Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences

Olli Kiljunen

Plug-in Interoperation in Extendable Platforms

Master's Thesis
Helsinki, April 13, 2021

Supervisor:	Professor Lauri Malmi
Advisor:	Juha Sorva D.Sc. (Tech.)

Aalto University
School of Science

Aalto University
School of Science
Master's Programme in Computer, Communication and Information Sciences — ABSTRACT OF MASTER'S THESIS

Author:	Olli Kiljunen		
Title: Plug-in Interoperation in Extendable Platforms			
Date:	April 13, 2021	**Pages:**	54
Major:	Computer Science	**Code:**	SCI3042
Supervisor:	Professor Lauri Malmi		
Advisor:	Juha Sorva D.Sc. (Tech.)		

In the recent decades, it has become customary that feature-rich software products are designed to be extendable with external components, *plug-ins*. While the architectural pattern enabling this design has become well established, a substantial limitation has been recognized: it only poorly supports interoperation between plug-ins, making plug-in dependencies difficult to maintain.

Practice has shown that in many systems designed in this way, plug-in interoperation is nevertheless needed. This raises the question: How could plug-in interoperation be designed to keep the problems that it causes in check?

This thesis tackles the question by analyzing the problems of plug-in interoperation. It turns out that dependencies between plug-ins are prone to incompatibility issues that are hard to predict and even harder to handle when they occur. Furthermore, due to the dynamic nature of the plug-in architecture, the integrity of a system with interdependent plug-ins is difficult to assure when releasing new versions.

To these design problems of plug-in interoperation, I search for potential solutions among the design patterns and practices of the software architecture literature. I consider the issue from the perspectives of both a plug-in and and the system it extends. Based on this analysis, I argue that by applying good software engineering practices, such as explicitly defined programming interfaces, controlled versioning, and sound code structure, software designers can remarkably reduce the complexity caused by plug-in interdependencies.

As a concrete example of a plug-in system, I study the IntelliJ IDE platform developed by JetBrains, and observe how plug-in interoperation materializes in it. As a case study, I present an IntelliJ plug-in that I co-developed as part of a team and review its design in the light of plug-in interoperability issues.

Keywords: software architecture, plug-in architecture, IntelliJ IDEA
Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tieto-, tietoliikenne- ja informaatiotekniikan maisteriohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Olli Kiljunen

Työn nimi:
Lisäosien välinen vuorovaikutus laajennettavissa ohjelmistoalustoissa

Päiväys:	13. huhtikuuta 2021	**Sivumäärä:**	54
Pääaine:	Tietotekniikka	**Koodi:**	SCI3042
Valvoja:	Professori Lauri Malmi		
Ohjaaja:	Tekniikan tohtori Juha Sorva		

Viime vuosikymmeninä on tullut tavanmukaiseksi, että laajatoiminnalliset ohjelmistotuotteet suunnitellaan laajennettaviksi ulkoisilla lisäosilla. Tähän tarkoitukseen usein käytetty arkkitehtoninen suunnittelumalli on vakiintuneisuudestaan huolimatta havaittu rajoittuneeksi: se ei kunnolla tue lisäosien välistä vuorovaikutusta tehden lisäosien välisten riippuvuuksien ylläpidosta hankalaa.

Kokemus on osoittanut, että monissa tällä tavoin suunnitelluissa ohjelmistoissa lisäosien välistä vuorovaikutusta kuitenkin tarvitaan. Tästä seuraa kysymys: Kuinka lisäosien välinen vuorovaikutus tulisi suunnitella, jotta siitä aiheutuvat ongelmat kyettäisiin hallitsemaan?

Tämä diplomityö lähestyy kysymystä analysoimalla lisäosien väliseen vuorovaikutukseen liittyvää ongelmakenttää. Osoittautuu, että lisäosien väliset riippuvuudet ovat alttiita yhteensopivuusongelmille, joita on hankala sekä ennustaa että jälkikäteen selvittää. Lisäosa-arkkitehtuurin dynaamisesta luonteesta johtuen lisäosien välisiä riippuvuuksia sisältävien järjestelmien eheyden varmistaminen on vaikeaa uusia versioita julkaistaessa.

Näihin lisäosien väliseen vuorovaikutukseen liittyviin ongelmakohtiin etsin mahdollisia ratkaisuja kirjallisuudessa tunnetuista suunnittelumalleista ja käytänteistä. Tarkastelen kysymystä niin lisäosan kuin alustaohjelmistonkin näkökulmasta. Analyysiini tukeutuen esitän, että ohjelmistosuunnittelijat voivat huomattavasti vähentää lisäosien välisistä riippuvuuksista aiheutuvaa mutkikkuutta soveltamalla ohjelmistokehityksen hyviä käytänteitä, kuten tarkasti määriteltyjä ohjelmointirajapintoja, hallittua versiointia ja hyvää lähdekoodin rakennetta.

Konkreettisena esimerkkinä laajennettavasta ohjelmistosta käsittelen JetBrainsin kehittämää IntelliJ-ohjelmointiympäristö-alustaa ja tutkin, miten lisäosien välinen vuorovaikutus siinä toteutuu. Tapaustutkimuksena esittelen ryhmämme kehittämän IntelliJ:n lisäosan ja arvioin sen ohjelmistoarkkitehtuuria lisäosien välisen vuorovaikutuksen kannalta.

Asiasanat: ohjelmistoarkkitehtuuri, lisäosa-arkkitehtuuri, IntelliJ IDEA

Kieli: Englanti

Contents

Chapter 1

Introduction

When designing feature-rich software products, it has become customary to avoid building them as closed systems with fixed sets of features. Rather they are built as platforms, on top of which their user communities and third-party vendors can produce their own extensions, plug-ins.

In his report about software architecture patterns, software architect Mark Richards discusses the plug-in architecture and states:

> Generally, plug-in modules should be independent of other plug-in modules, but you can certainly design plug-ins that require other plug-ins to be present. Either way, it is important to keep the communication between plug-ins to a minimum to avoid dependency issues. [25, ch. 3]

What is evident with large-scale, multi-vendor plug-in systems, Richards' principal recommendation of keeping plug-ins totally independent of each other, though reasonable, cannot be fulfilled in many practical cases: if one plug-in provides a significant amount of functionality required by another plug-in, re-implementing that functionality in the latter would not only be waste of resources but could also lead to an inconsistent configuration in a case where both of the plug-ins are present at the same time. Therefore, developers often have no other meaningful choice than make their plug-ins depend on other plug-ins of the same system.

Richards does not elaborate on what kind of dependency issues he thinks plug-in interoperation may cause but it is not difficult to see possible problems emerging when a dependee plug-in introduces a so called 'breaking change'. For a dependent plug-in, maintaining a dependency to an older version of another plug-in may become impossible if the platform does not allow more than one version of each plug-in to be installed at a time. As plug-ins can be developed by dozens of different vendors, coordinating common deploy schedule for all of them is not an option. It starts to seem that the software ecosystem, in which a plug-in developer operates, has quite a unique set of challenges. Developing high quality software in that kind of environment, undoubtedly, calls for ability to quickly adapt to changes while also sustaining high architectural standards and best software engineering practices.

1.1 Research question

In this thesis, I tackle the following research question:

> What issues should a software designer or architect take into account when designing software systems involving interoperating plug-ins?

I consider the question from the perspectives of both a platform system designer and a plug-in designer. As an answer to this question, I outline a toolkit of design principles and recommendations that a software designer can find useful in their brave quest to overcome the complications caused by plug-in dependencies.

1.2 Research methods

To find an answer to the research question, I analyze the problematic surrounding the plug-in interoperation detecting possible pitfalls that may occur in such a design. Then, I apply known patterns and engineering practices to model strategies that a software architect can use to overcome these problems.

In particular, I study a well-known plug-in platform system, IntelliJ IDEA and analyze the plug-in interoperation utilities it provides. To exemplify the subject of the study in practice, I present an IntelliJ plug-in developed by our team, and apply the found solutions to it.

1.3 Review of research approach

In her article *What makes good research in software engineering?*, Mary Shaw [28] outlines common methodology used in software engineering research. She classifies typical research methods and approaches of the field based on the type of their research questions, type of their research results, and how those results are validated. Sticking to Shaw's classification, the approach of this study can be characterized as follows:

- **Type of question**: *Method or means of development.*

  "*How can we do/create (or automate doing) X? What is a better way to do/create X?*" are examples of how this type of research questions are usually formulated, according to Shaw. The research question of this thesis, introduced above, fits well to this category, when X is set to 'plug-in interoperation'.

- **Type of result**: *Qualitative or descriptive model.*

  As examples of these types of research results, Shaw mentions "well-argued informal generalizations" and "guidance for integrating other results". These closely resemble the target of this study presented above.

- **Type of validation**: *Persuasion*, *experience*, and *example.*

  Persuasive result validation, Shaw notes, is often of a form "*I thought hard about this, and I believe...*". Even though this method is often non-ideal, Shaw distinguishes it from 'blatant assertion' which is always objectionable.

  Shaw differentiates between *experience* and *example*, considering them two separate validation methods. With the former, she refers to real-life implementations made by others than the author of a study, and with the latter to those made by the author. In this study, *experience* is gathered from IntelliJ plug-in system, and the plug-in developed by our team is presented as an *example.*

Based on this breakdown, the research approach of this study can be characterized more qualitative than quantitative, more inexact than exact, and based on more informal than formal models of its subject. As such, it does not differ from typical software engineering research papers surveyed by Shaw (especially, Shaw finds '*method or means of development*' the most common type of research questions among her sample of papers).

Shaw notes that it is typical that imprecise and persuasive research methods are used when the research area is new and there is little previous study. As the research on the area matures, it usually shifts to use more quantitative and formal methodology [28].

1.4 Structure of the thesis

In the chapter 2, I review the theoretical background of this study. After first briefly discussing *software architecture* and *architectural patterns* in general (2.1), I examine a pattern most noteworthy for this study – that is, the *plug-in architecture pattern* – in more detail in 2.2.1.

I decided to approach actual *plug-in systems* (2.2.3) from the perspective of the *software product lines* (2.2.2) – which someone could find a bit surprising. However, I have to say I am satisfied with how SPLs provide us with such a definition of plug-in systems that suits very well for this study's purposes.

The latter half of the background chapter deals with *software interfaces* (especially, *APIs*) and their *versioning* (2.3) as well as *software dependencies* (2.4). The background chapter ends with introducing the most important concept of this study: *a plug-in dependency* (2.5).

Chapter 3 discusses plug-in interoperation from different perspectives. In that chapter, I aim at putting together the most essential aspects within the topic that a software designer should take into account when designing systems involving plug-in interoperation. I also discuss possible solutions to the issues that arise but as we are still discussing plug-in systems in a general level, the solutions in that chapter are more theoretical guidelines than practical *how-to*s.

Chapter 4 concretizes discussion as I take one particular plug-in system, *IntelliJ*, in scrutiny, and observe how plug-in interoperation happens within it. In chapter 5, I demonstrate the subject in a single case study involving an IntelliJ plug-in we developed. Chapter 6 concludes the study.

Chapter 2

Background

2.1 Software architecture

There is no single, universally accepted definition of *software architecture*, and, bearing that in mind, the textbook writers of the field often provide their readers with more than one established definition (see *e.g.*, [2, 5]). Even though definitions vary, most of them, however, seem to lead to the same basic understanding of what kind of matters and aspects software architecture involves.[1]A definition expressed by Clements, Bass and Kazman [5] I find adequate enough to quote here, verbatim:

> The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. [5, ch. 1]

Designing a software system consists of a myriad of design decisions, big and small. Those of them that affect the system's architecture are usually called *architecturally significant design decisions* (ASDDs), and, as a rule of thumb, they can be thought as the most important design decisions concerning the system. A person responsible of making ASDDs is the system's *software architect* (whether or not they are aware of this title of theirs; see [33, ch. 2]).

