Antti Kalevi Hakala

Tool Integration in Eclipse

Master's Thesis
in Software Engineering
2nd November 2005

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Antti Kalevi Hakala
Contact information: antti.hakala@gmail.com
Title: Tool Integration in Eclipse
Työn nimi: Työkaluintegraatio Eclipsessä
Project: Master's Thesis in Software Engineering
Page count: 134
Abstract: In this thesis, Eclipse is studied as a platform for tool integration. As a theoretical framework, a tool-centric point of view to CASE systems is presented. This includes classification of CASE systems, tool interfacing paradigms, levels of integration, and the key challenges of CASE systems. Eclipse is placed in this framework. Also, tool encapsulation mechanisms in Eclipse are examined. As a practical example, a case study of tool integration in Eclipse is provided. This is a small open source project administered by the author.
Suomenkielinen tiivistelmä: Tämä pro gradu -työ tutkii Eclipseä alustana työkaluintegraatiolle. Teoreettisena kehyksenä esitetään työkalupainotteinen näkökulma CASE-järjestelmiin. Tämä kehys tutkii CASE-järjestelmien luokittelua, työkalujen liittämistä CASE-järjestelmiin, integroinnin eri tasoja sekä CASE-järjestelmien olennaisia haasteita. Myös Eclipse sijoitetaan tähän kehykseen. Tämän lisäksi tutkitaan kuinka työkalu kapseloidaan Eclipsessä. Käytännön esimerkkinä esitetään tapaustutkimus työkaluintegraatiosta Eclipsessä. Tämä on pieni kirjoittajan ylläpitämä avoimen lähdekoodin projekti.
Keywords: Eclipse, CASE, tool integration, tool interfacing
Avainsanat: Eclipse, CASE, työkaluintegraatio

Acknowledgements

I want to thank Jonne Itkonen and Tommi Kärkkäinen, who supervised this thesis and gave me good instructions regarding the content, structure, and writing in general. Your ciriticism spurred me on to write better. Additional thanks to Jonne Itkonen for the subject of the case study. Integrating UMLGraph into Eclipse was a subject of right proportion. I also want to thank Diomidis Spinellis, the author of UMLGraph, for answering my queries and wishing me good luck on my integration attempt. And last but not least, I want to thank Sami Nieminen for running and debugging LightUML in Windows environment.

Contents

Acronyms

AOSD	Aspect-Oriented Software Development
API	Application Programming Interface
AWT	Abstract Window Toolkit
BSD	Berkeley Software Distribution
CVS	Concurrent Versions System
DTD	Document Type Definition
EAI	Enterprise Application Integration
EMF	Eclipse Modeling Framework
EPL	Eclipse Public License
GUI	Graphical User Interface
HCI	Human-Computer Interaction
HTML	HyperText Markup Language
I-CASE	*See ICASE.*
ICASE	Integrated CASE environment
IDE	Integrated Development Environmet
IEEE	The Institute of Electrical and Electronics Engineers
IPSE	Integrated Project Support Environment
JAR	Java Archive
JDT	Java Development Tooling
JNI	Java Native Interface

JVM	Java Virtual Machine
PDE	Plug-In Development Environment
RCP	Rich Client Platform
SDE	Software Development Environment
SDK	Software Developer's Kit
SEE	Software Engineering Environment
SUA	Eclipse Foundation Software User Agreement
SW	Software
SWT	Standard Widget Toolkit
TDD	Test-Driven Development
UI	User Interface
UML	Unified Modeling Language
XML	eXtensible Markup Language

List of Figures

1 Introduction

A single distinct tool can help a software developer only to some extent. There exists no "tool-of-all-trades" that handles every task—a developer has to use several tools on the process. As the different development tasks are not completely distinct or isolated, transferring data between tools, or more generally tool interoperability, becomes an issue. If a high level of interoperability is required, tools have to be integrated into some common framework that provides the ground rules for tool interaction. Integrating a tool into a framework brings many benefits, but it also has its price.

The theoretical basis of this thesis is provided in Chapter 2. Firstly, the classification of different tools is discussed. This clarifies the meaning of the word "tool", which on this general level of discussion can mean any product that is used to aid in software engineering. Examination of tool integration is divided into tool interfacing, which describes the manner in which a tool is connected to a framework, and the levels of tool integration, which describe the conceptual levels of integration that are achieved. These are examined separately. The key challenges underlying tool integration are discussed last. These challenges are fundamental and not yet risen to. They could also be considered the reason why another interpretation of the title of this thesis, "Tool Integration in Eclipse", is valid. This interpretation suggests that tool integration is in eclipse (as in eclipse of the sun).

Eclipse, a modern open source tool integration framework, is introduced in Chapter 3. The introduction includes the basic concepts and an architectural overview. Also, the rather original license model and the pros and cons of the architecture are briefly discussed. Chapter 4 considers Eclipse and the introduced Eclipse concepts in terms of the theoretical framework provided earlier. Furthermore, in Chapter 5, Eclipse is discussed from a technical point of view. This discussion considers Eclipse tool encapsulation and construction mechanisms, a brief glance into Eclipse design decisions, and the general guidelines for tool construction in Eclipse. Lastly, Chapter 6 provides a case study of tool integration in Eclipse. This case study is an example of how the Eclipse mechanics for tool encapsulation and construction can be used in practice. It also demonstrates how to achieve the different conceptual levels of integration discussed earlier by providing a real usable application that has been achieved with tool integration in Eclipse.

2 CASE Systems

Computer-aided software engineering (CASE) means automated support for the software engineering process [48]. Like computer-aided engineering and design tools that are used by engineers in other disciplines, CASE tools help to ensure that the quality is designed in before the product is built [44]. IEEE Standard Glossary of Software Engineering Terminology [35] describes CASE as follows:

computer-aided software engineering (CASE). The use of computers to aid in the software engineering process. May include the application of software tools to software design, requirements tracing, code production, testing, document generation, and other software engineering activities.

In this thesis, the term CASE is used in the above sense. However, CASE as a term is somewhat ambiguous. Gane [30] suggests that the distinguishing characteristic of a CASE product is that it builds itself a design database, at a higher level than code statements or physical data element definitions. This definition disqualifies tools such as compiler and debugger. The term "CASE system" is used in this thesis in the same sense as Sommerville [48] uses it, meaning anything from a single tool to an entire environment (discussed in Section 2.1.1).

This chapter discusses CASE systems on a general level, using a tool-centric approach. The chapter begins with an examination of CASE classification in Section 2.1. The integration itself is an essential subject when discussing any "integrated environments", which consist of various separate tools. To investigate integration, it is reasonable to first discuss how the separation of separate tools is achieved in a CASE system, and at what levels the separation occurs. This is done in Section 2.2. After that, integration can be investigated more clearly, which is done in Section 2.3, in terms of tool (or intertool) integration. Section 2.4 deals with key challenges of CASE systems.

Various standards of CASE systems, including tool interfacing and tool integration related ones, are not discussed. Also, the more philosophical issues, what problems CASE systems solve and what they do not solve, are not assessed.

2.1 CASE Classification

CASE classification is examined from two different viewpoints: the breadth of support for software process provided, and the relative level of CASE integration. The first one is presented by Fuggetta [24] and the second is shortly described by Pressman [44] and much referenced in the literature on the part of ICASE and IPSE. See for example [9], [10], [47], [60], and [55]. Fuggetta's classification is used later as a reference model.

2.1.1 Breadth of Support for Software Process

The breadth of support for software process offered by CASE technology can be used as a criterion for classification. This section is based on such classification provided by Fuggetta [24], which classifies CASE technology into tools, workbenches, and environments. As Fuggetta points out, an ideal classification should define an equivalence relation on the considered domain, to make it possible to partition the domain in equivalent classes and assign each element in the domain to just one class. However, this is often not possible with CASE products.

The software process can be thought of as consisting of two subprocesses: a *production process* and a *metaprocess*. Production process consists of the "actual" production activities, rules, and so on, which are often very concrete for a software developer. The metaprocess is used to define and systematically assess, evolve, and improve the production process. It is also used to acquire and exploit new products. The metaprocess can also be supported by CASE technology. This CASE classification considers products in both the production process and the metaprocess technologies.

Fuggetta describes the terms *task* and *activity* as follows: "A production process may be viewed as a set of elementary tasks to be accomplished to produce a software application. Examples of tasks are compiling, editing, and generating test cases from the requirements specification. Tasks are grouped to from activities, sets of tasks supporting coarse-grained parts of the software-production process. For example, coding is an activity that includes editing, compiling, debugging, and so on."

A CASE **tool** is a software component supporting a specific task in the software-production process. Tools can be stand-alone products or components of workbenches or environments. CASE tools are classified into seven classes, which are described below.

- *Editing tools* contain two subclasses: textual editors and graphical editors. Textual editors include text editors, word processors, etc. Graphical editors include various drawing, painting, and diagramming tools.
- *Programming tools* are subclassed into coding and debugging tools, code generators, and code restructurers. First subclass includes compilers, interpreters, debuggers, etc. Second subclass contains tools that generate code starting from a high-level description, and third subclass contains tools for restructuring existing programs.
- *Verification and validation tools* include static and dynamic analyzers, comparators, symbolic executors, emulators and simulators, correctness proof assistants, test-case generators and test-management tools.
- *Configuration management tools*are tools for version management, item identification, configuration building, change control, and library management.
- *Metrics and measurement tools* collect data on programs and program execution and include analyzers and monitors.
- *Project management tools*are tools such as cost-estimation, project-planning, conference desk, e-mail, bulletin board, project agenda, and project notebook tools.
- *Miscellaneous tools* consist of tools which are difficult to classify. For example, hypertext systems and spreadsheets.

A **Workbench** integrates in a single application several tools supporting specific software-process activities. A workbench achieves a homogeneous and consistent interface (presentation integration, see Section 2.3.1), easy invocation of tools and tool chains (control integration, see Section 2.3.3), and access to common data set, managed in a centralized way (data integration, see Section 2.3.2). Workbench classes are described below.

- *Business planning and modeling workbenches* are used to build high-level enterprise models to assess the general requirements and information flow, and identify priorities in the development of information systems. Tools that are integrated include diagram editors, report generators, cross-reference generators, etc.

- *Analysis and design workbenches* contain "upper" CASE tools used in the early stages of software process that automate analysis and design methodologies.
- *User-interface development workbenches* let the developer easily create and test user-interface components and integrate them with the application program.
- *Programming workbenches* contain the usual programming tools such as a text editor, a compiler, a linker, and a debugger.
- *Verification and validation workbenches,* including tools from metrics and measurement class and verification and validation class to jointly analyze the quality of code and support actual verification and validation.
- *Maintenance and reverse-engineering workbenches* include tools like a code restructurer, a flowcharter and a cross-reference generator.
- *Configuration management workbenches* integrate tools for version control, configuration building, and change control.
- *Project management workbenches* include tools for distributed agenda, memo-distribution, distributed to-do lists, meeting schedule, project planning, task-assignment, etc.

An **Environment** is a collection of tools and workbenches that support the software process. Environment classes are described below.

- *Toolkits* are loosely integrated collections of products easily extended by aggregating different tools and workbenches. Support of toolkits is often limited to programming, configuration management, and message handling in project management.
- *Language-centered environments* are centered around a specific language such as Lisp or Smalltalk. The environment is often written in the same language for which it was developed, letting the user customize and extend it.
- *Integrated environments* provide uniform, consistent, and coherent tool and workbench interfaces. They have a specialized database managing all information produced and accessed in the environment. Control integration is achieved through powerful mechanisms to invoke tools and workbenches from within other components of the environment. Integrated environments do not explicitly tackle process integration.

- *Fourth generation environments* are sets of tools and workbenches supporting the development of a specific class of program, e.g. electronic data processing and business-oriented applications.

- *Process-centered environments* are based on a formal definition of the software process. These usually handle process-model production and process-model execution. Process-centered environments can be thought as environment generators, since they can create different, customized environments that follow the procedures and policies enforced by the process model.

Specific tool, workbench, and environment classes, and to which specific class a given CASE system should be classified, is not so interesting in the context of this thesis. The point is to introduce the diversity of CASE systems and to describe the classification with some level of detail. The division of CASE technology into tools, workbenches, and environments itself is more interesting. Figure 2.1 illustrates this division.

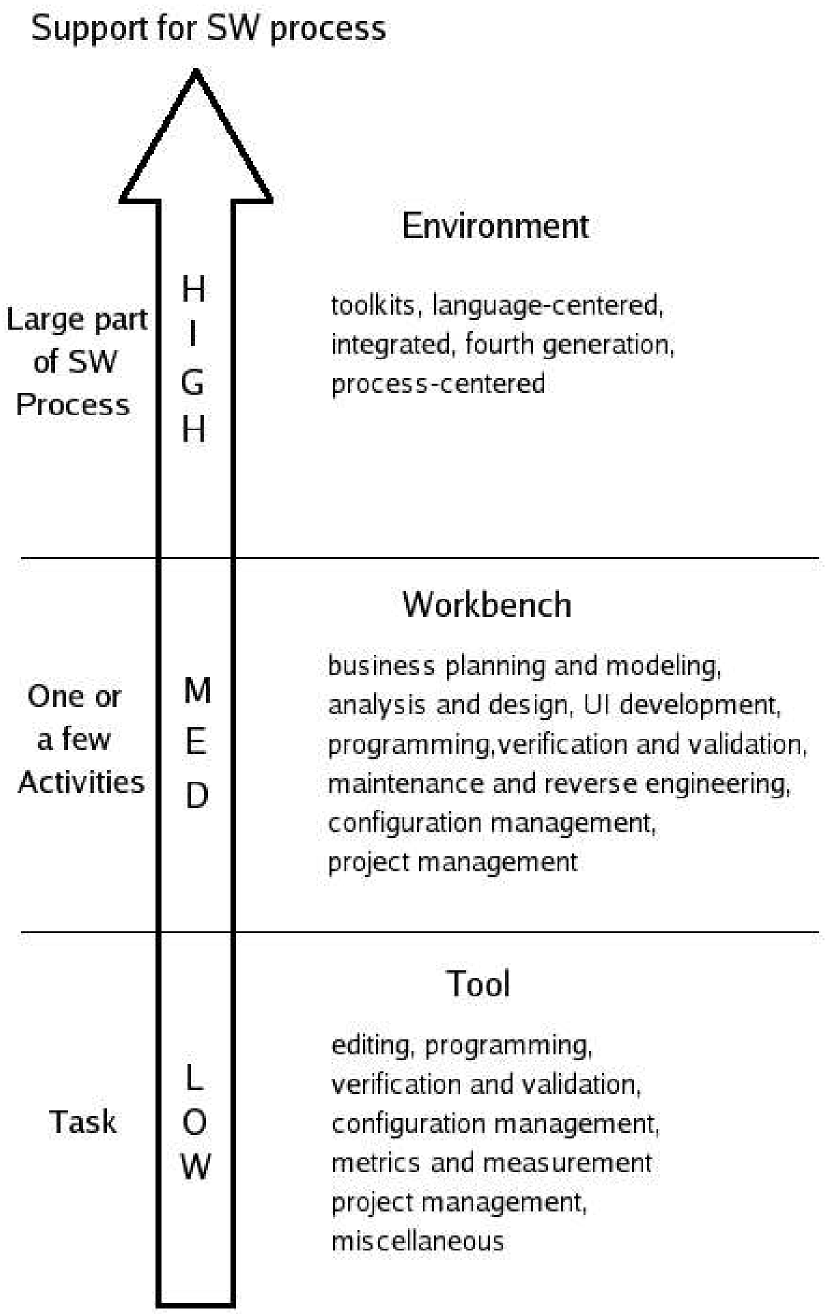

Support for SW process
Environment
toolkits, language-centered, integrated, fourth generation, process-centered
Large part of SW Process
HIGH
Workbench
business planning and modeling, analysis and design, UI development, programming,verification and validation, maintenance and reverse engineering, configuration management, project management
One or a few Activities
MED
Tool
editing, programming, verification and validation, configuration management, metrics and measurement project management, miscellaneous
Task
LOW

Figure 2.1: Illustration of Fuggetta's CASE classification.

2.1.2 Relative level of Integration

Pressman [44] uses four levels to describe CASE systems via the relative level of integration achieved. This is, however, not a complete classification in the sense of previous section.

1. **Invidual tool**. No integration.
2. **Tool bridges & partnership**. Invidual tools that provide facilities for data exchange. For example, standard data format for input/output.
3. **Single source integration**. Single CASE tool vendor integrates a number of different tools, usually in a closed architecture, and sells them as a package. Also called ICASE or I-CASE.
4. **IPSE** (Integrated Project Support Environment). IPSE creates standards for portability services and an integration framework (see Section 2.3.4, platform and framework integration) on top of the operating system and hardware platform. IPSE standards are used to build compatible tools with the IPSE and therefore compatible to one another.

There has been much juxtaposition between ICASE and IPSE. Sharon and Bell [47] explain the distinction between them:

- IPSEs are open, extensible environments and are intended to support multiple methods. ICASE environments support a single method and are not as readily extensible.
- IPSEs are targeted at teams working together on multiple projects. ICASE environments are designed primarily for teams working on a single project.
- IPSEs have evolved from scientific and engineering application development, ICASE environments are more common for information-system development.
- IPSEs are offered by single vendors with many integrated third-party components. ICASE environments are also sold by a single vendor, but most of the tool components are also from the same vendor.

Sharon and Bell also underscore that IPSE is a diverse environment including process management, project management, requirements management, configuration management, document management, repository, and project verification and validation components. Thus, there is a huge gap for open "environments" that can't reach the IPSE level, but are not single vendor products either. This can be a source of confusion, but as said, this is not a complete classification. If IPSE and ICASE are classified with Fuggetta's classification, IPSE would definitely classify as an environment. ICASE, on the other hand, could be just a workbench, e.g., for analysis and design, or an environment of some kind.

As terms, ICASE and IPSE have become at least somewhat outdated. By whom the CASE system is offered is not a very reasonable criterion for classification of CASE systems. At present, larger systems are very often composed of components offered by multiple vendors. Also, modern CASE systems can usually be extended almost without limits.

2.1.3 IDE Point of View

IDE stands for Integrated Development Environment. An IDE is generally a programming environment that has been packaged as an application program, typically consisting of a code editor, compiler, debugger, and graphical user interface (GUI) builder [54]. IDE might not be a scientific term, but it is widely used when discussing wide variety of programming or development "environments". An IDE is usually programming language specific, e.g. a Java development environment. IDE's aims are to increase its user's productivity and improve the quality of software produced. Achieving these goals is a sum of many things—different levels of tool integration discussed in Section 2.3 play an important role in it.

Using Fuggetta's CASE classification, IDE would be classified as a *programming workbench* in most cases, as IDEs do not usually support a large part of the software process activities. Usually commercial IDEs are closed source single-vendor applications, consisting of CASE products of one organization (ICASE). Non-commercial IDEs are often open source, but they are mainly not used by the masses, because effective usage tends to require more technical knowledge. An example of a non-commercial IDE of that nature is Emacs[1]. D'Anjou et. al [12] point out a problem

[1]Emacs manual says: "Emacs is the extensible, customizable, self-documenting real-time display editor.". Emacs has extensions for a wide variety of programming languages and many other "IDE-features".

concerning the production of IDEs:

> "It is no longer practical for vendors to produce the base application and IDE infrastructure upon which to deliver their product-specific functionality. Not only is it a wasteful use of programmer time, but it also has led to islands of tools and applications that are disjointed and inconsistent."

2.1.4 Discussion

So far, we have described the terms CASE and CASE system, examined CASE classification from two different viewpoints, and discussed how the term IDE relates to these classifications. However, the two different points of view to CASE classification discussed are not the only ones out there. For example, Wallnau and Feiler [55] discuss CASE coalition and CASE federation environments as evolutionary successors of IPSE. CASE coalition environments are characterized by ad hoc, control-oriented integration and CASE federation environments are characterized by flexible, services oriented integration (integration is discussed in Section 2.3).

In addition, there are terms like SEE (Software Engineering Environment) [8], SDE (Software Development Environment) [58], and a dozen others, not to forget different interpretations of them. The terminology has without doubt become an obstacle of some sort. Ironically, a non-scientific term like IDE seems to be a term with the clearest meaning, at least for a nonprofessional software engineer. It is something everybody with some software engineering experience is familiar with, and as a term it contains few built-in limits or requirements for its features, behaviour, or mechanics.

2.2 Separation of Concerns - Tool Interfacing

Fuggetta's classification (see Section 2.1.1) provides the following separate concerns: tools, workbenches, and environments. In this section, it is examined how the separation itself is carried out in a (host) CASE system consisting of multiple tools. This leads us to tool interfacing[2]. Pressman uses the terms tools management services and tools layer [44]. *Conformity* means that the software must conform to already existing interfaces and human institutions [7]. Our discussion on tool interfacing concerns one type of conformity, the conformity between the separate parts of a CASE system. However, one should note that the separation used in this section is only one way to achieve separation of concerns. A different separation could be achieved by using different kind of concern, e.g. a function of a CASE system.

Cohesion and coupling are often discussed when dividing a program into modules. They are also relevant on another (higher) level—when interfacing and integrating tools in a CASE system. Cohesion is not so much an issue on the tool level, as a tool usually supports a specific task, but when discussing cohesion of tools in a workbench or environment, it could be an issue. Coupling, on the other hand, is essential to tool interfacing. Yang and Han [58] use the level of coupling as a defining characteristic of tool interfacing. They describe tool interfacing in SDE (Software Development Environment) with a three level classification: uncoupled, tightly-coupled, and loosely-coupled interfacing paradigms. These are examined next. Figures 2.2, 2.3, and 2.4 describe tool interfacing using a conceptual architecture where there's one generic front-end, multiple back-end tools, and a generic interface between front-end and back-ends. These figures have been redrawn from [59].

2.2.1 Interfacing paradigms

In the uncoupled interfacing paradigm, all communication among tools is via operating system facilities and each tool has its separately executable code. This class of tool interfacing reduces the need for user input between tool activations and allows the feedback from the tools to be associated with the relevant sections of the input. See Figure 2.2.

In the tightly-coupled interfacing paradigm, all communication among tools is

[2]When tool interfacing is discussed in this section, discussion can also include interfacing workbenches when appropriate.

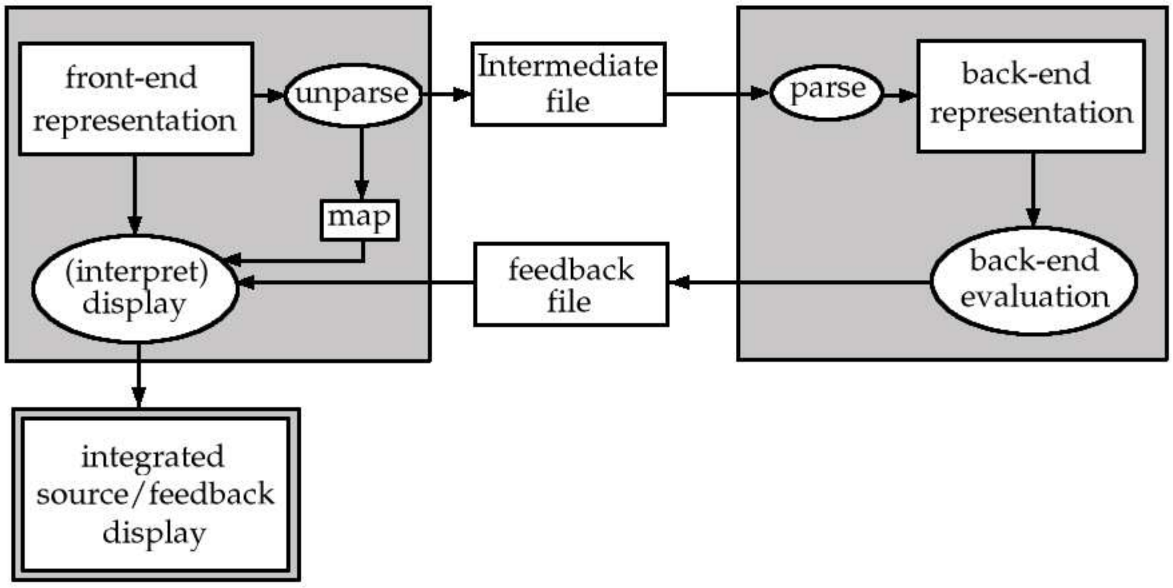


Figure 2.2: Information flow in the uncoupled interfacing paradigm [59].

computer generated and the tools are combined in a single system by linking compatibly compiled modules. In addition to the benefits of the previous class, tightly-coupled interfacing improves productivity by eliminating tool-generated delay, by exploiting techniques such as incremental data processing and concurrent execution of tools and, hence, by reducing the user's "thinking time". See Figure 2.3.

In the loosely-coupled interfacing paradigm, tools communicate in some manner smart enough to achieve user satisfaction, but each tool has sufficient separation to maintain system flexibility, e.g. by using database, object base, or message passing paradigm (data and control integration, see Sections 2.3.2 and 2.3.3). This is illustrated in Figure 2.4. [58]

2.2.2 Discussion

Yang et al. [59] discuss supporting multiple tool integration paradigms within a single environment, and conclude that in most SDEs tools are integrated via a tool interface based on a single interfacing paradigm. Whereas, Sharon and Bell [47] present a model of levels of tool integration, where single-vendor tool integration (ICASE) has some tightly integrated core tools (tightly-coupled interfacing paradigm) and another vendor's tools are integrated via a separating interface (uncoupled or loosely

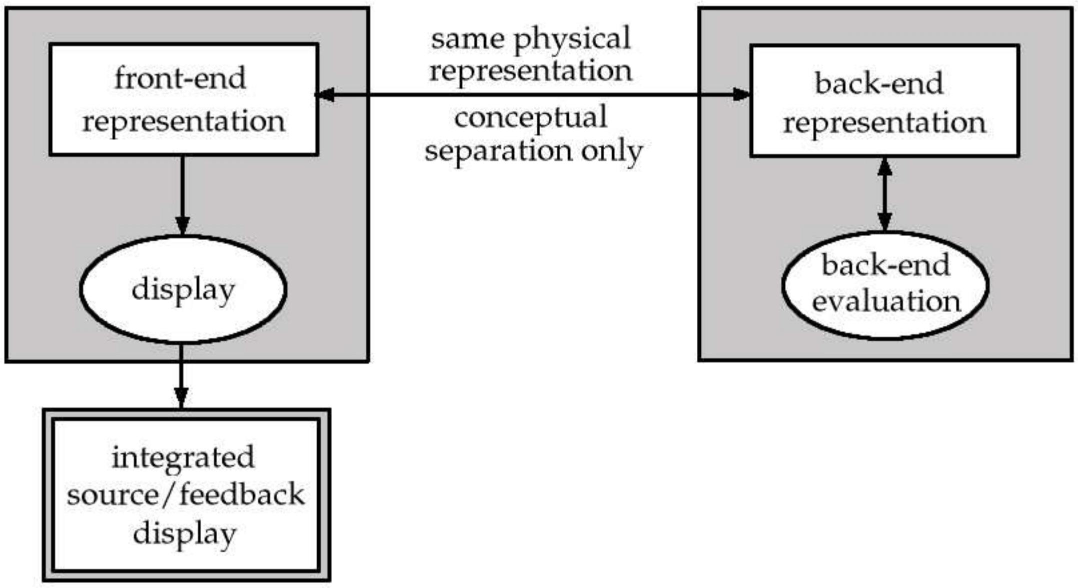

front-end representation
same physical representation
conceptual separation only
back-end representation
display
back-end evaluation
integrated source/feedback display

Figure 2.3: Information flow in the tightly-coupled interfacing paradigm [59].

