata for Eclipse plugins and OSGi bundles and uses the POM to configure and drive the build. Tycho supports bundles, fragments, features, update site projects and RCP applications. Tycho also knows how to run JUnit test plugins using the OSGi runtime and there is also support for sharing build results using Maven artefact repositories. Tycho plugins introduce new packaging types and the corresponding lifecycle bindings that allow Maven to use OSGi and Eclipse metadata during a Maven build.

OSGi rules are used to resolve project dependencies and package visibility restrictions are honoured by the OSGi-aware JDT-based compiler plugin. Tycho uses OSGi metadata and OSGi rules to calculate project dependencies dynamically and injects them into the Maven project model at build time. Tycho supports all attributes supported by the Eclipse OSGi resolver (Require-Bundle, Import-Package, Eclipse-GenericRequire, etc). Tycho uses proper class path access rules during compilation. It supports all project types supported by PDE and it uses PDE/JDT project metadata where possible. One important design goal in Tycho is to make sure there is no duplication of metadata between POM and OSGi metadata.

4.1. Project folder structure

The source code of the IDE Project and the other tools included in the DICE framework are available for download on GitHub [30].

In order to make Tycho work correctly, and to avoid errors during the building process, there are some best practices [31,32] that should be followed when defining the correct project folder structure. These best practises have been used when developing the DICE IDE.

To have some separation of the plug-ins a good practice is to have separate folders on the file system for them.

The proposed structure is detailed within Figure 3:

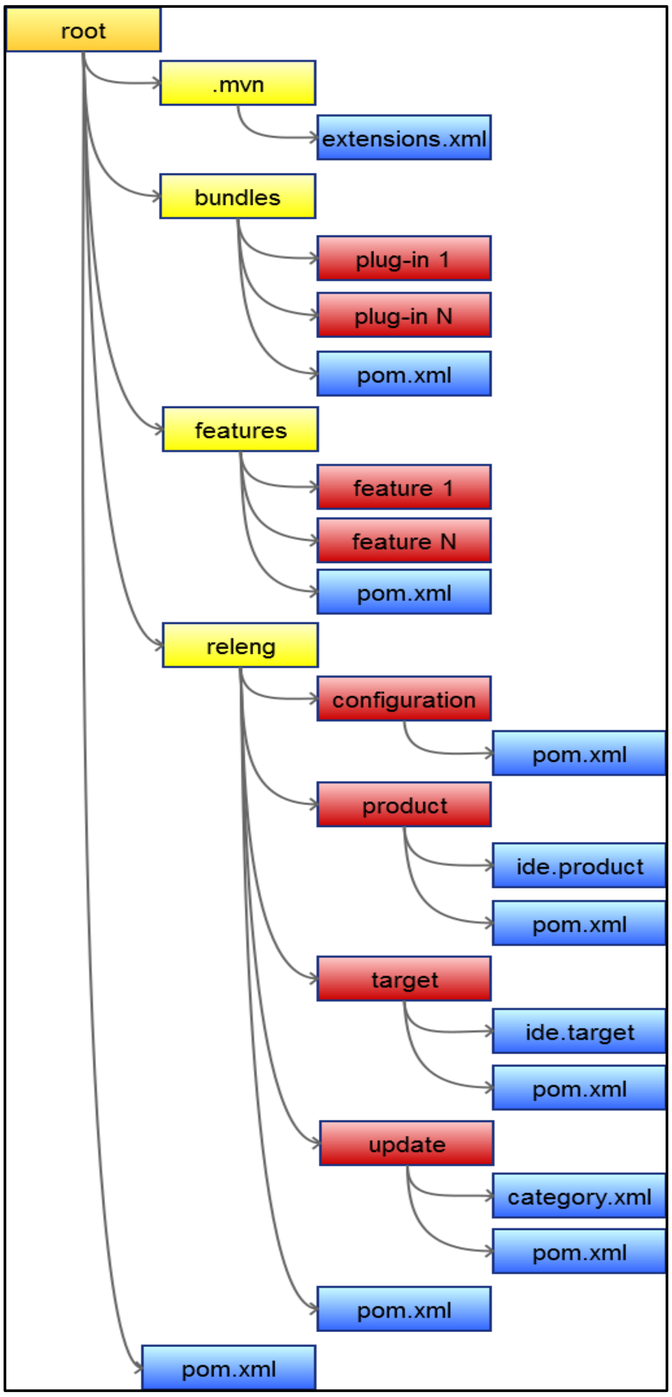


Figure 3: Structure of an Eclipse Application

In a Tycho build it is good practice to create a configuration pom file. This file includes the Tycho plug-ins and the definition of the build environment. The pom file describes the general configuration. The folder, which contains this build file, is typically called releng, which stands for "Release Engineering".

4.1.1. A root container

This root project contains all Eclipse components and the main pom file.

Although this *pom.xml* file resides in the root the folder, it is not the parent pom of the application. This pom only contains the modules related to plug-ins, features and releng (release engineering).

```
...
	<artifactId>org.dice.root</artifactId>
	<name>DICE IDE - Root</name>
	<packaging>pom</packaging>

	<parent>
		<groupId>org.dice</groupId>
		<artifactId>org.dice.configuration</artifactId>
		<version>${dice.version}</version>
		<relativePath>./releng/org.dice.configuration</relativePath>
	</parent>

	<modules>
		<module>bundles</module>
		<module>features</module>
		<module>releng</module>
	</modules>
...
```

code snippet 1: pom.xml

The parent pom is placed on a submodule of releng module named *configuration*, that contains all necessary information for the building process: version, properties, encoding, repositories, maven building plugins configuration, etc.

4.1.1.1. Configure pomless build (.mvn folder)

A pomless build approach is used to avoid declaring a *pom.xml* file on each plug-in and feature component. Instead, Maven is able to extract and complete the needed information from the *plugin.xml* and *feature.xml* file of the components. For instance, the name of each component is extracted from the ID of the plug-in or feature. The same for the version number.

Nevertheless, using a pomless approach does not prevent the user from defining a *pom.xml* for a concrete plug-in or feature to configure the build of this component.

In order to use the pomless Tycho build approach, it is necessary to indicate it in the file placed in the **.mvn/extensions.xml** file:

```
<extensions>
	<extension>
		<groupId>org.eclipse.tycho.extras</groupId>
		<artifactId>tycho-pomless</artifactId>
		<version>0.26.0</version>
	</extension>
</extensions>
```

code snippet 2: extensions.xml

4.1.1.2. *Plug-ins (bundles folder)*

This folder contains all the developed Eclipse plug-ins (OSGi bundles).

As the DICE IDE uses a pomless approach, it is not necessary to define a *pom.xml* file within each plug-in. All the necessary information is retrieved from the plugin.xml files and from the parent poms.

The *pom.xml* file placed in this *bundles* folder should define all the sub modules (plug-ins) in order to build all of them at building time.

```
...
        <artifactId>org.dice.bundles</artifactId>
        <name>DICE IDE - Bundles</name>
        <packaging>pom</packaging>

        <parent>
                <groupId>org.dice</groupId>
                <artifactId>org.dice.root</artifactId>
                <version>${dice.version}</version>
        </parent>

        <modules>
                <module>org.dice.rcp</module>
                <module>org.dice.ui</module>
                <module>org.dice.methodology</module>
                <module>org.dice.monitoring</module>
                <module>org.dice.optimization</module>
                <module>org.dice.simulation</module>
                <module>org.dice.verification</module>
        </modules>
...
```

code snippet 3 pom.xml (bundle)

4.1.1.2.1. RCP plug-in

There exists an important plug-in named RCP that contains some necessary information for each Eclipse application. It contains information such as

- name of the RCP
- CSS theme
- preferences values
- splash image
- welcome page
- icons.

All this information is defined in the *plugin.xml* file of the plug-in.

In this case, there exists a pom.xml file because it is needed to include some files at building time.

```
...
        <artifactId>org.dice.rcp</artifactId>
        <version>${dice.version}-SNAPSHOT</version>
        <packaging>eclipse-plugin</packaging>

        <parent>
                <groupId>org.dice</groupId>
                <artifactId>org.dice.bundles</artifactId>
                <version>${dice.version}</version>
        </parent>

        <build>
                <resources>
                        <resource>
                                <directory>.</directory>
                                <filtering>true</filtering>
                                <includes>
                                        <include>about.mappings</include>
                                </includes>
                        </resource>
                </resources>
        </build>
...
```

code snippet 4: plugin.xml

4.1.1.3. *Features (features folder)*

This folder contains all the developed Eclipse features.

As the plug-ins folder, it is not required to define any *pom.xml* file in these sub modules.

The *pom.xml* file placed in this folder should define all the sub modules (features) in order to build all of them at building time.

```
...
        <artifactId>org.dice.features</artifactId>
        <name>DICE IDE - Features</name>
        <packaging>pom</packaging>
        <parent>
                <groupId>org.dice</groupId>
                <artifactId>org.dice.root</artifactId>
                <version>${dice.version}</version>
        </parent>
        <modules>
                <module>org.dice.features.base</module>
                <module>org.dice.features.jdt</module>
                <module>org.dice.features.subversion</module>
                <module>org.dice.features.git</module>
                <module>org.dice.features.birt</module>
                <module>org.dice.features.emf</module>
                <module>org.dice.features.pnml</module>
                <module>org.dice.features.acceleo</module>
                <module>org.dice.features.qvto</module>
                <module>org.dice.features.uml2</module>
                <module>org.dice.features.gmf</module>
                <module>org.dice.features.xtext</module>
                <module>org.dice.features.wst</module>
                <module>org.dice.features.atl</module>
                <module>org.dice.features.papyrus</module>
                <module>org.dice.features.methodology</module>
                <module>org.dice.features.optimization</module>
                <module>org.dice.features.monitoring</module>
                <module>org.dice.features.verification</module>
                <module>org.dice.features.profiles</module>
                <module>org.dice.features.simulation</module>
                <module>org.dice.features.dicer</module>
        </modules>
...
```

code snippet 5: features pom.xml

4.1.1.4. Release Engineering (releng folder)

This folder contains all the necessary data about the building and release process (used by the Eclipse director component, the Platform Releng [33]).