Taylor, Medvidović and Dashofy [33, ch. 3] make a distinction between *prescriptive* and *descriptive architectures*. The former refers to the architec-

[1]It should be noted, however, that 'software architecture' can also refer to a clearly distinct, though related, concept than what is discussed in this section. That is to say, 'software architecture' can either refer to the structure of a software system (as it does in this section) or a subdiscipline of software engineering that deals with designing and studying such structures (practitioners of which we call *software architects*).

ture of a system as its architect intends it to be, while the latter refers to the architecture that reflects the actual realized structure of the system (during and after its implementation). Acknowledging that software systems are not always implemented according to their intended design, Taylor *et al.* remark that a software system's prescriptive and descriptive architectures may have significant differences. Even without committing to this terminology (or sliding into pessimism about software developers' ability to follow any instructions), it is useful to notice this dual nature of software architecture as a concept.

Clements *et al.* [5] put in commendable effort to convince their reader of the importance of the architecture in a software development process. They highlight architecture's significance in securing that the system obtains its desired quality, but also note how investing in architecture helps predicting the quality of the development outcome, and reasoning about project cost and schedule, among other aspects. Furthermore, Clements *et al.* remark how a good architecture enhances communication and makes it easier to get new team members familiar with the system. From the perspective of this study, their notion of how the architecture strongly determines the modifiability of the software [5, ch. 2] is noteworthy. Finally, Clements *et al.* accredit well-designed architecture for "allowing incorporation of independently developed components" [5, ch. 2] – something that gets conveniently actualized when we discuss the plug-in architecture in section 2.2.1.

2.1.1 Architectural patterns

In their highly influential textbook classic, Gamma *et al.* [12] (a.k.a the *Gang-of-Four*) write:

> One thing expert designers know not to do is solve every problem from first principles. Rather, they reuse solutions that have worked for them in the past. When they find a good solution, they use it again and again. [12, ch. 1]

The quote above captures the incentive behind a *design pattern*: a well tried, reusable solution to a certain design problem. Since the Gang-of-Four book came out in 1994 and consolidated a way and a language to speak of patterns in the context of software, the concept of a 'pattern' have been an irremovable part of the both practice and study of software design.

Although the Gang-of-Four originally considered mainly elements of lower-level design in object-oriented programming, the pattern philosophy swiftl found its way in the discussion on higher-level software architecture, too.

In their book first published in 1996, Buschmann *et al.* presents eight patterns that “express fundamental structural organization schemas for software systems” [4, ch. 2]. These kind of “highest-level patterns” they call *architectural patterns*. Below, I briefly describe two architectural patterns, the first of which is also present in the aforecited book. Besides serving as examples of architectural patterns here, they are referred back in chapter 3 where I discuss ways to construct plug-in code in such a manner that avoids unnecessary plug-in interdependencies.

2.1.2 Layered architecture

Layered architecture is a well-known and often used architectural pattern where the system is divided into several parts, *layers*, each of which concerns one technical aspect of the system [26]. The division into layers can be understood so that each layer provides an interface in a certain level of abstraction [4]. In their implementations, layers use functionality that the lower layers provide them with. The topmost layer is the external interface of the system.

A classical example of layered architecture, also mentioned by Buschmann *et al.* [4, ch. 2.2], is the OSI model of networking protocols (or its ‘real-life counterpart’, TCP/IP stack, that can be found implemented in operating systems). Perhaps an even more familiar example of the layered architecture is the division between presentation, business logic, persistence and database, which Richards and Ford call “four standard layers” [26, ch. 10]. Here, each layer’s level of abstraction corresponds to the conceptual distance of how close its operations are from those of the users, the topmost layer (presentation) providing the user interface.

2.1.3 Modular architecture

The modular architecture pattern is so elementary of its nature that it is seldom even brought up as a pattern of its own. It is, however, illustrative to mention it along with the layered architecture pattern; whereas the layered architecture splits a software system into technical partitions [26, ch. 10], in the modular architecture, the partitions arise from the topography of the domain concepts.

Basically, in the modular architecture, the system is divided into components, each of which implementing one area of the functionality of the system. The components that comes from this division we call here modules.

An example could be an office application with an email client and a calendar (such as Microsoft Outlook). Following the modular architecture, the application would be built in a way that the email functionality is implemented in one module and the calendar functionality in another. While these module components surely interact with each other (when a calendar invitation is sent or received via email), the interaction happens only via the components' public interface.

Modular and layered architecture can be (and often are) applied together. In our example, the office application could be built according to the aforementioned "four standard layers". That gives us $4 \times 2 = 8$ components in total. The interactions between the components now happen either vertically (within the same module but between the layers) or horizontally (on a single layer but between the modules).

2.2 Extensible software

By the 1990s, the software industry had recognized its growing need to be able to develop systems that allow "for components to be plugged into running system when needed" [31]. Obviously, this new requirement brought software engineers also new challenges, several of which were sharp-sightedly identified by Szyperski in his forward-looking conference paper from 1996 [31].

Surely, *extensibility* as a quality factor has been known (or at least intuited by engineers developing software systems) since the early days of computing. Generally, extensibility is understood as an attribute of a software system that indicates how easy it is to add new functionality to the system (*e.g.*, [15, ch. 4]).

In some software projects, it is clear from the start that the system, once completed, will be later extended with new functionality. Even if that was not the case, software architects should bear in mind that a need for new functionality may emerge anyway, as the requirements of a software system tend to change during the development process (they "always" do [29, p. 130]). Thus, extensibility rarely can be completely ruled out in the requirements engineering process.

In the most conventional case of software extension, the system is extended by its original developers or their successors in the same organization. On the other hand, especially in open-source software development, it may happen that projects bring forth multiple forks each of which extends the original system with its own additions (that may or may not be grabbed back to the original project's codebase). In all of these cases, however, combining two or more extensions always requires "a global integrity check" [27], typically in a form that the source code is re-compiled and its functionality is tested or otherwise assured. Thus, combining such extensions demands participation of a software engineer who is familiar with the system.

We end up with quite a different concept of extensibility, namely *independent extensibility* [27], if by an 'extension', we mean such a functional addition that can be included in the system without a global integrity check. This is the kind of extensibility Szyperski discusses in the aforecited [31]. In practice, independent extensibility typically means that extensions are compiled separately from their original systems and only their binaries are combined without recompiling the source code anymore.

Where this leads us is that the original system and its extensions effectively become separate software products released separately but operating together. In particular, if an extension is made by a third-party, that is, someone else than the developer of the system, neither the vendor of the original system nor the vendor of extension is responsible[2] for the whole extended system, but each of them is responsible for their own part of it.

Implementing independent extensibility calls for a specialized software engineering approach where software systems are not developed to be ready-made applications but open platforms on which new software products can be built. In the next section, we take a look at an architectural pattern designed to solve the programmatic side of the problem, and, in section 2.2.3, we discuss systems that, in practice, provide independent extensibility.

2.2.1 Plug-in architecture

The *plug-in architecture* pattern is an architectural pattern that targets to a system design with improved extensibility, variability, and modularity [26, ch. 12]. The description of plug-in architecture in this section is mainly based on the mentioned source, but the pattern is similarly described by many other authors, too, including Buschmann *et al.* [4, ch. 2.5] and Tarkoma [32, pp. 118–9].

(Before going into details, it should be noted that when we speak of 'plug-in architecture' we are not (yet) dealing with software artefacts we call 'plug-ins' in this thesis, but entities of more general level, namely 'plug-in components' as I like to call them. The plug-in architecture pattern can be, and definitely has been, applied to software systems of a great variety, only a small subset of which we consider actual 'plug-in platform systems'. See section 2.2.3.)

[2]I use word 'responsible' here vaguely, and mainly to reflect which party is considered 'the author' of the software (to whom, for example, users are expected to report bugs they find in the software). Actual (legal, moral, *etc.*) responsibilities of software makers I leave totally untouched here.

As its alternative name, *microkernel architecture*, hints, the plug-in architecture pattern has its historical roots in operating system design [25]. In the context of operating systems, microkernel architecture refers to a design where only the most essential functions of the system are executed in the kernel mode, while the vast majority of the operating systems functionality (for example, file systems and networking) are implemented as separate services that are run in user mode, essentially the same way as any application that is run in the system. This decoupling of kernel and and other OS services makes it easier to develop operating systems with better customizability and flexibility. [30, ch. 2.4]

The plug-in architecture pattern generalizes the idea behind this style of operating system design to all kind of software systems. The software component that takes the place of the operating system kernel in the pattern, is called the *core system*. Analogically to microkernel, it only contains the most essential functionality of the system. The non-essential features of the system are implemented as separate *plug-in components*, each of which can independently[3] either be included into the system or excluded from it.

The core system is not aware of any of the possible plug-in components that can be plugged into it. Instead, it discovers the plug-in components that are currently present by reading the *registry*, that is, a data structure inside the system where each plug-in component adds an entry, when they are included into the system. The entry contains information of the plug-in, based on which the core system knows how to call to the plug-in component's code when appropriate.

The communication between a plug-in component and the core system follows a *contract*, format of which is dictated by the core system design. The contract defines an entry point where the execution of the plug-in component's code begins, and what kind of data is transferred between the plug-in component and the core system and how. Plug-in components are built to fulfill the contract.

Usually, a plug-in component code shares the memory with the core system code, thus making it possible that the communication between the two can be simply implemented via normal function calls from the core system to a plug-in component and vice versa. The transferred data can then be objects or data structures passed as function parameters and return values. For example, in Java, it is convenient to define a plug-in contract as an interface that a class inside the plug-in component must implement.

[3] I say 'independently' here as I am describing the ideal format of the plug-in architecture pattern. Obviously, if that principle could always be followed in practice, there would not be the problem this thesis tries to solve.

Though not that usual, it is also possible to design the system in a way that plug-in components do not share memory with the core system. In that case, the communication must happen by some other means, *e.g.*, via sockets. In addition, the data must be converted into a serial format (such as XML) so it can be transferred between components. It is even possible that a plug-in component is not located in the same computer as the core system, and the two communicate only via network.

Utilizing the plug-in architecture, one can design software systems that are easily extensible. To add a new feature, it is enough to implement a new plug-in component with the desired functionality and introduce it into the system. Development of different features can happen without any overall coordination, and the code of the core system does not have to be changed at all.

By selecting some plug-in components to be included and others to be excluded, one can create multiple variations of the same core system with different sets of features. This is especially useful, when building a *software product line.*

2.2.2 Software product lines

Apel *et al.* motivate software product lines engineering as a software engineering approach that makes it possible to construct individualized software solutions “with the benefits of mass production” [1, p. 8]. Taylor *et al.* go as far as nominating it for one of the possible “silver bullets” of software architecture [33, ch. 15]. I discuss software product lines here briefly as it makes it probably easier to understand plug-in systems if we think of them as a special instance (or kind of a mutation) of software product lines.

By the Software Engineering Institute’s definition (qtd. in [5]), a *software product line* (SPL) is

> a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. [5, ch. 25]

The “common set of core assets” shapes a *platform system* which have several *variation points.* In each variation point, it is possible to make a selection of whether or not to include certain software components into the system. These variable software components correspond to variable features of the system. Each distinct selection of variable features yield a *variant* of the system. Variants with meaningful sets of features are made into products of the SPL.

The architecture of SPLs resembles the plug-in architecture model. The core system of theirs is the platform system where features can be introduced as plug-in components.