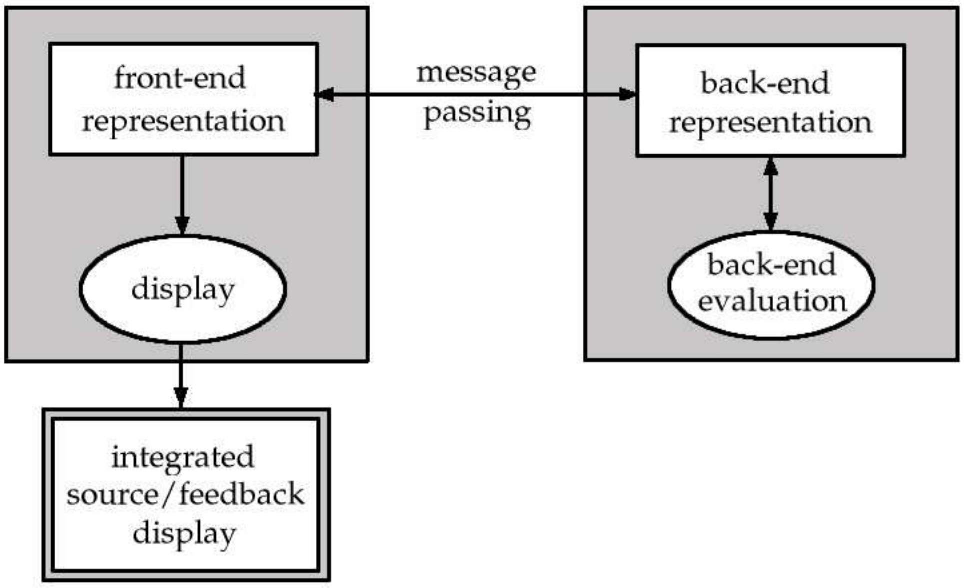


Figure 2.4: Information flow in the loosely-coupled interfacing paradigm [59].

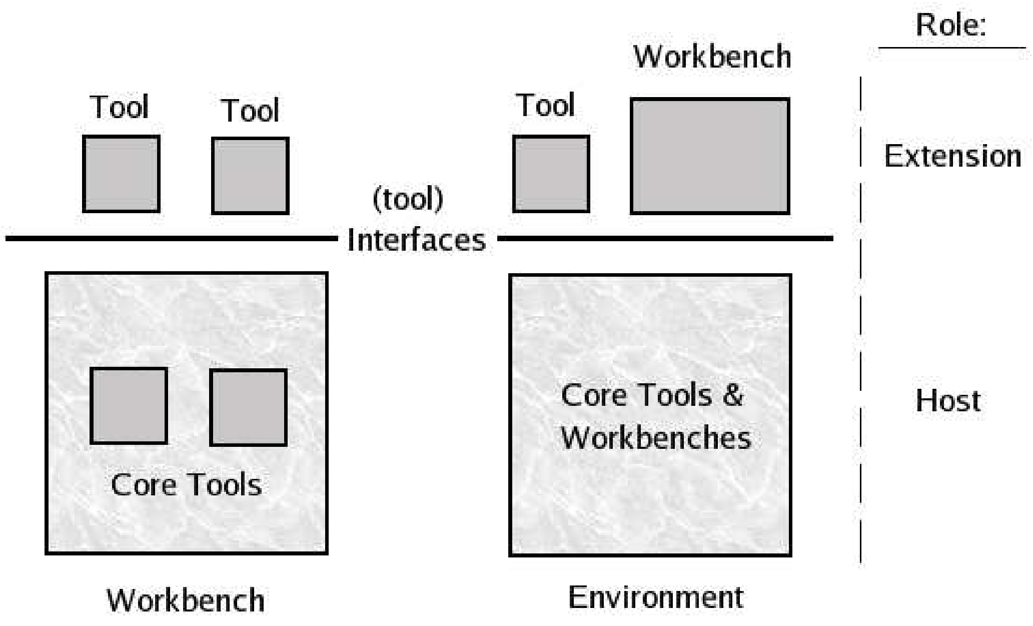


Figure 2.5: CASE system interfacing and roles.

coupled interfacing paradigm). In this model, IPSE uses only loosely-coupled interfacing paradigm with all the tools on the same side of repository manager and integration facilities (the tool interface). Figure 2.5 presents a simplified illustration of the possible roles of a CASE system, using Fuggetta's CASE classification. Simple tool-bridge interconnections between CASE systems are not considered. The roles are:

- *Host*. This is the hosting CASE system which defines the agreements or mechanics of integration—the infrastructure. A host can be an environment or a workbench and it includes the core tools and workbenches which are connected to it using the tightly-coupled interfacing paradigm.
- *Extension*. These tools all share the same interface with the host, each tool with a workbench, or each tool or workbench with an environment (see Figure 2.5). Extensions are connected using uncoupled or loosely-coupled interfacing paradigm.

2.3 Integration of Concerns - Levels of Tool Integration

"Any separation of concerns mechanism must include powerful integration mechanisms, to permit the selective integration of the separate concerns." [33]

Integration includes front-end integration, which consists of presentation or user-interface integration, and back-end integration, which consists of data and control integration [59]. These are presented in the light of properties provided by Thomas and Nejmeh [52], which characterize the various integration relationships between tools. A matter to note is that Thomas and Nejmeh consider binary relationships, relationships between two tools, and believe that the properties in integration of many tools can be derived as aggregate properties. Presentation, data, and control integration are conceptual levels of integration, and they do not describe the actual mechanics of integration, which are left mostly out of consideration.

Brown et al. [9] say that there are two distinct approaches to integration. First is based on providing a common infrastructure in which tools can be embedded (framework integration, see 2.3.4), the other concentrating on the tools themselves. They refer to the first as IPSE approach and latter as CASE approach.

Different types of integration are often encapsulated into services such as user interface services, data integration services, and message server services. CASE system can then provide invidual (tool) interfaces for particular services, and tools can integrate with services relevant to their needs. [55]

2.3.1 Presentation Integration

The aim of presentation integration is to reduce user's cognitive load by letting users interact with tools consistently. This makes new tools much easier to learn and leads to reduced training and support costs. Presentation integration deals almost exclusively with window-based graphical user interfaces nowadays, as text-based user interfaces are becoming rare. Window-based tools have four levels to encapsulate presentation integration: the window system, the window manager, the user-interface-development tool kit, and the look-and-feel guidelines. User interface also has to meet the user's response time expectations and ensure that the information it presents to user is useful and correct. [11, 52, 55, 61]

Essential properties of presentation integration, and the questions they answer, are [52]:

- *Appearance and behaviour*, "How easy is it to interact with one tool, having already learned to interact with the other?"
- *Interaction paradigm*, "How easy is it to interact with one tool having already learned the interaction paradigm of the other?"

2.3.2 Data Integration

Data integration means that tools can transfer information, and keep the information consistent. This data includes persistent and nonpersistent data. Nonpersistent data is information that does not survive the execution of the tools that are sharing and exchanging it. Information sharing methods between tools can be divided into four categories: direct, file-based, communication-based, and repository-based. Sharing metadata between tools can be used to achieve a higher degree of data integration. [11, 52]

The properties of data integration and questions they answer, given by Thomas and Nejmeh [52], are :

- *Interoperability*, "How much work must be done to make the data used by one tool manipulable by the other?"
- *Nonredundancy*, "How much data managed by a tool is duplicated in or can be derived from the data managed by the other?"
- *Data consistency*, "How well do the tools indicate the actions they perform on data that is subject to some semantic constraint so that other parts of the environment can act appropriately?"
- *Data exchange*, "How much work must be done to make the data generated by one tool usable by the other?"
- *Synchronization*, "How well does a tool communicate changes it makes to the values of nonpersistent, common data so that other tools it is cooperating with may synchronize their values for the data?"

2.3.3 Control Integration

Control integration is concerned with providing mechanisms for one tool to control the activation of other tools in the CASE system. Ideally, all the functions offered by all the tools in an environment should be accessible (as appropriate) to all other tools, and the tools that provide functions need not know what tools will be constructed to use their functions. Mechanisms for control integration include methods such as explicit message passing, time- or access-activated triggers, and message servers. [11, 48, 52]

Essential properties of control integration, and the questions they answer are [52]:

- *Provision*, "To what extent are a tool's services used by other tools in the environment?"
- *Use*, "To what extent does a tool use the services provided by other tools in the environment?"

2.3.4 Other Levels of Integration

Integration occurs also on some other levels with CASE systems. These levels can be considered as concerning the CASE system as a whole, and not so much invidual tools themselves. This is not intended to be an exhaustive list, but it describes shortly some commonly discussed levels of integration:

- **Framework Integration.** "The integration framework is a collection of specialized programs that enables invidual CASE tools to communicate with one another, to create project data base, to exhibit the same look and feel to the end-user (the software engineer)." [44] Tool integration frameworks define the mechanics for the conceptual levels of integration discussed earlier (presentation, data, and control). Frameworks define how these levels of integration are provided and utilized. This is achieved partially by providing means to register and encapsulate tools. Encapsulation establishes a "wrapper" that presents the essential aspects of the tool that is to be integrated [47].
- **Platform Integration.** A CASE system is built on hardware platform and operating system facilities. Platform integration is concerned with integrating the CASE system in a reasonable way to these facilities. This usually demands the use of portability services. Pressman [44] describes portability services as follows: "A set of portability services provides a bridge between CASE tools and

their integration framework and the environment architecture. Portability services allow CASE tools and their integration framework to migrate across different hardware platforms and operating systems without significant adaptive maintenance." Platform integration also includes migrating between different versions of the same operating system.

- **Process Integration**. Process integration means that the CASE system has embedded knowledge about the (software) process activities, their phasing, their constraints, and the tools needed to support these activities [48]. Aim of process integration is to make the conceptual task-execution chain explicit, flexible, and reusable; in a process-driven environment (or process-centered environment, described in Section 2.1.1), the process integration is achieved by a formal software process model and a process driver that interprets and executes it [39].
- **Organizational Integration**. A CASE system also has to work on an organizational context for it to be effective. Organizational integration can be divided into three levels; first and the highest level is enterprise level, which includes planning how development processes should be managed; second level is project level, which includes project and process management, impact analysis and change management, documentation, and reuse; lowest level is team and invidual level, which includes the execution of software process (see above) [11]. The lifecycle of a CASE system usually includes the introduction of a CASE system into an organization, customization, removal, and many activities in between. All these activities can also be thought of as part of organizational integration.

2.3.5 Discussion

Brown and McDermid [10] have a different way to look at tool integration than the one presented earlier. They divide tool integration into five levels, from low to high: carrier level, lexical level, syntactic level, semantic level, and method level. These levels progress from low to high in their ability to record, share, and transfer information among tools. *Carrier level* integration provides a common view on the data, like a byte stream on Unix, but each tool must analyze and process its entire input. On *lexical level* tools share data formats and operating conventions, but the conventions are embedded in the tools and tools don't understand operations carried out by other tools. On *syntactic level* tools agree on the rules governing formation of data structures and avoid repeating actions of analyzing, validating and converting them. On *semantic level* tools agree on the data structure definitions, as well as the meanings of the operations on those structures. On *method level* tools agree not only on the data structures and operations, but also the specific development process (process integration). [8, 10]

Viewers in different roles have different points of view to integration. Thomas and Nejmeh [52] describe the environment user's and the environment builder's points of view. User is concerned with the perceived integration, which is a seamless collection of tools, hopefully. The builder is concerned with the feasibility and effort needed to achieve this integration. Wallnau and Feiler [55] also mention these and add the vendor's perspective, who is looking for platform availability of integration mechanics. All these perspectives are important and must be assessed when designing a CASE system.

Tool integration (and interfacing) in CASE systems has some similarities with EAI (Enterprise Application Integration). Linthicum [37] describes EAI as follows: "Put briefly, EAI is the unrestricted sharing of data and business processes among *any* connected applications and data sources in the enterprise." He also continues: "The demand of the enterprise is to share data and processes *without* having to make sweeping changes to the applications or data structures. Only by creating a method of accomplishing this integration can EAI be both functional and cost effective." Although EAI may use the same mechanics of integration that are used to integrate tools in a CASE systems, and the integration occurs on the same conceptual levels (presentation, data, and control), the goals are different. As EAI tries to share data and processes between applications without having to make sweeping changes to them, the goal of tool integration in CASE systems is to produce a coherent "environment", a single application, which utilizes tools seamlessly and in a consistent manner, so that no tool it integrates has to be used as a separate application. Also, the scope of tool integration in CASE systems is smaller than in EAI, and concerns mostly CASE tools.

Brown et al. [8] discuss integration and reuse, and argue that these "twin goals" are fundamentally in conflict. They say that the tighter the integration between tools, the harder the tools are to be reused in another environment. This is an interesting point, as tight integration tends to make a tool specific to a particular context. This thought leads us to the next section.

2.4 Key Challenges

Harrison et al. [33] discuss the key challenges of software engineering tools and environments. This section is based on their discussion. Software engineering tools and environments in their discussion are considered equivalent to CASE systems. Section 2.4.4 provides some discussion on the matters examined.

The need for integration has been the driving force for new lines of research, and integration has been accomplished in many ways (integration was discussed in Section 2.3). Also, a vast collection of tools have been prototyped or marketed, some in the context of environments, some as stand-alone applications. But the state-of-the-practice in software engineering has not advanced as much as the tools and integration mechanisms would imply. The key reason for this is that each tool or environment is highly specific to some context—it might require the software it manipulates to be written in a particular language, or it might run only on a particular hardware or operating system, etc. Context-specific software is a common problem already mentioned with IDEs (see Section 2.1.3). A major challenge is to find ways to build and integrate tools so that they, or capabilities within them, can easily be adapted for use in new contexts.

Figure 2.6 illustrates the current relationship of software and change. Change forces software to integrate and interact with other pieces of software, and to evolve and adapt to new contexts, e.g. new hardware, business practices, etc. Brooks [7] explains the cause of change as follows: "In short, the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product." Current technology offers only an opportunity to anticipate and pre-plan for change, by building "open points" using techniques such as frameworks and design patterns [29]. However, it is impossible to anticipate all future needs, and the pressure to market and publish in many areas of business is too great to allow careful pre-planning and, hence, stable reuse.

2.4.1 Permanently Malleable Software

> "In a nutshell, software is currently like *clay*, and it needs to become like *gold*. Clay is soft and malleable initially, but then it hardens. After that, bumps can be added to it, but it cannot be changed or reshaped without breakage. Attempts to force a hardened clay peg into a hole of a different

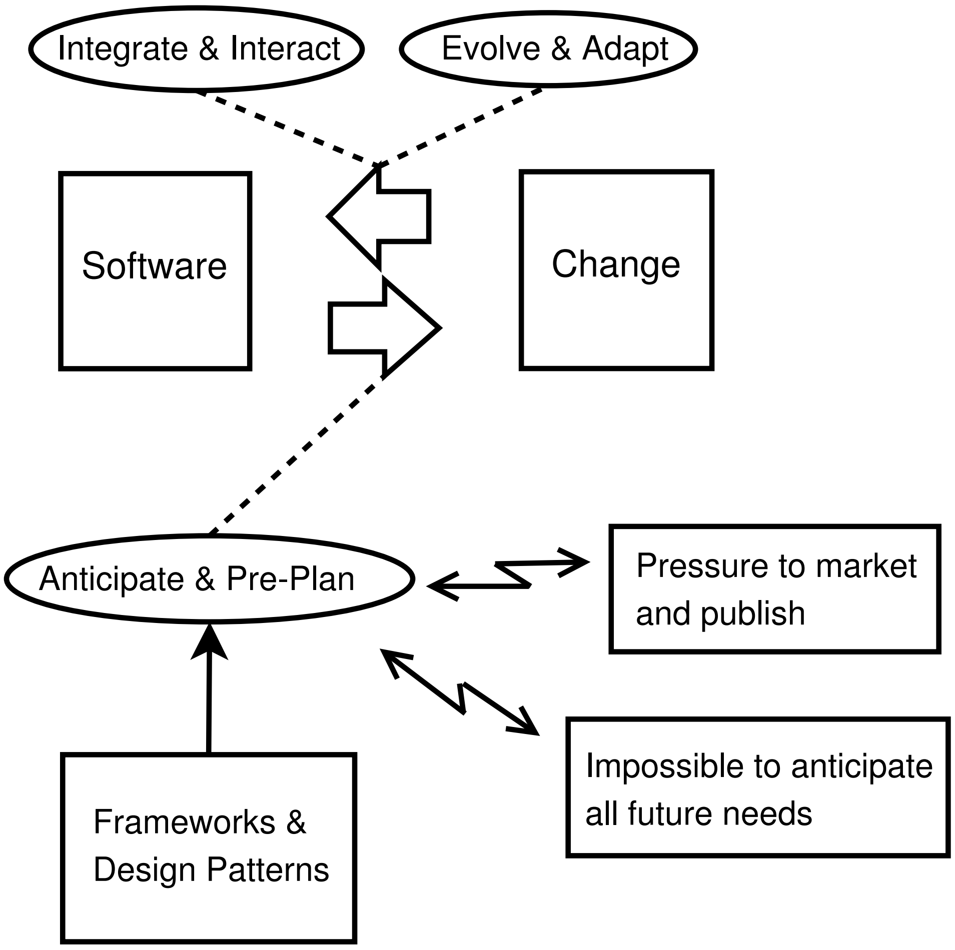

Integrate & Interact
Evolve & Adapt
Software
Change
Anticipate & Pre-Plan
Pressure to market and publish
Impossible to anticipate all future needs
Frameworks & Design Patterns

Figure 2.6: Current relationship of software and change [33].

> shape are likely to lead to breakage. Gold, on the other hand, remains malleable for life. It can be reshaped as needed, and will assume the shape of a hole into which it is hammered." [33]

Harrison et al. [33] also introduce the term *morphogenic*[3] *software* which refers to software that is malleable for life: sufficiently adaptable to allow context mismatch to be overcome with acceptable effort, repeatedly, as new, unancticipated contexts arise. Morphogenic software requires a commitment to integration, but not to any particular approach.

> "In other words, the challenge is to find ways to write software, and tools to manipulate software, that will facilitate both rapid initial development and later adaptation and integration in new contexts, without up-front knowledge of what those contexts will be." [33]

Brooks [7] says that all successful software gets changed, and as software is pure thought-stuff, it is infinitely malleable. However, making software adaptable with acceptable effort—or morphogenic, can be difficult.

2.4.2 Separation and Integration of Concerns

Major barrier for morphogenic software and software altogether is inadequate separation of concerns. Traditionally, separation is carried out through one dimension, e.g., functional or data decomposition. However, as described earlier, it should be carried out using multiple (often unorthogonal or overlapping) dimensions, including, for example, feature, aspect, role, viewpoint, and unit of change. This is called *multi-dimensional separation of concerns* [51]. Some concerns can be identified early in the development, some are identified later. As was mentioned in Section 2.3, after separation the separate concerns must also be integrated in a meaningful way. Fundamental requirements for morphogenesis are to identify the concerns in the software, to be able extract them as components, and to later integrate them into new contexts.

[3]"morphogenic" is derived from "morphogenesis" which means the development of form and structure in an organism during its growth from embryo to adult." [Collins English Dictionary, 5th Edition]

2.4.3 Meeting the Challenges

CASE systems play a crucial role in meeting the challenges discussed above. Multi-dimensional separation of concerns requires support of CASE system to make the separation an ongoing and consistent process of identifying new concerns, encapsulating them using modularization mechanisms, and finally integrating them by matching and reconciliating. Also, visualization of concerns and their relationships is essential for understanding, and for developers to be able to focus on particular concerns relevant for a specific situation.

"Fully achieving the goal of multi-dimensional separation of concerns will require new research and the synthesis of existing research in a variety of software engineering and other areas. These include software architecture, refactoring and reengineering, component based software engineering, software analysis, software specification, methodologies, programming (and other) languages, HCI, and visualization." [33]

CASE systems can also provide other kinds of support for engineering of morphogenic software. This includes managing dependencies and interactions, adaptation, and correctness and consistency management. Support for morphogenic software in CASE systems should cause minimum intrusion to development of software, or the market pressures will render it unacceptable. Adaptation is necessary after the software is extracted from its context. Software has to meet new needs in new contexts before it can be integrated. After it has been integrated, diverse checking and testing must be performed to ensure that the result is correct and consistent. All these tasks need desperately support from CASE systems.

2.4.4 Discussion

The concept of morphogenic software is relevant to CASE systems for two reasons. Firstly, tight integration of tools into a CASE system, which is a key theme with CASE systems, tends to make tools specific to the (host) CASE system in question. If a tool is to be used in a different CASE system, a new encapsulation "by hand" is needed with current technology. The new encapsulation might not even be possible without reconstructing the tool from scratch. Morphogenic extraction and adaptation mechanisms could be used instead. Secondly, support of CASE systems is needed in building of morphogenic software. A key challenge essential to morphogenic software, the multi-dimensional separation of concerns, and especially managing the multiple dimensions, their dependpencies, etc., can not be done without support of CASE systems.

An example of a programming technique that takes multi-dimensional separation of concerns into practice is AOSD (Aspect-Oriented Software Development) [22]. AOSD increases the expressiveness of object-oriented programming by providing mechanisms for simplifying and localizing the expressions of crosscutting concerns, working towards simpler system evolution, more comprehensive systems, adaptability, customizability, and easier reuse.

2.5 Summary

In this chapter, CASE systems were reviewed from various points of view. First, CASE classification was discussed from two different viewpoints, breadth of support for software development and relative level of integration, and the former classification was used as a reference framework. This classification provided us with the separate concerns: tools, workbenches, and environments, or the identification of concerns. Also, it was discussed how the term IDE relates to CASE classification. A separation of concerns in CASE systems was discussed, using a three-level classification of tool interfacing, which divides tool interfacing into uncoupled, tightly-coupled, and loosely-coupled levels. This was considered in respect of the CASE classification. After that, integration of the separate concerns, the "Holy Grail" of CASE technology, was discussed in terms of conceptual levels of tool integration: presentation, data, and control. Also, other research-based views to integration were discussed, with issues concerning integration in general. And finally, key challenges of CASE systems were discussed, which included achieving permanently malleable software and multi-dimensional separation of concerns, which are in fact challenges for software altogether, but need support from CASE systems.

This chapter has also been an introduction to the complexity of CASE systems. Complexity is an issue that concerns all software [7], but especially CASE systems as they can consist of several integrated tools. A role of a CASE system is to help software developers see relevant matters of software more clearly in a particular development situation. Also for clarity's sake, this discussion has not been tied to the dozens of standards and standardization attempts that have (in a way) burdened CASE systems throughout their history.

3 An Introduction to Eclipse

This chapter provides an introduction to Eclipse, including an explanation of the basic concepts, an architectural overview, and a description of the license model.

Eclipse is often identified as three different things [12]:

- **an integration platform for tools and applications**. Eclipse's plug-in[1] based framework provides an easy way for developers to construct and utilize their software tools in a cohesive way and provide them with a consistent look and feel. Eclipse platform works on various operating systems contributing a robust set of different application development tools. Eclipse platform was originally produced to solve the lack of interoperability among tools.

- **an open source community.** "Eclipse Foundation is non-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services." [18] Eclipse.org is the website of the Eclipse Foundation, and it contains information about Eclipse projects and various additional issues, including technical articles written by Eclipse developers. It was formed in November 2001 by Borland, IBM, MERANT, QNX Software Systems, Rational Software, Red Hat, SuSE, TogetherSoft, and Webgain. Later eclipse.org has been joined by many more, and now it consists of over a hundred member companies.

- **Java development environment.** Eclipse is also an efficient Java development environment (Java IDE). The Java development side is called Java Development Tooling (JDT). It contains tools for editing, debugging, refactoring, and much more. It also has integrated support for Apache Ant tool and unit testing with JUnit. One highly useful feature for developers is that Eclipse has advanced search facilities that allow the developer to search through the source code. Eclipse also ships with its own source code.

For our purposes the first is the most interesting, so from hereafter Eclipse is mostly considered as an integration platform. However, Eclipse is much more than just

[1]plug-in is basically an extension, see Section 2.2.2.

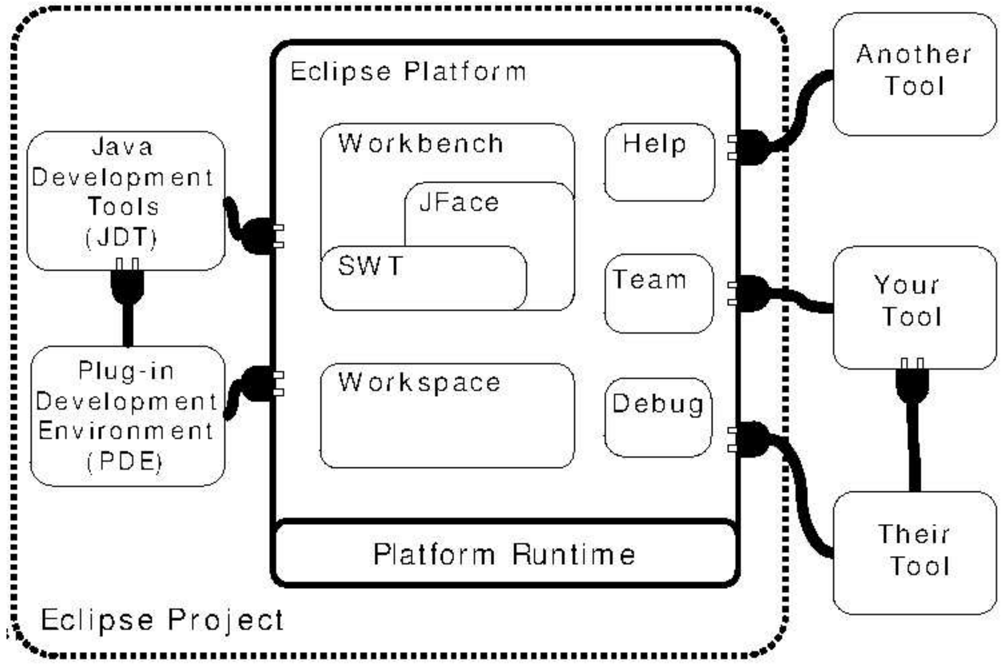
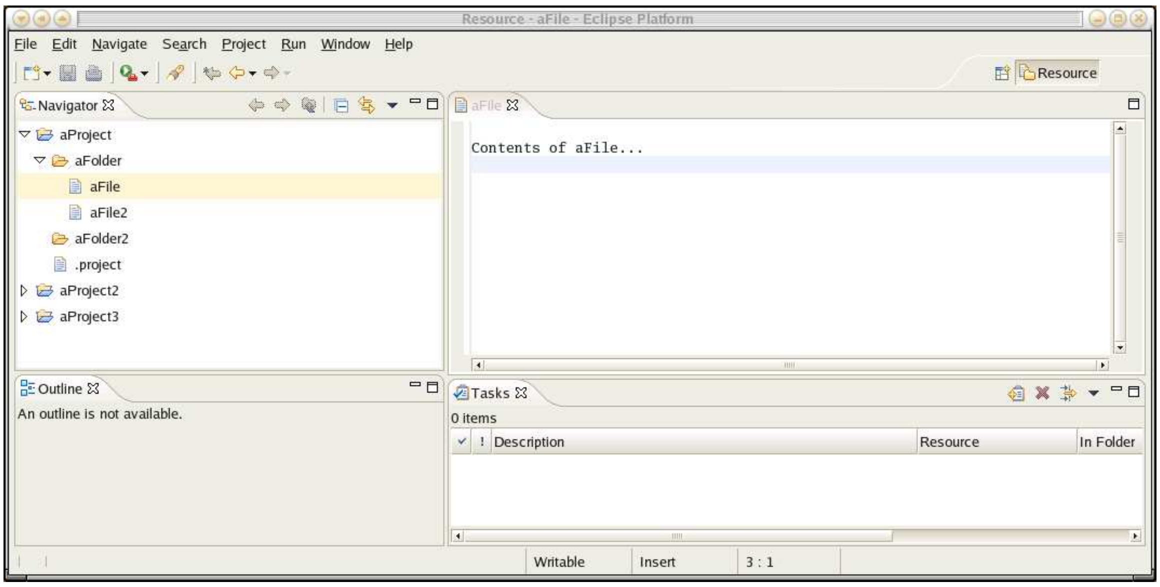


Figure 3.1: Eclipse Platform and Eclipse Project [34].

these three things. Eclipse is language-independent in the sense that IDEs for various languages can be, and have been built on Eclipse. Eclipse is also used for web technologies, modeling, version control, and various other things. Eclipse is not tied down to just extending the platform capabilities, as parts of Eclipse platform can be used to produce stand-alone applications. These are called Rich Client Platform (RCP) applications. [12, 18, 42]

The development of Eclipse is divided into projects which are hosted by the Eclipse Foundation. The platform itself is developed in the Eclipse Project, which also contains subprojects for JDT and PDE (Plug-in Development Environment). These three pieces form the Eclipse SDK (Software Developer's Kit), a complete environment for developing Eclipse tools. Coarse-grained components of Eclipse platform and Eclipse Project are depicted in Figure 3.1.



```
Contents of aFile...
```

An outline is not available.

0 items

✓	!	Description	Resource	In Folder

Figure 3.2: Eclipse Resource Perspective

3.1 Basic Concepts and Facilities

This section describes basic concepts and facilities of Eclipse Platform.