Along the building process, the Tycho process needs some configuration files that provide information like:

- the Platform Target configuration with all the dependencies to be used when compiling the sources
- configuration for creating the final product and configure it with the specified parameters
- list of update sites created to be used to update the generated product.

The *pom.xml* file of this module only contains the definition of the sub modules.

```
...
        <artifactId>org.dice.releng</artifactId>
        <name>DICE IDE - Releng</name>
        <packaging>pom</packaging>

        <parent>
                <groupId>org.dice</groupId>
                <artifactId>org.dice.root</artifactId>
                <version>${dice.version}</version>
        </parent>

        <modules>
                <module>org.dice.update</module>
                <module>org.dice.product</module>
                <module>org.dice.target</module>
        </modules>
...
```

code snippet 6: releng pom.xml

4.1.1.4.1. Building configuration (configuration folder)

This folder contains a *pom.xml* file that is the parent pom. This pom defines some necessary information for the building process such as:

- Properties: necessary maven properties used across all the maven modules (encoding, version, timestamp, url...).
- Repositories: the external repositories where the building process should look for the sources and dependencies.
- Building process plug-ins: necessary plugins used at building time. Also, the target environments (windows, linux, macOS...) can be defined.

```
...
        <groupId>org.dice</groupId>


        <artifactId>org.dice.configuration</artifactId>
        <version>${dice.version}</version>

        <name>DICE IDE - Configuration</name>
        <packaging>pom</packaging>

        <properties>
                ...
        </properties>
        <repositories>
                <!-- DICE dependencies -->
                <repository>
                        <id>eclipse-updates</id>
                        <layout>p2</layout>
                        <url>${eclipse-updates.url}</url>
                </repository>
                <!-- DICE projects -->
                <repository>
                        <id>deployments</id>
                        <layout>p2</layout>
                        <url>${dice-deployment.url}</url>
                </repository>
                ...
        </repositories>
        <build>
                <plugins>
                        <plugin>
                                <groupId>org.eclipse.tycho</groupId>
                                <artifactId>tycho-maven-plugin</artifactId>
                                <version>${tycho.version}</version>
                                <extensions>true</extensions>
                        </plugin>
                        <plugin>
                                <groupId>org.eclipse.tycho</groupId>
                                <artifactId>target-platform-configuration</artifactId>
                                <version>${tycho.version}</version>
                                <configuration>
                                        <!--   <target>   <artifact>   <groupId>org.dice</groupId>
<artifactId>org.dice.target</artifactId>
                                <version>${dice.version}</version> </artifact> </target> -->
                                        <environments>
                                                <environment>
                                                        <os>win32</os>
                                                        <ws>win32</ws>
                                                        <arch>x86_64</arch>
                                                </environment>
                                                <environment>
                                                        <os>linux</os>
                                                        <ws>gtk</ws>
                                                        <arch>x86_64</arch>
                                                </environment>
                                                <environment>
                                                        <os>macosx</os>
                                                        <ws>cocoa</ws>
                                                        <arch>x86_64</arch>
                                                </environment>
                                                <environment>
                                                        <os>win32</os>
                                                        <ws>win32</ws>
                                                        <arch>x86</arch>
                                                </environment>
                                                <environment>
                                                        <os>linux</os>
                                                        <ws>gtk</ws>
                                                        <arch>x86</arch>
                                                </environment>
                                        </environments>
                                </configuration>
                        </plugin>
                </plugins>
        </build>
...
```

code snippet 7: configuration pom.xml

4.1.1.4.2. Product configuration (*product* folder)

This folder contains 2 relevant files: a *xxx.product* and *pom.xml* files. The *xxx.product* file contains the necessary information for any Eclipse build:

- **Product ID**: the id of the product. It can be used to refer to the product in all configurations of all the *pom.xml* files
- **Version**: the version of the product
- **Contents (plug-ins or features)**: the list of components included in the RCP
- **Launching configuration**: launching configuration properties depending on the platforms and architectures
- **Icons configuration**: the icons used in the product
- **Splash configuration**: this is the picture that is seen when the application is launched.

The *pom.xml* file, contains information about the building flow process and tasks' goals. It also contains a simple configuration for the final name of the generated files.

```
 ...
<artifactId>org.dice.product</artifactId>
<name>DICE IDE - Product</name>
<packaging>eclipse-repository</packaging>
<parent>
        <groupId>org.dice</groupId>
        <artifactId>org.dice.releng</artifactId>
        <version>${dice.version}</version>
</parent>

<build>
        <plugins>
                <plugin>
                        <groupId>org.eclipse.tycho</groupId>
                        <artifactId>tycho-p2-repository-plugin</artifactId>
                        <version>${tycho.version}</version>
                        <configuration>
                                <includeAllDependencies>true</includeAllDependencies>
                        </configuration>
                </plugin>
                <plugin>
                        <groupId>org.eclipse.tycho</groupId>
                        <artifactId>tycho-p2-director-plugin</artifactId>
                        <version>${tycho.version}</version>
                        <executions>
                                <execution>
                                        <id>materialize-products</id>
                                        <goals>
                                                <goal>materialize-products</goal>
                                        </goals>
                                </execution>
                                <execution>
                                        <id>archive-products</id>
                                        <goals>
                                                <goal>archive-products</goal>
                                        </goals>
                                </execution>
                        </executions>
                        <configuration>
                                <products>
                                        <product>
                                                <id>org.dice.rcp.dice</id>
```

```
                                                    <archiveFileName>dice-
${dice.version}</archiveFileName>
                                            </product>
                                    </products>
                            </configuration>
                    </plugin>
            </plugins>
    </build>
...
```

code snippet 8: product pom.xml

4.1.1.4.3. Target configuration (*target* folder)

A target configuration allows developers to define a list of plug-ins and features available for the product, both at development and building time.

The target definition file *xxx.target*, defines a set of Update Sites on which all product dependencies rely. A set of necessary features are chosen for each Update Site, in order to define correctly the dependencies of the product.

It is normal that more than the strictly necessary plug-ins are loaded and available for the product to use it. This situation happens because a feature usually declares several plugins marked as dependencies, but some of them are not really necessary for the feature, either at developing time or at building time.

The *pom.xml* file does not provide any important information for the building process.

```
...
        <artifactId>org.dice.target</artifactId>
        <name>DICE IDE - Target definition</name>
        <packaging>eclipse-target-definition</packaging>

        <parent>
                <groupId>org.dice</groupId>
                <artifactId>org.dice.releng</artifactId>
                <version>${dice.version}</version>
        </parent>
...
```

code snippet 9: Target pom.xml

4.1.1.4.4. Update Site configuration (*update* folder)

This folder is necessary to build a P2 repository. A P2 repository contains all the necessary information to create an Update Site for the RCP. Normally, this folder should be published to an available URL and is used to update the RCP.

There exist two relevant files: *category.xml* and *pom.xml*.

The *category.xml* file defines the structure of the Update Site. The root element should be a category element. This is a simple aggregator for the whle IDE. Under this element appear all the DICE IDE features, in order to update only those that the user wants to update.

The *pom.xml* file does not provide any important information for the building process. But it is necessary to create the Update Site that allows the users to update the IDE once installed.

```
...
        <artifactId>org.dice.update</artifactId>
        <name>DICE IDE - Update Site</name>
        <packaging>eclipse-repository</packaging>
        <parent>
                <groupId>org.dice</groupId>
                <artifactId>org.dice.releng</artifactId>
                <version>${dice.version}</version>
        </parent>
...
```

code snippet 10: update pom.xml

4.2. Project logical structure

An Eclipse-based product, like DICE, is a stand-alone program built on top of the Eclipse platform. A product may optionally be packaged and delivered as one or more features, which are simply groupings of plug-ins that are managed as a single entity by the Eclipse update mechanisms.

The product can be configured to be built using either a plug-in based approach or a feature-based approach (features). Each developer decides which approach to follow. It is more comfortable to follow the feature based approach, because it updating the components easier, as this can be done independently of the components that do not need an update. If features are not used, the end user needs to update the whole application when a new update is released.

For this reason, the DICE IDE is feature-oriented, and therefore each component is grouped as a feature (see Figure 2).

When a new tool is integrated within the IDE, a new feature is created in order to enable its installation. The purpose is to control which tools can be integrated. In this way, when a new tool is added, extra information can be added to it, like cheat sheets, manuals, etc.

Each integrated tool has its feature(s) and its plug-in(s). More than one plug-in and feature can be used to develop each of them. The next sections explain how they have been integrated within the IDE.