Scholars have listed multiple advantages that software product lines have over the traditional model, where each software product is manufactured as a separate project. These advantages include reduced cost and shorter time in which new products are brought to the market [1, 23]. Use of SPLs in software production is not, however, only a matter of efficiency. SPLs are also said to improve quality in software products due to the fact that their core components are used in multiple systems and, thus, bugs in them are more easily found and fixed [1, 3]. What is more, with software product lines, vendors can provide their users with a diverse portfolio of software systems with similar user experience, making it easier for the users to adopt new tools [3].

2.2.3 Plug-in systems

In this thesis, I use term *plug-in (platform) system* to refer to a software system that Bosch [3] (from SPL engineering perspective) calls a software product line with an "open ecosystem". The plug-in systems are software systems, structured by the plug-in architecture, whose plug-in components, *i.e. plug-ins*, are not developed exclusively by the vendor of the platform system (or its subcontractors) but also by third parties without any party coordinating the overall development and deployment process[4].

In contrary to usual software product lines, when it comes to plug-in systems, it is not the producer of the software who combines certain features to make a software product. The *users* themselves select the plug-ins they want to include into their own tailored instance of the platform system.

The previous sentence is actually less of a statement and more of a definition of what I mean by a *user of a plug-in system*: that simply is, the person who selects and installs plug-ins. Depending on the nature of the plug-in system, that person, a user, may be an actual end-user of the software (such as a programmer writing code with IntelliJ IDEA) but not necessarily. A 'user' can also be, *e.g.*, an information systems technician (in a company with centrally managed software installation) or an administrator of such a web service application that embodies a plug-in system. In the context of this thesis, we are interested not that much of who, in reality, uses the

[4]It is, however, possible, and even usual, that the original vendor validates the external plug-ins before they are released. Additionally, the platform manufacturer may give only its certified partners permission to develop plug-ins. [3]

plug-in system but whose responsibility it is to configure and maintain its installation.

It is typical that a centralized catalog of existing plug-ins is maintained somewhere where it can be browsed by the users of the plug-in system (almost always it is located in the Web). Often the catalog can be browsed straight from the plug-in system software itself, and plug-ins can be purchased, downloaded and installed conveniently using a specialized plug-in manager tool. The catalog may be administrated by the platform system manufacturer or it can be maintained by the user community without any supervision 'from above'. Ratings and reviews given to the plug-ins by their users seem to be customary to these catalogs, undoubtedly, in order to help users to find most useful plug-ins.

2.2.4 Examples of notable plug-in systems

Many of the most-used programming and software development tools provide their user communities a way to extend their functionality with plug-ins. Classical text editors Emacs and Vim both support third-party plug-ins having long traditions and broad supply on them [14] [22, ch. 3]. Similarly, Atom, a newer programming editor, is designed to be easily extendable via its plug-in framework [13, ch. 2]. Popular modern IDEs such as Eclipse [7] and IntelliJ IDEA (latter of which is later discussed in large), have successfully followed this model and established active ecosystems for their third-party plug-in vendors.

Perhaps the best-known example of a plug-in system software is Mozilla Firefox web browser, whose most-used plug-ins have several millions of users worldwide[5]. It also serves as a good example of a plug-in system where users are not expected to be more tech-savvy than an average computer user.

Many large-scale software products with a wide userbase have also adopted the 'open ecosystem' approach even if extensibility were not their primary asset. Thus, for example, Microsoft's Office product line provides its users a possibility to extend their software with third-party plug-ins [20]. On the other hand, a highly extensible design with plug-in support have helped some smaller software to gain popularity among users. As an example of such a small-scale software product with a substantial plug-in developer community, one can mention *foobar2000* audio player [10] for Windows.

Interpreting our definition of 'plug-in' broadly, we can consider even mobile apps (such as those of Android system) plug-ins: unlike full-featured

[5]The list of Firefox browser add-ons sorted by number of users can be found here: `https://addons.mozilla.org/en-US/firefox/search/?sort=users`

(desktop) applications, mobile apps resemble plug-ins in a way that they have a platform system, that is, the mobile OS itself, which does not allow more than one instance of each app to be installed and executed at a time. In contrary to how most plug-in systems are designed, mobile apps do not share memory with each other let alone their platform system. That was not, however, a restrictive trait in our definition of plug-in.

2.2.5 About terminology

Different platform systems use a grand variety of terms when referring to their plug-ins. These terms include *plug-in* (or *plugin*), *extension*, *add-on*, *add-in*, *package* and *component* among others.

Within some plug-in systems, conceptual differences are seen between different terms. Hence, for example, Mozilla Firefox has traditionally distinguished *plug-ins* from *extensions* by certain technical criteria. As, nowadays, Firefox no longer support ‘plug-ins’ but only ‘extensions’, risk of confusion over terminology is low. Firefox also uses term *add-on* as an umbrella term for all kind of extra flavorings a user can add to their browser including aforementioned ‘extensions’ (and previously, ‘plug-ins’) but also things that hardly fit in our definition of plug-in, such as language translations and visual themes. See [21].

In this thesis, I persist in avoiding all confusion that might follow from varying terminology and, thus, use exclusively term *plug-in*.

2.3 Interfaces of software

Software are designed to interact with entities external to it. These entities include human users – in which case, we speak of human-computer interaction – but also other software. (Furthermore, software can interact with the physical environment via sensors and actuators but we do not discuss them here.)

To be able to interact coherently, software must have some kind of definition of by which means it can be given input, how it reacts to that input, and how it provides the output. This definition is called an *interface*.

For human users, a computer program provides a user interface (UI). Most common types of a user interface are command-line interface (CLI) and graphical user interface (GUI), both of which are surely familiar to an experienced computer user.

When software programs interact with each other, it is possible that they utilize the same interfaces as human users do. Command-line interfaces are typically designed to be called in batch scripts, and even large-scale applications may use CLIs to interact with, say, a file system or a version control system. In Unix pipeline, this kind of use of CLI in inter-software communication is harnessed to create a powerful, yet intuitive-to-use mechanism for batch processing.

Programs interacting via graphical user interface are more unusual and not without reason: unlike CLIs, GUIs are not based on a serial representation of information (such as text) but two-dimensional visual layout that is, undoubtedly, more difficult to process programmatically. As an example of software-to-software interaction via GUI, we can mention *web scraping*, a technique where software extracts information from HTML code sent by a server program and meant to be graphically rendered to the user in a web browser.

Even though it is possible that software interaction happens via UI, most often, that is not an optimal way. UIs are designed for human users, so they present data in a format that is easily readable to a human but not necessarily to a computer program. What is more, users can easily adopt to minor changes in the UI, whereas even the smallest change in the input/output protocol of an interface may require changes in a computer program using the denoted interface. Thus, in the most cases, it is preferable to design a separate interface that allows other software interact with the computer program, if that kind of interaction is needed.

An interface, through which other software can provide input and get output, is called application programming interface (API). Like UIs, APIs must be carefully designed to ensure their success.

2.3.1 Versioning

It can be expected that a software product needs to be changed after it has been first released. For being able to distinguish between different published versions of the software, it is customary that each version of the software is identified by a unique version number.

Changes between versions may or may not concern the API. Changes that do not concern the interface but only its implementation are either bug-fixes or refactoring. Even though versions that differ from each other only by such changes (*i.e.*, *patches*), appear functionally identical, they should still have unique version numbers for bug tracking purposes.

2.3.2 Backwards-compatibility

Changes in APIs can be classified according to whether or not they maintain the backwards-compatibility of the API.

A version of an API is said to be *backwards-compatible* to an older version of the same API, if for any allowed input, the new version of the API returns output (or otherwise reacts) according to the specification of the old version of the API. Changes that do not maintain backwards-compatibility are called *breaking changes*.[6]

If a new version of an API is backwards-compatible, to an older version, a client program can replace the old version with the new version without errors. Then again, breaking changes may cause the client program to fail. If a client program is updated to use a new version of an API which is not backward-compatible, its integrity must be re-assured and potential errors fixed.

Due to this difference, it is important for the developers of a client program to know whether a new version of an API is backwards-compatible. Generally, there is no way to automatically confirm that, so it is most reasonable to consider it the vendor's responsibility to announce to which older versions (if any) a new version is backwards-compatible.

2.3.3 Semantic Versioning

In software industry, it has become customary to use version numbering that consists of several numbers, often separated by a dot, which have descending significance when read from left to right, from the 'major' version number to more minor version numbers. Thus, for example, version '2.1' is generally understood to be more advanced than version '1.18' of a same software product. Regrettably often, these kinds of numbering schemes do not carry any other meaning than what the vendor considers a big enough change to bring about an increment in the major number (which decision may be more affected by marketing purposes than the technical reality of the product).

A welcome attempt of standardizing version numbering schemes in a way that does not only specify the syntax of version numbers but also their semantics, is known as *Semantic Versioning*, or *SemVer* [24]. In Semantic

[6]Thus, changes that remove functionality from the API are always breaking. On the other hand, changes that extend the API are not breaking, because they are only affected by inputs that would have not been allowed in the old version of the API. Changes that alters the way the API reacts to allowed input are not necessarily breaking, namely in cases where the change strengthens post-conditions defined in the old API but does not weaken them.

Versioning, the version numbers are of format **MAJOR.MINOR.PATCH**, where an increment of **MAJOR** part indicates a breaking change. An increment in **MINOR** indicates a new functionality and increment in **PATCH** a bug-fix; neither of them must not contain breaking changes, that is, they must be backward compatible.

The advantage of Semantic Versioning comes apparent with an example: Software A uses the API of software B. A was developed and integration-tested with version 2.3.5 of B that proclaims it uses Semantic Versioning. Now maintainers of A know that when versions 2.3.6, 2.4.0 and 2.5.0 of B are published, they must all be compatible with A. On the other hand, version 3.0.0 cannot be assumed to be compatible because it is known to contain breaking changes. Similarly, versions 2.2.0 and 1.8.0 are not necessarily compatible as they may lack functionality used by A. Finally, version 2.3.5 is known to be compatible by its interface but may contain bugs that were not found in the original integration tests of A.

As the example shows, Semantic Versioning makes it possible for maintainers to reason of backwards-compatibility only by looking at the version numbers of the APIs they use. It is important to note that committing to Semantic Versioning has little benefit unless it is also known by the developers of the client software. Thus, developers should explicitly mention when they are using Semantic Versioning. Also, one should not claim to follow Semantic Versioning unless they adhere to it to the letter.

2.4 Software dependencies

When a piece of software uses functionality provided by another piece of software, we say that the former is *dependent* on the latter. Utilizing external programs and libraries is a reasonable way to reduce the amount of work needed to implement software products. However, being dependent on other software makes a computer program also affected by the bugs and vulnerabilities of that software. Thus, keeping the dependencies up-to-date is an important part of software maintenance.

When accessing to its dependency, a computer program uses some of the interfaces provided by that dependency. Ideally, the interface should be an explicitly defined and documented API. If the dependency does not rely on an explicitly defined interface, any change in the dependency software should be considered potentially breaking. A thorough *dependency testing*, where the functionality of the dependency software is tested in a unit-test-like manner, is advisable, to raise the confidence in that it works as assumed – also after a (potentially breaking) change.

2.4.1 Library linking

Perhaps the most familiar way of software reuse is library linking, that is, making a program include other software's binaries (or, in some cases, unmodified source files) in its executable code and call their procedures from the main program. Linking can happen either in compile (*static linking*) or in execution time (*dynamic linking*).

Dynamic linking has certain advantages over static linking: it keeps executable files smaller and allows bugs in libraries to be fixed without recompiling all software using them. Furthermore, some software licenses (most notably, LGPL [11]) offer more relaxed terms for reuse via dynamic than static linking. Dynamic linking, however, is more prone to dependency problems, especially if libraries are distributed separately from the programs that use them: making sure that the required library can be accessed in the run time, and it is the same version with which the program was developed with can be tricky.