3.1.1 Workspace and Resources

Workspace is the "root" container of resources that the user, or the tools, interact with when using Eclipse. Workspace can contain only projects on the root level, and Eclipse uses a single workspace when running. *Projects* are top-level containers under the workspace which map into user-specified directories in the file system and contain the necessary files and folders. Projects, folders, and files are called *resources*.

3.1.2 Workbench User Interface

Workbench is the user interface of Eclipse which provides a consistent front-end to Eclipse tools. From user's point of view workbench and Eclipse are basically the same thing. Main components of workbench are editors, views, and perspectives.

- Editor is used to edit or browse resources.

- View is a visual component used mainly to navigate the hierarchy of information and to open editors, e.g. a tree view.

- Perspective is a group of views and editors in the workbench, a meaningful configuration for a specific purpose, e.g. Java development.

Figure 3.2 is a screenshot of the Eclipse resource perspective, a perspective for basic management of resources. The largest (top right) window is the editor area and the other three windows are different views. All four form the resource perspective in its default configuration. Perspectives can be customized to user's needs, for example by adding additional views.

Note: Workbench in Eclipse refers to the user-interface, whereas workbench discussed when examining CASE classification in Section 2.1.1 referred to several tools integrated in a single application.

3.1.3 Facilities

Eclipse platform provides various facilities in addition to the base platform, consistent user interface, and resource model. These are described below.

- **Ant**. Eclipse/Ant integration. Ant is a Java based build tool.

- **Compare**. Universal compare facility. For example, provides support for comparing and merging resources with local history.

- **CVS**. Platform CVS Integration. A pure Java client for CVS, which is tightly integrated into Eclipse.

- **Debug**. Generic execution debug framework. Provides language independent facilities for launching programs, defining and registering breakpoints, a language independent debug model and UI, etc.

- **Help**. Platform help system. Provides mechanisms for handling online documentation using native browser of the operating system.

- **Search**. Integrated search facility. Defines an extensible infrastructure for providing search functionality to the workbench.

- **Text**. Text editor framework. Provides basic building blocks for text and text editors.

- **Update**. Dynamic Update/Install/Field Service. The Update Manager of Eclipse, see Section 3.2.1.

- **Team**. Generic team support framework. Provides mechanisms for repository tools to integrate functionality of their repository solution into Eclipse.

These are all managed as separate components of the Platform subproject.[18]

3.2 Architecture

> "The architecture of Eclipse, like its development, is community oriented. A plug-in is a unit of functionality in a neighborhood of other plug-ins. Like a real community, one bad neighbor affects all of its residents." [36]

Eclipse's architecture differs somewhat from a typical extensible IDE. Eclipse was built with architectural top priorities in extensibility and integration. In Eclipse, there are no core tools in the platform itself, all tools are implemented as extensions, including the graphical user interface. Birsan [4] calls an architecture like this a *pure plug-in architecture*. In Eclipse, the only component that has to be always loaded is the run-time kernel (or Platform Runtime), which handles the activating of extensions equally. In a typical extensible IDE, there are tightly integrated or "built-in" core tools in the IDE in addition to extensions. Figure 3.3 illustrates the difference. [12, 42]

The Eclipse platform runs on a single Java virtual machine, and is built using object oriented architecture. The extensions in Eclipse are called plug-ins, and basically a plug-in consists of a declarative manifest file and Java JAR-files, which contain the functionality. With the manifest, Eclipse makes a distinction between public and published interfaces. Plug-ins explicitly declare their "open points", where other plug-ins can connect to, and they must do so by using uniform mechanics. Without uniform mechanics, it would be unlikely that Eclipse could function as a coherent whole, as every public method would be a potential point of extension. Plug-ins are examined more thoroughly in Chapter 5. [23, 26]

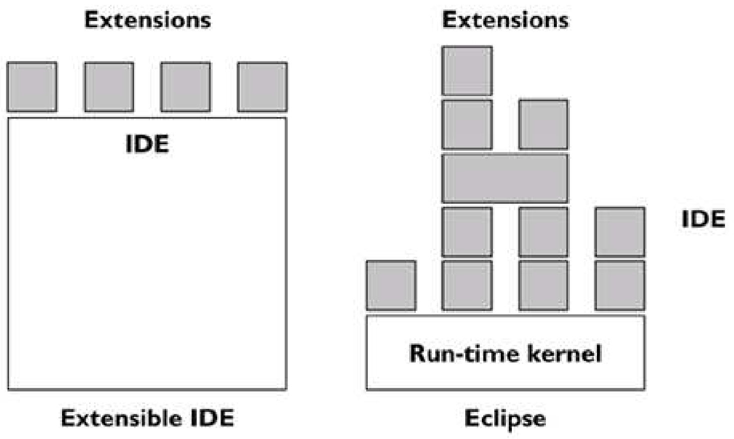


Figure 3.3: Typical extensible IDE architecture vs. Eclipse architecture [26].

3.2.1 Packaging

Eclipse has three kinds of installable components: plug-ins, fragments, and features. The basic component is plug-in, which is the smallest unit of function. Fragments are parts which provide additional functionality or content to existing plug-ins, like tongue or operating system dependent behaviour. Plug-ins and fragments are packaged into deliverable and maintainable units called features, which are managed by the Update Manager of Eclipse. Update Manager is used to install new features from update sites which can reside on web locations or on local filesystems. Update Manager installs features into an install location and Eclipse can use features from multiple install locations when run. Plug-ins (separate from features) can be installed into Eclipse without the use of Update Manager, but this can easily lead to problems with updating and uninstalling. Update Manager and features are the answer to efficient package management mechanics demanded by Eclipse's pure plug-in architecture. Components of package management and their relationships are illustrated in Figure 3.4.

3.2.2 UI Frameworks

Workbench is built using two frameworks, SWT and JFace, which are the means to provide presentation integration into Eclipse.

- **SWT** stands for Standard Widget Toolkit. Eclipse does not build its user-

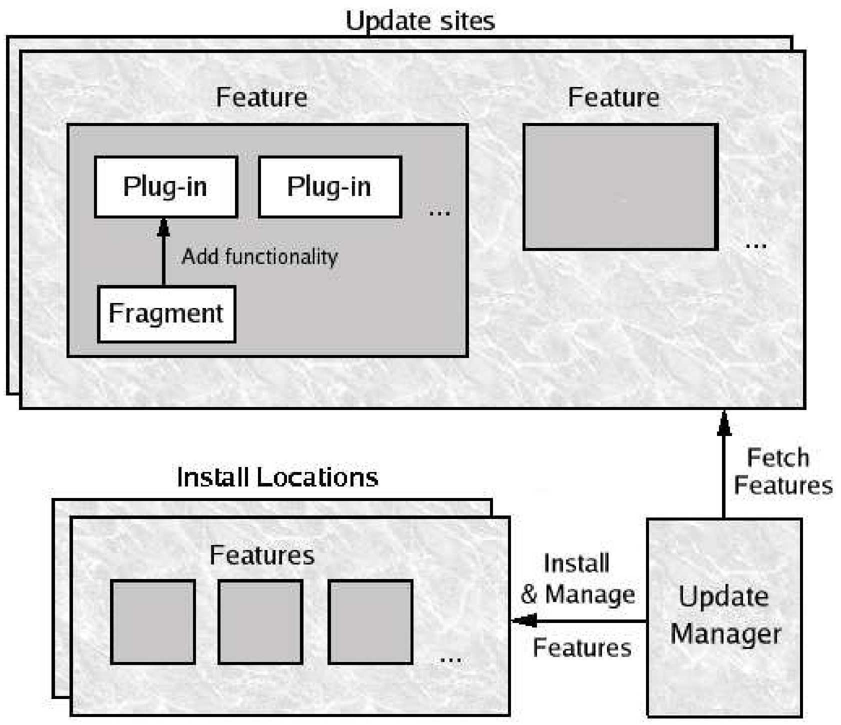


Figure 3.4: Eclipse packaging and feature management.

interface on traditional Java AWT (Abstract Window Toolkit) and Swing widget sets, it ships with a widget set of its own—SWT. More precisely, SWT is a Java API (Application Programming Interface) to operating system's native widget toolkit, so Eclipse looks like a native application depending on the operating system it was built for. SWT uses JNI (Java Native Interface) to access the native toolkit and enforces a one-to-one mapping between Java native methods and operating system calls. If no native widget is available for a specific task, SWT emulates the widget in Java. [12, 40]

- **JFace** builds itself upon SWT and is a set of frameworks for common UI tasks to be used in conjunction with, and to enhance SWT widgets. JFace provides helper classes such as viewers, actions, contributions, image and font registries, dialogs, and wizards. JFace does not try to hide SWT, or replace its functionality. [12, 14]

3.3 License Model

Eclipse's architecture is highly modular as it is based on features, plug-ins, and fragments. An Eclipse installation typically consists of parts authored and distributed by multiple parties with different licenses for their content. Collecting all the different licenses and notices into a single license file, which is sometimes done with less modular applications, is not very reasonable, readable, or maintainable with Eclipse.

The convention for delivering Eclipse based content is to use one umbrella agreement on the top (installation or root feature) level. This agreement is called Eclipse Foundation Software User Agreement (SUA) [20]. SUA does not grant any rights, it only describes the potential layout of legal documentation and references other invidual licenses that may reside in plug-in and feature directories. However, SUA states that if no invidual licenses are provided, Eclipse Public License (EPL) [21] applies for the content. Usually, SUA is displayed when installing a feature using the Eclipse Update Manager. Simply, SUA states that the user is responsible for reading the licenses in specific locations of installation directories of the invidual features and plug-ins, and must accept the terms and conditions if intending to use them. [17]

3.4 Discussion

A pure plug-in architecture, like the architecture of Eclipse, provides a way to offer modular functionality to customers, respond quickly to changes in requirements, and offer upgradeability. This assesses the pressure to market and publish that exists on many areas of business (mentioned in Section 2.4). But there is a price to pay. Birsan [4] describes four challenging issues of pure plug-in architectures. These are installing and updating, security, concurrent plug-in version support, and scalability. Installing and updating is a major issue, as users can install and use plug-ins from various sources, and there is a possibility that the resulting configuration has never been tested. The three other challenges follow (more or less directly) from installing and updating. Security is an issue as plug-in providers have to be trusted and plug-ins with serious bugs can do serious damage. Managing concurrent plug-in versions is not simple—if two different versions of the same plug-in are installed (on different install locations that are both used), it must be decided which one to use. Plug-in dependencies make the decision more difficult, as some plug-ins might require a specific version of another. Scalability is a challenge since the architecture should scale up to large systems of thousands of plug-ins and down to devices with limited resources like cell phones.

3.5 Summary

In this chapter, a short introduction to Eclipse was provided. First, three different things that Eclipse is usually associated with, were described: an integration platform for tools and applications, an open source community, and a Java development environment. The first, Eclipse platform, was chosen as a subject for closer inspection. Basic concepts of Eclipse were described, including the workspace, the resources, and the workbench user interface. Eclipse architecture was reviewed, and it was explained how it differs from a normal extensible IDE—by being a pure plug-in architecture. The packaging in Eclipse was described, including plug-ins as functional components and features as units of deployment. Also, the two user interface frameworks used to build workbench, SWT and JFace, were introduced. The license model was described and some discussion on pure plug-in architectures in general was provided.

4 Eclipse as a CASE System

This chapter considers how Eclipse relates to discussion on Chapter 2. This discussion considers only Eclipse Platform, with a few remarks to Eclipse SDK. First, Eclipse is discussed in respect of CASE classification by Fuggetta [24], which was introduced in Section 2.1.1. After that, tool interfacing and integration in Eclipse are examined. The key challenges of CASE systems in the context of Eclipse are discussed last.

4.1 Classification

IDEs for various languages are build on top of the Eclipse Platform, and in fact Eclipse SDK ships with two IDEs of its own: Java Development Tooling (JDT) and Plug-in Development Environment (PDE). An IDE should be classified as a programming workbench in most cases (see Section 2.1.3), so supporting multiple IDEs makes Eclipse an environment.

It is proposed that Eclipse should be classified as an *integrated environment* (see Section 2.1.1). Even though Eclipse does not have a proprietary database, it provides a common resource model that provides an opportunity for relatively tight data integration of invidual tools. Consistent presentation integration is provided by two UI frameworks: SWT and JFace. In addition, tools encapsulated using Eclipse plug-in model utilize powerful mechanics of control integration. However, Eclipse can be used to build stand-alone applications which have a select part of the building blocks of the environment itself and possible extensions in addition. This is clearly beyond the normal notion of any type of CASE environment.

4.2 Tool Interfacing

Basically, a tool can be interfaced into Eclipse by using the Eclipse APIs, or on invocation level without using the APIs. Besides that, the tool can be interfaced from inside or outside the Java virtual machine. Naturally, the most efficient way to connect to Eclipse is to connect via Eclipse APIs, but it requires encapsulating the tool

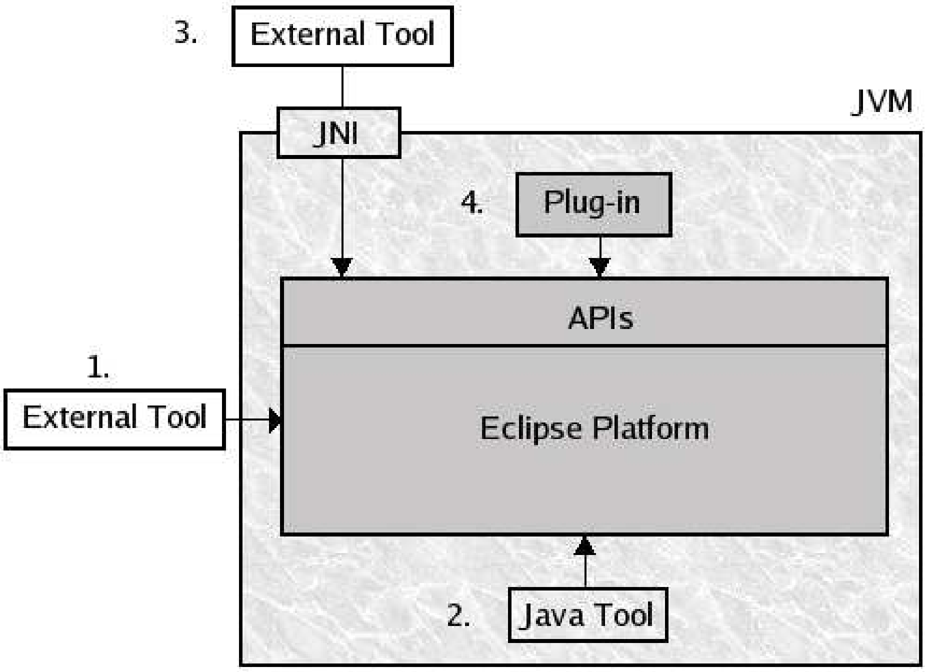


Figure 4.1: Tool interfacing in Eclipse.

using the plug-in model (discussed in Section 5.2.2). This is not always reasonable as using the plug-in model requires providing Java interface(s) for the tool, which is nontrivial when the tool is implemented in a different programming language. External tools can, however, be integrated with some level of integration without encapsulation.

The Eclipse APIs are also used to abstract operating system dependent concepts that Java virtual machine does not abstract adequately. An example is the Jobs API, which provides the responsiveness and concurrency framework of Eclipse.

The resulting four different cases for tool interfacing are (see Figure 4.1):

1. *An external tool is interfaced, but not through the APIs.* This is usually the case when user wants to associate a familiar external tool with a specific filetype in Eclipse. Eclipse uses an operating system independent registry mechanism that can launch an existing tool in an external process when appropriate. Another possibility is to use the integrated Apache Ant support of Eclipse, and launch external tools via Ant.

2. *A Java tool is interfaced, but not through the APIs.* A Java tool is not encapsulated

as a plug-in, but is used with Eclipse. If the tool is a complete Java application, then it can be launched from Eclipse in at least three ways (in addition to using Ant)—by using `java.lang.Runtime` API, by using API provided by Eclipse launching plug-in, or by using Eclipse launch configuration; launch configurations abstract many tedious details that must be assessed with the lower level APIs [57]. If the tool is not a complete Java application, it has to be packaged as a plug-in (or in a plug-in) before it can be used.

3. *An external tool is interfaced into Eclipse APIs through JNI.* A non-Java tool can also be encapsulated as a plug-in by providing implementations of the tool's interfaces through Java Native Interface. This way an external non-Java tool can also access the Eclipse APIs.

4. *A Java tool is encapsulated as a plug-in, and interfaced through Eclipse APIs.* This is the usual way to connect tools into Eclipse—by using Eclipse's plug-in model after encapsulating the tool in a plug-in and connecting it to relevant APIs. A tool in Eclipse is implemented in one or more plug-ins. However, the tool must be implemented in Java. Plug-in model is discussed in Chapter 5.

Cases 1 and 2 can be thought of as uncoupled interfacing. Cases 3 and 4 are considered loosely-coupled as they both use the APIs (tool interfaces). However, if the Java virtual machine (JVM) is considered a single system, then everything inside it would be considered tightly-coupled.

4.3 Levels of Tool Integration

Amsden [1] describes five levels of tool integration in Eclipse. These are: none, invocation, data, API, and UI. This is a bit confusing division since Eclipse API integration contains also APIs for data and UI integration. In this section, the levels of tool integration introduced in Section 2.3 (presentation, data, and control) are used as the basis for discussion.

Presentation integration in Eclipse can be achieved with or without using the Eclipse APIs, as SWT and JFace can be used entirely separately from Eclipse. Of the two frameworks, JFace concentrates on providing consistent *interaction paradigm*, while SWT concentrates on consistent *appearance and behaviour* (See Section 2.3.1). Cross-platform presentation integration requires using these frameworks. However, consistency in appearance differs from operating system to another, because

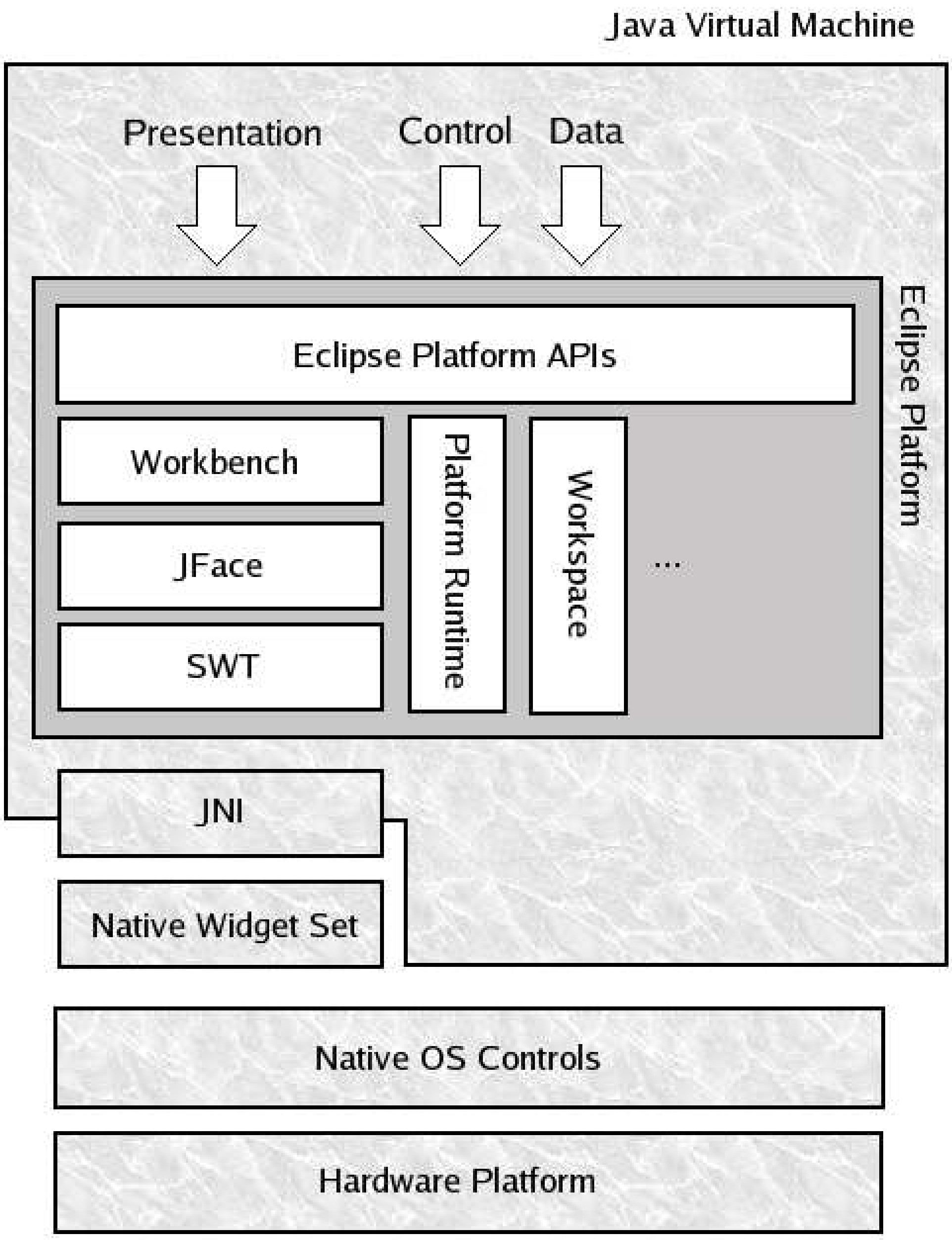


Figure 4.2: Eclipse APIs and levels of integration.

the native widget set is different on different platforms. Behaviour is attempted to keep as consistent as possible. Regardless, the tightest presentation integration with Eclipse requires using the workbench API, which enables the Eclipse workbench to be easily extended with additional menus and other UI components. Integrating with the workbench API uses a stack of APIs and frameworks: Workbench, JFace, SWT, JNI, and the operating systems native widget set (see Figure 4.2).

Data integration in Eclipse is achieved mainly by using the workspace API. Workspace has two views: a physical and a logical view. Physical view is a directory in the user's hard-drive, containing projects and project related information. Logical view is the internal in-memory representation of the contents of workspace—the internal resource model. Workspace API is the means to provide tighter data integration to Eclipse tools and keep the physical and logical views synchronized. A tool developer can use the components of the internal resource model to handle the processing of resources. This is not compulsory, but when the API is used, tools are integrated more tightly. Eclipse has no repository, but it has built-in client support for CVS. Externally launched tools can also make use of the Eclipse team and version control facilities when they are working with files in the Eclipse workspace. It is rather difficult to discuss the distinct features of data integration in the context of Eclipse. These features, introduced in Section 2.3.2, are: *interoperability*, *nonredundancy*, *data consistency*, *data exchange*, and *synchronization*. Eclipse provides facilities to utilize some of these, but does not enforce them. There is no common view of the data that tools must use, and nonredundancy and data consistency are not handled as these are mostly database issues. Eclipse platform or SDK doesn't provide a database or support for one. Data exchange happens largely on method level using the published interfaces of distinct tools. Eclipse also provides resource change listeners, persistent and nonpersistent resource properties, and much more. Still, data integration is probably the "weakest" conceptual level of integration of the three (presentation, data, and control). However, there already exists Eclipse Data Tools Platform project [13].

Control integration in Eclipse is achieved by using the Eclipse plug-in model (see Chapter 5). The plug-in model provides means for tools to offer services in a way standard to Eclipse, which promotes *provision* and *use*. This is what Eclipse builds upon. The Platform Runtime component handles the matching of service provider plug-ins and service user plug-ins. A low level of control integration (invocation) can also be achieved without the API by associating a tool with a specific type of resource, by using integrated Apache Ant support, or by using the internal launching framework for Java applications.

Eclipse integration mechanics concentrate on providing several frameworks for tool integration (the IPSE approach, see Section 2.1.2). These frameworks also provide portability services that allow Eclipse to be used on various platforms. Eclipse Platform (or even SDK) provides no process integration framework for its tools, but the platform can be used to build mechanics for one.

4.4 Discussion: Key Challenges

Eclipse promotes separation of concerns in many ways. By default, plug-ins are identified as units of function and features as units of deployment. Tools for multi-dimensional separation of concerns and management of concerns have also been developed for Eclipse, e.g. the AspectJ project [2], which is a seamless aspect-oriented extension to the Java programming language. AspectJ promises to enable "clean modularization of crosscutting concerns, such as error checking and handling, synchronization, context-sensitive behavior, performance optimizations, monitoring and logging, debugging support, and multi-object protocols." Eclipse has proven to be a platform that promotes easy integration of such new technologies. The vision of permanently malleable software has not been achieved, and might never be. When a tool is encapsulated as an Eclipse plug-in, the encapsulating mechanisms make the encapsulated tool specific to Eclipse. Also, the encapsulation is not fully automated, although Eclipse provides good tools for the encapsulation. Although Eclipse is said to be language-neutral, using Eclipse APIs requires providing Java interfaces for tools. If the tool itself is not implemented with Java, connecting it to the API can be too difficult or clumsy to be worth the trouble. Tools and complete IDEs for different languages can be constructed, but doing it through Eclipse APIs requires using Java. In any case, new and interesting directions in software development are researched using Eclipse technologies (e.g. [2], [6], and [46]).

4.5 Summary

In this chapter, it was proposed that Eclipse should be classified as an integrated environment. Tool interfacing and integration in Eclipse were also discussed. Discussion included different ways tools can be connected to Eclipse platform and the levels of integration that can be achieved. This discussion considered both Java and non-Java tools. The key challenges of CASE systems were also considered in the context of Eclipse.

5 Eclipse Platform: A Technical Overview

Everything is a contribution.

— *Contribution Rule*

This chapter discusses some essential Eclipse platform concepts, components, and design decisions. Two essential concepts on which the Eclipse plug-in model builds are explained first—extension points and extensions. The Platform Runtime component is introduced after that. Plug-ins are examined on logical and physical levels. Design decisions of Eclipse platform are discussed by introducing the Extension Object pattern, a design pattern used in Eclipse. Also, the rules of Eclipse are introduced.

This chapter illustrates different relationships using the UML (Unified Modeling Language) [41] notation. UML component, class, and sequence diagrams are used. UML diagrams are also used in Chapter 6 to illustrate relationships in a practical application.

5.1 Basic Concepts

This section describes the basic concepts of extensions, extension points, and the Platform Runtime. Extensions and extension points are the glue that is used to bind plug-ins together, whereas Platform Runtime is the component responsible for loading and binding the plug-ins.

5.1.1 Extension Points and Extensions

An *extension point* defines a place where other plug-ins can introduce new capabilities by contributing *extensions*. These capabilities can include additional functionality or content of some sort. An extension point is a published interface to the world outside the plug-in. There are no private extension points in Eclipse. An extension point can be thought of as a service provided by an invidual plug-in, whereas an *extension* is a contribution of a plug-in to some plug-in's extension point. An extension "extends" a plug-in. Gamma and Beck [26] compare extension point to a power strip and extensions to the power plugs that are plugged into the strip. [12, 26]

Typically, providing an extension to an extension point means introducing a class that implements particular interface or interfaces that the use of the extension point presumes. The extension point provider plug-in (host plug-in) can then process the extensions when appropriate.

Extension points provided by Eclipse platform are described in the Eclipse Platform Plug-in Developer Guide [14]. These extension points can be used to add functionality or content to the Eclipse platform. This includes, but is not limited to, workbench extensions such as menus and wizards, and core functionality such as new resource types.

Extension points are the means of Eclipse-based tools to provide control integration. Providing an extension point promotes *provision* and providing an extension to some other plug-in's extension point promotes *use* (See Section 2.3.3).

5.1.2 Platform Runtime

The Platform Runtime component handles the activation of plug-ins in Eclipse. Activating a plug-in means loading a plug-in's runtime class and instantiating and initializing its instance [5]. The Platform Runtime uses *lazy loading* (or late binding) of plug-ins. On startup, it loads representatives of plug-ins into the memory using plug-ins' manifest files. Every plug-in is treated equally. These representatives contain the minimal information needed about the plug-ins. The plug-in classes themselves are created only when needed. [26]

Eclipse 3.0 introduced a new Platform Runtime based on the OSGi framework [43]. The new platform made it possible to load new plug-ins dynamically while running Eclipse. With the old Platform Runtime, Eclipse had to be restarted for it to recognize new plug-ins [12]. The new platform presents a plug-in from two different points of view: in terms of Eclipse plug-in and in terms of the OSGi framework. Basically, the latter considers only installation and packaging related issues and the former everything else.