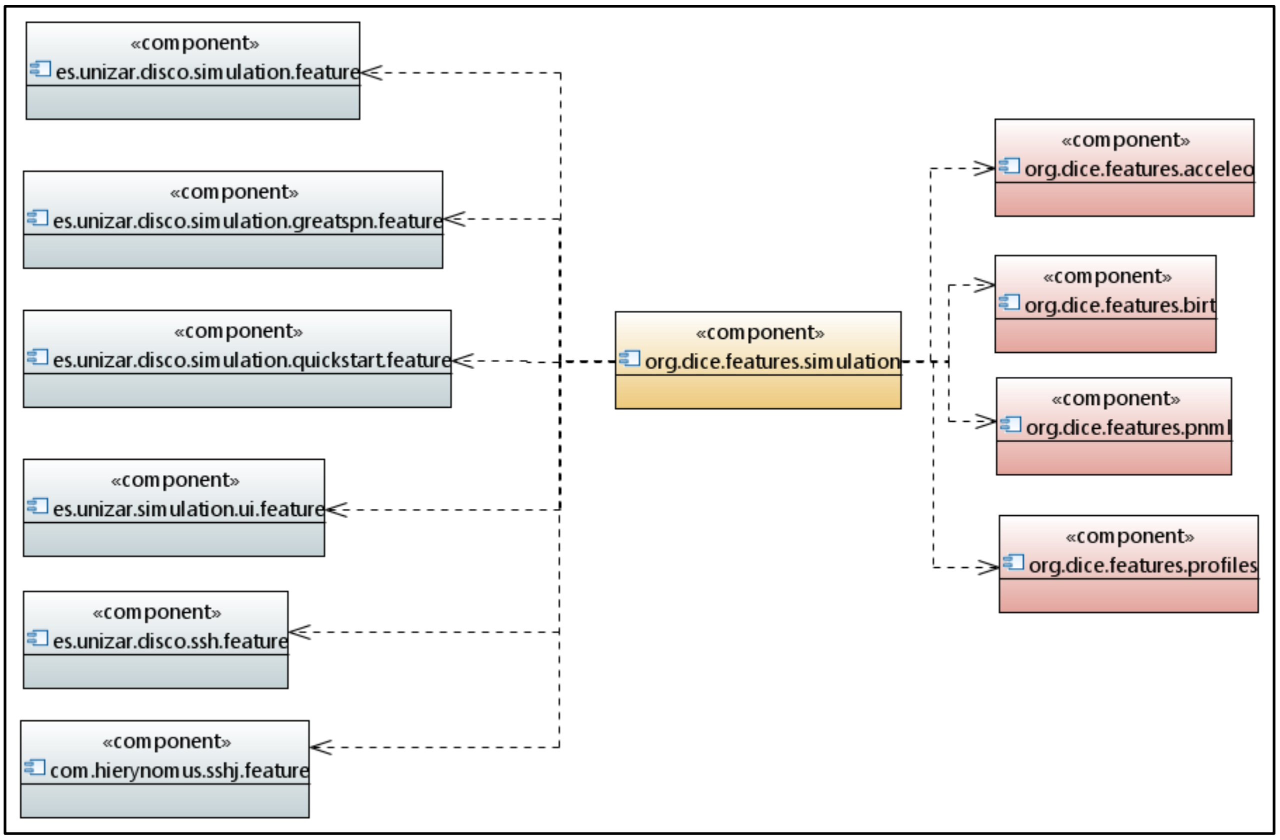
4.2.1. Simulation tool

The Simulation tool has a feature named *org.dice.features.simulation* that enables the tool for the IDE (see Figure 4). It basically contains a utility plug-in that adds a shortcut for the tool, and contains the dependency for the actual tool features:

- *es.unizar.disco.simulation.feature*
- *es.unizar.disco.simulation.greatspn.feature*
- *es.unizar.disco.simulation.quickstart.feature*
- *es.unizar.simulation.ui.feature*
- *es.unizar.disco.ssh.ui.feature*
- *com.hierynomus.sshj.feature*

Apart from these features, the tool depends on other third party components included in the DICE IDE product, necessary for this tool to work:

- *org.dice.features.acceleo*
- *org.dice.features.birt*
- *org.dice.features.profiles*
- *org.dice.features.pnml*


«component»
es.unizar.disco.simulation.feature
«component»
es.unizar.disco.simulation.greatspn.feature
«component»
es.unizar.disco.simulation.quickstart.feature
«component»
es.unizar.simulation.ui.feature
«component»
es.unizar.disco.ssh.feature
«component»
com.hierynomus.sshj.feature
«component»
org.dice.features.simulation
«component»
org.dice.features.acceleo
«component»
org.dice.features.birt
«component»
org.dice.features.pnml
«component»
org.dice.features.profiles
Figure 4: Components diagram and dependencies for Simulation Tool

Each of these features may contain dependencies to other features, and they also may contain plug-ins.

More information can be found in Section 1 of the Companion Document.

4.2.2. Verification tool

The Verification tool has a feature named *org.dice.features.verification* that enables the tool to be used with the IDE (see Figure 5). It basically contains a utility plug-in that adds a shortcut for the tool, and also the dependency for the actual tool features:

- *it.polimi.dice.core*
- *it.polimi.dice.core.ui*
- *it.polimi.dice.verification*
- *it.polimi.dice.verification.ui*
- *it.polimi.dice.verification.uml2json*
- *it.polimi.dice.vtconfig*

In addition to these features, the tool depends on other third-party components included in the DICE IDE product, which are necessary for this tool to work:

- *org.dice.features.base*

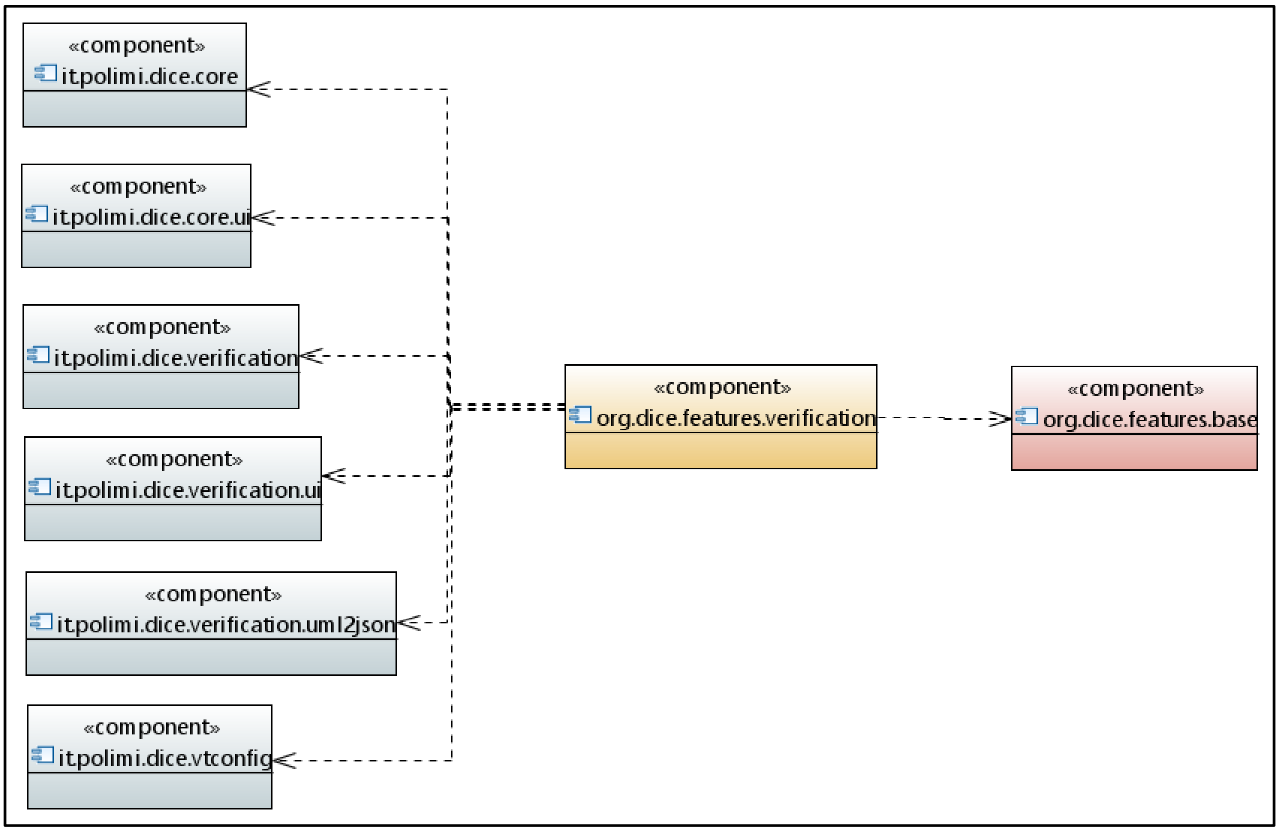


Figure 5: Components diagram and dependencies for Verification Tool

Each of these features may contain dependencies to other features, and they may also contain plug-ins.

More information can be found in Section 2 of the companion document.

4.2.3. Monitoring tool

The Monitoring tool has a feature named *org.dice.features.monitoring* that enables the tool for the IDE (see Figure 6). It basically contains a utility plug-in that adds a shortcut for the tool. In this case, this tool has no extra features because the core of this tool is externally integrated from the IDE.