While (dynamic) library linking has still a reputation as a common source of software errors, it can be conjectured that modern build systems and centralized project repositories have mitigated the problem from what it used to be when terms such as 'DLL Hell' and 'Dependency Hell' were coined a couple of decades ago. Some of the issues that once were associated with library linking now remains with plug-in dependencies as later discussed in 2.5.2.

2.4.2 Application interoperation

In addition to library linking, there are other ways in which pieces of software can be dependent on each other. These includes cases where neither of the software in question is in a position of a library but both of them are run as processes of their own, either in a same machine or separate machines. If none of them exists for the sole purpose to serve the other, we can talk of *applications* that interoperate together.

Application interoperation can happen in a multitude of ways, including via buffers, sockets and remote procedure calls. The API of the dependee software can be built on top of protocol standards such as HTTP(S) and SOAP, the support to which can be found for most of the popular programming languages.

The dependent application may itself launch the dependee application, or it may require it being already running, either locally or in a remote server specified by a URL. To resolve compatibility, the version information is usually either incorporated in the communication protocol ("handshakes" containing information of the version of the API), or, in cases of remote applications, each new version is published in a separate URL, while the old versions are kept alive for a reasonably long deprecation time.

2.5 Peculiar case of plug-in dependencies

In this section, I briefly illustrate how the question of software dependencies changes when we no longer speak of stand-alone applications or libraries but *plug-ins*.

2.5.1 What is a plug-in dependency?

I use the phrase *plug-in dependency* to mean one plug-in's dependency on another plug-in *of the same platform system*. Surely, a plug-in could be dependent on a plug-in of some other system but that situation hardly differs from that where a stand-alone application depends on another application, which were already discussed in section 2.4.2, so further discussion on that is omitted.

A plug-in dependency can be of any kind software dependency. For example, a plug-in may

- use another plug-in as a library,
- be built upon another plug-in that works as a framework for it,
- pass messages that it expects another plug-in to handle.

When it comes to the third point above, the platform system can provide plug-ins with a specialized message-passing framework or plug-ins can also use other means (*e.g.*, sockets) to communicate with each other.

Plug-in dependencies may be explicitly declared to the platform system, which may also check that all the dependencies are loaded in the system before enabling a plug-in. On the other hand, plug-ins may depend on other plug-ins without the platform system knowing of it. In that case, it is the user's responsibility to ensure that all the necessary plug-ins are present.

2.5.2 Next circle of 'Dependency Hell'?

As we saw in the previous sections, managing software dependencies sets a challenge in system maintenance. When these dependencies are between plug-ins of the same system, the situation is often even trickier.

In their study, Klatt and Krogmann [19] give a good example of a situation where dependency problem can become virtually impossible to solve: “if two extensions A and B require different versions of a third extension C but the software supports only one instance for each extension”.

Another kind of problem emerges for a maintainer of a plug-in with a dependency to another plug-in, whose maintenance is discontinued. Such a dependency can prevent the dependent plug-in from being ported to a newer version of the platform system, if that newer version no longer supports the obsolete dependency plug-in.

Similar issues are sometimes faced with library linking. A term ‘DLL Hell’ got popular in the 1990s to represent dependency issues related to *dynamic-link libraries* (DLLs) of Microsoft Windows. An umbrella term, *Dependency Hell* is also used in relation to problematic software dependencies in general. When turning discussion into plug-in dependencies, I have a thankless task of Virgil to take us to the inner circle of this inferno.

Compared to traditional library linking, plug-in dependencies are difficult to manage at least for the following reasons:

1. **Only one version of a plug-in can exist in a system at a time.** While this is the case also with some library linkers, more advanced ones have successfully solved it. For plug-ins, there is generally no way to overcome the restriction.

2. **Users may update dependee plug-ins at any time.** In more traditional library linking, when an old version of a library is replaced with a new one, the program is first integration tested, and only after that users get this version with the new library in it. Within plug-ins, it is generally impossible to stop users from installing a new version of a dependee plug-in as soon as it is released. There is no time window for integration testing of the dependent plug-in.

3. **Users have very different sets of plug-ins.** The coexisting plug-ins (possibly dependent on the same other plug-ins) can cause issues that are hard to predict – and hard to solve.

4. **Dependency problems must be solved by the user.** Depending on the system, users are not necessarily technology experts. Furthermore, they are not necessarily that aware of the interdependencies between the plug-ins.

Chapter 3

Aspects of plug-in interoperation

Implementing plug-in interoperation is a joint challenge of three software designers often working in different companies at different times and rarely communicating with each other by other means than those of software documentation. These are designers (or designer teams) of

- **plug-in system**: the plug-in system in question,
- **dependee plug-in**: a plug-in for that system, and,
- **dependent plug-in**: another plug-in for the same system dependent on the preceding one.

Ideally, each of these parties should take the possibility of plug-in interoperability into account in their design.

In this chapter, I discuss software design issues regarding plug-in interoperation from the point of view of each of the three above-mentioned parties. Each point of view is covered in a section of its own. The sections are in the same order in which the software are also implemented: first, **plug-in system** (in 3.1), then **dependee plug-in** (3.2), and, last, **dependent plug-in** (3.3).

It should be remembered that, of course, not all plug-in systems need to support plug-in interoperation, and neither do all plug-ins. The decision of leaving plug-in interoperability out of the requirements should, however, be a result of justified reasoning and not only because of lack of consideration. Especially, when designing a large plug-in system, it is difficult to foresee all possible ways of how plug-in developers will extend the system, so one should be extra careful when making that decision.

3.1 Designing plug-in system for interoperability

A plug-in system designer should communicate clearly their intention on whether or not plug-ins may interact with each other. Even if the designer of the system could not give a categorical answer to the question, their competent speculation over the issue could become valuable for plug-in developers who consider making their plug-in dependent on another plug-in.

A designer of a plug-in system should not be mistaken to think that solely not preventing plug-ins from interacting with each other would be sufficient to make plug-in interoperation a viable option for plug-in developers. Should they want to encourage plug-in interoperation, the least a designer of a plug-in system could do is to demonstrate to plug-in developers, what they think is a possible way to implement it. Preferably though, they should consider the rest of this section 3.1.

It should also be noted that although there is no strict lines between interoperation-supporting features that should be implemented by the platform system and features that could as well be left for plug-in developers to implement, it can be highly beneficial to implement them already as a part of the platform. If the plug-in system does not, for example, verify that all dependencies are satisfied when installing new plug-ins, that issue shifts to be solved individually by plug-in developers, or ultimately, users. Solving the problem once in the system level, can, thus, save a considerable amount of development work when considering the totality.

3.1.1 Explicit dependencies

If a plug-in system is intended to support plug-in dependencies, it is useful to make these dependencies visible to the system and make the system manage them. Each plug-in, when it is registered into the system, declares other plug-ins it depends on. The plug-in system can then inform user they have to install the dependee plug-ins before enabling the dependent plug-in. In addition, the system may automatically install the dependee plug-ins if they are available in a known plug-in repository.

In most direct mode of plug-in interoperation, two plug-ins are (dynamically) linked in the program code level and share the memory, allowing one plug-in to call functions and subroutines of the other, and pass not only values but also pointers to memory objects from plug-in to plug-in. It is, hence, a responsibility of a plug-in system to make sure that when a plug-in is installed, its program code is loaded in such a manner, it has an access to its dependees.

When a plug-in system is designed in this way, plug-in interoperation may even start looking like a solved case: a developer of a dependent plug-in can program their code pretending the code of the dependee plug-in was linked to it as a library. In the run time, when the dependent plug-in is installed into a plug-in system instance, the system automatically handles it so that the code of dependee plug-in is there and accessible to the dependent plug-in just like it was thought to be. Unfortunately, this was only the easy part of the issue.

3.1.2 Versions of dependencies

As discussed in chapter 2, the main challenge of plug-in interoperation is not making it happen once but maintaining it when newer versions of interoperating plug-ins gets published. Let us consider a plug-in system instance where there are installed a plug-in A and, as its dependency, plug-in B. If a new version of a dependee plug-in B is then installed to the system (either by the user or as a dependency of some other plug-in, say, C) the original version of it (in relation to which plug-in A was developed) cannot be kept in the system just for the plug-in A. The plug-in system must either disable plug-in A or keep it in the system and let it interoperate with the new version of B. Which option is the right thing to do, depends, of course, on whether or not the new version of B is compatible with the interoperation scheme that A used with the old version of B.

It becomes apparent that if a plug-in system is able to differentiate those version updates that maintain backwards-compatibility from those that do not, it can manage the issue in a quite sophisticated manner disabling dependent plug-ins only when their dependees gets switched to a version no longer backwards-compatible to the version that the dependent plug-in was developed with. It it, thus, useful if plug-in developers can somehow inform the platform system with which older versions of their plug-in a new version is backwards-compatible and with which it is not.

As noted before, Semantic Versioning, when correctly applied, offers a convenient way to differentiate between backwards-compatible versions and those versions that contain breaking changes. Hence, it appears a good solution for the purpose. Of course, to be able to automatically utilize the semantics of SemVer, the platform system must know, which plug-ins actually adhere to it in their versioning, or – which may be considered too restrictive – require it of all the plug-ins.

3.1.3 Circular dependencies

To end this section, let us briefly consider possibility of circular dependencies between plug-ins. Even though circular plug-in dependencies may not be a problem *per se*, they are virtually always a symptom of poor design: if two (or more) plug-ins can only be used together, they could as well be one plug-in. Supporting circular plug-in dependencies is, thus, hardly necessary for a plug-in system. As circular dependencies bring extra complexities to dependency management, it is advisable to simply prohibit them.

3.2 Designing plug-in for interoperability

Interoperability, extensibility and reusability are quality factors that may easily become diminished in favor of other properties in software requirements engineering. These properties, however, increase their importance when it comes to plug-in development: a plug-in is not designed to operate in a stand-alone manner but as a part of a plug-in system that is extensible by its very nature.

3.2.1 Modularizing plug-in code

Acknowledging the challenges of plug-in interoperation, a designer should not overlook possible ways to circumvent the problem altogether. Applying modular and layered software architecture to the plug-in design, allows other software (including other plug-ins) re-use some functionality of the plug-in by the means of usual library linking without making actual dependencies between the plug-ins themselves.

To adopt this solution, a developer must be careful to keep code that is dependant on the platform system separated from the code that can be used by other software. A natural architectural choice to achieve that is layering the software in such a way that only the topmost layer refers to the platform code. When functionality provided by the platform needs to accessed from the lower layers, it should happen via a *bridge*: the lower-level layer defines an interface – that is, a bridge – for which the topmost layer provides an implementation. The lower-level layer can use that implementation of the interface without being coupled with the framework code it uses.

This way, all the lower-level layers can be reused as a library by other software without coupling to the platform system. The dependent software must provide implementations of their own for bridge interfaces in the library code equipping them with the same functionality that was originally got from the platform system. If the dependent software is a plug-in of the same platform system, the implementation can simply refer to the same procedures as the original implementation did.

For even better reusability, the different areas of the plug-in's functionality should be put into separate modules, each of which is then made available as a library of its own. Hence, when some other software needs to reuse some functionality of the plug-in, it is enough to link to the module that is related to the desired functionality.

It should be noted that it is not always meaningful or even possible to circumvent an actual plug-in dependency this way (see section 3.3 below for closer discussion). The rest of this section considers what a developer still can do to make their plug-in more suitable as a dependee role in plug-in interoperation. However, preparing a plug-in for actual plug-in interoperation does not prevent from also applying layered and modular patterns as described above: they can still be useful for other reusers.