5.2 Plug-ins

Plug-ins are considered on two different levels. On the logical level, the different relationships between plug-ins are considered. On the physical level, it is discussed how plug-ins can be introduced and how the logical relationships can be declared for the Platform Runtime.

5.2.1 Logical level

On the logical level, Eclipse has an in-memory representation of each plug-in, an instantiation of the plug-in class or a representative of it. Each plug-in has its own class loader and a separate namespace.

There are two kinds of plug-in relationships in Eclipse: Dependency relationships and extension relationships. These relationships are defined in the plug-in manifests of plug-ins involved (see Section 5.2.2). The relationships and the roles they impose on plug-ins are examined next.

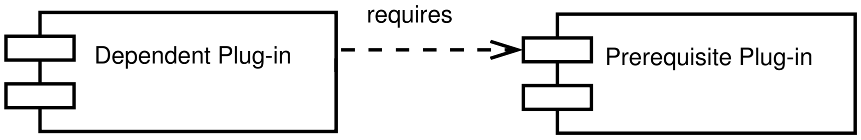


Figure 5.1: Plug-in dependency relationship.

Figure 5.1 depicts a dependency relationship, in which a dependent plug-in reguires a prerequisite plug-in. Plug-in dependency relationships are very common as the overall plug-in architecture forces a certain level of dependency. However, the plug-ins are not very fine-grained—largest Eclipse (SDK) feature (excluding the platform itself), JDT, consist of eleven plug-ins.

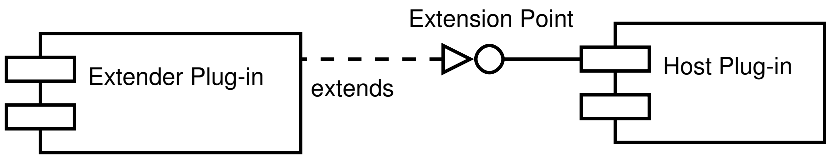


Figure 5.2: Plug-in extension relationship.

Figure 5.2 depicts an extension relationship. In an extension relationship, an extender plug-in extends a host plug-in via an extension point. There can be multiple extenders to each extension point and a host can have multiple extension points. A plug-in can also extend an extension point multiple times and a host plug-in can even extend its own extension point. Extending one's own extension point is utilized for example in the workbench pull-down menus. Workbench declares an extension point for adding menus and uses its own extension point to add the default menu actions. This way, every menu element is treated uniformly. [5, 38]

Figure 5.3 gives a sample extension scenario on the logical level, considering five plug-ins: A, B, C, D, and E. Plug-in D is dependent on plug-in A and extends extension point 1 of plug-in B and extension point 3 of plug-in C. Plug-in D also provides an extension point 4, which in part is extended by plug-in E. Extension point 4 could be a derived service of services provided by extension points 1 and 3.

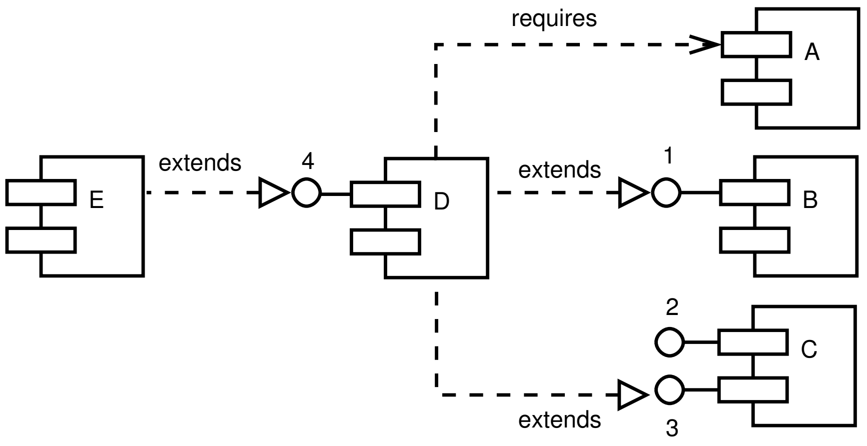


Figure 5.3: Sample plug-in extension scenario.

5.2.2 Physical level

On the physical level each plug-in is represented by a unique plug-in directory. The plug-in directory uses the Java package naming convention to avoid name clashes. This directory resides under the `plugins`-directory of Eclipse installation. The plug-in directory usually contains:

- `plugin.xml` (the plug-in manifest);
- Java JAR-files;
- additional resources (icons, etc.).

The only obligatory file for describing a plug-in is the plug-in manifest file. Since Eclipse 3.0, as a new Platform Runtime component was introduced, there have been two kinds of manifests: plug-in manifests and bundle manifests. A plug-in manifest is an XML-file which describes the extensions and extension points of a plug-in. In the versions of Eclipse previous to 3.0, the plug-in manifest was also the only place to describe plug-in dependencies and other related information. Since Eclipse 3.0, a second manifest file, bundle manifest, is used to describe the contents and dependencies of a plug-in to the new OSGi-based runtime, but a plug-in manifest is still needed to describe the extension relationships. If the Platform Runtime finds only a plug-in manifest, it transparently generates a bundle manifest from it and saves it in a configuration directory. Only the contents of a plug-in manifest are examined in this thesis. The simplest plug-in can contain only a plug-in manifest and use it to contribute some content to Eclipse. The possible functionality itself provided by the plug-in is contained in Java JAR-files. A plug-in directory can also contain some additional resources like icons, templates, and so on. Eclipse 3.1 also enables the developer to ship a plug-in packaged as a single JAR file.

In a dependency relationship, the dependent plug-in could declare its dependency in the following way in its plug-in manifest:

```
<requires>
   <import plugin="org.example.PreRequisite" />
</requires>
```

In an extension relationship, the host plug-in could have the following element in its plug-in manifest for declaring the extension point:

`<extension-point id="ExamplePoint"name="An␣example␣extension␣point"/>`

And an extender plug-in could have in its plug-in manifest:

`<extension point="org.foo.ExamplePoint"class="org.bar.AnExtension"/>`

This is the extension relationship in its simplest form on the physical level. The extension points and extensions have unique identifiers, and the extender plug-in has to provide a class that conforms to the defined interface of an extension point. This interface is often called the *callback interface*. In the example above, these identifiers are `org.foo.ExamplePoint` and `org.bar.AnExtension`. Some extension points may require more than one callback interface and some others do not require any [4]. See Appendix A.2.1 or A.3.1 for a complete plug-in manifest.

The DTD (Document Type Definition) of the plug-in manifest can be found in [14].

A host plug-in can also define an extension point schema for a given extension point that can be used to check that the extension declaration contains valid elements. Often declaring an extension to an extension point requires providing some structured information with XML-elements, some attributes and their expected values. [12]

5.3 Processing Extensions of an Extension Point

Eclipse platform provides API calls for plug-in developers to handle processing the extensions of their extension points. Platform provides parsed representations of extensions to the developer's extension point, which were declared in the manifests of the extender plug-ins. The following code snippet shows an example how extensions can be processed (adapted from [12]):

```
IExtensionRegistry er = Platform.getExtensionRegistry();
IExtensionPoint ep =
    er.getExtensionPoint(myPluginId, myExtensionPointId);
IExtension[] extensions = ep.getExtensions();

for(int i=0; i < extensions.length; i++) {
    IConfigurationElement[] ces =
    extensions[i].getConfigurationElements();
    // handle configuration elements...
}
```

The methods in the example are quite self-explanatory. The essential classes are listed below.

- `Platform` is the central class of Eclipse Platform Runtime, providing a facade for the general platform services. `Platform.getExtensionRegistry()`-method[1] is used to get a hold of the extension registry.
- `IExtensionRegistry` is the interface of the extension registry. The extension registry contains all information specified by extensions to the extension points.
- `IExtensionPoint` is an interface to an extension point of a host plug-in.

[1]In the versions of Eclipse previous to 3.0, `Platform.getPluginRegistry()`-method was used.

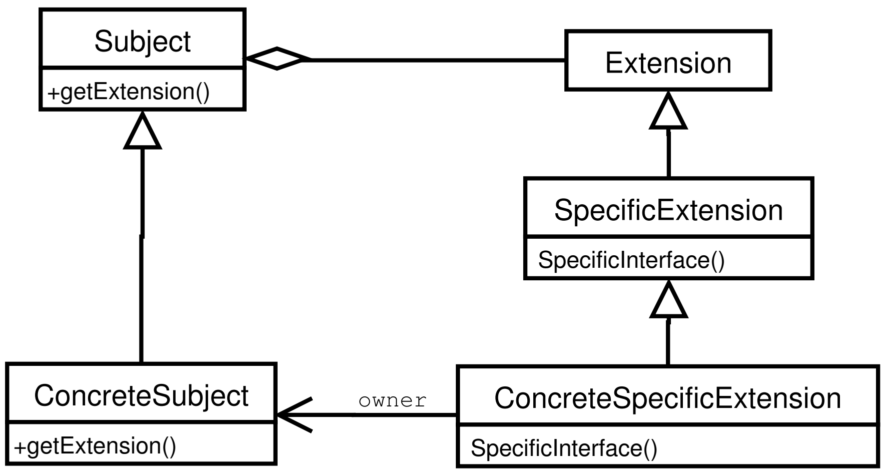


Figure 5.4: Basic structure of Extension Object pattern [28].

- `IExtension` is an interface to an extension of an extension point.
- `IConfigurationElement` is an interface to the parsed version of an extension element in a plug-in manifest of an extender plug-in.

The extensions to an extension point defined by a plug-in can be processed when the plug-in class is loaded, or when the plug-in actually needs information about the extensions. Former can cause some overhead when loading the plug-in. The latter is the preferred way in Eclipse (Lazy Loading Rule, rule 2 of Table 5.1). [5, 12]

5.4 Design Decisions - Extension Object Pattern

Eclipse's object-oriented design uses frameworks and a variety of design patterns. These were mentioned in Section 2.4 as present techniques used to prepare software for change. SWT and JFace are examples of (UI) frameworks used. An essential pattern used in Eclipse, the Extension Object Pattern [28], is described next.

Extension Object pattern (a.k.a. Extension Interface) [28] is utilized throughout the Eclipse architecture. The pattern's intent is to: "Anticipate that an object's interface needs to be extended in the future. Extension Object lets you add interfaces to a class and lets clients query whether an object has a particular extension." As consequences of using this pattern, bloated class interfaces for key abstractions can be avoided and different roles in different subsystems for key abstractions can be supported. A negative side effect is that clients become more complex, as using an extended interface itself is more complex. Also, the subject's interface does not express all of its behaviour anymore, and thus it may become more difficult to understand. In addition, there is a tension to abuse extensions for concepts that should be explicitly modeled.

Figure 5.4 illustrates the basic structure of Extension Object pattern. `Subject` declares an operation to return a particular extension given a specification, and defines how a specification is mapped to an extension. `Extension` is a base class for all extensions of a subject. `ConcreteSubject` implements `getExtension` for retrieving extensions, whereas `SpecificExtension` declares an interface for a specific extension. `ConcreteSpecificExtension` implements an extension for a particular `ConcreteSubject` and stores implementation- and extension-specific state. `ConcreteSpecificExtension` knows its owning subject.

A similar querying of interfaces is often used in object broker architectures and some programming languages provide extension facilities as default. For example, object-oriented scripting language called Ruby [45] provides means to extend objects dynamically. Objects can also be queried if they implement a specific interface (a mixin module in Ruby).

In Eclipse, the extension support is class-based. Behavior can be added to existing classes but not to specific existing instances of classes. Eclipse's adapted version of the Extension Object pattern includes the following classes (see Figure 5.5) :

- `IAdaptable` is the basic interface which defines the `getAdapter`-method for retrieving a specific extension. `getAdapter` is passed the class literal of the extension as an argument. A class can implement this interface, or subclass `PlatformObject` to get the default bahaviour.

- `IAdapterFactory` is an interface to an adapter factory, which encapsulates specific extensions for a particular type or types.

- `IAdapterManager` is an interface to an Adapter Manager which registers adapter factories for specific types. An Adapter Manager is provided by the `Platform` class.

- `PlatformObject` is a class provided by Eclipse platform which implements `IAdaptable` and can be subclassed. `PlatformObject` provides a dynamic implementation of Extension Object pattern using interfaces described above. See Figure 5.6 for collaboration details.

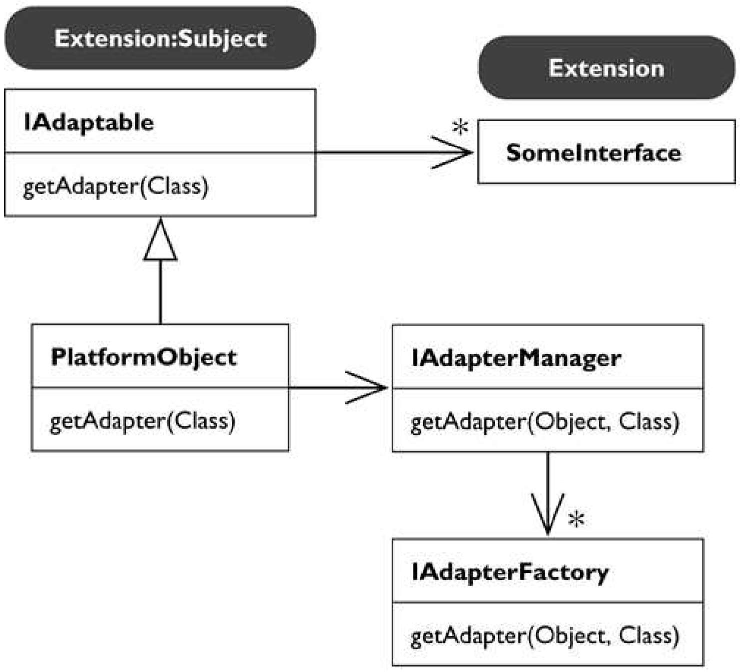


Figure 5.5: Extension Object pattern in Eclipse [26].

In Eclipse, there is no common parent class for extensions, nor a specific interface that all extensions must implement. The `getAdapter`-method (analogous to `getExtension`) is passed the class literal of the extension to be retrieved. Following code snippet illustrates usage of the pattern in Eclipse:

```
PlatformObject po = ... // get a PlatformObject
SomeInterface iface = (SomeInterface) po.getAdapter(SomeInterface.class);
if(iface != null)
{
    // do SomeInterface related things
}
```

Figure 5.6 provides a sequence diagram illustrating the collaboration of objects with the code above. `PlatformObject po` is asked for an adapter to interface `SomeInterface` by passing the class literal of `SomeInterface` as an argument for `getAdapter`. The class literal is abbreviated to `cl` in the figure. `po` asks `Platform` for an adapter manager, and receives a handle of adapter manager `am`. `po` forwards the call `getAdapter` to `am` and passes itself as the first argument and `cl` as the second. `am` looks for an adapter factory for the given combination, and if it is found,

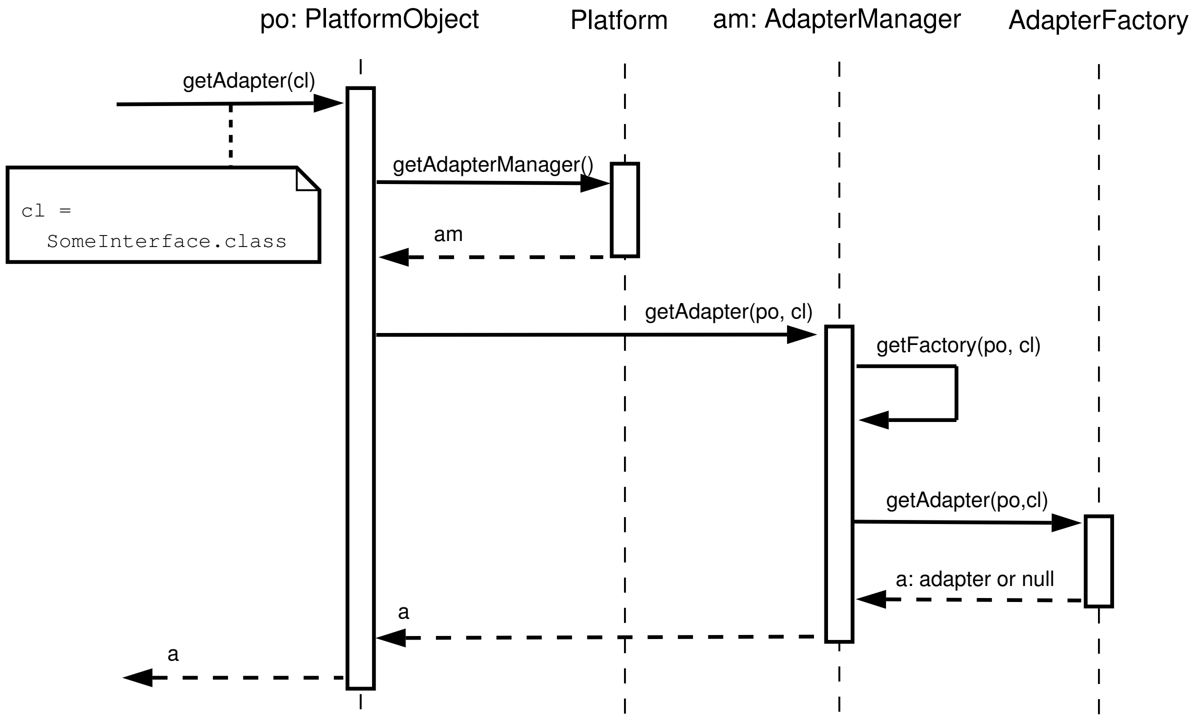


Figure 5.6: Sequence diagram of Extension Object pattern in Eclipse.

the call of `getAdapter` is forwarded to it. If it is not found, null is returned. Finally, adapter factory returns `a`, which is the requested adapter or null if none is found. `am` in part returns `a` to `po`, which returns it to the caller of the `getAdapter`-method. The adapter is an object castable to `SomeInterface`.

A common use of this pattern in Eclipse is to provide `IResource` adapters for different domain objects (e.g., for a Java project). Providing `IResource` adapters is a rule of Eclipse, which leads us to next section. [26]

5.5 Rules of Eclipse

Gamma and Beck [26] present a set of rules of Eclipse. Following these rules when developing plug-ins make the plug-ins themselves and the platform as a whole safer to use. As mentioned in the beginning of Section 3.2, a bad neighbor affects all the residents in the neighborhood. These rules also characterize matters that plug-in developers must take into account when developing plug-ins, and as such characterize the nature of developing plug-ins for Eclipse. The rules are listed in Table 5.1.

5.6 Summary

In this chapter Eclipse platform was examined from a more technical point of view. Basic concepts of Eclipse plug-in model were explained: extension points, extensions, and the Platform Runtime. After that, plug-in relationships were examined on logical and physical levels. This clarified how Eclipse-based tools interact and are encapsulated using the Eclipse plug-in model. It was also described how a plug-in developer can process the extensions of extension points. Design decisions in Eclipse were briefly discussed by introducing an essential pattern used in Eclipse, the Extension Object pattern, which largely contributes to the flexibility of the platform. Also, the rules of Eclipse were introduced, which are general guidelines that plug-in developers should follow.

Nr	Rule	Explanation
1	Contribution	Everything is a contribution.
2	Lazy Loading	Contributions are only loaded when they are needed.
3	Sharing	Add, don't replace.
4	Conformance	Contributions must conform to expected interfaces.
5	Monkey see / monkey do	Always start by copying the structure of a similar plug-in.
6	Relevance	Contribute only when you can successfully operate.
7	Safe platform	As the provider of an extension point, you must protect yourself against misbehaviour on the part of extenders.
8	Invitation	Whenever possible, let others contribute to your contributions.
9	Fair play	All clients play by the same rules, even me.
10	Explicit extension	Declare explicitly where a platform can be extended.
11	Diversity	Extension points accept multiple extensions.
12	Good fences	When passing control outside your code, protect yourself.
13	License	Always supply a license with every contribution.
14	Program to API contract	Check and program to the Eclipse API contract
15	Integration	Integrate, don't separate.
16	Responsibility	Clearly identify your plug-in as the source of problems.
17	User arbitration	When there are multiple applicable contributions, let the user decide which one to use.
18	Other	Make all contributions available, but the ones that don't typically apply to the current perspective appear in an Other? dialog.
19	Explicit API	Separate the API from internals.
20	Stability	Once you invite others to contribute, don't change the rules.
21	Defensive API	Reveal only the API in which you have confidence, but be prepared to reveal more API as clients ask for it.
22	User Continuity	Preserve the user interface state across sessions.
23	Adapt to `IResource`	Whenever possible, define an `IResource` adapter for your domain objects
24	Strata	Separate language-neutral functionality from language-specific functionality and separate core functionality from UI functionality.

Table 5.1: The rules of Eclipse [26]

6 Case Study: Integrating UMLGraph into Eclipse

This chapter provides a case study of tool integration in Eclipse. This specific case of tool integration includes integrating an external Javadoc-based tool and an external non-Java tool into a consistent customizable toolchain that can be controlled via the Eclipse UI.

6.1 Background

UMLGraph [53] is a *Javadoc doclet*[1] that is used to generate UML class diagrams from Java source files. The generation of class diagrams can be directed with Javadoc comments. The idea behind UMLGraph is that design models should be composed textually and graphs should be automatically generated from the textual declarations [49]. UMLGraph can also be used to generate sequence diagrams using a declarative syntax, but support for this is out of the scope of this case study. Although, some preliminary support has been developed to LightUML. The current features of UMLGraph are part of an ongoing effort aiming to provide support for all types of UML diagrams. UMLGraph produces a GraphViz diagram specification that can be converted into graphical format using GraphViz tools.

GraphViz [32] is an open source graph visualization software consisting of several graph layout programs. One of these programs is `dot`. It produces drawings of directed graphs and can be used to convert UMLGraph-generated diagram specifications into graphical format, such as Portable Network Graphics, PostScript, etc.

Most existing UML tools for Eclipse use two subprojects of the Eclipse Tools Project: EMF (Eclipse Modeling Framework) [16] and UML2 (an EMF-based implementation of the UML 2.0 metamodel for the Eclipse platform) [19]. This makes them quite “heavy-duty”, as various prerequisite plug-ins are needed. On the con-

[1]*Javadoc* is a tool that parses the declarations and documentation comments in a set of source files and produces a set of HTML pages describing the classes, interfaces, constructors, methods, and fields. You can use Javadoc doclets to customize Javadoc output. A *doclet* is a program written with the Doclet API that specifies the content and format of the output to be generated by the Javadoc tool. [50]

trary, the intent here is to integrate a lightweight UML tool into Eclipse that can be used in retrospect to generate a class diagram from a Java project or package. This is achieved by integrating UMLGraph doclet and external GraphViz `dot` tool via Apache Ant integration of Eclipse. The resulting Eclipse feature is called LightUML.

6.2 Toolchain

The toolchain to be executed involves four sequential steps. The steps are encapsulated into Eclipse `Job`-classes. Eclipse `Job`-classes are part of the `org.eclipse.-core.runtime` package, which provides an infrastructure for scheduling, executing, and managing concurrently running operations [14].

A common parent class for jobs in LightUML is `LightUMLJob`. Figure 6.1 illustrates the class structure of the jobs. `LightUMLJob` uses a scheduling rule (`Light-UMLSchedulingRule`) that prevents two `LightUMLJob`s from running concurrently, as they should be run in a sequential manner. However, they can be run concurrently with other jobs that might be running in Eclipse. The steps and their corresponding jobs are:

1. *Initializing the plug-in state location* (encapsulated in `Initialize`). Plug-in state location is a specific location on local harddrive that the plug-in is free to write to. This step involves copying the necessary files there if needed (see Section 6.3.5).

2. *Running UMLGraph doclet* with a run of Javadoc (encapsulated in `JavaToDot`). The Java source files of a Java project or package in question are used as input for UMLGraph. UMLGraph produces a `graph.dot`-file.

3. *Running GraphViz `dot` tool* to convert the `graph.dot` to a graphical format of choice that is supported by `dot` (encapsulated in `DotToGraphics`). The converted file is stored in the state location. This step requires that GraphViz is installed.

4. *Adding the class diagram to the selected Java project* (encapsulated in `AddGraph-ToProject`). The class diagram is added to the preferred folder under the project by using the Eclipse resource model.

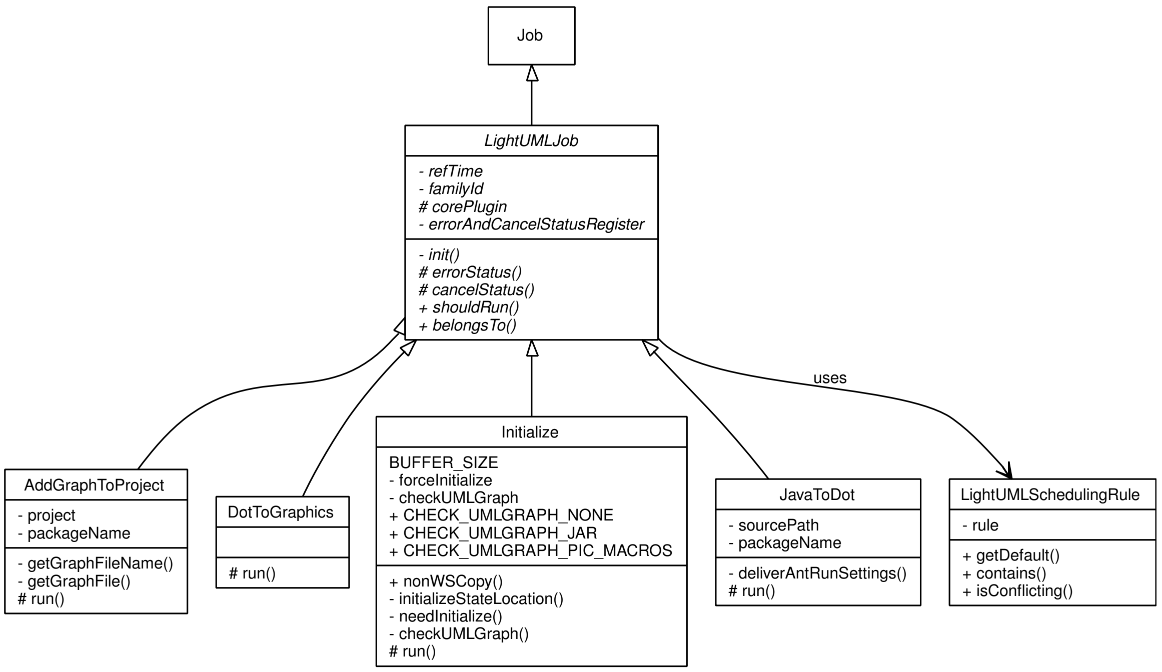


Figure 6.1: LightUMLJobs.

6.3 Structure

The structure of LightUML is examined from the following points of view in a top-down manner: feature structure, plug-in structure, package structure of plug-ins, and class structure of plug-ins. Directory structures of install locations and state locations of plug-ins are also examined. Complete source codes for the Java classes can be found in Appendix A.

6.3.1 Features

LightUML is packaged into a single deployable feature called `org.lightuml`. It can be installed using the Update Manager of Eclipse. It also contains the copyright notice and license agreement (License Rule, rule 13 of Table 5.1).