Nevertheless, the tool depends on other third-party components included in the DICE IDE product, necessary for this tool to get working:

- *org.dice.features.base*

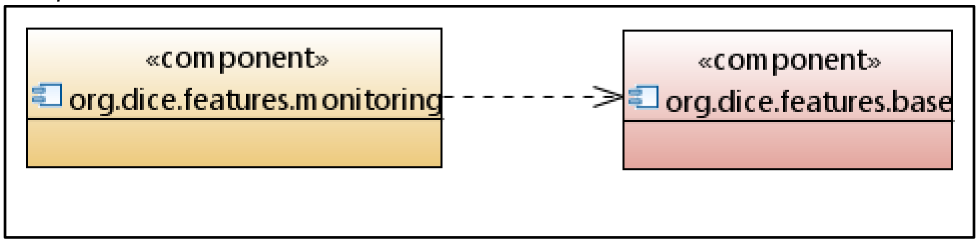


Figure 6: Components diagram and dependencies for Monitoring Tool

More information can be found in Section 3 of the Companion document.

4.2.4. Delivery/Deployment tool

The Delivery or Deployment tool has a feature named *org.dice.features.deployments* that enables the tool for the IDE (see Figure 7). It basically contains a utility plug-in that adds a shortcut for the tool, and also the dependencies on the actual tool features.

This tool is fully maintained by its owner. Hence, it does not contain dependencies to any features of the DICE IDE (see Figure 7)

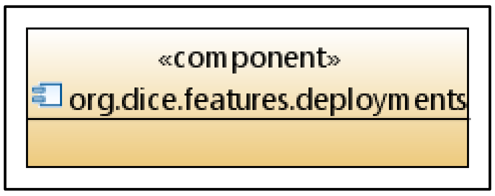


Figure 7: Components diagram and dependencies for Delivery Tool.

More information can be found in section 4 of the Companion document.

4.2.5. Optimization tool

The Optimization tool has a feature named *org.dice.features.optimization* that enables the tool for the IDE (see Figure 8). It basically contains a utility plug-in that adds a shortcut for the tool. In this case, this tool has no extra features because the core of this tool is externally integrated from the IDE.

Nevertheless, the tool depends on other third-party components included in the DICE IDE product, which are necessary for this tool to work:

- *org.dice.features.base*

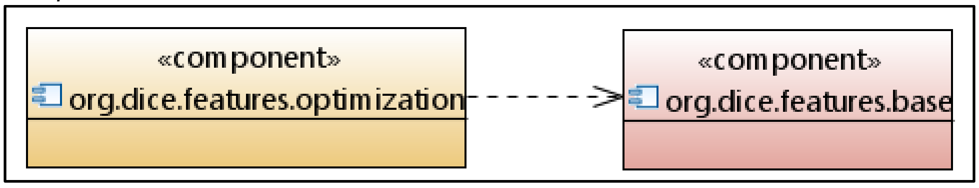


Figure 8: Components diagram and dependencies for Optimization Tool.

More information can be found in section 5 of the Companion document.

4.2.6. Deployment modeling (DICER) tool

The Deployment modelling (DICER) tool has a feature named *org.dice.features.dicer* that enables the tool for the IDE (see Figure 9). It basically contains the dependencies from the actual tool features:

- *it.polimi.dice.lwdg.loadDemo.feature*
- *it.polimi.dice.lwdg.launcher.feature*

Apart from these features, the tool depends on other third-party components included in the DICE IDE product, necessary for this tool to work:

- *org.dice.features.base*

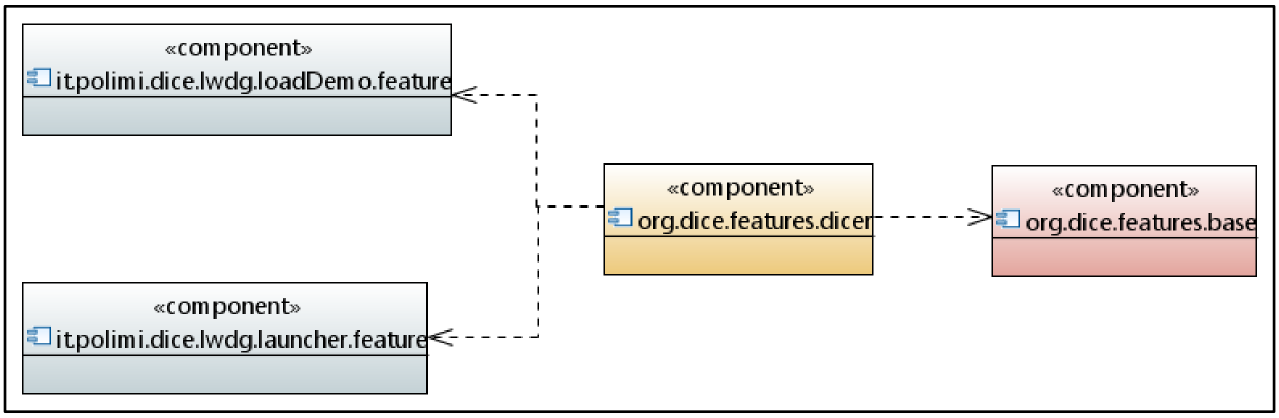


Figure 9: Components diagram and dependencies for DICER Tool.

Each of these features may contain dependencies to other features, and they may also contain plug-ins.

More information can be found in Section 12 of Companion document.

4.2.7. DICE Profiles

The profiles used in DICE have a feature named *org.dice.features.profiles* that enables them for the IDE (see Figure 10). It basically contains the dependencies for the actual profiles features:

- *com.masdes.dam.profile.feature*
- *com.masdes.dam.profile.ui.feature*
- *es.unizar.disco.dice.profile.feature*

In addition to these features, the DICE profiles depend on other third party components included in the DICE IDE product, necessary for this component to get working:

- *org.dice.features.papyrus*

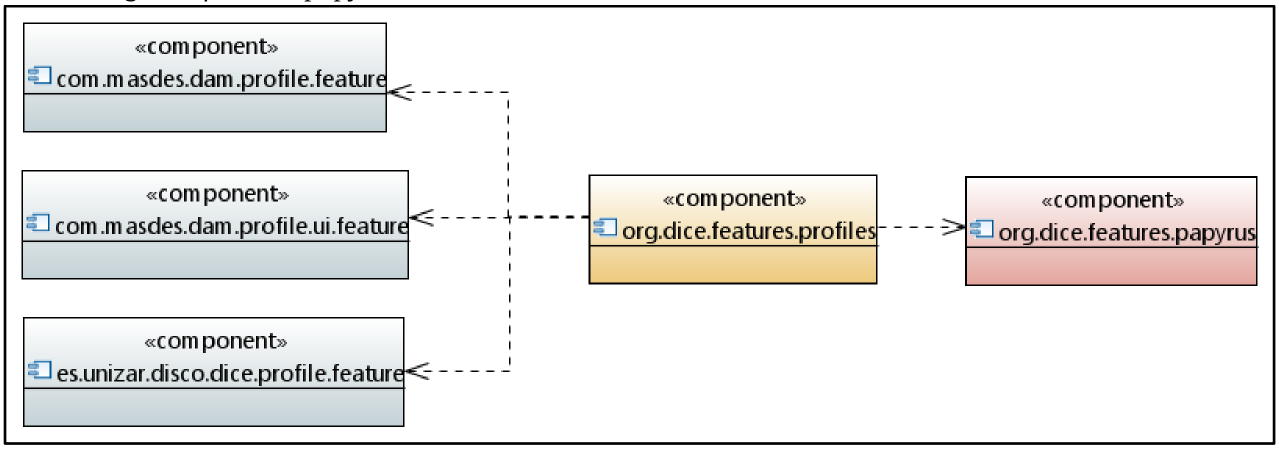

«component»
com.masdes.dam.profile.feature
«component»
com.masdes.dam.profile.ui.feature
«component»
org.dice.features.profiles
«component»
org.dice.features.papyrus
«component»
es.unizar.disco.dice.profile.feature

Figure 10: Components diagram and dependencies for DICE Profiles.

Each of these features may contain dependencies to other features, and also may contain the plug-ins with the development.

4.2.8. Tools that will be integrated in the future

The following tools are not currently integrated within the IDE, so they have no dependencies with the components of the DICE IDE.

- Anomaly detection tool (see section 6 of the Companion document)
- Trace checking tool (see section 7 of the Companion document)
- Enhancement tool (see section 8 of the Companion document)
- Fault injection tool (see section 9 of the Companion document)
- Configuration optimization tool (see section 10 of the Companion document)
- Quality testing tool (see section 11 of the Companion document)

5. Integration approaches

Since not all tools-for technological reasons (programming language, runtime tool, etc.)-have the ability to be fully integrated within the IDE, it is necessary to provide a solution for those that cannot be integrated. In fact, some tools are executed outside the IDE, for example in an external web site, or in an external server.

The DICE IDE offers two ways of to integrate tools:

- Fully integrated
- Externally integrated

Both types of integration are based on a common component for the integration within the IDE. This component contributes the IDE with a menu, through which the user can interact with all the integrated tools (see Figure 11).