3.2.2 Programming interface design

To make a plug-in suitable for plug-in interoperation, it is crucial to define a clear programming interface via which other plug-ins can access its functionality. While there are many possibilities how the plug-ins can communicate (e.g. via buffers or I/O streams) most typically it happens either via calling each other's procedures or via a messaging framework provided by the platform system. Especially in the former case, making it explicit which classes and methods are part of the public API allows non-public parts be modified without possibly introducing new breaking changes. For those parts, that are declared belonging to the API, breaking changes should be avoided if only possible.

If a platform systems has a messaging system, utilizing that for plug-in interoperation should be considered as an option. If the communication between the plug-ins only happens via explicitly defined types of messages, it is easy to keep the interface (in this case, the messaging protocol) coherent, and the risk that implementation details, by oversight, start to leak onto the API stays small.

The similar benefits for a conventional object-oriented API provides the *facade* design pattern (see [12, ch. 4]), where the whole API is concentrated into a single class delegating the calls to the actual methods responsible for that functionality. Using facade, it is also straightforward to keep supporting an old API even if there was a need for API re-design: just have a separate facade class for different versions of the API.

No matter how the API is designed, it should be properly versioned.

Following Semantic Versioning standard – and also clearly announcing that – is generally recommendable to make a clear distinction between breaking and non-breaking changes between the versions. Of course, one should make sure to comply with the versioning guidelines of the platform system. Letting developers of dependent plug-ins know beforehand when a part of the API is losing support is important, and those parts should be labelled as deprecated at least one release cycle before they are removed from the API.

3.2.3 Planning maintenance

If we think of, for example, some Java library (without any external dependencies) available as a JAR file and assume it contains only a few bugs and uses only non-exotic features of the standard library, we can quite safely trust the same exact bytecode can still be used in new Java programs after a few decades. The same does not generally apply to plug-ins.

Most plug-in systems tend to keep changing in a faster pace than well-established programming languages and their runtime systems. After all, they are developed first and foremost for their users, so backwards-compatibility of its programming interface takes the second place to fulfilling users' ever-changing needs. Thus, to keep a plug-in working in a new versions of the platform system, it must be actively maintained.

When it comes plug-in interoperation, it is important that the developers of the dependent plug-in get information of the maintenance plans of the dependee plug-in. That way, they can react in time, if the dependee plug-in is in danger to become obsolete. Ideally, the maintenance of the dependee plug-in could be handed to some other party It stands to reason that well-documented and structured code makes the handover a lot easier.

3.3 Making a plug-in to interact with another plug-in

In this chapter, we change our point of view, and start looking at the issue of plug-in interoperation from the perspective of a developer of a plug-in dependent on another plug-in. But before making a plug-in dependant on another, the developer should first consider if it is really necessary. The question is relevant, of course, for all the external dependencies – not only plug-ins: All external dependencies cost maintenance effort, and thus, should not be applied without a good reason. Because of the extra complications related to the plug-in dependencies, the consideration should also be carried out extra carefully.

We can (by cutting some corners) classify two different objectives of making a plug-in dependant on another:

- **Reusing existing code.** By using a functionality provided by another plug-in developers saves the effort of implementing the same by themselves. Existing software, especially if widely used, can also be expected to work more reliably than newly-created solutions.
- **Adding functionality to the other plug-in.** Sometimes it is not the platform system itself that one wants to extend but an existing plug-in of it. If the plug-in does not have a plug-in framework of its own (as it usually does not), the extension can be implemented as a plug-in that interacts with the original plug-in to add desired functionality to it.

While the classification is all but definitive, it is useful when considering the necessity of the plug-in dependency. To better understand the classification above, one can think that, in the former case, the user uses the dependent plug-in, which relies on the dependee plug-in, whereas, in the latter case, the user uses the dependee plug-in, whose functionality is extended by the dependent plug-in.

In the latter case, it is not meaningful to avoid plug-in interdependency. By contrast, in the former case, there is – at least theoretically – a possibility to circumvent plug-in interoperation by just re-implementing the necessary functionality in what would have become the dependent plug-in. Depending on the licensing and the design of the (would-be) dependent plug-in, it is possible that parts of its code can be linked as a library or a submodule to the dependent plug-in (this is where modular design of the dependent plug-in proves to be useful). In any way, the developer is in front of the question whether they should depend on the other plug-in or include that functionality in their own plug-in. Even though the first option may look easy (compared to writing and maintaining code that works similarly), it is important to fully realize the cost that plug-in dependency brings on with its complexity. Even if a developer decided to reuse in their plug-in some functionality provided by another plug-in, they should plan ahead what shall be done in a case where maintaining the dependency becomes impossible, and take it into account in their system design. One should also carefully study how does the future development of the dependee plug-in look like before making the decision.

3.3.1 Encapsulating the dependency

To relieve the maintenance effort, the parts of the code of the dependant plug-in that interact with the dependee plug-in should be encapsulated from the rest of the code. The interface between the encapsulated code – let it be called an *adapter* – and the rest of the plug-in should be designed to meet the needs of the plug-in itself: one should be careful not to just copy the design of the dependee plug-in's API, because it is not necessarily optimal from the perspective of the dependent plug-in's system – letting the API design "leak" strengthens the coupling between the plug-ins.

Using this kind of encapsulation and the adapter pattern[1] has various advantages. Apart from making the system design more structured and, thus, more understandable, it also minimizes the effect of problems that may occur if the dependee plug-in introduces breaking changes. In an extreme case, where the dependency must be abandoned (for example because of discontinuation of the dependee plug-in), the adapter pattern allows the dependee plug-in to be replaced with another component without changes in the rest of the system.

3.3.2 Taking other plug-ins into account

A developer should keep in mind that plug-in interoperation does not take place in a vacuum but as a part of a system where other plug-ins may interfere unpredictably. It is generally impossible to even think about – let alone test – all possible combinations of potentially interfering third-party plug-ins, when designing and implementing plug-in interaction. Hence, the best a developer can do, is stick with good design guidelines and trust (or hope) that developers of other plug-ins do the same. Essentially, it means that a dependent plug-in should not block other plug-ins to also using its dependee plug-in.

Situations where two plug-ins are both separately interacting with a third one and ending up in a mutual conflict may become difficult to fix for the user because it is difficult to reason which party is the "culprit". For the same reason, the error reports do not necessarily end up to the correct party who could fix the issue. When dealing with users' error reports, a plug-in developer should find out if there have been any third-party plug-ins present when the error happened.

[1]For the *Adapter* pattern, see [12, ch. 4].

Chapter 4

Plug-in interoperation in IntelliJ

4.1 IntelliJ

IntelliJ[1] is a software product line of integrated development environments, *i.e.*, IDEs. It has been developed by a software company named JetBrains along with the open source community. In the core of the product line, there is *IntelliJ Platform*, on top of which several IDEs have been built by JetBrains itself as well as third parties taking advantage of its permissive Apache 2.0 licensing. The IDEs of this product line include, most notably, IntelliJ IDEA (by JetBrains) and Android Studio (by Google), both of which are widely used in the software industry.

Undoubtedly, a crucial factor in IntelliJ's popularity has been its broad support for extensions via its mature plug-in system. As of early 2021, there are a total of 5343 plug-ins for IntelliJ Platform available in JetBrains Marketplace, a centralized plug-in repository on the web, and the most popular plug-in there has been downloaded more than 19 million times [16].

4.2 Architecture of IntelliJ's plug-in system

IntelliJ is run in Java Virtual Machine (JVM). The plug-ins for IntelliJ Platform are, thus, developed in JVM languages such as Java, Scala, and Kotlin. For distribution, plug-ins are packed as ZIP archives containing the plug-in

[1]It seems that JetBrains does not officially use name 'IntelliJ' (or, for that matter, any distinct name) when referring to the product line based on IntelliJ Platform. I use it here to differentiate between the product line (IntelliJ), its flagship product (IntelliJ IDEA), and its platform system (IntelliJ Platform).

code in JVM bytecode (a JAR file) as well as the JAR files of the libraries used by the plug-in.[2]

Being faithful to the principles of the plug-in architecture pattern, IntelliJ has only its most essential functionality in its core platform system. Each IntelliJ product has its own set of "built-in plug-ins" (called *modules*) providing features that are relevant to that product. In addition, each IntelliJ product comes bundled with plug-ins that are thought to be useful for most of the users. Even though many of the features provided by these bundled plug-ins may appear to a user as irremovable parts of the software, they actually are as loosely coupled to the platform as any plug-in, and can even be unloaded from the system by the user.

Loading plug-ins into the system happens in run-time. IntelliJ provides a tool for fetching plug-ins from plug-in repositories, and loading them into the system. Depending on the properties of a plug-in, a restart of the IntelliJ system might be required to get the plug-in's functionality into use.

IntelliJ Platform provides plug-in developers with a software framework, on top of which they can design their plug-in. The interaction between the platform system and plug-ins happens via this framework. In addition to defining abstract methods that plug-ins can implement to enable inversion of control[3], (these are the extension points of the platform system, see below) and classes modelling the domain objects of the platform, the framework contains a great deal of utility classes and GUI components that plug-ins can utilize (or even *should* utilize, in order to avoid inconsistent behavior). IntelliJ also provides its plug-ins with a messaging infrastructure that implements the *Publisher-Subscriber* pattern[4] [17].

When a plug-in is loaded into an IntelliJ system, its descriptor file is first read from its ZIP archive where it was included as a resource file of the plug-ins own JAR when the plug-in was built. The descriptor file, named **META-INF/plugin.xml**, is an XML file containing meta-information such as the plug-in's name, its version number and a short description of it but also entries defining to which extension points a plug-in connects and names of the classes that contains the relevant code.

[2]IntelliJ has a (far from complete but still impressively extensive) online documentation of its platform, known as IntelliJ Platform SDK [17].

[3]As software frameworks in general (or, arguably, by definition), the framework of IntelliJ Platform is designed according to *inversion of control*: the program flow of plug-in code is controlled by the framework, which then calls the plug-in code in certain parts of the program execution. For more about inversion of control, see [15, ch. 6]; *cf. Template Method* design pattern [12, ch. 5].

[4]For the *Publisher-Subscriber* pattern, see [4, ch. 3.6], *cf.* the *Observer* pattern [12, ch. 5].

The possible extension points include:

- *Actions.* Plug-ins can extend the platform by registering new actions: executable commands that can be triggered via *e.g.* a menu item, button or key shortcut. In fact, actions are considered an extension mechanism so fundamental that in IntelliJ's own terminology they are set apart from other 'extension points' as a category of their own.
- *Services.* Plug-ins can also define new functionality as a service that can be accessed from the code through the platform's centralized service manager.
- *Listeners.* Plug-ins can hook their functionality to events that occur in the system.
- *Tool windows.* Plug-ins can add customized UI components that can be docked into the IntelliJ's multiple document interface (MDI).
- *Custom settings.* Plug-ins can define their own settings and let the user configure those via a centralized control panel.

Importantly, especially from the perspective of this study, the plug-in descriptor file also defines platform-related dependencies of the plug-in: firstly, the version of the platform system with which the plug-in is compatible, and secondly, the other plug-ins on which it is dependent.

4.3 Plug-in dependencies in IntelliJ

In IntelliJ, declaring a plug-in dependent on another plug-in has a twofold consequence:

1. it makes the platform prevent installation of the dependent plug-in before the dependee plug-in is installed (unless the dependency is 'optional', see below);
2. it makes the classes of the dependee plug-in visible to the dependent plug-in.

The second point induces that the API of a plug-in can be simply built of public Java (or other JVM language) classes whose methods dependent plugins call. However, that is not a necessity: the API can also be based on other means of communication between the plug-ins, *e.g.*, the messaging framework provided by the platform. The plug-ins can, thus, interoperate also without dependency declaration.