6.3.2 Plug-ins

The UMLGraph integration is divided into two plug-ins: `org.lightuml.core` and `org.lightuml.ui`. Former consists of the core functionality, and latter has

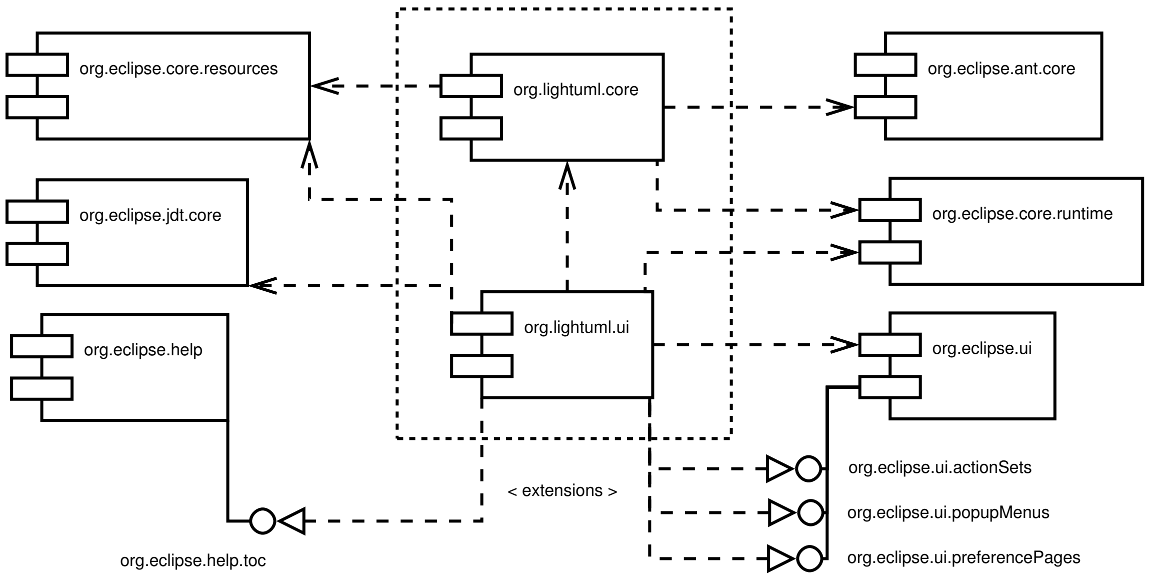


Figure 6.2: LightUML plug-in dependency and extension relationships.

the user interface related functionality. This division follows the Strata Rule (see Table 5.1, rule number 24). The plugin dependency and extension relationships are illustrated in Figure 6.2.

The core plug-in is kept as independent as possible from other plug-ins. It depends on three other plug-ins: `org.eclipse.ant.core`, `org.eclipse.core.resources`, and `org.eclipse.core.runtime`. Plug-in `org.eclipse.ant.core` defines the Apache Ant integration of Eclipse, which is needed for running the external tools (UMLGraph doclet and GraphViz `dot` tool). Plug-in `org.eclipse.core.resources` defines the Eclipse resource model, which is used to add the generated class diagram into the project. Plug-in `org.eclipse.core.runtime` is needed for the `Job` support.

The UI plug-in depends on the core plug-in in addition to `org.eclipse.core.resources` (see above), `org.eclipse.core.runtime` (see above), `org.eclipse.jdt.core`, and `org.eclipse.ui`. Plug-in `org.eclipse.jdt.core` defines the Java model of Eclipse. It is used when associating the contributed action (`GenerateClassDiagram`) with Java projects and packages. Plug-in `org.eclipse.ui` is needed for adding the actions and the prefererence pages to the Workbench user interface.

The LightUML UI plug-in extends four extension points of the Eclipse platform:

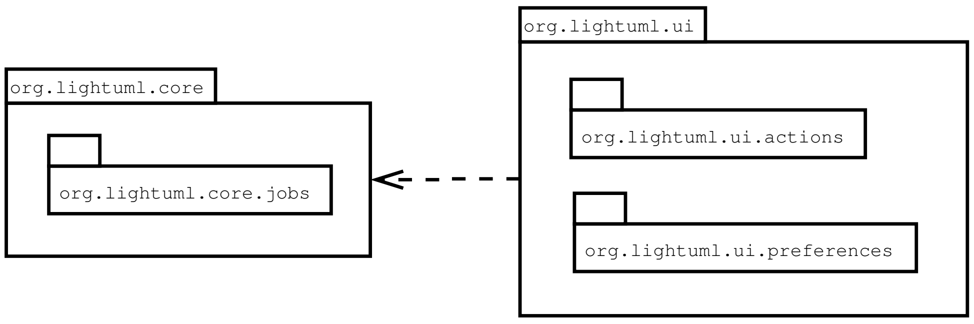


Figure 6.3: Package diagram of LightUML.

- `org.eclipse.ui.actionSets`. This extension point is used to add the `RestoreSettings` action to the workbench pulldown menu. See Figure 6.7.
- `org.eclipse.ui.popupMenus`. This extension point is used to add `GenerateClassDiagram` action to popup menu when a Java project or package is selected. Adding the action only when a Java project or package is selected follows the Relevance Rule (rule 6 of Table 5.1). See Figure 6.8.
- `org.eclipse.ui.preferencePages`. This extension point is used to add the preference pages of LightUML. See Figures 6.9, 6.10, and 6.11.
- `org.eclipse.help.toc`. This extension point is used to provide help content for the plug-in. LightUML help is added to the Eclipse help table of contents.

6.3.3 Packages

The core plug-in of LightUML has one subpackage, `org.lightuml.core.jobs`. The `Job`-classes contained were discussed in Section 6.2.

The UI plug-in has two subpackages: `org.lightuml.ui.actions`, which contains the actions added to workbench, and `org.lightuml.ui.preferences`, which contains the preference pages of LightUML.

Figure 6.3 is a package diagram of LightUML.

6.3.4 Classes

Figure 6.4 is a class diagram of `org.lightuml.core` plug-in (does not include `Job`-classes which were described earlier). The classes are described below.

- `Plugin`. The abstract superclass of all plug-in runtime class implementations. *Provided by Eclipse.*
- `LightUMLCorePlugin`. The core of LightUML. Directs the execution of jobs and runs of Apache Ant. Extends `Plugin`, and has one instance of `IGraph-Converter`. Can throw instances of `LightUMLCoreException`.
- `IGraphConverter`. An interface for converting diagram specifications into graphical form.
- `LocalGraphConverter`. Implementation of `IGraphConverter` that uses locally installed GraphViz via Apache Ant.
- `LightUMLCoreException`. An exception thrown by LightUML core. Extends `Exception`.
- `Exception`. The basic Java exception class.

Figure 6.5 is a class diagram of `org.lightuml.ui` plug-in. The classes are described below.

- `AbstractUIPlugin`. Abstract base class for plug-ins that integrate with the Eclipse platform UI. Provides capabilites for preferences, dialogs, and images. *Provided by Eclipse.*
- `LightUMLUIPlugin`. The plug-in class of `org.lightuml.ui`. Extends `AbstractUIPlugin` and adds the provided actions and preference pages to the Eclipse workbench.
- `GenerateClassDiagram`. The popup menu action that is used to trigger the generation of a class diagram. Implements `IObjectActionDelegate`, so that this action can be associated with Java projects and packages (Conformance Rule, rule 4 of Table 5.1).

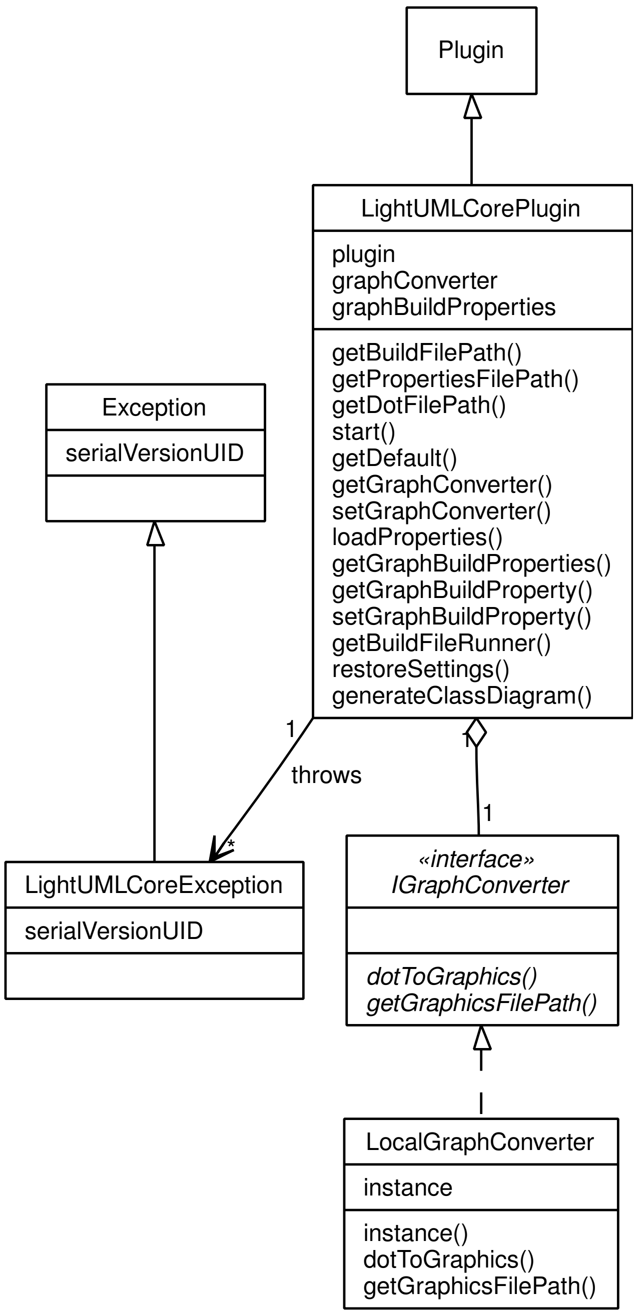


Figure 6.4: Class structure of `org.lightuml.core` plug-in.

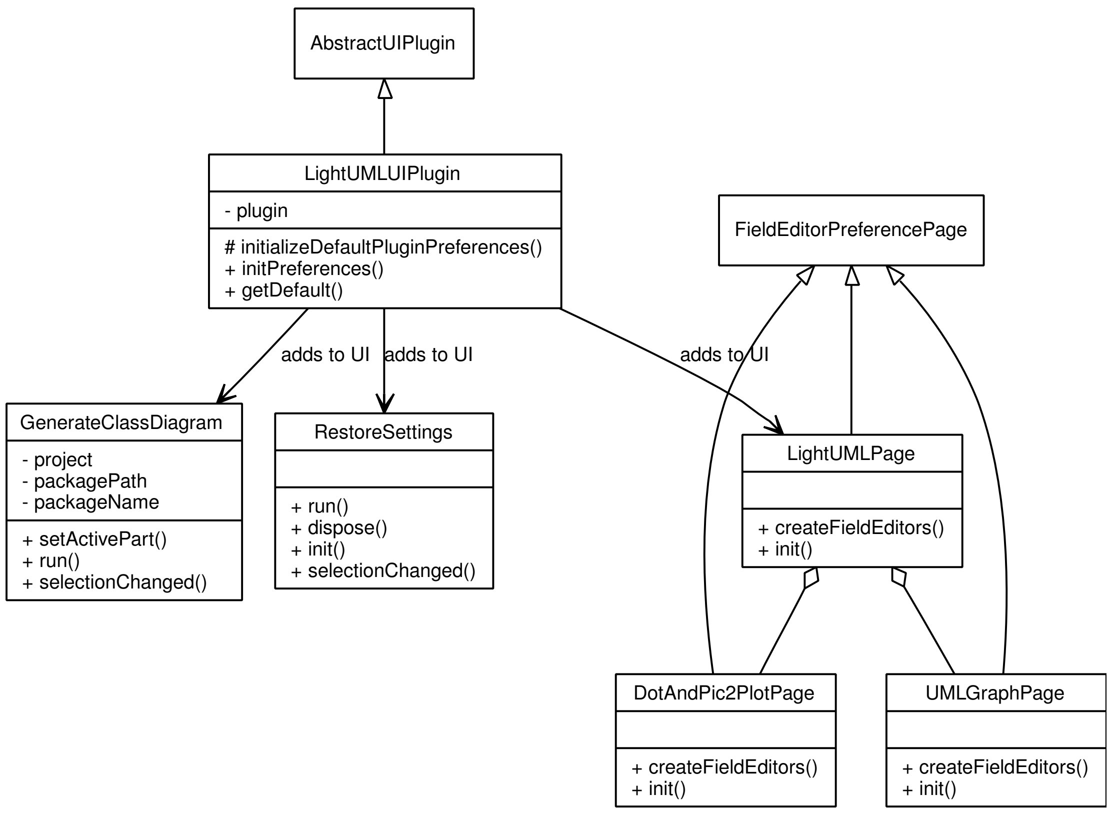


Figure 6.5: Class structure of `org.lightuml.ui` plug-in.

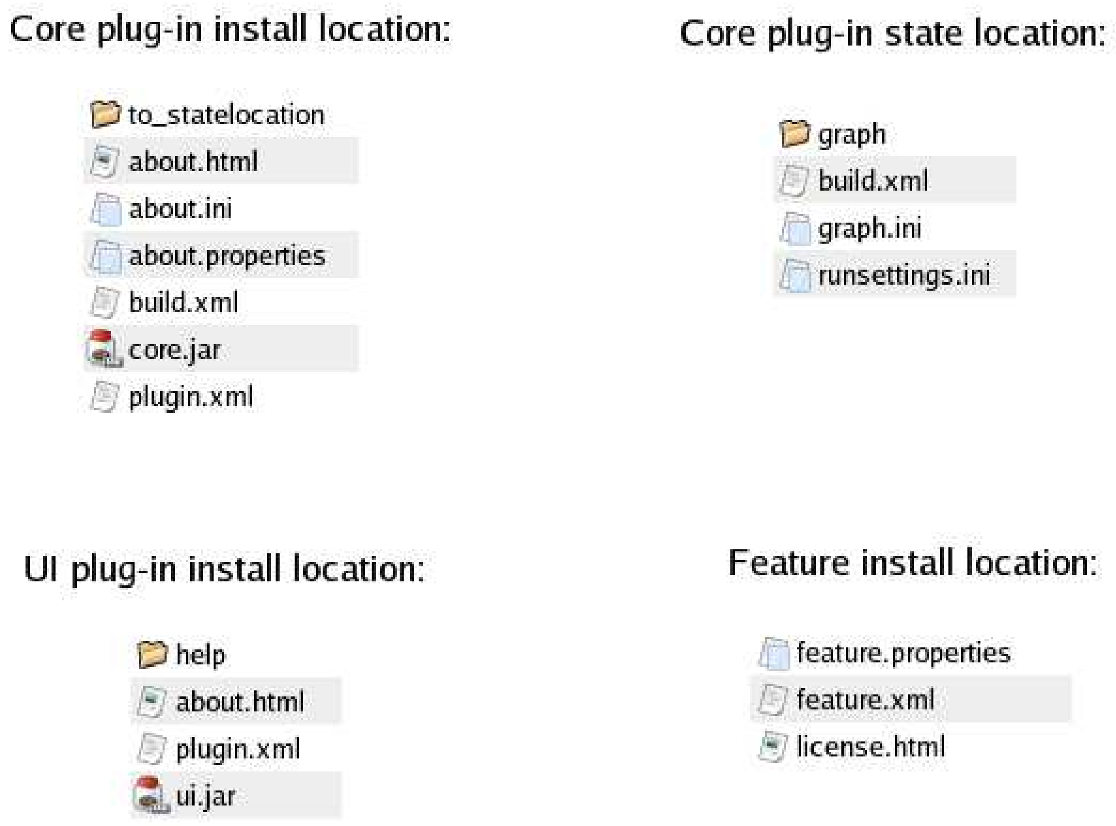
- RestoreSettings. The action for restoring default settings of LightUML. Implements IWorkbenchWindowActionDelegate, so that this action can be contributed to workbench window menu (pulldown menu) or toolbar.
- FieldEditorPreferencePage. A special abstract preference page to host field editors. A field editor presents the value of a preference to the end user. The value is loaded from a preference store; if modified by the end user, the value is validated and eventually stored back to the preference store. A field editor reports an event when the value, or the validity of the value, changes. *Provided by Eclipse.*
- DotAndPic2PlotPage. Is a subpage of LightUMLPage (defined via the plug-in manifest of LightUML UI plug-in), providing preferences for the dot executable. Extends FieldEditorPreferencePage.
- UMLGraphPage. Is a subpage of LightUMLPage, providing preferences for UMLGraph. Extends FieldEditorPreferencePage.
- LightUMLPage. This class provides the preference page for general preferences of LightUML. Extends FieldEditorPreferencePage.

6.3.5 Directories

LightUML has data on the following locations in the harddrive: install location of both plug-ins (org.lightuml.core and org.lightuml.ui), state location of the core plug-in, and install location of the feature (org.lightuml). Figure 6.6 depicts these directory structures.

Install location of the core plug-in contains the following files:

- to_statelocation. A directory containing the build.xml and graph.ini files which are copied to the state location of the core plug-in because plug-in install locations have read-only permissions.
- about.html. Contains the copyright note and license agreement.
- about.ini. Contains a short description of the purpose of the LightUML feature. Core plug-in is chosen as the branding plug-in of the feature, so this



Core plug-in install location:

- to_statelocation
- about.html
- about.ini
- about.properties
- build.xml
- core.jar
- plugin.xml

Core plug-in state location:

- graph
- build.xml
- graph.ini
- runsettings.ini

UI plug-in install location:

- help
- about.html
- plugin.xml
- ui.jar

Feature install location:

- feature.properties
- feature.xml
- license.html

Figure 6.6: LightUML directory structures.

file is stored here. Discussing feature branding is out of the scope of this case study.

- `about.properties`. Contains externalized strings for `about.ini`.
- `build.xml`. An Ant buildfile generated by Eclipse.
- `core.jar`. Contains the Java classes of the plug-in.
- `plug-in.xml`. The plug-in manifest. See Appendix A.2.1.

State location of the core plug-in contains the following files and directories (after it has been initialized by the core plug-in):

- `graph/` A directory where the temporary files in graph building are stored before the class diagram is added to the Java project.
- `build.xml`. The Ant buildfile (used by `JavaToDot` and `LocalGraphConverter`). See Appendix A.2.2.
- `graph.ini`. The properties file for properties used in graph building.
- `runsettings.ini`. Contains information delivered to Ant about a specific run of UMLGraph.

Install location of the UI plug-in contains the following files and directories:

- `help/` A directory containing the following help files: `introduction.html`, `legal.html`, `resources.html`, `toc.xml`, and `using.html`. The `toc.xml` file is a table of contents file that is added to the Eclipse help system using the `org.eclipse.help.toc` extension point. The other files provide the help content.
- `about.html`. Contains the copyright notice and the license agreement.
- `plugin.xml`. The plug-in manifest of the UI plug-in. See Appendix A.3.1.
- `ui.jar`. JAR-package containing the Java implementation classes of the plug-in.

Install location of the feature contains following files:

- `feature.properties`. Contains text version of the top-level user agreement, which is SUA (see Section 3.3).
- `feature.xml`. The feature manifest (see Appendix A.1).
- `license.html`. Contains the top-level user agreement in HTML (HyperText Markup Language).

6.4 Design Rationale and Development Notes

This section provides some discussion on the rationale of LightUML's design, some general issues about plug-in development, and a few development notes about LightUML that came across during the development process.

6.4.1 Design Rationale

The division of LightUML into two distinct plug-ins follows the Eclipse convention, other plug-in containing the core functionality and the other containing UI related functionality. See Strata Rule (see Table 5.1, rule number 24). Packaging these plug-ins into a feature is a convenient way to deploy LightUML. Thus, LightUML can be installed and updated with the Update Manager of Eclipse when hosted on a web site.

Dividing the toolchain into sequential steps gives more control over the execution. Encapsulating these steps into Eclipse `Job`-classes is a convenient way to enable concurrency with other operations that might be running and adding responsiveness to the user interface, including a progress monitor with the chance to cancel the execution of the toolchain.

To keep the Eclipse workspace synchronized, the Eclipse resource model has been used to add the generated class diagram into the project. Before adding the class diagram, the project has to be locked as multiple jobs might want to access the same resource. This has been achieved by providing an additional scheduling rule on `AddGraphToProject`-class. The generated file is marked as derived, so it will not get included in possible version controlling.

Using Apache Ant integration of Eclipse to run the external tools, the `dot` executable and the UMLGraph doclet, seems to be the most convenient way available, opposed to using a built-in execution mechanisms of Java. At least, it gives a more experienced end-user the possibility to customize the Ant buildfile.

Using the plug-in state location for storing the Ant buildfile, the properties used in graph generation, and the temporary files created, seems quite natural. At least the user's workspace will not get polluted with temporary files.

Preference pages are a good place to allow the customization of LightUML behaviour. Almost everything is parameterized in the buildfile used in Ant runs, and corresponding preferences are provided on a preference page for customization to achieve higher flexibility of use.

LightUML does not deliver UMLGraph as part of its content. It has to be downloaded separately and the path to the doclet has to be provided via LightUML preference pages. Another option would have been to deliver UMLGraph encapsulated as its own plug-in. However, there are some problems with this approach. Updating the UMLGraph plug-in as new versions of UMLGraph appear would have been needed. Also, delivering the UMLGraph doclet path from the UMLGraph plug-in to the LightUML core plug-in would have been an issue to assess. Delivering UMLGraph with LightUML is a bit questionable anyway, as UMLGraph is developed completely separate from LightUML.

6.4.2 General Plug-in Development Notes

- The install location of an Eclipse plug-in should not be referenced in any way in the source code. `Plugin.openStream` can be used to access files under the plug-in install location. This also works well when using PDE—`openStream` opens a stream relative to the development location of the plug-in.

- Eclipse resource model can be used only for file handling under the workspace. Basic Java file handling can be used elsewhere.

- Location of a resource should be accessed with `IResource.getLocation`. A resource is not always under the workspace directory.

- If a job is canceled can be detected only via progress monitors, `Job` does not provide a method for checking this.

- Testing a plug-in on different operating system platform can reveal undetected problems. Especially Apache Ant seems to work slightly differently on different platforms.

6.4.3 LightUML Development Notes

- LightUML provides no extension points of its own. This does not fulfill the Invitation Rule (rule 8 of Table 5.1). No obvious ways to invite other developers were found. Because of this, no extension processing of extension points was examined in practice.

- Dependency on the previously installed GraphViz is a bit awkward. One way around this would be to use a WebDot server for the "dot to graphics" conversion. WebDot is a CGI program that converts a graph description from a `.dot` file into an image that can be included on a web page [56]. However, using a remote WebDot server would add a dependency on a functioning server. On the other hand, integrating a non-Java tool was also examined now.

- The preferences shown in Figures 6.9, 6.10, and 6.11 are preferences in the preference store of the UI plug-in. When UI plug-in is loaded, it restores default preferences from the graph build properties file (see Appendix A.2.3). To cascade the properties changed in the preference page into properties used by Ant runs, a property listener is used (see Appendix A.3.2, line 39).

- LightUML uses SUA as its top-level agreement for the feature `org.lightuml`. The plug-ins are delivered with the original BSD license, which is also used with UMLGraph. Using SUA is the Eclipse convention and it provides some licensing flexibility if other developers want to contribute only to a specific part of LightUML, e.g. the UI.

- The "Monkey see / monkey do"-rule (see rule 5 of Table 5.1) could not be fulfilled properly. No similar plug-ins were found which used a Javadoc doclet as part of the toolchain to be executed.

- LightUML is still under development, so some inconsistencies and targets for refactoring exist.

LightUML was developed through numerous iterations by debugging, refactoring, and adding more functionality after each iteration. However, the importance of testing during development became clear only after practicing a few months of plug-in development. Especially detecting platform dependent code seems to require thorough testing. As plug-in development also seems to be very iterative by nature, using an agile methodology like test-driven development (TDD) [3] would seem to be a natural way to develop plug-ins. Also, the plug-in development environment of Eclipse (PDE) provides a specific JUnit plug-in unit test framework (see [27] for test-driven plug-in development).

6.5 Web Resources

- LightUML is hosted at SourceForge.
  - The home page is at:
    `http://lightuml.sourceforge.net`
  - The SourceForge project page is at
    `http://sourceforge.net/projects/lightuml`
- UMLGraph (and its very useful documentation) can be found at
  `http://www.spinellis.gr/sw/umlgraph/`
- GraphViz, a prerequisite for using LightUML, can be found at
  `http://www.graphviz.org/`

6.6 Summary

This chapter provided a case study of Eclipse tool integration called LightUML. LightUML integrated UMLGraph doclet and GraphViz `dot` tool, producing a seamless toolchain that generates a UML class diagram in a graphical format from a Java project or package. The capability to generate class diagrams was made available to an Eclipse user only when it is relevant, that is, when the user has selected a Java project or package. Also, the user was provided with the chance to customize the behaviour of LightUML via preference pages. The structure of LightUML was examined from feature, plug-in, Java package, and Java class points of view. Also, the directory structures and the design rationale of LightUML were examined and some development notes were listed. And lastly, a list of relevant web resources was given.

LightUML is a practical example of tool construction and integration in Eclipse. LightUML is constructed using the Eclipse plug-in model and thus it interfaces into Eclipse using *loosely-coupled interfacing paradigm* (see sections 2.2 and 4.2). LightUML achieves *presentation integration* by adding elements into the Eclipse workbench. These elements include additions in the menus and the preference pages. *Data integration* is achieved by adding the generated diagram into the project in question via the workspace API. LightUML does not provide any means for other tools to use the services provided by it. Thus, only one-way *control integration* is achieved by running Ant scripts to execute the external tools (no *provision*, see Section 2.3.3). LightUML should be classified as a *tool*, as it supports only a single task of generating class diagrams from Java source files. The Fuggetta's classification presented earlier in Section 2.1.1 does not consider reverse-engineering tools by themselves, but LightUML could be forged into a *reverse-engineering workbench* if support for UMLGraph sequence diagrams would be developed and integrated tightly with the Eclipse workbench. Connecting Eclipse Java model with the declarative syntax of UMLGraph's sequence diagrams could be an interesting task altogether. With the support of proper UI components the integration could propably enable relatively effortless reverse engineering of selected message sequences via UMLGraph sequence diagrams. However, LightUML support for sequence diagrams is still on experimental level and not considered in this case study.

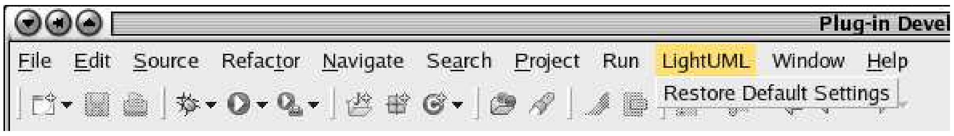

Plug-in Devel
File Edit Source Refactor Navigate Search Project Run LightUML Window Help
Restore Default Settings

Figure 6.7: LightUML pulldown menu contribution.