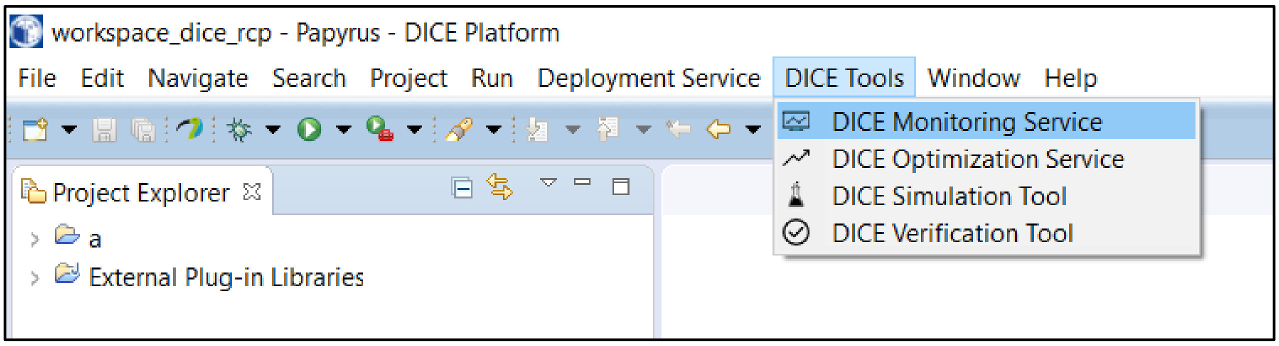

workspace_dice_rcp - Papyrus - DICE Platform
File Edit Navigate Search Project Run Deployment Service DICE Tools Window Help
DICE Monitoring Service
DICE Optimization Service
DICE Simulation Tool
DICE Verification Tool
Project Explorer
a
External Plug-in Libraries

Figure 11: The menu for DICE Tool in the IDE.

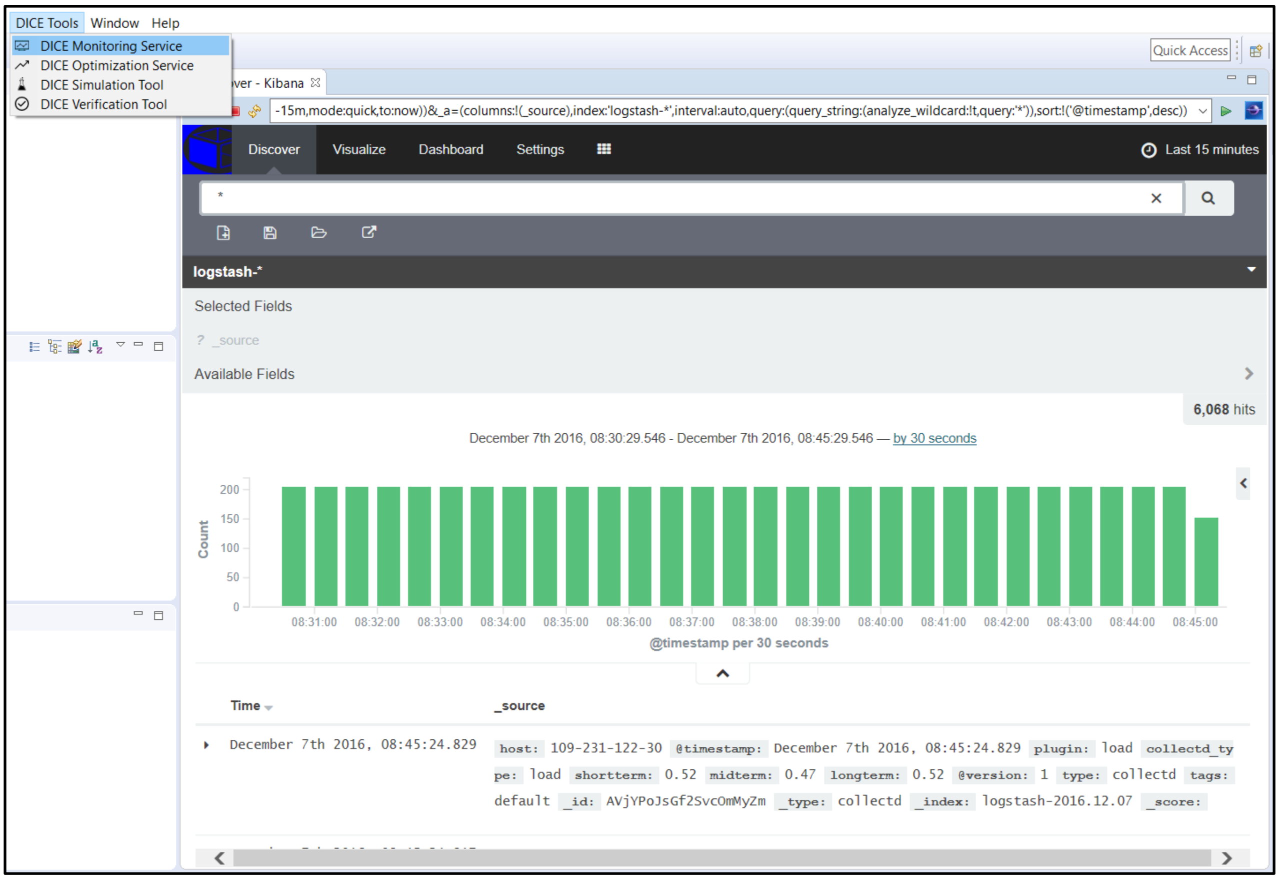
5.1. Externally integrated

This approach is the easiest. It is used when the actual execution environment of the tool resides outside the IDE, for instance on an external server or web service.

The only required information for this approach is to provide the needed information to connect to the external application, typically a URL:

- **Protocol**: HTTP or HTTPS
- **Server**: the address of the server
- **Port**: the port where the server is available
- **Parameters**: possible parameters to be passed when the web service is visited (user id, token...).

There exists a plug-in that implements an abstract mechanism that is offered to all tools that employ this kind of integration. This plug-in provides the support to open the internal web browser of Eclipse with the given page, allowing the user to access it within the IDE (see Figure 12).



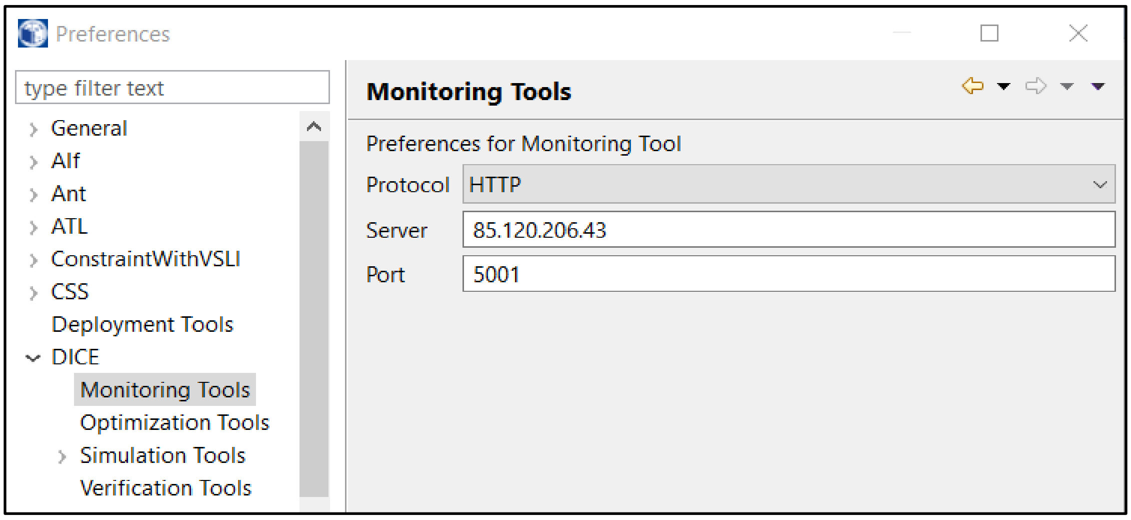
```
-15m,mode:quick,to:now))&_a=(columns:!(_source),index:'logstash-*',interval:auto,query:(query_string:(analyze_wildcard:!t,query:'*')),sort:!('@timestamp',desc))
```

December 7th 2016, 08:30:29.546 - December 7th 2016, 08:45:29.546 — by 30 seconds

```
December 7th 2016, 08:45:24.829   host: 109-231-122-30 @timestamp: December 7th 2016, 08:45:24.829 plugin: load collectd_ty
                                  pe: load shortterm: 0.52 midterm: 0.47 longterm: 0.52 @version: 1 type: collectd tags:
                                  default _id: AVjYPoJsGf2SvcOmMyZm _type: collectd _index: logstash-2016.12.07 _score:
```

Figure 12: Example of Monitoring Tool, an external tool integration.

It also provides an abstract Eclipse Preferences page that allows the user to modify these properties. In this way, the integration of the external web server tool can be modified dynamically by the user, if needed (see Figure 13).



Monitoring Tools

Preferences for Monitoring Tool

Protocol: HTTP

Server: 85.120.206.43

Port: 5001

Figure 13: Example of Monitoring Tool external web service configuration.