The plug-in dependencies, declared in the descriptor file, can be defined either mandatory or optional. IntelliJ platform does not allow installing a plug-in, if its mandatory dependencies are not installed first. By contrast, optional dependencies are allowed to be missing, in which case their functionality cannot, of course, be used.

For example, a dependency to the Scala plug-in (a plug-in that equips IntelliJ with support for Scala programming language) could be declared by including following lines in the descriptor file:

```
<idea-plugin>
  <depends>org.intellij.scala</depends>
</idea-plugin>
```

Here, **org.intellij.scala** is the identifier of the Scala plug-in.

If the dependency was intended to be optional, the middle line above should be replaced with:

```
<depends optional="true"
         config-file="com.example.plugin-withScala.xml">
  org.intellij.scala
</depends>
```

In addition to declaring the dependency optional, the listing above defines another plug-in descriptor file **com.example.plugin-withScala.xml**[5] that is enabled if and only if the dependee plug-in is present. In that file, the plug-in can connect itself to extension points that are defined in the dependee plug-in.

Regardless of whether the dependency is optional or not, to be able to build the plug-in, the dependency must also be defined in the build script of the plug-in project. If the project uses Gradle build system, and the build script file (**build.gradle**) is written in Groovy, the definition for the above-mentioned Scala plug-in dependency could be:

```
intellij {
    plugins 'org.intellij.scala:2020.3.16'
}
```

For plug-ins available in JetBrains Marketplace, this is all the information that is needed by the build system to locate them.

[5]Here we assume **com.example.plugin** is the identifier of the dependent plug-in. Giving special descriptor files names according to this convention, where the plug-in identifier is followed with a dash and a locally unique string, averts naming collisions when multiple plug-ins are loaded into one instance. See "Plugin Dependencies" section in [17].

As can be seen, the listing above defines that the build system uses version **2020.3.16**[6] of the Scala plug-in. The version number must be defined for all plug-ins except for those that come bundled with the IntelliJ product in question. It is important, however, to note that this version information of the dependee plug-in, is only used to check compile time integrity between the classes of the plug-ins. The version numbers of the dependee plug-ins are *not* distributed to the compiled product of the dependent plug-in, and thus, are not available in run time.

When a user tries to install a plug-in, the platform checks that all its dependee plug-ins are installed, and if some mandatory dependencies are missing, requires user to install those before enabling the plug-in.

4.4 Discussion

4.4.1 Versioning and dependencies

As we saw, where plug-in dependencies are declared in the descriptor file, no version information is given. Thus, the platform system does not know with which version(s) of its dependee, a plug-in is compatible.

We can basically divide the versions $B_1, B_2, B_3, ...$ of a dependee plug-in into five categories in regard of a certain version A of the dependent plug-in:

1. Versions $B_1, ..., B_{l-1}$ that do not yet have all the functionality required by A, and are thus incompatible.
2. Versions $B_l, ..., B_{m-1}$ that has all the functionality required by A and are thus compatible with it.
3. The version B_m with which A was compiled and tested and which, thus, is compatible with A.
4. Versions $B_{m+1}, ..., B_u$ that do not introduce such a breaking change to B_m that would make them incompatible with A.
5. Versions $B_{u+1}, ...$, version B_{u+1} being the first with such a breaking change that makes it (and further versions based on it) incompatible with A.

[6]The Scala plug-in uses version numbering where IntelliJ Platform with which version it is compatible (**2020.3**) is followed by a dot (**.**) and a running number (**16**).

Of course, this categorization is a harsh generalization for theoretical purposes because it may happen, for example, that breaking changes are reverted in further versions, but we do not have to get stuck in such special cases.

What we can learn from this model, is that for a version A of a dependent plug-in there are versions $B_l, ..., B_m, ..., B_u$ that are compatible with it (categories 2, 3, and 4) while the versions in categories 1 and 5 are not.

When we consider that IntelliJ platform always suggests updating a plug-in to its newest version (compatible with the platform system – but not necessarily with the other plug-ins!), it is apparent that category 1 versions ("too early") are not as problematic as category 5 ("too late"). With category 1 versions, the user can simply solve incompatibility issues by updating the dependee plug-in to a newer version of it.

When it comes to category 5 versions, the user is in front of a much greater challenge when aiming to get a compatible version of the dependee plug-in. IntelliJ does not provide a straightforward way to retreat to an earlier version of a plug-in. All the earlier versions of plug-ins distributed via JetBrains Marketplace are, though, available in the web page but they must be manually downloaded and installed by the user. The situation is especially difficult to solve if the user has no information of which versions are known to be compatible.

It can be certainly argued, that IntelliJ Platform overlooks the issue of inter-plug-in compatibility when it does not include any version information in plug-in dependency declarations. To compensate this shortcoming of the platform, plug-in developers can take some extra measures:

- The dependent plug-in can inform users about versions of its dependee plug-ins that are known to be compatible.
- Plug-ins should avoid, whenever possible, breaking changes in their APIs. The breaking changes should be well documented. Semantic Versioning can be used to make it easily noticeable, which versions contain breaking changes.

4.4.2 When to declare dependencies optional?

The possibility to define certain plug-in dependencies optional is an interesting design choice by the designers of IntelliJ platform. In particular, it can be thought to be useful for plug-ins that have some minor feature dependent on another plug-in while the rest of the functionality has no dependencies. The benefit of making the dependency optional, in that case, is that the users of the dependent plug-in can still use the rest of its functionality even if they had not the dependee plug-in[7], whereas from the viewpoint of the developer of the plug-in, the optionality of a dependency does not bring any real relief: the dependency has to still be maintained like all other dependencies.

Plug-in developers could, in situations where two related plug-ins are developed simultaneously, be tempted to design two-way optional dependencies. While two-way mandatory dependencies make very little sense (as plug-ins could never be installed separately, they could as well be just one plug-in), with optional dependencies, we can see some rational in it: each plug-in would work alone but when both of them are present in the same system instance, they can interact in such manner that both of them see each other's classes. (Similar reasoning would apply to a setup where dependency is mandatory to one direction but optional to the other.)

A developer could end up considering this kind of design, if they are uncertain of which plug-in's responsibility it would be to provide the functionality that involves both of them (a question that not always has a definite answer). Two-way (optional) dependency is, however, never the right choice, as it couples together two pieces of code that are 'naturally' separated to the extent that they are even distributed separately. The developer in question has to make a design decision and choose one plug-in to be dependee and provide an API that the other (dependent) plug-in uses. Alternatively, they can put the common functionality in a third plug-in that is dependent on the both.

When programming a plug-in with optional dependencies, one must be aware that not all classes available in build-time are yet there in run-time. If classes of an optionally dependee plug-in are called when the said plug-in is not installed, a run-time error occurs. Code that employs the optional dependency is, thus, recommendable to put behind a facade[8] that ensures the dependee plug-in is installed before delegating the calls to the dependent code. Better yet, if there are no direct calls between the "problematic" code and the rest of the plug-in but both of these subsystems are separately called by the framework.

Because of these complexities, developers should not see dependence optionality as a 'golden hammer' that applies to all situations. I would advise primarily defining dependencies mandatory, and only considering making them optional, if there is an actual need for it, and if its supposed benefits surpass the extra complications it brings forth. If a plug-in developer feels that some of their plugin's dependencies should be optional, it is advisable

[7]In addition to the fact that not all plug-ins are freely available (*i.e.* they cost money), there are other reasons why a user could prefer not to install plug-ins they do not need, such as to avoid their IDE becoming too bloated of unnecessary functionality.

[8]For *facade* design patter, see [12, ch. 4].

to reconsider if the part that relates to that dependency is so disconnected from the other functionality, it should be set in a totally separate plug-in.

4.5 Comparing IntelliJ to other system

When comparing IntelliJ's plug-in framework to other similar systems, the most natural "baseline" is OSGi [8], a well-established Java specification that specifies a way to build modularly developed and distributed Java frameworks, such as plug-in systems. Considering one of the greatest competitors of IntelliJ, namely Eclipse IDE, has its plug-in framework based on OSGi, one could have expected that IntelliJ had taken the same route. Yet, that is not the case: IntelliJ is not built on OGSi specification. The reasons behind this design decision are unknown to the author; it is not ruled out that the developers of IntelliJ deliberately wanted to avoid the specification so closely associated to its competitor.

There are certain resemblance between OSGi specification and IntelliJ's plug-in framework: for example, both of them provide plug-ins (*bundles* in OSGi) with a class loader of its own to avoid dependency clashes, and they both have similar support for services that plug-ins can register. However, it would require a review too detailed for the purposes of this thesis to tell whether or not the resemblance between IntelliJ and OSGi specification is strong enough that the results and discussion above concerning IntelliJ could be considered applicable to plug-in systems built on OSGi.

When it comes to plug-in systems even more different from IntelliJ, such as those that are not written in Java or other object-oriented languages, or those that are not run in a single machine with a shared memory but as a distributed system, it can be expected that concerns related to plug-in interaction in them appear different. While the basic principles laid out in chapter 3 can be seen common to most plug-in systems, the results we found in this chapter, should be applied to systems other than IntelliJ only with caution.

Similarly, the next chapter lies firmly within the realm of IntelliJ. Thus, before applying its content to other plug-in systems, a reader should make sure the system in question is similar enough to IntelliJ. However, the purposes of the next chapter – as a case study – is not to find a universal pattern for solving the problem but to gain practical insight of the issue and, thus, widen our understanding of the problem and its solutions in a more abstract level. The most important ones of theses findings – by no means restricted to IntelliJ only – are listed in the end of the chapter.

Chapter 5

Case Study: A+ Courses, an IntelliJ IDEA plug-in

In this chapter I introduce and study an IntelliJ IDEA plug-in developed by us, a team of software developers in Aalto University Department of Computer Science. I analyze the plug-ins interaction with another plug-in – namely, IntelliJ Scala plug-in – and review related design decisions against the design directives laid out in the previous chapters.

5.1 About A+ Courses plug-in

A+ Courses is a plug-in for IntelliJ IDEA, that co-operates with *A+ Learning Management System* (see [9, 18]) to provide students with an enhanced learning experience in programming courses. Its development started in the beginning of 2020, and it was first in production use in an introductory programming course at Aalto University in the autumn of that year.

The main features of the plug-in are the following:

- Ability to fetch template code and libraries from a web server, and get it set up in the IDE with one click.
- Equipping Scala REPL (read-evaluate-print-loop, *i.e.*, console) with learning-oriented and learning-supporting features, such as, specialized instructions in the console window, and automatically-imported packages.
- Ability to submit course assignments to A+ server, and getting feedback and information of previous submissions.