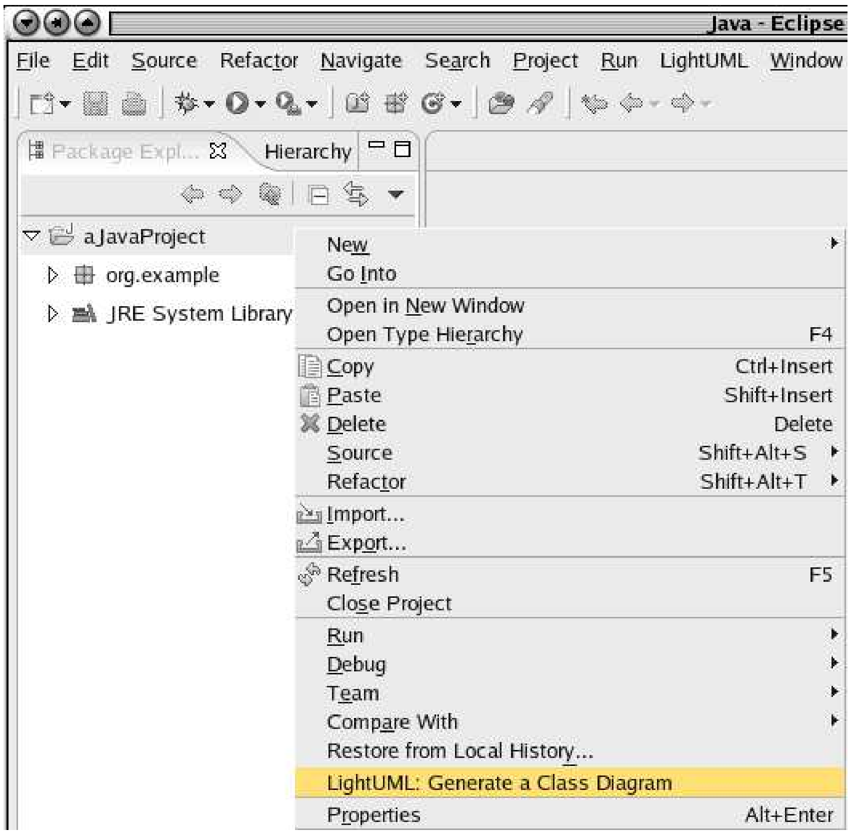
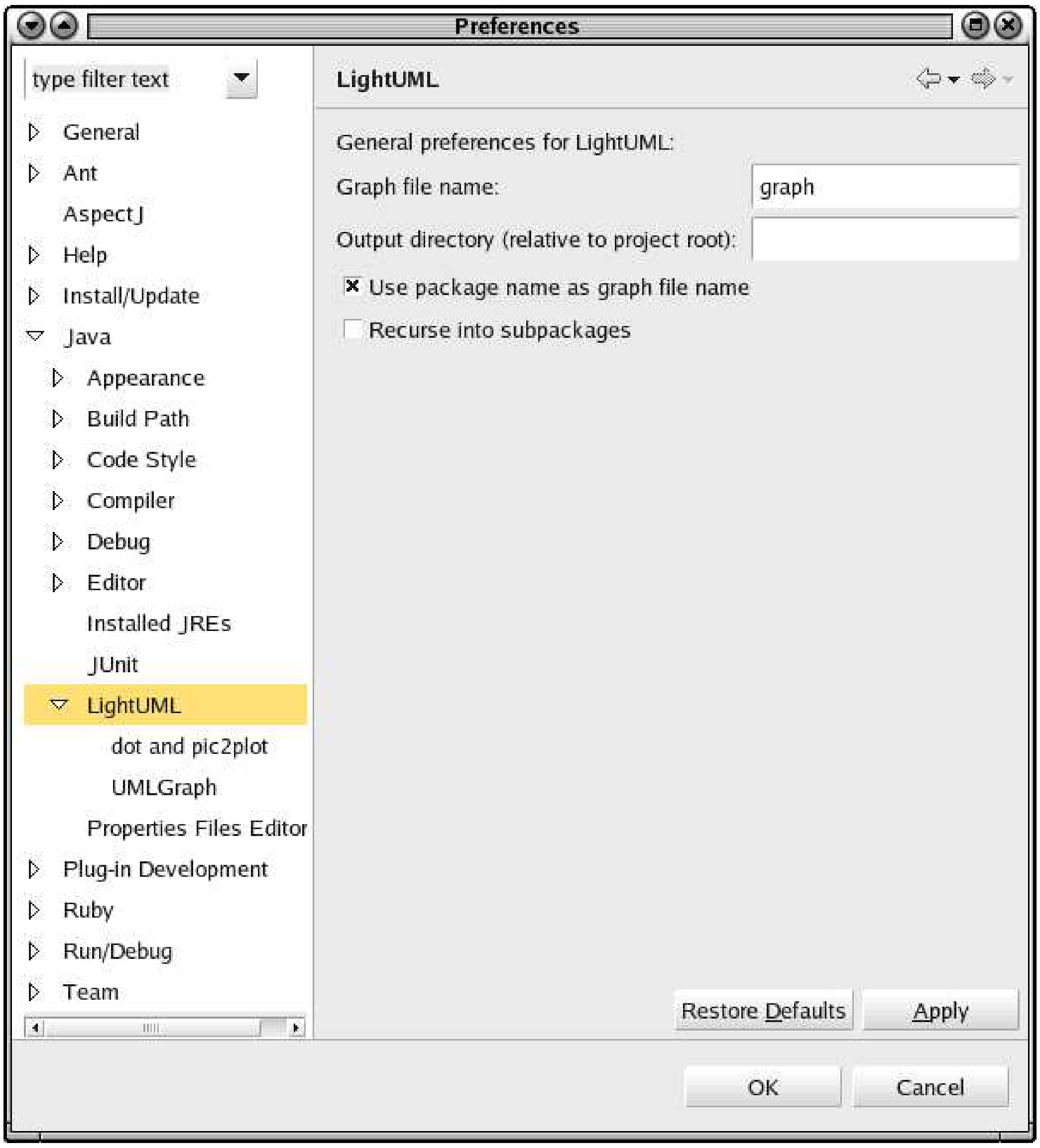


Figure 6.8: LightUML popup menu contribution.



LightUML

General preferences for LightUML:

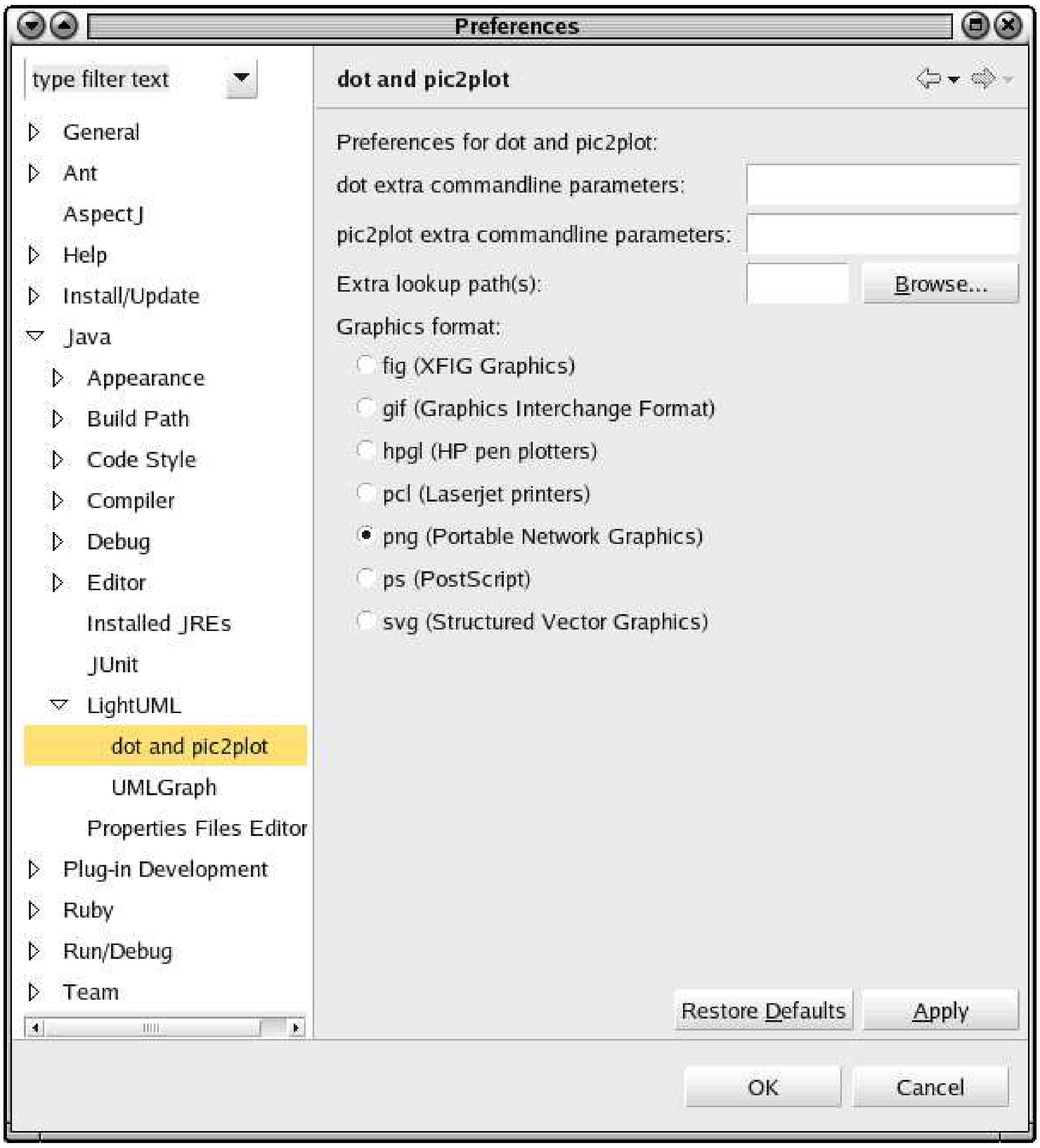
Graph file name: graph

Output directory (relative to project root):

[✓] Use package name as graph file name

[ ] Recurse into subpackages

Figure 6.9: LightUML preference page contribution, general preferences.



dot and pic2plot

Preferences for dot and pic2plot:

dot extra commandline parameters:

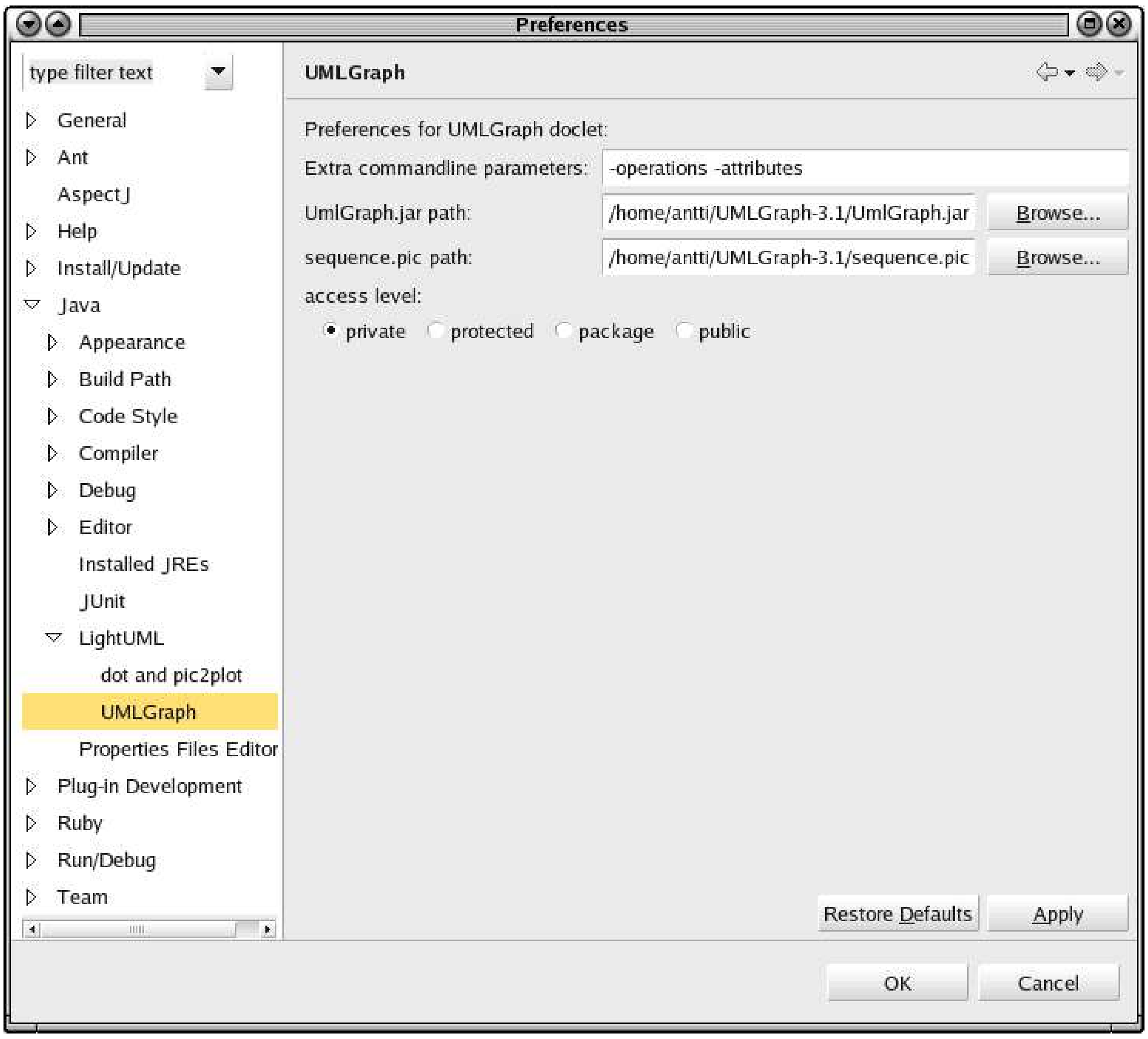
pic2plot extra commandline parameters:

Extra lookup path(s):

Graphics format:

- fig (XFIG Graphics)
- gif (Graphics Interchange Format)
- hpgl (HP pen plotters)
- pcl (Laserjet printers)
- [✓ png (Portable Network Graphics)]
- ps (PostScript)
- svg (Structured Vector Graphics)

Figure 6.10: LightUML preference page contribution, preferences for the dot executable.



Preferences for UMLGraph doclet:

Extra commandline parameters: -operations -attributes

UmlGraph.jar path: /home/antti/UMLGraph-3.1/UmlGraph.jar

sequence.pic path: /home/antti/UMLGraph-3.1/sequence.pic

access level: [✓ private] protected package public

Figure 6.11: LightUML preference page contribution, UMLGraph preferences.

7 Conclusion

As a theoretical framework, a tool-centric point of view to CASE systems was provided. This framework considered CASE classification, tool interfacing, the levels of tool integration, and the key challenges of CASE systems. Eclipse was introduced as a state-of-the-art CASE system to be placed in this framework. Tool interfacing and the levels of integration were considered in respect of tool encapsulation in Eclipse. As there exists no exact CASE system classification criteria, it was proposed that Eclipse should be classified as an integrated environment. It was found to be the best match characteristic-wise. The technical point of view to tool construction and encapsulation in Eclipse was also examined. Plug-ins, their relationships, and declaring and processing those relationships were examined from the plug-in developer's point of view. In addition, a big contributor to Eclipse's flexibility, the Extension Object pattern, was introduced. The general plug-in development guidelines were also given as a table. As a practical example, a case study of tool integration and construction in Eclipse was provided. This case study took the technical considerations into practice by introducing a tool called LightUML. Also, multiple points of view to the structure of an Eclipse extension were considered. Furthermore, LightUML was considered in terms of the earlier theoretical framework.

Integrating a tool into a framework brings many benefits, but it also has its price. Many benefits can be gained by integrating tools into Eclipse, including consistent appearance, behaviour, and interaction paradigm with other integrated tools, a common resource model, a common way to offer and use tool services, a huge selection of tool services to derive from, portability to various operating system platforms with (almost) no additional effort, the ability to deploy integrated tools as stand-alone applications, millions of potential users, a huge community to ask help from, etc. The price to pay is an old acquaintance: context-specific result. The integration can not be easily reused in other environments. In addition, the integration has to be done in Java. Besides, getting to know all the various frameworks needed in the integration takes time and is quite hard work.

Eclipse evolves fast. During the writing of this thesis, Eclipse has evolved from version 3.0 to 3.1.1, and it has also become dominant as a Java IDE [31]. So much is happening that it is virtually impossible to keep up with the development. This thesis has provided an introduction to Eclipse as a state-of-the-art tool integration platform built on a pure plug-in architecture, and it has been put into a context. Thousands of pages of guides have been written about Eclipse, so work presented here has only been able to pinpoint some issues that are relevant in tool integration practice. However, after a few years time it is possible that many things have changed so much in Eclipse that parts of this thesis regarding Eclipse have only some historical value.

8 References

[1] Anderson, J. Levels of Integration: Five ways you can integrate with the Eclipse Platform. Object Technology International, Mar. 2001. URL `http://www.eclipse.org/articles/Article-Levels-Of-Integration/Levels%20Of%20Integration.html`

[2] AspectJ Project Home Page. URL `http://www.eclipse.org/aspectj/`

[3] Beck, K. Test-Driven Development By Example. Addison Wesley, 2002.

[4] Birsan, D. "On Plug-ins and Extensible Architectures" in *Queue*, Volume 3, Issue 2. pp. 40 - 46. ACM Press, Mar. 2005.

[5] Bolour, A. Notes on the Eclipse Plug-in Architecture. Bolour Computing, Jul. 2003. URL `http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html`

[6] Bouillon, P., Krinke, J. Using Eclipse in Distant Teaching of Software Engineering in *Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. pp. 22 - 26. ACM Press, 2004.

[7] Brooks, F. No Silver Bullet: Essence and Accidents of Software Engineering in *Computer*, Volume 20, Issue 4. pp. 10 - 19. ACM Press, Apr. 1987.

[8] Brown, A. W., Earl, A. N., McDermid J. A. Software Engineering Environments: Automated Support for Software Engineering. McGraw-Hill, 1992.

[9] Brown, A. W., Feiler, P. H., Wallnau, K. C. Understanding Integration in a Software Development Environment. Technical Report CMU/SEI-91-TR-31 ESD-TR-91-31. Software Engineering Institute, 1992.

[10] Brown, A. W., McDermid, J. A. "Learning from IPSE's Mistakes" in *IEEE Software*, Volume 9, Issue 2. pp. 23 - 28. IEEE Press, Mar. 1992.

[11] Chen, M., Norman, R. J. "A Framework for Integrated CASE" in *IEEE Software,* Volume 9 Issue 2. pp. 18 - 22. EEE Press, Mar. 1992.

[12] D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P. The Java Developer's Guide to Eclipse, 2nd Ed. Addison-Wesley, 2005.

[13] Eclipse Foundation. Eclipse Data Tools Platform Project. Homepage URL `http://www.eclipse.org/datatools`

[14] Eclipse Foundation. Eclipse Platform Plug-in Developer Guide. Available via Eclipse platform help system. Also available at URL `http://help.eclipse.org`

[15] Eclipse Foundation. Eclipse Project FAQ. URL `http://www.eclipse.org/eclipse/faq/eclipse-faq.html`

[16] Eclipse Foundation. EMF - Eclipse Modeling Framework. Homepage URL `http://www.eclipse.org/emf/`

[17] Eclipse Foundation. Guide to Legal Documentation for Eclipse-based Content. Available at URL `http://www.eclipse.org/legal/guidetolegaldoc.html`

[18] Eclipse Foundation. Homepage of the Eclipse Foundation. URL `http://www.eclipse.org/`

[19] Eclipse Foundation. UML2 - EMF-based UML2.0 Metamodel Implementation. Homepage URL `http://www.eclipse.org/uml2/`

[20] Eclipse Foundation Software User Agreement. URL `http://www.eclipse.org/legal/epl/notice.html`

[21] Eclipse Public License Version 1.0. URL `http://www.eclipse.org/legal/epl-v10.html`

[22] Elrad, T., Filman, R. E., Bader, A. "Aspect-oriented programming: Introduction" in *Communications of the ACM*, volume 44, issue 10. pp. 29 - 32. ACM Press, Oct. 2001.

[23] Fowler, M. "Public versus Published Interfaces" in *IEEE Software*, Volume 19, Issue 2. pp. 18 - 19. IEEE Press, Mar. - Apr. 2002.

[24] Fuggetta, A. "Classification of CASE Technology" in *IEEE Computer,* Volume 26 Issue 12. pp. 25 - 38. IEEE Press, Dec. 1993.

[25] Gallardo, D., Burnette, E., McGovern, R. Eclipse In Action: A guide for Java developers. Manning, 2003.

[26] Gamma, E., Beck, K. Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Addison-Wesley, 2004.

[27] Gamma, E., Beck, K. "Test-Driven Plug-In Development" in Contributing to Eclipse: Principles, Patterns, and Plug-Ins. Chapter 12. Addison-Wesley, 2004. URL `http://today.java.net/today/2004/02/02/ch12Eclipse.pdf`

[28] Gamma, E. "Extension Object", in *Pattern Languages of Program Design 3*, eds. R. Martin, D. Riehle, F. Buschmann. Addison Wesley Longman, 1998.

[29] Gamma et al. Design Patterns: elements of reusable object-oriented software. Addison-Wesley, 1995.

[30] Gane, C. Computer-aided Software Engineering: the Methodologies, the Products, and the Future. Prentice-Hall, 1990.

[31] Geer, D. Eclipse Becomes the Dominant Java IDE in *IEEE Computer*, Volume 38, Issue 7. pp. 16 - 18. IEEE, July 2005.

[32] GraphViz. Homepage URL `http://www.graphviz.org`

[33] Harrison, W., Ossher, H., Tarr, P. "Software Engineering Tools and Environments: A Roadmap" in *Proceedings of the Conference on The Future of Software Engineering*. pp. 261 - 277. ACM Press, 2000.

[34] IBM Corporation and others. Eclipse project briefing materials. Eclipse.org, 2002, 2003. URL `http://eclipse.org/eclipse/presentation/eclipse-slides.pdf`

[35] IEEE Standard Glossary of Software Engineering Terminology. IEEE Press, 1990.

[36] Kestler, M. Factoring for Eclipse: Plug-in design techniques. Dr. Dobb's Journal, Nov. 2004. pp. 78 - 82.

[37] Linthicum, D. S. Enterprise Application Integration. Addison-Wesley, 2000.

[38] McAffer, J. Inside and Beyond the Eclipse 3.0 Runtime. IBM OTI Labs, Jan. 2004. URL `http://dev.eclipse.org/viewcvs/index.cgi/%7Echeckout%7E/platform-core-home/runtime/runtime.html`

[39] Mi, P., Scacchi, W. "Process Integration in CASE environments" in *IEEE Software,* Volume 9, Issue 2. pp. 45 - 53. IEEE Press, Mar. 1992.

[40] Northover, S. SWT: The Standard Widget Toolkit, Part 1: Implementation Strategy for Java Natives. Object Technology International, Mar. 2001. URL `http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html`

[41] Object Management Group. Unified Modeling Language. URL `http://www.uml.org`

[42] Object Technology International, Inc. Eclipse Platform Technical Overview. Feb. 2003. URL `http://www.eclipse.org/whitepapers/eclipse-overview.pdf`

[43] OSGi Alliance Home page. URL `http://www.osgi.org`

[44] Pressman, R. S. Software Engineering: A Practitioner's Approach (European adaptation, 5th ed.). McGraw-Hill, 2000.

[45] Programming Ruby: The Pragmatic Programmer's Guide, First Edition. URL `http://www.rubycentral.com/book/`

[46] Shaik, S., Corvin, R., Sudarsan, R., Javed, F., Ijaz, Q., Roychoudhury, S., Gray, J., Bryant, B. SpeechClipse: an Eclipse speech plug-in in *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*. pp. 84 - 88. ACM Press, 2003.

[47] Sharon, D., Bell, R. "Tools that Bind: Creating Integrated Environments" in *IEEE Software,* Volume 12 Issue 2. pp. 76 - 85. IEEE Press, Mar. 1995.

[48] Sommerville, I. Software Engineering (5th ed.). Addison-Wesley, 1995.

[49] Spinellis, D. "On the Declarative Specification of Models" in *IEEE Software*, Volume 20, Issue 2. pp. 96 - 95. IEEE Press, Mar. - Apr., 2003.

[50] Sun Microsystems Inc. "Javadoc 5.0 Tool" in JDK 5.0 Documentation. URL `http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/index.html`

[51] Tarr, P., Ossher, H., Harrison, W., Sutton, S. M. Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns" in *Proceedings of the 21st international conference on Software engineering*. pp. 107 - 119. IEEE Computer Society Press, 1999.

[52] Thomas, I., Nejmeh, B. A. "Definitions of Tool Integration for Environments" in *IEEE Software,* Volume 9 Issue 2. pp. 29 - 35. IEEE, Mar. 1992.

[53] UMLGraph Home Page. URL `http://www.spinellis.gr/sw/umlgraph/`

[54] Vaughan-Nichols, S. J. "The Battle over the Universal Java IDE" in *IEEE Computer*, Volume 36 Issue 4. pp. 21 - 23. ACM Press, Apr. 2003.

[55] Wallnau, K. C., Feiler, P. H. Tool Integration and Environment Architectures. Technical Report CMU/SEI-91-TR-011 ESD-91-TR-011. Software Engineering Institute, May 1991.

[56] WebDot Home page. URL `http://www.graphviz.org/webdot/`

[57] Wright, D. Launching Java Applications Programmatically. IBM OTI Labs, Aug. 2003. URL `http://www.eclipse.org/articles/Article-Java-launch/launching-java.html`

[58] Yang, Y., Han, J. "Classification of and Experimentation on Tool Interfacing in Software Development Environments" in *Proceedings of the Software Engineering Conference, 1996*. pp. 56 - 65. IEEE Press, Dec. 1996.

[59] Yang, Y., Welsh, J., Allison, W. "Supporting Multiple Tool Integration Paradigms within a Single Environment" in *CASE '93, Proceedings of the Sixth International Workshop on Computer-Aided Software Engineering*. pp. 364 - 374. IEEE Press, Jul. 1993.

[60] Zarrella, P.F. CASE Tool Integration and Standardization. Technical Report CMU/SEI-90-TR14, ESD-TR-90215. Software Engineering Institute, Dec. 1990.

[61] Zarrella, P. F., Smith, D. B., Morris, E. J. Issues in Tool Acquisition. Technical Report CMU/SEI-91-TR-008 ESD-TR-91-008. Software Engineering Institute, Sep. 1991.

All URLs have been checked to be valid on 30th of August, 2005.

A LightUML Source

This appendix provides LightUML source code, including contents of plug-in manifests, Java source files, and some other essential files. The feature manifest is presented first. The source code is listed per plug-in basis, first for the core plug-in and then for the UI plug-in. Source code in the base package is listed first, then subpackages are listed breadth-first style in the alphabetical order. Some source files contain UMLGraph control elements that were used in generating the class diagrams in this thesis and the license notes have been omitted. Also, some preliminary support for UMLGraph sequence diagrams has been developed.