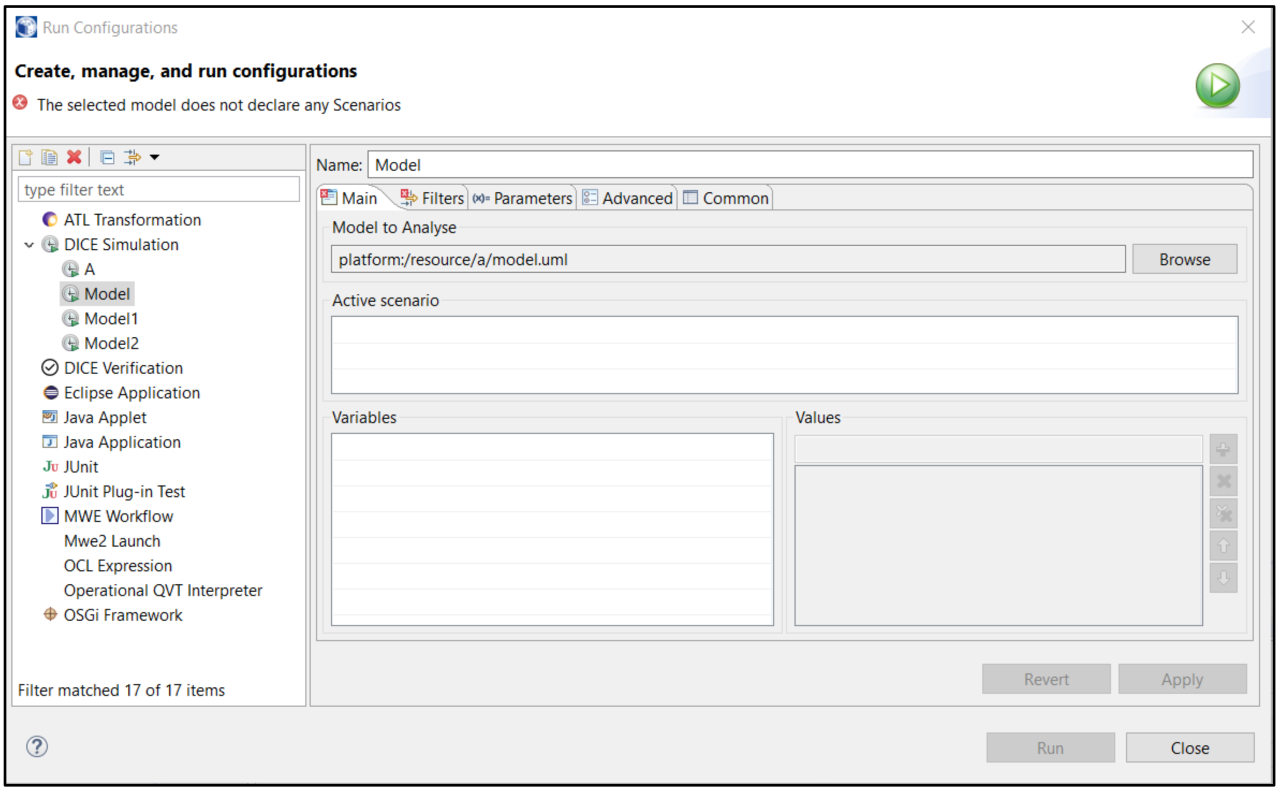
5.2. Fully integrated

This approach requires more effort from the tool owner, as it is intended to develop a fully functional architecture in the IDE that allows the user to interact with the tool and perform all the needed operations.

To integrate a tool with this approach come Eclipse development skills are required. Detailed information can be found online to assist with Eclipse development [34]. These resources can be used to learn how to develop Eclipse plug-ins and contribute the IDE [35] to provide new functionalities like wizards, dialogs, launchers and views.

Depending on the complexity of the tool the level of diffuculty in integrating it within the IDE varies.

Figure 14 shows an example of fully integrated tool, and in particular the Simulation tool.


Run Configurations

Create, manage, and run configurations

The selected model does not declare any Scenarios

type filter text

- ATL Transformation
- DICE Simulation
  - A
  - Model
  - Model1
  - Model2
- DICE Verification
- Eclipse Application
- Java Applet
- Java Application
- JUnit
- JUnit Plug-in Test
- MWE Workflow
- Mwe2 Launch
- OCL Expression
- Operational QVT Interpreter
- OSGi Framework

Filter matched 17 of 17 items

Name: Model

Main | Filters | Parameters | Advanced | Common

Model to Analyse: platform:/resource/a/model.uml

Active scenario

Variables

Values

Figure 14: An example of the Simulation Tool, a fully integrated tool.

6. Getting started with DICE IDE

The purpose of this section is to learn how to get started with the DICE IDE. For those who are beginners with Eclipse, the following are example use cases that could be useful to start with the IDE.

All use cases introduced can be found within the IDE implemented using the Eclipse Cheat Sheets solution [37], which provides an easy way for a user to use the multiple tools within the DICE IDE.

6.1. Initial Configuration of the IDE

In the first run of DICE, there are some properties the user can configure in the IDE in order to prepare it to work properly. To configure the IDE, the following steps should be performed:

1. **Open the preferences**. Select "Window->Preferences" from the main menu.
2. **Select the Papyrus entry in the tree**. In this section the user can modify the parameters of the Papyrus diagrams.
3. **Select the Java entry in the tree**. In this section the user can modify some attributes of Java, like the JDK or JRE to be used, etc.…
4. **Select the DICE entry in the tree**. In this section the user can modify some parameters of each DICE Tool; most of them are mandatory to be able to use the tool.

6.2. Install new software

New software can be installed in the IDE. This cheat sheet shows how to install new third party software in order to provide more functionalities to the IDE. The existing way to install new components on DICE is through the Update Sites approach.

1. **Open the Update Sites wizard** from the menu "Help → Install new software". In this dialog box the user can install the software giving the URL that points to a repository.
2. **Set the URL of the repository**. In the "Work with" field, the URL of the repository should be typed. The user should specify a label for the new repository.
3. **Select the software to be installed**. Once added, the list of available components will be filled. The user selects those to be installed.
4. **Filter the view**. The list can be filtered with the buttons at the bottom of the window. It can show only the latest versions of the items, group them by category, hide the installed items…
5. **Install the selected items**. Once selected the desired software, in the next step the user must accept the licenses and finally, when asked, the IDE should be restarted in order to apply the changes.

6.3. Update the components

The following steps are used to keep the IDE and the components updated.

1. **Configure the Preferences**. First configure properly the Preferences of the IDE in order to set the used Update Sites and the IDE to indicate the updating period.
   a. Select "Window → Preferences" from the main menu and select the "Install/Update" entry. Here the user can decide to see only the latest versions or all of them.

   b. Inside this entry, the user can find the "Automatic Updates" option. He can choose if the IDE should automatically check if there is any available update. Also, he can specify when the IDE should check it, and what to do: show only a notification or download it automatically.
   c. At the same level of the step above, the user can find the "Available Software Sites" option. This entry lists all used repositories where the IDE will look for the updates. When a new third-party component is installed, its repository will be included there automatically. Nevertheless, the user can add them manually.
2. **Update the IDE manually**. Also, the user can always manually check if there are any new updates through the "Help → Check for updates" menu entry.

6.4. Connect to SVN repositories

This cheat shows how to deal with SVN in the IDE. The user can configure some repositories and import the needed projects.

1. **Open SVN Perspective**. First of all, the SVN Perspective needs to be activated using the menu "Window → Perspective → Open Perspective → Other" and select "SVN Perspective".
2. **Add a new Repository**. In the "SVN Repositories" view, right click there and click on "New → Repository Location" menu
   a. Fill the URL field and click on Finish. If the URL is valid, the repository will be added automatically
3. **Import an existing Project into the Workspace**. Browse the added repository and select the folder about to be imported. Right click on it and select the "Checkout" action. Follow the steps in the wizard in order to create a new Project (if the folder is not an Eclipse Project) or import it (if the folder is yet an Eclipse Project).

6.5. Connect to GIT repositories

This cheat shows how to deal with GIT in the IDE. The user can configure some repositories and import the projects you need.

1. **Open GIT Perspective**. First of all, the SVN Perspective needs to be activated using the menu "Window → Perspective → Open Perspective → Other" and select "SVN Perspective"
2. **Clone an existing Repository from a URL**. In the "Git Repositories" view, click on the button "Clone a Git Repository and add the clone to this view". This will open a dialog where the user can type the URL and provide authentication if needed for the connection.
3. **Import an existing Project into the Workspace**. Browse the added repository (the "Working Tree" leaf) and select the folder you want to import. Right click on it and select the "Import projects" action. Follow the steps in the wizard in order to import the existing Eclipse Projects to the Workspace

6.6. Papyrus

The following Cheat Sheets show how to start working with Papyrus, in order to create a project within the IDE and create new UML diagrams.

6.6.1. Create UML models with Papyrus

This cheat sheet shows how the user can create a new UML Papyrus Project.

1. **Open Papyrus Perspective**. The first step is to enable the Papyrus Perspective. Click on the menu "Window → Perspective → Open Perspective → Other" and select Papyrus from the list.
2. **Create a new project**. To create a new Project, simply click on the menu "File → New → Papyrus Project". The wizard to create the new project opens.
3. **Configure the project properties**. In the first step, the user needs to specify the Language of the contained diagrams. Select UML as value.
4. **Set the project and default model name**. Set the project name in the field. Also, the user can set the name of the default model.
5. **Select the kind of diagrams by default**. The created model can contain one or more diagrams by default. The user can select them in this window. Also, it can be specified if the default basic primitives should be included or not. The name of the root element of the model can be specified here too. Finally, some profiles can be selected and applied to the whole model (this step can be performed after creating the model).
6. **Add new diagrams** to the created model if no one exists by clicking "Create View" button in the opened model.

6.6.2. Apply Profiles to UML models with Papyrus

The following cheat sheet shows how the user can apply some UML Profiles and stereotypes to the elements of a UML Papyrus model.