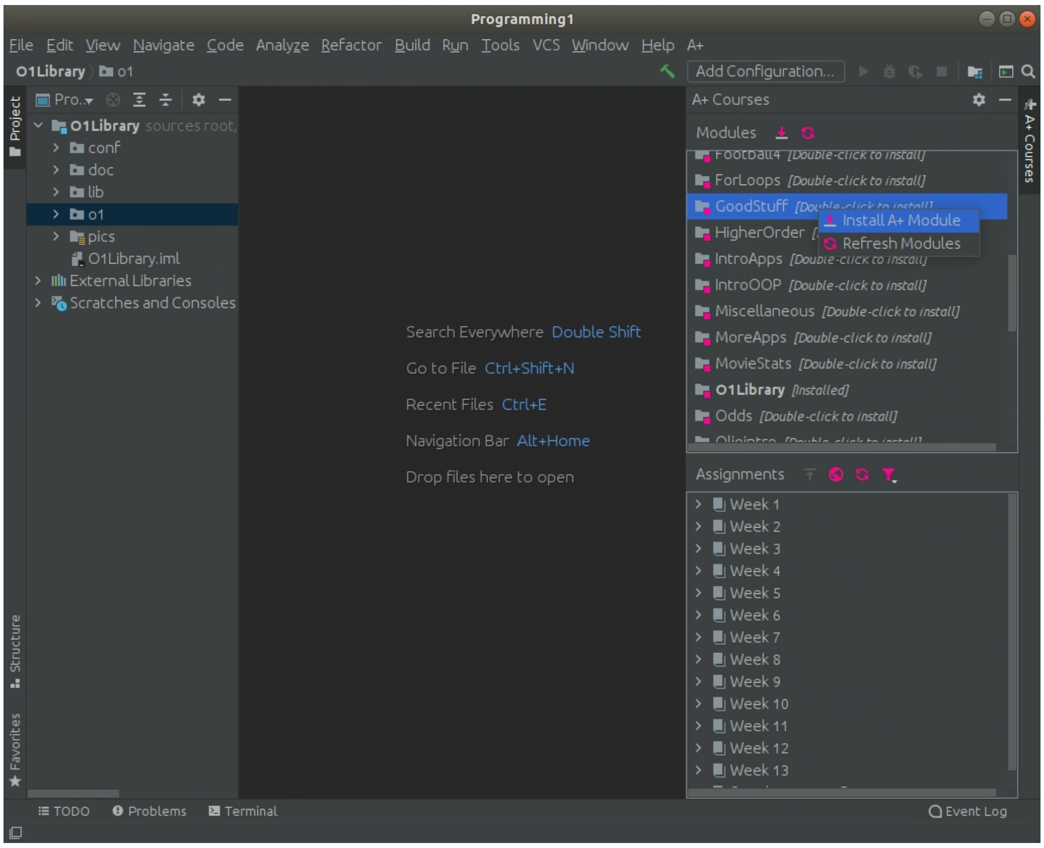
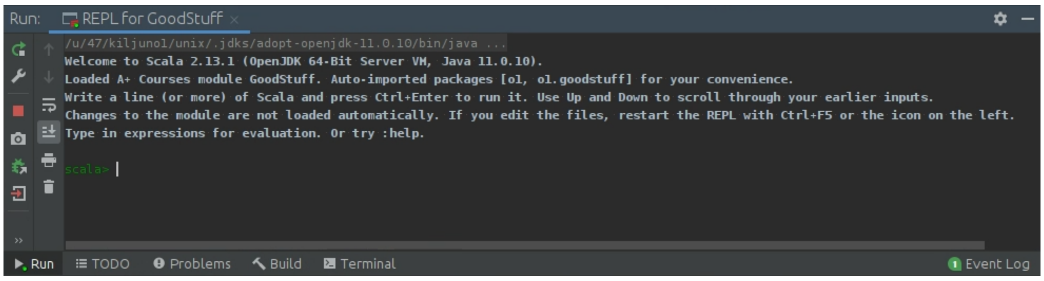


Figure 5.1: Screenshot
A+ Courses plug-in extends IntelliJ IDEA with a toolwindow that allows a student to download template code and submit their solutions to assignments.



```
/u/47/kiljunol/unix/.jdks/adopt-openjdk-11.0.10/bin/java ...
Welcome to Scala 2.13.1 (OpenJDK 64-Bit Server VM, Java 11.0.10).
Loaded A+ Courses module GoodStuff. Auto-imported packages [o1, o1.goodstuff] for your convenience.
Write a line (or more) of Scala and press Ctrl+Enter to run it. Use Up and Down to scroll through your earlier inputs.
Changes to the module are not loaded automatically. If you edit the files, restart the REPL with Ctrl+F5 or the icon on the left.
Type in expressions for evaluation. Or try :help.

scala>
```

Figure 5.2: Screenshot
A+ Courses plug-in also replaces the default Scala REPL with a console providing features suitable for programming education. Those include automatic package imports and instructive messages.

5.2 External dependencies

As an IntelliJ plug-in, A+ Courses depends on IntelliJ framework. In addition, A+ Courses depends on several software libraries: *Apache Commons IO*[1], *Zip4j*[2], and *JSON in Java*[3]. These libraries are bundled with the binaries of the plug-in, when the plug-in is distributed to users. As IntelliJ uses a separate class loader for each plug-in [17], these libraries do not cause any conflicts with the other plug-ins possibly present in a user's instance of IntelliJ.

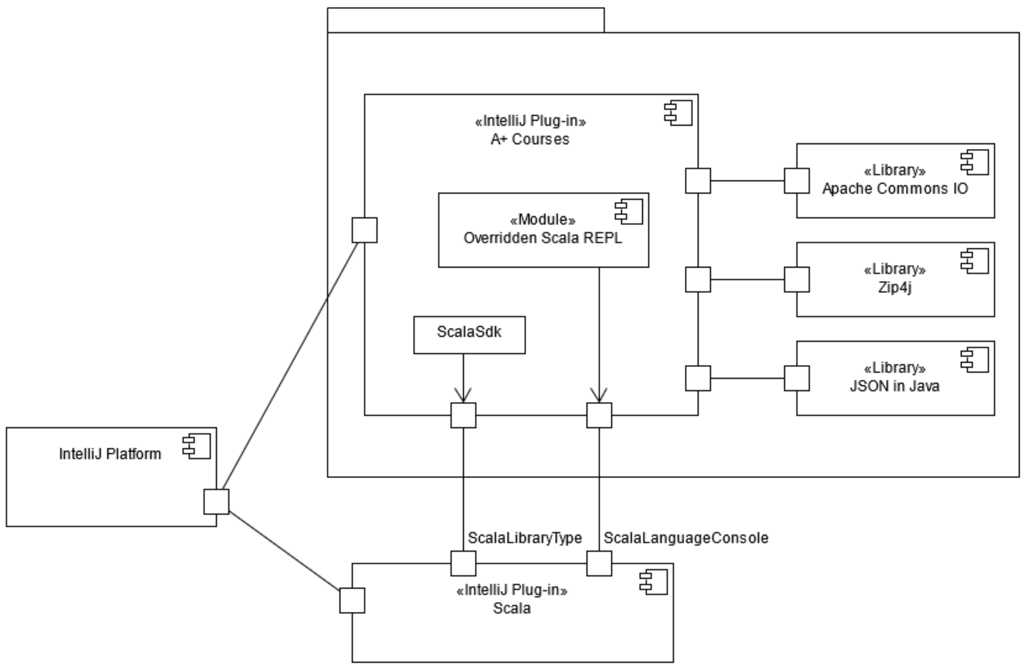


Figure 5.3: UML composite structure diagram
In the diagram, components in the same package as **A+ Courses** are loaded with the same class loader. We see that *A+ Courses* plug-in's dependency of the **Scala** plug-in is twofold: on the other hand, classes that implement the console replacement, depend on **ScalaLanguageConsole** (and related classes). On the other hand, **ScalaSdk** class depends on classes representing libraries of "Scala" type.

On top of the libraries listed above, A+ Courses depends on another IntelliJ plug-in, the Scala plug-in. In contrary to the libraries, the Scala

[1]`https://commons.apache.org/proper/commons-io/`
[2]`http://www.lingala.net/zip4j.html`
[3]`https://github.com/stleary/JSON-java`

plug-in is not bundled with A+ Courses, but, as it is the case with plug-in dependencies in general, it is installed separately by the user. This makes the dependency susceptible to the issues discussed in this thesis. Due to its relevancy here, we focus on A+ Courses's dependency on the Scala plug-in in this chapter. In the next subsection, the Scala plug-in is described in more detail to give a context to the sections following.

5.2.1 Scala plug-in

IntelliJ IDEA does not itself support other programming languages than Java. However, the vendor of the platform system, that is, JetBrains, develops plug-ins that are free to use and extend IntelliJ IDEA with support for other JVM languages. Such a plug-in is the *Scala* plug-in[4], which makes IntelliJ IDEA support development in Scala language.

While the Scala plug-in has many features including, *e.g.*, syntax highlighting of Scala code, the two features in which we are interested are

- configuring Scala projects to use certain Scala SDKs, and
- Scala REPL where a user can type Scala commands and see them evaluated immediately.

We discuss these further in the next section.

The Scala plug-in is actively developed, and new versions of it are published several times a month. The development process is driven and governed by JetBrains but as an open-source project, the Scala plug-in also welcomes external contributors – including developers of other plug-ins.

5.3 Interaction with Scala plug-in

The interaction between A+ Courses and Scala plug-in occurs in two aspects:

- A+ Courses automatically installs certain downloaded libraries as Scala SDKs. A+ Courses must use Scala plug-in's classes to make the library properly recognized as a Scala SDK.
- A+ Courses extends Scala plug-in's REPL and replaces the original REPL with the extended one.

The two aspects of interaction are separate and, thus, have been developed independently of each other.

[4]`https://plugins.jetbrains.com/plugin/1347-scala`

5.3.1 Automated installment of Scala SDKs

IntelliJ Platform allows libraries to be installed into a project and linked to its modules. The Scala plug-in extends this feature allowing a library to be treated as a Scala SDK, a special kind of library, that provides the standard library of Scala language as well as code required to run Scala compiler. As A+ Courses implements automated installation of libraries that are referred in the modules, in case of Scala SDKs, it cooperates with the Scala plug-in to get these libraries properly configured.

The library installation in A+ Courses is implemented as per the template method pattern[5]. The superclass, `IntelliJLibrary`, is unaware of the “kind” of the library (using IntelliJ’s terminology), and provides only a skeleton for the installation procedure in its `loadInternal` method. The subclasses, each being responsible for certain kind of libraries, are called by the template method to get kind-specific functionality. The methods called by the `loadInternal` and implemented by the subclasses are

- `getLibraryKind` that specifies the ”kind” of a library by returning a `PersistentLibraryKind` object representing it, and
- `initializeLibraryProperties` that equips a `LibraryProperties` object with kind-specific information.

(`PersistentLibraryKind` and `LibraryProperties` are classes of IntelliJ Platform.)

Now, class `ScalaSdk`, a subclass of `IntelliJLibrary` being responsible for “Scala” kind of libraries – that is, of course, Scala SDKs –, implements those two methods using classes of the Scala plug-in. This way, all code related to Scala SDK installation that is dependent on the Scala plug-in is encapsulated inside a single class. Because of the template method pattern, that class can reuse its superclass’s code to the highest possible extent, and only contain parts that are specific to installations of Scala SDKs.

5.3.2 Overriding Scala REPL

When developing A+ Courses plug-in we wanted to make certain modifications to the REPL provided by the Scala plug-in. To that end, we inherited `ScalaLanguageConsole` class and overrode some of its methods to match our needs. However, the Scala plug-in does not declare a proper extension point for its REPL, so there was no straightforward way to make the Scala plug-in use our extended version of the REPL class.

[5]For *Template Method* design patter, see [12, ch. 5].

To overcome the issue, we ended up replacing the action of the Scala plug-in that launches the REPL with an action that launches our modified REPL instead. Here, we used a poorly documented feature of IntelliJ that allows plug-ins to replace actions defined by the platform or, as in this case, other plug-ins.[6]

The overriding action class is itself a subclass of the action class it overrides. The subclass changes its behavior to instantiate the REPL class of our own, instead of the original one.

5.4 Discussion

5.4.1 The issues related to overriding Scala REPL

As described above, in the lack of proper extension point for the Scala REPL, we had to rely on an alternative way to get our access into the Scala REPL. As it seems unlikely that the developers of the Scala plug-in have considered a possibility of this kind of reliance, it is highly possible they do not care of backwards-compatibility of this part of their plug-in in their future releases (it does not belong to the intended API of the Scala plug-in). Thus, it sets a risk of a future release of the Scala plug-in – even a minor one – breaking the compatibility with A+ Courses plug-in without a notice. The users who had installed the new, non-compatible version of the Scala plug-in would then no longer be able to install A+ Courses without downgrading their version of Scala plug-in.