A.1 Feature manifest (`feature.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<feature
      id="org.lightuml"
      label="LightUML"
      version="1.2.1"
      provider-name="Antti␣Hakala"
      plugin="org.lightuml.core">
   <install-handler/>

   <description>
      LightUML provides a lightweight option for generating class diagrams out of a Java
           project or package. It integrates UmlGraph doclet (by Diomidis Spinellis) via
           Apache Ant and uses external GraphViz &apos;dot&apos; tool to convert .dot files
           to graphical form. Read: you need to have GraphViz installed.
   </description>

   <license url="%licenseURL">
      %license
   </license>

   <url>
      <update label="LightUML" url="http://lightuml.sourceforge.net/updatesite"/>
   </url>

   <requires>
      <import plugin="org.eclipse.ant.core"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.ui"/>
```

```
        <import plugin="org.eclipse.jdt.core"/>
    </requires>

    <plugin
            id="org.lightuml.core"
            download-size="0"
            install-size="0"
            version="1.2.1"/>

    <plugin
            id="org.lightuml.ui"
            download-size="0"
            install-size="0"
            version="1.2.1"/>

</feature>
```

A.2 `org.lightuml.core`

A.2.1 Plug-in Manifest (`plugin.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
    id="org.lightuml.core"
    name="LightUML"
    version="1.2.1"
    class="org.lightuml.core.LightUMLCorePlugin"

    provider-name="Antti␣Hakala">

    <runtime>
        <library name="core.jar">
            <export name="*"/>
        </library>
    </runtime>
    <requires>
        <import plugin="org.eclipse.ant.core"/>
        <import plugin="org.eclipse.core.runtime"/>
        <import plugin="org.eclipse.core.resources"/>
    </requires>
</plugin>
```

A.2.2 Ant Buildfile (`build.xml`)

```
<?xml version="1.0"?>
<!-- ==================================================================
     Ant buildfile for LightUML plug-in.
     ================================================================== -->

<project name="lightuml.buildfile" default="java-to-dot" basedir=".">
     <description>
```

```
        This ant buildfile is for running the UmlGraph doclet
        and GraphViz dot-tool.
    </description>

    <!-- Set the properties in the property file graph.ini-->
    <property file="graph.ini" />

    <!--get ${statelocation} from location of antfile -->
    <dirname property="statelocation" file="${ant.file}"/>

    <!-- ================================
            target: java-to-dot
            run UmlGraph to convert .java files
            to a .dot file

            needs ${source-path}

         ================================ -->

    <target name="java-to-dot">
        <!-- load setting for this run -->
        <property file="runsettings.ini" />
        <!-- run Javadoc -->
        <javadoc
            docletpath="${umlgraph-jar-path}"
            access="${javadoc-access-level}"
            useexternalfile="yes"
            additionalparam="${umlgraph-extra-param}"
            failonerror="true">

            <doclet name="UmlGraph">
                <param name="-output" value="${statelocation}/graph/graph.dot"/>
            </doclet>

            <fileset dir="${source-path}">
                <include name="${scope}"/>
            </fileset>
        </javadoc>
    </target>

    <!-- ================================
            target: dot-to-graphics

         convert .dot to some graphics form
         using locally installed dot tool
         ================================ -->

    <target name="dot-to-graphics">
        <property environment="env"/>
        <exec
            executable="dot"
            searchpath="true"
            errorproperty="dot.error">

            <env key="PATH" path="${extra-lookup-path}:${env.PATH}"/>
            <arg line="${dot-extra-param}"/>
            <arg value="-T${graphics-format}"/>
            <arg value="-ograph/graph.${graphics-format}"/>
            <arg value="graph/graph.dot"/>
        </exec>
        <fail message="Error␣executing␣Graphviz␣'dot'␣---␣${dot.error}">
            <condition>
                <length string="${dot.error}" when="greater" length="0"/>
            </condition>
        </fail>
    </target>

    <!-- ================================
            target: pic-to-graphics

         convert .pic to some graphics form
         using locally installed pic2plot tool
         ================================ -->

    <target name="pic-to-graphics">
        <property environment="env"/>
        <property file="runsettings.ini" />
        <copy file="${pic-macros-path}" tofile="sequence.pic"/>
        <exec
            executable="pic2plot"
            searchpath="true"
            output="graph/graph.${graphics-format}"
            errorproperty="pic.error">

            <env key="PATH" path="${extra-lookup-path}:${env.PATH}"/>
            <arg line="${pic2plot-extra-param}"/>
            <arg value="-T${graphics-format}"/>
            <arg value="${pic-file-path}"/>
        </exec>
        <fail message="Error␣executing␣plotutils␣'pic2plot'␣---␣${pic.error}">
            <condition>
                <length string="${pic.error}" when="greater" length="0"/>
            </condition>
        </fail>
    </target>

</project>
```

A.2.3 Graph Build Properties (`graph.ini`)

```
dot-extra-param=
extra-lookup-path=
graph-file-name=graph
graphics-format=png
javadoc-access-level=private
pic-macros-path=
pic2plot-extra-param=
project-output-dir=
recurse-packages=false
umlgraph-extra-param=-operations -attributes
umlgraph-jar-path=
use-package-name=true
```

A.2.4 `IBuildConstants`

```
package org.lightuml.core;

import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.Path;

/**
 * <p>
 * Constant filenames and paths.
 * </p>
 *
 * @author Antti Hakala
 * @hidden
 */
public interface IBuildConstants {
    public IPath BUILD_FILE_NAME = new Path("build")
            .addFileExtension("xml");

    // without the extension which is dynamic
    public IPath INTERNAL_GRAPH_FILE_NAME = new Path("graph");

    // Graph file name used internally when generating graphs.
    // When the graph is added to the project, the name is taken
    // from the preferences.
    public IPath INTERNAL_DOT_FILE_NAME = INTERNAL_GRAPH_FILE_NAME
            .addFileExtension("dot");

    // this is defined in build.xml too..
    public IPath GRAPH_BUILD_PROPERTIES_FILE_NAME = new Path(
            "graph").addFileExtension("ini");

    // files to be copied to statelocation are stored here
    public IPath TO_STATELOCATION_DIR = new Path("to_statelocation");

    // statelocation related stuff
    public IPath STATELOC_OUTPUT_DIR = new Path("graph");
    public IPath STATELOC_RUNSETTINGS = new Path("runsettings")
            .addFileExtension("ini");

}
```

A.2.5 `IErrorMessages`

```
package org.lightuml.core;

```

```
/**
 * <p>
 * Error messages.
 * </p>
 *
 * @author anthakal
 * @hidden
 */
public interface IErrorMessages {
    public String ERRMSG_NO_UMLGRAPH_JAR = "No UmlGraph.jar found. See that you have set 
        the UmlGraph.jar path in Preferences > Java > LightUML > UMLGraph.";

    public String ERRMSG_NO_PIC_MACROS = "No sequence.pic found! See that you have set the 
        sequence.pic path in Preferences > Java > LightUML > UMLGraph.";
}
```

A.2.6 `IGraphBuildProperties`

```
package org.lightuml.core;

/**
 * <p>
 * Property names used in the graph build properties file.
 * </p>
 *
 * @author Antti Hakala
 * @hidden
 */
public interface IGraphBuildProperties {

    // property names (used in build.xml also)
    public String P_GRAPHICS_FORMAT = "graphics-format";

    public String P_GRAPH_FILE_NAME = "graph-file-name";

    public String P_PROJECT_OUTPUT_DIR = "project-output-dir";

    public String P_EXTRA_LOOKUP_PATH = "extra-lookup-path";

    public String P_DOT_EXTRA_PARAM = "dot-extra-param";

    public String P_PIC2PLOT_EXTRA_PARAM = "pic2plot-extra-param";

    public String P_UMLGRAPH_EXTRA_PARAM = "umlgraph-extra-param";

    public String P_USE_PACKAGE_NAME = "use-package-name";

    public String P_RECURSE_PACKAGES = "recurse-packages";

    public String P_JAVADOC_ACCESS_LEVEL = "javadoc-access-level";

    public String P_UMLGRAPH_JAR_PATH = "umlgraph-jar-path";

    public String P_PIC_MACROS_PATH = "pic-macros-path";

    public String[][] OUTPUT_FORMATS = { { "fig␣(XFIG␣Graphics)", "fig" },
            { "gif␣(Graphics␣Interchange␣Format)", "gif" },
            { "hpgl␣(HP␣pen␣plotters)", "hpgl" },
            { "pcl␣(Laserjet␣printers)", "pcl" },
            { "png␣(Portable␣Network␣Graphics)", "png" },
            { "ps␣(PostScript)", "ps" },
            { "svg␣(Structured␣Vector␣Graphics)", "svg" } };

    public String[][] ACCESS_LEVELS = { { "private", "private" },
            { "protected", "protected" }, { "package", "package" },
            { "public", "public" } };
}
```

A.2.7 IGraphConverter

```
package org.lightuml.core;

import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.IProgressMonitor;

/**
 * <p>
 * An interface for converting .dot and .pic files to graphical form
 * </p>
 * <p>
 * Output has to be of format
 * LightUMLCorePlugin.getGraphBuildProperty(P_GRAPHICS_FORMAT).
 * P_GRAPHICS_FORMAT is found in interface IGraphBuildProperties.
 * </p>
 * <p>
 * A class implementing this interface also has to return the location of
 * generated graphics file with getGraphicsFilePath().
 * </p>
 * <p>
 * Progress monitor is provided for canceling support.
 * </p>
 *
 * @author Antti Hakala
 */
public interface IGraphConverter {
    /**
     * <p>
     * Convert .dot file to some graphical form for viewing.
     * </p>
     *
     * @param pm
     *            A progress monitor to use.
     * @throws LightUMLCoreException
     */
    public void dotToGraphics(IPath dotFilePath, IProgressMonitor pm)
            throws LightUMLCoreException;

    /**
     * <p>
     * Convert a .pic file to graphical form.
     * </p>
     *
     * @param pm
     *            A progress monitor to use.
     * @throws LightUMLCoreException
     * @hidden
     */
    public void picToGraphics(IPath picFilePath, IProgressMonitor pm)
            throws LightUMLCoreException;

    /**
     * <p>
     * Get path to the converted graphics file.
     * </p>
     *
     * @return the path to the (coverted) graphics file
     */
    public IPath getGraphicsFilePath() throws LightUMLCoreException;
}
```

A.2.8 LightUMLCoreException

```
package org.lightuml.core;

/**
 * <p>
 * An exception of LightUMLCorePlugin.
 * </p>
 *
 * @author Antti Hakala
 */
public class LightUMLCoreException extends Exception {
    private static final long serialVersionUID = -7915803826993423881L;

    /**
     * <p>
     * Contructor.
     * </p>
     *
     * @param e
     *            The cause.
     */
    public LightUMLCoreException(Throwable e) {
        super(e);
    }

    /**
     * <p>
     * Constructor with a message.
     * </p>
     *
     * @param message
     *            Exception message.
     */
    public LightUMLCoreException(String message) {
        super(new Throwable(message));
    }
}
```

A.2.9 LightUMLCorePlugin

```
package org.lightuml.core;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

import org.eclipse.ant.core.AntRunner;
import org.eclipse.core.resources.IFile;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.Plugin;
import org.eclipse.core.runtime.jobs.IJobChangeListener;
import org.lightuml.core.jobs.AddGraphToProject;
import org.lightuml.core.jobs.DotToGraphics;
import org.lightuml.core.jobs.Initialize;
import org.lightuml.core.jobs.JavaToDot;
import org.lightuml.core.jobs.PicToGraphics;
import org.osgi.framework.BundleContext;

/**
 * <p>
 * The "core" of LightUML. A plug-in that directs the execution of UmlGraph and
 * other external tools.
 * </p>
 *
 * @author Antti Hakala
 * @has 1 - 1 org.lightuml.core.IGraphConverter
 * @navassoc 1 throws * org.lightuml.core.LightUMLCoreException
 */
public class LightUMLCorePlugin extends Plugin implements IBuildConstants,
        IErrorMessages {
    /**
     * <p>
     * The static plugin instance.
     * </p>
     */
    private static LightUMLCorePlugin plugin = null;

    /**
     * <p>
     * The graph converter to use.
     * </p>
     */
    private IGraphConverter graphConverter;

    /**
     * <p>
     * Properties for graph building.
     * </p>
     */
    private Properties graphBuildProperties;

    /**
     * <p>
     * Get the path to buildfile.
     * </p>
     *
     * @return A path to buildfile.
     */
    public IPath getBuildFilePath() {
        return getStateLocation().append(BUILD_FILE_NAME);
    }

    /**
     * <p>
     * Get the path to graph build properties.
     * </p>
     *
     * @return A path to properties file.
     */
    public IPath getPropertiesFilePath() {
        return getStateLocation().append(GRAPH_BUILD_PROPERTIES_FILE_NAME);
    }

    /**
     * <p>
     * Convenience method
     * </p>
     *
     * @throws LightUMLCoreException
     * @return path representing the location where the .dot file is (to be)
     *         generated.
     */
    public IPath getDotFilePath() {
        return getStateLocation().append(STATELOC_OUTPUT_DIR).append(
                INTERNAL_DOT_FILE_NAME);
    }

    /**
     * <p>
     * Constructor.
     * </p>
     */
    public LightUMLCorePlugin() {
        super();
        plugin = this;
        setGraphConverter(LocalGraphConverter.instance());
        graphBuildProperties = new Properties();
    }

    /**
     * <p>
     * Try to load the graph build properties when the plug-in is started.
     * </p>
     *
     * @see Plugin#start(org.osgi.framework.BundleContext)
     */
    public void start(BundleContext context) throws Exception {
        super.start(context);
        try {
            loadProperties();
        } catch (LightUMLCoreException e) {
            // Exception while loading properties -> force an Initialize
            (new Initialize(System.currentTimeMillis(), true,
                    Initialize.CHECK_UMLGRAPH_NONE)).schedule();
        }
    }

    /**
     * <p>
     * The static accessor to get a handle to the singleton. Note: this plug-in
     * class should be created only by the platform.
     * </p>
     *
     * @return The LightUMLCorePlugin.
     */
    public static LightUMLCorePlugin getDefault() {
        if (plugin == null)
            plugin = new LightUMLCorePlugin();
        return plugin;
    }

    /**
     * <p>
     * Accessor for graphConverter
     * </p>
     *
     * @return IGraphConverter
     */
    public IGraphConverter getGraphConverter() {
        return graphConverter;
    }

    /**
     * <p>
     * Setter for graphConverter
     * </p>
     *
     * @param c
     *            The graphconverter.
     */
    public void setGraphConverter(IGraphConverter c) {
        graphConverter = c;
    }

    /**
     * <p>
     * Load properties for graph building.
     * </p>
     *
     * @throws LightUMLCoreException
     *
     */
    public synchronized void loadProperties() throws LightUMLCoreException {
        try {
            graphBuildProperties.load(new FileInputStream(
                    getPropertiesFilePath().toFile()));
        } catch (IOException e) {
            throw new LightUMLCoreException(e);
        }
    }

    /**
     * <p>
     * Getter for graphBuildProperties.
     * </p>
     *
     * @return graphBuildProperties
     * @throws LightUMLCoreException
     */
    public Properties getGraphBuildProperties() throws LightUMLCoreException {
        return graphBuildProperties;
    }

    /**
     * <p>
     * Getter for a graph build property.
     * </p>
     *
     * @param property
     *            Name of the property.
     * @return value of the property.
     *
     */
    public String getGraphBuildProperty(String property) {
        return graphBuildProperties.getProperty(property);
    }

    /**
```

```
     * <p>
     * Setter for a graph build property.
     * </p>
     *
     * @param key
     *            Key (name) of the property.
     * @param value
     *            The new value for the porperty.
     * @throws LightUMLCoreException
     */
    public void setGraphBuildProperty(String key, String value)
            throws LightUMLCoreException {
        graphBuildProperties.setProperty(key, value);
        try { // always store so that ant builds are up to date with
            // properties

            graphBuildProperties
                    .store(new FileOutputStream(getPropertiesFilePath()
                            .toFile()), "Graph␣Build␣Properties");
        } catch (IOException e) {
            throw new LightUMLCoreException(e);
        }
    }

    /**
     * <p>
     * Getter for the runner that is used to run targets in BUILD_FILE.
     * </p>
     *
     * @return A new Antrunner for BUILD_FILE.
     */
    public AntRunner getBuildFileRunner() {
        AntRunner runner = new AntRunner();
        runner.setBuildFileLocation(getBuildFilePath().toString());
        // devel time
        runner.addBuildLogger("org.apache.tools.ant.DefaultLogger");
        runner.setMessageOutputLevel(4);
        return runner;
    }

    // ---------- Methods below this line can be called from the UI ----------
    /**
     * <p>
     * This method is used to restore default settings. Called from the UI.
     * </p>
     * <p>
     * A JobChangeListener is needed if the UI wants to update its preferences
     * that depend on graphBuildProperties (that get reset with Initialize).
     * </p>
     *
     */
    public synchronized void restoreSettings(IJobChangeListener listener) {
        // 1 - init statelocation (& read default properties)
        Initialize initJob = new Initialize(System.currentTimeMillis(), true,
                Initialize.CHECK_UMLGRAPH_NONE);
        if (listener != null)
            initJob.addJobChangeListener(listener);
        initJob.schedule();
    }

    /**
     * <p>
     * Generate a class diagram for a given project. Called from the UI.
     * </p>
     * <ul>
     * <li> LightUMLSchedulingRule prevents LightUMLJobs from running
     * concurrently (they should be run sequentially). </li>
     * <li> All jobs are scheduled at once. This is mainly because otherwise
     * generating two class diagrams "at the same time" could confuse the
     * diagram files (in .dot and in graphics form). </li>
     * <li> Jobs support canceling => responsiveness. </li>
     * <li>Note that Eclipse Java model is used only at UI-level.</li>
     * </ul>
     *
     * @param project
     *            The project to generate a class diagram for.
     * @param packagePath
     *            A project relative path under which the source files will be
     *            looked for.
     * @param packageName
     *            Name of the package or null if generating a diagram for a
     *            whole project.
     *
     */
    public synchronized void generateClassDiagram(IProject project,
            IPath packagePath, String packageName) {
        final long refTime = System.currentTimeMillis();
        IPath sourcePath = project.getLocation();
        if (packagePath != null)
            sourcePath = sourcePath.append(packagePath);

        // 1 - initialize if needed
        (new Initialize(refTime, false, Initialize.CHECK_UMLGRAPH_JAR))
                .schedule();

        // 2 - run the UmlGraph doclet to convert .java files to a single .dot
        // file
        (new JavaToDot(refTime, sourcePath, packageName)).schedule();

        // 3 - convert .dot file to a graphics file
        (new DotToGraphics(refTime)).schedule();

        // 4 - add graphics file to the project
        (new AddGraphToProject(refTime, project, packageName)).schedule();
    }

    /**
     * <p>
     * Generate a sequence diagram from a .pic file.
     * </p>
     *
     * @param picFile
     *            The .pic file to generate from.
     * @hidden
     */
    public synchronized void generateSequenceDiagram(IFile picFile) {
        final long refTime = System.currentTimeMillis();

        // 1 - initialize if needed
        (new Initialize(refTime, false, Initialize.CHECK_UMLGRAPH_PIC_MACROS))
                .schedule();

        // 2 - convert .pic to graphics
        (new PicToGraphics(refTime, picFile.getLocation())).schedule();

        // 3 - add graphics file to the project
        (new AddGraphToProject(refTime, picFile.getProject(), null)).schedule();
    }
}
```

A.2.10 **LocalGraphConverter**

```
package org.lightuml.core;

import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

import org.eclipse.ant.core.AntRunner;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.IProgressMonitor;

/**
 * <p>
 * Implements IGraphConverter using locally installed GraphViz tools (dot).
 * </p>
 *
 * @author Antti Hakala
 */
public class LocalGraphConverter implements IGraphConverter, IBuildConstants,
        IGraphBuildProperties {
    /**
     * <p>
     * Static instance of LocalGraphConverter.
     * </p>
     */
    private static LocalGraphConverter instance = null;

    /**
     * <p>
     * Protected constructor (singleton pattern).
     * </p>
     */
    private LocalGraphConverter() {
    };

    /**
     * <p>
     * Static accessor.
     * </p>
     *
     * @return LocalGraphConverter
     */
    static public LocalGraphConverter instance() {
        if (instance == null)
            instance = new LocalGraphConverter();
        return instance;
    }

    /**
     * <p>
     * Convert .dot file to graphics format.
     * </p>
     *
     * @param dotFilePath
     *            UNUSED atm.
     * @param pm
     *            The progress monitor to be used.
     * @throws LightUMLCoreException
     * @see IGraphConverter#dotToGraphics(IProgressMonitor)
     */
    public void dotToGraphics(IPath dotFilePath, IProgressMonitor pm)
            throws LightUMLCoreException {
        AntRunner runner = LightUMLCorePlugin.getDefault().getBuildFileRunner();
        runner.setExecutionTargets(new String[] { "dot-to-graphics" });
        try {
            runner.run(pm);
        } catch (CoreException e) {
            throw new LightUMLCoreException(e);
        }
    }

    /**
     * <p>
     * Path to the graphics file to be generated.
     * </p>
     *
     * @see org.lightuml.core.IGraphConverter#getGraphicsFilePath()
     * @return The path to the graphics file (to be) generated.
     */
    public IPath getGraphicsFilePath() throws LightUMLCoreException {
        LightUMLCorePlugin cp = LightUMLCorePlugin.getDefault();
        return cp.getStateLocation().append(STATELOC_OUTPUT_DIR).append(
                INTERNAL_GRAPH_FILE_NAME).addFileExtension(
                cp.getGraphBuildProperty(P_GRAPHICS_FORMAT));
    }

    /**
     * <p>
     * Delivers the path to the .pic file to Ant.
     * </p>
     *
     * @throws IOException
     * @hidden
     */
    private void deliverPicFilePath(IPath picFilePath) throws IOException {
        Properties prop = new Properties();

        prop.setProperty("pic-file-path", picFilePath.toString());
        IPath p = LightUMLCorePlugin.getDefault().getStateLocation().append(
                STATELOC_RUNSETTINGS);
        prop.store(new FileOutputStream(p.toFile()), "Ant␣run␣settings");
    }

    /**
     * <p>
     * Convert .pic to graphics.
     * </p>
     * @throws IOException
     * @hidden
     */
    public void picToGraphics(IPath picFilePath, IProgressMonitor pm)
            throws LightUMLCoreException {

        try {
            deliverPicFilePath(picFilePath);
            AntRunner runner = LightUMLCorePlugin.getDefault().getBuildFileRunner();
            runner.setExecutionTargets(new String[] { "pic-to-graphics" });
            runner.run(pm);
        } catch (IOException e) {
            throw new LightUMLCoreException(e);
        } catch (CoreException e) {
            throw new LightUMLCoreException(e);
        }
    }
}
```

A.2.11 AddGraphToProject

```
package org.lightuml.core.jobs;

import java.io.FileInputStream;
import java.io.FileNotFoundException;

import org.eclipse.core.resources.IFile;
import org.eclipse.core.resources.IFolder;
import org.eclipse.core.resources.IProject;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Path;
import org.eclipse.core.runtime.Status;
import org.lightuml.core.IGraphBuildProperties;
import org.lightuml.core.LightUMLCoreException;

/**
 * <p>
 * This job adds the generated graphics file to project.
 * </p>
 *
 * @author Antti Hakala
 */
public class AddGraphToProject extends LightUMLJob implements
        IGraphBuildProperties {
    /**
     * <p>
     * Project to add the graph to.
     * </p>
     */
    private IProject project;

    /**
     * <p>
     * Package name, or null if no package selected
     * </p>
     */
    private String packageName;

    /**
     * <p>
     * Constructor.
     * </p>
     *
     * @param p
     *            Project to add the graph to.
     * @param t
     *            Reference time. Graph to add has to be newer than this.
     * @param pn
     *            Package name, or null if no package selected.
     */
    public AddGraphToProject(long t, IProject p, String pn) {
        // Gets a lock to project p
        // A lock to the whole project is ok, since we might need to create a
        // folder.
        super(t, "Adding␣a␣graph␣to␣project␣" + p.getName(), p);
        project = p;
        packageName = pn;
    }

    /**
     * <p>
     * Internal method for getting the name of the graph file.
     * <p>
     *
     * @return name of the graph file in the project
     * @throws LightUMLCoreException
     */
    private IPath getGraphFileName() throws LightUMLCoreException {
        IPath graphFileName;

        // check if the graph should be named like the package
        if ((packageName != null)
                && corePlugin.getGraphBuildProperty(P_USE_PACKAGE_NAME).equals(
                        "true"))
            graphFileName = new Path(packageName);
        else
            graphFileName = new Path(corePlugin
                    .getGraphBuildProperty(P_GRAPH_FILE_NAME));
        // add extension
        graphFileName = graphFileName.addFileExtension(corePlugin
                .getGraphBuildProperty(P_GRAPHICS_FORMAT));

        return graphFileName;
    }

    /**
     * <p>
     * Internal method for getting the graph file.
     * If a folder is created, it is marked as derived.
     * </p>
     *
     * @return the graph file in the project that represents the generated
     *         diagram
     * @throws CoreException
     * @throws LightUMLCoreException
     */
    private IFile getGraphFile() throws CoreException, LightUMLCoreException {
        IPath graphFileName = getGraphFileName(), projPath = project
                .getFullPath(), outDirPath = new Path(corePlugin
                .getGraphBuildProperty(P_PROJECT_OUTPUT_DIR));
        IFile f;

        // check where to add the graph
        if (projPath.append(outDirPath).equals(projPath))
            f = project.getFile(graphFileName);
        else {
            IFolder folder = project.getFolder(outDirPath);
            if (!folder.exists()) {
                folder.create(true, true, null);
                folder.setDerived(true);
            }
            f = folder.getFile(graphFileName);
        }
        return f;
    }

    /**
     * <p>
     * Adds the generated graph file to the project, marking it a derived
     * resource.
     * </p>
     *
     * @see org.eclipse.core.runtime.jobs.Job#run(org.eclipse.core.runtime.IProgressMonitor
     *     )
     * @param monitor
     *            The progress monitor to be used.
     * @return Status with OK severity if successful, otherwise ERROR severity.
     */
    protected IStatus run(IProgressMonitor monitor) {
        try {
            IFile f = getGraphFile();
            FileInputStream fis = new FileInputStream(corePlugin
                    .getGraphConverter().getGraphicsFilePath().toFile());

            if (f.exists())
                f.setContents(fis, true, false, monitor);
            else
                f.create(fis, true, monitor);

            f.setDerived(true);
            monitor.done();

        } catch (LightUMLCoreException e) {
            return errorStatus(e);
        } catch (CoreException e) {
            return errorStatus(e);
        } catch (FileNotFoundException e) {
            return errorStatus(e);
        }
        if(monitor.isCanceled())
            return cancelStatus();
        return Status.OK_STATUS;
    }
}
```

A.2.12 DotToGraphics

```
package org.lightuml.core.jobs;

import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
```

```java
import org.lightuml.core.IErrorMessages;
import org.lightuml.core.LightUMLCoreException;

/**
 * <p>
 * Converts .dot file to graphical form. Delegates the actual conversion to
 * graphConverter of LightUMLCorePlugin.
 * </p>
 *
 * @author Antti Hakala
 */
public class DotToGraphics extends LightUMLJob implements IErrorMessages {
    /**
     * Constructor
     *
     * @param t
     *            Reference time. The .dot file used has to be newer than this.
     */
    public DotToGraphics(long t) {
        super(t, "Converting to graphics");
    }

    /**
     * @see org.eclipse.core.runtime.jobs.Job#run(org.eclipse.core.runtime.IProgressMonitor
     *      )
     */
    protected IStatus run(IProgressMonitor monitor) {
        try {
            corePlugin.getGraphConverter().dotToGraphics(corePlugin.getDotFilePath(),
                  monitor);
            monitor.done();
        } catch (LightUMLCoreException e) {
            return errorStatus(e);
        }
        if(monitor.isCanceled())
            return cancelStatus();
        return Status.OK_STATUS;
    }
}
```

A.2.13 Initialize

```java
package org.lightuml.core.jobs;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;

import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.lightuml.core.IBuildConstants;
import org.lightuml.core.IErrorMessages;
import org.lightuml.core.IGraphBuildProperties;
import org.lightuml.core.LightUMLCoreException;

/**
 *
 * <p>
 * Initializes the plug-in statelocation by unzipping the statelocation.zip into
 * the plug-in state location. This is done only if forced or needed.
 * </p>
 *
 * @author Antti Hakala
 */
public class Initialize extends LightUMLJob implements IBuildConstants,
        IGraphBuildProperties, IErrorMessages {
    final static int BUFFER_SIZE = 2048;

    private boolean forceInitialize;

    private int checkUMLGraph;

    public final static int CHECK_UMLGRAPH_NONE = 0, CHECK_UMLGRAPH_JAR = 1,
            CHECK_UMLGRAPH_PIC_MACROS = 2;

    /**
     * <p>
     * Constructor.
     * </p>
     *
     * @param time
     *            The reference time.
     * @param forceInitializeFlag
     *            Force initialization?
     * @param checkUMLGraphFlag
     *            Check if UMLGraph is found?
     */
    public Initialize(long time, boolean forceInitializeFlag,
            int checkUMLGraphFlag) {
        super(time, "Initializing␣LightUML");
        forceInitialize = forceInitializeFlag;
        checkUMLGraph = checkUMLGraphFlag;
    }

    /**
     * <p>
     * Used for non-workspace copying.
     * </p>
     *
     * @param is
     *            inputstream to copy from
     * @param p
     *            path to the destination
     * @throws IOException
     */
    public static void nonWSCopy(InputStream is, IPath p) throws IOException {
        byte data[] = new byte[1024];
        File f = p.toFile();
        f.createNewFile();
        FileOutputStream fos = new FileOutputStream(f);
        int count;
        while ((count = is.read(data)) != -1)
            fos.write(data, 0, count);
        is.close();
        fos.close();
    }

    /**
     * <p>
     * Initializes statelocation. Overwrites properties / preferences for graph
     * generation.
     * </p>
     * <ul>
     * <li> Create statelocation output directory where generated graphs will be
     * stored. </li>
     * <li>copy "build.xml" to statelocation</li>
     * <li>copy "graph.ini" to statelocation</li>
     * </ul>
     * <p>
     *  TODO: refactor: copy everything in the dir to statelocation?
     * </p>
     * @throws IOException
     */
    private void initializeStateLocation() throws IOException {
        corePlugin.getStateLocation().append(STATELOC_OUTPUT_DIR).toFile()
                .mkdir();
        nonWSCopy(corePlugin.openStream(TO_STATELOCATION_DIR
                .append(BUILD_FILE_NAME)), corePlugin.getStateLocation()
                .append(BUILD_FILE_NAME));
        nonWSCopy(corePlugin.openStream(TO_STATELOCATION_DIR
                .append(GRAPH_BUILD_PROPERTIES_FILE_NAME)), corePlugin
                .getStateLocation().append(GRAPH_BUILD_PROPERTIES_FILE_NAME));
    }

    /**
     * <p>
     * Check if initialize is needed. Checks if the necessary files are found
     * and if the last initialize was with the same version of plug-in, i.e.
     * files are up to date.
     * </p>
     *
     * @return true if initialize is needed, false otherwise.
     * @throws LightUMLCoreException
     */
    private boolean needInitialize() throws LightUMLCoreException {
        String initVersion;
        if (!corePlugin.getBuildFilePath().toFile().exists()
                || !corePlugin.getPropertiesFilePath().toFile().exists()
                || ((initVersion = corePlugin
                        .getGraphBuildProperty("bundle-version")) == null)
                || !initVersion.equals(corePlugin.getBundle().getHeaders().get(
                        "Bundle-Version")))
            return true;
        return false;
    }

    /**
     * <p>
     * Check if UMLGraph is found.
     * </p>
     *
     * @throws LightUMLCoreException
     *
     */
    private void checkUMLGraph() throws LightUMLCoreException {
        if ((checkUMLGraph & CHECK_UMLGRAPH_JAR) != 0) {
            String pathToDoclet = corePlugin
                    .getGraphBuildProperty(P_UMLGRAPH_JAR_PATH);
            if ((pathToDoclet == null) || !new File(pathToDoclet).exists())
                throw new LightUMLCoreException(ERRMSG_NO_UMLGRAPH_JAR);
        }
        if ((checkUMLGraph & CHECK_UMLGRAPH_PIC_MACROS) != 0) {
            String pathToMacros = corePlugin
                    .getGraphBuildProperty(P_PIC_MACROS_PATH);
            if ((pathToMacros == null) || !new File(pathToMacros).exists())
                throw new LightUMLCoreException(ERRMSG_NO_PIC_MACROS);
        }
    }

    /**
     * @see org.eclipse.core.runtime.jobs.Job#run(org.eclipse.core.runtime.IProgressMonitor
     *      )
     */
    protected IStatus run(IProgressMonitor monitor) {
        try {
            if (forceInitialize || needInitialize()) {
                // Unzip the files in the state location.
                initializeStateLocation();
                // Load default properties.
                corePlugin.loadProperties();
                // Store the bundle version to properties.
                corePlugin.setGraphBuildProperty("bundle-version", corePlugin
                        .getBundle().getHeaders().get("Bundle-Version")
                        .toString());
            }

            checkUMLGraph();

        } catch (LightUMLCoreException e) {
                return errorStatus(e);
            } catch (IOException e) {
                return errorStatus(e);
            }
            if (monitor.isCanceled())
                return cancelStatus();
            return Status.OK_STATUS;
        }
}
```