1. **Apply profiles to the model**. First of all, the profiles should be applied to the model:
    a. Open the model about to be profiled from the Project Explorer.
    b. Open the Properties View through the menu "Window → Show view → Other" and select "General → Properties".
    c. Select the Root Element of the model. Open the Papyrus Perspective and in the Model Explorer View select the top root element. This element is the root of the whole model and contains all the needed information applied to all its contents (like the profiles).
    d. Select the Profile section in the Properties. Having selected the root element, the user can select in the Properties view opened previously the Profile tab. Here all the existing profiles can be added to the model.
    e. Apply an existing registered Profile. In this tab, clicking the button "Apply registered profile" a dialog opens listing all the registered profiles. Here the user can select one or more profiles to be applied to the model. All the listed profiles are available from the Plug-ins.
2. **Apply stereotypes to the model elements**. Once loaded the profiles in the model, the model is ready for applying the profile stereotypes to the elements. The following example will be applied to a Class Diagram:
    a. Select an existing element in the diagram (or create a new one). Having always the Profile tab opened in the Properties view, select an element in the diagram. If no one exists, create a new Class.
    b. In the Profiles tab, click the "Apply stereotype" button. A new dialog appears and the user can select those Stereotypes that are valid to be applied to the selected element, according the loaded Profiles.

c. See how the element in the diagram has changed and how it now shows the applied Stereotypes.
d. Each Stereotype may have properties. The user can set all of them by expanding the Stereotype list in the Profiles tab. The available values depend on the type of the Stereotype.

6.7. DICE Tools

The Cheat Sheets listed below show how the user can start working with the DICE Tools that are integrated within the IDE.

6.7.1. Profiles component

This cheat sheet explains how to apply the DICE profiles in the UML models. The DICE Profiles are an extension of the MARTE-DAM profiles.

The mechanisms for applying the DICE-specific stereotypes are the same as those presented in Section 5.6.2. The user simply needs to select the DICE profiles when asked about which profiles to apply. There exists a special Cheat Sheet named "Profile and Stereotype Papyrus models" for applying profiles and stereotype UML elements, and this one simply refers to the other one, inviting the user to use the DICE profiles.

6.7.2. Simulation Tool

The Cheat Sheet fot the Simulation Tool provides an example that allows the user to know how to start working with this tool. The steps included are as follows:

1. **Create a new Papyrus project**. First of all, there should exists a Papyrus project created in the workspace. If no one exists, the user can follow the Cheat Sheet for that purpose, then come back to this Cheat Sheet
2. **Apply MARTE and DICE profiles to the model**. The created model should be profiled using the MARTE and DICE profiles. For DICE, only the *DPIM* profile, and for MARTE the *MARTE_AnalysisModel* profile should be applied (see the Cheat Sheet for applying profiles and stereotypes).
3. **Create 2 diagrams to the model**. Two new diagrams should be created in the model. It can be done by opening the model and adding a Deployment diagram and an Activity diagram from the Model Explorer view.
4. **Complete the Deployment diagram**. The Deployment diagram should be completed adding some elements:
   a. Add a "*Node*" element from the palette
   b. Add an "*Artifact*" element inside the created Node
5. **Complete the Activity diagram**. The Activity diagram should be completed adding some elements:
   a. Add an "*Initial Node*" element inside the Activity
   b. Add an "*Activity Partition*" element inside the Activity. Then open the Properties view, and select the UML tab and set the property "*Represents*" to the Artifact created in the previous step. Use the "3 points" button to browse the tree model and find this element
   c. Add an "*Opaque Action*" element inside the Activity Partition
   d. Add an "*Activity Final Node*" element inside the Activity. This element is placed in the same entry of the Initial Node in the Palette
   e. Add a "*Control flow*" link between the "*Initial Node*" and the "*Opaque Action*"

   f. Add a "*Control flow*" link between the "*Opaque Action*" and the "*Activity Final Node*"

6. **Stereotype the Deployment elements**. There are some elements in the Deployment model that should be stereotyped by following these steps:
   a. Select the "*Artifact*" element, and then open the Properties view and select the Profile tab. Add a <<*PaLogicalResource*>> stereotype
   b. In the <<*PaLogicalResource*>> stereotype, browse the properties and set the value of the "*poolSize*" property to 1
7. **Stereotype the Activity elements**. Also, some elements in the Activity model should be stereotyped by following these steps
   a. Select the "*Opaque Action*", and go to the Properties view and select the Profile tab. Add a <<*GaStep*>> stereotype
   b. In the <<*GaStep*>> stereotype, browse its properties and set the value of the "*hostDemand*" property to `(expr=0.6,unit=s,statQ=mean,source=est)`. This is a MARTE known value format for the duration of the activity in seconds
   c. Select the "*Initial Node*", and go to the Properties view and select the Profile tab. Add a <<*GaWorkloadEvent*>> stereotype
   d. In the <<*GaWorkloadEvent*>> stereotype, browse its properties and set the value of the "*pattern*" property to `open=(arrivalRate=(expr=$rate,unit=Hz, statQ=mean,source=est))`. This is a MARTE known value format for the work load
   e. Select the "*Activity*", and go to the Properties view and select the Profile tab. Add a <<*GaAnalysisContext*>> and a <<*DPIMscenario*>> stereotypes
   f. In the <<*GaAnalysisContext*>> stereotype, browse its properties and set the value of the "*context*" property to `out$responseTime` and `in$rate`
   g. In the <<*DPIMscenario*>> stereotype, browse its properties and set the value of the "*respT*" property to `(expr=$responseTime,statQ=mean,soUurce=calc)`
8. **Configure the launching**. The launching should be configured in order to obtain valid results.
   a. The configuration can be launched from the menu "DICE Tools → DICE Simulation Tool", or right click over the model in the Project Explorer, click on “Run As...” and then on the “DICE Simulation” entry
   b. In the configuration window of the selected model, there is a section group for the Variables and the given Values. Here the user must select the Rate variable and set as value "1"
   c. Finally, if the configuration is valid, the user can run it and see the results
9. **See the results of the launching**. If the configuration is correct, the final step is to see the results
   a. Open the view to see the results. In the menu "Window → Show View → Other → DICE Simulation → Invocations Registry"
   b. Double click over the identifier of the simulation
   c. A tree editor will be opened showing the results of the execution. It can be found the calculated response time in the simulation

6.7.3. Monitoring Tool

The Monitoring Tool is integrated within the IDE, and the following Cheat Sheet shows how the user can use this tool via the IDE.

1. **Configure the tool**. Before launching the tool, the user needs to check the connection properties to the external web service

   a. Open the preferences page of the Monitoring Tool (Window → Preferences → DICE → Monitoring Tool)
   b. Check the attributes of the preferences page and modify the values properly. These values are used when launching the tool
2. **Connect to the Monitoring Service Admin UI**. The Admin interface of Monitoring Service gives access to operations that manage the monitoring platform and the monitored nodes. Simply click on the menu "DICE Tools → DICE Monitoring Service", and the web server will be opened
3. **Launch the Monitoring Service Visualization UI**. The Visualization interface of the Monitoring Service enables end-users to create graphs out of collected monitored data. This interface is available on the menu "DICE Tools → DICE Monitoring Service Visualization UI"

6.7.4. Optimization Tool

The Optimization Tool is externally integrated, and this Cheat Sheet shows how the user can use it.

1. **Configure the tool**. Before launching the tool, the user needs to check the connection properties to the external web server
   a. Open the preferences page of the Optimization Tool (Window → Preferences → DICE → Optimization Tool)
   b. Check the attributes of the preferences page and modify the values properly. This values are used when launching the tool
2. **Launch the Optimization Service**. As the tool is externally integrated, the execution is very easy. Simply click on the menu "DICE Tools → DICE Optimization Service", and the web server will be opened

6.7.5. Delivery Tool

The Delivery Tool is externally integrated, and this Cheat Sheet shows how the user to deploy a Data Intensive Application in the test bed.

1. **Install the DICE Deployment Service**. Before running the Deployment Tool from the IDE, the user needs to install the DICE deployment service within a server. The instructions are available at the following link: https://github.com/dice-project/DICE-Deployment-Service/wiki/Installation
2. **Create a container for the IDE**. A new container should be created for the IDE. The instructions are available at this link: https://github.com/dice-project/DICE-Deployment-Service/wiki/Installation#container-management
3. **Configure the tool connection in the IDE**. Before launching the tool, the user needs to check the connection properties to the configured server
   a. Open the preferences page of the Delivery Tool (Window → Preferences → Deployment Tools)
   b. Check the attributes of the preferences page and modify the values properly. This values are used when launching the tool
4. **Open a TOSCA blueprint**. The user needs to download some TOSCA blueprint examples. Depending on whether he is using OpenStack or FCO, he needs to download one example or other. He can do it by cloning the whole repository. Import or copy the downloaded blueprint into the project
   a. If using OpenStack, take the files from the folder "storm/storm-openstack.yaml"

   b. If using FCO, take the files from the folder "storm/storm-fco.yaml"
5. **Open a Deployment Service panel**. As the tool is externally integrated, the execution is very easy. Simply click on the menu "Deployment Service → DICE Deployments Tool UI", and the web server will be opened
   a. Open the web server
   b. Log into the DICE deployment service using your username and password.
6. **Deploy the blueprint**. The following steps allows users to deploy the blueprint. These steps should be performed in the opened web browser
   a. In the DICE - Deployment Service panel, click on the Upload Blueprint.
   b. In the file dialog, select the .yaml blueprint file and confirm the selection to close the dialog.
   c. Click the Upload button that appears.
   d. Wait for the deployment to finish. This may take several minutes or longer. If all goes well, the DICE - Deployment Service will show a "Blueprint successfully deployed" message.

6.7.6. Verification Tool

This section describes the methodological steps that the DICE user follows for verifying DICE UML models with D-VerT, the verification tool of the DICE platform.

D-VerT is useful for assessing the temporal behaviour of a DIA. The validation is carried out at the DTSM level to either:

- verify the presence of bottleneck nodes in a Storm application
- verify the temporal feasibility of a Spark job.

The goal of the guide is to show how to model a simple DIA called WikiStats, which analyses web pages from the popular Wikimedia website (“source node”) and stores the result of the analysis into a database (“storage node”).