To avoid this risk, we should start the collaboration with the Scala plug-in developers to make their plug-in declare a proper extension point for the REPL, and, once supported, make A+ Courses plug-in use that. The initial steps in communication with the Scala plug-in developers about the issue have already been taken.

[6]It is unclear whether or not third-party plug-ins are expected to use this feature. In any case, action replacement is achieved by registering the action with the same ID as the replaceable action in **plugin.xml**, and enclosing it with `overrides="true"` attribute. To see, how this is handled in the platform code, see `https://github.com/JetBrains/intellij-community/blob/212.443/platform/platform-impl/src/com/intellij/openapi/actionSystem/impl/ActionManagerImpl.java#L725-L750`.

5.4.2 Could REPL-related functionality make a separate plug-in?

Considering both the above-discussed maintenance risk caused by the REPL functionality of A+ Courses as well as its conceptual and functional detachment of the rest of the plug-in's functionality, it could be advisable to split A+ Courses into two separate plug-ins, one of which containing the REPL and the other the rest. In that case, even if the above-mentioned risk actualized and made the REPL plug-in incompatible, the use of the functionality of the other plug-in would not be affected.

This change would be also interesting because then the other plug-in would depend on the Scala plug-in only in relation to the installment of the Scala SDKs. As this feature is only used in cases where a user installs modules containing Scala code, one can think of many use cases, where the plug-in's functionality is totally independent of the Scala plug-in. Thus, for the convenience of the users with no intentions of using Scala language, the plug-in's dependency on the Scala plug-in could be defined optional. This way, users would have to install the Scala plug-in only if they needed to use it with Scala modules.

The current design of the plug-in is favorable for making the Scala plug-in dependency optional in the described case. As it was said in 5.3.1, the code using the Scala plug-in is encapsulated in a single class, `ScalaSDK`. That class is, in turn, referred only once, in a factory class where it is instantiated when an installation of Scala SDK is initiated. Thus, no class dependent on the Scala plug-in has to be loaded unless it is actually needed. If the dependency to the Scala plug-in was made optional, the factory class should, however, check that the Scala plug-in is installed, before calling the constructor of the `ScalaSDK` class. In case the Scala plug-in is not installed, the current operation should be cancelled and the user should be instructed to install the said plug-in before retrying.

5.4.3 Overall review of the implementation of the plug-in interaction

The way we implemented the interaction between A+ Courses plug-in and the Scala plug-in has proven functional in practice, as no problems related to it have arisen so far. As discussed above, the design of REPL overriding sets, however, some technical debt for the maintainers to solve.

Importantly, this technical debt as well as risks related to it have been understood and taken into account, and it is known how the problematic design can be fixed in cooperation with the developers of the Scala plug-in.

Meanwhile, if considered necessary, the REPL can be separated from the rest of the plug-in's functionality to loosen the coupling to the Scala plug-in. Due to the competent design decisions we made, the architecture of A+ Courses allows us to do that easily, as explained above.

5.4.4 What did we learn?

This case study presented us a situation where the developers of a dependent plug-in (*i.e.*, us) were forced to settle for suboptimal design due to the fact that the most robust design required of the dependee plug-in functionality that was not present at the time. It appears, that when implementing plug-in interoperation, one has to take into account realities that are involved in the distributed nature of the development ecosystem. Such insights that this case study brought up include:

- The plug-in developers cannot control the whole system they develop, which means they are constrained by what is or is not supported by the plug-ins of which they are dependent.
- Designing plug-in interoperation may require trading off good design to get the work done; good overall architectural choices, however, help keeping problematic parts contained and design-related risks minimal.
- It cannot be trusted that the release environment of the plug-in stays the same forever: as the dependent plug-ins and the platform system keep changing, the plug-in maintainers must stay alert and be ready to react to changes when necessary.

Chapter 6

Conclusions and further directions

As it was discussed above, such software systems that can be extended by externally developed components – plug-ins – have become more and more popular in the early decades of the twenty-first century. Although it is has been well-recognized by software architects that in such a system, dependencies between plug-ins cause challenges in software design, professional literature seems to have very little guidance on the issue – other than mere "avoid whenever possible".

There are cases where it is not feasible to keep plug-ins independent of each other. To equip readers of this thesis with more helpful and concrete guidelines for such situations, I have analyzed design principles that could simplify the complexity caused by dependencies between interacting plug-ins.

Unsurprisingly, there seems to be no silver bullet that would make plug-in dependencies trouble-free once and for all. However, with careful design, software developers can avoid many pitfalls inhering in plug-in interdependencies and substantially relieve the burden of the software maintenance. Solutions I have suggested in this thesis are not new tricks; instead, they are well-known general design principles and patterns which I have applied to plug-in dependencies and argued for their usefulness in that aim.

In this thesis, I have covered the issue from three perspectives: those of the developers of a plug-in system, a plug-in that depends on another plug-in, and a plug-in that allows other plug-ins to be dependent on it. This division reflects the fact that implementing plug-in interoperation is a joint effort of these three software developers or developer teams, all of whom (in a general case) work in different organizations without common coordination between them. What is more, it is not untypical in this 'collaboration' that the only way knowledge is transferred from a party to another is via (often deficient)

API documentations.

Considering the tightly-connected nature of interoperating plug-ins – which this thesis highlights – it is reasonable to question whether their development calls for more interactive and collaborative development process than software components in general. Examining how communication happens and how knowledge gets shared between the developers of interoperating plug-ins and plug-in systems could give us deeper understanding of what kind of collaboration process models could be practiced to design and implement plug-in interoperation of a high-level quality. This would serve as an interesting topic of future research.

For those interested in A+ Courses plug-in, I recommend Nikolai Denissov's – my friend and colleague's – Master's thesis [6] as a further reading. He was one the three original developers of A+ Courses plug-in (among Nikolas Drosdek and me), and his thesis discusses the development process in a more general level than here. Although Denissov and Drosdek will no longer be involved in the project, the development of the plug-in continues by me and three new developers – Jaakko Närhi, Paweł Strozanski, and Styliani Tsovou – who joined the team in January 2021. New scientific research related to A+ Courses plug-in can be expected in the future, though it is likely that its topic differs remarkably from this thesis's.

Bibliography

[1] Apel, S., Batory, D., Kästner, C., and Gunter, S. Software product lines. In *Feature-Oriented Software Product Lines: Concepts and Implementation*, S. Apel, D. Batory, C. Kästner, and S. Gunter, Eds. Springer, Berlin, Germany, 2013, pp. 3–15.

[2] Bass, L., Clements, P., and Kazman, R. *Software Architecture in Practice*, 3rd ed. Addison-Wesley, Upper Saddle River, NJ, 2012.

[3] Bosch, J. Software product line engineering. In *Systems and Software Variability Management*, R. Capilla, J. Bosch, and K.-C. Kang, Eds. Springer, Berlin, Germany, 2013, ch. 1, pp. 3–24.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture*, vol. 1. Wiley, Chichester, West Sussex, UK, 1996.

[5] Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., and Stafford, J. *Documenting Software Architectures: Views and Beyond*, 2nd ed. Addison-Wesley, Upper Saddle River, NJ, 2010.

[6] Denissov, N. Creating an educational plugin to support online programming learning a case of intellij idea plugin for a+ learning management system. Master's thesis, Aalto University, 2021.

[7] Eclipse Foundation. Eclipse documentation, Eclipse IDE 2020-09. `https://help.eclipse.org/2020-09/`.

[8] Eclipse Foundation. OSGi core release 8 specification. `https://docs.osgi.org/specification/osgi.core/8.0.0/`, 2020.

[9] EDIT in the Department of Computer Science at Aalto University. A+ LMS: The extendable learning management system. `https://apluslms.github.io/`.

[10] foobar2000 web site. `http://www.foobar2000.org/`. Accessed: 2020-10-06.

[11] Free Software Foundation. GNU lesser general public license, version 2.1. `https://www.gnu.org/licenses/old-licenses/lgpl-2.1.html`, 1991.

[12] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Upper Saddle River, NJ, 1994.

[13] GitHub. Atom flight manual, v1.51.0. `https://flight-manual.atom.io/`.

[14] Glickstein, B. *Writing GNU Emacs Extensions.* O'Reilly, Sebastopol, CA, 1997.

[15] Ingeno, J. *Software Architect's Handbook.* Packt, Birmingham, West Midlands, UK, 2018.

[16] JetBrains. Marketplace. `https://plugins.jetbrains.com/`. Accessed on 2021-01-21.

[17] JetBrains. IntelliJ Platform SDK. `https://plugins.jetbrains.com/docs/intellij/`, 2021.

[18] Karavirta, V., Ihantola, P., and Koskinen, T. Service-oriented approach to improve interoperability of e-learning systems. In *2013 IEEE 13th International Conference on Advanced Learning Technologies* (Los Alamitos, CA, 2013), IEEE, pp. 341–345.

[19] Klatt, B., and Krogmann, K. Software extension mechanisms. In *Proceedings of the Thirteenth International Workshop on Component-Oriented Programming* (Karlsruhe, Germany, 2008), pp. 11–18.

[20] Microsoft. Office add-ins. `https://docs.microsoft.com/en-us/office/dev/add-ins/`. Accessed: 2020-10-07.

[21] Mozilla. MDN web docs, Mozilla add-ons. `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons`. Accessed: 2020-10-07.

[22] Osipov, R. *Mastering Vim.* Packt, Birmingham, West Midlands, UK, 2018.

[23] Pohl, K., and Metzger, A. Software product lines. In *The Essence of Software Engineering*, V. Gruhn and R. Striemer, Eds. Springer, Cham, Switzerland, 2018, pp. 185–201.

[24] Preston-Werner, T. Semantic Versioning 2.0.0. `https://semver.org/spec/v2.0.0.html`, 2013.

[25] Richards, M. *Software Architecture Patterns.* O'Reilly, Sebastopol, CA, 2015.

[26] Richards, M. *Fundamentals of Software Architecture.* O'Reilly, Sebastopol, CA, 2020.

[27] Rytter, M., and Jørgensen, B. N. Independently extensible contexts. In *Software Architecture, 4th European Conference Proceedings, ECSA 2010* (Copenhagen, Denmark, Aug 2010), M. A. Babar and I. Gorton, Eds., no. 6285 in Lecture Notes in Computer Science, Springer, Berlin, Germany, pp. 327—334.

[28] Shaw, M. What makes good research in software engineering? *International Journal on Software Tools for Technology Transfer 4*, 1 (2002), 1–7.

[29] Sommerville, I. *Software Engineering, Global Edition.* Pearson, Harlow, Essex, UK, 2016.

[30] Stallings, W. *Operating Systems*, 8th ed. Pearson, Harlow, Essex, UK, 2014.

[31] Szyperski, C. Independently extensible systems: Software engineering potential and challenges. In *ACSC'96: Nineteenth Australasian Computer Science Conference Proceedings* (Melbourne, Australia, Jan–Feb 1996), K. Ramamohanarao, Ed., vol. 18 of *Australian Computer Science Communications*, pp. 203–212.

[32] Tarkoma, S. *Mobile Middleware: Architecture, Patterns and Practice.* Wiley, Chichester, West Sussex, UK, 2009.

[33] Taylor, R., Medvidović, N., and Dashofy, E. *Software Architecture: Foundations, Theory, and Practice.* Wiley, Hoboken, NJ, 2009.