A.2.14 `JavaToDot`

```
package org.lightuml.core.jobs;

import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Properties;

import org.eclipse.ant.core.AntRunner;
import org.eclipse.core.runtime.CoreException;
import org.eclipse.core.runtime.IPath;
import org.eclipse.core.runtime.IProgressMonitor;
import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.lightuml.core.IBuildConstants;
import org.lightuml.core.IGraphBuildProperties;
import org.lightuml.core.LightUMLCoreException;

/**
 * <p>
 * Converts java source files to a single .dot format file.
 * </p>
 *
 * @author Antti Hakala
 */
public class JavaToDot extends LightUMLJob implements IBuildConstants,
        IGraphBuildProperties {
    /**
     * <p>
     * Path under which to find source files.
     * </p>
     */
    private IPath sourcePath;

    /**
     * <p>
     * Package name.
     * </p>
     */
    private String packageName;

    /**
     * Constructor.
     *
     * @param time
     *            The reference time.
     * @param sp
     *            Path under which to find source files.
     * @param pn
     *            Package name or null.
     */
    public JavaToDot(long time, IPath sp, String pn) {
        super(time, "Running␣UmlGraph");
        sourcePath = sp;
        packageName = pn;
    }

    /**
     * The source path and the scope of the graph generation is delivered via a
     * property file. Argument -D<path> tends to get broken.
     *
     * @throws IOException
     * @throws LightUMLCoreException
     */
    private void deliverAntRunSettings() throws IOException,
            LightUMLCoreException {
        Properties prop = new Properties();
        String scope;

        // define the scope of the graph generation
        if ((packageName == null)
                || corePlugin.getGraphBuildProperty(P_RECURSE_PACKAGES).equals(
                        "true"))
            scope = "**/*.java";
        else
            scope = "*.java";

        prop.setProperty("scope", scope);
        prop.setProperty("source-path", sourcePath.toString());
        IPath p = corePlugin.getStateLocation().append(STATELOC_RUNSETTINGS);
        prop.store(new FileOutputStream(p.toFile()),
                "Ant␣run␣settings␣for␣java-to-dot");
    }

    /**
     * @see org.eclipse.core.runtime.jobs.Job#run(org.eclipse.core.runtime.IProgressMonitor
     *      )
     */
    protected IStatus run(IProgressMonitor pm) {
        try {
            deliverAntRunSettings();
            AntRunner runner = corePlugin.getBuildFileRunner();
            runner.setExecutionTargets(new String[] { "java-to-dot" });
            runner.run(pm);
            pm.done();

        } catch (CoreException e) {
            return errorStatus(e);
        } catch (IOException e) {
            return errorStatus(e);
        } catch (LightUMLCoreException e) {
            return errorStatus(e);
        }
        if(pm.isCanceled())
            return cancelStatus();
        return Status.OK_STATUS;
    }
}
```

A.2.15 `LightUMLJob`

```
package org.lightuml.core.jobs;

import java.util.Collection;
import java.util.Vector;

import org.eclipse.core.runtime.IStatus;
import org.eclipse.core.runtime.Status;
import org.eclipse.core.runtime.jobs.ISchedulingRule;
import org.eclipse.core.runtime.jobs.Job;
import org.eclipse.core.runtime.jobs.MultiRule;
import org.lightuml.core.LightUMLCorePlugin;

/**
 * <p>
 * The abstract parent class of LightUML jobs.
 * </p>
 *
 * @author Antti Hakala
 * @navassoc - uses - org.lightuml.core.jobs.LightUMLSchedulingRule
 */
public abstract class LightUMLJob extends Job {
    /**
     * <p>
     * Reference time (time this toolchain was created).
     * </p>
     */
    private long refTime;

    /**
     * <p>
     * String representing the id of this family of jobs.
     * </p>
     */
    private String familyId;

    /**
     * <p>
     * static instance of LightUMLCorePlugin (for convenience)
```

```
	 * </p>
	 */
	protected static LightUMLCorePlugin corePlugin = null;

	/**
	 * <p>
	 * A register for error and cancel statuses that have been returned
	 * by LightUMLJobs. Used to see if a job in a specific toolchain should
	 * run, or if an earlier job in the chain has already been canceled or
	 * has returned an error status.
	 * </p>
	 */
	private static Collection errorAndCancelStatusRegister = null;

	/**
	 * <p>
	 * Internal init method.
	 * </p>
	 *
	 * @param time
	 *            reference time of this job
	 * @param rule
	 *            Scheduling rule for this job.
	 */
	private void init(long time, ISchedulingRule rule) {
		refTime = time;
		familyId = new StringBuffer("org.lightuml.core").append(refTime).toString();
		setRule(rule);
		if (corePlugin == null)
			corePlugin = LightUMLCorePlugin.getDefault();
		if (errorAndCancelStatusRegister == null)
			errorAndCancelStatusRegister = java.util.Collections
					.synchronizedCollection(new Vector());
	}

	/**
	 * <p>
	 * Return an error status for this job. Only one error (first one)
	 * should occur from a toolchain.
	 * </p>
	 * <p>
	 * Uses reference times of jobs as toolchain identifiers. If jobs have the
	 * same reference time, they belong to the same toolchain. Cancels the other
	 * jobs that belong to same toolchain.
	 * </p>
	 *
	 * @param e
	 *            Exception that caused the error.
	 * @return IStatus that can be returned in the run() of LightUMLJob (if an
	 *         exception occured).
	 */
	protected IStatus errorStatus(Exception e) {
		errorAndCancelStatusRegister.add(familyId);
```

```
        // return the error status
        return new Status(Status.ERROR, "org.lightuml.core", Status.OK, e
                .getMessage(), e);
    }
    /**
     * <p>
     * Return a cancel status and add familyId to register. See above.
     * </p>
     * @return cancel status
     */
    protected IStatus cancelStatus() {
        errorAndCancelStatusRegister.add(familyId);
        return Status.CANCEL_STATUS;
    }
    /**
     * <p>
     * Run only if family not canceled or erred.
     * </p>
     */
    public boolean shouldRun() {
        if(errorAndCancelStatusRegister.contains(familyId))
            return false;
        return true;
    }

    /**
     * <p>
     * constructor
     * </p>
     *
     * @param str
     *            Name passed to parent class Job.
     */
    public LightUMLJob(long time, String str) {
        super(str);
        init(time, LightUMLSchedulingRule.getDefault());
    }

    /**
     * <p>
     * Alternative constructor with an additional scheduling rule.
     * </p>
     *
     * @param time
     *            The reference time.
     * @param str
     *            Name passed to parent class Job.
     * @param rule
     *            Additional scheduling rule for this job.
     */
    public LightUMLJob(long time, String str, ISchedulingRule rule) {
        super(str);
        init(time, MultiRule.combine(rule, LightUMLSchedulingRule.getDefault()));
    }

    /**
     * @see org.eclipse.core.runtime.jobs.Job#belongsTo(java.lang.Object)
     */
    public boolean belongsTo(Object family) {
        return (family.equals(familyId));
    }
}
```

A.2.16 `LightUMLSchedulingRule`

```
package org.lightuml.core.jobs;

import org.eclipse.core.runtime.jobs.ISchedulingRule;

/**
 * <p>
 * Scheduling rule of LightUMLJobs. Prevents concurrent execution of
 * LightUMLJobs.
 * </p>
 *
 * @author Antti Hakala
 */
public class LightUMLSchedulingRule implements ISchedulingRule {
    /**
     * <p>
     * Static instance of LightUMLSchedulingRule.
     * </p>
     */
    private static LightUMLSchedulingRule rule = null;

    /**
     * Constructor.
     */
    private LightUMLSchedulingRule() {
        rule = this;
    }

    /**
     * Static accessor (singleton pattern).
     *
     * @return LightUMLSchedulingRule
     */
    public static LightUMLSchedulingRule getDefault() {
        if (rule == null)
            rule = new LightUMLSchedulingRule();
        return rule;
    }

    /**
     * @see org.eclipse.core.runtime.jobs.ISchedulingRule#contains(org.eclipse.core.runtime
           .jobs.ISchedulingRule)
     */
    public boolean contains(ISchedulingRule rule) {
        return rule == this;
    }

    /**
     * @see org.eclipse.core.runtime.jobs.ISchedulingRule#isConflicting(org.eclipse.core.
         runtime.jobs.ISchedulingRule)
     */
    public boolean isConflicting(ISchedulingRule rule) {
        return rule == this;
    }

}
```

A.3 `org.lightuml.ui`

A.3.1 Plug-in Manifest (`plugin.xml`)

```
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin
   id="org.lightuml.ui"
   class="org.lightuml.ui.LightUMLUIPlugin"
   name="LightUML UI"
   version="1.2.1"
   provider-name="Antti Hakala">

   <runtime>
      <library name="ui.jar">
         <export name="*"/>
      </library>
   </runtime>

   <requires>
      <import plugin="org.eclipse.ui"/>
      <import plugin="org.lightuml.core"/>
      <import plugin="org.eclipse.core.resources"/>
      <import plugin="org.eclipse.core.runtime"/>
      <import plugin="org.eclipse.jdt.core"/>
      <import plugin="org.eclipse.jface.text"/>
      <import plugin="org.eclipse.ui.workbench.texteditor"/>
      <import plugin="org.eclipse.ui.editors"/>
   </requires>

   <extension
         point="org.eclipse.ui.popupMenus">
      <objectContribution
            objectClass="org.eclipse.jdt.core.IJavaProject"
            id="org.lightuml.ui.javaprojectcontribution">
            <menu label="LightUML" path="additions" id="org.lightuml.ui.popupmenu" />
            <action
```

```
                label="LightUML:␣Generate␣a␣Class␣Diagram"
                class="org.lightuml.ui.actions.GenerateClassDiagram"
                menubarPath="org.lightuml.ui.popupmenu"
                enablesFor="1"
                id="org.lightuml.ui.actions.GenerateClassDiagram">
            </action>
    </objectContribution>

    <objectContribution
            objectClass="org.eclipse.jdt.core.IPackageFragment"
            id="org.lightuml.ui.javapackagecontribution">
            <menu label="LightUML" path="additions" id="org.lightuml.ui.popupmenu" />
            <action
                label="LightUML:␣Generate␣a␣Class␣Diagram"
                class="org.lightuml.ui.actions.GenerateClassDiagram"
                menubarPath="org.lightuml.ui.popupmenu"
                enablesFor="1"
                id="org.lightuml.ui.actions.GenerateClassDiagram">
            </action>
    </objectContribution>

    <objectContribution
            objectClass="org.eclipse.core.resources.IFile"
            nameFilter="*.pic"
            id="org.lightuml.ui.picfilecontribution">
            <menu label="LightUML" path="additions" id="org.lightuml.ui.popupmenu" />
            <action
                label="LightUML:␣Generate␣a␣Sequence␣Diagram"
                class="org.lightuml.ui.actions.GenerateSequenceDiagram"
                menubarPath="org.lightuml.ui.popupmenu"
                enablesFor="1"
                id="org.lightuml.ui.actions.GenerateSequenceDiagram">
            </action>
    </objectContribution>
</extension>

<extension point="org.eclipse.ui.actionSets">
      <actionSet id="org.lightuml.ui.actionSet" label="LightUML" visible="true">
            <menu id="org.lightuml.ui.menu" label="LightUML" path="additions">
                 <separator name="group"/>
            </menu>
            <action
                label="Restore␣Default␣Settings"
                class="org.lightuml.ui.actions.RestoreSettings"
                menubarPath="org.lightuml.ui.menu/group"
                id="org.lightuml.ui.actions.RestoreSettings">
            </action>
      </actionSet>
 </extension>

<extension point="org.eclipse.ui.preferencePages">
    <page
            class="org.lightuml.ui.preferences.LightUMLPage"
```

```
                name="LightUML"
                id="org.lightuml.ui.preferences.LightUMLPage"
                category="org.eclipse.jdt.ui.preferences.JavaBasePreferencePage" />
        <page
                class="org.lightuml.ui.preferences.UMLGraphPage"
                name="UMLGraph"
                id="org.lightuml.ui.preferences.UMLGraphPage"
                category="org.lightuml.ui.preferences.LightUMLPage" />
        <page
                class="org.lightuml.ui.preferences.DotAndPic2PlotPage"
                name="dot␣and␣pic2plot"
                id="org.lightuml.ui.preferences.DotAndPic2PlotPage"
                category="org.lightuml.ui.preferences.LightUMLPage" />
    </extension>

    <extension point="org.eclipse.help.toc">
        <toc file="help/toc.xml" primary="true" />
    </extension>

    <extension point="org.eclipse.ui.editors">
     <editor
        id="org.lightuml.ui.editor.piceditor"
        name="Pic␣Editor"
        icon="icons/pic.gif"
        extensions="pic"
        class="org.lightuml.ui.editor.PicEditor"
        default="true" />
    </extension>

</plugin>
```

A.3.2 LightUMLUIPlugin

```
package org.lightuml.ui;

import java.util.Iterator;
import java.util.Properties;

import org.eclipse.jface.preference.IPreferenceStore;
import org.eclipse.jface.util.IPropertyChangeListener;
import org.eclipse.jface.util.PropertyChangeEvent;
import org.eclipse.ui.plugin.AbstractUIPlugin;
import org.lightuml.core.LightUMLCoreException;
import org.lightuml.core.LightUMLCorePlugin;

/**
 * <p>
 * Plug-in class of org.lightuml.ui.
 * </p>
 *
 * @author Antti Hakala
 *
 * @navassoc - "adds to UI" - org.lightuml.ui.actions.GenerateClassDiagram
 * @navassoc - "adds to UI" - org.lightuml.ui.actions.RestoreSettings
 * @navassoc - "adds to UI" - org.lightuml.ui.preferences.LightUMLPage
 *
 */
public class LightUMLUIPlugin extends AbstractUIPlugin {

    private static LightUMLUIPlugin plugin;

    /**
     * <p>
     * A property change listener for the preference store of this plug-in.
     * Tells org.lightuml.core plug-in to update its properties if they're
     * changed in the UI (via preference page).
     * </p>
     *
     * @see org.eclipse.jface.util.IPropertyChangeListener#propertyChange(org.eclipse.jface
          .util.PropertyChangeEvent)
     * @hidden
     */
    private class PropertyListener implements IPropertyChangeListener {
        /**
         * @see IPropertyChangeListener#propertyChange(org.eclipse.jface.util.
              PropertyChangeEvent)
         */
        public void propertyChange(PropertyChangeEvent event) {
            LightUMLCorePlugin cp = LightUMLCorePlugin.getDefault();

            try {
                if (cp.getGraphBuildProperties().getProperty(
                        event.getProperty()) != null)
                    cp.setGraphBuildProperty(event.getProperty(),
                            (String) event.getNewValue().toString());
            } catch (LightUMLCoreException e) {
                e.printStackTrace();
            }
        }
    }

    /**
     * <p>
     * Overridden from AbstractUIPlugin.
     * </p>
     */
    protected void initializeDefaultPluginPreferences() {
        initPreferences();
        getPreferenceStore().addPropertyChangeListener(new PropertyListener());
    }

    /**
     * <p>
     * Initialize preferences. Adds graph build properties of LightUMLCorePlugin
     * into preference store of this plug-in.
     * </p>
     */
    public void initPreferences() {
        IPreferenceStore store = getPreferenceStore();
        Properties p;
        try {
            p = LightUMLCorePlugin.getDefault().getGraphBuildProperties();
            Iterator i = p.keySet().iterator();
            String propName;
            while (i.hasNext()) {
                propName = (String) i.next();
                store.setDefault(propName, p.getProperty(propName));
                store.setValue(propName, p.getProperty(propName));
            }
        } catch (LightUMLCoreException e) {
            e.printStackTrace();
        }
    }

    /**
     * Constructor.
     */
    public LightUMLUIPlugin() {
        super();
        plugin = this;
    }

    /**
     * <p>
     * static accessor.
     * </p>
     *
     * @return LightUMLUIPlugin
     */
    public static LightUMLUIPlugin getDefault() {
        if (plugin == null)
            plugin = new LightUMLUIPlugin();
        return plugin;
    }
}
```

A.3.3 GenerateClassDiagram

```
package org.lightuml.ui.actions;

import org.eclipse.core.resources.IProject;
import org.eclipse.core.resources.IResource;
import org.eclipse.core.runtime.IPath;
import org.eclipse.jdt.core.IJavaProject;
import org.eclipse.jdt.core.IPackageFragment;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.jface.viewers.StructuredSelection;
import org.eclipse.ui.IObjectActionDelegate;
import org.eclipse.ui.IWorkbenchPart;
import org.lightuml.core.LightUMLCorePlugin;

/**
 *
 * <p>
 * The popup menu action that is used to trigger generation of a class diagram.
 * </p>
 *
 * @author Antti Hakala
 *
 */
public class GenerateClassDiagram implements IObjectActionDelegate {
    /**
     * <p>
     * The project to generate a class diagram for.
     * </p>
     */
    private IProject project = null;

    /**
     * <p>
     * The Java package path (if a package is selected).
     * </p>
     */
    private IPath packagePath = null;

    /**
     * <p>
     * Name of the Java package (if a package is selected).
     * </p>
     */
    private String packageName = null;

    /**
     * <p>
     * Constructor for GenerateClassDiagram.
     * </p>
     */
    public GenerateClassDiagram() {
        super();
    }

    /**
     * @see IObjectActionDelegate#setActivePart(org.eclipse.jface.action.IAction,
     *      org.eclipse.ui.IWorkbenchPart)
     */
    public void setActivePart(IAction action, IWorkbenchPart targetPart) {
    }

    /**
     * @see IActionDelegate#run(org.eclipse.jface.action.IAction)
     */
```

```java
    public void run(IAction action) {
        LightUMLCorePlugin.getDefault().generateClassDiagram(project, packagePath,
                packageName);
    }

    /**
     * @see IActionDelegate#selectionChanged(org.eclipse.jface.action.IAction,
     *      org.eclipse.jface.viewers.ISelection)
     */
    public void selectionChanged(IAction action, ISelection selection) {

        Object target = ((StructuredSelection) selection).getFirstElement();

        // target selection is a java project
        if (target instanceof IJavaProject) {
            project = ((IJavaProject) target).getProject();
            packagePath = null;
            packageName = null;
        }
        // target selection is a java package
        else if (target instanceof IPackageFragment) {
            IPackageFragment pf = (IPackageFragment) target;
            IResource resource = pf.getResource();

            project = resource.getProject();
            packagePath = resource.getProjectRelativePath();
            packageName = pf.getElementName();
        }
    }
}
```

A.3.4 RestoreSettings

```java
package org.lightuml.ui.actions;

import org.eclipse.core.runtime.jobs.IJobChangeEvent;
import org.eclipse.core.runtime.jobs.JobChangeAdapter;
import org.eclipse.jface.action.IAction;
import org.eclipse.jface.viewers.ISelection;
import org.eclipse.ui.IWorkbenchWindow;
import org.eclipse.ui.IWorkbenchWindowActionDelegate;
import org.lightuml.core.LightUMLCorePlugin;
import org.lightuml.ui.LightUMLUIPlugin;

/**
 * <p>
 * Action for restoring default settings.
 * </p>
 *
 * @author Antti Hakala
 *
 */
public class RestoreSettings implements IWorkbenchWindowActionDelegate {

    /**
     * <p>
     * Runs the restoreSettings()-method in LightUMLCorePlugin.
     * </p>
     * <p>
     * JobChangeAdapter is used to tell LightUMLUIPlugin to init its preferences
     * when LightUMLCorePlugin is done resetting default properties for graph
     * building. Thus, UI preferences are kept in sync with core properties.
     * JobChangeAdapter is an adapter for IJobChangeListener interface.
     * </p>
     *
     * @see org.eclipse.ui.IActionDelegate#run(org.eclipse.jface.action.IAction)
     *
     */
    public void run(IAction action) {
        LightUMLCorePlugin.getDefault().restoreSettings(new JobChangeAdapter() {
            public void done(IJobChangeEvent event) {
                LightUMLUIPlugin.getDefault().initPreferences();
            }
        });
    }

    /*********************************************************************
     * @see org.eclipse.ui.IWorkbenchWindowActionDelegate#dispose()
     */
    public void dispose() {
    }

    /**
     * @see org.eclipse.ui.IWorkbenchWindowActionDelegate#init(org.eclipse.ui.
         IWorkbenchWindow)
     */
    public void init(IWorkbenchWindow window) {
    }

    /**
     * @see org.eclipse.ui.IActionDelegate#selectionChanged(org.eclipse.jface.action.
         IAction,
     *      org.eclipse.jface.viewers.ISelection)
     */
    public void selectionChanged(IAction action, ISelection selection) {
    }
}
```

A.3.5 DotAndPic2PlotPage

```
package org.lightuml.ui.preferences;

import org.eclipse.jface.preference.DirectoryFieldEditor;
import org.eclipse.jface.preference.FieldEditorPreferencePage;
import org.eclipse.jface.preference.RadioGroupFieldEditor;
import org.eclipse.jface.preference.StringFieldEditor;
```

```java
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;
import org.lightuml.core.IGraphBuildProperties;
import org.lightuml.ui.LightUMLUIPlugin;

/**
 * <p>
 * Preference subpage for dot executable.
 * </p>
 *
 * @author Antti Hakala
 */
public class DotAndPic2PlotPage extends FieldEditorPreferencePage implements
        IWorkbenchPreferencePage, IGraphBuildProperties {
    /**
     * <p>
     * Constructor.
     * </p>
     */
    public DotAndPic2PlotPage() {
        super(GRID);
        setPreferenceStore(LightUMLUIPlugin.getDefault().getPreferenceStore());
        setDescription("Preferences for dot and pic2plot:");
    }

    /**
     * <p>
     * Creates the field editors.
     * </p>
     */
    public void createFieldEditors() {
        addField(new StringFieldEditor(P_DOT_EXTRA_PARAM,
                "dot extra commandline parameters:", getFieldEditorParent()));
        addField(new StringFieldEditor(P_PIC2PLOT_EXTRA_PARAM,
                "pic2plot extra commandline parameters:", getFieldEditorParent()));

        addField(new DirectoryFieldEditor(P_EXTRA_LOOKUP_PATH, "Extra lookup path(s):",
                getFieldEditorParent()));
        addField(new RadioGroupFieldEditor(P_GRAPHICS_FORMAT,
                "Graphics format:", 1, OUTPUT_FORMATS, getFieldEditorParent()));
    }

    /**
     * @see IWorkbenchPreferencePage#init(org.eclipse.ui.IWorkbench)
     */
    public void init(IWorkbench workbench) {
    }
}
```

A.3.6 LightUMLPage

```java
package org.lightuml.ui.preferences;

```

```
import org.eclipse.jface.preference.BooleanFieldEditor;
import org.eclipse.jface.preference.FieldEditorPreferencePage;
import org.eclipse.jface.preference.StringFieldEditor;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;
import org.lightuml.core.IGraphBuildProperties;
import org.lightuml.ui.LightUMLUIPlugin;

/**
 * <p>
 * Note: most comments from Eclipse template for preference page extension
 * point.
 * </p>
 * <p>
 * This class represents a preference page that is contributed to the
 * Preferences dialog. By subclassing <samp>FieldEditorPreferencePage</samp>,
 * we can use the field support built into JFace that allows us to create a page
 * that is small and knows how to save, restore and apply itself.
 * </p>
 * <p>
 * This page is used to modify preferences only. They are stored in the
 * preference store that belongs to the main plug-in class. That way,
 * preferences can be accessed directly via the preference store.
 * </p>
 * <p>
 * General preferences for LightUML.
 * </p>
 *
 * @has - - - org.lightuml.ui.preferences.DotAndPic2PlotPage
 * @has - - - org.lightuml.ui.preferences.UMLGraphPage
 * @author Antti Hakala
 */
public class LightUMLPage extends FieldEditorPreferencePage implements
        IWorkbenchPreferencePage, IGraphBuildProperties {
    /**
     * <p>
     * Constructor. Sets the preference store to preference store of
     * LightUMLUIPlugin.
     * </p>
     *
     */
    public LightUMLPage() {
        super(GRID);
        setPreferenceStore(LightUMLUIPlugin.getDefault().getPreferenceStore());
        setDescription("General preferences for LightUML:");
    }

    /**
     * Creates the field editors. Field editors are abstractions of the common
     * GUI blocks needed to manipulate various types of preferences. Each field
     * editor knows how to save and restore itself.
     */

    public void createFieldEditors() {
        addField(new StringFieldEditor(P_GRAPH_FILE_NAME, "Graph file name:",
                getFieldEditorParent()));
        addField(new StringFieldEditor(P_PROJECT_OUTPUT_DIR,
                "Output directory (relative to project root):",
                getFieldEditorParent()));
        addField(new BooleanFieldEditor(P_USE_PACKAGE_NAME,
                "Use package name as graph file name", getFieldEditorParent()));
        addField(new BooleanFieldEditor(P_RECURSE_PACKAGES,
                "Recurse into subpackages", getFieldEditorParent()));
    }

    /**
     * @see IWorkbenchPreferencePage#init(org.eclipse.ui.IWorkbench)
     */
    public void init(IWorkbench workbench) {
    }
}
```

A.3.7 UMLGraphPage

```
package org.lightuml.ui.preferences;

import org.eclipse.jface.preference.FieldEditorPreferencePage;
import org.eclipse.jface.preference.FileFieldEditor;
import org.eclipse.jface.preference.RadioGroupFieldEditor;
import org.eclipse.jface.preference.StringFieldEditor;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;
import org.lightuml.core.IGraphBuildProperties;
import org.lightuml.ui.LightUMLUIPlugin;

/**
 * <p>
 * Preference subpage for UMLGraph doclet.
 * </p>
 *
 * @author Antti Hakala
 */
public class UMLGraphPage extends FieldEditorPreferencePage implements
        IWorkbenchPreferencePage, IGraphBuildProperties {
    /**
     * <p>
     * Constructor.
     * </p>
     */
    public UMLGraphPage() {
        super(GRID);
        setPreferenceStore(LightUMLUIPlugin.getDefault().getPreferenceStore());
        setDescription("Preferences for UMLGraph doclet:");
    }

    /**
```

```
     * <p>
     * Creates the field editors.
     * </p>
     */
    public void createFieldEditors() {
        addField(new StringFieldEditor(P_UMLGRAPH_EXTRA_PARAM,
                "Extra␣commandline␣parameters:", getFieldEditorParent()));
        addField(new FileFieldEditor(P_UMLGRAPH_JAR_PATH, "UmlGraph.jar␣path:",
                true, getFieldEditorParent()));
        addField(new FileFieldEditor(P_PIC_MACROS_PATH, "sequence.pic␣path:",
                true, getFieldEditorParent()));
        addField(new RadioGroupFieldEditor(P_JAVADOC_ACCESS_LEVEL,
                "access␣level:", 4, ACCESS_LEVELS, getFieldEditorParent()));
    }

    /**
     * @see IWorkbenchPreferencePage#init(org.eclipse.ui.IWorkbench)
     */
    public void init(IWorkbench workbench) {
    }
}
```