1. **DIA design and verification in practice**. The first step is to create the UML project and initialize the Class Diagram:
   a. Create a new Papyrus UML project (select the “Class diagram”).
   b. Open the class diagram and instantiate two packages, one for the DPIM model and another for the DTSM model and apply on the packages the DICE::DPIM and the DICE:DTSM UML profiles, respectively. Specifically, as this guide exemplifies the creation of a Storm application, add to the project the “Core” and the “Storm” metamodels/profile that can be found in the DTSM entry.
2. **DPIM modelling**. In the DPIM package, the user models the high-level architecture of the DIA, as a class diagram representing the computations over various data sources. To this end, the following steps should be performed:
   a. Instantiate a new class and apply the <<DPIMComputationNode>> stereotype on it
   b. Model the data sources, which can be either profiled by using the <<DPIMSourceNode>> of the <<DPIMStorageNode>> stereotypes, depending on the kind of data source
   c. Finally, associate the computation nodes to the available data sources.
3. **DTSM modelling**. In the DTSM package, the user specifies which technologies implement the various components of the DIA. In particular, the user models the actual implementation of the computations declared in the DPIM, plus all the required technology-specific details. We use Apache Storm as target technology for “WikistatsApplication” <<DiceComputationNode>> of the DPIM model.

   a. By drag-and-dropping from right panel, add to the design all the nodes defining the application.
   b. From the bottom panel, select the proper stereotype for each component of the application. The stereotype is chosen according to the kind of the node, that can be either a data source (<<StormSpout>>) or a computational node (<<StormBolt>>).
   c. Connect the nodes together through directed associations. Each association defines the subscription relation between two nodes: the subscriber, at the beginning of the arrow, and the subscribed node, at the end of the arrow.
   d. The final topology is obtained, which will be verified with D-VerT.
4. **DTSM modelling - specify the stereotype values**. Before running the verification tool, specify the values of the parameters related to the technology implementing the application.
   a. Select a node and define, in the bottom panel, all the information needed for the verification. The values that are required to verify the topology are the following:
      i. *parallelism, alpha, sigma*, for the bolts
      ii. *parallelism, averageEmitRate* for the spouts.
5. **Verify the topology with D-VerT**. Verify the topology with D-VerT by clicking on the "Run configurations" button. In "Run configuration", provide the following information:
   a. The model to be verified (from the Browse menu)
   b. The number of time positions to be used in the verification process (time bound)
   c. The plugin that D-VerT uses to verify the model
   d. The bolts that the user wants to test for undesired behaviours.
6. **Run D-VerT and monitor running**. Run D-VerT and monitor its running on the server in the D-VerT dashboard. The following information are available:
   a. The result of the verification, that is SAT, if anomalies are observed, or UNSAT.
   b. In case of SAT, the output trace produced by the model-checker shows the temporal evolution of all the model elements in detail and the graphical representation of the verification outcome shows the anomalies for a qualitative inspection.

6.7.7. DICER Tool

This Cheat Sheet shows an example of use of the DICE DICER Tool.

1. **Prepare a DDSM diagram**. First of all, the user is assumed to have prepared the DDSM diagram, in case the user did not do this, he needs to execute the appropriate Cheat Sheet to do that.
2. **Generate the associated deployable TOSCA blueprint**. Following on, a rollout and deployment procedure can ensue exploiting the DICER plugin to generate the associated deployable TOSCA blueprint. In order to do so right-click on the UML model on the model-navigator pane (usually in the left-hand side of the DICE IDE) and select to run through it a run configuration, specifically pressing the "DICER launcher" run-configuration.
3. **Configure DICER run**. Pressing the appropriate run-configuration button ("Generate TOSCA Blueprint") will lead to the run-configuration configuration: here the user can set the URL to contact the DICER service (a local default is always present and pre-configured); the user is able to optionally specify the path to the input model and for the output model to be placed.
4. **Run the configuration**. Finally, the user can click on the "run" button in the lower-right corner of the run-configuration pop-up window, to generate the TOSCA blueprint; Pressing F5 and refreshing the workspace will reveal the generated blueprint.

5. **See the results on the Delivery tool**. The TOSCA blueprint should be submitted to the DICE Deployment Service for further processing - see the Deployment Service Cheat Sheet.

7. Conclusions and future plans

The overall goal of the DICE IDE is to become the main access gateway for all end users who want to follow the DICE methodology in their practice. This document presented in detail the initial version of the DICE Framework, which includes an extensive presentation of the DICE IDE, used by all DICE tools.

Moreover, it showed the commitment of the DICE consortium to keep the IDE updated according to the Eclipse's Framework releases, in order to ensure that the IDE will always use the latest possible versions of the integrated components. The IDE will be delivered with a default set tools. It is up to each tool owner to keep each independent tool updated. Once a tool is updated, it can be updated also in the DICE IDE via the Update Site mechanism offered by Eclipse.

A critical point is the plan for integrating all tools within the IDE. A detailed Integration Plan is available in **D1.4 Architecture definition and integration plan (M24)**. Nevertheless, let us provide here a brief summary of this topic.

The next iteration of this deliverable, to be submitted in M30, will include a detailed and final description of the **DICE IDE (v1.0.0)**, presenting all critical elements and differences with the current version. Moreover, it will present a short plan of how the IDE will be maintained and updated after the end of the project.

Table 1 summarize the main achievement of the DICE framework tools in terms of compliance with the initial set of requirements presented in the deliverable D1.2

Requirement	Title	Priority	Level of fulfilment
R1.1	Stereotyping of UML diagrams with DICE profile	MUST	✓
R1.2	Guides through the DICE methodology	MUST	✓
R1.6	Quality testing tools IDE integration	MUST	✓
R1.7	Continuous integration tools IDE integration	MUST	✓
R1.7.1	Running tests from IDE without committing to VCS	MUST	✓
R2IDE.1	IDE support to the use of profile	MUST	✓
R3IDE.1	Metric selection	MUST	✓
R3IDE.2	Time out specification	MUST	✓
R3IDE.3	Usability	MUST	✓
R3IDE.4	Loading the annotated UML model	MUST	✓
R3IDE.4.1	Usability of the IDE-VERIFICATION_TOOLS interaction	MUST	✓
R3IDE.4.2	Loading of the property to be verified	MUST	✓
R3IDE.5	Graphical output	MUST	✓

R3IDE.5.1	Graphical output of erroneous behaviours	MUST	✓
R3IDE6	Loading a DDSM level model	MUST	✓
R3IDE7	Output results of simulation in user-friendly format	MUST	✓
R4IDE1	Resource consumption breakdown	MUST	✓
R4IDE2	Bottleneck Identification	MUST	✓
R4IDE3	Model parameter uncertainties	MUST	✓
R4IDE4	Model parameter confidence intervals	MUST	✓
R4IDE5	Visualization of analysis results	MUST	✔
R4IDE6	Safety and privacy properties loading	MUST	✔
R4IDE7	Feedback from safety and privacy properties monitoring to UML models concerning violated time bounds	MUST	✓
R4IDE8	Relation between ANOMALY_TRACE_TOOLS and IDE	MUST	✓

Table 1: Level of compliance of the current version with the initial set of requirements

8. References

[1] https://wiki.eclipse.org/Rich_Client_Platform

[2] https://www.eclipse.org/eclipse/platform-core

[3] https://www.eclipse.org/orbit

[4] https://www.eclipse.org/e4

[5] https://www.eclipse.org/equinox

[6] https://eclipse.org/gef

[7] https://www.eclipse.org/pde

[8] https://www.eclipse.org/swt

[9] https://www.eclipse.org/jdt

[10] https://www.eclipse.org/modeling/emf

[11] https://wiki.eclipse.org/Acceleo

[12] https://projects.eclipse.org/projects/modeling.mmt.qvt-oml

[13] https://wiki.eclipse.org/OCL

[14] https://wiki.eclipse.org/MDT/UML2

[15] https://wiki.eclipse.org/Graphical_Modeling_Framework

[16] https://www.eclipse.org/Xtext

[17] https://projects.eclipse.org/projects/webtools

[18] https://www.eclipse.org/atl

[19] https://en.wikipedia.org/wiki/Papyrus_(software)

[20] https://papyrusuml.wordpress.com/2016/10/10/marte-1-2-1

[21] https://projects.eclipse.org/free-tags/marte

[22] https://github.com/subclipse/subclipse

[23] https://www.eclipse.org/egit

[24] http://pnml.lip6.fr

[25] https://github.com/dice-project/DICE-Profiles

[26] https://github.com/dice-project/DICE-Simulation

[27] https://github.com/dice-project/DICE-Deployment-IDE-Plugin

[28] https://github.com/dice-project/DICE-Verification

[29] https://eclipse.org/tycho

[30] https://github.com/dice-project/DICE-Platform

[31] http://blog.vogella.com/2015/12/15/pom-less-tycho-builds-for-structured-environments

[32] http://www.vogella.com/tutorials/EclipseTycho/article.html

[33] https://www.eclipse.org/eclipse/platform-releng

[34] http://www.vogella.com/tutorials/eclipse.html

[35] http://www.vogella.com/tutorials/EclipsePlugin/article.html

[36] https://en.wikipedia.org/wiki/BIRT_Project

[37] https://www.eclipse.org/pde/pde-ui/articles/cheat_sheet_dev_workflow

[38] http://www.vogella.com/tutorials/Eclipse3RCP/article.html

[39] https://github.com/dice-project/DICER